A crash course on the tidyverse

Eugene Katsevich

September 4, 2023

Contents

1 Data import, representation, and tidying **2**
- 1.1 Data import (`readr`) . . . 2
- 1.2 Tibbles . . . 2
- 1.3 Tidy data . . . 3
- 1.4 Recognizing untidy data . . . 3
- 1.5 Making data tidy (`tidyr`) . . . 4

2 Data visualization (`ggplot2`) **5**
- 2.1 Basic structure of a ggplot . . . 5
- 2.2 Adding geometric objects . . . 5
- 2.3 Customizing aesthetics . . . 6
- 2.4 Adding labels . . . 6
- 2.5 Creating different types of plots . . . 7
- 2.6 Faceting for Multiple Plots . . . 7

3 Data transformation (`dplyr`) **8**
- 3.1 Selecting columns with `select()` . . . 8
- 3.2 Filtering rows with `filter()` . . . 8
- 3.3 Arranging rows with `arrange()` . . . 9
- 3.4 Creating new columns with `mutate()` . . . 9
- 3.5 Summarizing data with `summarize()` . . . 9
- 3.6 Group-wise operations . . . 9
- 3.7 Combining data with joins . . . 10
- 3.8 Tips for efficient data transformation . . . 10

This document is a crash course on the `tidyverse`, the state-of-the-art paradigm for data science in R. The `tidyverse` is a collection of R packages for data import (`readr`), tidying (`tidyr`), transformation (`dplyr`), visualization (`ggplot2`), and others.

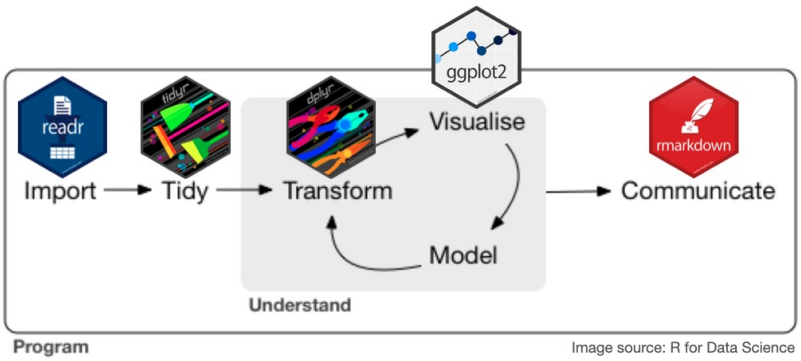


Image source: R for Data Science

All of these packages can be loaded with the following single command:

```
library(tidyverse)
```

1 Data import, representation, and tidying

1.1 Data import (`readr`)

The first step to analyzing data is to get it into R. When working with the `tidyverse`, data import is typically handled by the `readr` package. This package offers a set of functions to efficiently read rectangular data (like CSVs, TSVs, and other delimited formats).

Reading CSV files. CSV (Comma Separated Values) is one of the most common formats for sharing data. `readr` offers the `read_csv()` function to read such files:

```
data <- read_csv("data.csv") # compare to base R's read.csv() function
```

Reading TSV files. TSV (Tab Separated Values) is another popular format, especially for datasets where values might contain commas. To read TSV files, use `read_tsv()`:

```
data <- read_tsv("data.tsv")
```

Reading Excel files. While `readr` doesn't directly handle Excel files, the `readxl` package (part of the wider `tidyverse`) does. It's simple and works well with both `.xls` and `.xlsx` formats:

```
data <- read_excel("data.xlsx")
```

Reading from databases. For database connections, the `dbplyr` and `DBI` packages are typically used. They allow you to connect to a variety of databases, write SQL queries, and pull data directly into a tibble.

For efficient data import, consider employing the following tips:

- Use the `n_max` argument to limit the number of rows read. This is useful when previewing large datasets.
- Set `skip` to bypass initial rows, such as metadata at the top of a file.
- For large datasets, consider `vroom` (another package in the `tidyverse` universe) which is extremely fast for reading text data.

