Entering / cleaning data 1

Getting data into R

Basics of getting data into R

Basic approach:

- Download data to your computer
- Save the data in your R Project directory for the project you're using it for or in a subdirectory within that directory ("data" is a good name for this subdirectory)
- Read data into R (functions in `readr`: `read_csv`, `read_table`, `read_delim`, `read_fwf`, etc.)
- Check to make sure the data came in correctly (`dim`, `ncol`, `nrow`, `head`, `tail`, `str`, `colnames`)

Reading data into R

What kind of data can you get into R?

The sky is the limit...

- **Flat files**
- Files from other statistical packages (SAS, Excel, Stata, SPSS)
- Tables on webpages
- Data in a database (e.g., SQL)
- Data stored in XML and JSON
- Complex data formats (e.g., netCDF files from climate folks, MRI data stored in Analyze, NIfTI, and DICOM formats)
- Data through APIs (e.g., GoogleMaps, Twitter, many government agencies)
- Incredibly messy data using `scan` and `readLines`

Flat files

R can read in data from *a lot* of different formats. The only catch: you need to tell R how to do it.

To start, we'll look at **flat files**, which are plain text files (i.e., you can read them when you open them in a text editor, unlike a file in a binary format, like an Excel or Word file) with a two-dimenional structure (a row for each observation and a column for each variable).

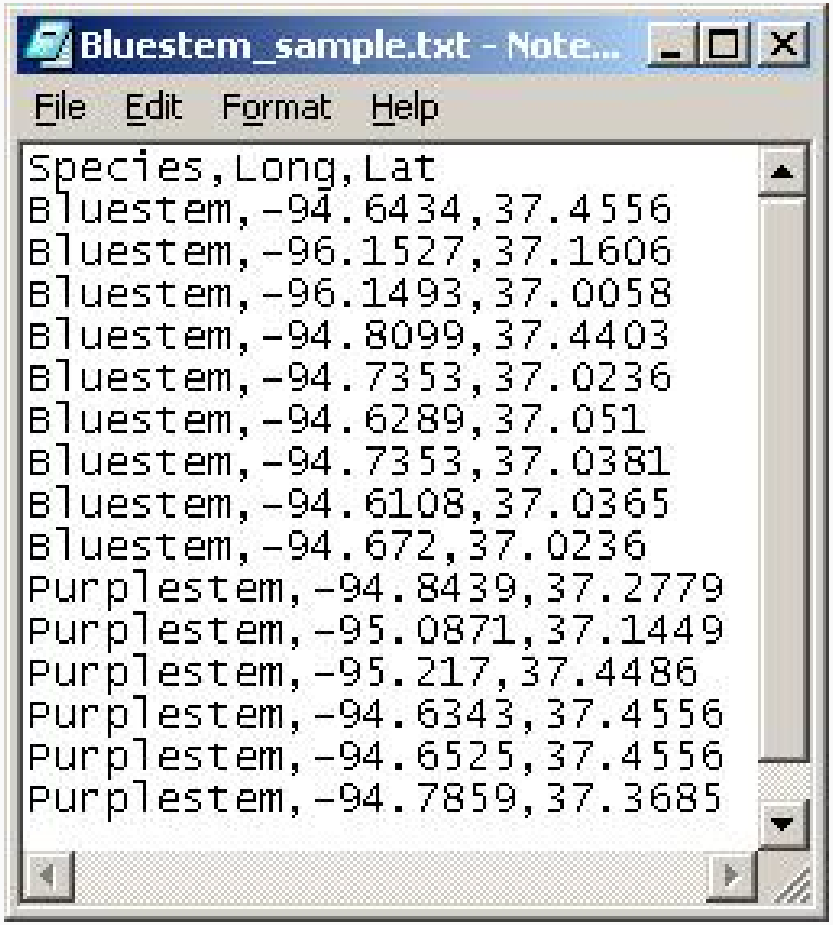
Types of flat files

There are two main types of flat files:

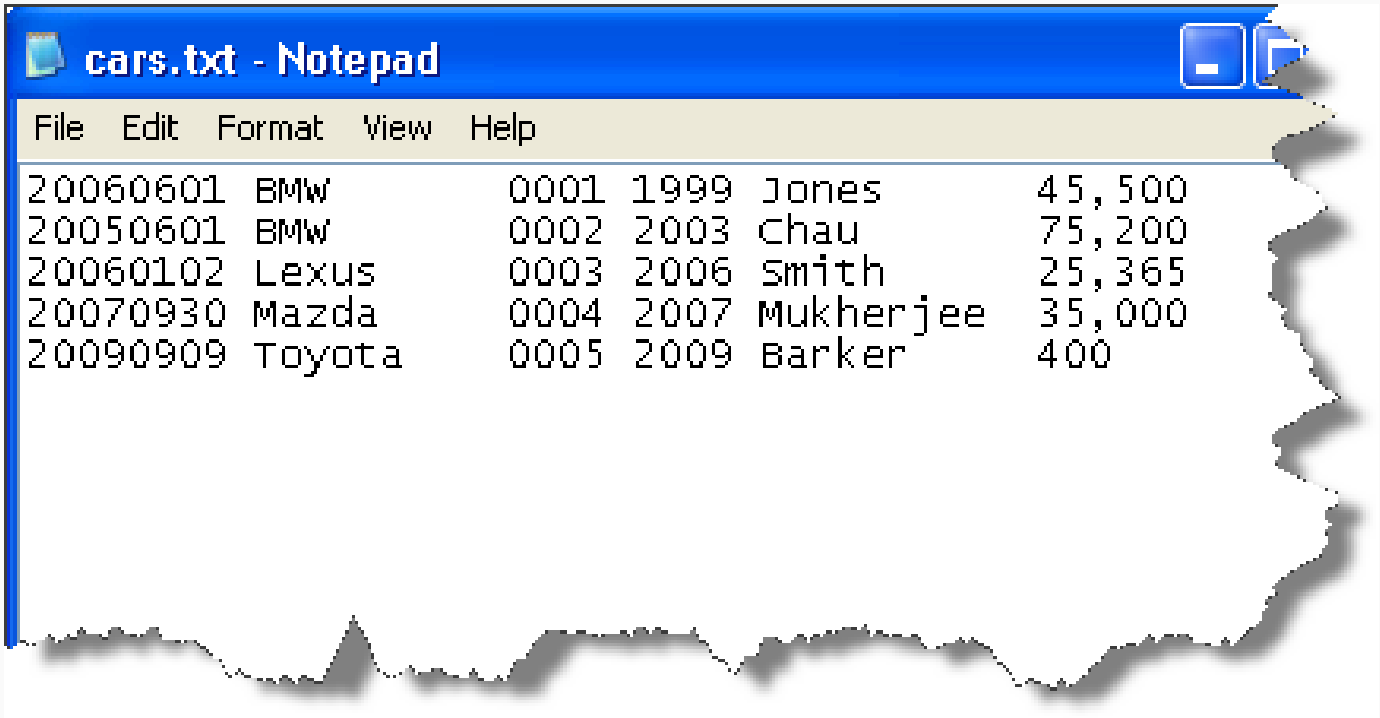
1. **Fixed width files**: Each column is a certain number of characters wide. (If you printed it out, you could draw vertical lines that separate the columns.)
2. **Delimited files**: In each row, a certain symbol (**delimiter**) separates the data into columns values for that observation.
    - ".csv": Comma-separated values
    - ".tab", ".tsv": Tab-separated values
    - Other possible delimiters: colon, semicolon, pipe ("|")

See if you can identify what types of files the following files are...

What type of file?



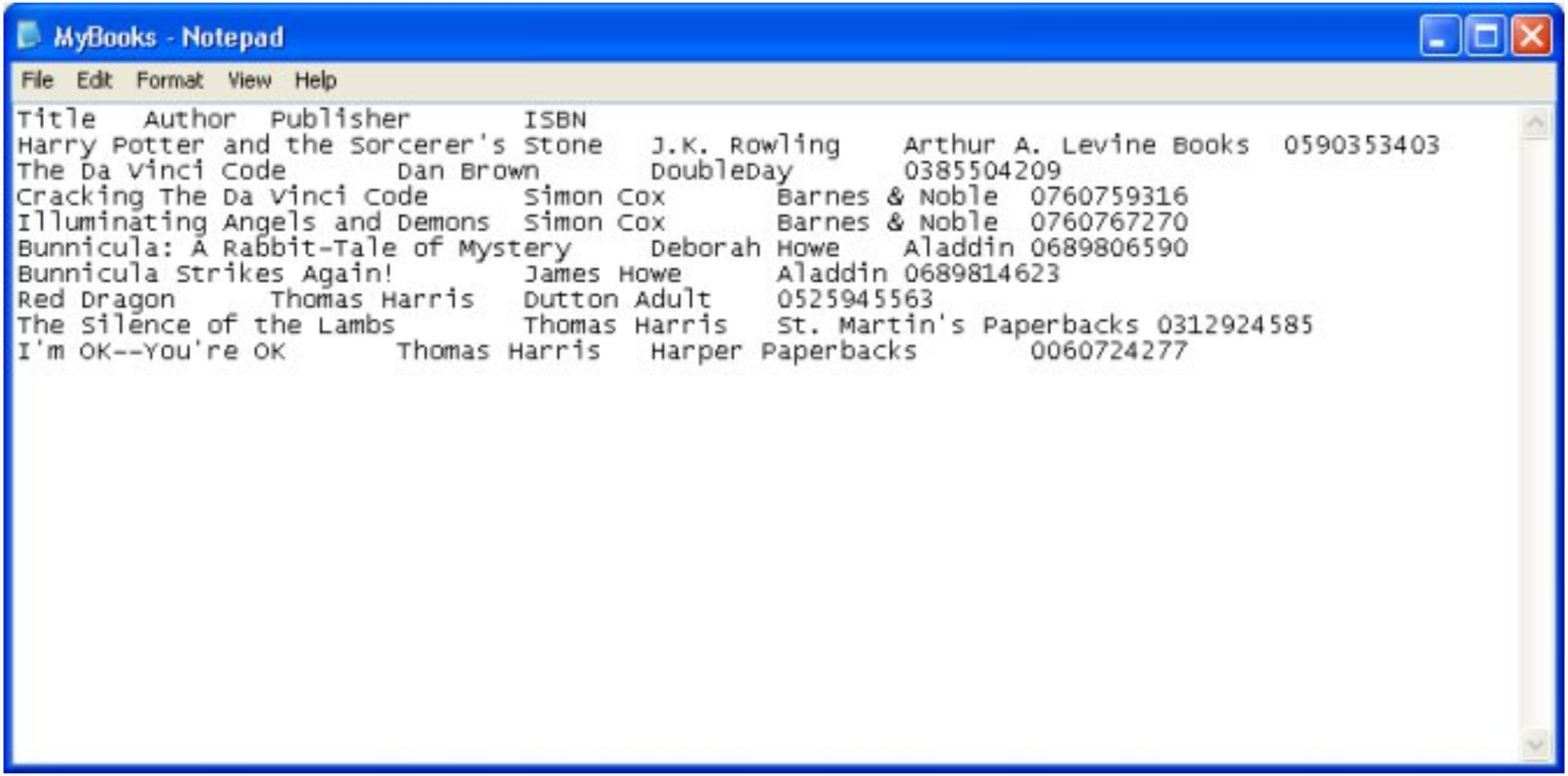
What type of file?



