Data Analysis
Week 2: Tidying and Wrangling data using R

1 Getting started

This week we will demonstrate various techniques for **tidying** and **wrangling** data in R. From the 'Introduction to R Programming' course we are familiar with a data frame in R: a rectangular spreadsheet-like representation of data in R where the rows correspond to observations and the columns correspond to variables describing each observation. In Week 1 of Data Analysis, we started exploring the data frame `flights` included in the `nycflights13` package by creating visualisations of the data contained within said data frame.

Here we will discover a type of data formatting called **tidy** data. You will see that having data stored in the **tidy** format is about more than what the colloquial definition of the term **tidy** might suggest of having your data "neatly organised" in a spreadsheet. Instead, we define the term **tidy** in a more rigorous fashion, outlining a set of rules by which data can be stored and the implications of these rules on analyses.

Note: This session is based on Chapters 4 and 5 of the open-source book An Introduction to Statistical and Data Science via R which can be consulted at any point.

First, start by opening **RStudio** by going to `Desktop -> Maths-Stats -> RStudio`. Once RStudio has opened create a new R script by going to `File -> New File -> R Script`. Next go to `File -> Save As...` and save the script into your personal drive, either `M:` or `K:` (do not save it to the `H:` drive). We shall now load into R all of the libraries we will need for this session. This can be done by typing the following into your R script:

```
library(dplyr)
library(tidyr)
library(ggplot2)
library(readr)
library(stringr)
library(nycflights13)
library(fivethirtyeight)
```

The first five libraries above are part of the `tidyverse` collection of R packages, a powerful collection of data tools for transforming and visualising data. In particular, the first library `dplyr` provides functions for data wrangling or manipulation using a consistent 'grammar'. The second library `tidyr` helps us create **tidy** data, which we will now introduce. The final two libraries contain interesting data sets that we shall examine.

2 What is tidy data?

What does it mean for your data to be **tidy**? Beyond just being organised, having **tidy** data means that your data follows a standardised format. This makes it easier for you and others to visualise your data, to wrangle/transform your data, and to model your data. We will follow Hadley Wickham's definition of **tidy data** here:

> A data set is a collection of values, usually either numbers (if quantitative) or strings AKA text data (if qualitative). Values are organised in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units. An observation contains all values measured on the same unit (like a person, or a day, or a city) across attributes.

Tidy data is a standard way of mapping the meaning of a data set to its structure. A data set is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In tidy data:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

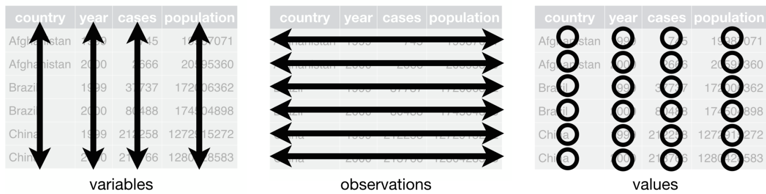


Figure 1: Tidy data graphic from http://r4ds.had.co.nz/tidy-data.html

For example, say the following table consists of stock prices:

Table 1: Stock Prices (Non-Tidy Format)

Date	Boeing Stock Price	Amazon Stock Price	Google Stock Price
2009-01-01	$173.55	$174.90	$174.34
2009-01-02	$172.61	$171.42	$170.04

Although the data are neatly organised in a spreadsheet-type format, they are not in tidy format since there are three variables corresponding to three unique pieces of information (Date, Stock Name, and Stock Price), but there are not three columns. In tidy data format each variable should be its own column, as shown below. Notice that both tables present the same information, but in different formats.

Table 2: Stock Prices (Tidy Format)

Date	Stock Name	Stock Price
2009-01-01	Boeing	$173.55
2009-01-02	Boeing	$172.61
2009-01-01	Amazon	$174.90
2009-01-02	Amazon	$171.42
2009-01-01	Google	$174.34
2009-01-02	Google	$170.04

However, consider the following table:

Table 3: Date, Boeing Price, Weather Data

Date	Boeing Price	Weather
2009-01-01	$173.55	Sunny
2009-01-02	$172.61	Overcast

In this case, even though the variable **Boeing Price** occurs again, the data *is* tidy since there are three variables corresponding to three unique pieces of information (Date, Boeing stock price, and the weather on that particular day).

The non-tidy data format in the original table is also known as wide format whereas the tidy data format in the second table is also known as long/narrow data format. In this course, we will work mostly with data sets that are already in the tidy format.

Task: Consider the following data frame of average number of servings of beer, spirits, and wine consumption in three countries as reported in the FiveThirtyEight article Dear Mona Followup: Where Do People Drink The Most Beer, Wine And Spirits?

```
# A tibble: 3 x 4
  country     beer_servings spirit_servings wine_servings
  <chr>               <int>           <int>         <int>
1 Canada                240             122           100
2 South Korea           140              16             9
3 USA                   249             158            84
```

This data frame is not in tidy format. What would it look like if it were?

3 Observational units

Recall the `nycflights13` package with data about all domestic flights departing from New York City in 2013 that we used in Week 1 to create visualisations. In particular, let's revisit the `flights` data frame:

```
dim(flights)  # Returns the dimensions of a data frame (number obs. and variables)
```

```
[1] 336776     19
```

```
head(flights) # Returns the first 6 rows of the object
```

```
# A tibble: 6 x 19
   year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1  2013     1     1      517            515         2      830
2  2013     1     1      533            529         4      850
3  2013     1     1      542            540         2      923
4  2013     1     1      544            545        -1     1004
5  2013     1     1      554            600        -6      812
6  2013     1     1      554            558        -4      740
# ... with 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

```
glimpse(flights) # Lists the variables in an object with their first few values
```

```
Observations: 336,776
Variables: 19
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
$ month          <int> 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,...
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
```

```
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
```

We see that `flights` has a rectangular shape with each row corresponding to a different flight and each column corresponding to a characteristic of that flight. This matches exactly with the first two properties of tidy data, namely:

1. Each variable forms a column.
2. Each observation forms a row.

But what about the third property?

3. Each type of observational unit forms a table.

The observational unit in the `flights` data set is an individual flight and we can see above that this data set consists of 336,776 flights with 19 variables. In other words, rows of this data set don't refer to a measurement on an airline or on an airport; they refer to characteristics/measurements on a given flight from New York City in 2013. This illustrates the third property of tidy data, i.e. each observational unit is fully described by a single data set.

Note that there is only one observational unit of interest in any analysis. For example, also included in the `nycflights13` package are data sets with different observational units:

- `airlines`
- `planes`
- `weather`
- `airports`

The organisation of this data follows the third **tidy** data property: observations corresponding to the same observational unit are saved in the same data frame.

Task: For each of the data sets listed above (other than `flights`), identify the observational unit and how many of these are described in each of the data sets.

4 Identification vs measurement variables

There is a subtle difference between the kinds of variables that you will encounter in data frames: **measurement variables** and **identification variables**. The `airports` data frame contains both these types of variables. Recall that in `airports` the observational unit is an airport, and thus each row corresponds to one particular airport. Let's pull them apart using the `glimpse` function:

```r
glimpse(airports)
```

```
Observations: 1,458
Variables: 8
$ faa   <chr> "04G", "06A", "06C", "06N", "09J", "0A9", "0G6", "0G7", ...
$ name  <chr> "Lansdowne Airport", "Moton Field Municipal Airport", "S...
$ lat   <dbl> 41.13047, 32.46057, 41.98934, 41.43191, 31.07447, 36.371...
$ lon   <dbl> -80.61958, -85.68003, -88.10124, -74.39156, -81.42778, -...
$ alt   <int> 1044, 264, 801, 523, 11, 1593, 730, 492, 1000, 108, 409,...
```

