# Chapter 2: Exploring Data

Shujia Wang

## Contents

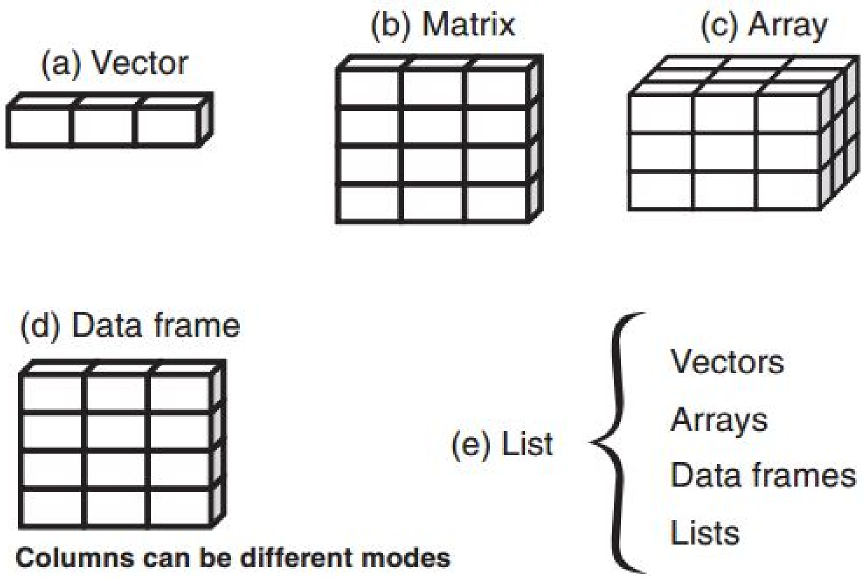
**Overview**

The main objective in this chapter is to introduce data science toolbox using *tidyverse* packages.

- ▷ Section 1: Structures of data in R (Kabacoff 2015)
- ▷ Section 2: Visualizing data via the ggplot2 package (Wickham and Grolemund 2016)
- ▷ Section 3: Wrangling data via the dplyr package (Wickham and Grolemund 2016)
- ▷ Section 4: How to tidy your data (Wickham and Grolemund 2016)
  - Understanding the concept of "tidy" data as a standardized data input format for all packages in the tidyverse.
  - Import Data with readr
  - Convert data to a tibble
  - Tidy data with tidyr
- ▷ Finally, we introduce the basic descriptive statistics in Section 5 (Ugarte, Militino, and Arnholt 2016).

# 1 Data Structure

**Objects for holding data**




## 1.1 Vectors

**Vectors**

**Vectors** are one-dimensional arrays that can hold numeric data, character data, or logical data

▷ The combine function c() is used to form the vector

```
> a<-c(1, 2, 5, 3, 4) # numeric vector
> b<-c("one","two","three") # character vector
> c<-c(TRUE,TRUE,FALSE) # logical vector
> a

[1] 1 2 5 3 4

> b[2]

[1] "two"
```

## 1.2 Matrices

**Matrices**

A **matrix** is a two-dimensional array in which each element has the *same mode* (numeric, character, or logical).

```
mymatrix <- matrix(vector, nrow=m, ncol=n,
          byrow=logical_value, dimnames=list(
          char_vector_rownames, char_vector_colnames))
```

```
> y<-matrix(0:9, nrow=2, ncol=5);y

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    2    4    6    8
[2,]    1    3    5    7    9

> y[,1]; y[2,]

[1] 0 1
[1] 1 3 5 7 9
```

### Matrices

```
> cells<-c(1,26,24,68)
> rnames<-c("R1","R2")
> cnames<-c("C1","C2")
> mymatrix<-matrix(cells,nrow=2,ncol=2,byrow=TRUE,
+           dimnames=list(rnames, cnames))
> mymatrix

   C1 C2
R1  1 26
R2 24 68
```

## 1.3 Arrays

### Arrays

**Arrays** are similar to matrices but can have more than two dimensions

```
myarray <- array(vector, dimensions, dimnames)
```

```
> dim1<-c("A1","A2")
> dim2<-c("B1","B2","B3")
> dim3<-c("C1","C2","C3","C4")
> z<-array(1:24,c(2,3,4),dimnames=list(dim1,dim2,dim3))
```

### Arrays

```
> z

, , C1

   B1 B2 B3
A1  1  3  5
A2  2  4  6
```

```
, , C2

   B1 B2 B3
A1  7  9 11
A2  8 10 12

, , C3

   B1 B2 B3
A1 13 15 17
A2 14 16 18

, , C4

   B1 B2 B3
A1 19 21 23
A2 20 22 24
```

## 1.4 Data Frame

**Data Frame: Most used structure in Statistics**

A **data frame** is more general than a matrix in that different columns can contain *different modes* of data (numeric, character, and so on)

```
mydata <- data.frame(col1, col2, col3,...)
```

where col1, col2, col3, and so on are column vectors of any type .

**Example**

```
> patientID<-c(1, 2, 3, 4)
> age<-c(25,34,28,52)
> diabetes<-c("Type1","Type2","Type1","Type1")
> status<-c("Poor","Improved","Excellent","Poor")
> patientdata<-data.frame(patientID,age,diabetes,status)
> patientdata

  patientID age diabetes    status
1         1  25    Type1      Poor
2         2  34    Type2  Improved
3         3  28    Type1 Excellent
4         4  52    Type1      Poor
```

**Frequently used: str() and summary()**

str(object) gives the structure of an object

```
> str(patientdata)

'data.frame': 4 obs. of  4 variables:
 $ patientID: num  1 2 3 4
 $ age      : num  25 34 28 52
 $ diabetes : Factor w/ 2 levels "Type1","Type2": 1 2 1 1
 $ status   : Factor w/ 3 levels "Excellent","Improved",..: 3 2 1 3
```

```
> summary(patientdata)

   patientID          age          diabetes        status
 Min.   :1.00   Min.   :25.00   Type1:3   Excellent:1
 1st Qu.:1.75   1st Qu.:27.25   Type2:1   Improved :1
 Median :2.50   Median :31.00             Poor     :2
 Mean   :2.50   Mean   :34.75
 3rd Qu.:3.25   3rd Qu.:38.50
 Max.   :4.00   Max.   :52.00
```

**Frequently used: head() and tail()**

head(object) lists the first part of an object. tail(object)lists the last part of an object. They are useful for quickly scanning large datasets.

```
> head(patientdata)

  patientID age diabetes    status
1         1  25    Type1      Poor
2         2  34    Type2  Improved
3         3  28    Type1 Excellent
4         4  52    Type1      Poor
```

