Introduction to R

Econ 337

Ray Caraher

2023-10-12

Section 1

Section: R Basics

Features of R

R is a programming language and software environment primarily used for statistical computing, data analysis, and data visualization.

- Open Source: R is an open-source language, which means it is freely available for anyone to use, modify, and distribute. This open nature has led to a vibrant community of users and developers who create and share packages, making R a dynamic and continually evolving language.
- Statistical Analysis: R is most known for its statistical capabilities. It offers a wide range of statistical techniques, from basic descriptive statistics to advanced modeling and hypothesis testing. The Open Source nature means R is the first language to tools to use offer cutting-edge methods.
- Data Manipulation: R provides extensive tools for data manipulation and transformation. It's particularly powerful when working with structured data, such as data frames and matrices. Packages like dplyr and tidyr make data cleaning and wrangling efficient.

Features of R

- Extensibility: R is highly extensible. Users can create their own functions and packages to extend R's capabilities. Many specialized packages have been developed for various fields, including economics, health, and labor economics.
- Reproducibility: R is designed for reproducible research. You can save your entire analysis process in a script, making it easy for others to replicate your work.
- Community and Documentation: R has a large and active user community. There are numerous online resources, forums, and documentation available for learning and troubleshooting.
- Integration: R can be integrated with other programming languages and tools

What is R Studio?

An integrated development environment (IDE) designed for working with the R programming language. It provides a user-friendly interface and a set of tools that make it easier to write, run, and manage R code and projects.

- Script Editor: RStudio includes a code editor where you can write and edit your R scripts.
- Console: The R console is an interactive environment where you can execute R code line by line.
- File Viewer: RStudio has a file viewer pane for navigating your project's directory structure.
- Plots and Graphics: RStudio supports the creation of plots and graphics using R packages like ggplot2.
- Customization: RStudio is highly customizable. You can adjust its appearance, configure code styling, and add or create custom extensions using R packages.

Coding basics

You can do basic math in R:

```
2 + 2
```

```
## [1] 4
```

```
20 * 5 / 87
```

```
## [1] 1.149425
```

```
sin(pi / 2)
```

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

```
sqrt(6)
```

```
## [1] 2.44949
```

Objects

R is an "Object Orientated Language", meaning data, variables, filepaths, and nearly everything else is stored as an "Object"

New objects can be created the the assignment operator:

```
x <- 2 + 5
```

Now the value of x is not printed, it's just stored. If you want to view the value, type x in the console.

```
x
```

```
## [1] 7
```

Using objects

You can then use this object like any other math variable

```
x + 10
```

```
## [1] 17
```

And you can assign something new to the object name

```
x <- x + 50
x
```

```
## [1] 57
```

All object assignment follows the object_name <- value format.

In general, good to use snake_case for naming objects, where everything is lowercase and spaces are seperated with "_".

Object types

R objects come in several different types:

- Vectors: The simpilist object, these contain numbers (1, 2, etc.), characters ("c", "hello", "my name is ray"), or logical values (TRUE, FALSE).
- Functions: Functions are also objects in R. They are used to encapsulate a set of instructions that can be executed repeatedly.
- Data frames: Data frames are used to store structured data, similar to a table or spreadsheet.
- Matrices: A matrix is a two-dimensional array that stores elements of the same data type.
- Factors: Factors are used to represent categorical data

Vectors

Vectors can be created using the c() function:

```
primes <- c(2, 3, 5, 7, 11, 13)
```

And basic arithmetic on vectors is applied to every element of of the vector:

```
primes * 2
```

```
## [1]  4  6 10 14 22 26
```

```
primes - 1
```

```
## [1]  1  2  4  6 10 12
```

You can also do math with vectors.

```
odds <- c(1, 3, 5, 6, 9, 11)
primes + odds
```

```
## [1]  3  6 10 13 20 24
```

Functions

R has a large collection of built-in functions that are "called" like this:

function_name(argument1 = value1, argument2 = value2, ...).

For example, the seq() function generates a vector of numbers based on the pattern you provide.

```
bb <- seq(from = 1, to = 30, by = 3)
bb
```

```
##  [1]  1  4  7 10 13 16 19 22 25 28
```

Here, from, to, and by are the arguments of the function, and they are assigned to values 1, 30, and 3, respectively.

