Enter the Tidyverse: An Introduction to Tidy Data Analysis in R

August 3, 2024

Table of contents

Objectives

This notebook addresses the following items.

- How do I install and load packages in R? In particular we'll work with the `tidyverse`.
- How do I read data into R from both local and remote sources?
- How do I interact with, and manipulate, data using the tools and principles of the `tidyverse`?

Installing and Loading Packages

We can install R packages using the command `install.packages("PACKAGE_NAME")`. Once packages are installed, we can load them into an R Session by running `library(PACKAGE_NAME)`. While packages only need to be *installed* once, they must be loaded in each R Session you intend to use them in (**note:** an R Session begins when R/RStudio are opened and ends when they are closed or terminated). We can install and load the `tidyverse` by running the code below:

```
install.packages("tidyverse")
library(tidyverse)
```

1. Open RStudio and run these commands in the Console pane (left/lower-left). We'll be using the `kableExtra` and `tidymodels` "packages" in our course – install both of these packages as well. Load the `kableExtra` package since we'll be using it here.

Loading Data

Now that you have the `tidyverse` loaded, the next thing we'll need is actual data to manipulate. The `tidyverse` comes with a few standard data sets for practicing with, but we'll be much more interested in working with our own data which we'll either find locally (stored on your own computer) or remotely (accessed via a web URL). The `tidyverse` includes several functions for reading data in a variety of formats:

- `read_csv("PATH_TO_FILE")` can be used to read data from a comma separated values (csv) file.
- `read_delim("PATH_TO_FILE", delim = "DELIMITER")` is a more general version of the `read_csv()` function – we can use this to read text files whose delimiter is something other than a comma. Common delimiters are the tab (`\t`) or space (`\s`).
- `read_excel("PATH_TO_FILE", sheet = "SHEET_NAME")` can be used to read data from a particular sheet within an xls or xlsx file.

The following examples show how we can read a variety of files into an R Session.

```
#Read the MAT241 sheet from the grades.xls file in
#the Spring 2021 folder on my computer's desktop
grades <- read_excel("C:/Users/agilb/Desktop/Spring 2021/grades.xls", sheet = "MAT241")

#Read in data from a csv file of Tate Gallery Artists housed
#in a public github repository on the web
tate_artists <- read_csv("https://github.com/tategallery/collection/raw/master/artist_data.c
```

```
#Read in data from a csv file of Tate Gallery Artworks housed
#in a public github repository on the web
#*Note* that read_csv() would have worked just fine here too
tate_works <- read_delim("https://raw.githubusercontent.com/rfordatascience/tidytuesday/maste
```

Viewing Data

Now that we've got data, the first thing we should do is look at it. There are a few really handy R functions for *getting a feel* for the data you have access to. The `View()`, `head()`, `tail()`, and `glimpse()` functions are four that are really commonly used. For the remainder of this notebook we'll use a data frame called `mpg` which is built into the `tidyverse`.

- Running `View(mpg)` will open a file viewer which allows you to navigate the data frame in a familiar spreadsheet format.
- Using `head(mpg)` and `tail(mpg)` give us a convenient method for looking at the first six and last six rows of a data frame, respectively. This is typically enough to give us an idea of the type of data we are working with. Running both of these functions can also make us aware of potential inconsistencies in data collection.

```
head(mpg) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

```
tail(mpg) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
volkswagen	passat	1.8	1999	4	auto(l5)	f	18	29	p	midsize
volkswagen	passat	2.0	2008	4	auto(s6)	f	19	28	p	midsize
volkswagen	passat	2.0	2008	4	manual(m6)	f	21	29	p	midsize

volkswagen	passat	2.8	1999	6	auto(l5)	f	16	26	p	midsize
volkswagen	passat	2.8	1999	6	manual(m5)	f	18	26	p	midsize
volkswagen	passat	3.6	2008	6	auto(s6)	f	17	26	p	midsize

- Note that the `kable() %>% kable_styline(bootsrap_options = c("hover", "striped"))` commands are used to produce visually appealing tables in our html output – they don't actually do anything to transform our data. You are encouraged (though not required) to use these lines when you want to print out tabular output. You can see what the output looks like without using `kableExtra` below. I'll continue to utilize `kableExtra` throughout our course.

```r
head(mpg)
```

```
# A tibble: 6 x 11
  manufacturer model displ  year   cyl trans      drv     cty   hwy fl    class
  <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p     compa~
2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p     compa~
3 audi         a4      2    2008     4 manual(m6) f        20    31 p     compa~
4 audi         a4      2    2008     4 auto(av)   f        21    30 p     compa~
5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p     compa~
6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p     compa~
```

```r
tail(mpg)
```

```
# A tibble: 6 x 11
  manufacturer model  displ  year   cyl trans      drv     cty   hwy fl    class
  <chr>        <chr>  <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <chr>
1 volkswagen   passat   1.8  1999     4 auto(l5)   f        18    29 p     mids~
2 volkswagen   passat   2    2008     4 auto(s6)   f        19    28 p     mids~
3 volkswagen   passat   2    2008     4 manual(m6) f        21    29 p     mids~
4 volkswagen   passat   2.8  1999     6 auto(l5)   f        16    26 p     mids~
5 volkswagen   passat   2.8  1999     6 manual(m5) f        18    26 p     mids~
6 volkswagen   passat   3.6  2008     6 auto(s6)   f        17    26 p     mids~
```