What type of file?

```
H|20110606|pizza.txt|
D|10|Chicken Pesto|20|23|30|5.5|7.4|9.9||
D|10|Meatball|10|53|60|6.5|8.4|10.9|
D|10|Fire Cracker|3|13|60|5.8|7.9|11.9|
D|10|Spinach|1|2|5|5.5|7.0|8.8|
D|10|BBQ Chicken|35|102|95|6.5|7.9|10.9|
D|10|Vegetarian|5|13|28|4.5|7.9|9.5|
D|10|Mexican|11|33|36|5.5|7.4|9.9|
D|10|The Monaco|22|53|7|5.5|7.5|8.9|
D|10|Chilli Prawn|5|5|6|5.5|7.4|9.9|
D|10|Chefs Special|8|18|40|5.8|7.8|9.8|
D|10|Marinara|3|17|41|5.5|7.4|9.0|
D|10|Supreme|50|52|58|5.5|7.4|9.2|
D|10|Margherita|9|19|87|5.0|7.0|8.0|
D|10|Napoli|60|85|66|5.2|7.2|9.2|
D|10|Caprice|31|32|38|5.5|7.4|9.3|
D|10|Ham and Pineapple|18|39|28|5.8|7.0|9.0|
T|16|
```

What type of file?



What type of file?

File Edit Format View Help

```
Title,Subtitle,Larger Work,Contributor #1,Contributor #2,Contributor #3,Contributor
#4,Genre,Publisher,Published Location,Date
Published,Instrumentation,Key,Location,Indiana Connection,Sheet Music
Consortium,Notes,Complete
"""A"" You're Adorable",The alphabet song,,Buddy Kaye,Sidney Lippman,Fred Wise,,Popular
standard,Laurel Music Corporation,"New York, NY",1948,Voice and
piano/guitar or ukulele,C Major,,None,Yes,Perry Como pictured on cover,
"Aba Daba Honey Moon, The",,"""Two Weeks with Love"" Motion Picture",Arthur Fields,Walte
Donovan,,,"Popular Standard, Movie Selection",Leo Feist Inc.,"New
York, NY",1942,Voice and Piano,C Minor,,None,Yes,,
Abi Bezunt,,"""Mamele"" Motion Picture",Abraham Ellstein,Molly Picon,,,"Popular Standard,
Movie Selection",Metro Music Co.,"New York, NY",1939,Voice and
Piano,E Minor,,None,No,Molly Picon pictured on cover,
Abdul the Bulbul Ameer,,,Bob Kaai,Jim Smock,,,Popular Standard,Calumet Music Co.
,"Chicago, IL",1935,"Voice, Piano, Hawaiian Guitar, Ukulele",G
Major,,None,Yes,Ben Pollack pictured on cover,
About A Quarter to Nine,,"""Go Into Your Dance"" Motion Picture",Harry Warren,Al
Dubin,,,"Popular Standard, Movie Selection",M. Witmark & Sons,"New York,
NY",1935,"Voice, Piano, Guitar, Ukelele",E Minor,,None,No,Al Jolson and Ruby Keeler
pictured on cover,
Absent,,,John. W. Metcalf,Catherine Young Glen,,,Popular Standard,Arthur P.
Schmidt,"Boston, MA",1899,Voice and Piano,G Major,,None,Yes,,
The Academy Two-Step,,,Barclay Walker,,,,Popular Standard,Carlin & Lennox,"Indianapolis,
IN",,Piano,F Major,,Composer,No,,
Ac-cent-tchu-ate the Positive,Mister In Between,"""Here Come the Waves"" Motion
Picture",Harold Arlen,Johnny Mercer,,,"Popular Standard, Movie
Selection",Edwin H. Morris & Co.,"New York, NY",1944,"Voice, Piano, Guitar",F
Major,,None,Yes,Bing Crosby and Betty Hutton pictured on cover,
Across the Alley From the Alamo,,,Joe Greene,,,,Popular Standard,Leslie Music
```

What type of file?

```
1000233   Miralda              John
1000234   Faley                Nick
1000235   Baylog               Cathy
1000236   Gallardo             Mike
1000237   Christian            Daniel
1000238   Baufield             Daniel
1000239   Frazier              Robert
1000240   Garrido              Edward
1000241   Williams             Zachary
1000242   Morel                David
          Padilla              Damian
1000244   Rosenberg            Wayne
1000245   Blanchard            Phong S
1000246   Wiggins              David
1000247   Miller               Jeffrey
1000248   Coon                 Terry
1000249   Chretien             Walter
1000250   Myers                Timothy

1000233   Miralda              John
1000234   Faley                Nick
1000235   Baylog               Cathy
```

Types of flat files

Flat files will often end in file extensions like ".txt", ".csv", ".fwf", and ".tsv".

To figure out the structure of a flat file, start by opening it in a text editor. RStudio can also be used as a text editor to open and explore flat files (right click on the file name and then choose "Open With" and "RStudio").

Reading in flat files

R can read any of the types of files we just looked at by using one of the functions from the `readr` package:

File type	General function
Delimited	`read_delim`
Fixed width	`read_fwf`

You will just need to be able to clearly tell R *how* to read the file in, including what type of flat file it is and what delimiter it uses.

Reading in flat files

For example, the file "AWOIS_Wrecks_KnownYear.tab" is a flat delimited file with tabs as delimiters containing the subset of the Office of the Coast Survey's Automated Wreck and Obstruction Information System (AWOIS) for which the year the vessel sank is known.

You can download this file by going to this link and using the "Raw" button in the top right hand corner (right click and select "Download Linked File").

Reading in flat files

If save this file in your working directory, to read it in and assign it the name `shipwrecks`, you can run:

```
library("readr")
shipwrecks <- read_delim("AWOIS_Wrecks_KnownYear.tab",
                         delim = "\t")
```

`readr` family of functions

Some of the interesting options with the `readr` family of functions are:

Option	Description
`skip`	How many lines of the start of the file should you skip?
`col_names`	What would you like to use as the column names?
`col_types`	What would you like to use as the column types?
`n_max`	How many rows do you want to read in?
`na`	How are missing values coded?

Reading in flat files

The "daily_show_guests.csv" file you worked with in the previous In-Course Exercise is a delimited flat file with commas as the delimiters. **It also has four lines of information about the data, before the actual data begins.**

```
1  ## Obtained from GitHub page of FiveThirtyEight under the
2  ## Creative Commons Attribution 4.0 International License
3  ## https://github.com/fivethirtyeight/data/tree/master/daily-show-guests
4  ##
5  YEAR,GoogleKnowlege_Occupation,Show,Group,Raw_Guest_List
6  1999,actor,1/11/99,Acting,Michael J. Fox
7  1999,Comedian,1/12/99,Comedy,Sandra Bernhard
8  1999,television actress,1/13/99,Acting,Tracey Ullman
```

Reading in flat files

You can handle this by using the `skip` option to tell R to skip the first four lines:

```
read_delim("daily_show_guests.csv", delim = ",", skip = 4)
```

```
## # A tibble: 2,693 x 5
##     YEAR GoogleKnowlege_~ Show  Group
##    <dbl> <chr>            <chr> <chr>
##  1  1999 actor            1/11~ Acti~
##  2  1999 Comedian         1/12~ Come~
##  3  1999 television actr~ 1/13~ Acti~
##  4  1999 film actress     1/14~ Acti~
##  5  1999 actor            1/18~ Acti~
##  6  1999 actor            1/19~ Acti~
##  7  1999 Singer-lyricist  1/20~ Musi~
##  8  1999 model            1/21~ Media
##  9  1999 actor            1/25~ Acti~
## 10  1999 stand-up comedi~ 1/26~ Come~
## # ... with 2,683 more rows, and 1 more variable:
## #   Raw_Guest_List <chr>
```

`readr` family of functions

Many members of the `readr` package that read delimited files are doing the same basic thing. The only difference is what defaults they have for the delimiter (`delim`).

Some key members of the `readr` family for delimited data:

Function	Delimiter
`read_csv`	comma
`read_csv2`	semi-colon
`read_table2`	whitespace
`read_tsv`	tab

`readr` family of functions

For any type of delimited flat files, you can also use the more general `read_delim` function to read in the file. However, you will have to specify yourself what the delimiter is (e.g., `delim = ","` for a comma-separated file).

For example, the following two calls do the same thing:

```
read_delim("daily_show_guests.csv", delim = ",", skip = 4)
read_csv("daily_show_guests.csv", skip = 4)
```

`readr` family of functions

The `readr` package also includes some functions for reading in fixed width files:

- `read_fwf`
- `read_table`

These allow you to specify field widths for each fixed width field, but they will also try to determine the field-widths automatically.

Reading data from other files types

You can also read data in from a variety of other file formats, including:

File type	Function	Package
Excel	`read_excel`	`readxl`
SAS	`read_sas`	`haven`
SPSS	`read_spss`	`haven`
Stata	`read_stata`	`haven`

Reading in flat files

Once you read the data in, you should investigate it to make sure it looks like it was read in without bugs.

For example, you may want to look at a subset of the data using the tools you learned last week:

```
# Check out a subset of the data
library("dplyr")
slice(.data = select(.data = shipwrecks, c(2, 4, 5, 9)), 1:4)
```

```
## # A tibble: 4 x 4
##   VESSLTERMS    LATDEC LONDEC YEARSUNK
##   <chr>          <dbl>  <dbl>    <dbl>
## 1 SUBCHASER 187   37.3  -75.5     1918
## 2 BIRCH LAKE      37.3  -75.6     1943
## 3 PACIFIC         37.3  -75.6     1925
## 4 UNKNOWN         37.3  -75.7     1916
```

Reading in flat files

You can also use the `head` function to look at the first few rows:

```
head(x = shipwrecks, n = 3)
```

```
## # A tibble: 3 x 10
##   RECRD VESSLTERMS FEATURE_TYPE LATDEC LONDEC
##   <dbl> <chr>      <chr>          <dbl>  <dbl>
## 1   959 SUBCHASER~ Wreck - Sub~   37.3  -75.5
## 2   960 BIRCH LAKE Wreck - Sub~   37.3  -75.6
## 3   968 PACIFIC    Wreck - Sub~   37.3  -75.6
## # ... with 5 more variables: GP_QUALITY <chr>,
## #   DEPTH <dbl>, SOUNDING_TYPE <chr>,
## #   YEARSUNK <dbl>, HISTORY <chr>
```

Reading in flat files

Or you can also use the `tail` function to look at the last few rows:

```
tail(x = shipwrecks, n = 3)
```

```
## # A tibble: 3 x 10
##   RECRD VESSLTERMS FEATURE_TYPE LATDEC LONDEC
##   <dbl> <chr>      <chr>          <dbl>  <dbl>
## 1  2764 EVENING S~ Not Charted     38.9  -74.6
## 2  2600 ACARA      Wreck - Sub~    40.6  -73.6
## 3  7722 HOWARD     Wreck - Sub~    40.6  -73.5
## # ... with 5 more variables: GP_QUALITY <chr>,
## #   DEPTH <dbl>, SOUNDING_TYPE <chr>,
## #   YEARSUNK <dbl>, HISTORY <chr>
```

