CHESTER ISMAY AND ALBERT Y. KIM

# A MODERN DIVE INTO DATA WITH R

*Contents*

# 1
# Prerequisites

This book was written using the **bookdown** R package from Yihui Xie. In order to follow along and run the code in this book on your own, you'll need to have access to R and RStudio. You can find more information on both of these with a simple Google search for "R" and for "RStudio". An introduction to using R, RStudio, and R Markdown is also available in a free book here (**?**). It is recommended that you refer back to this book frequently as it has GIF screen recordings that you can follow along with as you learn.

We will keep a running list of R packages you will need to have installed to complete the analysis as well here in the `needed_pkgs` character vector. You can check if you have all of the needed packages installed by running all of the lines below. The last lines including the `if` will install them as needed (i.e., download their needed files from the internet to your hard drive).

You can run the `library` function on them to load them into your current analysis. Prior to each analysis where a package is needed, you will see the corresponding `library` function in the text.

```
needed_pkgs <- c("nycflights13", "dplyr", "ggplot2", "knitr",
  "devtools", "ggplot2movies", "dygraphs", "rmarkdown")

new.pkgs <- needed_pkgs[!(needed_pkgs %in% installed.packages())]

if(length(new.pkgs)) {
  install.packages(new.pkgs, repos = "http://cran.rstudio.com")
}
```

**Book was last updated:**

```
## [1] "By Chester on Friday, August 26, 2016 16:45:08 PDT"
```

## 1.1 Colophon

The source of the book is available here and was built with versions of R packages given here. This may not be of importance for initial readers of this book, but you can reproduce a duplicate of this book by installing these versions of the packages.

| package | * | version | date | source |
|---|---|---|---|---|
| assertthat | | 0.1 | 2013-12-06 | CRAN (R 3.3.0) |
| base64enc | | 0.1-3 | 2015-07-28 | CRAN (R 3.3.0) |
| BH | | 1.60.0-2 | 2016-05-07 | CRAN (R 3.3.0) |
| bitops | | 1.0-6 | 2013-08-17 | CRAN (R 3.3.0) |
| caTools | | 1.17.1 | 2014-09-10 | CRAN (R 3.3.0) |
| colorspace | | 1.2-6 | 2015-03-11 | CRAN (R 3.3.0) |
| curl | | 1.2 | 2016-08-13 | CRAN (R 3.3.0) |
| DBI | | 0.5 | 2016-08-11 | CRAN (R 3.3.0) |
| devtools | | 1.12.0 | 2016-06-24 | CRAN (R 3.3.0) |
| dichromat | | 2.0-0 | 2013-01-24 | CRAN (R 3.3.0) |
| digest | | 0.6.10 | 2016-08-02 | CRAN (R 3.3.0) |
| dplyr | * | 0.5.0 | 2016-06-24 | CRAN (R 3.3.0) |
| dygraphs | * | 1.1.1-1 | 2016-08-06 | CRAN (R 3.3.0) |
| evaluate | | 0.9 | 2016-04-29 | CRAN (R 3.3.0) |
| formatR | | 1.4 | 2016-05-09 | CRAN (R 3.3.0) |
| ggplot2 | * | 2.1.0 | 2016-03-01 | CRAN (R 3.3.0) |
| ggplot2movies | | 0.0.1 | 2015-08-25 | CRAN (R 3.3.0) |
| git2r | | 0.15.0 | 2016-05-11 | CRAN (R 3.3.0) |
| gtable | | 0.2.0 | 2016-02-26 | CRAN (R 3.3.0) |
| highr | | 0.6 | 2016-05-09 | CRAN (R 3.3.0) |
| htmltools | | 0.3.5 | 2016-03-21 | CRAN (R 3.3.0) |
| htmlwidgets | | 0.7 | 2016-08-02 | CRAN (R 3.3.0) |
| httr | | 1.2.1 | 2016-07-03 | CRAN (R 3.3.0) |
| jsonlite | | 1.0 | 2016-07-01 | CRAN (R 3.3.0) |
| knitr | * | 1.14 | 2016-08-13 | CRAN (R 3.3.0) |
| labeling | | 0.3 | 2014-08-23 | CRAN (R 3.3.0) |
| lattice | | 0.20-33 | 2015-07-14 | CRAN (R 3.3.1) |
| lazyeval | | 0.2.0 | 2016-06-12 | CRAN (R 3.3.0) |
| magrittr | | 1.5 | 2014-11-22 | CRAN (R 3.3.0) |
| markdown | | 0.7.7 | 2015-04-22 | CRAN (R 3.3.0) |
| MASS | | 7.3-45 | 2016-04-21 | CRAN (R 3.3.1) |
| memoise | | 1.0.0 | 2016-01-29 | CRAN (R 3.3.0) |
| mime | | 0.5 | 2016-07-07 | CRAN (R 3.3.0) |
| munsell | | 0.4.3 | 2016-02-13 | CRAN (R 3.3.0) |
| nycflights13 | * | 0.2.0 | 2016-04-30 | CRAN (R 3.3.0) |
| openssl | | 0.9.4 | 2016-05-25 | CRAN (R 3.3.0) |
| plyr | | 1.8.4 | 2016-06-08 | CRAN (R 3.3.0) |
| R6 | | 2.1.3 | 2016-08-19 | CRAN (R 3.3.0) |
| RColorBrewer | | 1.1-2 | 2014-12-07 | CRAN (R 3.3.0) |

| | | | |
|---|---|---|---|
| Rcpp | 0.12.6 | 2016-07-19 | CRAN (R 3.3.0) |
| reshape2 | 1.4.1 | 2014-12-06 | CRAN (R 3.3.0) |
| rmarkdown | 1.0 | 2016-07-08 | CRAN (R 3.3.0) |
| rstudioapi | 0.6 | 2016-06-27 | CRAN (R 3.3.0) |
| scales | 0.4.0 | 2016-02-26 | CRAN (R 3.3.0) |
| stringi | 1.1.1 | 2016-05-27 | CRAN (R 3.3.0) |
| stringr | 1.1.0 | 2016-08-19 | CRAN (R 3.3.0) |
| tibble | 1.1 | 2016-07-04 | CRAN (R 3.3.0) |
| whisker | 0.3-2 | 2013-04-28 | CRAN (R 3.3.0) |
| withr | 1.0.2 | 2016-06-20 | CRAN (R 3.3.0) |
| xts | 0.9-7 | 2014-01-02 | CRAN (R 3.3.0) |
| yaml | 2.1.13 | 2014-06-12 | CRAN (R 3.3.0) |
| zoo | 1.7-13 | 2016-05-03 | CRAN (R 3.3.0) |

# 2
# *Introduction*

## *2.1 Preamble*

This book is inspired by three books:

- "Mathematical Statistics with Resampling and R" (**?**),
- "Intro Stat with Randomization and Simulation" (**?**), and
- "R for Data Science" (**?**).

These books provide excellent resources on how to use resampling to build statistical concepts like normal distributions using computers instead of focusing on memorization of formulas. The last two books also provide a path towards free alternatives to the traditionally expensive introductory statistics textbook. When looking over the vast number of introductory statistics textbooks we found that there wasn't one that incorporated many of the new R packages directly into the text. Additionally, there wasn't an open-source, free textbook available that showed new learners all of the following

1. how to use R to explore and visualize data
2. how to use randomization and simulation to build inferential ideas
3. how to effectively create stories using these ideas to convey information to a lay audience.

We will introduce sometimes difficult statistics concepts through the medium of data visualization. In today's world, we are bombarded with graphics that attempt to convey ideas. We will explore what makes a good graphic and what the standard ways are to convey relationships with data. You'll also see the use of visualization to introduce concepts like mean, median, standard deviation, distributions, etc. In general, we'll use visualization as a way of building almost all of the ideas in this book.

Additionally, this book will focus on the triad of computational thinking, data thinking, and inferential thinking. We'll see throughout the book how these three modes of thinking can build effective ways to work with, describe, and convey statistical knowledge. In order to do so, you'll see the importance of literate programming to develop literate data science. In other words, you'll see how to write code and descriptions that are useful not just for a computer to execute but also for readers to understand exactly what a statistical analysis is doing and how it works. Hal Abelson coined the phrase that we will follow throughout this book:

"Programs must be written for people to read, and only incidentally for machines to execute."

## *2.2 Two driving data sources*

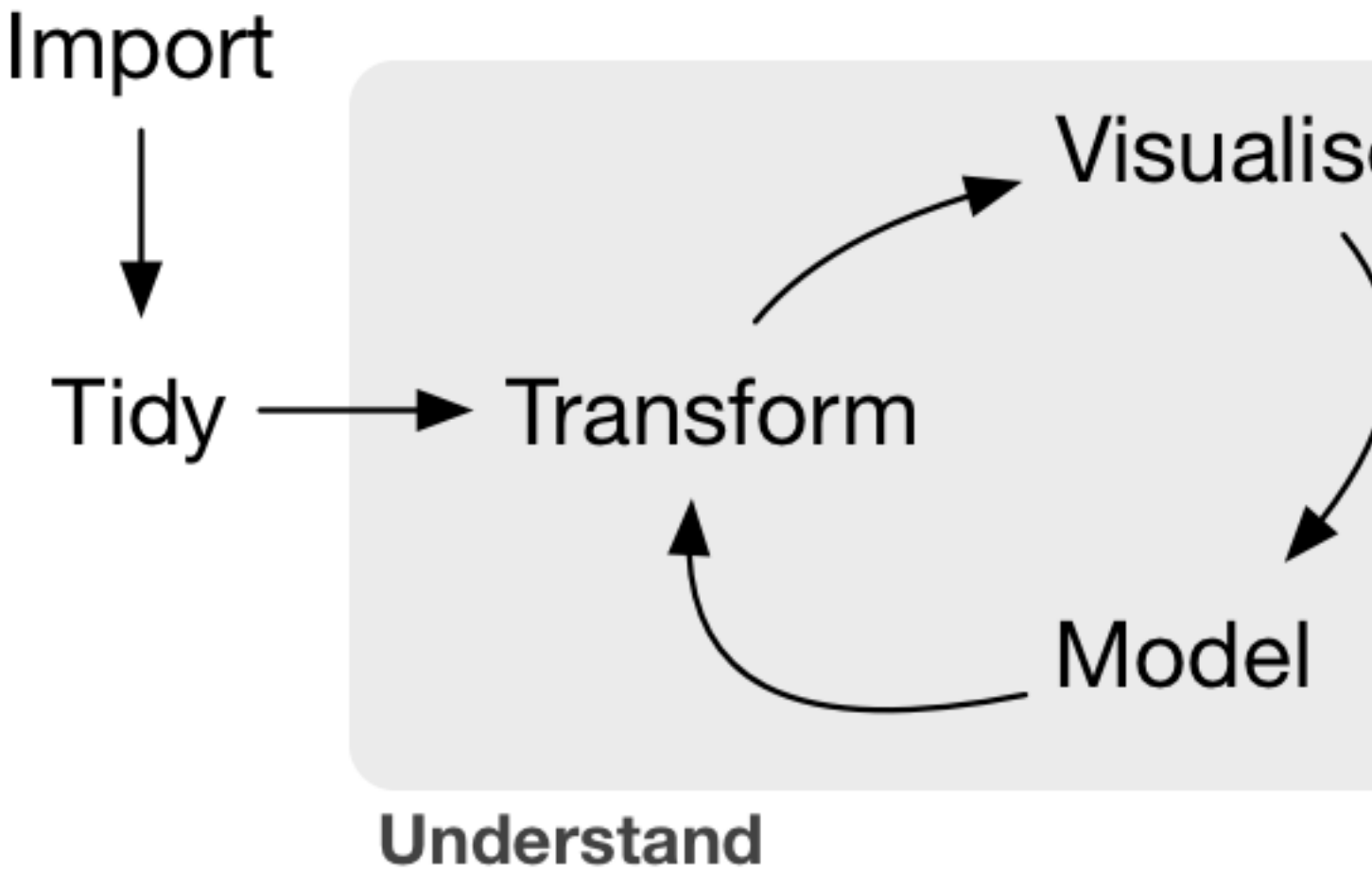
Instead of hopping from one data set to the next, we've decided to focus throughout the book on two different data sources: flights leaving New York City in 2013 and IMDB (Internet Movie DataBase) ratings on movies. By focusing on just two large data sources, it is our hope that you'll be able to see how each of the chapters is interconnected. You'll see how the data being tidy leads into data visualization and manipulation and how those concepts tie into inference and regression.

## *2.3 Data/science pipeline*

You may think of statistics as just being a bunch of numbers. We commonly hear the phrase "statistician" when listening to broadcasts of sporting events. Statistics (in particular, data analysis), in addition to describing numbers like with baseball batting averages, plays a vital role in all of the sciences. You'll commonly hear the phrase "statistically significant" thrown around in the media. You'll see things that say "Science now shows that chocolate is good for you." Underpinning these claims is data analysis. By the end of this book, you'll be able to better understand whether these claims should be trusted or whether we should be weary. Inside data analysis are many sub-fields that we will discuss throughout this book (not necessarily in this order):

- data collection
- data manipulation
- data visualization
- data modeling
- inference
- interpretation of results
- data storytelling

This can be summarized in a graphic that is commonly used by Hadley Wickham:




We will begin with a discussion on what is meant by tidy data and then dig into the gray **Understand** portion of the cycle and conclude by talking about interpretting and discussing the results of our models via **Communication**. These steps are vital to any statistical analysis. But why should you care about statistics? "Why did they make me take this class?"

There's a reason so many fields require a statistics course. Scientific knowledge grows through an understanding of statistical significance and data analysis. You needn't be intimidated by statistics. It's not the beast that it used to be and paired with computation you'll see how reproducible research in the sciences particularly increases scientific knowledge.

## *2.4 Reproducibility*

> "The most important tool is the *mindset*, when starting, that the end product will be reproducible." – Keith Baggerly

Another large goal of this book is to help readers understand the importance of reproducible analyses. The hope is to get readers into the habit of making their analyses reproducible from the very beginning. This means we'll be trying to help you build new habits. This will take

practice and be difficult at times. You'll see just why it is so important for you to keep track of your code and well-document it to help yourself later and any potential collaborators as well.

Copying and pasting is not the way that efficient and effective scientific research is conducted. It's much more important for time to be spent on data collection and data analysis and not on copying and pasting plots back and forth across a variety of programs.

In a traditional analyses if an error was made with the original data, we'd need to step through the entire process again: recreate the plots and copy and paste all of the new plots and our statistical analsis into your document. This is error prone and a frustrating use of time. We'll see how to use R Markdown to get away from this tedious activity so that we can spend more time doing science.

"We are talking about *computational* reproducibility." - Yihui Xie

Reproducibility means a lot of things in terms of different scientific fields. Are experiments conducted in a way that another researcher could follow the steps and get similar results? In this book, we will focus on what is known as **computational reproducibility**. This refers to being able to pass all of one's data analysis and conclusions to someone else and have them get exactly the same results on their machine. This allows for time to be spent doing actual science and interpretting of results and assumptions instead of the more error prone way of starting from scratch or follow a list of steps that may be different from machine to machine.

## *2.5 Who is this book for?*

This book is targetted at students taking a traditional intro stats class in a small college environment using RStudio and preferably RStudio Server. We assume no prerequisites: no calculus and no prior programming experience. This is intended to be a gentle and nice introduction to the practice of statistics in terms of how data scientists, statisticians, and other scientists analyze data and write stories about data. We have intentionally avoided the use of throwing formulas at you and instead have focused on developing statistical concepts via data visualization and statistical computing. We hope this is a more intuitive experience than the way statistics has traditionally been taught in the past (and how it is commonly perceived from the outside). We additionally hope that you see the value of reproducible research via R as you continue in your studies. We understand that there will initially be growing pains in learning to program but we are here to help you and you should know that there is a huge community of R users that are always happy to help newbies along.

Now let's get into learning about how to create good stories about and with data!

# Part I

# Data Exploration

# 3
# *Tidy data*

In this chapter, we'll discuss the importance of tidy data. You may think that this means just having your data in a spreadsheet, but you'll see that it is actually more specific than that. Data actually comes to us in a variety of formats from pictures to text and to just numbers. We'll focus on datasets that can be stored in a spreadsheet throughout this book as that is the most common way data is collected in the sciences.

Having tidy data will allow us to more easily create data visualizations as we will see in Chapter **??**. It will also help us with manipulating data in Chapter **??** and in all subsequent chapters when we discuss statistical inference. You may not necessarily understand the importance for **tidy data** but it will become more and more apparent as we proceed through the book.

## *3.1 What is tidy data?*

You have surely heard the word "tidy" in your life:

- "Tidy up your room!"
- "Please write your homework in a tidy way so that it is easier to grade and to provide feedback."
- Marie Kondo's best-selling book *The Life-Changing Magic of Tidying Up: The Japanese Art of Decluttering and Organizing*
- "I am not by any stretch of the imagination a tidy person, and the piles of unread books on the coffee table and by my bed have a plaintive, pleading quality to me - 'Read me, please!' " - Linda Grant

So what does it mean for your data to be **tidy**? Put simply: it means that your data is organized. But it's more than just that. It means that your data follows the same standard format making it easy for others to find elements of your data, to manipulate and transform your data, and for our purposes continuing with the common theme: it makes it easier to visualize your data and the relationships between different variables in your data.

We will follow Hadley Wickham's definition of **tidy data** here (**?**):

> A dataset is a collection of values, usually either numbers (if quantitative) or strings (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 race) across attributes.

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset 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.

Reading over this definition, you can begin to think about datasets that won't follow this nice format.

---

***Learning check***

**(LC3.1)** Give an example dataset that doesn't follow this format.

- What features of this dataset might make it difficult to visualize?
- How could the dataset be tweaked to make it **tidy**?