```
$ tz    <dbl> -5, -6, -6, -5, -5, -5, -5, -5, -5, -8, -5, -6, -5, -5, ...
$ dst   <chr> "A", "A", "A", "A", "A", "A", "A", "A", "U", "A", "A", "...
$ tzone <chr> "America/New_York", "America/Chicago", "America/Chicago"...
```

The variables `faa` and `name` are what we will call **identification variables**: variables that uniquely identify each observational unit. They are mainly used to provide a unique name to each observational unit, thereby allowing us to uniquely identify them. `faa` gives the unique code provided by the Federal Aviation Administration in the USA for that airport, while the `name` variable gives the longer more natural name of the airport. The remaining variables (`lat`, `lon`, `alt`, `tz`, `dst`, `tzone`) are often called **measurement** or **characteristic** variables: variables that describe properties of each observational unit, in other words each observation in each row. For example, `lat` and `long` describe the latitude and longitude of each airport.

Furthermore, sometimes a single variable might not be enough to uniquely identify each observational unit: combinations of variables might be needed (see **Task** below). While it is not an absolute rule, for organisational purposes it is considered good practice to have your identification variables in the far left-most columns of your data frame.

Task: What properties of the observational unit do each of `lat`, `lon`, `alt`, `tz`, `dst`, and `tzone` describe for the `airports` data frame?

Task: From the data sets listed above, find an example where combinations of variables are needed to uniquely identify each observational unit.

5 Importing spreadsheets into R

Up to this point, we have been using data stored inside of an R package. In the real world, your data will usually come from a spreadsheet file either on your computer or online. Spreadsheet data is often saved in one of two formats:

- A **Comma Separated Values** `.csv` file. You can think of a CSV file as a bare-bones spreadsheet where:
  - Each line in the file corresponds to one row of data/one observation.
  - Values for each line are separated with commas. In other words, the values of different variables are separated by commas.
  - The first line is often, but not always, a *header* row indicating the names of the columns/variables.
- An **Excel** `.xlsx` file. This format is based on Microsoft's proprietary Excel software. As opposed to bare-bones `.csv` files, `.xlsx` Excel files contain a lot of *metadata*, i.e. data about the data. Examples include the use of bold and italic fonts, colored cells, different column widths, and formula macros etc.

We'll cover two methods for importing data in R: one using the R console and the other using RStudio's graphical interface.

5.1 Method 1: From the console

First, let's download a **Comma Separated Values** (CSV) file of ratings of the level of democracy in different countries spanning 1952 to 1992: https://moderndive.com/data/dem_score.csv. We use the `read_csv()` function from the `readr` package to read it off the web:

```
dem_score <- read_csv("https://moderndive.com/data/dem_score.csv")
```

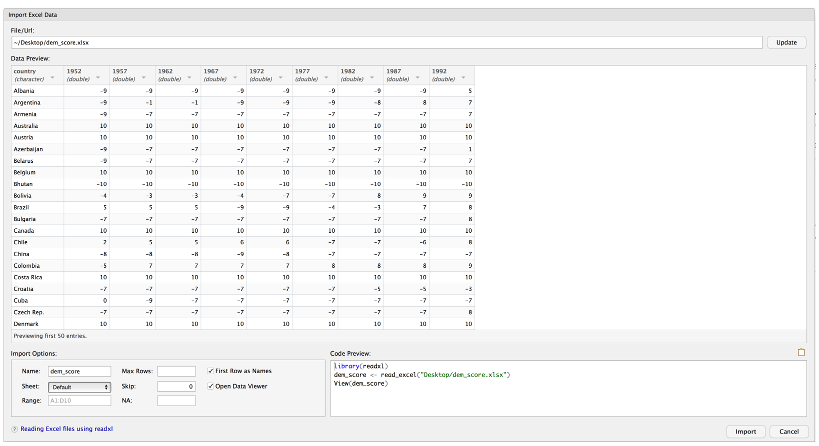
```
# A tibble: 96 x 10
   country  `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
   <chr>     <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
 1 Albania      -9     -9     -9     -9     -9     -9     -9     -9      5
 2 Argenti~     -9     -1     -1     -9     -9     -9     -8      8      7
 3 Armenia      -9     -7     -7     -7     -7     -7     -7     -7      7
```

```
 4 Austral~     10     10     10     10     10     10     10     10     10
 5 Austria      10     10     10     10     10     10     10     10     10
 6 Azerbai~     -9     -7     -7     -7     -7     -7     -7     -7      1
 7 Belarus      -9     -7     -7     -7     -7     -7     -7     -7      7
 8 Belgium      10     10     10     10     10     10     10     10     10
 9 Bhutan      -10    -10    -10    -10    -10    -10    -10    -10    -10
10 Bolivia      -4     -3     -3     -4     -7     -7      8      9      9
# ... with 86 more rows
```

In this `dem_score` data frame, the minimum value of -10 corresponds to a highly autocratic nation whereas a value of 10 corresponds to a highly democratic nation.

5.2 Method 2: Using RStudio's interface

Let's read in the same data saved in Excel format this time at https://moderndive.com/data/dem_score.xlsx, but using RStudio's graphical interface instead of via the R console. First download the Excel file, then go to the `Files -> Import Dataset -> From Excel...` and navigate to the directory where your downloaded `dem_score.xlsx` using `Browse...`. You should see something similar to the image below:



After clicking on the **Import** button on the bottom-right save this spreadsheet's data in a data frame called `dem_score` and display its contents in the spreadsheet viewer (`View()`). Furthermore you'll see the code that read in your data in the console; you can copy and paste this code to reload your data again later instead of repeating the above manual process.

Task: Read in the life expectancy data stored at https://moderndive.com/data/le_mess.csv, either using the R console or RStudio's interface.

6 Converting to tidy data format

In this section, we will see how to convert a data set that is not in the **tidy** format i.e. wide format, to a data set that is in the **tidy** format i.e. long/narrow format. Let's use the `dem_score` data frame we loaded from a spreadsheet in the previous section but focus on only data corresponding to the country of Guatemala.

```
guat_dem <- dem_score %>%
  filter(country == "Guatemala")
```

```
# A tibble: 1 x 10
  country   `1952` `1957` `1962` `1967` `1972` `1977` `1982` `1987` `1992`
  <chr>      <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>  <int>
```

```
1 Guatemala       2      -6      -5       3       1      -3      -7       3       3
```

Note: We will revisit this code for subsetting data later in the session.

Now let's produce a plot showing how the democracy scores have changed over the 40 years from 1952 to 1992 for Guatemala. Let's start by laying out how we would map our aesthetics to variables in the data frame:

- The `data` frame is `guat_dem` so we use `data = guat_dem`.

We would like to see how the democracy score has changed over the years in Guatemala. But we have a problem. We see that we have a variable named `country` but its only value is `Guatemala`. We have other variables denoted by different year values. Unfortunately, we've run into a data set that is not in the appropriate format to apply the **Grammar of Graphics** in `ggplot2`. Remember that `ggplot2` is a package in the `tidyverse` and, thus, needs data to be in a tidy format. We'd like to finish off our mapping of aesthetics to variables by doing something like

- The `aes`thetic mapping is set by `aes(x = year, y = democracy_score)`,

but this is not possible with our wide-formatted data. We need to take the values of the current column names in `guat_dem` (aside from `country`) and convert them into a new variable that will act as a key called `year`. Then, we'd like to take the numbers on the inside of the table and turn them into a column that will act as values called `democracy_score`. Our resulting data frame will have three columns: `country`, `year`, and `democracy_score`.

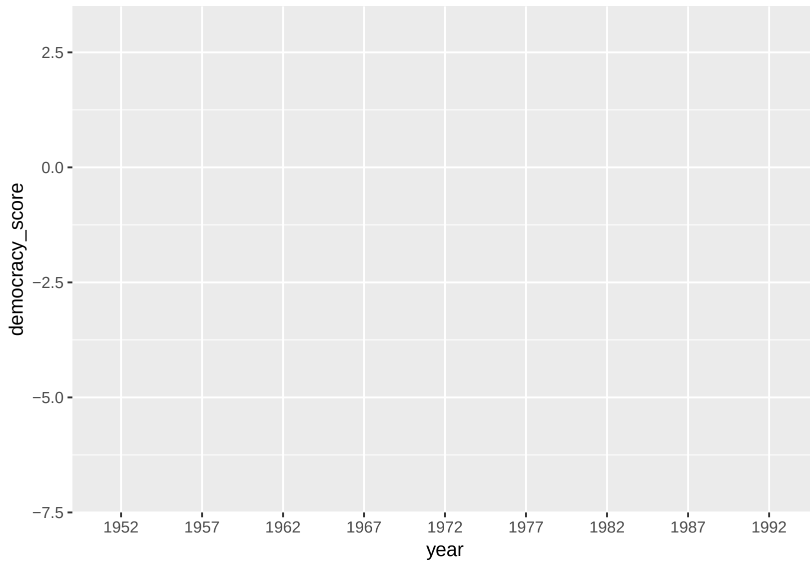
The `gather` function in the `tidyr` package can complete this task for us. The first argument to `gather`, just as with `ggplot2`, is the `data` argument where we specify which data frame we would like to tidy. The next two arguments to `gather` are `key` and `value`, which specify what we would like to call the new columns that convert our wide data into tidy/long format. Lastly, we include a specification for variables we would like to NOT include in the tidying process using a `-`.