Reading in flat files

Also, check that you have the number of rows and columns that you expect:

```
dim(shipwrecks)
```

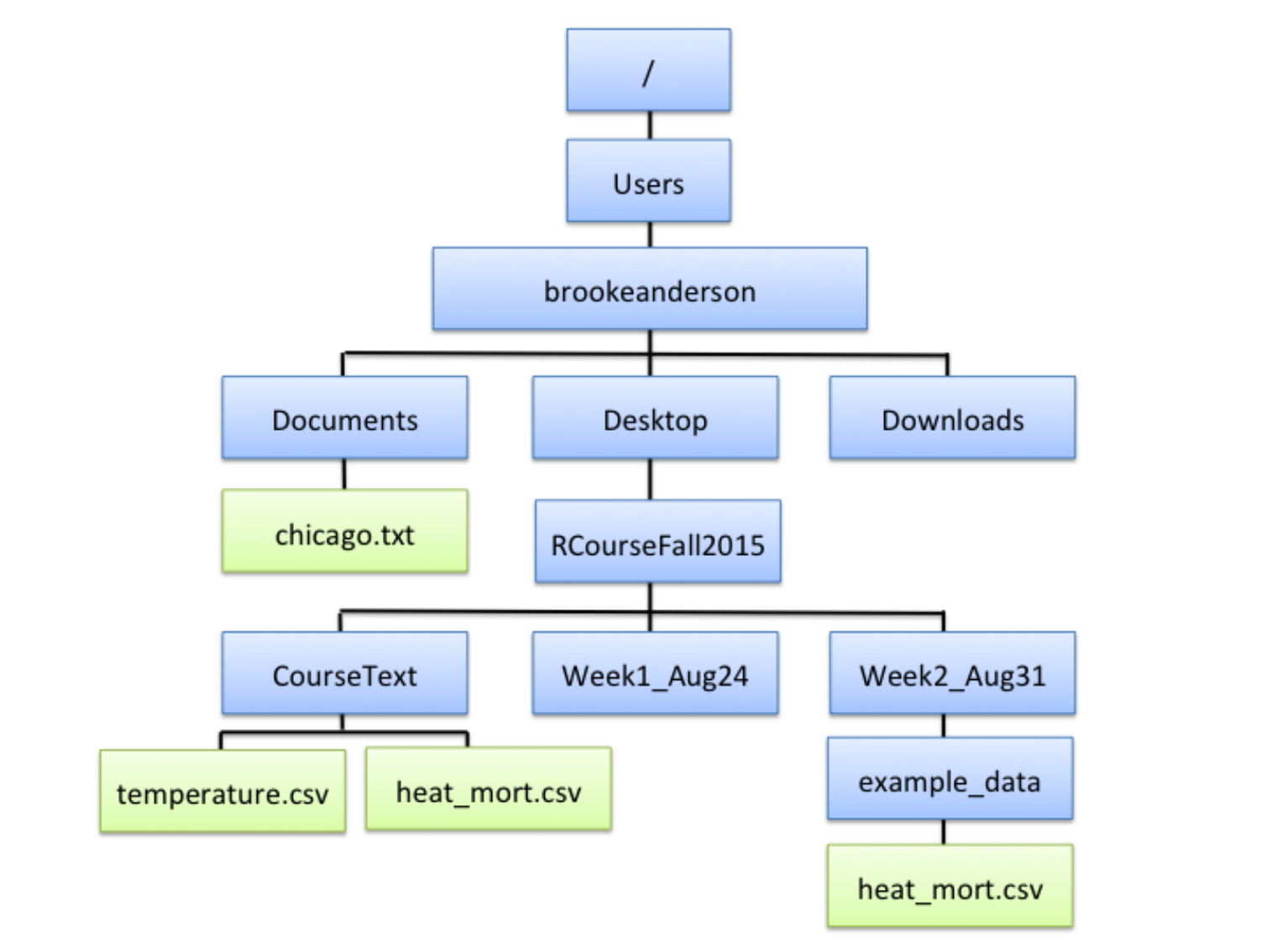
```
## [1] 197  10
```

In-course exercise

We'll take break here to work on the start of the In-Course Exercise (Sections 1 and 2).

Directory structure

Computer directory structure



Working directory

The **working directory** is the directory within your directory structure from which your R session is currently working.

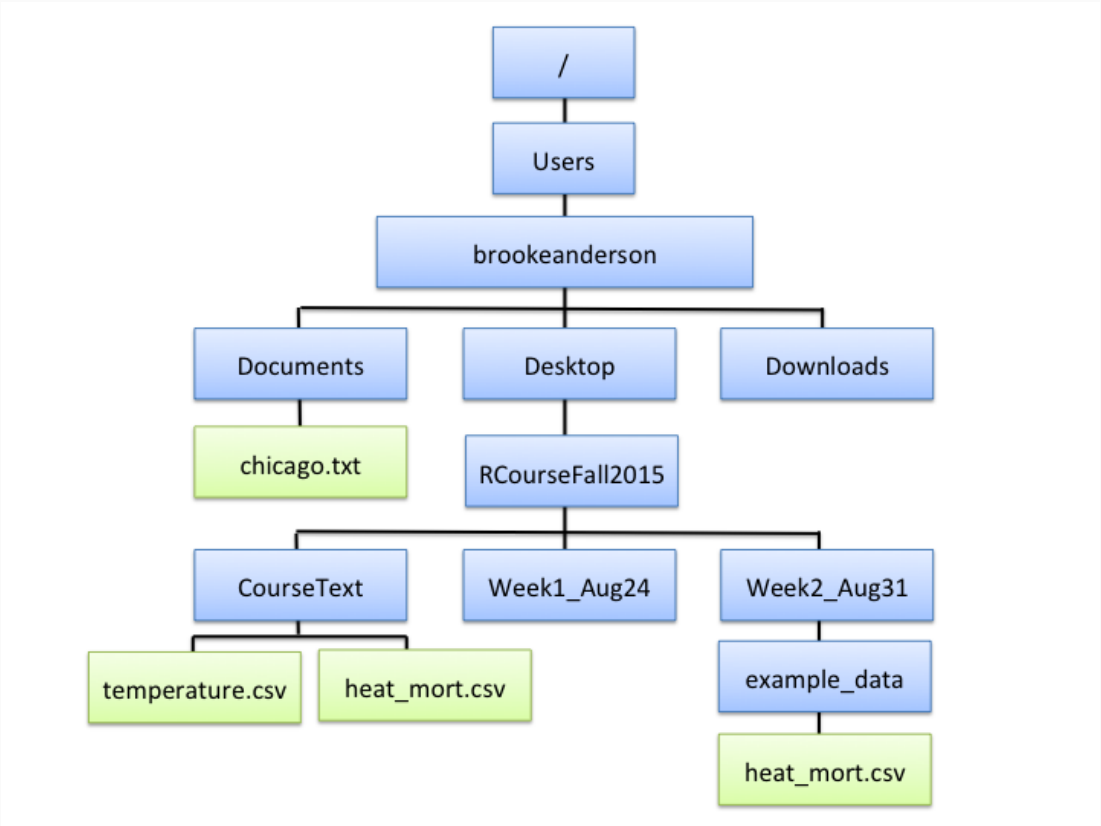
When you use **R Projects** to organize your work and files in R, anytime you open one of your R Projects, your working directory will automatically be that project's directory.

Working directory

To confirm this, open one of your R Projects and print out your working directory using the function `getwd()`:

```
getwd()
```

```
[1] "/Users/brookeanderson/Desktop/RCourseFall2015"
```

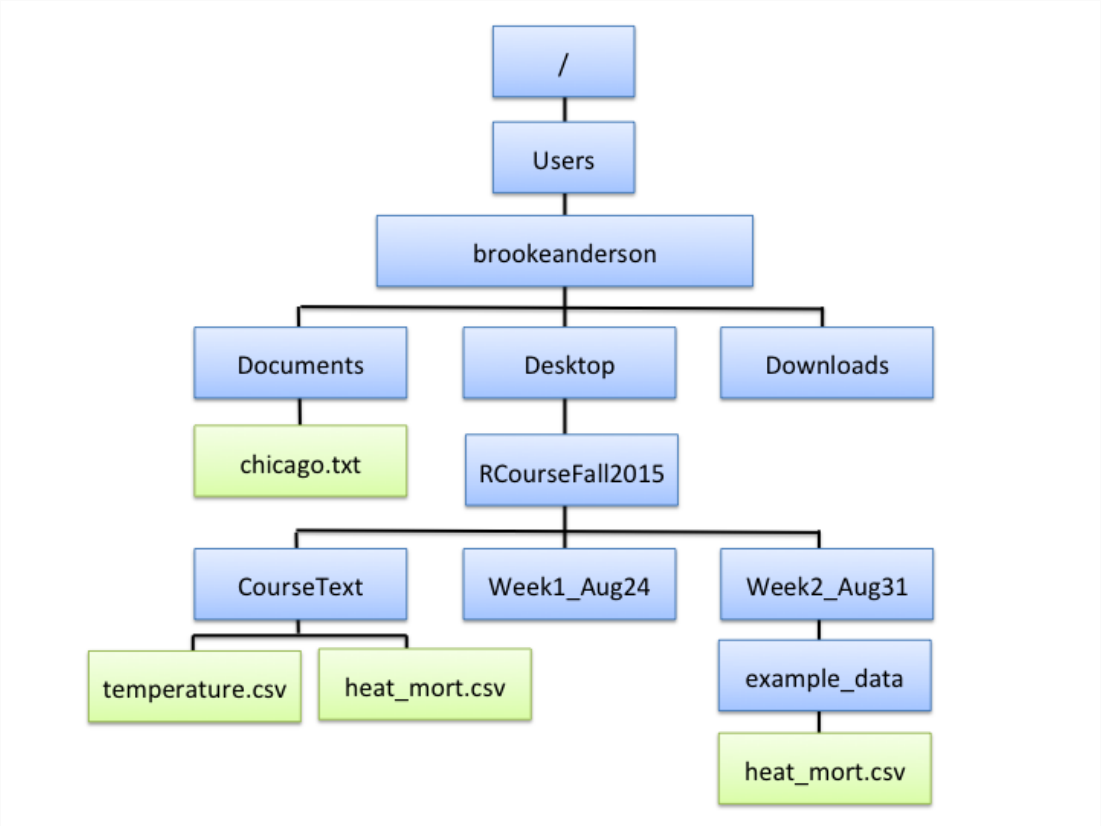


Working directory

You can use the `list.files()` function to list all the files in your current working directory:

```
list.files()
```