---

## *3.2 The `nycflights13` datasets*

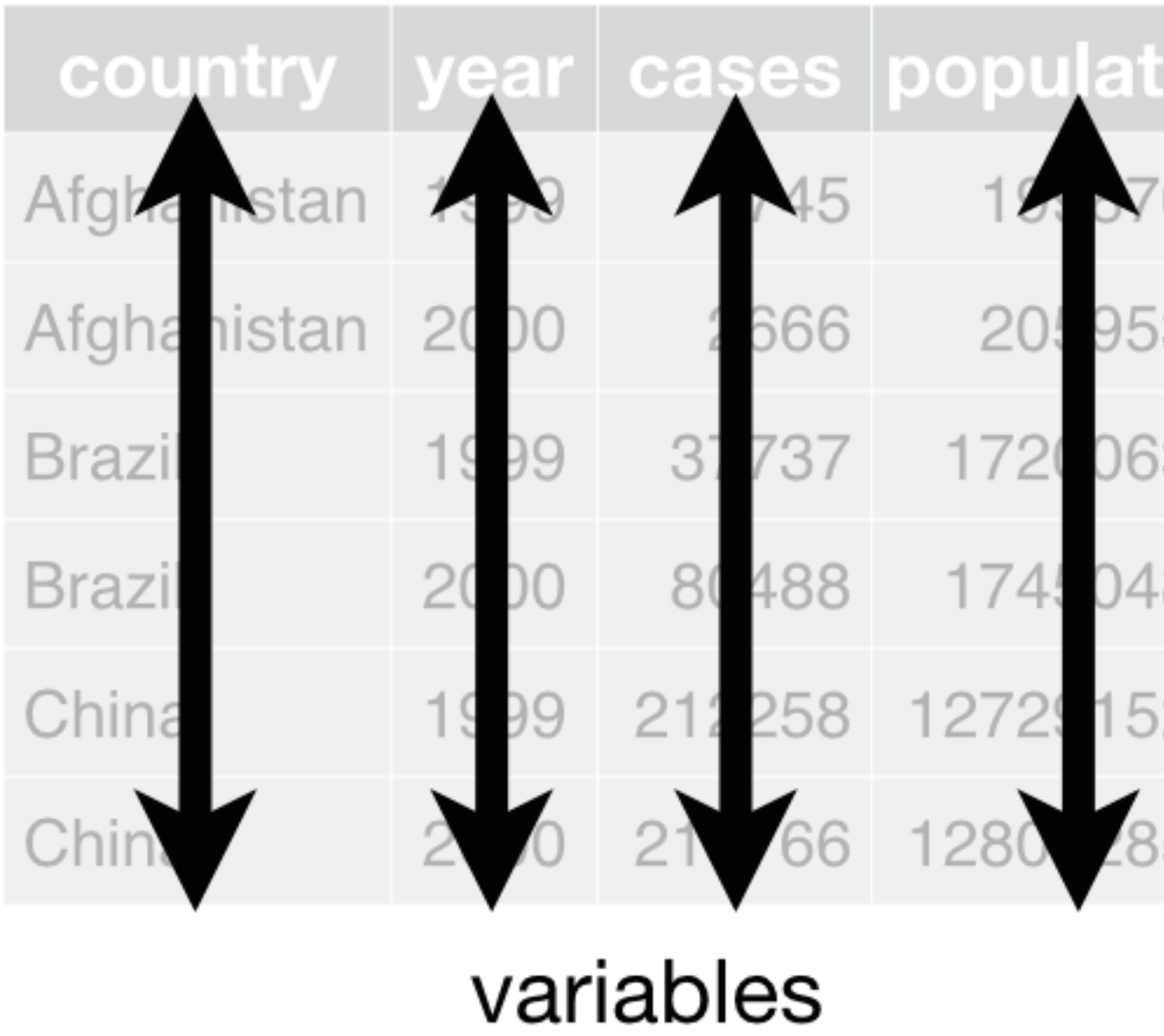
We likely have all flown on airplanes or know someone that has. Air travel has become an ever-present aspect of our daily lives. If you live in or are visiting a relatively large city and you walk around that city's airport, you see gates showing flight information from many different airlines. And you will frequently see that some flights are delayed because of a variety of conditions. Are there ways that we can avoid having to deal with these flight delays?

We'd all like to arrive at our destinations on time whenever possible. (Unless you secretly love hanging out at airports. If you are one of these people, pretend for the moment that you are very much anticipating being at your final destination.) Hadley Wickham (herein just referred to as "Hadley") created multiple datasets containing information about departing flights from the New York City area in 2013 (**?**). We will begin by loading in one of these datasets, the `flights` dataset, and getting an idea of its structure:

```
library(nycflights13)
data(flights)
```

The `library` function here loads the R package `nycflights13` into the current R environment in which you are working. (Note that you'll get an error if you try to load this package

Figure 3.1:
Tidy data graphic from http://r4ds.had.co.nz/tidy-data.html



in and it hasn't been installed. Check Chapter **??** to make sure the package has been downloaded.) The next line of code `data(flights)` loads in the `flights` dataset that is stored in the `nycflights13` package.

This dataset and most others presented in this book will be in the `data.frame` format in R. Dataframes are ways to look at collections of variables that are tightly coupled together. Frequently, the best way to get a feel for a dataframe is to use the `View` function in RStudio. This command will be given throughout the book as a reminder, but the actual output will be hidden.

```
View(flights)
```

---

***Learning check***

**(LC3.4)** What does any *ONE* row in this dataset refer to?

- A. Data on an airline
- B. Data on a flight
- C. Data on an airport
- D. Data on multiple flights

---

By running `View(flights)`, we see the different **variables** listed in the columns and we see that there are different types of variables. Some of the variables like `distance`, `day`, and `arr_delay` are what we will call **quantitative** variables. These variables vary in a numerical way. Other variables here are **categorical**.

Note that if you look in the leftmost column of the `View(flights)` output, you will see a column of numbers. These are the row numbers of the dataset. If you glance across a row with the same number, say row 5, you can get an idea of what each row correspond to. In other words, this will allow you to identify what object is being referred to in a given row. This is often called the **observational unit**. The **observational unit** in this example is an individual flight departing New York City in 2013.

**Note**: Frequently the first thing you should do when given a dataset is to

- identify the observation unit,
- specify the variables, and
- give the types of variables you are presented with.

```
str(flights)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    336776 obs. of  19 variables:
```

```
##  $ year          : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ month         : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ day           : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dep_time      : int  517 533 542 544 554 554 555 557 557 558 ...
##  $ sched_dep_time: int  515 529 540 545 600 558 600 600 600 600 ...
##  $ dep_delay     : num  2 4 2 -1 -6 -4 -5 -3 -3 -2 ...
##  $ arr_time      : int  830 850 923 1004 812 740 913 709 838 753 ...
##  $ sched_arr_time: int  819 830 850 1022 837 728 854 723 846 745 ...
##  $ arr_delay     : num  11 20 33 -18 -25 12 19 -14 -8 8 ...
##  $ carrier       : chr  "UA" "UA" "AA" "B6" ...
##  $ flight        : int  1545 1714 1141 725 461 1696 507 5708 79 301 ...
##  $ tailnum       : chr  "N14228" "N24211" "N619AA" "N804JB" ...
##  $ origin        : chr  "EWR" "LGA" "JFK" "JFK" ...
##  $ dest          : chr  "IAH" "IAH" "MIA" "BQN" ...
##  $ air_time      : num  227 227 160 183 116 150 158 53 140 138 ...
##  $ distance      : num  1400 1416 1089 1576 762 ...
##  $ hour          : num  5 5 5 5 6 5 6 6 6 6 ...
##  $ minute        : num  15 29 40 45 0 58 0 0 0 0 ...
##  $ time_hour     : POSIXct, format: "2013-01-01 05:00:00" ...
```

---

***Learning check***

**(LC3.5)** What are some examples in this dataset of **categorical** variables? What makes them different than **quantitative** variables?

**(LC3.6)** What does `int`, `num`, and `chr` mean in the output above?

**(LC3.7)** How many different columns are in this dataset?

**(LC3.8)** How many different rows are in this dataset?

---

Another way to view the properties of a dataset is to use the `str` function ("str" is short for "structure"). This will give you the first few entries of each variable in a row after the variable. In addition, the type of the variable is given immediately after the `:` following each variable's name. Here, `int` and `num` refer to quantitative variables. In contrast, `chr` refers to categorical variables. One more type of variable is given here with the `time_hour` variable: **POSIXct**. As you may suspect, this variable corresponds to a specific date and time of day.

Another nice feature of R is the help system. You can get help in R by simply entering a question mark before the name of a function or an object and you will be presented with a page showing the documentation. Note that this output help file is omitted here but can be accessed here on page 3 of the PDF document.

```
?flights
```

Another aspect of tidy data is a description of what each variable in the dataset represents. This helps others to understand what your variable names mean and what they correspond to. If we look at the output of `?flights`, we can see that a description of each variable by name is given.

An important feature to **ALWAYS** include with your data is the appropriate units of measurement. We'll see this further when we work with the `dep_delay` variable in Chapter **??**. (It's in minutes, but you'd get some really strange interpretations if you thought it was in hours or seconds. UNITS MATTER!)

## *3.3 How is `flights` tidy?*

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 how Hadley defined tidy data:

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.

We identified earlier that the observational unit in the `flights` dataset is an individual flight. And we have shown that this dataset consists of 336776 flights with 19 variables. In other words, some rows of this dataset don't refer to a measurement on an airline or on an airport. They specifically refer to characteristics/measurements on a given **flight** from New York City in 2013.

By contrast, also included in the `nycflights13` package are datasets with different observational units (**?**):

- `weather`: hourly meteorological data for each airport
- `planes`: construction information about each plane
- `airports`: airport names and locations
- `airlines`: translation between two letter carrier codes and names

You may have been asking yourself what `carrier` refers to in the `str(flights)` output above. The `airlines` dataset provides a description of this with each airline being the observational unit:

```
data(airlines)
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.
```

As can be seen here when you just enter the name of an object in R, by default it will print the contents of that object to the screen. Be careful! It's usually better to use the `View()` function in RStudio since larger objects may take awhile to print to the screen and it likely won't be helpful to you to have hundreds of lines outputted.

### *3.4 Normal forms of data*

The datasets included in the `nycflights13` package are in a form that minimizes redundancy of data. We will see that there are ways to *merge* (or *join*) the different tables together easily. We are capable of doing so because each of the tables have *keys* in common to relate one to another. This is an important property of **normal forms** of data. The process of decomposing dataframes into less redundant tables without losing information is called **normalization**. More information is available on Wikipedia.

We saw an example of this above with the `airlines` dataset. While the `flights` dataframe could also include a column with the names of the airlines instead of the carrier code, this would be repetitive since there is a unique mapping of the carrier code to the name of the airline/carrier.

Below an example is given showing how to **join** the `airlines` dataframe together with the `flights` dataframe by linking together the two datasets via a common **key** of `"carrier"`. Note that this "joined" dataframe is assigned to a new dataframe called `joined_flights`.

```
library(dplyr)
joined_flights <- inner_join(x = flights, y = airlines, by = "carrier")
```

```
View(joined_flights)
```

If we `View` this dataset, we see a new variable has been created called (We will see in Subsection **??** ways to change `name` to a more descriptive variable name.)

More discussion about joining dataframes together will be given in Chapter **??**. We will see there that the names of the columns to be linked need not match as they did here with `"carrier"`.

---

---

***Review questions***

**(RQ3.1)** What are common characteristics of "tidy" datasets?

**(RQ3.2)** What makes "tidy" datasets useful for organizing data?

**(RQ3.4)** How many variables are presented in the table below? What does each row correspond to? (**Hint:** You may not be able to answer both of these questions immediately but take your best guess.)

| students | faculty |
|---|---|
| 4 | 2 |
| 6 | 3 |

**(RQ3.5)** The confusion you may have encountered in Question 4 is a common one those that work with data are commonly presented with. This dataset is not tidy. Actually, the dataset in Question 4 has three variables not the two that were presented. Make a guess as to what these variables are and present a tidy dataset instead of this untidy one given in Question 4.

**(RQ3.6)** The actual data presented in Question 4 is given below in tidy data format:

| role | Sociology? | Type of School |
|---|---|---|
| student | TRUE | Public |
| student | TRUE | Public |
| student | TRUE | Public |
| student | TRUE | Public |
| student | FALSE | Public |
| student | FALSE | Public |
| student | FALSE | Private |
| student | FALSE | Private |
| student | FALSE | Private |
| student | FALSE | Private |
| faculty | TRUE | Public |
| faculty | TRUE | Public |
| faculty | FALSE | Public |
| faculty | FALSE | Private |
| faculty | FALSE | Private |

- What does each row correspond to?
- What are the different variables in this dataframe?
- The `Sociology?` variable is known as a logical variable. What types of values does a logical variable take on?

**(RQ3.7)** What are some advantages of data in normal forms? What are some disadvantages?

_____

_____

## *3.5 What's to come?*

In Chapter **??**, we will further explore the distribution of a variable in a related dataset to `flights`: the `temp` variable in the `weather` dataset. We'll be interested in understanding how this variable varies in relation to the values of other variables in the dataset. We will see that visualization is often a powerful tool in helping us see what is going on in a dataset. It will be a useful way to expand on the `str` function we have seen here for tidy data.

# 4
# *Visualizing Data*

In Chapter **??**, we discussed the importance of datasets being **tidy**. You will see in examples here why having a tidy dataset helps us immensely with plotting our data. In plotting our data, we will be able to gain valuable insights from our data that we couldn't initially see from just looking at the raw tidy data. We will focus on using Hadley's `ggplot2` package in doing so, which was developed to work specifically on datasets that are **tidy**. It provides an easy way to customize your plots and is based on data visualization theory given in *The Grammar of Graphics* (**?**).

Graphics provide a nice way for us to get a sense for how quantitative variables compare in terms of their center and their spread. The most important thing to know about graphics is that they should be created to make it obvious for your audience to see the findings you want to get across. This requires a balance on not including too much in your plots, but also including enough so that relationships and interesting findings can be easily seen. As we will see, plots/graphics also help us to identify patterns and outliers in our data. We will see that a common extension of these ideas is to compare the distribution (i.e., what the spread of a variable looks like) as we go across the levels of a different categorical variable.

## *4.1 Five Named Graphs - The FNG*

For our purposes here, we will be working with five different types of graphs. (Note that we will use a lot of different words here in regards to plotting - "graphs", "plots", and "charts" are all ways to discuss a resulting graphic. You can think of them as all being synonyms.) These five plots are:

- histograms
- boxplots
- barplots
- scatter-plots
- line-graphs

With this toolbox of plots, you can visualize just about any type of variable thrown at you. We will discuss some other variations of these but with the FNG in your repertoire you can do big things! Something we will also stress here is that certain plots only work for categor-

ical/logical variables and others only for quantitative variables. You'll want to quiz yourself often on which plot makes sense with a given problem set-up.

We now introduce another dataframe in the `nycflights13` package introduced in Chapter **??**.

```
library(nycflights13)
data(weather)
```

## *4.2 Histograms*

Our focus now turns to the `temp` variable in this `weather` dataset. We would like to visualize what the 26130 temperatures look like. Looking over the `weather` dataset[1] and running `?weather`, we can see that the `temp` variable corresponds to hourly temperature (in Fahrenheit) recordings at weather stations near airports in New York City. We could just produce points where each of the different values appears on something similar to a number line:

[1] Recall that to view a dataset in spreadsheet format in RStudio, you can run the `View()` function with the dataframe as its argument.

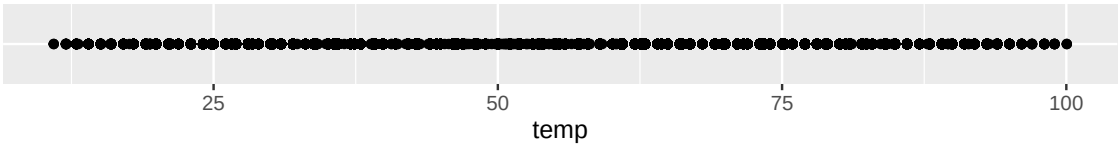



Figure 4.1: Strip Plot of Hourly Temperature Recordings from NYC in 2013

This gives us a general idea of how the values of `temp` differ. We see that temperatures vary from around 11 up to 100 degrees Fahrenheit. The area between 40 and 60 degrees appears to have more points plotted than outside that range.

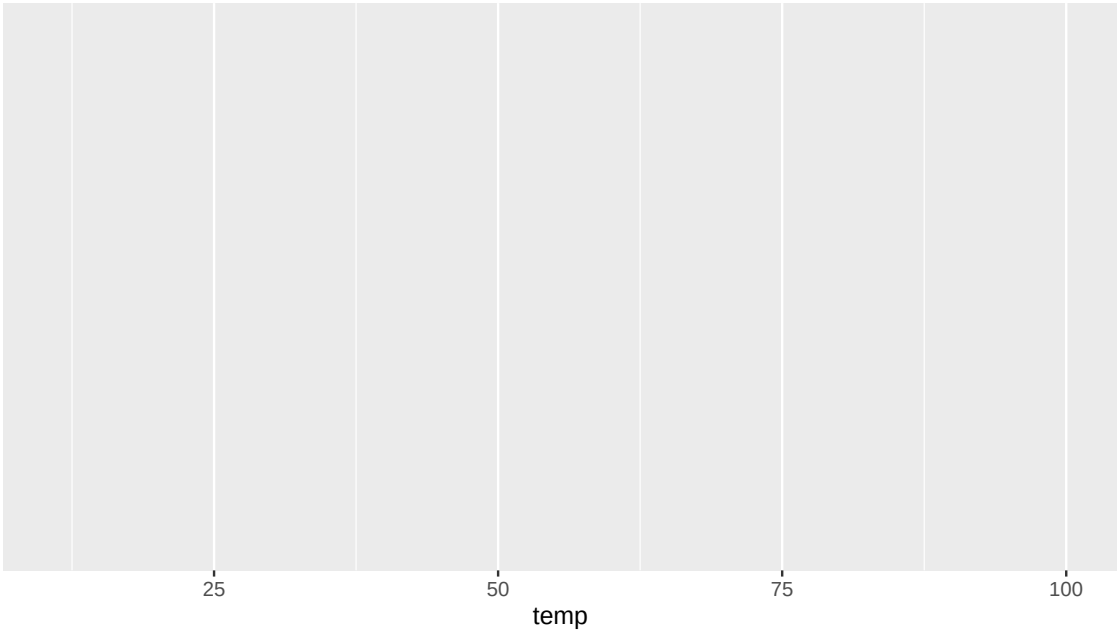
What is commonly produced instead of this strip plot is a plot known as a **histogram**. The **histogram** show how many elements of the variable fall in specified **bins**. These **bins** may correspond to between 0-10°F, 10-20°F, etc.

To produce a histogram, we introduce the Hadley's `ggplot2` package (**?**). We will use the `ggplot` function which expects at a bare minimal as arguments

- the dataframe where the variables exist and
- the names of the variables to be plotted.

The names of the variables will be entered into the `aes` function as arguments where `aes` stands for "aesthetics".

```
ggplot(data = weather, mapping = aes(x = temp))
```




The plot given above is not a histogram, but the output does show us a bit of what is going on with `ggplot(data = weather, mapping = aes(x = temp))`. It is producing a backdrop onto which we will "paint" elements.

We next proceed by adding a layer—hence, the use of the `+` symbol—to the plot to produce a histogram. (Note also here that we don't have to specify the `data =` and `mapping =` text in our function calls. This is covered in more detail in Chapter 5 of the "Getting Used to R, RStudio, and R Markdown" book (**?**)).

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

```
## Warning: Removed 1 rows containing non-finite values (stat_bin).
```

We have the power to specify how many bins we would like to put the data into as an argument in the `geom_histogram` function. By default, this is chosen to be 30 somewhat arbitrarily and we have received a warning above our plot that this was done. We also notice here that another warning about 1 missing value is given. This value is omitted from the plot. This warning is ignored for future customizations of the plot.

Missing values are a common problem when working with data and there are entire fields of study on how to work with missing data. Frequently what is done is to remove instances from the data set that are missing. This is certainly the easy way to deal with them, but not necessarily the correct decision in all cases. For our purposes, we will usually ignore potential warnings about missing data since R can usually handle them by ignoring them.