We can save the output of functions to objects to use later, or as inputs in other functions!

Using ? before a function will pull up the docs where you can learn more about it (i.e., ?seq).

Data frames

Designed to represent structured data in a way that resembles a spreadsheet or database table. We can use head() to look at the first few rows of a data frame.

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Data frames

We can look at the different variable names/columns.

```
nrow(mtcars)
```

```
## [1] 32
```

```
ncol(mtcars)
```

```
## [1] 11
```

```
colnames(mtcars)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"
## [11] "carb"
```

Data frames

Unlike vectors, data frames can store data of all different types.

Each column in the data frame is stored as a vector which we can access using "$":

```
head(mtcars$mpg)
```

```
## [1] 21.0 21.0 22.8 21.4 18.7 18.1
```

Note here I am again using head() to get only the first few elements of the vector.

Comments

Anything presided by the "#" symbol will not be run in R.

We call these "comments" and they care useful for writing instructions, notes for yourself, to-do lists, or anything else you need to keep in your script that isn't code.

```
# Find the mean miles per gallon
mean(mtcars$mpg)
```

```
## [1] 20.09062
```

Section 2

Data Manipulation

The tidyverse

You will rarely get data in the right form to analyze it. Often you will need to summarize variables, created new variables, delete or re-code variables.

We will learn how to do all this using the dplyr package from the tidyverse.

We will learn all this using the nycflights13 data.

First, we need to load these packages.

Packages are extra functions, data, and other features that the R community has developed.

If you haven't done so yet, install them. You will only need to do this once (and then run it again every so often to update the packages).

```
#install.packages("tidyverse")
#install.packages("nycflights13")
```

And then we need to load them.

```
library(nycflights13)
library(tidyverse)
```

nycflights13

The nycflights13 data contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics.

You can review the documentation using ?flights.

The flights data is a "tibble" object, which is similar to a data frame but tidyverse-specific and has some special features.

nycflights13

Let's look at some data.

```
flights
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

nycflights13

There are better ways at looking at the data. With R Studio, we can use the View(flights) function to see the data as a spreadsheet. Another helpful way is to use head() as discussed above, or glimpse(), which shows information about the type of the columns and the first few values.

```
glimpse(flights)
```

```
## Rows: 336,776
## Columns: 19
## $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2~
## $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
## $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 558, ~
## $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 600, ~
## $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2, -1~
## $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 849,~
## $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 851,~
## $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7, -1~
## $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6", "~
## $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301, 4~
## $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N394~
## $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LGA",~
## $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IAD",~
## $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149, 1~
## $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 733, ~
## $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6, 6~
## $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 59, 0~
## $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0~
```

dplyr basics

These functions will solve the majority of data manipulation problems. They are designed to resemble "verbs" which operate on the underlying data.

They all work with the same logic.

- The first argument is always a data frame (or tibble)
- The subsequent arguments describe which columns to operate on using the variable names
- The output is always a new data frame, which can either be printed to the console or saved as a new object.

dplyr's verbs operate in 4 ways: on rows, on columns, on groups, or on tables.

We combine these dplyr "verbs" with the pipe-operator, $|>$, which can be thought of as the word "then".

Row verb 1: filter()

filter() allows you to keep rows based on the values of a column or set of columns. The first argument is a data frame, and the other arguments are conditions which must be true to keep the row.

For example, we could find all flights that departed more than 120 minutes (two hours) late:

```
flights |>
  filter(dep_delay > 120)
```

```
## # A tibble: 9,723 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      848       1835     853    1001    1950     851 MQ
##  2  2013     1     1      957        733     144    1056     853     123 UA
##  3  2013     1     1     1114        900     134    1447    1222     145 UA
##  4  2013     1     1     1540       1338     122    2020    1825     115 B6
##  5  2013     1     1     1815       1325     290    2120    1542     338 EV
##  6  2013     1     1     1842       1422     260    1958    1535     263 EV
##  7  2013     1     1     1856       1645     131    2212    2005     127 AA
##  8  2013     1     1     1934       1725     129    2126    1855     151 MQ
##  9  2013     1     1     1938       1703     155    2109    1823     166 EV
## 10  2013     1     1     1942       1705     157    2124    1830     174 MQ
## # ... with 9,713 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Row verb 1: filter()