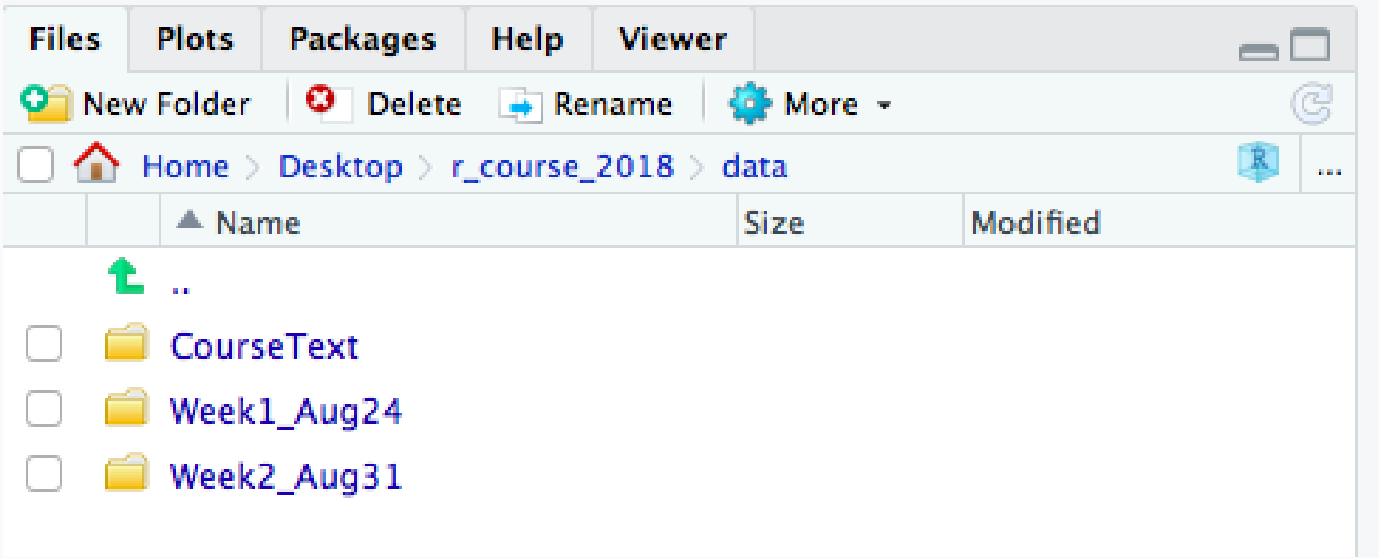
```
[1] "CourseText"    "Week1_Aug24"    "Week2_Aug31"
```



Working directory

The "Files" pane in RStudio (often on the lower right) will also show you the files available in your current working directory.

This should line up with what you get if you run `list.files()`.

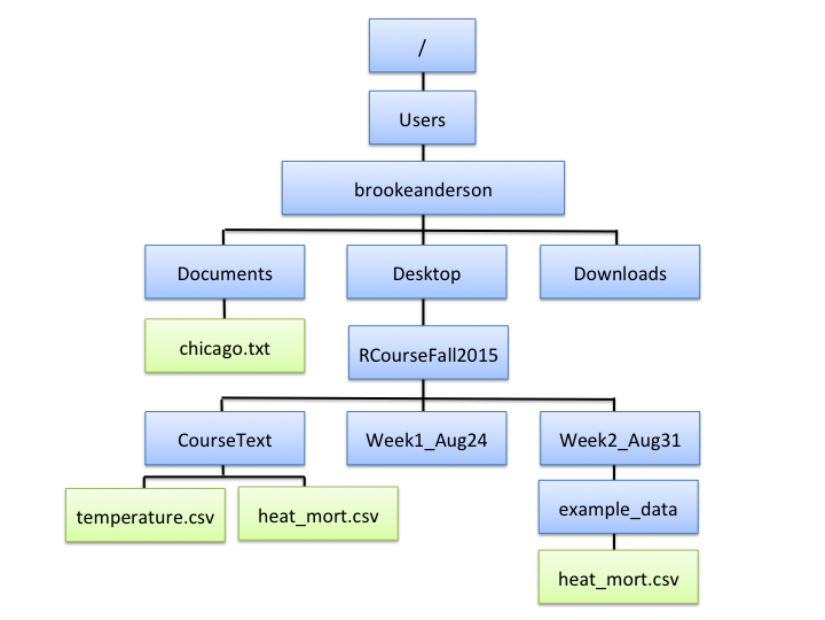


Working directory

When you run `list.files()`, if there is a name without a file extension, it's probably the name of a **subdirectory** of your current working directory. To list the files in one of these subdirectories, you can use `list.files` with that subdirectory's name.

```
list.files(path = "CourseText")
```

```
[1]  "temperature.csv"      "heat_mort.csv"
```



Relative versus absolute pathnames

When you want to reference a directory or file that is not in your working directory, you need to give R the directions for how to find the file. You can use one of two types of pathnames:

- *Relative pathname*: How to get to the file or directory from your current working directory
- *Absolute pathname*: How to get to the file or directory from anywhere on the computer

Relative versus absolute pathnames

Say your current working directory was `/Users/brookeanderson/RProgrammingForResearch` and you wanted to get into the subdirectory `data`. Here are examples of referencing that subdirectory using the two types of pathnames:

Absolute:

```
"/Users/brookeanderson/RProgrammingForResearch/data"
```

Relative:

```
"data"
```

Relative versus absolute pathnames

Both methods of writing filenames have their own advantages and disadvantages:

- *Relative pathname*: Which file you are indicating depends on which working directory you are in, which means that your code will break if you try to re-run it from a different working directory. However, relative pathways in your code make it easier for you to share a working version of a project with someone else. For most of this course, we will focus on using relative pathnames, especially when you start collaborating.
- *Absolute pathname*: No matter what working directory you're in, it is completely clear to your computer which file you mean when you use an absolute pathname. However, your code will not work on someone else's computer without modifications (because the structure of their computer's full directory will be different).

Relative versus absolute pathnames

I **strongly** recommend saving your data files somewhere in the directory structure of the R Project in which you're working and then using **relative pathnames** to reference that file when you need to read it in.

This practice establishes good habits for making your research computationally reproducible.

Getting around directories

There are a few abbreviations you can use to represent certain relative locations:

Shorthand	Meaning
`.`	Current working directory
`..`	One directory up from current working directory (parent directory)
`../..`	Two directories up from current working directory
`../data`	The 'data' subdirectory of the parent directory

One other useful abbreviation is `~`, which is shorthand for your **home directory**. On Windows, this will take you to your "Documents" folder.

Relative versus absolute pathnames

Here are some examples of relative pathnames that use these abbreviations:

If `data` is a subdirectory of your current parent directory (i.e., from your working directory, you need to go "up and over"):

```
"../data"
```

If `data` is a subdirectory of the subdirectory `Ex` of your current working directory:

```
"Ex/data"
```

Reading in an online flat file

Once you understand this idea of giving R directions to a file to read it, you shouldn't be too surprised that R can also do this for flat files that are hosted online.

In this case, the file isn't even on your computer, so you need to give R the directions to find it online. You do that by putting the file's online address as the file name. For example, to read in the shipwreck data directly from GitHub, you can run:

```
shipwreck_url <- paste0("https://raw.githubusercontent.com/",
                        "geanders/RProgrammingForResearch/",
                        "master/data/",
                        "AWOIS_Wrecks_KnownYear.tab")
shipwrecks <- read_tsv(shipwreck_url)
```

Taking advantage of `paste0`

You can create an object with your directory name using `paste0`, and then use that to set your directory. We'll take a lot of advantage of this for reading in files.

The convention for `paste0` is:

```
## Generic code
[object name] <- paste0("[first thing you want to paste]",
                        "[what you want to add to that]",
                        "[more you want to add]")
```

Taking advantage of `paste0`

Here's an example:

```
shipwreck_url <- paste0("https://raw.githubusercontent.com/",
                        "geanders/RProgrammingForResearch/",
                        "master/data/",
                        "AWOIS_Wrecks_KnownYear.tab")
shipwrecks <- read_tsv(shipwreck_url)
```

In-course exercise

We'll take a break now to do the next part of the in-course exercise (Section 3).

Data cleaning

Cleaning data

Common data-cleaning tasks include:

Task	`dplyr` functions
Renaming columns	`rename`
Extracting certain columns	`select`
Extracting or arranging rows	`slice`, `sample_n`, `filter`, `arrange`
Adding or changing columns	`mutate`

The “tidyverse”

Today, we’ll talk about using functions from the `dplyr` pakcage, as well as a package for working with a specific type of data (`stringr` for character strings, which is part of the “tidyverse”, like the `readr` package.

To use these functions, you’ll need to load those packages:

```
library("dplyr")
library("stringr")
```

Cleaning data

As an example of cleaning data, we'll work with the Daily Show data:

```
daily_show <- read_csv("data/daily_show_guests.csv",
                       skip = 4)
```

```
head(daily_show, 3)
```

```
## # A tibble: 3 x 5
##    YEAR GoogleKnowlege_~ Show  Group
##   <dbl> <chr>            <chr> <chr>
## 1  1999 actor            1/11~ Acti~
## 2  1999 Comedian         1/12~ Come~
## 3  1999 television actr~ 1/13~ Acti~
## # ... with 1 more variable: Raw_Guest_List <chr>
```

Re-naming columns

A first step is often re-naming columns. It can be hard to work with a column name that is:

- long
- doesn't following the naming rules for R objects
- includes upper case

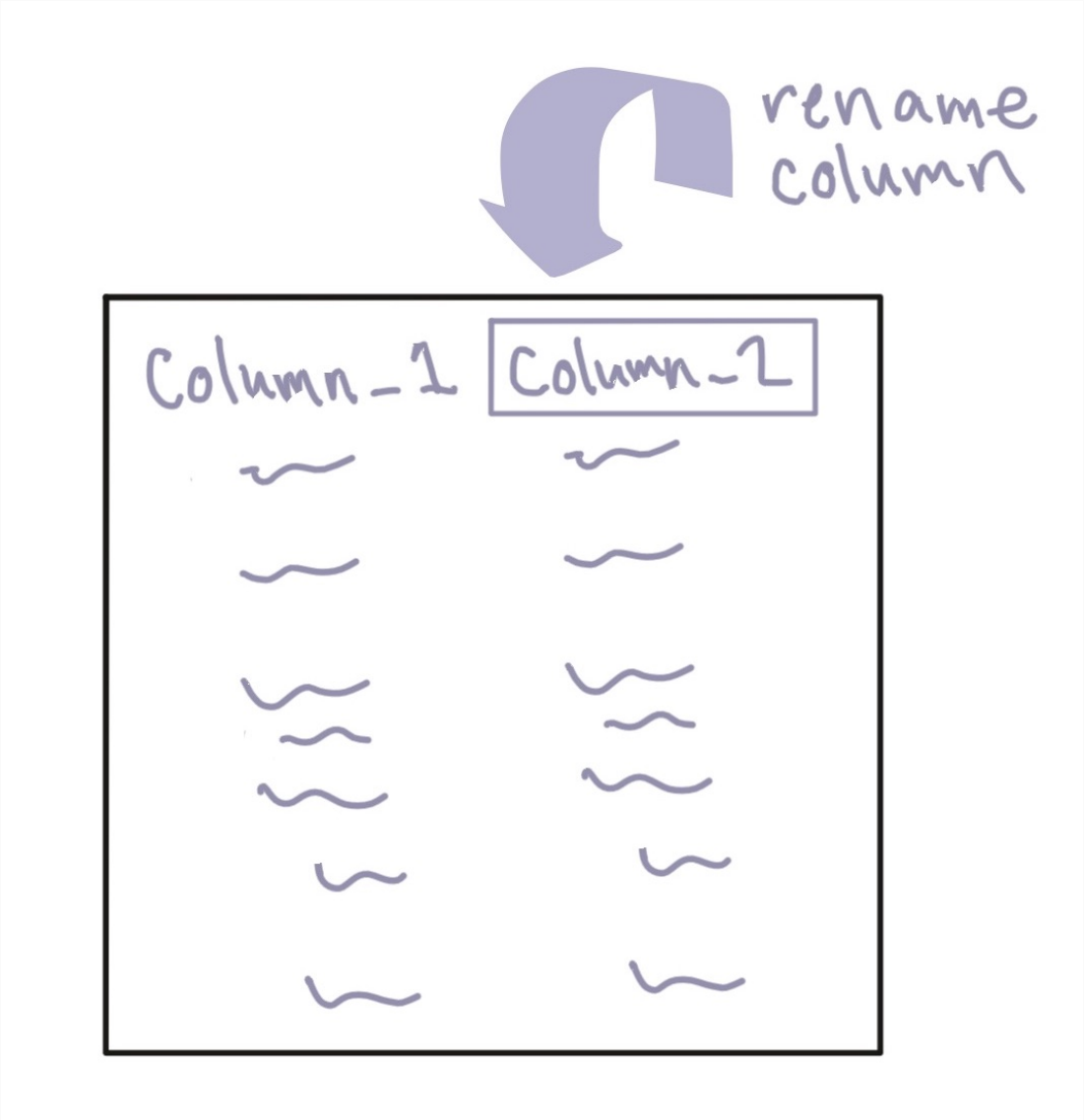
Several of the column names in `daily_show` have some of these issues:

```
colnames(daily_show)
```

```
## [1] "YEAR"
## [2] "GoogleKnowlege_Occupation"
## [3] "Show"
## [4] "Group"
## [5] "Raw_Guest_List"
```