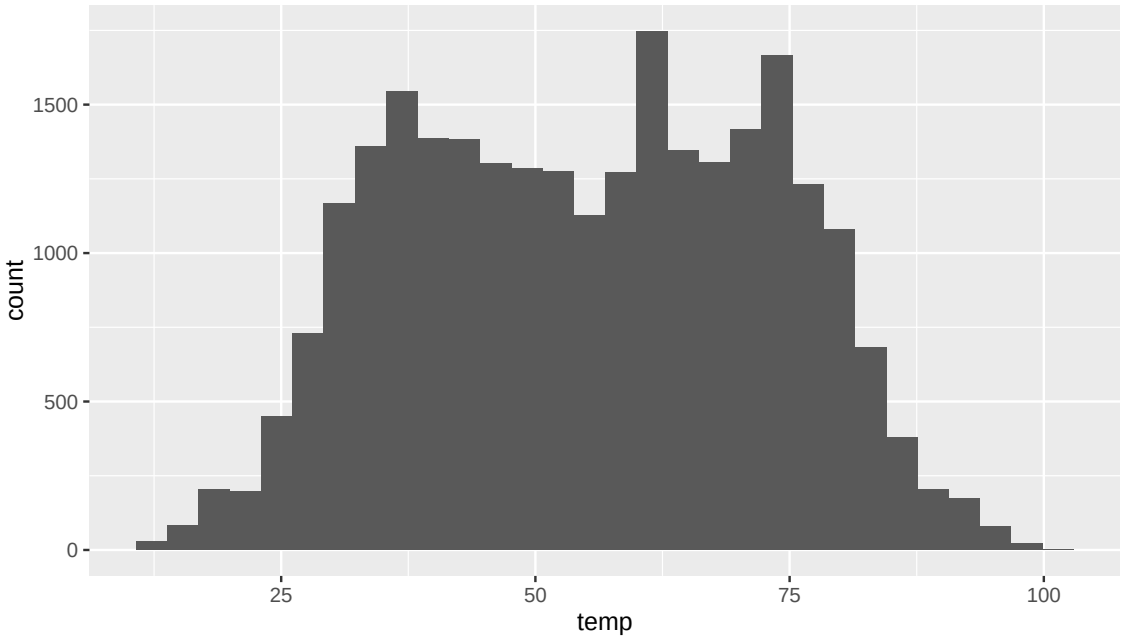



Figure 4.2: Histogram of Hourly Temperature Recordings from NYC in 2013

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(bins = 60)
```

We can tweak the plot a little more by specifying the width of the bins (instead of how many bins to divide the variable into) by using the `binwidth` argument in the `geom_histogram` function. We can also add some color to the plot by invoking the `fill` and `color` arguments. A listing of all of the built-in colors to R by name and color is available here.

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 10, color = "white", fill = "forestgreen")
```

---

***Learning check***

**(LC4.1)** What does changing the number of bins from 30 to 60 tell us about the distribution of temperatures?

**(LC4.2)** Would you classify the distribution of temperatures as symmetric or skewed?

**(LC4.3)** What would you guess is the "center" value in this distribution? Why did you make that choice?

**(LC4.4)** Is this data spread out greatly from the center or is it close? Why?

---

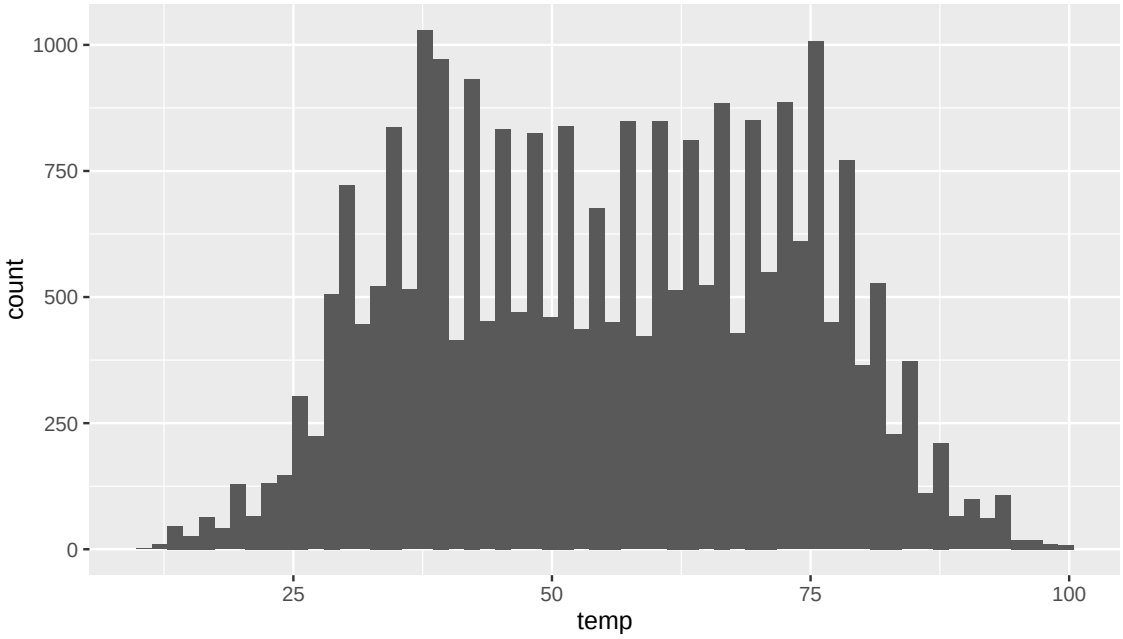



Figure 4.3: Histogram of Hourly Temperature Recordings from NYC in 2013 - 60 Bins

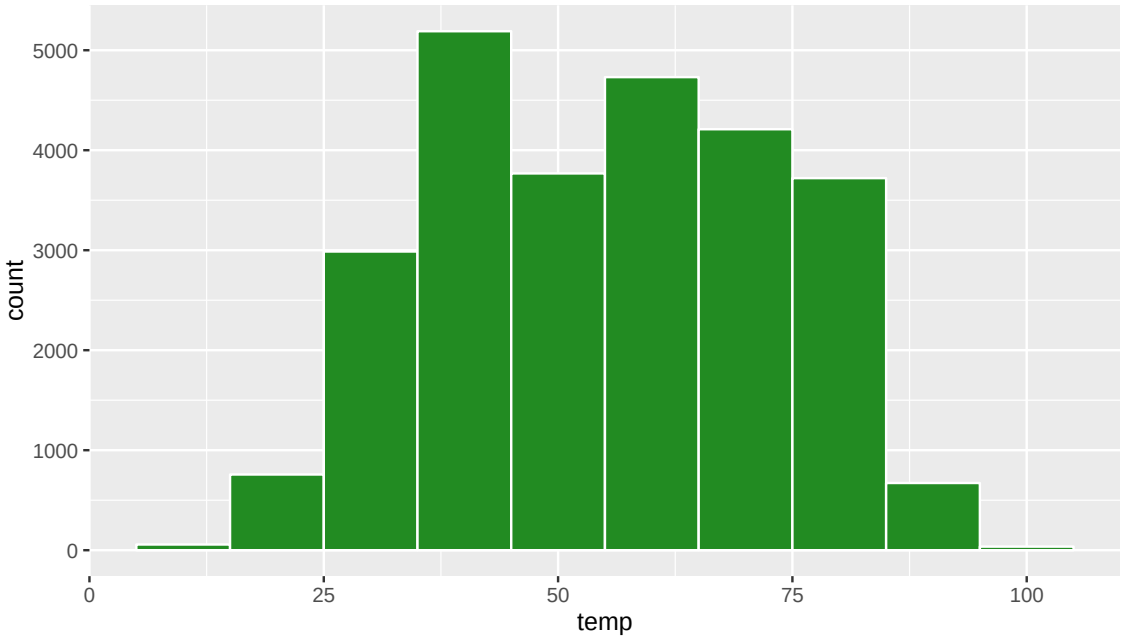



Figure 4.4: Histogram of Hourly Temperature Recordings from NYC in 2013 - Binwidth = 10

### *4.2.1 Continuous data summaries*

The `temp` variable is a **continuous** quantitative variable (frequently just called a **continuous variable**). "A variable is **continuous** if you can arrange its values in order and an infinite number of unique values can exist between any two values of the variable"(**?**). Some common examples of continuous variables are time and height. Between any two times there are an infinitely many number of time units that fall between them.

It is often easier to think about quantitative variables that are not continuous to help us better understand continuity. The best example is counts. If we are looking to count the number of flights that depart on a given day from New York City, this variable would not be continuous. It falls on a **discrete** scale.

We can examine some summary information about this `temp` variable. To do so, we introduce the `summary` function. (We will see in Chapter **??** how to use the `summarize` function in the `dplyr` package to produce similar results.)

The syntax here is a little different than what we have seen before. (A further discussion about R syntax is available in Chapter 5 of the "R basics" book (**?**)). Here, `summary` is the function and it is expecting an object to be summarized as its argument. The object here is the `temp` variable in the `weather` dataframe. To focus on just this one variable `temp` in `weather`, we separate them by the dollar sign symbol `$`. Order matters here: the dataframe comes before the `$` and the variable/column name comes after.

```
summary(weather$temp)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   10.94   39.92   55.04   55.20   69.98  100.00       1
```

This tells us what is known as the **five-number summary** for our variable as well as the **mean** value of the variable. More information on both of these concepts is given in Appendix B - Chapter **??**.

This `summary` gives us some numerical summaries of our temperature variable. The minimum recorded temperature is 10.94 degrees Fahrenheit and the maximum is 100.04 degrees Fahrenheit. We have one missing value denoted as an `NA` in the observations of this variable. The median Fahrenheit temperature of 55.04 and mean of 55.2035149 are quite close. This is a property of symmetric distributions.

The last two entries given by `summary` correspond to the $25^{\text{th}}$ percentile and the $75^{\text{th}}$ percentile. If we sorted all of the temperatures in increasing order, we would see that 25% of them would fall below 39.92 and that 75% of them would fall below 69.98. This implies that the middle 50% of data values lie between 39.92 and 69.98 degrees Fahrenheit.

Another common measure to determine the variability of a data set is the **standard deviation**. You can read further about the mathematical details of standard deviation in Appendix B - Chapter **??**, but you can think of it as being a measure of how far the data is, on average, from the mean. Let's investigate this further by calculating the standard deviation of our temperature variable:

```
sd(weather$temp)
```

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

Remember that there were some missing values in our temperature data when we plotted it earlier. By default, the `sd` function that computes standard deviation of a variable will give the value of `NA` if any of the data is missing. To further understand this, you can look at the help of the `sd` function in the R Console:

```
?sd
```

Under **Usage** you can see that `na.rm` is set to `FALSE` by default. We'd like for missing values to be removed from our analysis so we set that value to `TRUE`:

```
sd(weather$temp, na.rm = TRUE)
```

```
## [1] 17.78212
```

Standard deviation is always in the same units as our original data set. So in this case the standard deviation is in degrees Fahrenheit. Thus, we can interpret this value saying:

For a randomly selected element in our temperature column, we can expect it to be about 17.7821236 degrees Fahrenheit from our mean value of ‘55.2035149. By combining this knowledge with the plots above, we can have a good idea for whether this data is very spread out from its center or if it is tightly bunched.

### *4.2.2 Summary*

Histograms provide a useful way of looking at how ONE continuous variable varies. They allow us to answer questions such as

- Are there values far away from the center? These are commonly called **outliers** and can frequently be easily identified on a histogram.
- Are most values close to the center? If so, the spread of the variable is small. If not, the spread is large.
- How spread out are the values? One measure of this spread is **standard deviation** discussed above.

The histogram shows how many entries fall in different groupings of this variable. Another common property of distributions is symmetry and as we saw it is quite easily identified by looking over the histogram produced from the variable's values.

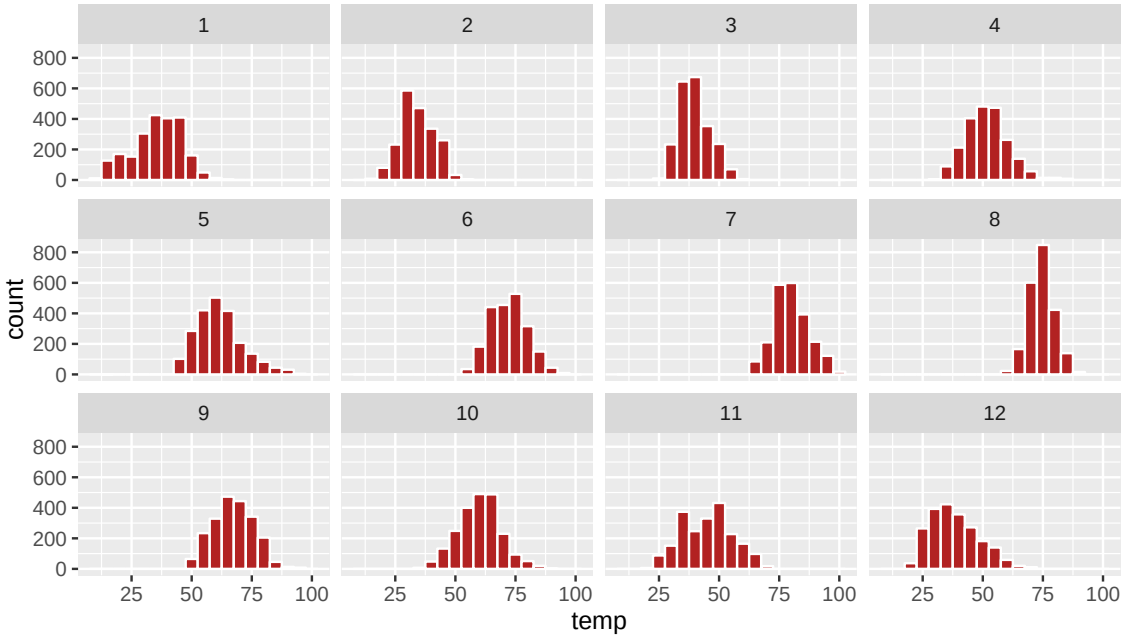
## *4.3 Boxplots*

Histograms can also be produced to compare the distribution of a variable over another variable. Suppose we were interested in looking at how the temperature recordings we saw in the last section varied by month. This is what is meant by "the distribution of a variable over another variable": `temp` is one variable and `month` is the other variable.

### *4.3.1 Faceting*

In order to look at histograms of `temp` for each month, we introduce a new concept called **facetting**. Faceting is used when we'd like to create small multiples of the same plot over a different categorical variable. By default, all of the small multiples will have the same vertical axis. An example will help here. We will discuss the concept of faceting in further detail in Section **??**.

```
ggplot(data = weather, mapping = aes(x = temp)) +
  geom_histogram(binwidth = 5, color = "white", fill = "firebrick") +
  facet_wrap(~month)
```




As we might expect, the temperature tends to increase as summer approaches and then decrease as winter approaches.

---

***Learning check***

**(LC4.5)** What other things do you notice about the faceted plot above? How does a faceted plot help us see how relationships between two variables?

**(LC4.6)** What do the numbers 1-12 correspond to in the plot above? What about 25, 50, 75, 100?

**(LC4.7)** What could be done to make the faceted plot above more readable? (Focus on tweaking the histograms and not on making a different type of plot here.)

**(LC4.8)** For which types of datasets would these types of faceted plots not work well in comparing relationships between variables? Draw or give an example.

**(LC4.9)** Does the `temp` variable in the `weather` data set have a lot of variability? Why do you say that?

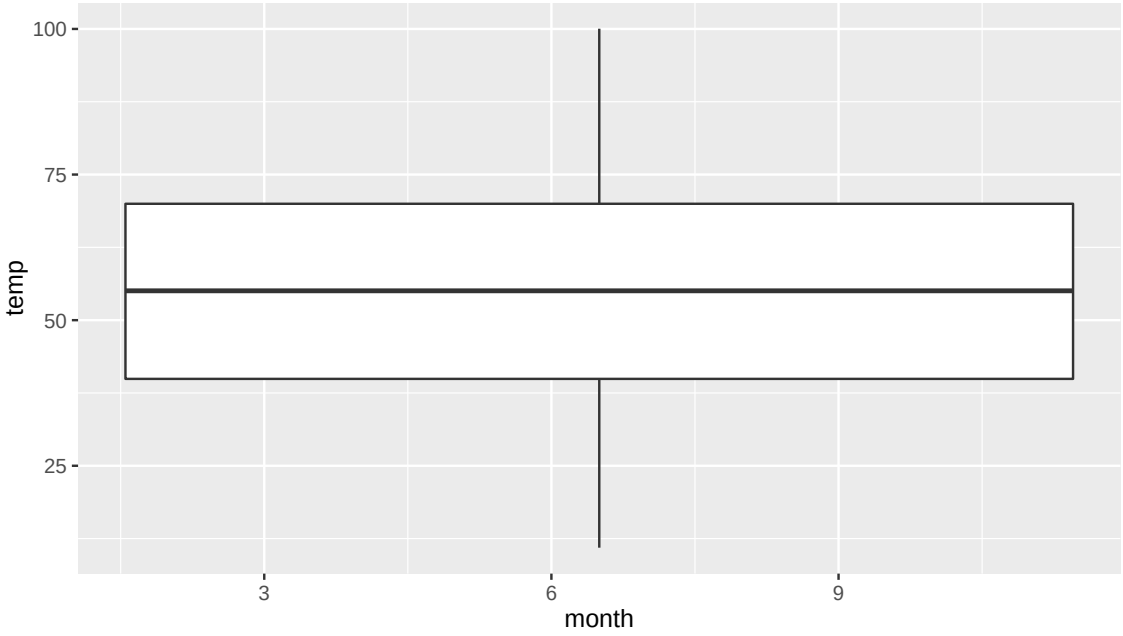
---

Histograms can provide a way to compare distributions across groups as we see above when we looked at temperature over months. Frequently, a plot called a **boxplot** (also called a **side-by-side boxplot**) is done instead. The **boxplot** uses the information provided in the **five-number summary** referred to in the previous section when we used the `summary` function. It gives a way to compare this summary information across the different levels of a group. Let's create a boxplot to compare the monthly temperatures as we did above with the faceted histograms.

```
ggplot(data = weather, mapping = aes(x = month, y = temp)) +
  geom_boxplot()
```

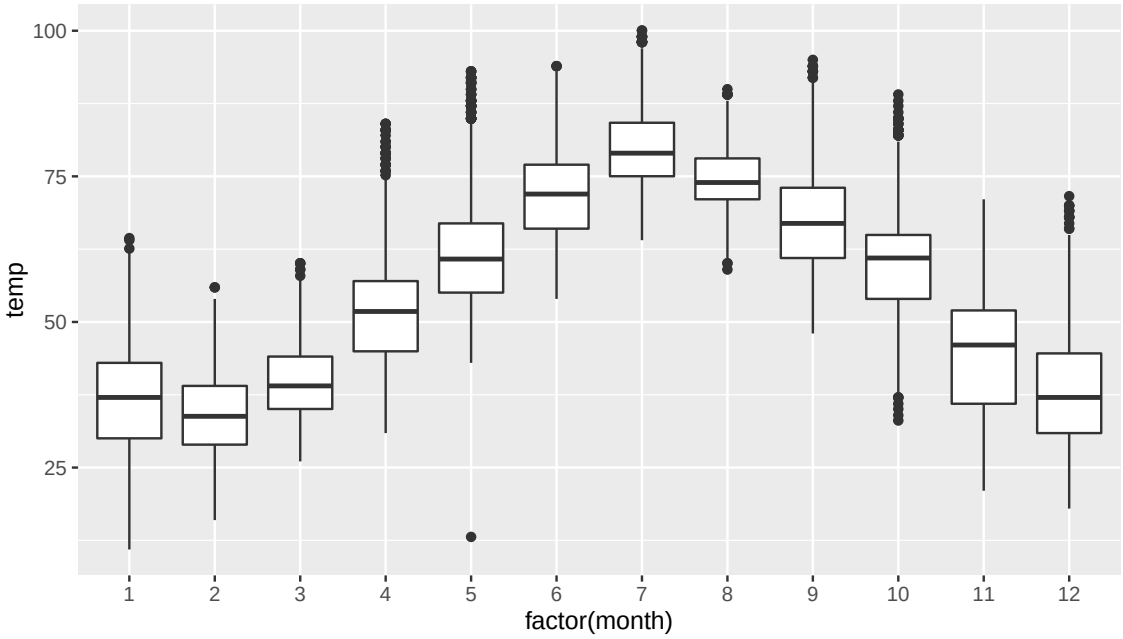
```
## Warning: Continuous x aesthetic -- did you forget aes(group=...)?
```

```
## Warning: Removed 1 rows containing non-finite values (stat_boxplot).
```




Note the first warning that is given here. (The second one corresponds to missing values in the dataframe and it is turned off on subsequent plots.) This plot does not look like what we were expecting. We were expecting to see the distribution of temperatures for each month (so 12 different boxplots). This gives us the overall boxplot without any other groupings. We can get around this by introducing a new function for our `x` variable.

```
ggplot(data = weather, mapping = aes(x = factor(month), y = temp)) +
  geom_boxplot()
```




We have introduced a new function called `factor()` here. One of the things this function does is to convert a numeric value like `month` (1, 2, ..., 12) into a categorical variable. The "box" part of this plot represents the $25^{\text{th}}$ percentile, the median ($50^{\text{th}}$ percentile), and the $75^{\text{th}}$ percentile. The dots correspond to **outliers**. (The specific formulation for these outliers is discussed in Appendix B - Chapter **??**.) The lines show how the data varies that is not in the center 50% defined by the first and third quantiles. Longer lines correspond to more variability and shorter lines correspond to less variability.