```
guat_tidy <- gather(data = guat_dem,
                    key = year,
                    value = democracy_score,
                    - country)
```

```
# A tibble: 9 x 3
  country   year  democracy_score
  <chr>     <chr>           <int>
1 Guatemala 1952                2
2 Guatemala 1957               -6
3 Guatemala 1962               -5
4 Guatemala 1967                3
5 Guatemala 1972                1
6 Guatemala 1977               -3
7 Guatemala 1982               -7
8 Guatemala 1987                3
9 Guatemala 1992                3
```

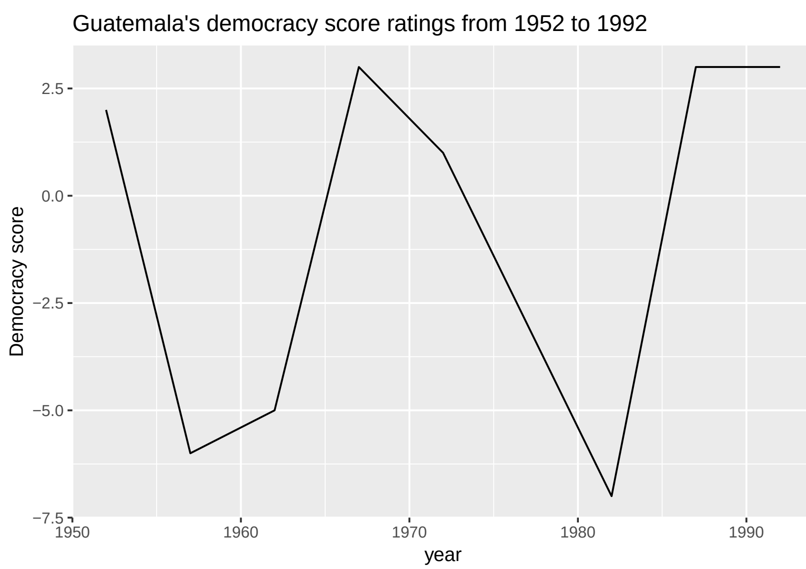
We can now create a plot showing how democracy score in Guatemala has changed from 1952 to 1992 using a linegraph and `ggplot2`.

```
ggplot(data = guat_tidy, mapping = aes(x = year, y = democracy_score)) +
  geom_line() +
  labs(x = "year")
```



Observe that the `year` variable in `guat_tidy` is stored as a character vector since we had to circumvent the naming rules in R by adding backticks around the different year columns in `guat_dem`. This is leading to `ggplot` not knowing exactly how to plot a line using a categorical variable. We can fix this by using the `parse_number` function in the `readr` package:

```
ggplot(data = guat_tidy, mapping = aes(x = parse_number(year), y = democracy_score)) +
  geom_line() +
  labs(x = "year", y = "Democracy score",
       title = "Guatemala's democracy score ratings from 1952 to 1992")
```



We'll see later how we could use the `mutate` function to change `year` to be a numeric variable during the tidying process. Notice now that the mappings of aesthetics to variables makes sense in the figure:

- The `data` frame is `guat_tidy` by setting `data = guat_tidy`;
- The `x` `aes`thetic is mapped to `year`;
- The `y` `aes`thetic is mapped to `democracy_score`; and
- The `geom_`etry chosen is `line`.

Task: Convert the `dem_score` data frame into a tidy data frame and assign the name of `dem_score_tidy` to the resulting long-formatted data frame.

Task: Convert the life expectancy data set you created in a previous task into a tidy data frame.

7 Introduction to data wrangling

We are now able to import data and perform basic operations on the data to get it into the **tidy** format. In this and subsequent sections we will use tools from the `dplyr` package to perform data **wrangling** which includes transforming, mapping and summarising variables.

7.1 The pipe `%>%`

Before we dig into data wrangling, let's first introduce the pipe operator (`%>%`). Just as the `+` sign was used to add layers to a plot created using `ggplot`, the pipe operator allows us to chain together `dplyr` data wrangling functions. The pipe operator can be read as **then**. The `%>%` operator allows us to go from one step in `dplyr` to the next easily so we can, for example:

- `filter` our data frame to only focus on a few rows **then**
- `group_by` another variable to create groups **then**
- `summarize` this grouped data to calculate the mean for each level of the group.

The piping syntax will be our major focus throughout the rest of this course and you'll find that you'll quickly be addicted to the chaining with some practice.

7.2 Data wrangling verbs

The `d` in `dplyr` stands for data frames, so the functions in `dplyr` are built for working with objects of the data frame type. For now, we focus on the most commonly used functions that help wrangle and summarise data. A description of these verbs follows, with each subsequent section devoted to an example of that verb, or a combination of a few verbs, in action.

1. `filter`: Pick rows based on conditions about their values
2. `summarize`: Compute summary measures known as "summary statistics" of variables
3. `group_by`: Group rows of observations together
4. `mutate`: Create a new variable in the data frame by mutating existing ones
5. `arrange`: Arrange/sort the rows based on one or more variables
6. `join`: Join/merge two data frames by matching along a "key" variable. There are many different `join`s available. Here, we will focus on the `inner_join` function.

All of the verbs are used similarly where you: take a data frame, pipe it using the `%>%` syntax into one of the verbs above followed by other arguments specifying which criteria you would like the verb to work with in parentheses.

8 Filter observations using filter

The `filter` function allows you to specify criteria about values of a variable in your data set and then chooses only those rows that match that criteria. We begin by focusing only on flights from New York City to Portland, Oregon. The `dest` code (or airport code) for Portland, Oregon is `PDX`. Run the following code and look at the resulting spreadsheet to ensure that only flights heading to Portland are chosen:

```
portland_flights <- flights %>%
  filter(dest == "PDX")
portland_flights[,-(6:12)]
```

```
# A tibble: 1,354 x 12
    year month   day dep_time sched_dep_time origin dest  air_time distance
   <int> <int> <int>    <int>          <int> <chr>  <chr>    <dbl>    <dbl>
 1  2013     1     1     1739           1740 JFK    PDX        341     2454
 2  2013     1     1     1805           1757 EWR    PDX        336     2434
 3  2013     1     1     2052           2029 JFK    PDX        331     2454
 4  2013     1     2      804            805 EWR    PDX        310     2434
 5  2013     1     2     1552           1550 JFK    PDX        305     2454
 6  2013     1     2     1727           1720 EWR    PDX        351     2434
 7  2013     1     2     1738           1740 JFK    PDX        322     2454
 8  2013     1     2     2024           2029 JFK    PDX        325     2454
 9  2013     1     3     1755           1745 JFK    PDX        325     2454
10  2013     1     3     1814           1727 EWR    PDX        320     2434
# ... with 1,344 more rows, and 3 more variables: hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

```
# We do not display columns 6-11 so we can see the destination (dest) variable.
```

Note the following:

- The ordering of the commands:
  - Take the data frame `flights` **then**
  - `filter` the data frame so that only those where the `dest` equals `PDX` are included.
- The double equals sign `==` tests equality, and not a single equals sign `=`.

You can combine multiple criteria together using operators that make comparisons:

- `|` corresponds to **or**
- `&` corresponds to **and**

We can often skip the use of `&` and just separate our conditions with a comma. You'll see this in the example below.

In addition, you can use other mathematical checks (similar to `==`):

- `>` corresponds to **greater than**
- `<` corresponds to **less than**
- `>=` corresponds to **greater than or equal to**
- `<=` corresponds to **less than or equal to**
- `!=` corresponds to **not equal to**

To see many of these in action, let's select all flights that left JFK airport heading to Burlington, Vermont (`BTV`) or Seattle, Washington (`SEA`) in the months of October, November, or December. Run the following

```
btv_sea_flights_fall <- flights %>%
  filter(origin == "JFK", (dest == "BTV" | dest == "SEA"), month >= 10)
btv_sea_flights_fall[,-(6:12)]
```

```
# A tibble: 815 x 12
    year month   day dep_time sched_dep_time origin dest  air_time distance
   <int> <int> <int>    <int>          <int> <chr>  <chr>    <dbl>    <dbl>
```

```
 1  2013    10     1      729            735 JFK    SEA          352      2422
 2  2013    10     1      853            900 JFK    SEA          362      2422
 3  2013    10     1      916            925 JFK    BTV           48       266
 4  2013    10     1     1216           1221 JFK    BTV           49       266
 5  2013    10     1     1452           1459 JFK    BTV           46       266
 6  2013    10     1     1459           1500 JFK    SEA          348      2422
 7  2013    10     1     1754           1800 JFK    SEA          338      2422
 8  2013    10     1     1825           1830 JFK    SEA          366      2422
 9  2013    10     1     1925           1930 JFK    SEA          332      2422
10  2013    10     1     2238           2245 JFK    BTV           48       266
# ... with 805 more rows, and 3 more variables: hour <dbl>, minute <dbl>,
#   time_hour <dttm>
```

```
# We do not display columns 6-11 so we can see the "origin" and "dest" variables.
```

Note: even though colloquially speaking one might say "all flights leaving Burlington, Vermont *and* Seattle, Washington," in terms of computer logical operations, we really mean "all flights leaving Burlington, Vermont *or* Seattle, Washington." For a given row in the data, `dest` can be `BTV`, `SEA`, or something else, but not `BTV` **and** `SEA` at the same time.