Renaming columns

To rename these columns, use the `rename` function from `dplyr`.



Renaming columns

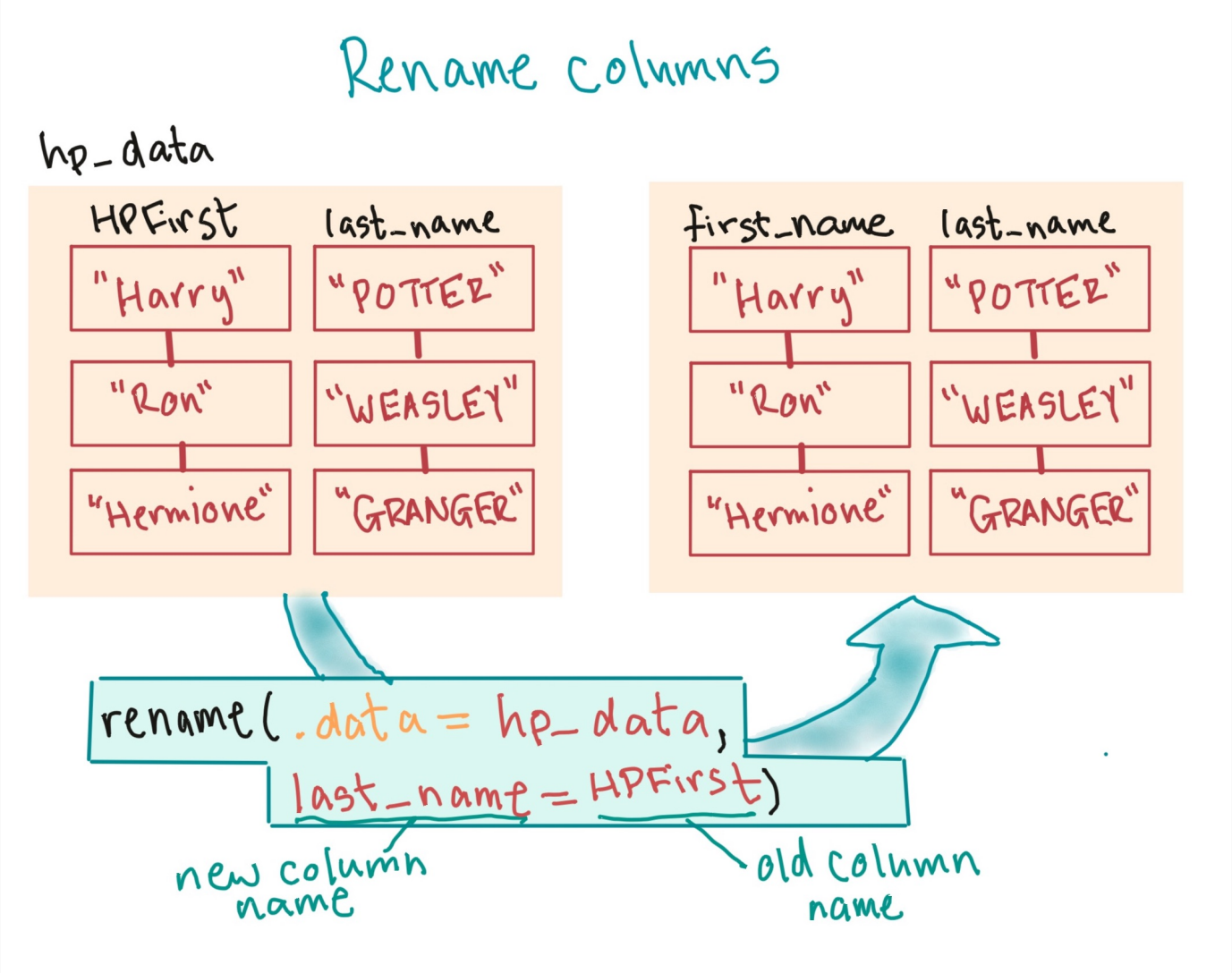
The basic syntax of `rename` is:

```
## Generic code
rename(dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
```

If you want to change column names in the saved object, be sure you reassign the object to be the output of `rename`.

Renaming columns

Here's a basic example of using `rename`:



Renaming columns

To rename columns in the daily_show data, then, use:

```
daily_show <- rename(daily_show,
                     year = YEAR,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
head(daily_show, 3)
```

```
## # A tibble: 3 x 5
##    year job           date   category guest_name
##   <dbl> <chr>         <chr>  <chr>    <chr>
## 1  1999 actor         1/11/~ Acting   Michael J. ~
## 2  1999 Comedian      1/12/~ Comedy   Sandra Bern~
## 3  1999 television a~ 1/13/~ Acting   Tracey Ullm~
```

Renaming columns

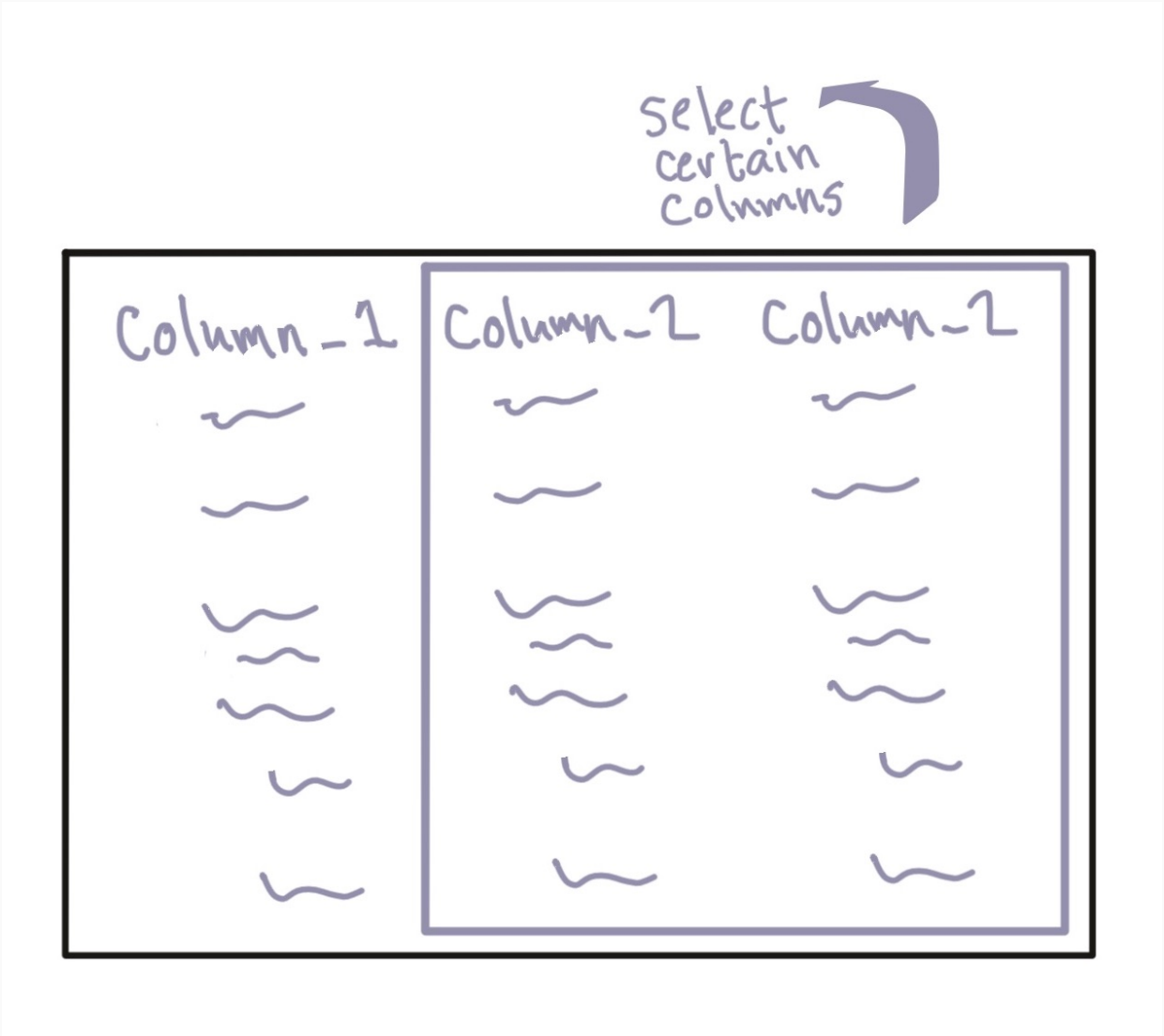
As a quick check, what is the difference between these two calls?

```
# 1.
rename(daily_show,
       year = YEAR,
       job = GoogleKnowlege_Occupation,
       date = Show,
       category = Group,
       guest_name = Raw_Guest_List)
```

```
# 2.
daily_show <- rename(daily_show,
                     year = YEAR,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
```

Extracting columns

Next, you may want to extract some columns of the dataframe. You can use the `select` function for this.



Extracting columns

You have already used `select` to extract columns by position. You can also use it to extract columns by name.

In this case, the basic structure of this command is:

```
## Generic code
select(.data = dataframe,
       column_name_1, column_name_2, ...)
```

Where `column_name_1`, `column_name_2`, etc., are the names of the columns you want to keep.