---

***Learning check***

**(LC4.10)** What does the dot at the bottom of the plot for May correspond to? Explain what might have occurred in May to produce this point.

**(LC4.11)** Which months have the highest variability in temperature? What reasons do you think this is?

**(LC4.12)** We looked at the distribution of a continuous variable over a categorical variable here with this boxplot. Why can't we look at the distribution of one continuous variable over the distribution of another continuous variable? Say temperature across pressure, for example?

**(LC4.13)** Boxplots provide a simple way to identify outliers. Why may outliers be easier to identify when looking at a boxplot instead of a faceted histogram?

### *4.3.2 Summary*

Boxplots provide a way to compare and contrast the distribution of ONE quantitative variable across multiple levels of ONE categorical variable. One can easily look to see where the median falls across the different groups by looking at the center line in the box. You can also see how spread out the variable is across the different groups by looking at the width of the box and also how far out the lines stretch from the box. Lastly, outliers are even more easily identified when looking at a boxplot than when looking at a histogram.

## *4.4 Barplots*

Both histograms and boxplots represent ways to visualize the variability of continuous variables. Another common task is to present the distribution of a categorical variable. This is a simpler task since we will be interested in how many elements from our data fall into the different categories of the categorical variable. We need not bin the data or identify the different quantiles for categorical variables.

Frequently, the best way to visualize these different counts (also known as **frequencies**) is via a barplot. Consider the distribution of airlines that flew out of New York City in 2013. This can be plotted by invoking the `geom_bar` function in `ggplot2`:

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar()
```

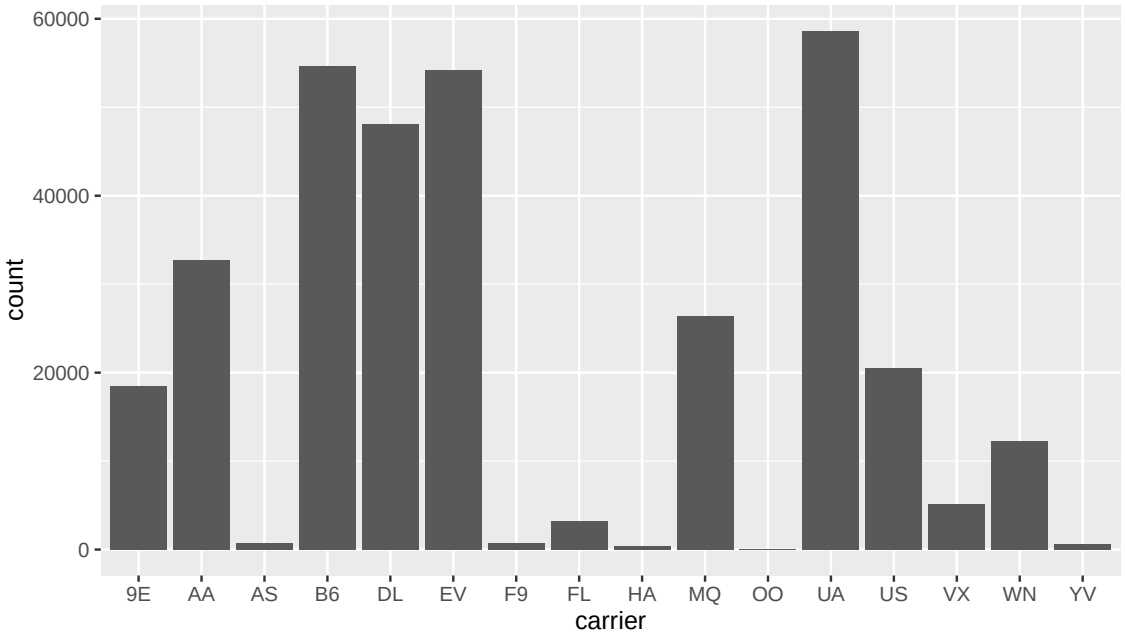



Figure 4.5: Number of flights departing NYC in 2013 by airline

Recall the `airlines` dataset discussed in Chapter **??**.

```
data(airlines)
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 United Air Lines, JetBlue Airways, and ExpressJet Airlines had the most flights depart New York City in 2013. To get the actual number of flights by each airline we can use the `count` function in the `dplyr` package on the `carrier` variable in `flights`:

```
library(dplyr)
flights_table <- count(x = flights, vars = carrier)
flights_table
```

```
## # A tibble: 16 x 2
##     vars     n
##    <chr> <int>
## 1     9E 18460
## 2     AA 32729
## 3     AS   714
## 4     B6 54635
## 5     DL 48110
## 6     EV 54173
## 7     F9   685
## 8     FL  3260
## 9     HA   342
## 10    MQ 26397
## 11    OO    32
```

```
## 12    UA 58665
## 13    US 20536
## 14    VX  5162
## 15    WN 12275
## 16    YV   601
```

---

***Learning check***

**(LC4.14)** Why are histograms inappropriate for visualizing categorical variables?
**(LC4.15)** What is the difference between histograms and barplots?
**(LC4.16)** How many Envoy Air flights departed NYC in 2013?
**(LC4.17)** What was the seventh highest airline in terms of departed flights from NYC in 2013?

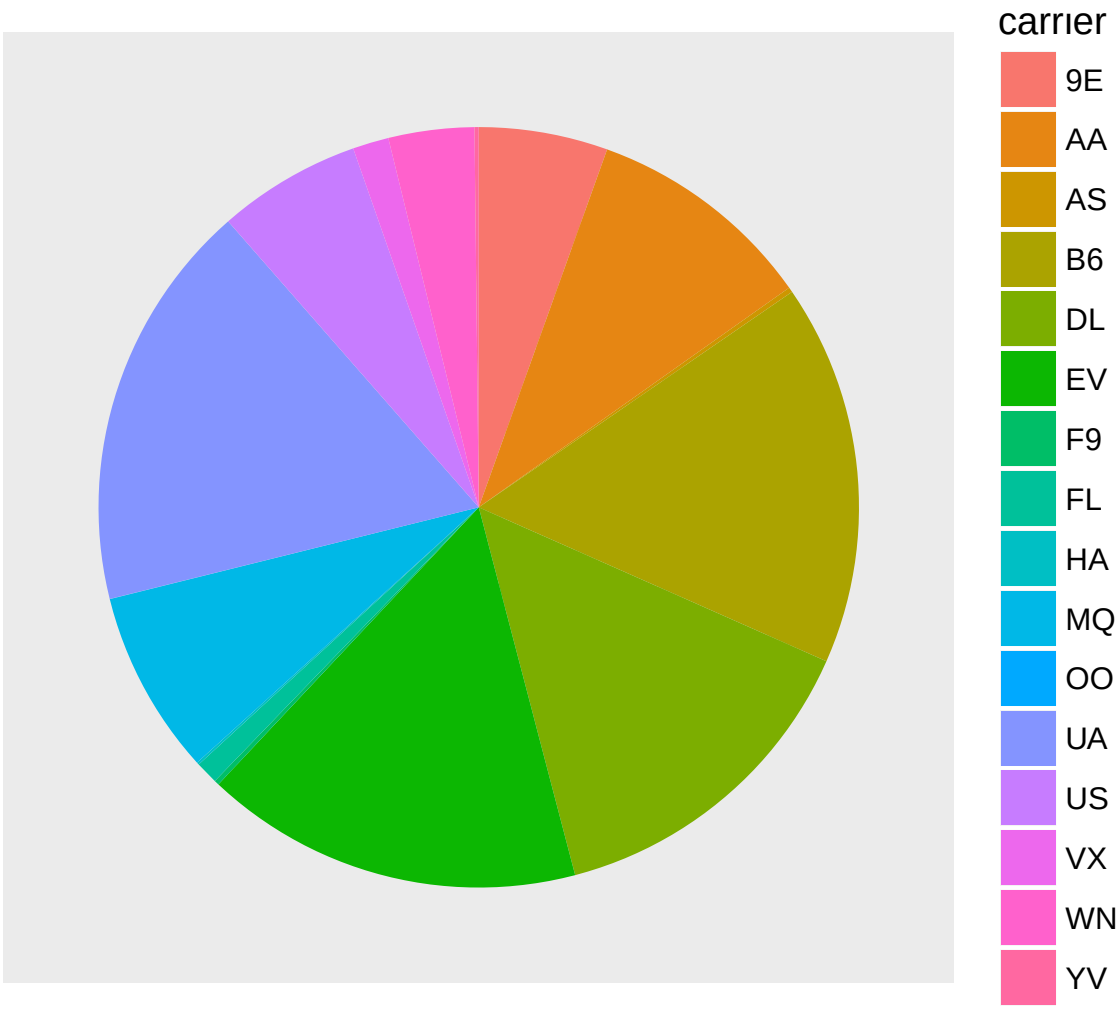
---

### *4.4.1 Must avoid pie charts!*

Unfortunately, one of the most common plots seen today for categorical data is the pie chart. While they may see harmless enough, they actually present a problem in that humans are unable to judge angles well. As Naomi Robbins describes in her book "Creating More Effective Graphs", we overestimate angles greater than 90 degrees and we underestimate angles less than 90 degrees. In other words, it is difficult for us to determine relative size of one piece of the pie compared to another.

Let's examine our previous barplot example on the number of flights departing NYC by airline. This time we will use a pie chart. As you review this chart, try to identify

- how much larger the portion of the pie is for ExpressJet Airlines (`EV`) compared to US Airways (`US`),
- what the third largest carrier is in terms of departing flights, and
- how many carriers have fewer flights than United Airlines (`UA`)?




While it is quite easy to look back at the barplot to get the answer to these questions, it's quite difficult to get the answers correct when looking at the pie graph. Barplots can always present the information in a way that is easier for the eye to determine relative position. There may be one exception from Nathan Yau at FlowingData.com but we will leave this for the reader to decide:


---

***Learning check***

**(LC4.18)** Why should pie charts be avoided and replaced by barplots?
**(LC4.19)** What is your opinion as to why pie charts continue to be used?

---

### *4.4.2 Using barplots to compare two variables*

Barplots are the go-to way to visualize the frequency of different categories of a categorical variable. They make it easy to order the counts and to compare one group's frequency to another. Another use of barplots (unfortunately, sometimes inappropriately and confusingly) is to compare two categorical variables together. Let's examine the distribution of outgoing flights from NYC by `carrier` and `airport`.

We begin by getting the names of the airports in NYC that were included in the `flights` dataset. Remember from Chapter **??** that this can be done by using the `inner_join` function in the `dplyr` package.

```
library(dplyr)
flights_namedports <- inner_join(flights, airports, by = c("origin" = "faa"))
```

After running `View(flights_namedports)`, we see that `name` now corresponds to the name of the airport as referenced by the `origin` variable. We will now plot `carrier` as the horizontal variable. When we specify `geom_bar`, it will specify `count` as being the vertical variable. A new addition here is `fill = name`. Look over what was produced from the plot to get an idea of what this argument gives.

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar()
```

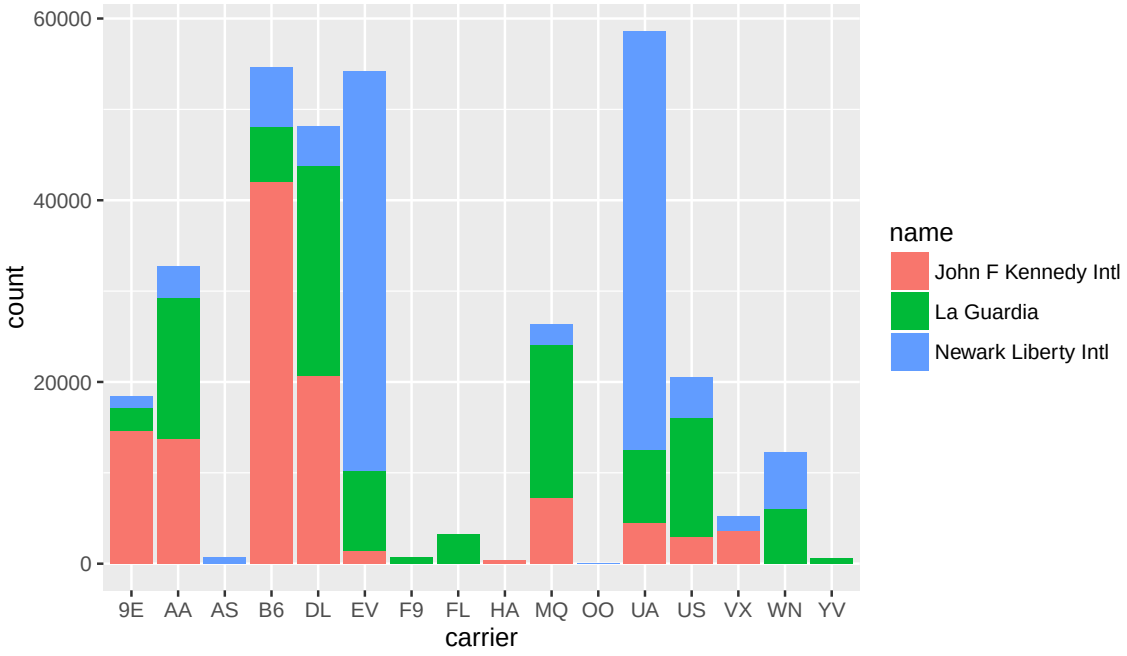



Figure 4.6: Stacked barplot comparing the number of flights by carrier and airport

This plot is what is known as a **stacked barplot**. While simple to make, it often leads to many problems.

***Learning check***

**(LC4.20)** What kinds of questions are not easily answered by looking at the above figure?

**(LC4.21)** What can you say, if anything, about the relationship between airline and airport in NYC in 2013 in regards to the number of departing flights?

---

Another variation on the **stacked barplot** is the **side-by-side barplot**.

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar(position = "dodge")
```

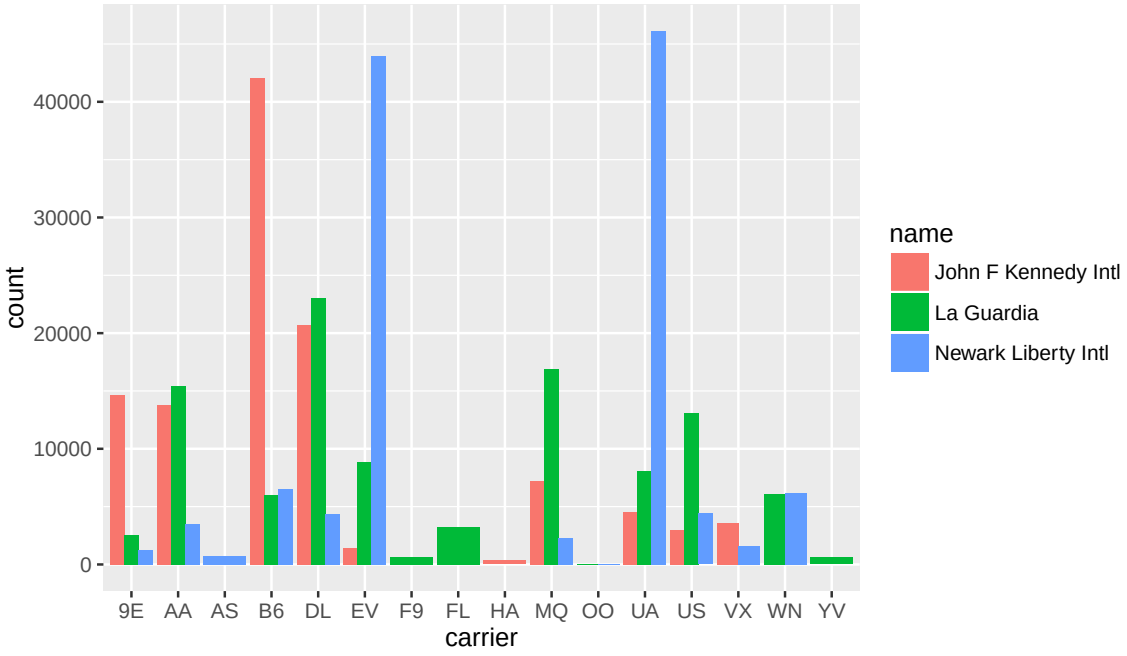



Figure 4.7: Side-by-side barplot comparing the number of flights by carrier and airport