Other conditions we can use are: - $>$ (greater than) - $<$ (less than) - $<=$ (less than or equal to) - $>=$ (greater than or equal to) - $==$ (equal to) - $!=$ (not equal to)

Conditions can be combined with & (and) or | (or)

Row verb 1: filter()

For flights that departed January 1:

```
flights |>
  filter(month == 1 & day == 1)
```

```
## # A tibble: 842 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 832 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Row verb 1: filter()

For flights that departed in January of February:

```
flights |>
  filter(month == 1 | month == 2)
```

```
## # A tibble: 51,955 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 51,945 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Row verb 1: filter()

Don't make these common mistakes!

- Using $=$ instead of $==$ when testing for equality
  - WRONG: filter(month $=$ 1)
  - RIGHT: filter(month $==$ 1)
- Writing "or" statements like in English
  - WRONG: filter(month $==$ 1 | 2)
  - RIGHT: filter(month $==$ 1 | month $==$ 2)

Row verb 2: arrange()

arrange() changes the order of the rows based on a column value.

Let's arrange the rows by departure time:

```
flights |>
  arrange(dep_time)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1    13        1       2249      72     108    2357      71 B6
##  2  2013     1    31        1       2100     181     124    2225     179 WN
##  3  2013    11    13        1       2359       2     442     440       2 B6
##  4  2013    12    16        1       2359       2     447     437      10 B6
##  5  2013    12    20        1       2359       2     430     440     -10 B6
##  6  2013    12    26        1       2359       2     437     440      -3 B6
##  7  2013    12    30        1       2359       2     441     437       4 B6
##  8  2013     2    11        1       2100     181     111    2225     166 WN
##  9  2013     2    24        1       2245      76     121    2354      87 B6
## 10  2013     3     8        1       2355       6     431     440      -9 B6
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Row verb 2: arrange()

If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns.

For example, the following code sorts by the departure time, which is spread over four columns. We get the earliest years first, then within a year the earliest months, etc.

```
flights |>
  arrange(year, month, day, dep_time)
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Row verb 2: arrange()

Use desc() on a column inside of arrange() to re-order the data frame based on that column in descending (big-to-small) order. For example, this code orders flights from most to least delayed:

```
flights |>
  arrange(desc(dep_delay))
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     9      641        900    1301    1242    1530    1272 HA
##  2  2013     6    15     1432       1935    1137    1607    2120    1127 MQ
##  3  2013     1    10     1121       1635    1126    1239    1810    1109 MQ
##  4  2013     9    20     1139       1845    1014    1457    2210    1007 AA
##  5  2013     7    22      845       1600    1005    1044    1815     989 MQ
##  6  2013     4    10     1100       1900     960    1342    2211     931 DL
##  7  2013     3    17     2321        810     911     135    1020     915 DL
##  8  2013     6    27      959       1900     899    1236    2226     850 DL
##  9  2013     7    22     2257        759     898     121    1026     895 DL
## 10  2013    12     5      756       1700     896    1058    2020     878 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

Column row 1: mutate()

mutate() adds new columns that are calculated from existing columns or other objects.

Let's calculate gain, or how much time a delayed flight made up while traveling, and speed in miles per hour:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
```

```
## # A tibble: 336,776 x 21
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 11 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, gain <dbl>, speed <dbl>, and abbreviated
## #   variable names 1: sched_dep_time, 2: dep_delay, 3: arr_time,
## #   4: sched_arr_time, 5: arr_delay
```

Column row 1: mutate()

Where are the new columns? By default, mutate() adds them to the end of the data frame, so we can use the .before argument to add them to to the left instead.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
```