Extracting columns

For example, to extract all columns except `year` (since that information is already included in `date`), run:

```
select(.data = daily_show,
       job, date, category, guest_name)
```

```
## # A tibble: 2,693 x 4
##    job                date    category guest_name
##    <chr>              <chr>   <chr>    <chr>
##  1 actor              1/11/99 Acting   Michael J. Fox
##  2 Comedian           1/12/99 Comedy   Sandra Bernha~
##  3 television act~    1/13/99 Acting   Tracey Ullman
##  4 film actress       1/14/99 Acting   Gillian Ander~
##  5 actor              1/18/99 Acting   David Alan Gr~
##  6 actor              1/19/99 Acting   William Baldw~
##  7 Singer-lyricist    1/20/99 Musician Michael Stipe
##  8 model              1/21/99 Media    Carmen Electra
##  9 actor              1/25/99 Acting   Matthew Lilla~
```

Extracting columns

As a reminder, we could have selected these columns using position, too:

```
select(.data = daily_show, 2:5)
```

```
## # A tibble: 2,693 x 4
##    job                date    category guest_name
##    <chr>              <chr>   <chr>    <chr>
##  1 actor              1/11/99 Acting   Michael J. Fox
##  2 Comedian           1/12/99 Comedy   Sandra Bernha~
##  3 television act~    1/13/99 Acting   Tracey Ullman
##  4 film actress       1/14/99 Acting   Gillian Ander~
##  5 actor              1/18/99 Acting   David Alan Gr~
##  6 actor              1/19/99 Acting   William Baldw~
##  7 Singer-lyricist    1/20/99 Musician Michael Stipe
##  8 model              1/21/99 Media    Carmen Electra
##  9 actor              1/25/99 Acting   Matthew Lilla~
## 10 stand-up comed~    1/26/99 Comedy   David Cross
## # ... with 2,683 more rows
```

Extracting columns

The `select` function also provides some time-saving tools. For example, in the last example, we wanted all the columns except one. Instead of writing out all the columns we want, we can use - with the columns we don't want to save time:

```
daily_show <- select(daily_show, -year)
head(daily_show, 3)
```

```
## # A tibble: 3 x 4
##   job                 date   category guest_name
##   <chr>               <chr>  <chr>    <chr>
## 1 actor               1/11/~ Acting   Michael J. Fox
## 2 Comedian            1/12/~ Comedy   Sandra Bernha~
## 3 television actre~ 1/13/~ Acting   Tracey Ullman
```

Extracting columns

Another cool trick with `select` is that, if you want to keep several columns in a row, you can use a colon (`:`) with column names (rather than column position numbers) to select those columns:

```
daily_show <- select(daily_show, job:guest_name)
```

This call says that we want to select all columns from the one named "job" to the one named "guest_name".

Extracting columns

The `select` function has some cool extra options that we'll explore later in the course, including:

- Selecting all columns that start with a certain pattern
- Selecting all columns that end with a certain pattern
- Selecting all columns that contain a certain pattern

Extracting and rearranging rows

Next, we'll go deeper into how to extract certain rows, building on what we covered in the first week.

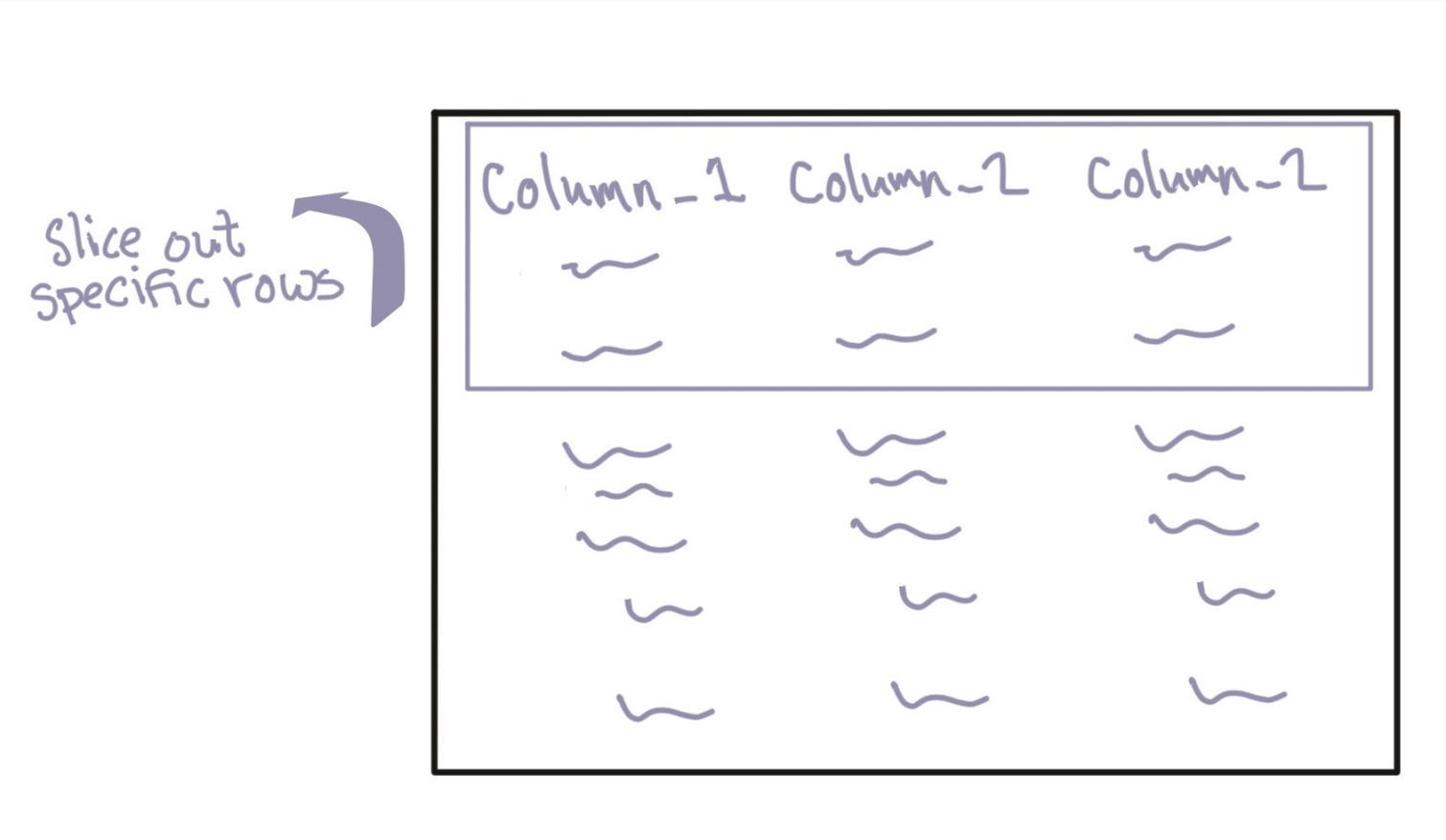
There are a few functions that are useful for extracting or rearranging rows in a dataframe:

- `slice`
- `sample_n`
- `arrange`
- `filter`

We'll go through what each of these does and how to use them.

Slicing to certain rows

The `slice` function from the `dplyr` package can extract certain rows based on their position in the dataframe.



Slicing to certain rows

Last week, you learned how to use the `slice` function to limit a dataframe to certain rows by row position.

For example, to print the first three rows of the `daily_show` data, you can run:

```
library("dplyr")
slice(.data = daily_show, 1:3)
```

```
## # A tibble: 3 x 4
##   job                 date   category guest_name
##   <chr>               <chr>  <chr>    <chr>
## 1 actor               1/11/~ Acting   Michael J. Fox
## 2 Comedian            1/12/~ Comedy   Sandra Bernha~
## 3 television actre~ 1/13/~ Acting   Tracey Ullman
```

Randomly sampling rows

There are some other functions you can use to extract rows from a tibble dataframe, all from the "dplyr" package.

For example, if you'd like to extract a random subset of *n* rows, you can use the `sample_n` function, with the `size` argument set to *n*.

To extract two random rows from the `daily_show` dataframe, run:

```
sample_n(tbl = daily_show, size = 2)
```

```
## # A tibble: 2 x 4
##   job       date     category guest_name
##   <chr>     <chr>    <chr>    <chr>
## 1 actor     2/9/99   Acting   Brendan Fraser
## 2 Professor 12/11/13 Academic Reza Aslan
```

Arranging rows

There is also a function, `arrange`, you can use to re-order the rows in a dataframe based on the values in one of its columns. The syntax for this function is:

```
# Generic code
arrange(.data = dataframe, column_to_order_by)
```

If you run this function to use a character vector to order, it will order the rows alphabetically by the values in that column. If you specify a numeric vector, it will order the rows by the numeric value.

Arranging rows

For example, we could reorder the `daily_show` data alphabetically by the values in the `category` column with the following call:

```
daily_show <- arrange(.data = daily_show, category)
slice(.data = daily_show, 1:3)
```

```
## # A tibble: 3 x 4
##   job       date    category guest_name
##   <chr>     <chr>   <chr>    <chr>
## 1 professor 10/3/01 Academic Stephen S. Morse
## 2 Professor 12/3/01 Academic Nadine Strossen
## 3 Historian 11/4/03 Academic Michael Beschloss
```

Arranging rows

If you want the ordering to be reversed (e.g., from "z" to "a" for character vectors, from higher to lower for numeric, latest to earliest for a Date), you can include the `desc` function.

For example, to reorder the `daily_show` data by job category in descending alphabetical order, you can run:

```
daily_show <- arrange(.data = daily_show,
                      desc(x = category))
slice(.data = daily_show, 1:2)
```

```
## # A tibble: 2 x 4
##   job          date    category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 neurosurgeon 4/28/03 Science  Dr Sanjay Gupta
## 2 scientist    1/13/04 Science  Catherine Weitz
```

Filtering to certain rows

Next, you might want to filter the dataset down so that it only includes certain rows. You can use the `filter` function from the `dplyr` package to do that. The syntax is:

```
## Generic code
filter(.data = dataframe, logical expression)
```

Filtering to certain rows

The **logical expression** gives the condition that a row must meet to be included in the output data frame. For example, you might want to pull:

- Rows from 2015
- Rows where the guest was an academic
- Rows where the job is not missing

Filtering to certain rows

For example, the == logical operator tests if two values are equal. So if you want to create a data frame that only includes guests who were scientists, you can run:

```
scientists <- filter(.data = daily_show, category == "Science")
head(scientists)
```

```
## # A tibble: 6 x 4
##   job          date    category guest_name
##   <chr>        <chr>   <chr>    <chr>
## 1 neurosurgeon 4/28/03 Science  Dr Sanjay Gupta
## 2 scientist    1/13/04 Science  Catherine Weitz
## 3 physician    6/15/04 Science  Hassan Ibrahim
## 4 doctor       9/6/05  Science  Dr. Marc Siegel
## 5 astronaut    2/13/06 Science  Astronaut Mike Mu~
## 6 Astrophysic~ 1/30/07 Science  Neil deGrasse Tys~
```

Common logical and relational operators in R

To build a logical statement to use in `filter`, you'll need to know some of R's logical and relational operators:

Operator	Meaning	Example
`==`	equals	`category == "Acting"`
`!=`	does not equal	`category != "Comedy`
`%in%`	is in	`category %in% c("Academic", "Science")`
`is.na()`	is NA	`is.na(job)`
`&`	and	`(year == 2015) & (category == "Academic")`
`|`	or	`(year == 2015) | (category == "Academic")`

Common logical and relational operators in R

We will cover logical operators and expressions in depth next week.