---

***Learning check***

**(LC4.22)** Why might the side-by-side barplot be preferable to a stacked barplot in this case?

**(LC4.23)** What are the disadvantages of using a side-by-side barplot, in general?

---

Lastly, an often preferred type of barplot is the **faceted barplot**. We already saw this concept of faceting and small multiples in Subsection **??**. This gives us a nicer way to compare the distributions across both `carrier` and airport/`name`.

```
ggplot(data = flights_namedports, mapping = aes(x = carrier, fill = name)) +
  geom_bar() +
  facet_grid(name ~ .)
```

Note how the `facet_grid` function arguments are written here. We are wanting the names of the airports vertically and the `carrier` listed horizontally. As you may have guessed, this

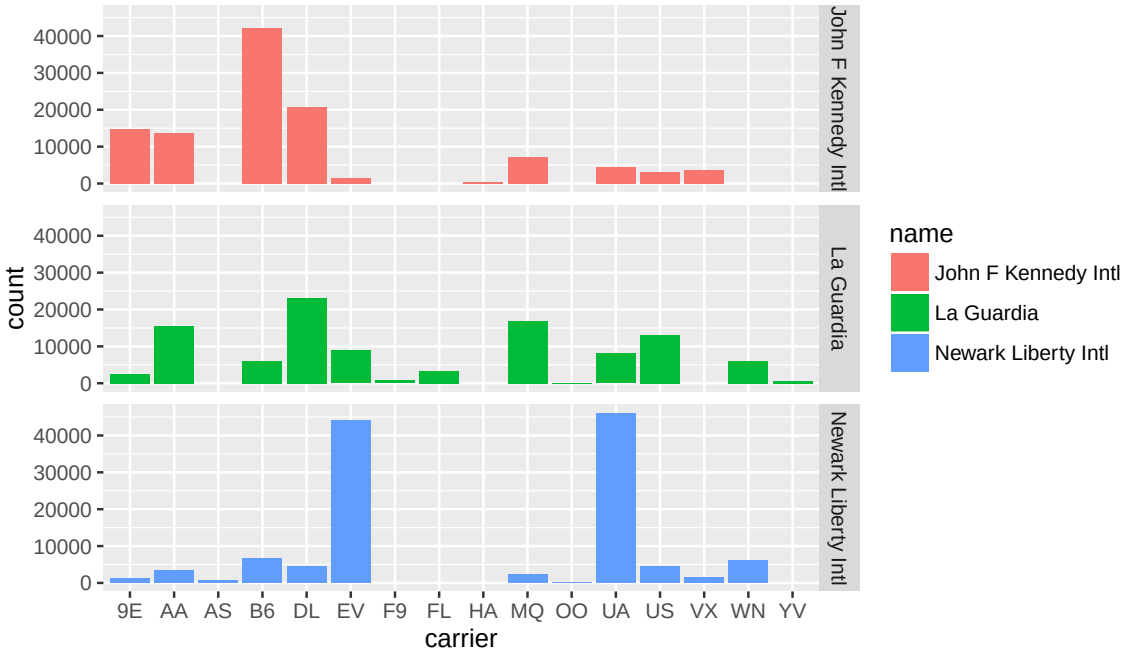



Figure 4.8: Faceted barplot comparing the number of flights by carrier and airport

argument and other *formulas* of this sort in R are in `y ~ x` order. We will see more examples of this in Chapter **??**.

---

***Learning check***

**(LC4.24)** Why is the faceted barplot preferred to the side-by-side and stacked barplots in this case?

**(LC4.25)** What information about the different carriers at different airports is more easily seen in the faceted barplot?

---

### *4.4.3 Summary*

Barplots are the preferred way of displaying categorical variables. They are easy-to-understand and to make comparisons across groups of a categorical variable. When dealing with more than one categorical variable, faceted barplots are frequently preferred over side-by-side or stacked barplots. Stacked barplots are sometimes nice to look at, but it is quite difficult to compare across the levels since the sizes of the bars are all of different sizes. Side-by-side barplots can provide an improvement on this, but the issue about comparing across groups still must be dealt with.

## *4.5 Scatter-plots*

We have seen that boxplots are most appropriate when plotting the distribution of ONE continuous variable across different levels/groups of ONE categorical variable. Barplots (preferably the faceted type) are best when looking at the distribution of ONE categorical variable across different levels of another categorical variable. But what if we are looking to investigate the relationship between TWO continuous variables? What is commonly produced is the well-known **scatter-plot**, which shows the points corresponding to the values of each of the variables scattered around.

We will now investigate arrival delays (the vertical "y" axis variable) versus departure delays (the horizontal "x" axis variable) for Alaska Airlines flights leaving NYC in 2013. Notice the new function that is invoked here: `filter`, which resides in the `dplyr` package. You will see many more examples using this function in Chapter **??**. The `filter` function goes through the dataframe specified (`flights` here) and selects only those rows which meet the condition given (`carrier == "AS"` here).

```
alaska_flights <- filter(flights, carrier == "AS")
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_point()
```

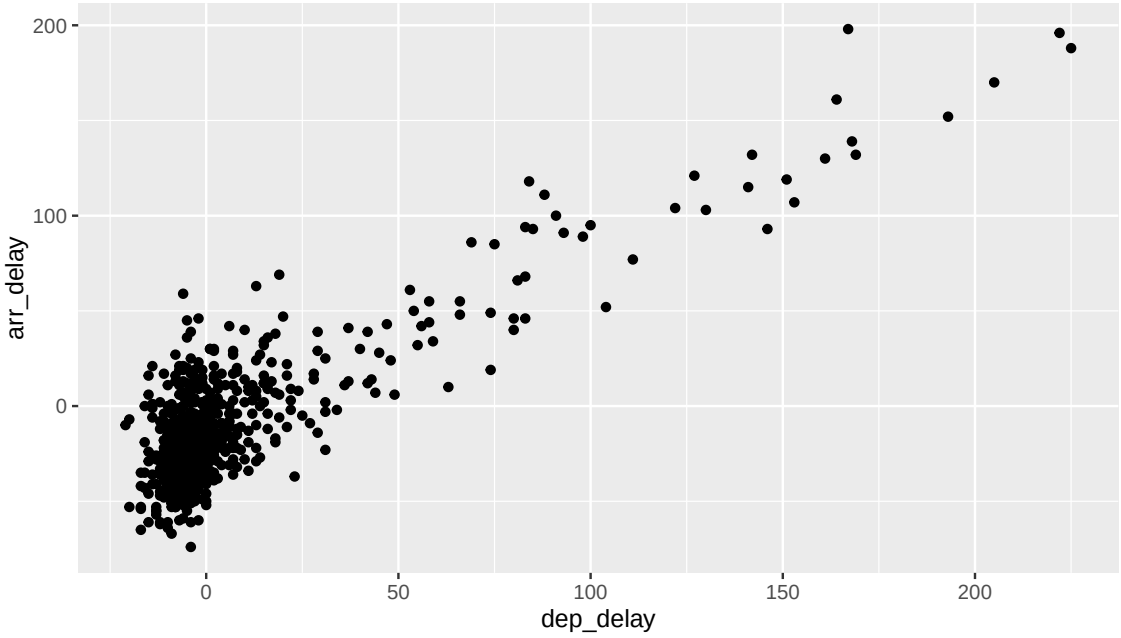



Figure 4.9: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013

We see that a positive relationship exists between `dep_delay` and `arr_delay`: as departure delays increase, arrival delays tend to also increase. We also note that the majority of points fall near the point (0, 0) here. There is a large mass of points clustered there.

***Learning check***

**(LC4.26)** What are some practical reasons why `dep_delay` and `arr_delay` have a positive relationship?

**(LC4.27)** What variables (not necessarily in the `flights` dataframe) would you expect to have a negative correlation (i.e. a negative relationship) with `dep_delay`? Why? Remember that we are focusing on continuous variables here.

**(LC4.28)** Why do you believe there is a cluster of points near (0, 0)?

- What does (0, 0) correspond to in terms of the Alaskan flights?

**(LC4.29)** What are some other features of the plot that stand out to you?

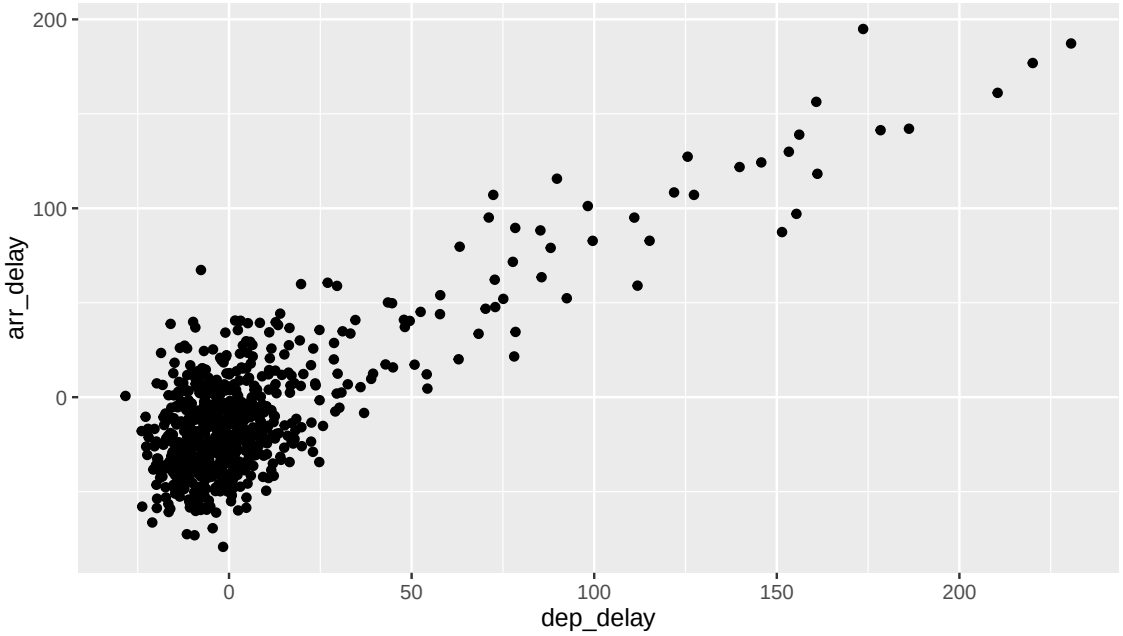
---

### *4.5.1 Jittering*

The large mass of points near (0, 0) can cause some confusion. This is the result of a phenomenon called **over-plotting**. As one may guess, this corresponds to values being plotted on top of each other *over* and *over* again. It is often difficult to know just how many values are plotted in this way when looking at a basic scatter-plot as we have here.

One way of relieving this issue of **over-plotting** is to **jitter** the points a bit. In other words, we are going to add just a bit of random noise to the points to better see them and remove some of the over-plotting. You can think of "jittering" as shaking the points a bit on the plot. Instead of using `geom_point`, we use `geom_jitter` to perform this shaking and specify around how much jitter to add with the `width` and `height` arguments. This corresponds to how hard you'd like to shake the plot in units corresponding to those for both the horizontal and vertical variables (minutes here).

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30)
```




This has helps us a little bit in getting a sense for the over-plotting, but with a relatively large dataset like this one (714 flights), it is often useful to change the transparency of the points as seen in the next section.

### *4.5.2 Setting transparency*

One of the arguments that can be changed with `geom_point` is `alpha`. By default, this value is set to `1`. We can change this value to a smaller fraction to change the transparency of the points in the plot:

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_point(alpha = 0.2)
```

We can also specify the `alpha` argument in `geom_jitter`:

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +
  geom_jitter(width = 30, height = 30, alpha = 0.3)
```

---

***Learning check***

**(LC4.30)** Why is setting the `alpha` argument value useful with scatter-plots?

- What further information does it give you that a regular scatter-plot cannot?

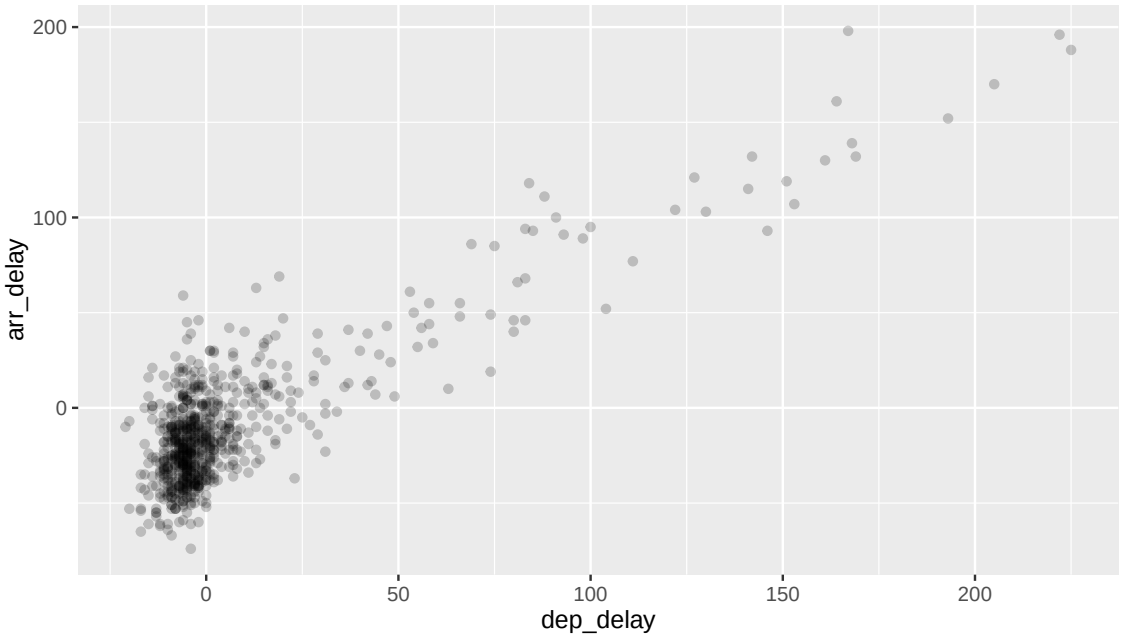



Figure 4.10: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013 - alpha=0.2

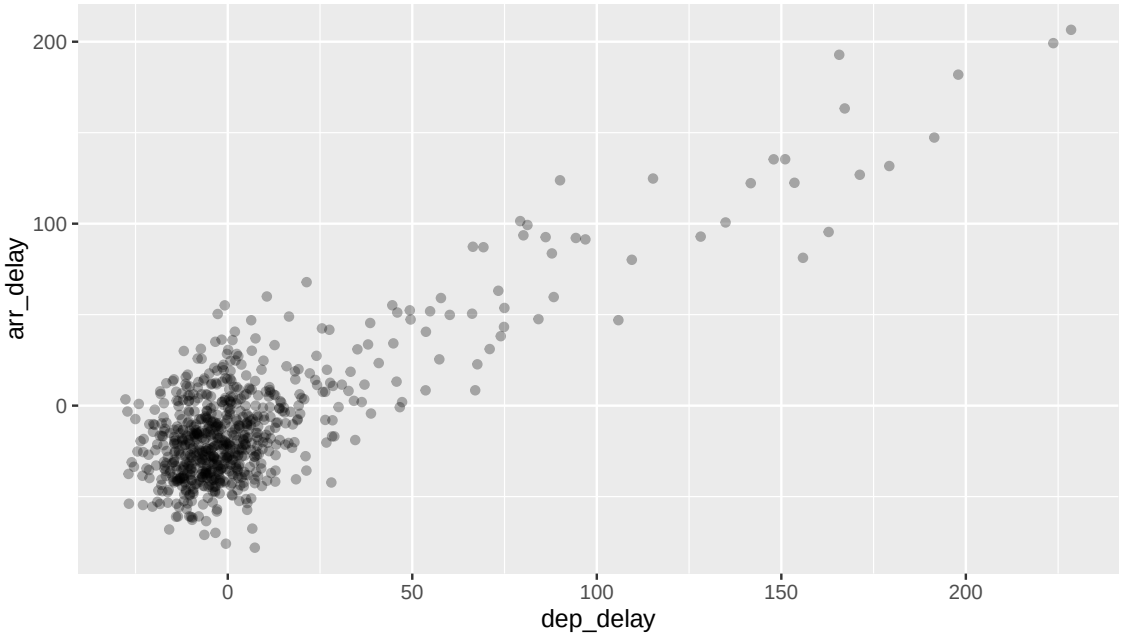



Figure 4.11: Arrival Delays vs Departure Delays for Alaska Airlines flights from NYC in 2013 - jitter and alpha added

**(LC4.31)** After viewing the **??** above, give a range of arrival times and departure times that occur most frequently?

- How has that region changed compared to when you observed the same plot without the `alpha = 0.2` set in **??**?

---

**Maybe include a shading of the points by another variable example here for multivariate thinking?**

### *4.5.3 Summary*

Scatter-plots may be the most used plot today and they can provide an immediate way to see the trend in one variable versus another. Remember that they only make sense when plotting a continuous variable versus a continuous variable though. If you try to create a scatter-plot where either one of the two variables is not quantitative, you will get strange results. Be careful!

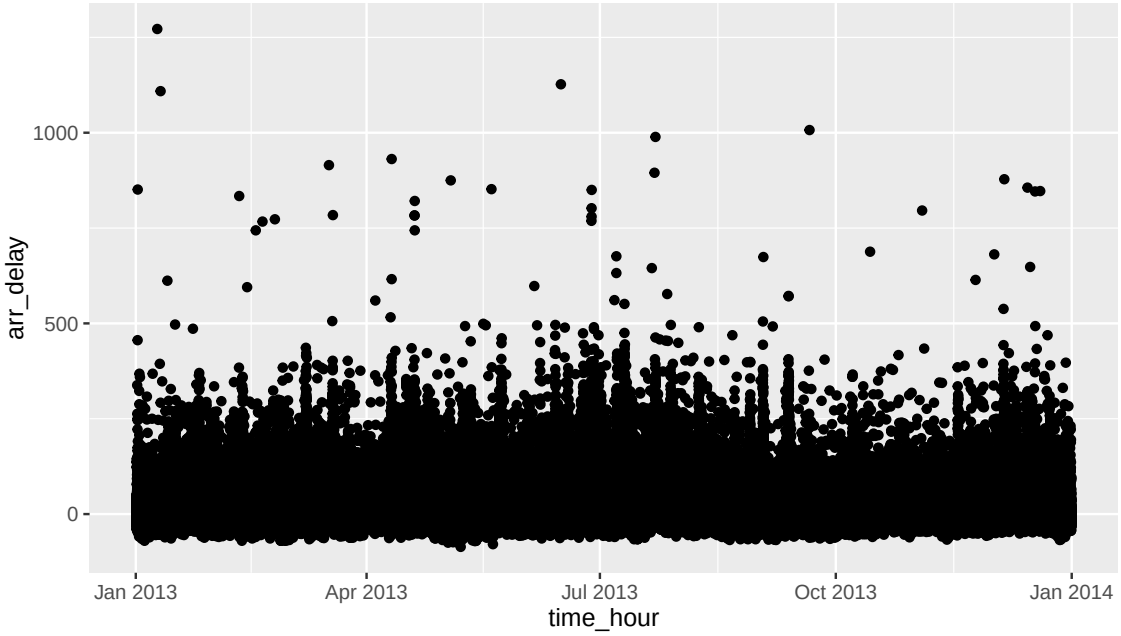
With medium to large datasets, you may need to tweak arguments in both `geom_jitter` and the `alpha` parameter in order to get a good feel for relationships in your data. This tweaking is often a fun part of data visualization since you'll have the chance to see different relationships come about as you make subtle changes to your plots.

## *4.6 Line-graphs*

The last of the FNG is a line-graph. They are most frequently used when the horizontal axis is time. Time represents a variable that is connected together by each day following the previous day. In other words, time has a natural ordering. Line-graphs should be avoided when there is not a clear ordering to the explanatory ("x" variable).

We are interested in exploring the arrival delays by day throughout the year of 2013 from outgoing flights from New York City. If we plotted all of these values, we obtain the following scatter-plot:

```
ggplot(flights, aes(x = time_hour, y = arr_delay)) +
  geom_point()
```




We see that this plot is difficult to understand based on the sheer number of points plotted. We see some outlier points with more than 500 minutes of arrival delay, but even some jittering and transparency is not going to help us here.

Instead of plotting all of the values for each hour for all flights, it might make sense to plot the average value for each day in terms of arrival delays. Of course, we also need to address which average we should use: mean or median. With there being some outliers here, we have chosen to use the median arrival delay (since the mean is heavily influence by outliers).

You may think that this is a difficult task but the `group_by` and `summarize` functions make this a breeze. You'll see more examples using these two functions in Chapter **??**. Here we will create a new variable, which corresponds to the month and day combined, from the `time_hour` column using the `mutate` function and create a new dataframe called `flights_day`. Notice from running `View(flights_day)` that the new variable added called `date` appears on the far right of the dataset.