Another example uses `!` to pick rows that *do not* match a condition. The `!` can be read as **not**. Here, we are selecting rows corresponding to flights that **did not** go to Burlington, VT or Seattle, WA.

```
not_BTV_SEA <- flights %>%
  filter(!(dest == "BTV" | dest == "SEA"))
not_BTV_SEA[,-(6:12)]
```

```
# A tibble: 330,264 x 12
    year month   day dep_time sched_dep_time origin dest  air_time distance
   <int> <int> <int>    <int>          <int> <chr>  <chr>    <dbl>    <dbl>
 1  2013     1     1      517            515 EWR    IAH        227     1400
 2  2013     1     1      533            529 LGA    IAH        227     1416
 3  2013     1     1      542            540 JFK    MIA        160     1089
 4  2013     1     1      544            545 JFK    BQN        183     1576
 5  2013     1     1      554            600 LGA    ATL        116      762
 6  2013     1     1      554            558 EWR    ORD        150      719
 7  2013     1     1      555            600 EWR    FLL        158     1065
 8  2013     1     1      557            600 LGA    IAD         53      229
 9  2013     1     1      557            600 JFK    MCO        140      944
10  2013     1     1      558            600 LGA    ORD        138      733
# ... with 330,254 more rows, and 3 more variables: hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

```
# We do not display columns 6-11 so we can see the "origin" and "dest" variables.
```

As a final note we point out that `filter` should often be the first verb you'll apply to your data. This narrows down the data to just the observations your are interested in.

Task: What is another way of using the **not** operator `!` to filter only the rows that are not going to Burlington, VT nor Seattle, WA in the `flights` data frame?

9 Summarise variables using summarize

The next common task is to be able to summarise data: take a large number of values and summarise them with a single value. While this may seem like a very abstract idea, something as simple as the sum, the smallest value, and the largest values are all summaries of a large number of values.

We can calculate the standard deviation and mean of the temperature variable `temp` in the `weather` data frame of `nycflights13` in one step using the `summarize` (or equivalently using the UK spelling `summarise`) function in `dplyr`

```
summary_temp <- weather %>%
  summarize(mean = mean(temp), std_dev = sd(temp))
```

mean	std_dev
NA	NA

We have created a small data frame here called `summary_temp` that includes both the mean (`mean`) and standard deviation (`std_dev`) of the `temp` variable in `weather`. Notice, the data frame `weather` went from many rows to a single row of just the summary values in the data frame `summary_temp`.

But why are the values returned `NA`? This stands for **not available or not applicable** and is how R encodes **missing values**; if in a data frame for a particular row and column no value exists, `NA` is stored instead. Furthermore, by default any time you try to summarise a number of values (using `mean()` and `sd()` for example) that has one or more missing values, then `NA` is returned.

Values can be missing for many reasons. Perhaps the data was collected but someone forgot to enter it? Perhaps the data was not collected at all because it was too difficult? Perhaps there was an erroneous value that someone entered that was changed to read as missing? You'll often encounter issues with missing values.

You can summarise all non-missing values by setting the `na.rm` argument to TRUE (`rm` is short for remove). This will remove any `NA` missing values and only return the summary value for all non-missing values. So the code below computes the mean and standard deviation of all non-missing values. Notice how the `na.rm=TRUE` are set as arguments to the `mean` and `sd` functions, and not to the `summarize` function.

```
summary_temp <- weather %>%
  summarize(mean = mean(temp, na.rm = TRUE), std_dev = sd(temp, na.rm = TRUE))
```

mean	std_dev
55.26039	17.78785

It is not good practice to include `na.rm = TRUE` in your summary commands by default; you should attempt to run code first without this argument as this will alert you to the presence of missing data. Only after you have identified where missing values occur and have thought about the potential issues of these should you consider using `na.rm = TRUE`. In the upcoming Tasks we will consider the possible ramifications of blindly sweeping rows with missing values under the rug.

What other summary functions can we use inside the `summarize` verb? Any function in R that takes a vector of values and returns just one. Here are just a few:

- `mean`: the mean (or average)
- `sd`: the standard deviation, which is a measure of spread
- `min` and `max`: the minimum and maximum values, respectively
- `IQR`: the interquartile range
- `sum`: the sum

- `n`: a count of the number of rows/observations in each group. This particular summary function will make more sense when `group_by` is used in the next section.

Task: Say a doctor is studying the effect of smoking on lung cancer for a large number of patients who have records measured at five year intervals. She notices that a large number of patients have missing data points because the patient has died, so she chooses to ignore these patients in her analysis. What is wrong with this doctor's approach?

Task: Modify `summary_temp` from above to also use the `n` summary function: `summarize(count = n())`. What does the returned value correspond to?

Task: Why does the code below not work? Run the code line by line instead of all at once, and then look at the data. In other words, run `summary_temp <- weather %>% summarize(mean = mean(temp, na.rm = TRUE))` first.

```
summary_temp <- weather %>%
  summarize(mean = mean(temp, na.rm = TRUE)) %>%
  summarize(std_dev = sd(temp, na.rm = TRUE))
```

10 Group rows using group_by

It is often more useful to summarise a variable based on the groupings of another variable. Let's say we are interested in the mean and standard deviation of temperatures but *grouped by month*. To be more specific: we want the mean and standard deviation of temperatures

1. split by month.
2. sliced by month.
3. aggregated by month.
4. collapsed over month.

Run the following code:

```
summary_monthly_temp <- weather %>%
  group_by(month) %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))
```

This code is identical to the previous code that created `summary_temp`, with an extra `group_by(month)` added. Grouping the `weather` data set by `month` and then passing this new data frame into `summarize` yields a data frame that shows the mean and standard deviation of temperature for each month in New York City. Note, since each row in `summary_monthly_temp` represents a summary of different rows in `weather`, the observational units have changed.

It is important to note that `group_by` doesn't change the data frame. It sets *meta-data* (data about the data), specifically the group structure of the data. It is only after we apply the `summarize` function that the data frame changes.

If we would like to remove this group structure meta-data, we can pipe the resulting data frame into the `ungroup` function. For example, say the group structure meta-data is set to be by month via `group_by(month)`, all future summarisations will be reported on a month-by-month basis. If however, we would like to no longer have this and have all summarisations be for all data in a single group (in this case over the entire year of 2013), then pipe the data frame in question through `ungroup` to remove this.

```
summary_monthly_temp <- weather %>%
  group_by(month) %>%
  ungroup() %>%
  summarize(mean = mean(temp, na.rm = TRUE),
            std_dev = sd(temp, na.rm = TRUE))
```

mean	std_dev
55.26039	17.78785

We now revisit the `n` counting summary function we introduced in the previous section. For example, suppose we would like to get a sense for how many flights departed each of the three airports in New York City:

```
by_origin <- flights %>%
  group_by(origin) %>%
  summarize(count = n())
```

origin	count
EWR	120835
JFK	111279
LGA	104662

We see that Newark (`EWR`) had the most flights departing in 2013 followed by `JFK` and lastly by LaGuardia (`LGA`). Note, there is a subtle but important difference between `sum` and `n`. While `sum` simply adds up a large set of numbers, the latter counts the number of times each of many different values occur.

10.1 Grouping by more than one variable

You are not limited to grouping by one variable. Say you wanted to know the number of flights leaving each of the three New York City airports *for each month*, we can also group by a second variable `month`: `group_by(origin, month)`.

```
by_origin_monthly <- flights %>%
  group_by(origin, month) %>%
  summarize(count = n())
```

```
# A tibble: 36 x 3
# Groups:   origin [?]
   origin month count
   <chr>  <int> <int>
 1 EWR        1  9893
 2 EWR        2  9107
 3 EWR        3 10420
 4 EWR        4 10531
 5 EWR        5 10592
 6 EWR        6 10175
 7 EWR        7 10475
 8 EWR        8 10359
 9 EWR        9  9550
10 EWR       10 10104
# ... with 26 more rows
```

We see there are 36 rows for `by_origin_monthly` because there are 12 months times 3 airports (`EWR`, `JFK`, and `LGA`). Let's now pose two questions. First, what if we reverse the order of the grouping, i.e. `group_by(month, origin)`?