**Specifying elements of a data frame**

```
> patientdata$age #variable age from patientdata

[1] 25 34 28 52

> patientdata[1:2]

  patientID age
1         1  25
2         2  34
3         3  28
4         4  52

> patientdata[c("diabetes", "status")]

  diabetes    status
1    Type1      Poor
2    Type2  Improved
3    Type1 Excellent
4    Type1      Poor
```

## 1.5 Factors

**Types of variables**

▷ Nominal variables

- are categorical, without an implied order. e.g. Diabetes (Type1, Type2)

▷ Ordinal variables

- categorical, imply order but not amount. e.g. Status (poor, improved, excellent)

▷ Continuous variables

- can take on any value within some range, and both order and amount are implied

▷ *Categorical* (nominal) and *ordered categorical* (ordinal) variables in R are called factors

### The use of factor()

```
> diabetes<-c("Type1","Type2","Type1","Type1")
> diabetes

[1] "Type1" "Type2" "Type1" "Type1"

> diabetes<-factor(diabetes)
> diabetes

[1] Type1 Type2 Type1 Type1
Levels: Type1 Type2

> levels(diabetes)

[1] "Type1" "Type2"

> class(diabetes)

[1] "factor"
```

### Ordered factor

```
> status<-c("Poor","Improved","Excellent","Poor")
> status1<-factor(status,order=TRUE)
> status1

[1] Poor      Improved  Excellent Poor
Levels: Excellent < Improved < Poor

> status2<-factor(status,order=TRUE,levels=c("Poor","Improved","Excellent"))
> status2

[1] Poor      Improved  Excellent Poor
Levels: Poor < Improved < Excellent

> status3<-ordered(status)
> status3

[1] Poor      Improved  Excellent Poor
Levels: Excellent < Improved < Poor
```

## 1.6 Lists

**List: the most flexible and richest structure in R**

Basically, a **list** is an ordered collection of objects (components).

A **list** allows you to gather a variety of (possibly unrelated) objects under one name.

**list()**

```
mylist<-list(object1,object2,...)
```

or

```
mylist<-list(name1=object1,name2=object2,...)
```

### Example of a list

```
> g<-"My First List"
> h<-c(25, 26, 18, 39)
> j<-matrix(1:10,nrow=2)
> k<-c("one", "two", "three")
> mylist<-list(title=g,ages=h,j,k)
> mylist

$title
[1] "My First List"

$ages
[1] 25 26 18 39

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

[[4]]
[1] "one"   "two"   "three"

> mylist[[2]]

[1] 25 26 18 39

> mylist[["ages"]]

[1] 25 26 18 39
```

### Data types summary

### What type is your data?

| Data structure | Instruction in R | Description |
|---|---|---|
| vector | `c()` | Sequence of elements of the **same nature**. |
| matrix | `matrix()` | Two-dimensional table of elements of the **same nature**. |
| multidimensional table | `array()` | More general than a matrix; table with several dimensions. |
| list | `list()` | Sequence of R structures of any (and possibly different) nature. |
| individual×variable table | `data.frame()` | Two-dimensional table. The columns can be of different natures, but must have the same length. |
| factor | `factor(), ordered()` | Vector of character strings associated with a modality table. |
| dates | `as.Date()` | Vector of dates. |
| time series | `ts()` | Values of a variable observed at several time points. |

| Type | Description |
|---|---|
| `class()` | Class from which object inherits (vector, matrix, function, logical, list, ⋯) |
| `mode()` | Numeric, character, logical, ⋯ |
| `storage.mode()` | Mode used by R to store object (double, integer, character, logical, ⋯) |
| `is.function()` | Logical (TRUE if function) |
| `is.na()` | Logical (TRUE if missing) |
| `names()` | Names associated with object |
| `dimnames()` | Names for each dim of array |
| `attributes()` | Names, class, etc. |

# 2 Data visualisation with ggplot2

## 2.1 Data and Aesthetics mapping

**Introduction to ggplot2**

ggplot2 is a powerful and a flexible R package, implemented by *Hadley Wickham*, for producing elegant graphics.

The *gg* means *Grammar of Graphics:*

**"Plot = data + Aesthetics + Geometry"**

**data** is a data frame

**Aesthetics** is used to indicate x and y variables. It can be also used to control the color, the size or the shape of points, the height of bars, etc.....

**Geometry** corresponds to the *type of graphics* (histogram, box plot, line plot, density plot, dot plot, ....)

**Dataset mpg**

Contains observations collected by the US Environment Protection Agency on 38 models of cars.

```
> 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(~ f        18    29 p     comp~
2 audi         a4      1.8  1999     4 manua~ f        21    29 p     comp~
3 audi         a4      2    2008     4 manua~ f        20    31 p     comp~
4 audi         a4      2    2008     4 auto(~ f        21    30 p     comp~
5 audi         a4      2.8  1999     6 auto(~ f        16    26 p     comp~
6 audi         a4      2.8  1999     6 manua~ f        18    26 p     comp~
```