1.2 Tibbles

The import functions in `readr` will return data in the form of a tibble. In the tidyverse, a tibble is an enhanced version of a data frame. It provides a more modern and consistent way to work with tabular data. Tibbles have several advantages over traditional data frames, including better printing of large data, stricter handling of column types, and more informative error messages.

```
# Creating a tibble
my_tibble <- tibble(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 22),
  City = c("New York", "San Francisco", "Chicago")
)

# Printing a tibble
my_tibble
```

```
## # A tibble: 3 x 3
##   Name      Age City
```

```
##   <chr>   <dbl> <chr>
## 1 Alice      25 New York
## 2 Bob        30 San Francisco
## 3 Charlie    22 Chicago
```

1.3 Tidy data

Most of the packages in the tidyverse are meant to operate on *tidy data*, i.e. data which adheres to three key principles:

1. Each variable is a column; each column is a variable.
2. Each observation is a row; each row is an observation.
3. Each value is a cell; each cell is a single value.

Operating on tidy data makes data manipulation and analysis more streamlined.

1.4 Recognizing untidy data

Unfortunately, after importing data, usually we do not find it to be in a tidy format. Below are several examples of untidy data.

Column headers are values, not variable names. Consider this dataset of people's scores over different years:

```
## # A tibble: 3 x 3
##   Name    `2020` `2021`
##   <chr>    <dbl>  <dbl>
## 1 Alice       85     88
## 2 Bob         90     91
## 3 Charlie     87     85
```

Here, the column headers (2020 and 2021) are actual values of a variable (Year) rather than variable names.

Multiple variables are stored in one column. In this dataset, city and state are combined in a single column:

```
## # A tibble: 3 x 2
##   Name  Location
##   <chr> <chr>
## 1 Diane Austin, TX
## 2 Eva   Denver, CO
## 3 Frank Miami, FL
```

The `Location` column stores both the city and the state, which are distinct variables.

Variables are stored in both rows and columns. Imagine a dataset where the product types are both in rows and columns:

```
## # A tibble: 3 x 3
##   Product `Sold in Store` `Sold Online`
##   <chr>             <dbl>         <dbl>
## 1 A                    10             5
## 2 B                    20            15
## 3 Total                30            20
```

Here, the "Total" row is a summary, making the Product variable mixed with actual data and aggregated data.

A single observational unit is spread across multiple tables. Imagine you have customer details and their addresses in two separate tables:

```
## # A tibble: 2 x 3
##   CustomerID Name    Age
##        <dbl> <chr> <dbl>
## 1          1 Ian      28
## 2          2 Jenny    32
```

```
## # A tibble: 2 x 2
##   CustomerID Address
##        <dbl> <chr>
## 1          1 123 Main St
## 2          2 456 Elm St
```

The customer information is spread across two tables, making it more challenging to work with and analyze.

1.5 Making data tidy (`tidyr`)

The `tidyr` package provides several key functions to make data tidy:

- `pivot_longer()`: When you want to make your data longer, or gather columns.
- `pivot_wider()`: When you want to make your data wider, or spread rows into columns.
- `separate()`: When you need to separate one column into multiple columns.
- `unite()`: When you need to unite multiple columns into one.

To exemplify how `tidyr` works, consider the first example from the previous section:

```
## # A tibble: 3 x 3
##   Name    `2020` `2021`
##   <chr>    <dbl>  <dbl>
## 1 Alice       85     88
## 2 Bob         90     91
## 3 Charlie     87     85
```

Here, the column headers are values (i.e., the years 2020 and 2021). We can use the `pivot_longer()` function from the `tidyr` package to tidy this data.

```
tidy_scores <- scores_by_year |>
  pivot_longer(
    cols = c(`2020`, `2021`), # columns to be pivoted into longer format
    names_to = "Year",        # name of the new column that will store old column names
    values_to = "Score"       # name of the new column that will store the values
  )
tidy_scores
```

```
## # A tibble: 6 x 3
##   Name    Year  Score
##   <chr>   <chr> <dbl>
## 1 Alice   2020     85
## 2 Alice   2021     88
## 3 Bob     2020     90
## 4 Bob     2021     91
## 5 Charlie 2020     87
## 6 Charlie 2021     85
```

Now, the dataset is in a tidy format where each row is an observation, and each column is a variable. The years, which were previously column headers, are now values in the "Year" column, and the scores are in the "Score" column.