```
by_monthly_origin <- flights %>%
  group_by(month, origin) %>%
  summarize(count = n())
```

```
# A tibble: 36 x 3
# Groups:   month [?]
   month origin count
   <int> <chr>  <int>
 1     1 EWR     9893
 2     1 JFK     9161
 3     1 LGA     7950
 4     2 EWR     9107
 5     2 JFK     8421
 6     2 LGA     7423
 7     3 EWR    10420
 8     3 JFK     9697
 9     3 LGA     8717
10     4 EWR    10531
# ... with 26 more rows
```

In `by_monthly_origin` the `month` column is now first and the rows are sorted by `month` instead of origin. If you compare the values of `count` in `by_origin_monthly` and `by_monthly_origin` using the `View` function, you'll see that the values are actually the same, just presented in a different order.

Second, why do we `group_by(origin, month)` and not `group_by(origin)` and then `group_by(month)`? Let's investigate:

```
by_origin_monthly_incorrect <- flights %>%
  group_by(origin) %>%
  group_by(month) %>%
  summarize(count = n())
```

```
# A tibble: 12 x 2
   month count
   <int> <int>
 1     1 27004
 2     2 24951
 3     3 28834
 4     4 28330
 5     5 28796
 6     6 28243
 7     7 29425
 8     8 29327
 9     9 27574
10    10 28889
11    11 27268
12    12 28135
```

What happened here is that the second `group_by(month)` overrode the first `group_by(origin)`, so that in the end we are only grouping by `month`. The lesson here, is if you want to `group_by` two or more variables, you should include all these variables in a single `group_by` function call.

Task: Recall from Week 1 when we looked at plots of temperatures by months in NYC. What does the standard deviation column in the `summary_monthly_temp` data frame tell us about temperatures in New

York City throughout the year?

Task: Write code to produce the mean and standard deviation temperature for each day in 2013 for NYC?

Task: Recreate `by_monthly_origin`, but instead of grouping via `group_by(origin, month)`, group variables in a different order `group_by(month, origin)`. What differs in the resulting data set?

Task: How could we identify how many flights left each of the three airports for each `carrier`?

Task: How does the `filter` operation differ from a `group_by` followed by a `summarize`?

11 Create new variables/change old variables using mutate

When looking at the `flights` data set, there are some clear additional variables that could be calculated based on the values of variables already in the data set. Passengers are often frustrated when their flights depart late, but change their mood a bit if pilots can make up some time during the flight to get them to their destination close to when they expected to land. This is commonly referred to as "gain" and we will create this variable using the `mutate` function. Note that we will be overwriting the `flights` data frame with one including the additional variable `gain` here, or put differently, the `mutate` command outputs a new data frame which then gets saved over the original `flights` data frame.

```
flights <- flights %>%
  mutate(gain = dep_delay - arr_delay)
```

Let's take a look at `dep_delay`, `arr_delay`, and the resulting `gain` variables in our new `flights` data frame:

```
# A tibble: 336,776 x 3
   dep_delay arr_delay  gain
       <dbl>     <dbl> <dbl>
 1         2        11    -9
 2         4        20   -16
 3         2        33   -31
 4        -1       -18    17
 5        -6       -25    19
 6        -4        12   -16
 7        -5        19   -24
 8        -3       -14    11
 9        -3        -8     5
10        -2         8   -10
# ... with 336,766 more rows
```

The flight in the first row departed 2 minutes late but arrived 11 minutes late, so its "gained time in the air" is actually a loss of 9 minutes, hence its `gain` is `-9`. Contrast this to the flight in the fourth row which departed a minute early (`dep_delay` of `-1`) but arrived 18 minutes early (`arr_delay` of `-18`), so its "gained time in the air" is 17 minutes, hence its `gain` is `+17`.

Why did we overwrite `flights` instead of assigning the resulting data frame to a new object, like `flights_with_gain`? As a rough rule of thumb, as long as you are not losing information that you might need later, it's acceptable practice to overwrite data frames. However, if you overwrite existing variables and/or change the observational units, recovering the original information might prove difficult. In this case, it might make sense to create a new data object.

Let's look at summary measures of this `gain` variable and plot it in the form of a histogram:

```
gain_summary <- flights %>%
  summarize(
    min = min(gain, na.rm = TRUE),
    q1 = quantile(gain, 0.25, na.rm = TRUE),
    median = quantile(gain, 0.5, na.rm = TRUE),
    q3 = quantile(gain, 0.75, na.rm = TRUE),
    max = max(gain, na.rm = TRUE),
    mean = mean(gain, na.rm = TRUE),
    sd = sd(gain, na.rm = TRUE),
    missing = sum(is.na(gain))
```

min	q1	median	q3	max	mean	sd	missing
-196	-3	7	17	109	5.659779	18.04365	9430

We have recreated the `summary` function we saw in Week 1 here using the `summarize` function in `dplyr`.

```
ggplot(data = flights, mapping = aes(x = gain)) +
  geom_histogram(color = "white", bins = 20)
```

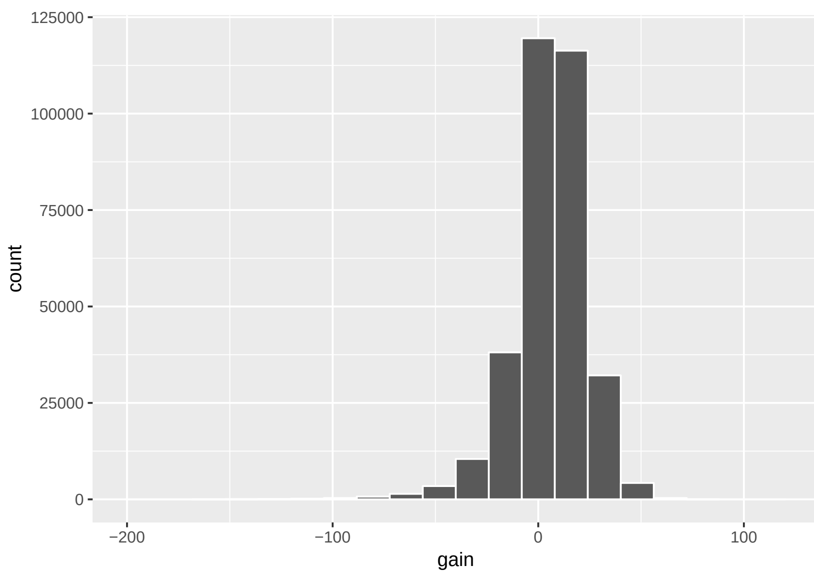


Figure 2: Histogram of gain variable.

We can also create multiple columns at once and even refer to columns that were just created in a new column.

```
flights <- flights %>%
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours
  )
```

Task: What do positive values of the gain variable in flights correspond to? What about negative values? And what about a zero value?

Task: Could we create the dep_delay and arr_delay columns by simply subtracting dep_time from sched_dep_time and similarly for arrivals? Try the code out and explain any differences between the result and what actually appears in flights.

Task: What can we say about the distribution of gain? Describe it in a few sentences using the plot and the gain_summary data frame values.

12 Reorder the data frame using arrange

One of the most common things people working with data would like to do is sort the data frames by a specific variable in a column. Have you ever been asked to calculate a median by hand? This requires you to put the data in order from smallest to highest in value. The dplyr package has a function called arrange that we will use to sort/reorder our data according to the values of the specified variable. This is often used after we have used the group_by and summarize functions as we will see.

Let's suppose we were interested in determining the most frequent destination airports from New York City in 2013:

```
freq_dest <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n())
```

```
# A tibble: 105 x 2
   dest  num_flights
   <chr>       <int>
 1 ABQ           254
 2 ACK           265
 3 ALB           439
 4 ANC             8
 5 ATL         17215
 6 AUS          2439
 7 AVL           275
 8 BDL           443
 9 BGR           375
10 BHM           297
# ... with 95 more rows
```

You'll see that by default the values of dest are displayed in alphabetical order here. We are interested in finding those airports that appear most:

```
freq_dest %>%
  arrange(num_flights)
```

```
# A tibble: 105 x 2
   dest  num_flights
   <chr>       <int>
 1 LEX             1
 2 LGA             1
 3 ANC             8
 4 SBN            10
 5 HDN            15
 6 MTJ            15
 7 EYW            17
```

```
 8 PSP            19
 9 JAC            25
10 BZN            36
# ... with 95 more rows
```

This is actually giving us the opposite of what we are looking for. It tells us the least frequent destination airports first. To switch the ordering to be descending instead of ascending we use the `desc` (`desc`ending) function:

```r
freq_dest %>%
  arrange(desc(num_flights))
```

```
# A tibble: 105 x 2
   dest  num_flights
   <chr>       <int>
 1 ORD         17283
 2 ATL         17215
 3 LAX         16174
 4 BOS         15508
 5 MCO         14082
 6 CLT         14064
 7 SFO         13331
 8 FLL         12055
 9 MIA         11728
10 DCA          9705
# ... with 95 more rows
```

13 Joining data frames

Another common task is joining (merging) two different data sets. For example, in the `flights` data, the variable `carrier` lists the carrier code for the different flights. While `UA` and `AA` might be somewhat easy to guess for some (United and American Airlines), what are VX, HA, and B6? This information is provided in a separate data frame `airlines`.