```
flights_day <- mutate(flights, date = as.Date(time_hour))
```

```
flights_summarized <- flights_day %>% group_by(date) %>%
  summarize(median_arr_delay = median(arr_delay, na.rm = TRUE))
flights_summarized
```

```
## # A tibble: 365 x 2
##          date median_arr_delay
##        <date>            <dbl>
## 1  2013-01-01                3
## 2  2013-01-02                4
## 3  2013-01-03                1
```

```
## 4  2013-01-04               -8
## 5  2013-01-05               -7
## 6  2013-01-06               -1
## 7  2013-01-07              -10
## 8  2013-01-08               -7
## 9  2013-01-09               -6
## 10 2013-01-10              -11
## # ... with 355 more rows
```

You will see the "pipe" operator `%>%` explained in more detail in Chapter **??**, but you can read it as "and then". Here, we take the `flights_day` dataframe that we just created and then group it together by `date`. This goes through the dataframe and puts together all rows that have `2013-01-01` together, all rows that have `2013-01-02` together, ..., and all rows that have `2013-12-31` together. And then it looks at the median value of `arr_delay` over each one of the days. You can get a glimpse of the first few rows of this new dataset above since we invoked the `head` function on it.

Note also that there are missing values in this data set so we need to exclude them from the analysis. This is why the `na.rm = TRUE` argument is invoked. Many functions require this extra specification so it's always a good idea to run a `?median` or `?mean` before you try to run the function. Or you can always run it afterwards as well when you get strange results.

Now getting back to our line-graph. We want to plot the median arrival delay over all airlines on all days in 2013 from departing flights in NYC. This syntax should look similar to what we have seen before with plots involving `ggplot`. Notice that we are using the `flights_summarized` dataset here and not the `flights_day` or `flights` dataframes.

```
ggplot(data = flights_summarized, aes(x = date, y = median_arr_delay)) +
  geom_line()
```

### *4.6.1 Interactive line-graphs*

Another useful tool for viewing line-graphs such as this is the `dygraph` function in the `dygraphs` package in combination with the `dyRangeSelector` function. This allows us to zoom in on a selected range and get an interactive plot for us to work with:

```
library(dygraphs)
rownames(flights_summarized) <- flights_summarized$date
flights_summarized <- select(flights_summarized, -date)
dyRangeSelector(dygraph(flights_summarized))
```

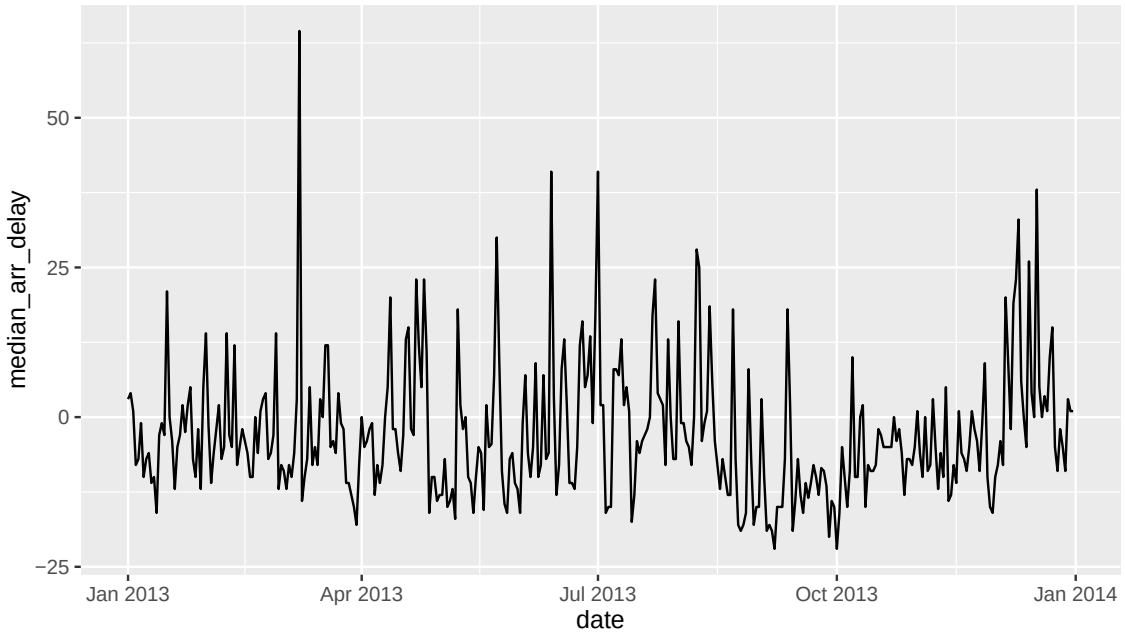
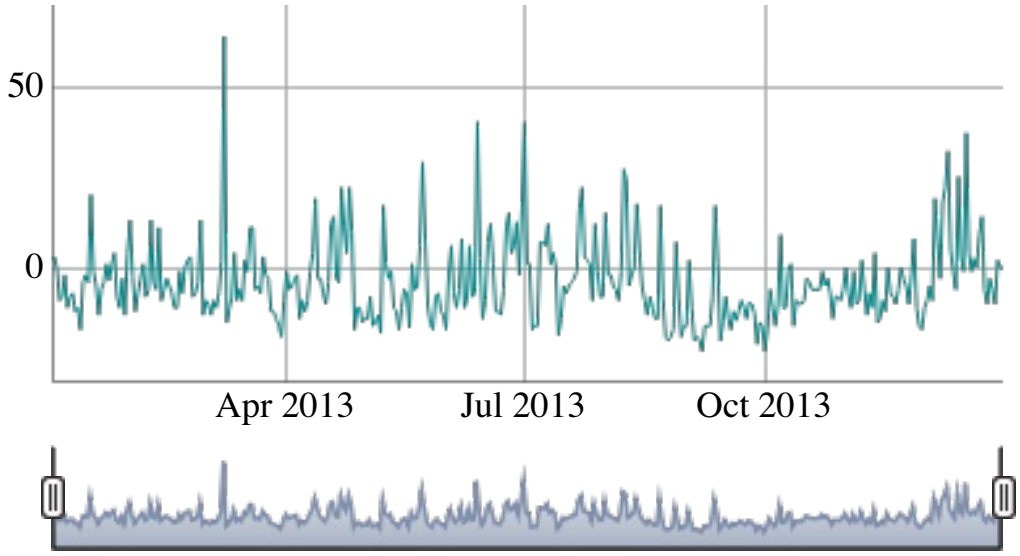



Figure 4.12: Line-graph of median arrival delay for flights leaving NYC in 2013 versus day of the year




The syntax here is a little different than what we have covered so far. The `dygraph` function is expecting for the dates to be given as the `rownames` of the object. We then remove the `date` variable from the `flights_summarized` dataframe since it is accounted for in the `rownames`. Lastly, we run the `dygraph` function on the new dataframe that only contains the median arrival delay as a column and then provide the ability to have a selector to zoom in on the interactive plot via `dyRangeSelector`. (Note that this plot will only be interactive in the HTML version of this book.)

***Learning check***

**(LC4.32)** Why should line-graphs be avoided when there is not a clear ordering of the horizontal axis?

**(LC4.33)** Why are line-graphs frequently used when time is the explanatory variable?

**(LC4.34)** Why did we use the `flights_summarized` dataframe to produce the line-graph in Figure **??** instead of `flights` or `flights_day`?

**(LC4.35)** Are the largest median arrival delays where you expected them to occur on the line-graph above in Figure **??**?

**(LC4.36)** Use the interactive line-graph to determine the highest median arrival delay for flights from NYC in 2013. What date was it and what do you think contributed to it?

---

### *4.6.2 Summary*

Line-graphs provide a useful tool for viewing a continuous variable that is plotted versus time. We need to be careful to not be too entrenched in using line-graphs whenever we wish though. They only make sense when the explanatory variable (the one on the explanatory variable) has a natural ordering. We can mislead our audience if that isn't the case.

## *4.7 Brief Review of The Grammar of Graphics*

The FNG discussion above has introduced you to all of the major pieces behind "The Grammar of Graphics", which serves as the basis for the `ggplot2` package. This theoretical framework given by Leland Wilkinson (**?**) helps us identify what the pieces are that make up a statistical graphic:

> In brief, the grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (color, shape, size) of geometric objects (points, lines, bars).
>
> Specially, we can break a graphic into the following components:

- `aes`: mappings of data to *aesthetics* we can perceive on a graphic. These include x/y position, color, size, and shape. Each aesthetic can be mapped to a variable in our data set. If not assigned, they are set to defaults.
- `geom`: (geometric objects) This refers to our type of plot: points, lines, bars, etc.
- `stat`: (statistical transformations to summarize data) This includes smoothing, binning values into a histogram, or just itself `"identity"`. You'll see this when we get to regression later in Chapter **??**.
- `facet`: how to break up data into subsets and display broken down plots as small multiples
- `scales` both

  - convert **data units** to **physical units** the computer can display
  - draw a legend and/or axes, which provide an inverse mapping to make it possible to read the original data values from the graph.

- `coord`: coordinate system for x/y values: typically cartesian.
- `position` adjustments

There are other extra attributes that can be tweaked as well including titles for the plot and each of the axes and also over-arching themes for the plot. In general, the Grammar of Graphics allows for customizability but also keeping with a consistent framework that allows the user to easily tweak their creations as they wish or need to in order to convey a message about their data.

We'll see (and have seen) that you don't necessarily need to include all of these in your code to produce a plot but each of the components are set by default and do exist with each plot produced using `ggplot2`.

An excellent resource as you begin to create plots using the `ggplot2` package is a cheatsheet that RStudio has put together entitled "Data Visualization with ggplot2" available here. This covers more than what we've discussed in this chapter but provides nice visual descriptions of what each function produces.

## *4.8 What's to come?*

In Chapter **??**, we'll further explore data by grouping our data, creating summaries based on those groupings, filtering our data to match conditions, selecting specific columns of our data, and other manipulations with our data including defining new columns/variables. These data manipulation procedures will go hand-in-hand with the data visualizations you've produced here.

# 5
# *Manipulating Data*

Let's briefly recap where we have been so far and where we are headed. In Chapter **??**, we discussed what it means for data to be tidy. We saw that this corresponds to observations to correspond to rows and for variables to be stored in columns. The entries in the data frame correspond to different combinations of observational units and variables. In the `flights` data frame, we saw that each row corresponded to a different flight leaving New York City. (In other words, the observational unit of that tidy data frame is a flight.) The variables are listed as columns and for `flights` they include both quantitative variables like `dep_delay` and `distance` but also categorical variables like `carrier` and `origin`. An entry in the table corresponds to a particular flight and a particular value of a given variable representing that flight.

We saw in Chapter **??** that organizing data in this tidy way makes it easy for use to produce graphics. We can simply specify what variable/column we would like on one axis, what variable we'd like on the other axis, and what type of plot we'd like to make. We can also do things such as changing the color by another variable or change the size of our points by a fourth variable given this tidy data set.

In Chapter **??**, we also introduced some ways to summarize and manipulate data to suit your needs. This chapter focuses more on the details of this by giving a variety of examples using the five main verbs in the `dplyr` package. There are more advanced operations that can be done than these and you'll see some examples of this near the end of the chapter.

As we saw with the RStudio cheatsheet on data visualization, RStudio has also created a cheatsheet for data manipulation entitled "Data Wrangling with dplyr and tidyr" available here. We will focus only on the `dplyr` functions in this book, but you are encouraged to also explore `tidyr` if you are presented with data that is not in the tidy format that we have specified as the preferred option for our purposes.

## *5.1 Five Main Verbs - The FMV*

If you scan over the Data Wrangling cheatsheet, you may be initially overwhelmed by the amount of functions available. You'll see the use of all of these as you work more and more with data frames in R. The `d` in `dplyr` stands for data frames so the functions here work when you are working with objects of the data frame type.

It's most important for you to focus on the five most commonly used functions that help

us manipulate and summarize data. A description of these verbs follows with each subsection devoted to seeing an example of that verb in play (or a combination of a few verbs):

- `select`: Choose variables/columns by their names
- `filter`: Pick rows based on conditions about their values
- `summarize`: Create summary measures of variables (or groups of observations on variables using `group_by`)
- `mutate`: Make a new variable in the data frame
- `arrange`: Sort the rows based on one or more variables

Just as we had the FNG (The Five Named Graphs in Chapter **??** using `ggplot2`), we have the FMV here (The Five Main Verbs in `dplyr`):

### *5.1.1 Select variables using* `select`

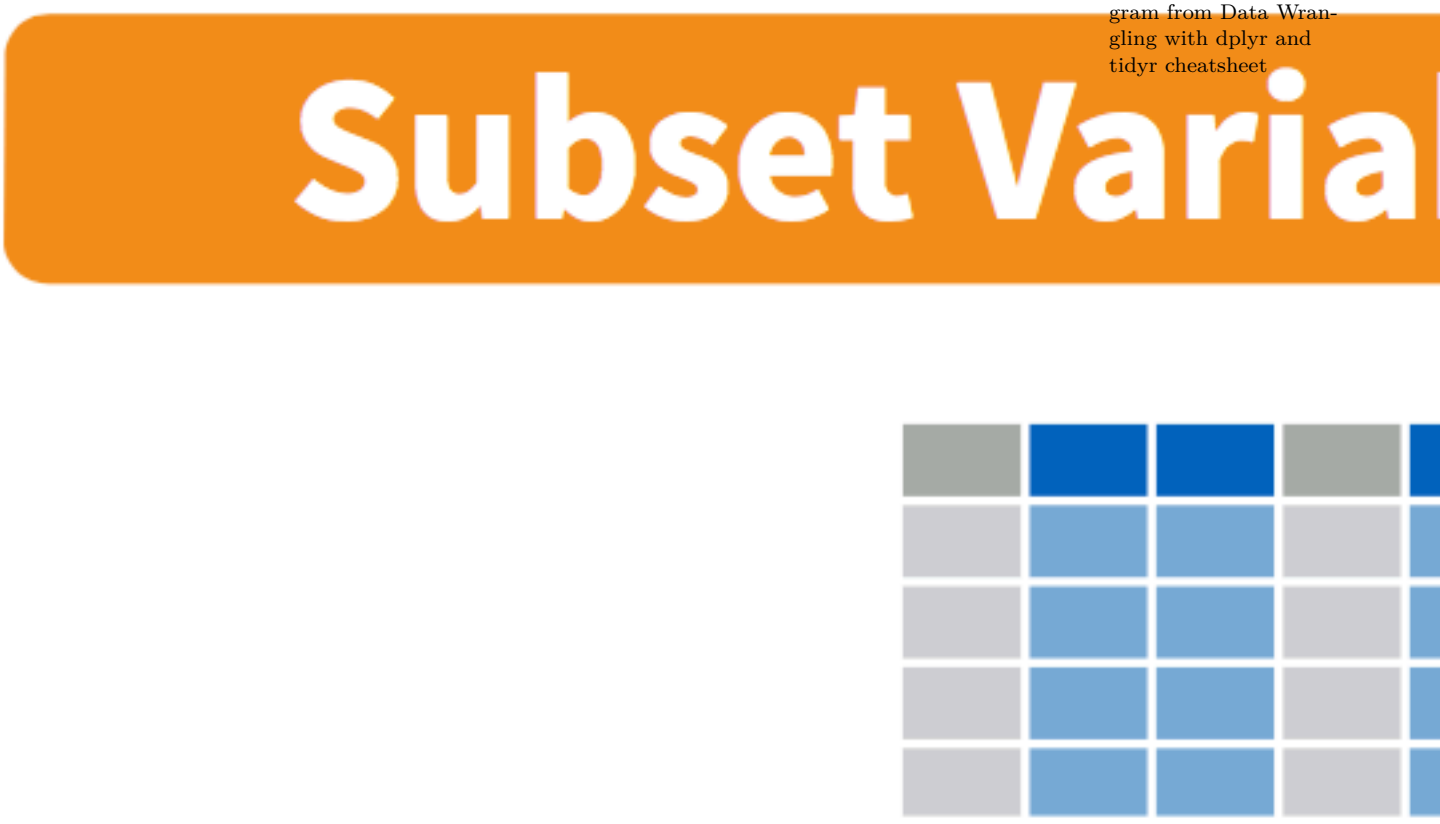



Figure 5.1: Select diagram from Data Wrangling with dplyr and tidyr cheatsheet

We've seen that the `flights` data frame in the `nycflights13` package contains many different variables (19 in fact). You can identify this by running the `dim` function:

```
library(nycflights13)
data(flights)
dim(flights)
```

```
## [1] 336776     19
```

One of these variables is `year`. If you remember the original description of the `flights` data frame (or by running `?flights`), you'll remember that this data correspond to flights in 2013 departing New York City. The `year` variable isn't really a variable here in that it doesn't vary... `flights` actually comes from a larger data set that covers many years. We may want to remove the `year` variable from our data set. To do so easily, we use the `select` variable:

```
library(dplyr)
flights <- select(.data = flights, -year)
names(flights)
```

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

The `names` function gives a listing of all the columns in a data frame. We see that `year` has been removed. This was done using a `-` in front of the name of the column we'd like to remove.

We could also select specific columns (instead of deselecting columns) by listing them out:

```
flight_dep_times <- select(flights, month, day, dep_time, sched_dep_time)
flight_dep_times
```

```
## # A tibble: 336,776 x 4
##    month   day dep_time sched_dep_time
##    <int> <int>    <int>          <int>
## 1      1     1      517            515
## 2      1     1      533            529
## 3      1     1      542            540
## 4      1     1      544            545
## 5      1     1      554            600
## 6      1     1      554            558
## 7      1     1      555            600
## 8      1     1      557            600
## 9      1     1      557            600
## 10     1     1      558            600
## # ... with 336,766 more rows
```

Or we could specify a ranges of columns:

```
flight_arr_times <- select(flights, month:day, arr_time:sched_arr_time)
flight_arr_times
```

```
## # A tibble: 336,776 x 4
##    month   day arr_time sched_arr_time
##    <int> <int>    <int>          <int>
## 1      1     1      830            819
## 2      1     1      850            830
## 3      1     1      923            850
## 4      1     1     1004           1022
## 5      1     1      812            837
## 6      1     1      740            728
## 7      1     1      913            854
## 8      1     1      709            723
## 9      1     1      838            846
## 10     1     1      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'd 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 <- select(flights, month:day, hour:time_hour, everything())
names(flights_reorder)
```

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

Lastly, the helper functions `starts_with`, `ends_with`, and `contains` can be used to choose column names that match those conditions:

```
flights_begin_a <- select(flights, starts_with("a"))
flights_begin_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 <- select(flights, ends_with("delay"))
flights_delays
```

```
## # 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 <- select(flights, contains("time"))
flights_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 variables: time_hour <time>
```

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 <- rename(flights_time,
                       departure_time = dep_time,
                       arrival_time = arr_time)
names(flights_time)
```

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

It's easy to forget if the new name comes before or after the equals sign. I usually remember this as "New Before, Old After" or NBOA.