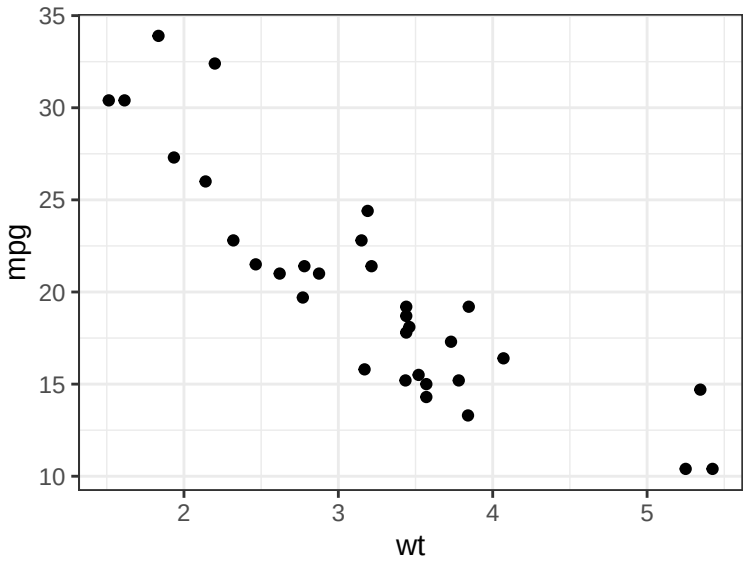
2 Data visualization (ggplot2)

ggplot2 offers an approachable and expressive syntax for creating a diverse range of visualizations. Let's explore its capabilities with examples using the mtcars dataset, a dataset included with R which contains various attributes of 32 car models.

2.1 Basic structure of a ggplot

Start with the ggplot() function to set up your visualization:

```
# Base plot with data
ggplot(data = mtcars, aes(x = wt, y = mpg)) +
  geom_point()
```

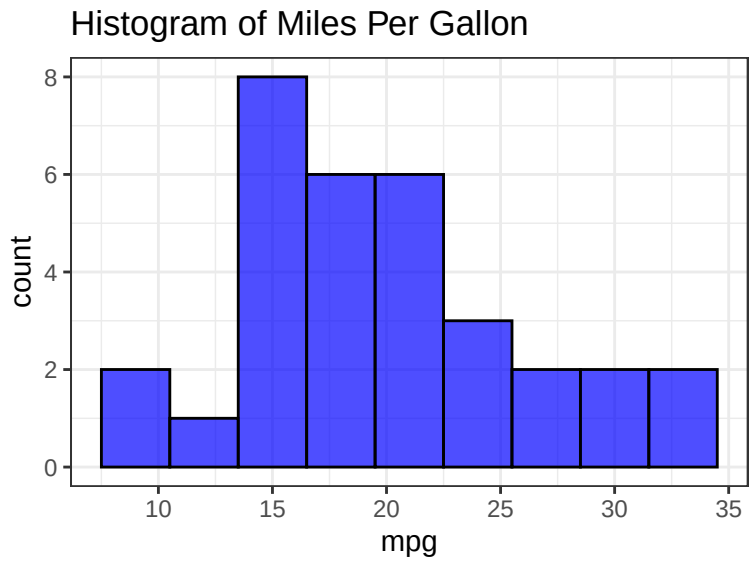


This creates a simple scatter plot displaying car weights (wt) against miles per gallon (mpg).