**Variables involved in mpg**

**hwy** Fuel efficiency on the highway, in miles per gallon

**year** year of manufacture

**displ** Engine size, in liters

**model** model name

**drv** f = front-wheel drive, r = rear wheel drive, 4 = 4wd

**trans** type of transmission

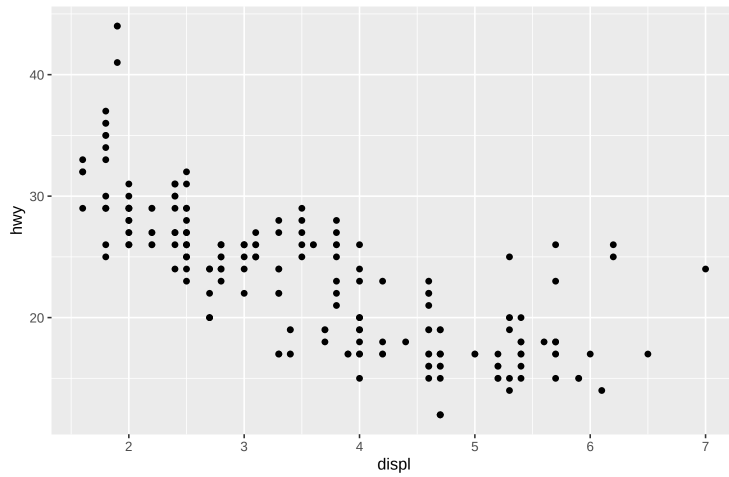
**cyl** number of cylinders

**cty** city miles per gallon

**class** "type" of car

**Creating a ggplot**

```
> ggplot(data = mpg) +
+       geom_point(mapping = aes(x = displ, y = hwy))
```

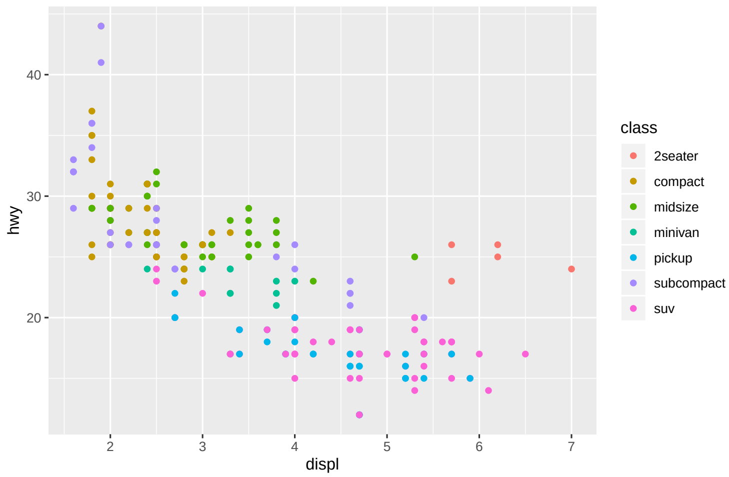



### Save ggplots

```
# Print the plot to a pdf file

   pdf("myplot.pdf")
   myplot <- ggplot(...)
   print(myplot)
   dev.off()

# Print the plot to a png file

   png("myplot.png")
   print(myplot)
   dev.off()

# Save the plot to a pdf

   ggsave("myplot.pdf")

# OR save it to png file

   ggsave("myplot.png")
```

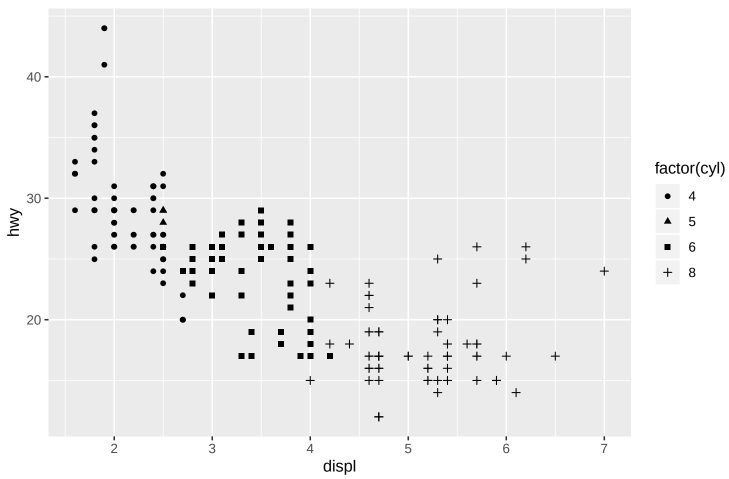
### with colors

```
> ggplot(data = mpg) +
+  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```



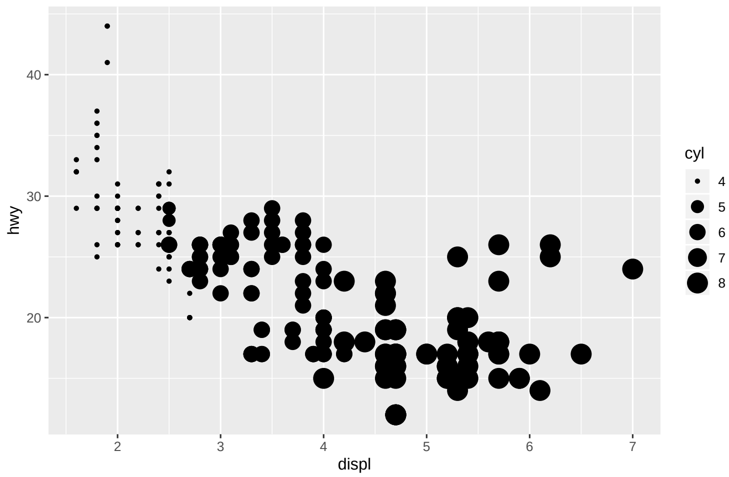

## Shape of the points

```
> ggplot(data = mpg) +
+   geom_point(mapping = aes(x = displ, y = hwy, shape = factor(cyl)))
```



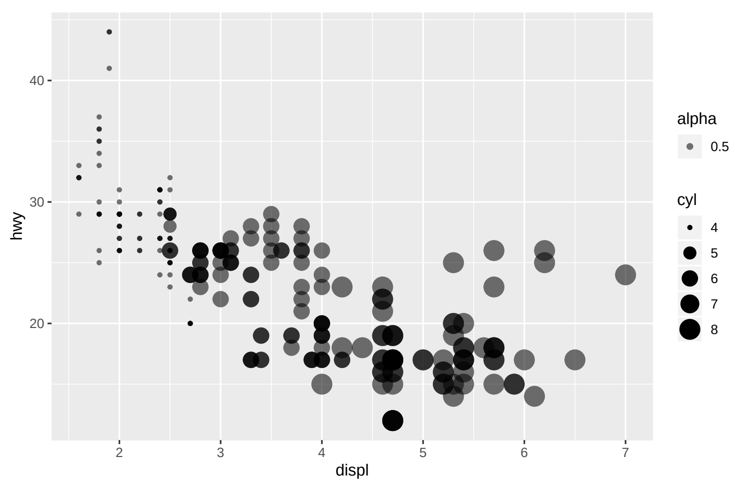

## Variable size of points

```
> ggplot(data = mpg) +
+   geom_point(mapping = aes(x = displ, y = hwy, size = cyl))
```



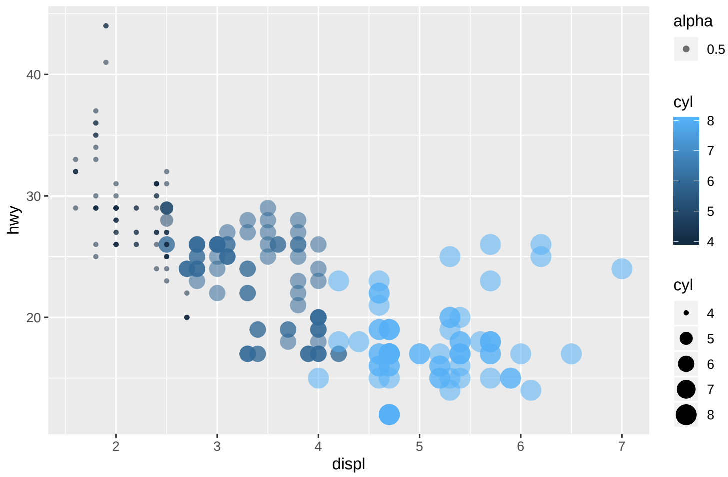

## Variable points: size and transparency