You'll receive an error if you try to do it the other way:

```
Error: Unknown variables: departure_time, arrival_time.
```

---

***Learning check***

**(LC5.1)** How many different ways are there to select all three of `dest`, `air_time`, and `distance` variables from `flights`? Give the code showing how to do all of them you can think of.

**(LC5.2)** How could one use `starts_with`, `ends_with`, and `contains` to select columns from a dataset with 100 or so columns? Think up a dataset that might have that many columns and discuss how each of these functions could be used to make smaller data sets.

**(LC5.3)** Why might we want to use the `select` function on a data frame?

---

### *5.1.2 Filter observations using `filter`*

All of the FMVs follow the same syntax with the first argument to the function/verb being the name of the data frame and then the other arguments specifying which criteria you'd like the verb to work with.

The `filter` function here works much like the "Filter" option in Microsoft Excel. It allows you to specify criteria about values of 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"`:

```
portland_flights <- filter(flights, dest == "PDX")
portland_flights
```

```
## # A tibble: 1,354 x 18
##    month   day dep_time sched_dep_time dep_delay arr_time
```

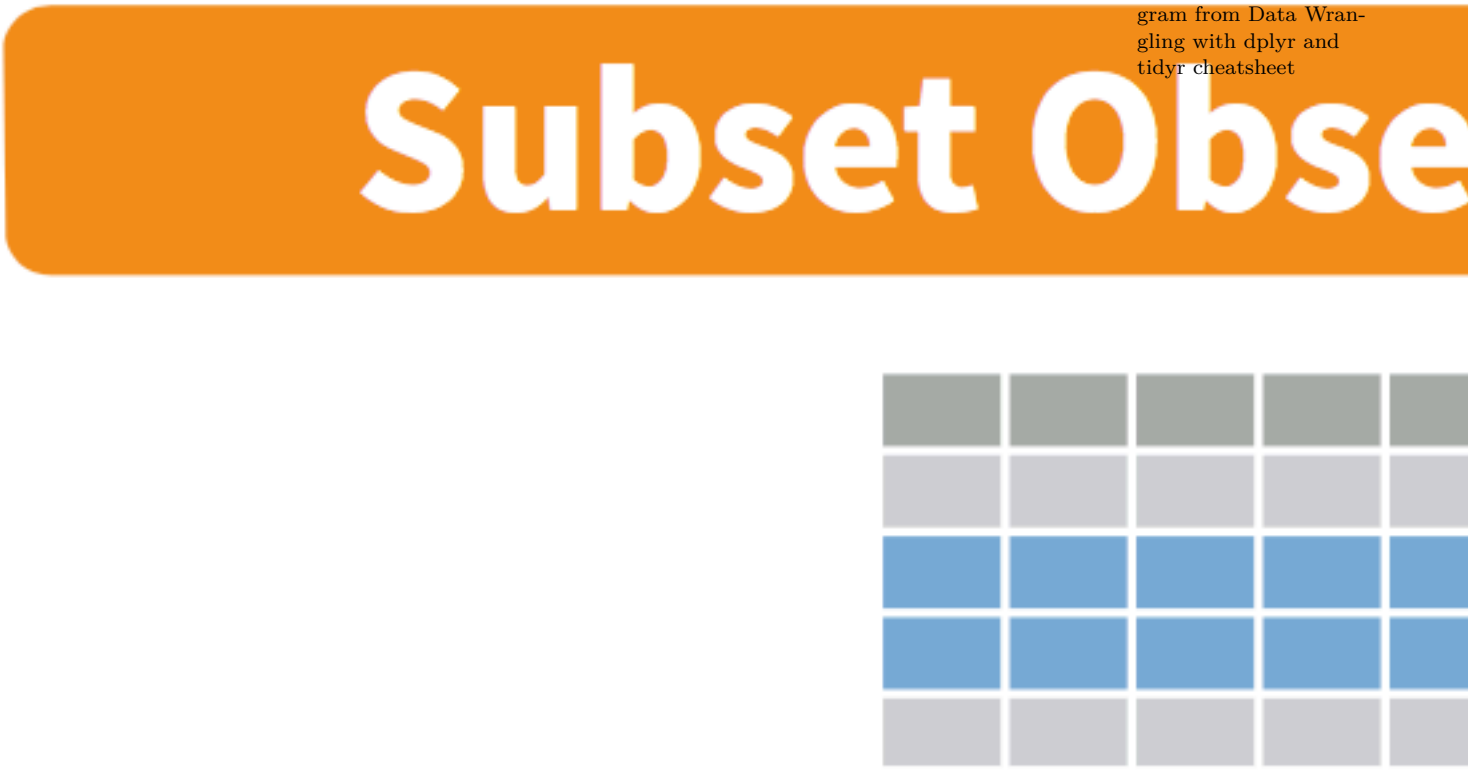



Figure 5.2: Filter diagram from Data Wrangling with dplyr and tidyr cheatsheet

```
##    <int> <int>    <int>          <int>     <dbl>    <int>
## 1      1     1     1739           1740        -1     2051
## 2      1     1     1805           1757         8     2117
## 3      1     1     2052           2029        23     2349
## 4      1     2      804            805        -1     1039
## 5      1     2     1552           1550         2     1853
## 6      1     2     1727           1720         7     2042
## 7      1     2     1738           1740        -2     2028
## 8      1     2     2024           2029        -5     2314
## 9      1     3     1755           1745        10     2110
## 10     1     3     1814           1727        47     2108
## # ... with 1,344 more rows, and 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 <time>
```

Note the second equals sign here. You are almost guaranteed to make the mistake at least one of only including one equals sign. Let's see what happens when we make this error:

```
portland_flights <- filter(flights, dest = "PDX")
```

```
Error: filter() takes unnamed arguments. Do you need `==`?
```

We see that there were 1354 flights from New York City to Portland in 2013. Let's combine this with what we saw in Subsection **??** to ensure that Portland flights were selected:

```
reordered_flights <- select(flights, dest, everything())
pdx_flights <- filter(reordered_flights, dest == "PDX")
pdx_flights
```

```
## # A tibble: 1,354 x 18
##     dest month   day dep_time sched_dep_time dep_delay arr_time
##    <chr> <int> <int>    <int>          <int>     <dbl>    <int>
## 1    PDX     1     1     1739           1740        -1     2051
## 2    PDX     1     1     1805           1757         8     2117
## 3    PDX     1     1     2052           2029        23     2349
## 4    PDX     1     2      804            805        -1     1039
## 5    PDX     1     2     1552           1550         2     1853
## 6    PDX     1     2     1727           1720         7     2042
## 7    PDX     1     2     1738           1740        -2     2028
## 8    PDX     1     2     2024           2029        -5     2314
## 9    PDX     1     3     1755           1745        10     2110
## 10   PDX     1     3     1814           1727        47     2108
## # ... with 1,344 more rows, and 11 more variables:
## #   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, flight <int>,
```

```
## #   tailnum <chr>, origin <chr>, air_time <dbl>, distance <dbl>,
## #   hour <dbl>, minute <dbl>, time_hour <time>
```

We see that Portland flights were selected here. You could also run `View(pdx_flights)` to glance at the data in spreadsheet form.

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:

```
btv_sea_flights_fall <- filter(flights,
                               origin == "JFK",
                               (dest == "BTV") | (dest == "SEA"),
                               month >= 10)
```

Another example uses the `!` to choose pick rows that **DON'T** match a condition. Here we are referring to excluding the Northern Hemisphere summer months of June, July, and August.

```
not_summer_flights <- filter(flights,
                             !between(month, 6, 8))
not_summer_flights
```

```
## # A tibble: 249,781 x 18
##    month   day dep_time sched_dep_time dep_delay arr_time
##    <int> <int>    <int>          <int>     <dbl>    <int>
## 1      1     1      517            515         2      830
## 2      1     1      533            529         4      850
## 3      1     1      542            540         2      923
## 4      1     1      544            545        -1     1004
## 5      1     1      554            600        -6      812
## 6      1     1      554            558        -4      740
## 7      1     1      555            600        -5      913
```

```
## 8      1     1      557           600        -3      709
## 9      1     1      557           600        -3      838
## 10     1     1      558           600        -2      753
## # ... with 249,771 more rows, and 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 <time>
```

To check that we are correct here we can use the `count` function on the `month` variable in our `not_summer_flights` data frame to ensure June, July, and August are not selected:

```
count(not_summer_flights, month)
```

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

The function `between` is a shortcut. We could also have written the following to get the same result:

```
not_summer2 <- filter(flights, month <= 5 | month >= 9)
count(not_summer2, month)
```

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

***Learning check***

**(LC5.4)** What's another way using `!` we could filter only the rows that are not summer months (June, July, or August) in the `flights` data frame?

---

### *5.1.3 Summarize variables using `summarize`*


Figure 5.3: Summarize diagram from Data Wrangling with dplyr and tidyr cheatsheet

We saw in Subsection **??** a way to calculate the standard deviation and mean of the temperature variable `temp` in the `weather` data frame of `nycflights`. We can do so in one step using the `summarize` function in `dplyr`:

```
summarize(weather,
          mean = mean(temp),
          std_dev = sd(temp))
```

```
## # A tibble: 1 x 2
```


Figure 5.4: Another summarize diagram from Data Wrangling with dplyr and tidyr cheatsheet

```
##    mean std_dev
##   <dbl>   <dbl>
## 1    NA      NA
```

What happened here? The mean and the standard deviation temperatures are missing? Remember that by default the `mean` and `sd` functions do not ignore missing values. We need to specify `TRUE` for the `na.rm` parameter:

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

```
## # A tibble: 1 x 2
##       mean  std_dev
##      <dbl>    <dbl>
## 1 55.20351 17.78212
```

We've created a small data frame here called `summary_temp` that includes both the `mean` and the `std_dev` of the `temp` variable in `weather`. If we'd like to access either of these values directly we can use the `$` to specify a column in a data frame:

```
summary_temp$mean
```

```
## [1] 55.20351
```

```
summary_temp$std_dev
```

```
## [1] 17.78212
```

It's often more useful to summarize a variable based on the groupings of another variable. Let's say we were interested in the mean and standard deviation of temperatures for each month. We believe that you will be amazed at just how simple this is:

Figure 5.5: Group by and summarize diagram from Data Wrangling with dplyr and tidyr cheatsheet

```
grouped_weather <- group_by(weather, month)
summary_tempXmonth <- summarize(grouped_weather,
          mean = mean(temp, na.rm = TRUE),
          std_dev = sd(temp, na.rm = TRUE)
          )
summary_tempXmonth
```

```
## # A tibble: 12 x 3
##    month     mean   std_dev
##    <dbl>    <dbl>     <dbl>
## 1      1 35.64127 10.185459
## 2      2 34.15454  6.940228
## 3      3 39.81404  6.224948
## 4      4 51.67094  8.785250
## 5      5 61.59185  9.608687
## 6      6 72.14500  7.603356
```

```
## 7      7 80.00967  7.147631
## 8      8 74.40495  5.171365
## 9      9 67.42582  8.475824
## 10    10 60.03305  8.829652
## 11    11 45.10893 10.502249
## 12    12 38.36811  9.940822
```

By simply grouping the `weather` data set by `month` first and then passing this new data frame into `summarize` we get a resulting data frame that shows the mean and standard deviation temperature for each month in New York City.

Another useful function is the `n` function which gives a count of how many entries appeared in the groupings. Suppose we'd like to get a sense for how many flights departed each of the three airports in New York City:

```
grouped_flights <- group_by(flights, origin)
by_origin <- summarize(grouped_flights,
                       count = n())
by_origin
```

```
## # A tibble: 3 x 2
##   origin  count
##    <chr>  <int>
## 1    EWR 120835
## 2    JFK 111279
## 3    LGA 104662
```

We see that Newark (`"EWR"`) had the most flights departing in 2013 followed by `"JFK"` and lastly by LaGuardia (`"LGA"`).

***Learning check***

**(LC5.5)** What does the standard deviation column in the `summary_tempXmonth` data frame tell us about temperatures in New York City throughout the year?

**(LC5.6)** What code would be required to get the mean and standard deviation temperature for each day in 2013 for NYC?

**(LC5.7)** How could we identify how many flights left each of the three airports in each of the months of 2013?

---

### *5.1.4 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


Figure 5.6: Mutate diagram from Data Wrangling with dplyr and tidyr cheatsheet

frustrated when their flights departs 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:

```
flights_plus <- mutate(flights,
         gain = arr_delay - dep_delay)
```

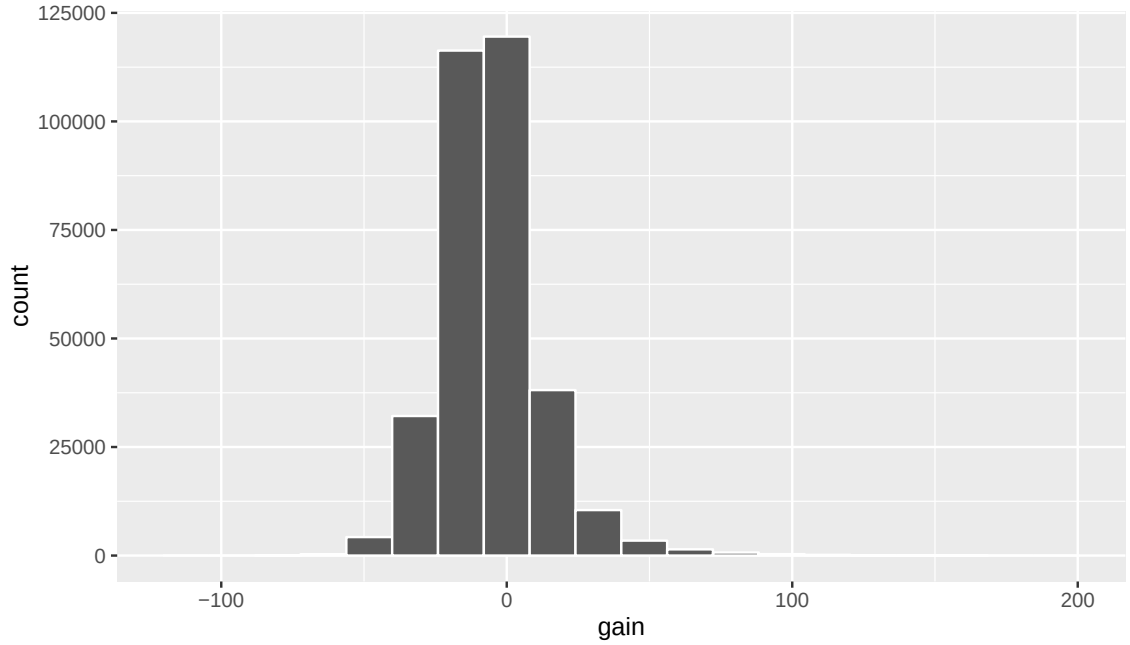
We can now look at summary measures of this `gain` variable and even plot it in the form of a histogram:

```
gain_summary <- summarize(flights_plus,
          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))
)
gain_summary
```

```
## # A tibble: 1 x 8
##     min    q1 median    q3   max      mean        sd missing
##   <dbl> <dbl>  <dbl> <dbl> <dbl>     <dbl>    <dbl>   <int>
## 1  -109   -17     -7     3   196 -5.659779 18.04365    9430
```

We've recreated the `summary` function we say in Chapter **??** here using the `summarize` function in `dplyr`.

```
library(ggplot2)
ggplot(flights_plus, aes(x = gain)) +
  geom_histogram(color = "white", bins = 20)
```




We can also create multiple columns at once and even refer to columns that were just created in a new column. Hadley produces one such example in Chapter 5 of "R for Data Science" (**?**):

```
flights_plus2 <- mutate(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
```

---

***Learning check***

**(LC5.8)** What do positive values of the `gain` variable in `flights_plus` correspond to? What about negative values? And what about a zero value?

**(LC5.9)** 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`.

**(LC5.10)** 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.

---

### *5.1.5 Reorder the data frame using `arrange`*

If you've ever worked with data, one of the most common things you'd like to do is sort it. 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 most frequently 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:

```
by_dest <- group_by(flights, dest)
freq_dest <- summarize(by_dest, num_flights = n())
freq_dest
```

```
## # 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. Remember to use `View()` in the R Console to look at all the values of `freq_dest` in spreadsheet format. We are interested in finding those airports that appear most:

```
arrange(freq_dest, 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. To switch the ordering to be descending instead of ascending we use the `desc` function:

```
arrange(freq_dest, 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
```

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

```
top_n(freq_dest, n = 10, wt = num_flights)
```

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

We'll still need to arrange this by `num_flights` though:

```
arrange(top_n(freq_dest, n = 10, wt = num_flights), desc(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
```

**Note:** Remember that I didn't pull the `n` and `wt` arguments out of thin air. They can be found by using the `?` function on `top_n`.