As a preview, the == operator will check each element of a vector against each corresponding element of another vector to see if it's equal.

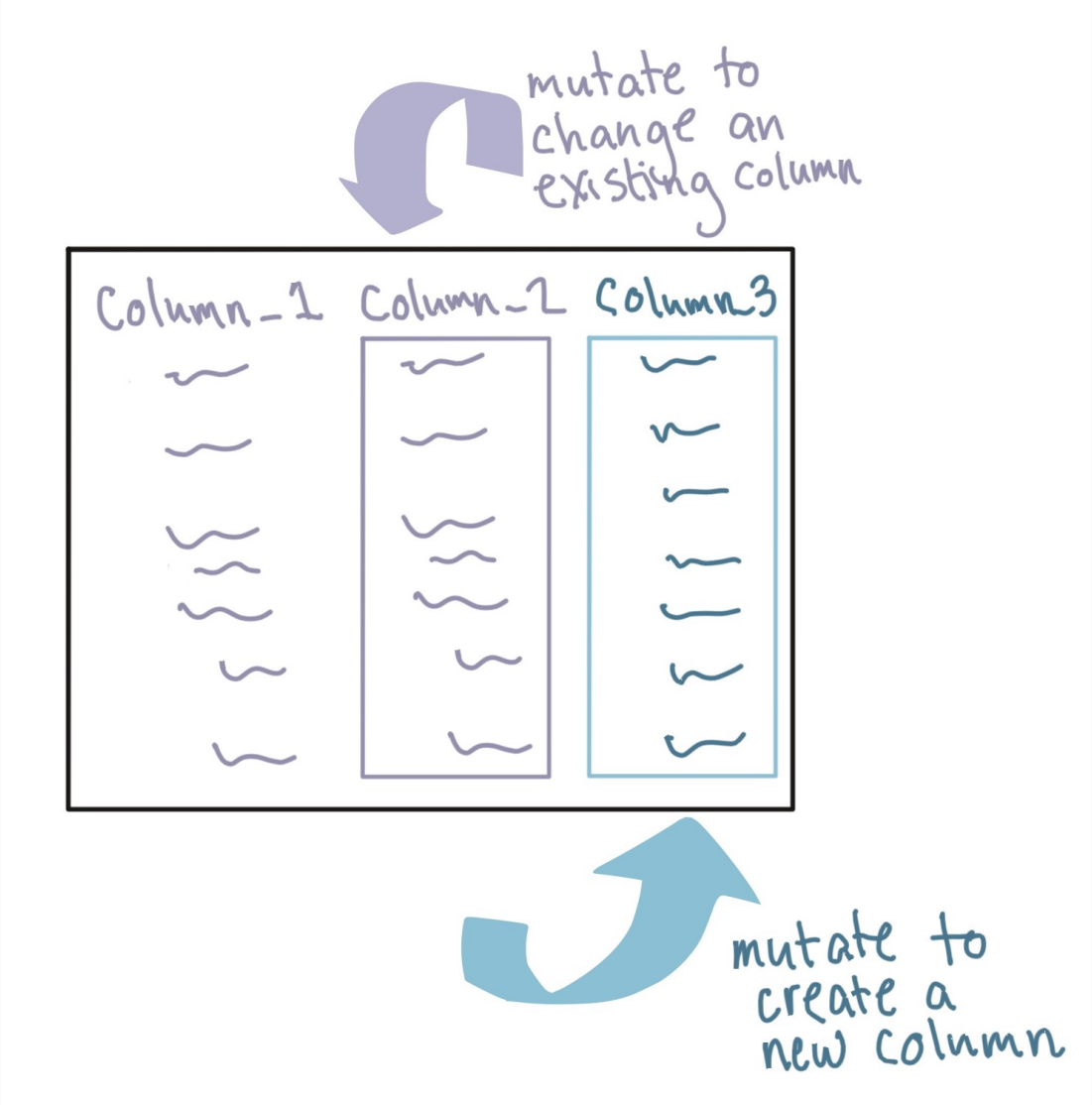
The result will be a **logical vector**:

```
c(1, 2, 3) == c(1, 2, 4)
```

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

Add or change columns

You can **change** a column or **add** a new column using the `mutate` function from the `dplyr` package.



Add or change columns

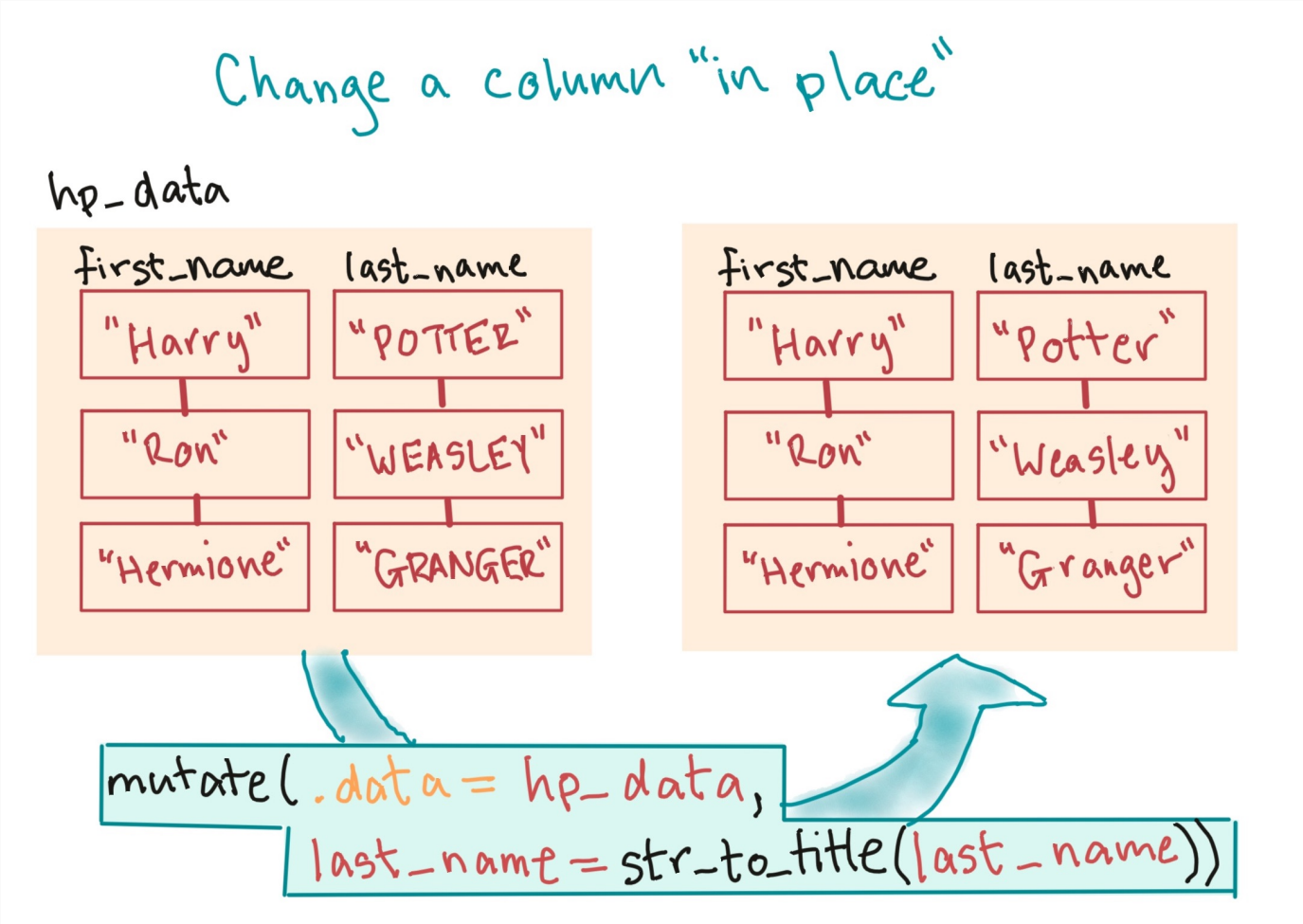
The `mutate` function has the syntax:

```
# Generic code
mutate(.data =dataframe,
       changed_column = function(original column),
       new_column = function(original columns))
```

- If you want to just **change** a column (in place), use its original name on the left of the equation.
- If you want to **add** a new column, use a new column name (you pick what that name will be) on the left of the equation (this will be the name of the new column).

Add or change columns

You can use `mutate` to *change* an existing column.



Add or change columns

For example, the `job` column in `daily_show` sometimes uses upper case and sometimes does not:

```
daily_show$job[1:10]
```

```
##  [1] "neurosurgeon"   "scientist"
##  [3] "physician"      "doctor"
##  [5] "astronaut"      "Astrophysicist"
##  [7] "Surgeon"        "physician"
##  [9] "Astrophysicist" "Neuroscientist"
```

Add or change columns

There is a package called `stringr` for working with character strings.

We could use the `str_to_lower` function from the `stringr` package to make all listings lowercase:

```
library("stringr")
daily_show <- mutate(.data = daily_show,
                     job = str_to_lower(string = job))
```

Add or change columns

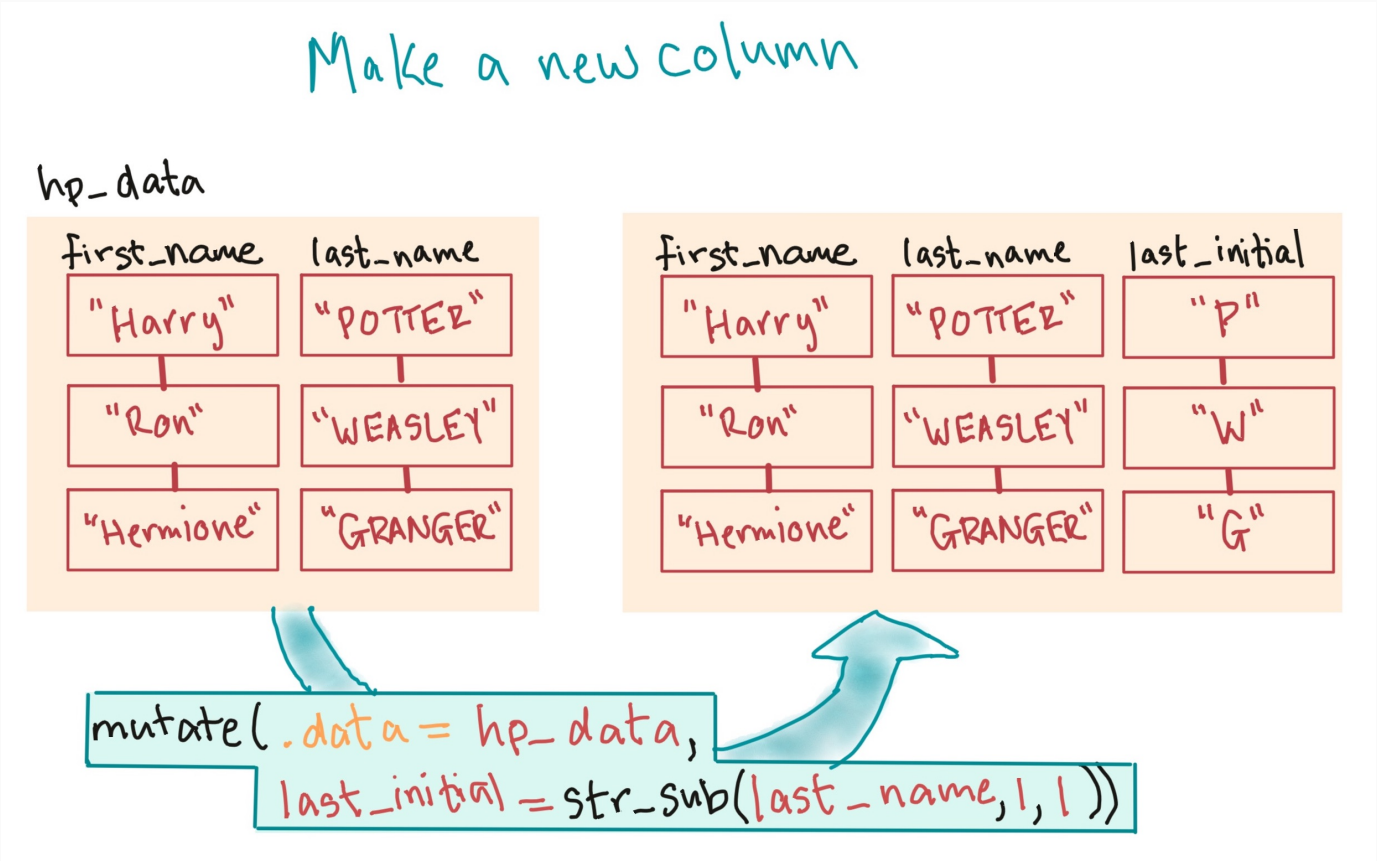
Now all job listings are in lowercase:

```
daily_show$job[1:10]
```

```
##  [1] "neurosurgeon"   "scientist"
##  [3] "physician"      "doctor"
##  [5] "astronaut"      "astrophysicist"
##  [7] "surgeon"        "physician"
##  [9] "astrophysicist" "neuroscientist"
```