```
> ggplot(data = mpg) +
+    geom_point(mapping = aes(x = displ, y = hwy,
+             size = cyl, alpha = 0.5))
```



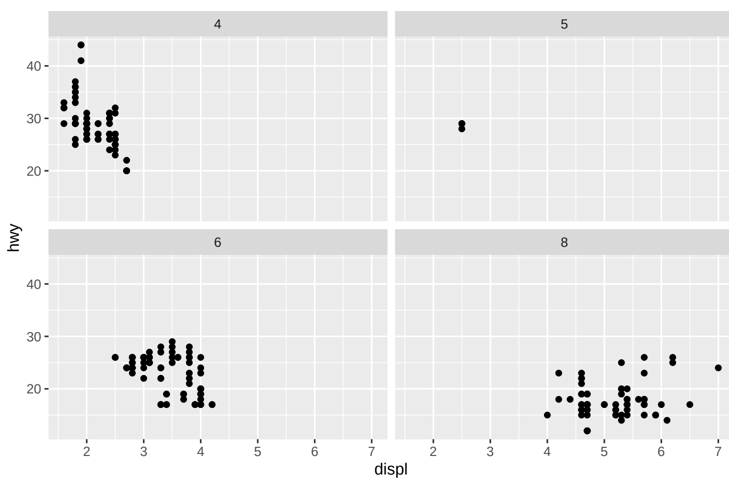

## Variable points: size, colors and transparency

```
> ggplot(data = mpg) +
+   geom_point(mapping = aes(x = displ, y = hwy,
+     size = cyl, color = cyl, alpha = 0.5))
```




### Facets

```
> ggplot(data = mpg) +
+   geom_point(mapping = aes(x = displ, y = hwy)) +
+   facet_wrap(~ cyl, nrow = 2)
```




## 2.2 Geometric Objects

### Plot one variable

▷ For one continuous variable:

**geom_histogram()** for histogram plot
**geom_area()** for area plot
**geom_density()** for density plot

**geom_dotplot()** for dot plot

**geom_freqpoly()** for frequency polygon

**stat_ecdf()** for empirical cumulative density function

**stat_qq()** for quantile - quantile plotting

▷ For one discrete variable:

**geom_bar()** for bar plot

### Plot two variables

**geom_point()** for scatter plot

**geom_smooth()** for adding smoothed line such as regression line

**geom_boxplot()** for comparison of continuous y and discrete x

**geom_quantile()** for adding quantile lines

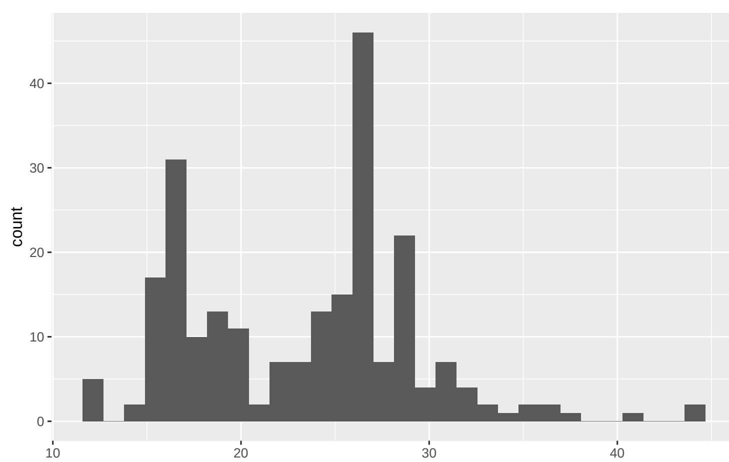
**geom_rug()** for adding a marginal rug

**geom_jitter()** for avoiding overplotting

**geom_text()** for adding textual annotations

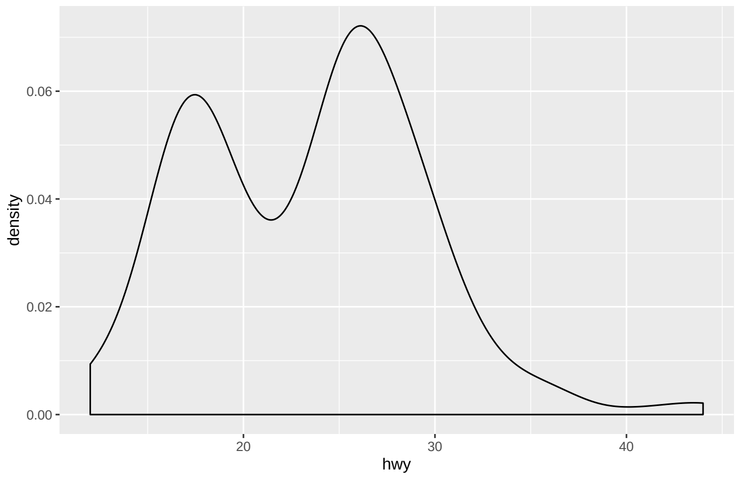
### Plot of hwy: histogram

```
> ggplot(data = mpg) +
+   geom_histogram(aes(x = hwy))
```



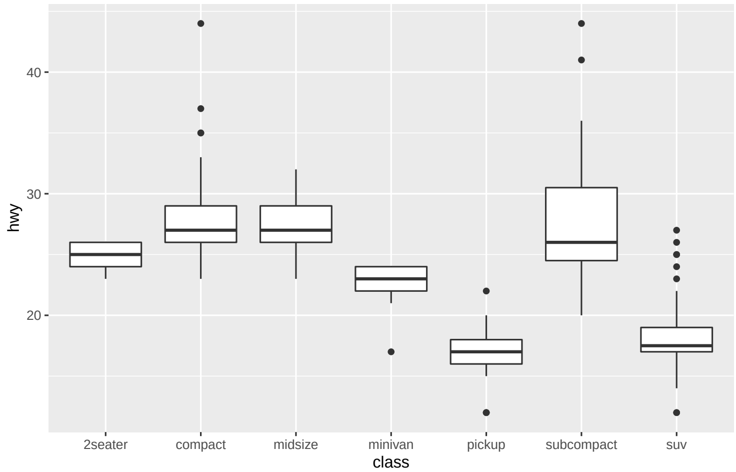

### Plot of hwy: density