```
## # A tibble: 336,776 x 21
##     gain speed  year month   day dep_t~1 sched~2 dep_d~3 arr_t~4 sched~5 arr_d~6
##    <dbl> <dbl> <int> <int> <int>   <int>   <int>   <dbl>   <int>   <int>   <dbl>
##  1    -9  370.  2013     1     1     517     515       2     830     819      11
##  2   -16  374.  2013     1     1     533     529       4     850     830      20
##  3   -31  408.  2013     1     1     542     540       2     923     850      33
##  4    17  517.  2013     1     1     544     545      -1    1004    1022     -18
##  5    19  394.  2013     1     1     554     600      -6     812     837     -25
##  6   -16  288.  2013     1     1     554     558      -4     740     728      12
##  7   -24  404.  2013     1     1     555     600      -5     913     854      19
##  8    11  259.  2013     1     1     557     600      -3     709     723     -14
##  9     5  405.  2013     1     1     557     600      -3     838     846      -8
## 10   -10  319.  2013     1     1     558     600      -2     753     745       8
## # ... with 336,766 more rows, 10 more variables: carrier <chr>, flight <int>,
## #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: dep_time, 2: sched_dep_time, 3: dep_delay, 4: arr_time,
## #   5: sched_arr_time, 6: arr_delay
```

mutate() is a super versatile function with a lot of different arguments, so you should experiment with it and look at the documentation!

Column row 2: select()

Datasets can often come with thousands of variables you dont need. select() allows you to just keep the columns you are intersted in.

You can select by the name of the columns:

```
flights |>
  select(year, month, day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Column row 2: select()

You can select all the columns between year and day:

```
flights |>
  select(year:day)
```

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
## # ... with 336,766 more rows
```

Column row 2: select()

Or you can select all the columns which are characters:

```
flights |>
  select(where(is.character))
```

```
## # A tibble: 336,776 x 4
##    carrier tailnum origin dest
##    <chr>   <chr>   <chr>  <chr>
##  1 UA      N14228  EWR    IAH
##  2 UA      N24211  LGA    IAH
##  3 AA      N619AA  JFK    MIA
##  4 B6      N804JB  JFK    BQN
##  5 DL      N668DN  LGA    ATL
##  6 UA      N39463  EWR    ORD
##  7 B6      N516JB  EWR    FLL
##  8 EV      N829AS  LGA    IAD
##  9 B6      N593JB  JFK    MCO
## 10 AA      N3ALAA  LGA    ORD
## # ... with 336,766 more rows
```

Column row 2: select()

There is a ton of other stuff you can do!

- starts_with("abc"): matches names that begin with "abc".
- ends_with("xyz"): matches names that end with "xyz".
- contains("ijk"): matches names that contain "ijk".
- num_range("x", 1:3): matches x1, x2 and x3

and even more (look at the documentation with ?select)

The pipe

We have seen how |> is used when going from your data frame to your verb. But we can also use it to combine verbs! Recall that |> kind of functions like the word "then".

For example, imagine that you wanted to find the fast flights to Houston's IAH airport: you need to combine filter(), mutate(), select(), and arrange().

```
flights |>
  filter(dest == "IAH") |>
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))
```

```
## # A tibble: 7,198 x 7
##     year month   day dep_time carrier flight speed
##    <int> <int> <int>    <int> <chr>    <int> <dbl>
##  1  2013     7     9      707 UA         226  522.
##  2  2013     8    27     1850 UA        1128  521.
##  3  2013     8    28      902 UA        1711  519.
##  4  2013     8    28     2122 UA        1022  519.
##  5  2013     6    11     1628 UA        1178  515.
##  6  2013     8    27     1017 UA         333  515.
##  7  2013     8    27     1205 UA        1421  515.
##  8  2013     8    27     1758 UA         302  515.
##  9  2013     9    27      521 UA         252  515.
## 10  2013     8    28      625 UA         559  515.
## # ... with 7,188 more rows
```

Groups

dplyr excels at doing operations on data by group.

group_by()

Use group_by() to divide your dataset into groups meaningful for your analysis, such as by month, year, or destination.

```
flights |>
  group_by(month)
```

```
## # A tibble: 336,776 x 19
## # Groups:   month [12]
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

group_by()

group_by() doesn't change the data but, if you look closely at the output, you'll notice that the output indicates that it is "grouped by" month (Groups: month [12]).