- Running `glimpse(mpg)` provides us with a bit more technical information about how R is interpreting the columns of the `mpg` data frame. Knowing how R is interpreting our variables (columns) is important because certain operations are possible with numerical data but are not possible with categorical data, and vice-versa. Common data types in R are `chr/fct` (categorical data) and `num/dbl/int` (numerical data).

```r
glimpse(mpg)
```

```
Rows: 234
Columns: 11
$ manufacturer <chr> "audi", "audi", "audi", "audi", "audi", "audi", "audi", "~
$ model        <chr> "a4", "a4", "a4", "a4", "a4", "a4", "a4", "a4 quattro", "~
$ displ        <dbl> 1.8, 1.8, 2.0, 2.0, 2.8, 2.8, 3.1, 1.8, 1.8, 2.0, 2.0, 2.~
$ year         <int> 1999, 1999, 2008, 2008, 1999, 1999, 2008, 1999, 1999, 200~
$ cyl          <int> 4, 4, 4, 4, 6, 6, 6, 4, 4, 4, 4, 6, 6, 6, 6, 6, 6, 8, 8, ~
$ trans        <chr> "auto(l5)", "manual(m5)", "manual(m6)", "auto(av)", "auto~
$ drv          <chr> "f", "f", "f", "f", "f", "f", "f", "4", "4", "4", "4", "4~
$ cty          <int> 18, 21, 20, 21, 16, 18, 18, 18, 16, 20, 19, 15, 17, 17, 1~
$ hwy          <int> 29, 29, 31, 30, 26, 26, 27, 26, 25, 28, 27, 25, 25, 25, 2~
$ fl           <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p~
$ class        <chr> "compact", "compact", "compact", "compact", "compact", "c~
```

Manipulating Data

Now that we know how to load and view our data, let's talk about manipulating it. We can restrict the data we are working with, produce summaries of the data, transform the data, and more.

Pipes %>%

Pipes are a functionality that is included in a package that is part of `tidyverse` library. At first, the syntax may seem a bit strange, but pipes allow you to easily manipulate data without having to rename and save the dataset along the way. I strongly encourage you get used to working with pipes! In the previous section we saw how to use R's `head()` function to look at the first six rows of the dataset. Here's how to achieve the same outcome with the use of the pipe (`%>%`) operator.

```r
mpg %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact

audi	a4	2.8	1999	6	manual(m5)	f	18	26	p	compact

You can read the code above as saying "take the `mpg` dataset, and plug it into the `head()` function". Putting `head()` indented on a new line is not necessary for the code to work, but it does make the code easier to read. This new method of asking for the `head()` of the dataset may seem silly and inefficient, but the real magic of the pipe is that it allows us to chain operations together in a way that mimics the way humans think about instructions. We'll see this in action as we get exposure to more data manipulation tools below.

Restricting Data

The most common methods for restricting data deal with filtering out rows or columns so that we are only working with a subset of our original data set.

Filtering Rows (`filter()`)

Sometimes we are not interested in all of the observations in a particular dataset, but only those satisfying certain criteria. For example, maybe we only want to see vehicles falling into the class of *subcompact* cars. The `filter()` function will allow us to get rid of all other classes of vehicle.

```
mpg %>%
  filter(class == "subcompact") %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
ford	mustang	3.8	1999	6	manual(m5)	r	18	26	r	subcompact
ford	mustang	3.8	1999	6	auto(l4)	r	18	25	r	subcompact
ford	mustang	4.0	2008	6	manual(m5)	r	17	26	r	subcompact
ford	mustang	4.0	2008	6	auto(l5)	r	16	24	r	subcompact
ford	mustang	4.6	1999	8	auto(l4)	r	15	21	r	subcompact
ford	mustang	4.6	1999	8	manual(m5)	r	15	22	r	subcompact