```
> ggplot(data = mpg) +
+   geom_density(aes(x = hwy))
```



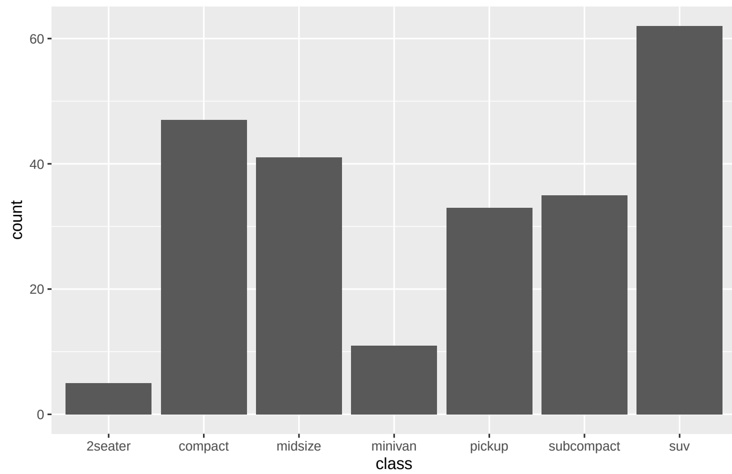

### Plot of hwy: box plot for comparison

```
> ggplot(data = mpg) +
+   geom_boxplot(aes(x = class, y = hwy))
```



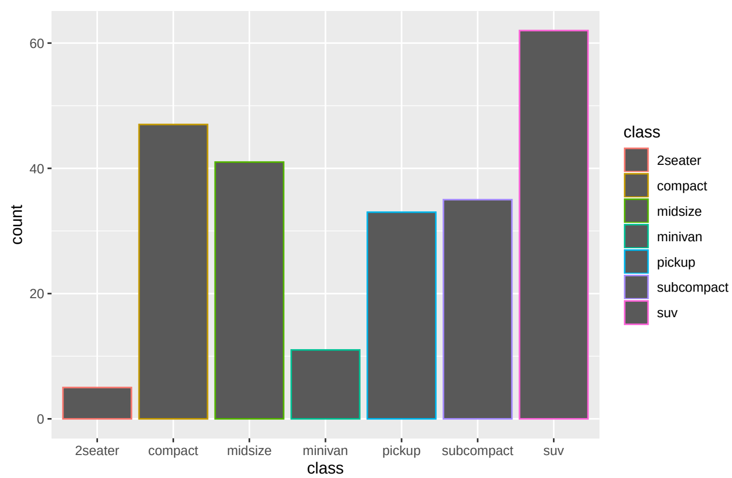

### Plot of class: bar chart

```
> ggplot(data = mpg) +
+   geom_bar(aes(x = class))
```



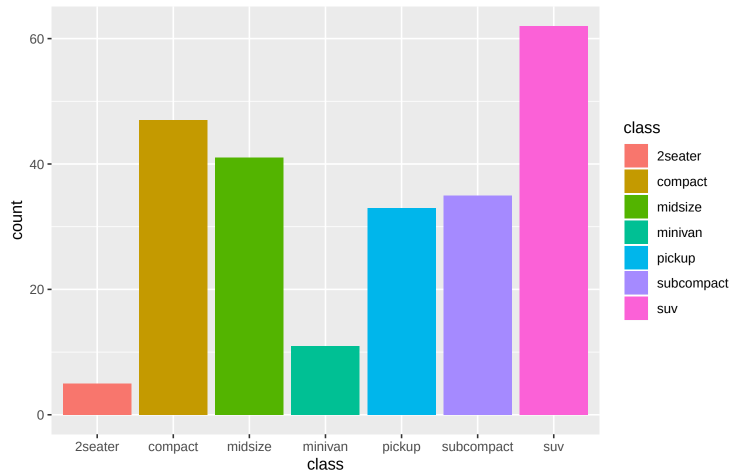

### Plot of class: bar chart with colors

```
> ggplot(data = mpg) +
+   geom_bar(aes(x = class, color = class))
```




### Plot of class: fill in colors into bars

```
> ggplot(data = mpg) +
+     geom_bar(aes(x = class, fill = class))
```




**Plot two variables: scatter plot**

**Scatter Plots:**

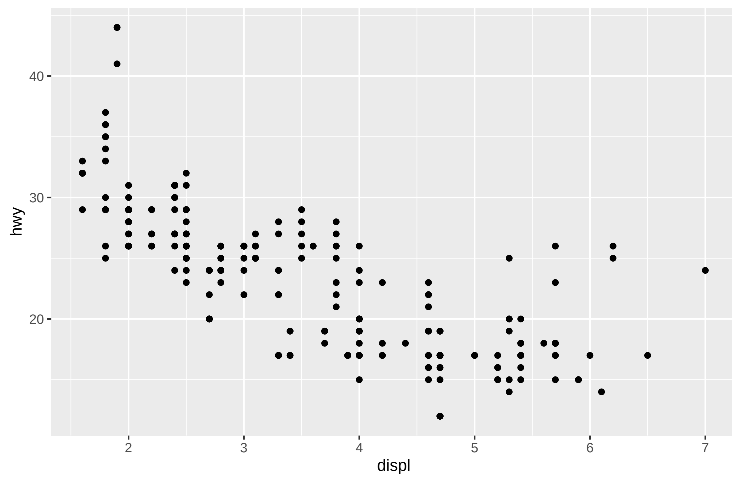
- ▷ Key function: geom_point()
- ▷ Key arguments to customize the plot: alpha, color, fill, shape and size

**Add regression line or smoothed conditional mean:**

- ▷ Key functions: geom_smooth() and geom_abline()
- ▷ Key arguments to customize the plot: alpha, color, fill, shape, linetype and size

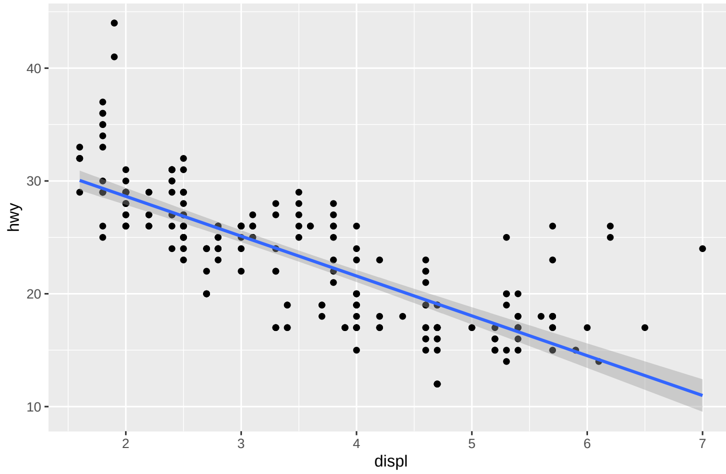
### Scatter plot