2.2 Adding geometric objects

Beyond scatter plots, there are numerous geometries:

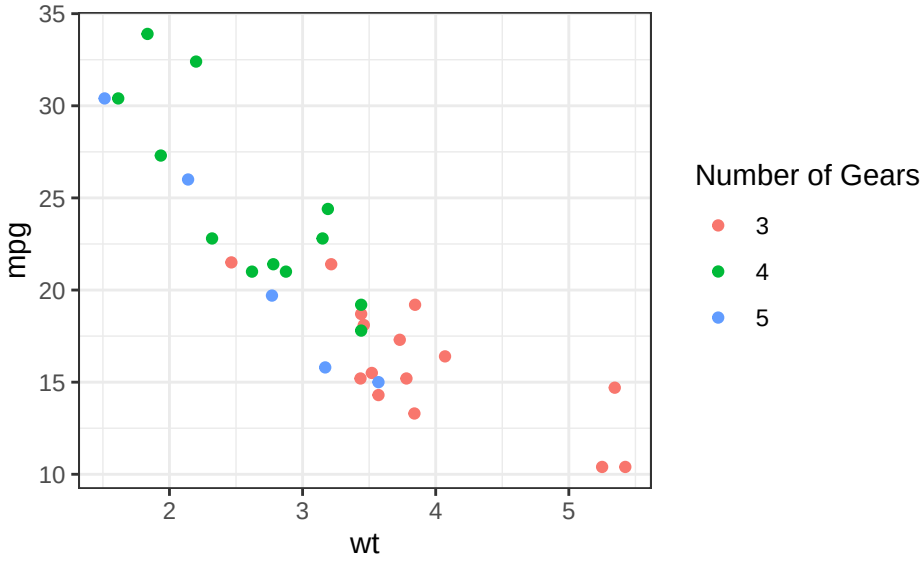
```
# Histogram of car miles per gallon
ggplot(data = mtcars, aes(x = mpg)) +
  geom_histogram(binwidth = 3, fill = "blue", color = "black", alpha = 0.7) +
  labs(title = "Histogram of Miles Per Gallon")
```



2.3 Customizing aesthetics

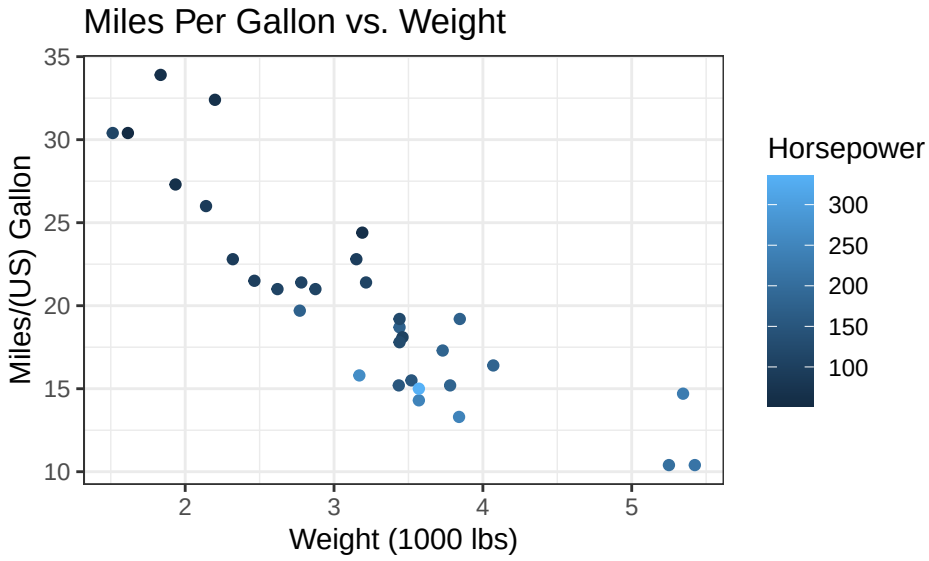
Map aesthetics to data variables for deeper insights:

```
# Scatter plot colored by number of gears
ggplot(mtcars, aes(x = wt, y = mpg, color = as.factor(gear))) +
  geom_point() +
  labs(color = "Number of Gears")
```