We can also use more complex conditions on which rows to see using *and* (`&`) and *or* (`|`) statements. Maybe we want to see only those vehicles in the made by `subaru` or getting at least a 35 highway mile per gallon rating (`hwy`).

```r
mpg %>%
  filter(manufacturer == "subaru" | hwy >= 35) %>%
  head()  %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
honda	civic	1.8	2008	4	auto(l5)	f	25	36	r	subcompact
honda	civic	1.8	2008	4	auto(l5)	f	24	36	c	subcompact
subaru	forester awd	2.5	1999	4	manual(m5)	4	18	25	r	suv
subaru	forester awd	2.5	1999	4	auto(l4)	4	18	24	r	suv
subaru	forester awd	2.5	2008	4	manual(m5)	4	20	27	r	suv
subaru	forester awd	2.5	2008	4	manual(m5)	4	19	25	p	suv

Selecting Columns (`select()`)

Similarly to the way we can filter rows, we can select only those columns we are interested in. We can pass the names of the columns we are interested in to R's `select()` function so that we only see those selected columns returned.

```r
mpg %>%
  select(manufacturer, model, year, cty, hwy, class) %>%
  head()  %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	year	cty	hwy	class
audi	a4	1999	18	29	compact
audi	a4	1999	21	29	compact
audi	a4	2008	20	31	compact
audi	a4	2008	21	30	compact
audi	a4	1999	16	26	compact
audi	a4	1999	18	26	compact

We can also select all columns *except* certain ones by preceding the column name with a `-`.

```
mpg %>%
  select(-displ,-cyl) %>%
  head()  %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	year	trans	drv	cty	hwy	fl	class
audi	a4	1999	auto(l5)	f	18	29	p	compact
audi	a4	1999	manual(m5)	f	21	29	p	compact
audi	a4	2008	manual(m6)	f	20	31	p	compact
audi	a4	2008	auto(av)	f	21	30	p	compact
audi	a4	1999	auto(l5)	f	16	26	p	compact
audi	a4	1999	manual(m5)	f	18	26	p	compact

The `select()` function is also useful for changing the order of the columns.

```
mpg %>%
  select(cty, hwy, manufacturer) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

cty	hwy	manufacturer
18	29	audi
21	29	audi
20	31	audi
21	30	audi
16	26	audi
18	26	audi

Combining the Two

We can combine `filter()` and `select()` through the pipe as well. For any pipe, the result of the "upstream" code (the code before the pipe) is passed into the function that follows the pipe.

```r
mpg %>%
  filter(year >= 2005) %>%
  select(manufacturer, model, year, cty, hwy, class) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	year	cty	hwy	class
audi	a4	2008	20	31	compact
audi	a4	2008	21	30	compact
audi	a4	2008	18	27	compact
audi	a4 quattro	2008	20	28	compact
audi	a4 quattro	2008	19	27	compact
audi	a4 quattro	2008	17	25	compact

A Note on Pipes: The advantage to the pipe operator is probably pretty clear by now. The code we just wrote says *take the `mpg` data set, filter it so that we only see cars manufactured since 2005, show me only the few columns I am interested in, and just let me see the first six rows for now.* The alternative to this would be writing code that looks a lot less readable:

```r
head(select(filter(mpg, year >= 2005), manufacturer, model, year, cty, hwy, class))
```

Summarizing Data

There are lots of ways we can summarize our data. We can provide simple counts, compute averages, even build out our own summary functions.

Summarizing Categorical Data with Counts

We can start with a simple question like, *how many cars from each manufacturer are contained in this dataset?* To answer this, we simply pipe the `mpg` data frame into the `count()` function, identifying the `manufacturer` column as the column we wish to count.

```r
mpg %>%
  count(manufacturer) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	n
audi	18
chevrolet	19
dodge	37
ford	25
honda	9
hyundai	14

The counts are displayed in alphabetical order by manufacturer. We might be interested in the most well-represented manufacturers. We'll do this with `arrange()` – we can pass this function the argument `desc(n)` to say that we want to arrange by our new count column in descending order, and let's ask for the top 10 rows instead of the top 6.

```
mpg %>%
  count(manufacturer) %>%
  arrange(desc(n)) %>%
  head(n = 10) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	n
dodge	37
toyota	34
volkswagen	27
ford	25
chevrolet	19
audi	18
hyundai	14
subaru	14
nissan	13
honda	9

Let's say we wanted to know how many different models of car each manufacturer has released since the year 2000. This is a more complicated question. We would first need to filter the data so that we are only considering cars manufactured since the year 2000. Then we would subset to include only the `manufacturer` and `model` columns. There are lots of duplicates here, so we would want to remove them with a function called `distinct()`, and then finally we could count occurrences within each `manufacturer`

```
mpg %>%
  filter(year >= 2000) %>%
  select(manufacturer, model) %>%
  distinct() %>%
  count(manufacturer) %>%
  arrange(desc(n)) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	n
toyota	6
chevrolet	4
dodge	4
ford	4
volkswagen	4
audi	3

Summarizing Numerical Data

Summarizing categorical data is most often done with counts, but we've got many more choices when we are working with numerical data. We have several measures of center or spread that we could choose from – we could even define our own metrics. Let's say we wanted to know the median highway mile per gallon rating across all vehicles in our dataset. We'll need the help of R's `summarize()` function as well as the `median()` function for this.

```
mpg %>%
  summarize(median_hwy = median(hwy)) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

median_hwy
24

With the use of `summarize()` we can get multiple summaries at once. Let's compute the mean and standard deviation for both the highway and city mile per gallon ratings across all of the vehicles in our data set.