```
> b <- ggplot(mpg, aes(x = displ, y = hwy))
> b + geom_point()
```



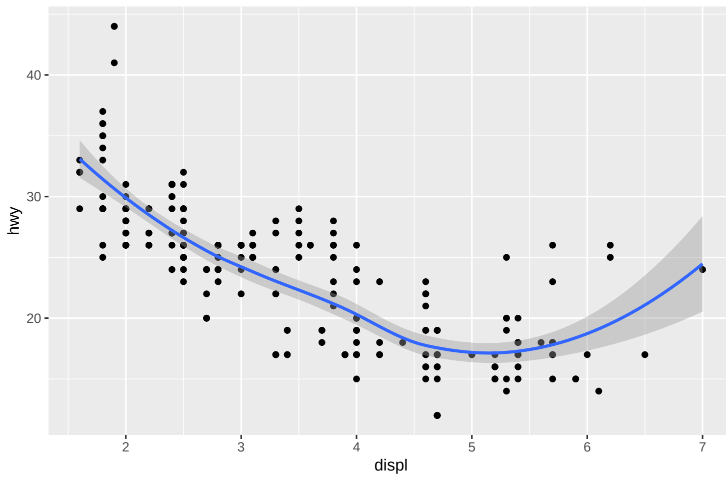
### Scatter plot with regression line

```
> b + geom_point() + geom_smooth(method = lm)
```



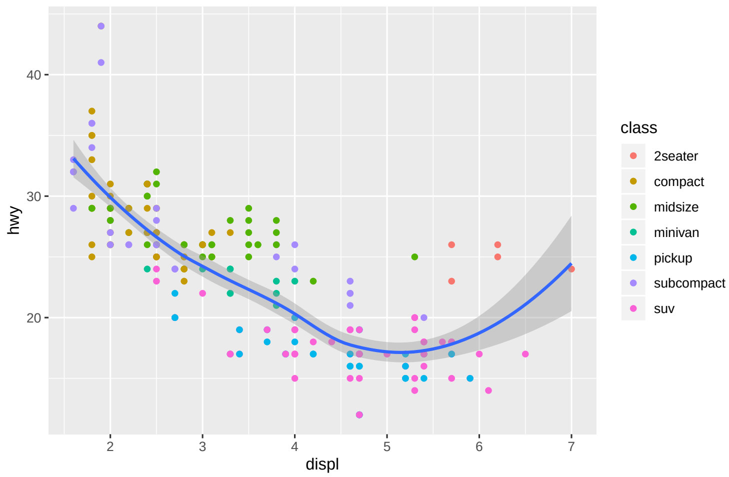

### Loess method: local regression fitting

```
> b + geom_point() + geom_smooth()
```



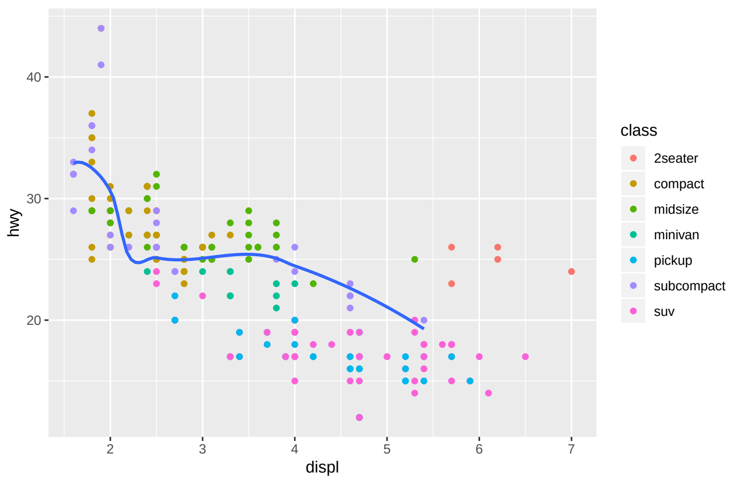

### Local mappings for a layer

```
> b + geom_point(mapping = aes(color = class)) + geom_smooth()
```




**Displays just a subset of the dataset**

```
> b + geom_point(mapping = aes(color = class)) +
+ geom_smooth(data = filter(mpg, class == "subcompact"),
+ se = FALSE)
```



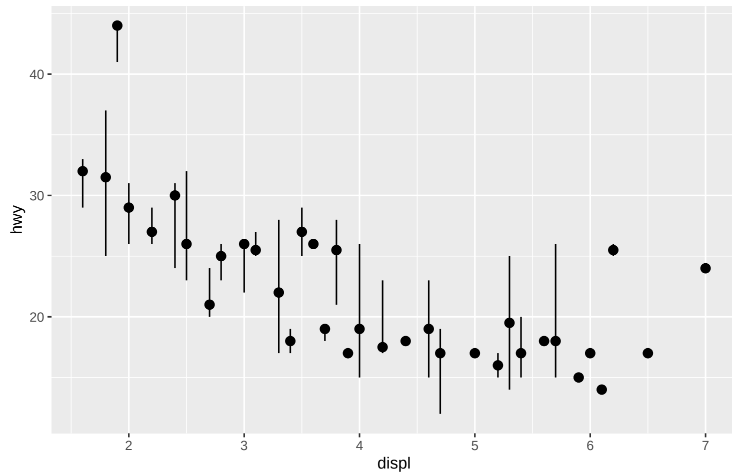

## 2.3 Statistical Transformations

**stat**

- ▷ The algorithm used to calculate new values for a graph is called a *stat*
- ▷ Some plots visualize a transformation of the original data set. In this case, an alternative way to build a layer is to use *stat_*()* functions.
- ▷ geom_bar() = stat_count(): *?geom_bar* shows the default value for *stat* is "*count*"

### Statistical transformation in your code