```r
airlines
```

```
# A tibble: 16 x 2
   carrier name
   <chr>   <chr>
 1 9E      Endeavor Air Inc.
 2 AA      American Airlines Inc.
 3 AS      Alaska Airlines Inc.
 4 B6      JetBlue Airways
 5 DL      Delta Air Lines Inc.
 6 EV      ExpressJet Airlines Inc.
 7 F9      Frontier Airlines Inc.
 8 FL      AirTran Airways Corporation
 9 HA      Hawaiian Airlines Inc.
10 MQ      Envoy Air
11 OO      SkyWest Airlines Inc.
12 UA      United Air Lines Inc.
13 US      US Airways Inc.
14 VX      Virgin America
15 WN      Southwest Airlines Co.
16 YV      Mesa Airlines Inc.
```

We see that in `airports`, `carrier` is the carrier code while `name` is the full name of the airline. Using this table, we can see that VX, HA, and B6 correspond to Virgin America, Hawaiian Airlines Inc., and JetBlue Airways, respectively. However, will we have to continually look up the carrier's name for each flight in the `airlines` data set? No! Instead of having to do this manually, we can have R automatically do the "looking up" for us.

Note that the values in the variable `carrier` in `flights` match the values in the variable `carrier` in `airlines`. In this case, we can use the variable `carrier` as a *key variable* to join/merge/match the two data frames by. Key variables are almost always identification variables that uniquely identify the observational units as we saw back in the **Identification vs Measurement Variable** section. This ensures that rows in both data frames are appropriately matched during the join. This diagram helps us understand how the different data sets are linked by various key variables:

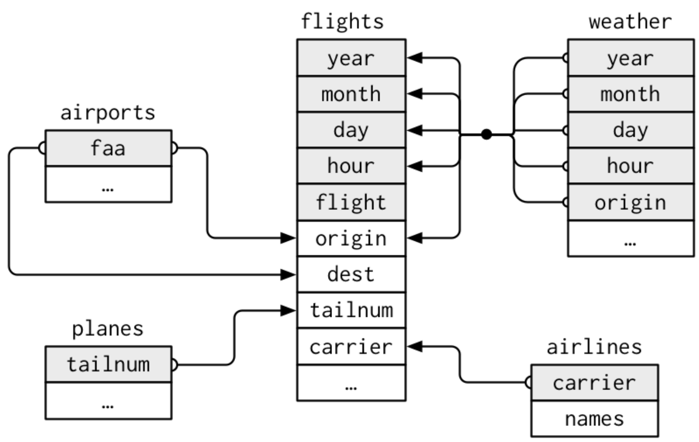


Figure 3: Data relationships in nycflights13 from R for Data Science, Hadley and Garrett (2016).

13.1 Joining by "key" variables

In both `flights` and `airlines`, the key variable we want to join/merge/match the two data frames with has the same name in both data sets: `carriers`. We make use of the `inner_join` function to join by the variable `carrier`.

```
flights_joined <- flights %>%
  inner_join(airlines, by = "carrier")
flights
```

```
# A tibble: 336,776 x 22
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 336,766 more rows, and 15 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
```

```
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>, gain <dbl>, hours <dbl>,
#   gain_per_hour <dbl>
```

```
flights_joined
```

```
# A tibble: 336,776 x 23
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 336,766 more rows, and 16 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>, gain <dbl>, hours <dbl>,
#   gain_per_hour <dbl>, name <chr>
```

We observe that the `flights` and `flights_joined` are identical except that `flights_joined` has an additional variable `name` whose values were drawn from `airlines`.

A visual representation of the `inner_join` is given below:

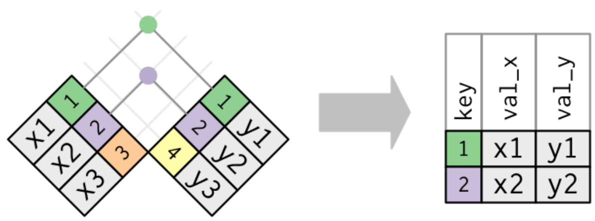


Figure 4: Diagram of inner join from R for Data Science.

There are more complex joins available, but the `inner_join` will solve nearly all of the problems you will face here.

13.2 Joining by "key" variables with different names

Say instead, you are interested in all the destinations of flights from NYC in 2013 and ask yourself:

- "What cities are these airports in?"
- "Is `ORD` Orlando?"
- "Where is `FLL`?"

The `airports` data frame contains airport codes:

```
airports
```

```
# A tibble: 1,458 x 8
   faa   name                       lat    lon   alt    tz dst   tzone
```

```
   <chr> <chr>                <dbl>  <dbl> <int> <dbl> <chr> <chr>      
 1 04G   Lansdowne Airport     41.1  -80.6  1044    -5 A     America/New_~
 2 06A   Moton Field Municip~  32.5  -85.7   264    -6 A     America/Chic~
 3 06C   Schaumburg Regional   42.0  -88.1   801    -6 A     America/Chic~
 4 06N   Randall Airport       41.4  -74.4   523    -5 A     America/New_~
 5 09J   Jekyll Island Airpo~  31.1  -81.4    11    -5 A     America/New_~
 6 0A9   Elizabethton Munici~  36.4  -82.2  1593    -5 A     America/New_~
 7 0G6   Williams County Air~  41.5  -84.5   730    -5 A     America/New_~
 8 0G7   Finger Lakes Region~  42.9  -76.8   492    -5 A     America/New_~
 9 0P2   Shoestring Aviation~  39.8  -76.6  1000    -5 U     America/New_~
10 0S9   Jefferson County In~  48.1 -123.    108    -8 A     America/Los_~
# ... with 1,448 more rows
```

However, looking at both the `airports` and `flights` and the visual representation of the relations between the data frames in the figure above, we see that in:

- `airports` the airport code is in the variable `faa`
- `flights` the airport code is in the variable `origin`

So to join these two data sets, our `inner_join` operation involves a `by` argument that accounts for the different names:

```
flights %>% 
  inner_join(airports, by = c("dest" = "faa"))
```

Let's construct the sequence of commands that computes the number of flights from NYC to each destination, but also includes information about each destination airport:

```
named_dests <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n()) %>%
  arrange(desc(num_flights)) %>%
  inner_join(airports, by = c("dest" = "faa")) %>%
  rename(airport_name = name)
```

```
# A tibble: 101 x 9
   dest  num_flights airport_name      lat    lon   alt    tz dst   tzone
   <chr>       <int> <chr>           <dbl>  <dbl> <int> <dbl> <chr> <chr>
 1 ORD         17283 Chicago Ohare ~  42.0  -87.9   668    -6 A     Ameri~
 2 ATL         17215 Hartsfield Jac~  33.6  -84.4  1026    -5 A     Ameri~
 3 LAX         16174 Los Angeles In~  33.9 -118.    126    -8 A     Ameri~
 4 BOS         15508 General Edward~  42.4  -71.0    19    -5 A     Ameri~
 5 MCO         14082 Orlando Intl     28.4  -81.3    96    -5 A     Ameri~
 6 CLT         14064 Charlotte Doug~  35.2  -80.9   748    -5 A     Ameri~
 7 SFO         13331 San Francisco ~  37.6 -122.     13    -8 A     Ameri~
 8 FLL         12055 Fort Lauderdal~  26.1  -80.2     9    -5 A     Ameri~
 9 MIA         11728 Miami Intl       25.8  -80.3     8    -5 A     Ameri~
10 DCA          9705 Ronald Reagan ~  38.9  -77.0    15    -5 A     Ameri~
# ... with 91 more rows
```

In case you didn't know, `ORD` is the airport code of Chicago O'Hare airport and `FLL` is the main airport in Fort Lauderdale, Florida, which we can now see in our `named_dests` data frame.

13.3 Joining by multiple "key" variables

Say instead we are in a situation where we need to join by multiple variables. For example, in the first figure in this section we see that in order to join the `flights` and `weather` data frames, we need more than one key variable: `year`, `month`, `day`, `hour`, and `origin`. This is because the combination of these 5 variables act to uniquely identify each observational unit in the `weather` data frame: hourly weather recordings at each of the 3 NYC airports.

We achieve this by specifying a vector of key variables to join by using the concatenate function `c`. Note the individual variables need to be wrapped in quotation marks.