***Learning check***

**(LC5.11)** Create a new data frame that shows the top 5 airports with the largest arrival delays from NYC in 2013.

---

## *5.2 The pipe %>%*

Just as the `+` sign was used to add layers to a plot created using `ggplot` we will use the pipe operator (`%>%`) to chain together `dplyr` functions. We'll see that we can even chain together `dplyr` functions and plotting code. (Both `ggplot2` and `dplyr` were created by Hadley after all.)

You may have been a little confused by the last chunk we created above:

```
arrange(top_n(freq_dest, n = 10, wt = num_flights), desc(num_flights))
```

If we don't create temporary variables like we did before with `by_dest`, `grouped_flights`, etc., we start to get into the issue of trying to match parentheses. We could separate this code a bit to help with this:

```
arrange(
  top_n(freq_dest,
        n = 10,
```

```
        wt = num_flights),
  desc(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
```

Even this make it difficult to understand what is exactly happening though. `desc(num_flights)` is an argument to `arrange`. The best way to fix this problem is the use of the chaining operator called the pipe (`%>%`):

```
freq_dest %>%
  top_n(n = 10, wt = num_flights) %>%
  arrange(desc(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
```

Recall from Chapter @**??**viz) that we read the pipe operator as "and then". So here we take the `freq_dest` data frame **AND THEN** we determine the top 10 values of `num_flights` **AND THEN** we arrange these top 10 flights according to a descending numbers of flights (from highest to lowest).

We can go one stop 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()) %>%
  top_n(n = 10) %>%
  arrange(desc(num_flights))
```

```
## Selecting by num_flights
```

***Learning check***

**(LC5.12)** Recreate each of the chunks of code above Subsection **??** in this chapter using the `%>%` operator. Note that sometimes you can combine multiple subsequent chunks of code together. Do so whenever possible.

**(LC5.13)** What benefits can you see to using the pipe instead of the other way of doing things as you saw throughout this chapter? Give specific examples.

**(LC5.14)** Write out exactly how the `ten_freq_dests` data set was created using the "and then" verbiage.

---

The piping syntax will be our major focus throughout the rest of this book and you'll find that you'll quickly be addicted to the chaining with some practice. If you'd like to see more examples on using `dplyr`, the FMV (in addition to some other `dplyr` verbs), and `%>%` with the `nycflights13` data set, you can check out Chapter 5 of Hadley and Garrett's book (**?**).

## *5.3 Joining/merging data frames*

Something you may have thought to yourself as you looked at the most freqent destinations of flights from NYC in 2013 is

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

The `nycflights13` data source contains multiple data frames. Instead of having to manually look up different values of airport names corresponding to airport codes like `ORD`, we can have R automatically do this "looking up" for us.

To do so, we'll need to tell R how to match one data frame to another data frame. Let's first check out the `airports` data frame inside of R:

```
View(airports)
```

The first column `faa` corresponds to the airport codes that we saw in `dest` in our `flights` and subsequent `ten_freq_dests` data sets. Hadley and Garrett (**?**) created the following diagram to help us understand how the different data sets are linked:

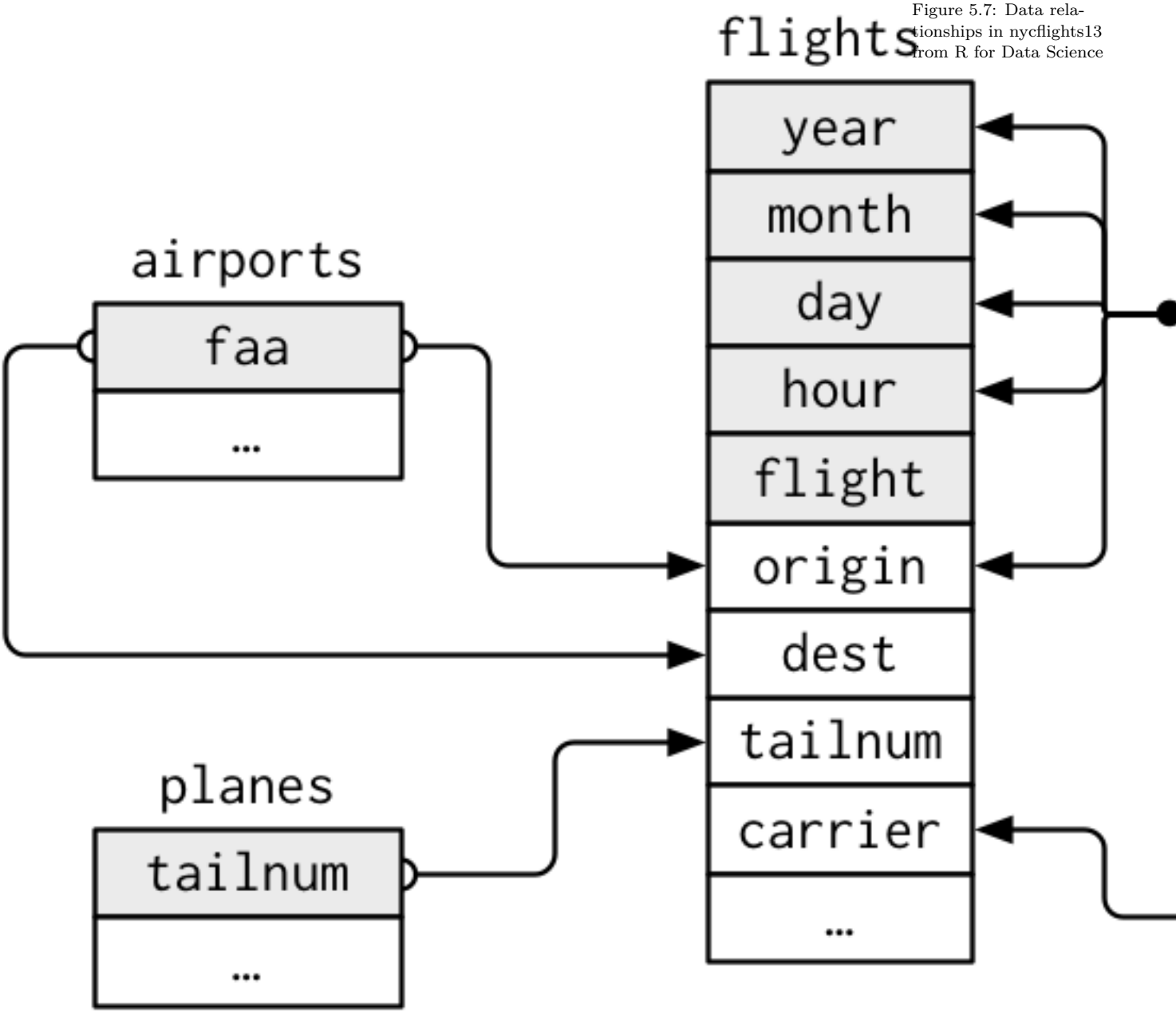



Figure 5.7: Data relationships in nycflights13 from R for Data Science

We see from `View(airports)` that `airports` contains a lot of other information about 1396. We are only really interested here in the `faa` and `name` columns. Let's use the `select` function to only use those variables:

```
airports_small <- airports %>%
  select(faa, name)
```

So if we identify the names of the airports we can use the `inner_join` function. Note that we will also rename the subsequent column `name` as `airport_name`:

```
named_freq_dests <- ten_freq_dests %>%
  inner_join(airports_small, by = c("dest" = "faa")) %>%
  rename(airport_name = name)
named_freq_dests
```

```
## # A tibble: 10 x 3
##     dest num_flights                        airport_name
##    <chr>       <int>                               <chr>
## 1    ORD       17283                 Chicago Ohare Intl
## 2    ATL       17215    Hartsfield Jackson Atlanta Intl
## 3    LAX       16174                   Los Angeles Intl
## 4    BOS       15508 General Edward Lawrence Logan Intl
## 5    MCO       14082                       Orlando Intl
## 6    CLT       14064             Charlotte Douglas Intl
## 7    SFO       13331                 San Francisco Intl
## 8    FLL       12055     Fort Lauderdale Hollywood Intl
## 9    MIA       11728                         Miami Intl
## 10   DCA        9705      Ronald Reagan Washington Natl
```

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.

A visual representation of the `inner_join` is given (**?**):

There are more complex joins available, but the `inner_join` will solve nearly all (if not all) of the problems you'll face in our experience.

---

***Learning check***

**(LC5.15)** What happens when you try to `inner_join` the `ten_freq_dests` data frame with `airports` instead of `airports_small`? How might one use this result to answer further questions about the top 10 destinations?

**(LC5.16)** What surprises you about the top 10 destinations from NYC in 2013?

---

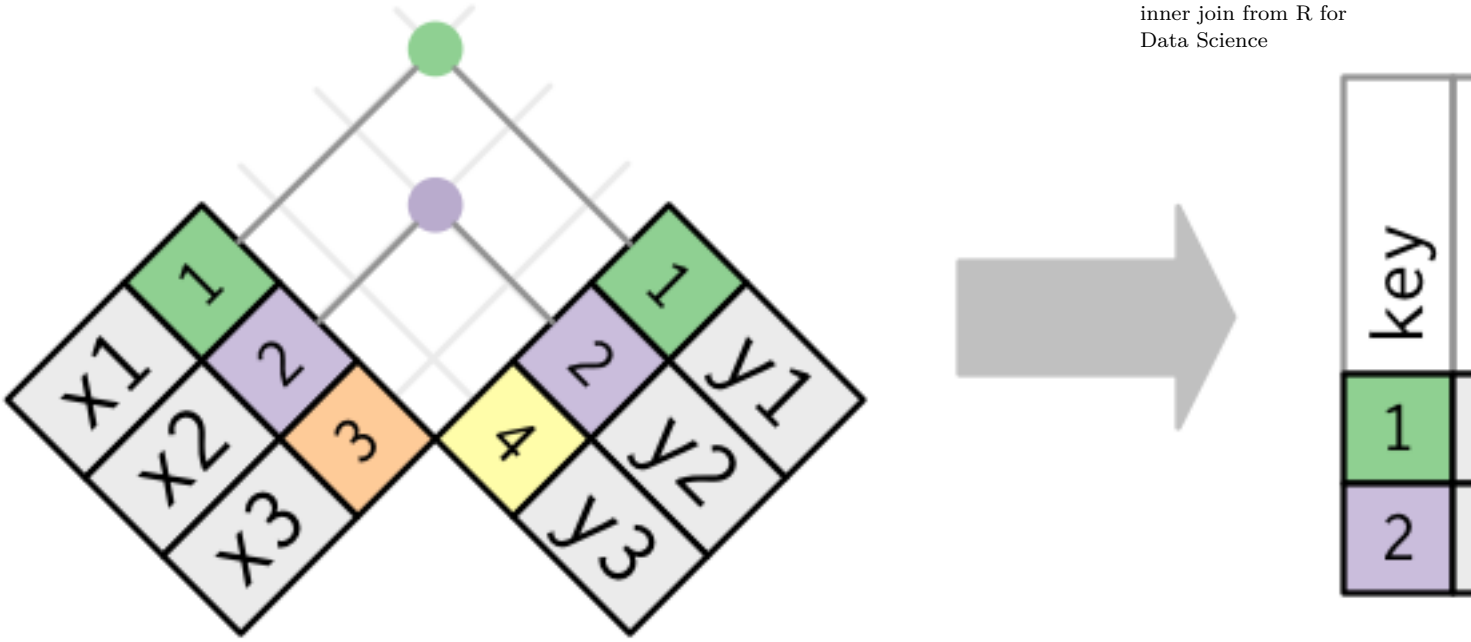



Figure 5.8: Diagram of inner join from R for Data Science

## *5.4 What's to come?*

This concludes the **Data Exploration** unit of this book. You should be pretty proficient in both plotting variables (or multiple variables together) in various data sets and manipulating data as we've done in this chapter. You are encouraged to step back through the code in earlier chapters and make changes as you see fit based on your updated knowledge.

In Chapter **??**, we'll begin to build the pieces needed to understand how this unit of **Data Exploration** can tie into statistical inference in the **Inference** part of the book. Remember that the focus throughout is on data visualization and we'll see that next when we discuss sampling.

# 6
# *Collecting Data*

More traditional in feel

- (random) Sampling: representativeness/generalizability/bias
  - Image of sampling: population, soup idea
  - Representativeness/bias: OkCupid to make statements. Population
- Observational study vs randomized experiment (fit with correlation)
  - OkCupid is an observational study
- Correlation is not causation
  - Confounding variables
  - Shoes on vs waking up with headache
  - Think up data example to include in earlier chapters

---

***Learning check***

**(LC6.1)**

---

## *6.1 What's to come?*

# Part II

# Inference

# 7
# *Inference Basics*

- Idea of histogram of sample being a visual approximation to population distribution.
  - Shiny
  - Take normal distribution using mean/SD of okcupid heights and plot count (not density) curve on top of histogram.
- Sampling distribution:
  - Permutation test:
    * Flight delays for American vs United
    * Do women smoke more than men in OkCupid?
    * Think up more dichotomies than male vs female.
    *
- Standard errors and sampling distribution via randomization
  - Developing traditional inference from randomization
  - two-sample permutation test -> null distribution
  - Showing what happens when assumptions/conditions aren't met
  - Then show normal/t-test and show how it fits on top of sampling distribution
  - Shiny Google Forms

---

***Learning check***

**(LC7.1)**

---

## *7.1 What's to come?*

# *8*
# *Hypothesis Testing*

- Hypothesis testing
  - The theory of hypothesis. Criminal justice. Question: what do you do with a problem like alpha?
  - Example of non-rejection of H0: heights of men and women, flights from NYC to Boston vs flights from NYC to SF.

---

***Learning check***

**(LC8.1)**

---

## *8.1 What's to come?*

# 9
# *Confidence Intervals*

---

***Learning check***

**(LC9.1)**

---

## *9.1 What's to come?*

# *10*
# *Simple and Multiple Regression*

- Simple linear regression
- Multiple regression
  - Regression/correlation/multiple regression/confounding
  - Categorical predictor and baseline
  - Implement `tidy`, `broom::augment`, and `glance` in `broom` package to get results
  - Model selection is a can of worms
  - Cross-validation
    * Take half of dataset for fit, use to predict other half.
      · Show random sampling of half matters
      · Is only once enough?

---

***Learning check***

**(LC10.1)**

---

## *10.1 What's to come?*

# Part III

# Conclusion

# *11*
# *Concluding Remarks*

# Part IV

# Appendix

# 12
# Appendix A: Intermediate R

## 12.1 Sorted barplots

Building upon the example in Section **??**:

```
library(nycflights13)
library(ggplot2)
flights_table <- table(flights$carrier)
flights_table
```

```
##
##    9E    AA    AS    B6    DL    EV    F9    FL    HA    MQ    OO    UA
## 18460 32729   714 54635 48110 54173   685  3260   342 26397    32 58665
##    US    VX    WN    YV
## 20536  5162 12275   601
```

```
#library(dplyr)
#carrier_counts <- flights %>% count(carrier)
#carrier_counts
```

We can sort this table from highest to lowest counts by using the `sort` function:

```
sorted_flights <- sort(flights_table, decreasing = TRUE)
names(sorted_flights)
```

```
##  [1] "UA" "B6" "EV" "DL" "AA" "MQ" "US" "9E" "WN" "VX" "FL" "AS" "F9"
## [14] "YV" "HA" "OO"
```

```
#carrier_counts <- carrier_counts %>%
#  arrange(desc(n))
```

It is often preferred for barplots to be ordered corresponding to the heights of the bars. This allows the reader to more easily compare the ordering of different airlines in terms of departed

flights (**?**). We can also much more easily answer questions like "How many airlines have more departing flights than Southwest Airlines?".

We can use the sorted table giving the number of flights defined as `sorted_flights` to **reorder** the `carrier`.

```
ggplot(data = flights, mapping = aes(x = carrier)) +
  geom_bar() +
  scale_x_discrete(limits = names(sorted_flights))
```

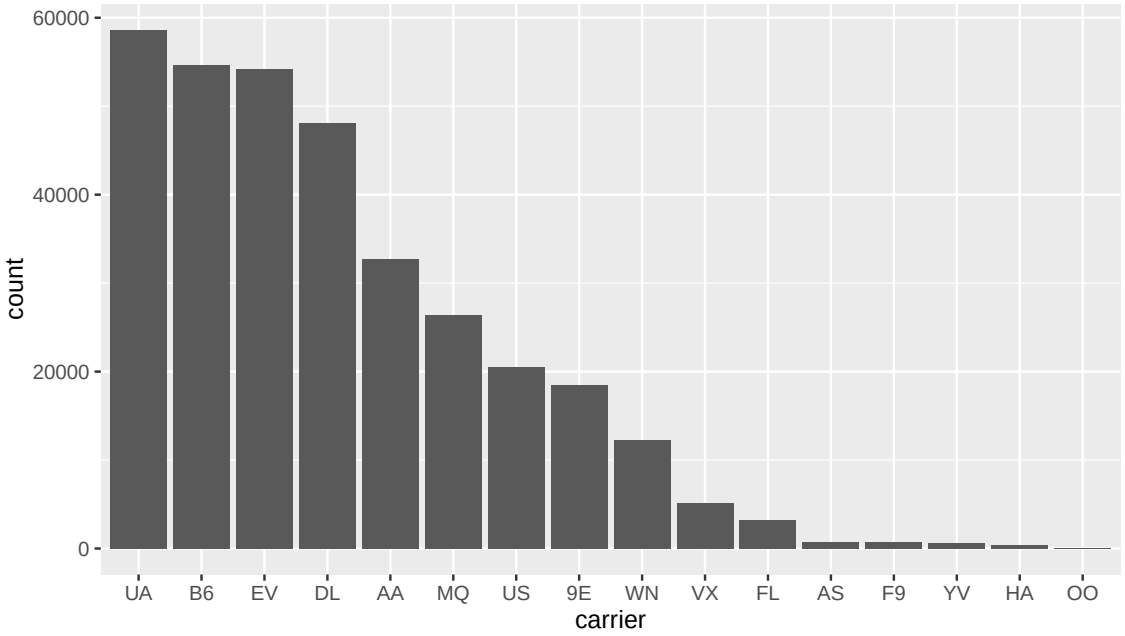



Figure 12.1: Number of flights departing NYC in 2013 by airline - Descending numbers

```
#ggplot(data = carrier_counts, mapping = aes(x = carrier, y = n)) +
#  geom_bar(stat = "identity") +
#  scale_x_discrete(limits = carrier)
```

The last addition here specifies the values of the horizontal `x` axis on a discrete scale to correspond to those given by the entries of `sorted_flights`.

** ** What are three specific questions that can be more easily answered by looking at Figure 4.6 instead of Figure 4.5?

---

- Changing the labels of a plot (x-axis, y-axis)
- Changing the theme for ggplots (`ggthemes` package too)
- Adding `code_folding` and `code_download` to YAML
- `kable` function from `knitr`
- Reading in data from files in different formats
- Reshaping the data with `tidyr`

# *13*
# *Appendix B: Statistical Basics*

## *13.1 Basic statistical terms*

- Mean
- Median
- Standard deviation
- Five-number summary
- Distribution
- Outliers

# Part V

# Bibliography