```
> ggplot(data = mpg) +
+ stat_summary( mapping = aes(x = displ, y = hwy),
+     fun.ymin = min, fun.ymax = max, fun.y = median )
```



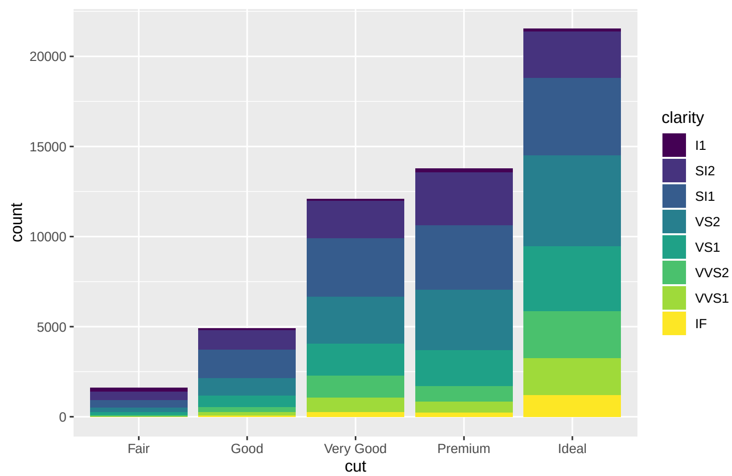

## 2.4 Position Adjustments

### Position in bar chart

What will happen if you map the *fill* aesthetic to another variable, like *clarity?*

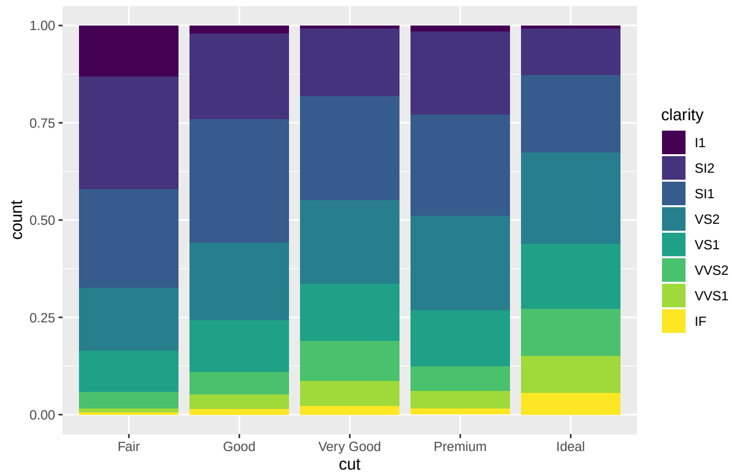
The stacking is performed automatically by the position adjustment specified by the *position* argument.

```
> ggplot(data = diamonds) + geom_bar(mapping = aes(x = cut, fill = clarity))
```



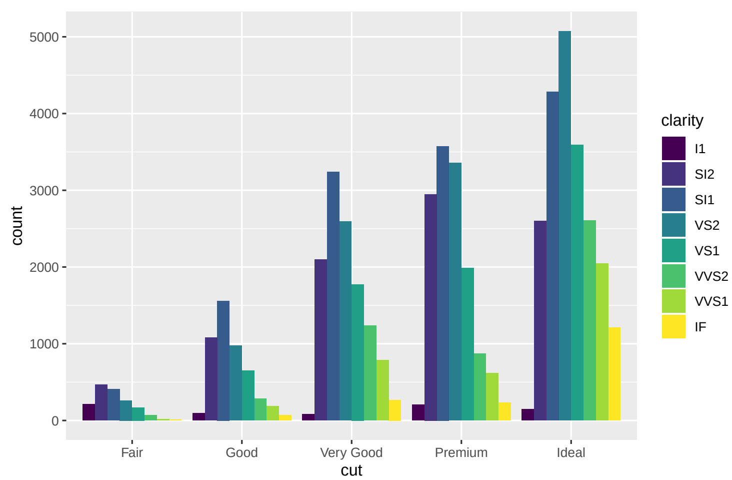
**bar chart: position = "fill"**

```
> ggplot(data = diamonds) +
+ geom_bar( mapping = aes(x = cut, fill = clarity),
+ position = "fill" )
```



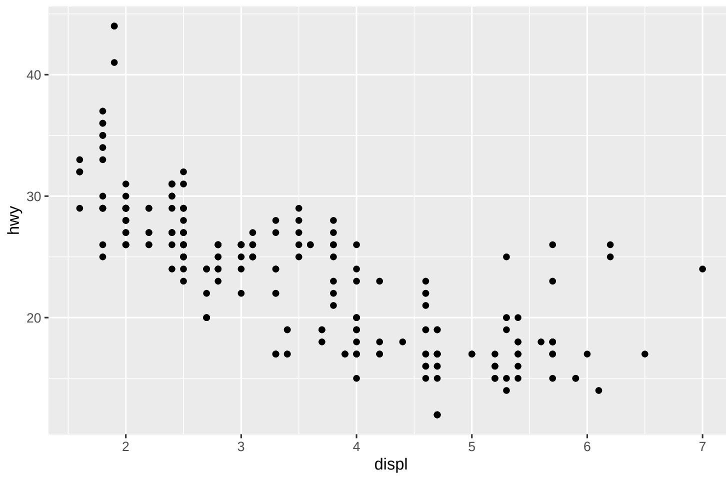

**bar chart: position = "dodge"**

```
> ggplot(data = diamonds) +
+ geom_bar( mapping = aes(x = cut, fill = clarity),
+ position = "dodge" )
```

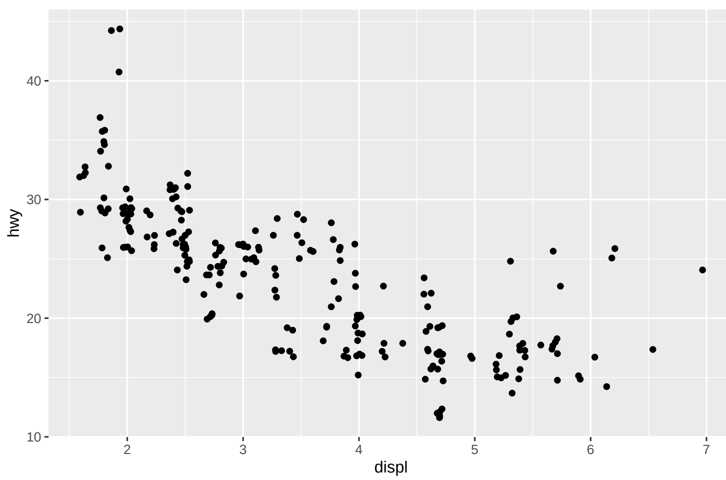



**Position in scatter plots**

Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset? This is known as *overplotting*.




**scatter plots: position = "jitter"**




# 3 Data transformation with dplyr

## 3.1 Introduction

**Aims**

In this section, we will learn about *data wrangling* , the art of getting your data into R for visualization and modeling.

- ▷ how to transform your data using the *dplyr* package
- ▷ a new dataset on flights departing New York City in 2013.

**Prerequisites**

This section and the next will explore the package library(tidyverse).

Core packages in *tidyverse* include:

**dplyr** for data manipulation.

**tidyr** for data tidying.

**tibble** for tibbles, a modern re-imagining of data frames

**readr** for data import.

**nycights13**

This dataset contains all 336,776 flights that departed from New York City in 2013 (From the *US Bureau of Transportation Statistics*).

```
> library(nycflights13)
> flights

# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

**Tibble is a new format of data frame**

- ▷ Tibbles are a modern take on data frames.
- ▷ They keep the features that have stood the test of time, and drop the features that used to be convenient but are now frustrating (i.e. converting character vectors to factors).