All of the next operations will now be done "by month."

summarize()

group_by() isn't useful on its own, but when combined with summarize() it is very powerful. Let's say that we want to find the average delay by month. We can use group_by(), summarize(), and mean() to do this.

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay)
  )
```

```
## # A tibble: 12 x 2
##    month avg_delay
##    <int>     <dbl>
##  1     1        NA
##  2     2        NA
##  3     3        NA
##  4     4        NA
##  5     5        NA
##  6     6        NA
##  7     7        NA
##  8     8        NA
##  9     9        NA
## 10    10        NA
## 11    11        NA
## 12    12        NA
```

summarize()

What happened? We just got as bunch of NA values, R's method of dealing with missing data. This happened because some of the observed flights had missing data in the delay column, and so when we calculated the mean including those values, we got an NA result. We use the na.rm = T argument in mean() to tell R to drop these missing values.

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = T)
  )
```

```
## # A tibble: 12 x 2
##    month avg_delay
##    <int>     <dbl>
##  1     1     10.0
##  2     2     10.8
##  3     3     13.2
##  4     4     13.9
##  5     5     13.0
##  6     6     20.8
##  7     7     21.7
##  8     8     12.6
##  9     9      6.72
## 10    10      6.24
## 11    11      5.44
## 12    12     16.6
```

summarize()

We can do any number of oberations in the summarize() function. Let's also have R count the number of observations in each group using the n() function so we know the mean values aren't based on small sample sizes.

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = T),
    n_mnth = n()
  )
```

```
## # A tibble: 12 x 3
##    month avg_delay n_mnth
##    <int>     <dbl>  <int>
##  1     1     10.0   27004
##  2     2     10.8   24951
##  3     3     13.2   28834
##  4     4     13.9   28330
##  5     5     13.0   28796
##  6     6     20.8   28243
##  7     7     21.7   29425
##  8     8     12.6   29327
##  9     9      6.72  27574
## 10    10      6.24  28889
## 11    11      5.44  27268
## 12    12     16.6   28135
```

summarize()

You can also find the largest delay for each month

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = T),
    max_delay = max(dep_delay, na.rm = T),
    n_mnth = n()
  )
```

```
## # A tibble: 12 x 4
##    month avg_delay max_delay n_mnth
##    <int>     <dbl>     <dbl>  <int>
##  1     1     10.0       1301  27004
##  2     2     10.8        853  24951
##  3     3     13.2        911  28834
##  4     4     13.9        960  28330
##  5     5     13.0        878  28796
##  6     6     20.8       1137  28243
##  7     7     21.7       1005  29425
##  8     8     12.6        520  29327
##  9     9      6.72      1014  27574
## 10    10      6.24       702  28889
## 11    11      5.44       798  27268
## 12    12     16.6        896  28135
```

summarize()

Or maybe more usefully, we can find the average delay, largest delay, and number of flights by destination.

```
flights |>
  group_by(dest) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = T),
    max_delay = max(dep_delay, na.rm = T),
    n_flghts = n()
  )
```

```
## Warning: There was 1 warning in `summarize()`.
## i In argument: `max_delay = max(dep_delay, na.rm = T)`.
## i In group 52: `dest = "LGA"`.
## Caused by warning in `max()`:
## ! no non-missing arguments to max; returning -Inf
```

```
## # A tibble: 105 x 4
##    dest  avg_delay max_delay n_flghts
##    <chr>     <dbl>     <dbl>    <int>
##  1 ABQ       13.7        142      254
##  2 ACK        6.46       219      265
##  3 ALB       23.6        323      439
##  4 ANC       12.9         75        8
##  5 ATL       12.5        898    17215
##  6 AUS       13.0        351     2439
##  7 AVL        8.19       222      275
##  8 BDL       17.7        252      443
##  9 BGR       19.5        248      375
## 10 BHM       29.7        325      297
## # ... with 95 more rows
```

group_by() for multiple variables

We can also group by multiple variables. For example, if we want to create a group for each day, we can do the following. Notice how I am saving this grouped data frame as a separate object.

```
daily <- flights |>
  group_by(year, month, day)
daily
```

```
## # A tibble: 336,776 x 19
## # Groups:   year, month, day [365]
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

group_by() for muliple variables

Let's find the number of daily flights from NYC using this:

```
daily_flights <- daily |>
  summarize(
    n = n(),
    .groups = "keep"
  )
daily_flights
```

```
## # A tibble: 365 x 4
## # Groups:   year, month, day [365]
##     year month   day     n
##    <int> <int> <int> <int>
##  1  2013     1     1   842
##  2  2013     1     2   943
##  3  2013     1     3   914
##  4  2013     1     4   915
##  5  2013     1     5   720
##  6  2013     1     6   832
##  7  2013     1     7   933
##  8  2013     1     8   899
##  9  2013     1     9   902
## 10  2013     1    10   932
## # ... with 355 more rows
```

ungroup()

If no longer want do do operations by group on a data frame, we can use ungroup()

```
daily |>
  ungroup()
```

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      542        540       2     923     850      33 AA
##  4  2013     1     1      544        545      -1    1004    1022     -18 B6
##  5  2013     1     1      554        600      -6     812     837     -25 DL
##  6  2013     1     1      554        558      -4     740     728      12 UA
##  7  2013     1     1      555        600      -5     913     854      19 B6
##  8  2013     1     1      557        600      -3     709     723     -14 EV
##  9  2013     1     1      557        600      -3     838     846      -8 B6
## 10  2013     1     1      558        600      -2     753     745       8 AA
## # ... with 336,766 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

ungroup()

What happens if we do summarize() on an un-grouped data frame? It treats it as one big group (calculates the operations for all rows)

```
daily |>
  ungroup() |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    flights = n()
  )
```

```
## # A tibble: 1 x 2
##   avg_delay flights
##       <dbl>   <int>
## 1      12.6  336776
```

Putting it all together

You may now see how powerful the dpplyr verbs can be, especially when we chain them togther with our pipe.

For example, let's say we want to fly to LA (LAX) but want to chose the carrier which is the least likely to have a bad delay.

dplyr allows us to chain together our verbs like a sentence and get the answer.

```
flights |>
  filter(dest == "LAX") |>
  group_by(carrier) |>
  summarize(mean_delay = mean(dep_delay, na.rm = T)) |>
  arrange(mean_delay)
```

```
## # A tibble: 5 x 2
##   carrier mean_delay
##   <chr>        <dbl>
## 1 DL            5.67
## 2 B6            8.99
## 3 AA            9.17
## 4 VX           10.6
## 5 UA           10.8
```

Discussion questions for next week

We can also make our own tibbles within R using the tibble() function. For example, this table shows who is assigned to do discussion questions for next week.

```
disc_qs <- tibble(
  first_name = c("Kyle", "Paisley", "Shivani", "Timothy"),
  last_initial = c("H.", "G.", "A", "X."),
  week_num = rep(7, 4)
)
disc_qs
```

```
## # A tibble: 4 x 3
##   first_name last_initial week_num
##   <chr>      <chr>            <dbl>
## 1 Kyle       H.                   7
## 2 Paisley    G.                   7
## 3 Shivani    A                    7
## 4 Timothy    X.                   7
```

Group exercises

Do all of these in an R script in R studio.

1. Create a vector called x composed of all even numbers between 2 and 50 (hint: use seq()).
2. Transform this vector to one containing all the odd numbers between 3 and 51 without using seq().
3. Using the flights data, create a new data frame which contains only the flights going to LAX. Name this new object lax_flights (don't forget to use library()).
4. Using ?, find out what the sched_dep_time variable means and how to interpret it.
5. Using the lax_flights data frame, create a new variable which calculates the speed (in miles per hour) of each flight. Speed should be calculated as the distance divided by air_time times 60.
6. Find the average speed of all flights to LA, the maximum speed, and the minimum speed (don't forget to exclude NAs). Look at the documentation for min(), which we have yet to use.
7. Using the lax_flights data frame, group_by, and summarize(), find the NYC airport (Newerk [EWR], LaGuardia [LGA], and JFK) which has the lowest average departure delay. Notice that one origin airport is missing. What does that mean?
8. For each airport, find the carrier which has the largest number of flights to LA.