```
flights_weather_joined <- flights %>%
  inner_join(weather, by = c("year", "month", "day", "hour", "origin"))
```

```
# A tibble: 335,220 x 32
    year month   day dep_time sched_dep_time dep_delay arr_time
   <dbl> <dbl> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 335,210 more rows, and 25 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour.x <dttm>, gain <dbl>, hours <dbl>,
#   gain_per_hour <dbl>, temp <dbl>, dewp <dbl>, humid <dbl>,
#   wind_dir <dbl>, wind_speed <dbl>, wind_gust <dbl>, precip <dbl>,
#   pressure <dbl>, visib <dbl>, time_hour.y <dttm>
```

Task: Looking at the first figure in this section, when joining `flights` and `weather` (or, in other words, matching the hourly weather values with each flight), why do we need to join by all of `year`, `month`, `day`, `hour`, and `origin`, and not just `hour`?

14 Other verbs

14.1 Select variables using select

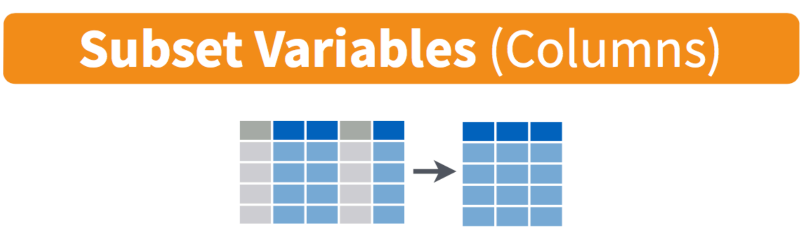


Figure 5: Select diagram from Data Wrangling with dplyr and tidyr cheatsheet.

We've seen that the `flights` data frame in the `nycflights13` package contains many different variables. The `names` function gives a listing of all the columns in a data frame; in our case you would run `names(flights)`. You can also identify these variables by running the `glimpse` function in the `dplyr` package:

```
glimpse(flights)
```

```
Observations: 336,776
Variables: 22
$ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013,...
$ month          <int> 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,...
$ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 55...
$ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 60...
$ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2...
$ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 7...
$ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 7...
$ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -...
$ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV",...
$ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79...
$ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN...
$ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR"...
$ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL"...
$ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138...
$ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 94...
$ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5,...
$ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, ...
$ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013...
$ gain           <dbl> -9, -16, -31, 17, 19, -16, -24, 11, 5, -10, 0, ...
$ hours          <dbl> 3.7833333, 3.7833333, 2.6666667, 3.0500000, 1.9...
$ gain_per_hour  <dbl> -2.3788546, -4.2290749, -11.6250000, 5.5737705,...
```

However, say you only want to consider two of these variables, say `carrier` and `flight`. You can `select` these:

```
flights %>%
  select(carrier, flight)
```

```
# A tibble: 336,776 x 2
   carrier flight
   <chr>    <int>
 1 UA        1545
 2 UA        1714
 3 AA        1141
 4 B6         725
 5 DL         461
 6 UA        1696
 7 B6         507
 8 EV        5708
 9 B6          79
10 AA         301
# ... with 336,766 more rows
```

This function makes navigating data sets with a very large number of variables easier for humans by restricting consideration to only those of interest, like `carrier` and `flight` above. So for example, this might make viewing the data set using the `View` spreadsheet viewer more digestible. However, as far as the computer is concerned it does not care how many additional variables are in the data set in question, so long as `carrier` and `flight` are included.

Another example involves the variable `year`. If you remember the original description of the `flights` data frame (or by running `?flights`), you will remember that this data corresponds to flights in 2013 departing

New York City. The `year` variable isn't really a variable here in that it doesn't vary, the `flights` data set actually comes from a larger data set that covers many years. We may want to remove the `year` variable from our data set since it won't be helpful for analysis in this case. We can deselect `year` by using the `-` sign:

```
flights_no_year <- flights %>%
  select(-year)
```

Or we could specify a ranges of columns:

```
flight_arr_times <- flights %>%
  select(month:dep_time, arr_time:sched_arr_time)
```

```
# A tibble: 336,776 x 5
   month   day dep_time arr_time sched_arr_time
   <int> <int>    <int>    <int>          <int>
 1     1     1      517      830            819
 2     1     1      533      850            830
 3     1     1      542      923            850
 4     1     1      544     1004           1022
 5     1     1      554      812            837
 6     1     1      554      740            728
 7     1     1      555      913            854
 8     1     1      557      709            723
 9     1     1      557      838            846
10     1     1      558      753            745
# ... with 336,766 more rows
```

The `select` function can also be used to reorder columns in combination with the `everything` helper function. Let's suppose we would like the `hour`, `minute`, and `time_hour` variables, which appear at the end of the `flights` data set, to actually appear immediately after the `day` variable:

```
flights_reorder <- flights %>%
  select(month:day, hour:time_hour, everything())
```

```
 [1] "month"          "day"            "hour"           "minute"
 [5] "time_hour"      "year"           "dep_time"       "sched_dep_time"
 [9] "dep_delay"      "arr_time"       "sched_arr_time" "arr_delay"
[13] "carrier"        "flight"         "tailnum"        "origin"
[17] "dest"           "air_time"       "distance"       "gain"
[21] "hours"          "gain_per_hour"
```

in this case `everything()` picks up all remaining variables. Lastly, the helper functions `starts_with`, `ends_with`, and `contains` can be used to choose **variables / column names** that match those conditions:

```
flights_begin_a <- flights %>%
  select(starts_with("a"))
```

```
# A tibble: 336,776 x 3
   arr_time arr_delay air_time
      <int>     <dbl>    <dbl>
 1      830        11      227
 2      850        20      227
 3      923        33      160
 4     1004       -18      183
 5      812       -25      116
 6      740        12      150
 7      913        19      158
```

```
 8      709       -14       53
 9      838        -8      140
10      753         8      138
# ... with 336,766 more rows
```

```
flights_delays <- flights %>%
  select(ends_with("delay"))
```

```
# A tibble: 336,776 x 2
   dep_delay arr_delay
       <dbl>     <dbl>
 1         2        11
 2         4        20
 3         2        33
 4        -1       -18
 5        -6       -25
 6        -4        12
 7        -5        19
 8        -3       -14
 9        -3        -8
10        -2         8
# ... with 336,766 more rows
```

```
flights_time <- flights %>%
  select(contains("time"))
```

```
# A tibble: 336,776 x 6
   dep_time sched_dep_time arr_time sched_arr_time air_time
      <int>          <int>    <int>          <int>    <dbl>
 1      517            515      830            819      227
 2      533            529      850            830      227
 3      542            540      923            850      160
 4      544            545     1004           1022      183
 5      554            600      812            837      116
 6      554            558      740            728      150
 7      555            600      913            854      158
 8      557            600      709            723       53
 9      557            600      838            846      140
10      558            600      753            745      138
# ... with 336,766 more rows, and 1 more variable: time_hour <dttm>
```

14.2 Rename variables using rename

Another useful function is `rename`, which as you may suspect renames one column to another name. Suppose we wanted `dep_time` and `arr_time` to be `departure_time` and `arrival_time` instead in the `flights_time` data frame:

```
flights_time <- flights %>%
  select(contains("time")) %>%
  rename(departure_time = dep_time,
         arrival_time = arr_time)
```

```
[1] "departure_time" "sched_dep_time" "arrival_time"   "sched_arr_time"
[5] "air_time"       "time_hour"
```

Note that in this case we used a single `=` sign with `rename`. eg. `departure_time = dep_time`. This is because we are not testing for equality like we would using `==`, but instead we want to assign a new variable `departure_time` to have the same values as `dep_time` and then delete the variable `dep_time`.

14.3 Find the top number of values using top_n

We can also use the `top_n` function which automatically tells us the most frequent `num_flights`. We specify the top 10 airports here:

```
named_dests %>%
  top_n(n = 10, wt = num_flights)
```

```
# A tibble: 10 x 9
   dest  num_flights airport_name      lat    lon   alt    tz dst   tzone
   <chr>       <int> <chr>           <dbl>  <dbl> <int> <dbl> <chr> <chr>
 1 ORD         17283 Chicago Ohare ~  42.0  -87.9   668    -6 A     Ameri~
 2 ATL         17215 Hartsfield Jac~  33.6  -84.4  1026    -5 A     Ameri~
 3 LAX         16174 Los Angeles In~  33.9 -118.    126    -8 A     Ameri~
 4 BOS         15508 General Edward~  42.4  -71.0    19    -5 A     Ameri~
 5 MCO         14082 Orlando Intl     28.4  -81.3    96    -5 A     Ameri~
 6 CLT         14064 Charlotte Doug~  35.2  -80.9   748    -5 A     Ameri~
 7 SFO         13331 San Francisco ~  37.6 -122.     13    -8 A     Ameri~
 8 FLL         12055 Fort Lauderdal~  26.1  -80.2     9    -5 A     Ameri~
 9 MIA         11728 Miami Intl       25.8  -80.3     8    -5 A     Ameri~
10 DCA          9705 Ronald Reagan ~  38.9  -77.0    15    -5 A     Ameri~
```

Note: The arguments `n` and `wt` arguments can be found by using the `?` function on `top_n`, i.e. ?top_n.

We can go one step further and tie together the `group_by` and `summarize` functions we used to find the most frequent flights:

```
ten_freq_dests <- flights %>%
  group_by(dest) %>%
  summarize(num_flights = n()) %>%
  arrange(desc(num_flights)) %>%
  top_n(n = 10)
```