2.4 Adding labels

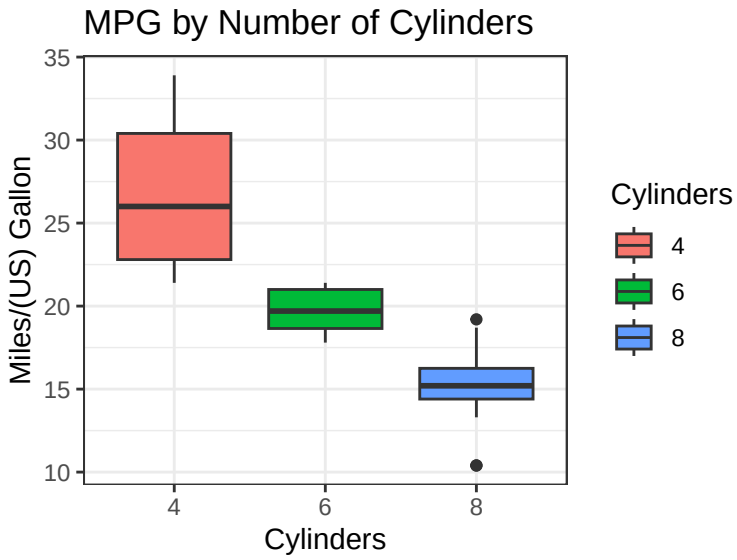
```
# Custom scatter plot
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point(aes(color = hp)) +
  labs(title = "Miles Per Gallon vs. Weight",
       x = "Weight (1000 lbs)",
       y = "Miles/(US) Gallon",
       color = "Horsepower")
```



2.5 Creating different types of plots

Experiment with different visual representations:

```
# Boxplot of miles per gallon by number of cylinders
ggplot(mtcars, aes(x = as.factor(cyl), y = mpg)) +
  geom_boxplot(aes(fill = as.factor(cyl))) +
  labs(title = "MPG by Number of Cylinders",
       x = "Cylinders",
       y = "Miles/(US) Gallon",
       fill = "Cylinders")
```

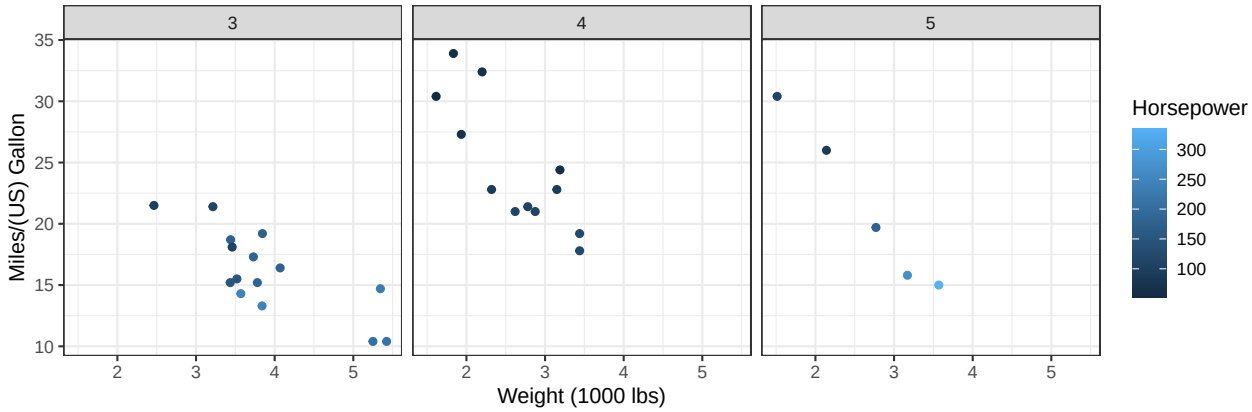


2.6 Faceting for Multiple Plots

Show multiple plots simultaneously:

```
# Scatter plots of mpg vs weight for each number of gears
ggplot(mtcars, aes(x = wt, y = mpg)) +
  geom_point(aes(color = hp)) +
```

```
  facet_wrap(~ gear) +
  labs(color = "Horsepower", x = "Weight (1000 lbs)", y = "Miles/(US) Gallon")
```



By leveraging the functionalities of `ggplot2`, you can generate informative and aesthetic visualizations that help decipher complex datasets like `mtcars`.