Add or change columns

Alternatively, you can also use `mutate` to *add* a new column to the dataframe.



Add or change columns

For example, you could add a column called `uc_job` with the job name in uppercase letters:

```
daily_show <- mutate(.data = daily_show,
                     uc_job = str_to_upper(string = job))
slice(select(.data = daily_show, job, uc_job), 1:3)
```

```
## # A tibble: 3 x 2
##   job          uc_job
##   <chr>        <chr>
## 1 neurosurgeon NEUROSURGEON
## 2 scientist    SCIENTIST
## 3 physician    PHYSICIAN
```

In-course exercise

We'll take a break now and do sections 4 and 5 of the In-Course Exercise for Week 2.

Piping

“Composing” to combine function calls

There’s one additional way to compose several function calls into one R expression.

There’s a special “pipe” function (%>%) in a package called “magrittr” that you can use to compose a large R expression from several smaller function calls. This will “pipe” the object that is created from one function call into the first argument for the next function call.

For composing two simple function calls, here’s how that looks:

```
library(package = "magrittr")

paste(x = "Hello world") %>%
  print()
```

```
## [1] "Hello world"
```

“Composing” to combine function calls

Here’s another example. Earlier, we used something like the following code to extract certain rows and columns from a dataframe:

```
select(.data = slice(.data = daily_show, c(1:2)), c(1, 4))
```

You can use a pipe to express this much more cleanly:

```
slice(.data = daily_show, c(1:2)) %>%
  select(c(1, 4))
```

“Composing” to combine function calls

You can even move the dataframe out of the first function call, and move it to the start of the expression, where you “pipe” it into the first function call in the expression:

```
daily_show %>%
  slice(c(1:2)) %>%
  select(c(1, 4))
```

Piping

Piping

If you look at the format of these dplyr functions, you'll notice that they all take a dataframe as their first argument:

```
# Generic code
rename(.data = dataframe,
       new_column_name_1 = old_column_name_1,
       new_column_name_2 = old_column_name_2)
select(.data = dataframe, column_name_1, column_name_2)
slice(.data = dataframe, starting_row:ending_row)
arrange(.data = dataframe, column_to_order_by)
filter(.data = dataframe, logical_expression)
mutate(.data = dataframe,
       changed_column = function(original_column),
       new_column = function(original_columns))
```

Piping

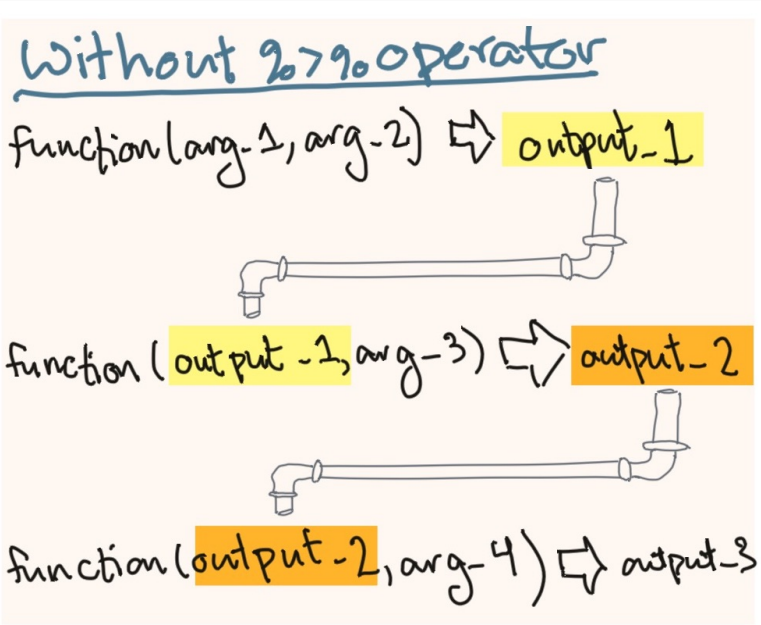
Classically, you would clean up a dataframe in R by reassigning the dataframe object at each step:

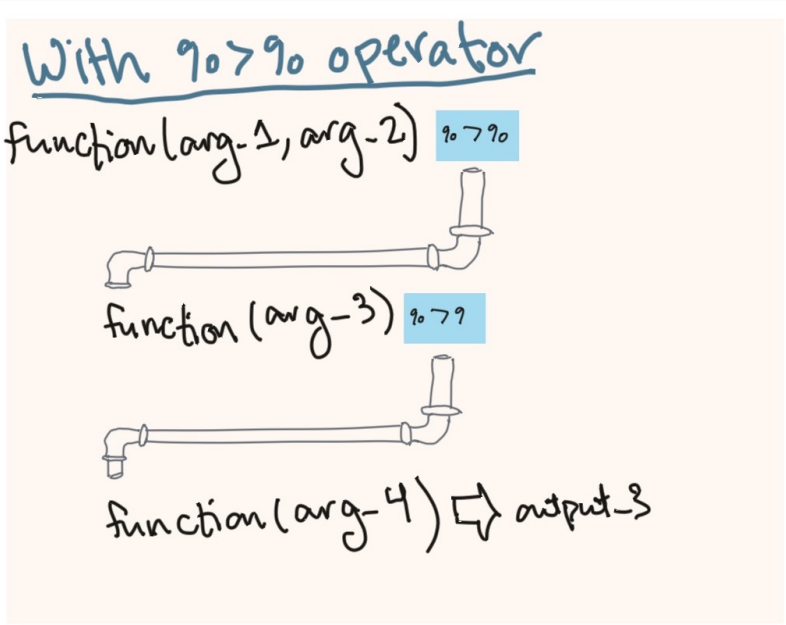
```
daily_show <-read_csv(file = "../data/daily_show_guests.csv",
                      skip = 4)
daily_show <- rename(.data = daily_show,
                     job = GoogleKnowlege_Occupation,
                     date = Show,
                     category = Group,
                     guest_name = Raw_Guest_List)
daily_show <- select(.data = daily_show, -YEAR)
daily_show <- mutate(.data = daily_show,
                     job = str_to_lower(job))
daily_show <- filter(.data = daily_show,
                     category == "Science")
```

Piping

"Piping" lets you clean this code up a bit. It can be used with any function that inputs a dataframe as its first argument. It "pipes" the dataframe created right before the pipe (%>%) into the function right after the pipe.

Piping


Without %>% operator
function(arg-1, arg-2)
output_1
function(output_1, arg-3)
output_2
function(output_2, arg-4)
output_3


With %>% operator
function(arg-1, arg-2)
%>%
function (arg-3)
%>%
function(arg-4)
output_3

Piping

With piping, the same data cleaning looks like:

```
daily_show <- read_csv("../data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #1: As you are trying to figure out what "piped" code like this is doing, try highlighting from the start of the code through just part of the pipe and run that. For example, try highlighting and running just from `read_csv` through the before the %>% in the line with `select`, and see what that output looks like.

```
daily_show <- read_csv("../data/daily_show_guests.csv",
                       skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #2: When you are writing an R script that uses piping, first write it and make sure you have it right **without** assigning it to an R object (i.e., no <-). Often you'll use piping to clean up an object in R, but if you have to work on the piping code, you end up with different versions of the object, which will cause frustrations.

```
read_csv("../data/daily_show_guests.csv", skip = 4) %>%
  rename(job = GoogleKnowlege_Occupation,
         date = Show,
         category = Group,
         guest_name = Raw_Guest_List) %>%
  select(-YEAR) %>%
  mutate(job = str_to_lower(job)) %>%
  filter(category == "Science")
```

Piping

Piping tip #3: There is a keyboard shortcut for the pipe symbol:

`Command-Shift-m`

In-course exercise

We'll take a break now and do section 6 of the In-Course Exercise for Week 2.

Odds and ends

`dplyr` versus base R

Just so you know, all of these actions also have alternatives in base R:

`dplyr`	Base R equivalent
`rename`	Reassign `colnames`
`select`	Square bracket indexing (`[ , ]`)
`slice`	Square bracket indexing (`[ , ]`)
`filter`	`subset`
`mutate`	Replacement assignment expression

You will see these alternatives used in older code examples.

Alternative package::function notation

The `library` function is the most common way you'll access functions from R packages, but it's not the only way to do this. There is another type of notation that will allow you to use any external function from any package you have installed on your computer: the `package::function` notation.

You can call functions by specifying the package name, a double colon, and then the function name you want to use from that package. For example:

```
stringr::str_to_upper("Astrophysicist")
```

```
## [1] "ASTROPHYSICIST"
```

Alternative package::function notation

The `package::function` notation is not typically used because it substantially increases how much you have to type in your code.

However, there can be cases where a function name is ambiguous. For example, you might want to use functions from two different packages that have the same name. In this case, using the `package::function` notation makes it crystal clear which function you mean.

In practice, this problem is most likely to come up when you've loaded both `plyr` and `dplyr`, which share several function names and are both popular packages.

Alternative package::function notation

There is another useful trick that you can do with the `package::function` notation.

RStudio has tab complemention, which means that once you start typing an object name or function, if you press the Tab key, RStudio will give you suggestions for how you can finish the name.

If you want to scroll through the names of all the external functions in a package, you can do so by typing something like `?stringr::` in the console (don't press Return) and then pressing the Tab key.