```
Selecting by num_flights
```

```
# A tibble: 10 x 2
   dest  num_flights
   <chr>       <int>
 1 ORD         17283
 2 ATL         17215
 3 LAX         16174
 4 BOS         15508
 5 MCO         14082
 6 CLT         14064
 7 SFO         13331
 8 FLL         12055
 9 MIA         11728
10 DCA          9705
```

Task: What are some ways to select all three of the `dest`, `air_time`, and `distance` variables from `flights`? Give the code showing how to do this in at least three different ways.

Task: How could one use `starts_with`, `ends_with`, and `contains` to select columns from the `flights` data frame? Provide three different examples in total: one for `starts_with`, one for `ends_with`, and one for `contains`.

Task: Create a new data frame that shows the top 5 airports with the largest average arrival delays from NYC in 2013.

15 Summary

The table below lists a selection of the data wrangling verbs and summarises what they do. Using these verbs and the pipe `%>%` operator, you'll be able to write easily legible code to perform almost all the data wrangling necessary for the rest of this course.

Table 4: Summary of data wrangling verbs

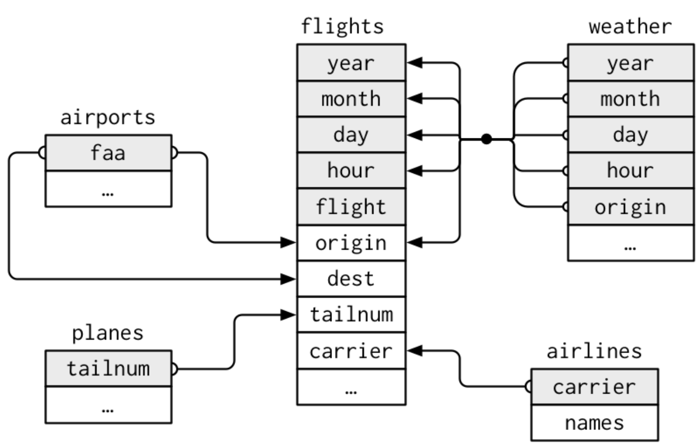
Verb	Operation
filter()	Pick out a subset of rows
summarize()	Summarise many values to one using a summary statistic function like mean(), median(), etc.
group_by()	Add grouping structure to rows in data frame. Note this does not change the values in the data frame.
mutate()	Create new variables by mutating existing ones
arrange()	Arrange rows of a data variable in ascending (default) or descending order
inner_join()	Join/merge two data frames, matching rows by a key variable
select()	Pick out a subset of columns to make data frames easier to view

15.1 Task

An airline industry measure of a passenger airline's capacity is the available seat miles, which is equal to the number of seats available multiplied by the number of miles or kilometers flown. So for example say an airline had 2 flights using a plane with 10 seats that flew 500 miles and 3 flights using a plane with 20 seats that flew 1000 miles, the available seat miles would be $2 \times 10 \times 500 + 3 \times 20 \times 1000 = 70{,}000$ seat miles.

Using the data sets included in the `nycflights13` package, compute the available seat miles for each airline sorted in descending order. After completing all the necessary data wrangling steps, the resulting data frame should have 16 rows (one for each airline) and 2 columns (airline name and available seat miles). Here are some hints:

1. Take a close look at all the data sets using the `View`, `head` or `glimpse` functions: `flights`, `weather`, `planes`, `airports`, and `airlines` to identify which variables are necessary to compute available seat miles.
2. This diagram (from the **Joining section**) will also be useful.



3. Consider the data wrangling verbs in the table above as your toolbox!

If you want to work through it **step by step**, here are some hints:

Step 1: To compute the available seat miles for a given flight, we need the `distance` variable from the `flights` data frame and the `seats` variable from the `planes` data frame, necessitating a join by the key variable `tailnum`. To keep the resulting data frame easy to view, we'll `select` only these two variables and `carrier`.

Step 2: Now for each flight we can compute the available seat miles `ASM` by multiplying the number of seats by the distance via a `mutate`.

Step 3: Next we want to sum the `ASM` for each carrier. We achieve this by first grouping by `carrier` and then summarising using the `sum` function.

Step 4: However, if it was the case that some carriers had certain flights with missing `NA` values, the resulting table above would also return `NA`'s (NB: this is not the case for this data). We can eliminate these by adding the `na.rm = TRUE` argument to `sum`, telling R that we want to remove the `NA`'s in the sum.

Step 5: Finally, `arrange` the data in `desc`ending order of `ASM`.

16 Further Tasks

16.1 Further Task 1

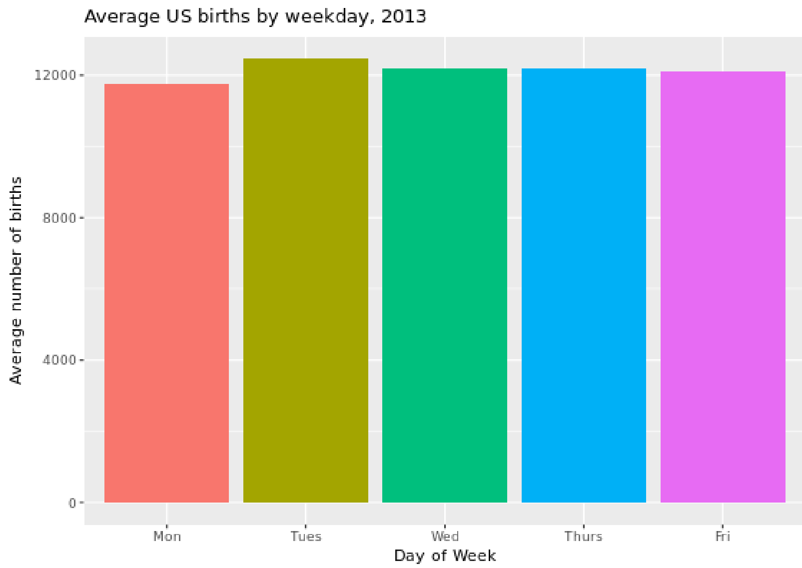
In this task we will work with the data set analysed and reported in the 2016 article from FiveThirtyEight.com entitled Some People Are Too Superstitious To Have A Baby On Friday The 13th. The data set is called `US_births_2000_2014` and is within the `fivethirtyeight` package.

1. Create an object called `US_births_2013` which focuses only on data corresponding to 2013 births.
2. By only choosing birth data for the years 2010, 2011, 2012, and 2014 create a new data frame called `US_births_small` and check that this resulting data frame has 1461 rows. Note that there are many different ways to do this, but try and come up with three different ways using:

- the "or" operator `|`
- the `%in%` operator
- the "not" operator `!`

or combinations of them.

3. Suppose we are interested in choosing rows for only weekdays (not Saturdays or Sundays) for `day_of_week` in year 2013. Write the code to do so and give the name `US_births_weekdays_2013` to the resulting data frame. Note that you may want to run `US_births_2000_2014 %>% distinct(day_of_week)` to identify the specific values of `day_of_week`.
4. Using what you covered in Week 1, produce an appropriate plot looking at the pattern of births on all weekdays in 2013 coloured by the particular day of the week.
5. The plot in the previous task has shown there are some outliers in the data for US births on weekdays in 2013. We can use the `summarize` function to get an idea for how these outliers may affect the shape of the births variable in `US_births_weekdays_2013`. Write some code to calculate the mean and median values for all weekday birth totals in 2013. Store this aggregated data in the data frame `birth_summ`. What do these values suggest about the effects of the outliers?
6. Instead of looking at the overall mean and median across all of 2013 weekdays, calculate the mean and median for each of the five different weekdays throughout 2013. Using the same names for the columns as in the `birth_summ` data frame in the previous exercise, create a new data frame called `birth_day_summ`.
7. Using the aggregated data in the `birth_day_summ data` frame, produce this barplot.



16.2 Further Task 2

In this task we will work with the data set analysed and reported in the 2014 article from FiveThirtyEight.com entitled 41 Percent Of Fliers Think You're Rude If You Recline Your Seat. The data set is called `flying` and is within the `fivethirtyeight` package.

1. Write code to determine the proportion of respondents in the survey that responded with **Very** when asked if a passenger reclining their seat was rude. You should determine this proportion across the different levels of `age` and `gender` resulting in a data frame of size 8 x 3. Assign the name `prop_very` to this calculated proportion in this aggregated data frame.

Hint 1: We can obtain proportions using the `mean` function applied to logical values. For example suppose we want to count the proportion of "heads" in five tosses of a fair coin. If the results of the five tosses are stored in

```
tosses <- c("heads", "tails", "tails", "heads", "heads")
```

then we can use `mean(tosses == "heads")` to get the resulting answer of 0.6.

Hint 2: Including the function `na.omit(TRUE)` in the 'pipe' (`%>%`) removes all entries that are not complete whereas including the argument `na.rm=TRUE` in the `mean` function removes just those entries where the relevant variable value is missing.

2. Using the aggregated data you've created, produce two bar plots (one stacked, the other side-by-side) to show the differences between the sexes of the proportion of people who believe reclining your seat is 'very' rude, within each age group. Also, consider

   - What stands out to you as you review these proportions?
   - What gender and age-range pairings have the highest and lowest proportions thinking reclining airline seats is very rude in this survey?