3 Data transformation (`dplyr`)

`dplyr` provides a suite of tools for efficiently manipulating datasets in R. It focuses on tools for working with data frames (or tibbles), its primary datatype.

3.1 Selecting columns with `select()`

This function lets you quickly isolate columns of interest:

```
# Selecting 'mpg' and 'hp' columns from mtcars
selected_data <- mtcars |>
  select(mpg, hp)

head(selected_data)
```

```
##                   mpg  hp
## Mazda RX4         21.0 110
## Mazda RX4 Wag     21.0 110
## Datsun 710        22.8  93
## Hornet 4 Drive    21.4 110
## Hornet Sportabout 18.7 175
## Valiant           18.1 105
```

3.2 Filtering rows with `filter()`

Isolate observations based on their values:

```
# Filtering cars that have more than 30 mpg
efficient_cars <- mtcars |>
  filter(mpg > 30)

head(efficient_cars)
```

```
##                mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Fiat 128      32.4   4 78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic   30.4   4 75.7  52 4.93 1.615 18.52  1  1    4    2
```

```
## Toyota Corolla 33.9   4 71.1  65 4.22 1.835 19.90  1  1    4    1
## Lotus Europa   30.4   4 95.1 113 3.77 1.513 16.90  1  1    5    2
```

3.3 Arranging rows with `arrange()`

Order the rows of your data:

```
# Ordering cars based on horsepower
sorted_cars <- mtcars |>
  arrange(hp)

head(sorted_cars)
```

```
##                 mpg cyl  disp hp drat    wt  qsec vs am gear carb
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1  1    4    2
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1  0    4    2
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1  1    4    1
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1  1    4    1
## Fiat X1-9      27.3   4  79.0 66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3 91 4.43 2.140 16.70  0  1    5    2
```

3.4 Creating new columns with `mutate()`

Generate new variables:

```
# Creating a new column 'hp_per_mpg' as a ratio of horsepower to mpg
modified_data <- mtcars |>
  mutate(hp_per_mpg = hp / mpg)

head(modified_data)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb hp_per_mpg
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4   5.238095
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4   5.238095
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1   4.078947
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1   5.140187
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2   9.358289
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1   5.801105
```

3.5 Summarizing data with `summarize()`

Generate summary statistics:

```
# Calculating mean mpg for the dataset
avg_mpg <- mtcars |>
  summarize(mean_mpg = mean(mpg))

avg_mpg
```

```
##   mean_mpg
## 1 20.09062
```

3.6 Group-wise operations

Perform operations on subsets of your data:

```r
# Calculating mean mpg for each number of cylinders
cyl_mpg <- mtcars |>
  summarize(mean_mpg = mean(mpg), .by = cyl)

cyl_mpg
```

```
##   cyl mean_mpg
## 1   6 19.74286
## 2   4 26.66364
## 3   8 15.10000
```

3.7 Combining data with joins

Join multiple datasets based on common columns:

```r
# Assuming a second data frame 'car_brands' that has a 'model' column and a 'brand' column
joined_data <- mtcars |>
  left_join(car_brands, by = "model")
```

3.8 Tips for efficient data transformation

- Use the pipe (`|>`) to chain operations, making your code more readable.
- Use `rename()` if you need to change column names.
- Remember functions like `n_distinct()` for counting distinct values or `tally()` for counting occurrences.
- Familiarize yourself with `dplyr`'s join functions (`inner_join()`, `full_join()`, etc.) for more complex merging operations.