```r
mpg %>%
  summarize(mean_hwy = mean(hwy), std_deviation_hwy = sd(hwy), mean_cty = mean(cty), std_devi
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

mean_hwy	std_deviation_hwy	mean_cty	std_deviation_cty
23.44017	5.954643	16.85897	4.255946

It might be useful if we could get grouped summary statistics. Let's use `group_by()` to see how these measures vary across the different vehicle classes.

```r
mpg %>%
  group_by(class) %>%
  summarize(mean_hwy = mean(hwy), std_deviation_hwy = sd(hwy), mean_cty = mean(cty), std_devi
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

class	mean_hwy	std_deviation_hwy	mean_cty	std_deviation_cty
2seater	24.80000	1.303840	15.40000	0.5477226
compact	28.29787	3.781620	20.12766	3.3854999
midsize	27.29268	2.135930	18.75610	1.9465416
minivan	22.36364	2.062655	15.81818	1.8340219
pickup	16.87879	2.274280	13.00000	2.0463382
subcompact	28.14286	5.375012	20.37143	4.6023377
suv	18.12903	2.977973	13.50000	2.4208791

Let's arrange the result here by mean highway mile per gallon rating in the default ascending order.

```r
mpg %>%
  group_by(class) %>%
  summarize(mean_hwy = mean(hwy), std_deviation_hwy = sd(hwy), mean_cty = mean(cty), std_devi
  arrange(mean_hwy) %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

class	mean_hwy	std_deviation_hwy	mean_cty	std_deviation_cty
pickup	16.87879	2.274280	13.00000	2.0463382
suv	18.12903	2.977973	13.50000	2.4208791
minivan	22.36364	2.062655	15.81818	1.8340219
2seater	24.80000	1.303840	15.40000	0.5477226
midsize	27.29268	2.135930	18.75610	1.9465416
subcompact	28.14286	5.375012	20.37143	4.6023377
compact	28.29787	3.781620	20.12766	3.3854999

That's pretty informative although not totally surprising. Subcompact cars seem to have a high level of variation in their mpg ratings though!

Transforming Data

Often, you may be in a situation where you would like to create new columns, using the existing columns. This can be done using the `mutate()` command. The syntax is

```
dataset %>%
  mutate(new_column_name = function_of_old_columns)
```

In the `mpg` dataset, let's add a column which is the ratio between the city `cty` and highway `hwy` gas milages, and use the `arrange()` function to find cars with the highest city to highway gas milages:

```
mpg %>%
  mutate(mpg_ratio = cty/hwy) %>%
  select(manufacturer,model,cty,hwy,mpg_ratio) %>%
  arrange(desc(mpg_ratio)) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	cty	hwy	mpg_ratio
nissan	pathfinder 4wd	15	17	0.8823529
toyota	4runner 4wd	15	17	0.8823529
toyota	toyota tacoma 4wd	15	17	0.8823529
honda	civic	28	33	0.8484848

toyota	toyota tacoma 4wd	15	18	0.8333333
chevrolet	k1500 tahoe 4wd	14	17	0.8235294

Once pretty common step in an analysis is to create a categorical column from a variable which was originally numeric. In order to do this we can use the `if_else()` function. The three arguments of `if_else()` are a condition, and the values you want to fill if the condition is true or false, respectively.

```
mpg %>%
  mutate(pre_2000 = if_else(year < 2000, "yes", "no")) %>%
  select(manufacturer,model,year,pre_2000) %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("hover", "striped"))
```

manufacturer	model	year	pre_2000
audi	a4	1999	yes
audi	a4	1999	yes
audi	a4	2008	no
audi	a4	2008	no
audi	a4	1999	yes
audi	a4	1999	yes

Final Thoughts

There is a lot more to learn about data manipulation and R in general. Sticking to the `tidyverse` and the other package groups within the *tidy*-ecosystem (ie. `tidytext`, `tidymodels`, etc.) will be beneficial because they are all built on common syntax and programmatic principles. You can read more about this in the TidyTools Manifesto.

You won't be an expert after working through this document, but it should provide you with a solid start. Please feel free to add your own notes to this markdown file as we encounter more advanced functionality.