**Type:**

**int** stands for integers.

**dbl** stands for doubles, or real numbers.

**chr** stands for character vectors, or strings.

**dttm** stands for date-times (a date + a time)

**dplyr Basics**

**filter()** Pick observations by their values.

**arrange()** Reorder the rows.

**select()** Pick variables by their names.

**mutate()** Create new variables with functions of existing variables.

**summarize()** Collapse many values down to a single summary.

**group_by()** Group rows.

## 3.2 Filter Rows with filter()

***filter()* allows you to subset observations**

All dplyr work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

**Display filtered results**

```
> filter(flights, month == 1, day == 1)

# A tibble: 842 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 832 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 <dttm>
```

**Display and save results**

```
> (dec25 <- filter(flights, month == 12, day == 25))
```

```
# A tibble: 719 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013    12    25      456            500        -4      649
 2  2013    12    25      524            515         9      805
 3  2013    12    25      542            540         2      832
 4  2013    12    25      546            550        -4     1022
 5  2013    12    25      556            600        -4      730
 6  2013    12    25      557            600        -3      743
 7  2013    12    25      557            600        -3      818
 8  2013    12    25      559            600        -1      855
 9  2013    12    25      559            600        -1      849
10  2013    12    25      600            600         0      850
# ... with 709 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 <dttm>
```

### Logical operators

Boolean operators:

- ▷ "&" is "and,"
- ▷ "|" is "or,"
- ▷ "!" is "not,"
- ▷ "!=" is "not equal,"
- ▷ " ==" is "equal,"
- ▷ "x %in% y" select every row where x is one of the values in y.

```
filter(flights, month == 11 | month == 12)
nov_dec <- filter(flights, month %in% c(11, 12))
```

### Missing values

One important feature of R that can make comparison tricky is missing values, or NAs ( "not availables" ).

```
> df <- tibble(x = c(1, NA, 3))
> filter(df, x > 1)

# A tibble: 1 x 1
      x
  <dbl>
1     3

> filter(df, is.na(x) | x > 1)

# A tibble: 2 x 1
      x
  <dbl>
1    NA
2     3
```

## 3.3 Arrange Rows with arrange()

### arrange()

- *arrange()* works similarly to *filter()* except that instead of selecting rows, it *changes their order*.
- It takes a data frame and a set of column names (or more complicated expressions) to order by.

#### Arrange by a set of column names

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

# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
#   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

#### Use desc() to reorder by a column in descending order

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

# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     9      641            900      1301     1242
 2  2013     6    15     1432           1935      1137     1607
 3  2013     1    10     1121           1635      1126     1239
 4  2013     9    20     1139           1845      1014     1457
 5  2013     7    22      845           1600      1005     1044
 6  2013     4    10     1100           1900       960     1342
 7  2013     3    17     2321            810       911      135
 8  2013     6    27      959           1900       899     1236
 9  2013     7    22     2257            759       898      121
10  2013    12     5      756           1700       896     1058
# ... with 336,766 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 <dttm>
```

## 3.4 Select Columns with select()

### select()

*select()* allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

```
> # Select columns by name
> select(flights, year, month, day)

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

### Select all columns between year and day

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

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

### Select all columns except those from year to day

```
> select(flights, -(year:day))

# A tibble: 336,776 x 16
   dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
      <int>          <int>     <dbl>    <int>          <int>     <dbl>
 1      517            515         2      830            819        11
 2      533            529         4      850            830        20
 3      542            540         2      923            850        33
```

```
 4      544           545        -1     1004           1022       -18
 5      554           600        -6      812            837       -25
 6      554           558        -4      740            728        12
 7      555           600        -5      913            854        19
 8      557           600        -3      709            723       -14
 9      557           600        -3      838            846        -8
10      558           600        -2      753            745         8
# ... with 336,766 more rows, and 10 more variables: carrier <chr>,
#   flight <int>, tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#   distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

#### Select anything that starts with "d"

```
> select(flights, starts_with("d"))

# A tibble: 336,776 x 5
     day dep_time dep_delay dest  distance
   <int>    <int>     <dbl> <chr>    <dbl>
 1     1      517         2 IAH       1400
 2     1      533         4 IAH       1416
 3     1      542         2 MIA       1089
 4     1      544        -1 BQN       1576
 5     1      554        -6 ATL        762
 6     1      554        -4 ORD        719
 7     1      555        -5 FLL       1065
 8     1      557        -3 IAD        229
 9     1      557        -3 MCO        944
10     1      558        -2 ORD        733
# ... with 336,766 more rows
```

#### Select anything related to delays

```
> select(flights, ends_with("delay"))

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

## 3.5 Add New Variables with mutate()

#### Add new variables *gain* (think about why?)

```
> flights_sml <- select(flights, year:day, ends_with("delay"),
+                 distance, air_time)
> mutate(flights_sml,
+                  gain = arr_delay - dep_delay,
+         speed = distance / air_time * 60)

# A tibble: 336,776 x 9
    year month   day dep_delay arr_delay distance air_time  gain speed
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9  370.
 2  2013     1     1         4        20     1416      227    16  374.
 3  2013     1     1         2        33     1089      160    31  408.
 4  2013     1     1        -1       -18     1576      183   -17  517.
 5  2013     1     1        -6       -25      762      116   -19  394.
 6  2013     1     1        -4        12      719      150    16  288.
 7  2013     1     1        -5        19     1065      158    24  404.
 8  2013     1     1        -3       -14      229       53   -11  259.
 9  2013     1     1        -3        -8      944      140    -5  405.
10  2013     1     1        -2         8      733      138    10  319.
# ... with 336,766 more rows
```

**New added variable can be used**

```
> mutate(flights_sml,
+ gain = arr_delay - dep_delay,
+ hours = air_time / 60,
+ gain_per_hour = gain / hours )

# A tibble: 336,776 x 10
    year month   day dep_delay arr_delay distance air_time  gain hours
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
 1  2013     1     1         2        11     1400      227     9 3.78
 2  2013     1     1         4        20     1416      227    16 3.78
 3  2013     1     1         2        33     1089      160    31 2.67
 4  2013     1     1        -1       -18     1576      183   -17 3.05
 5  2013     1     1        -6       -25      762      116   -19 1.93
 6  2013     1     1        -4        12      719      150    16 2.5
 7  2013     1     1        -5        19     1065      158    24 2.63
 8  2013     1     1        -3       -14      229       53   -11 0.883
 9  2013     1     1        -3        -8      944      140    -5 2.33
10  2013     1     1        -2         8      733      138    10 2.3
# ... with 336,766 more rows, and 1 more variable: gain_per_hour <dbl>
```

## 3.6 Summarize variables with summarize()

**This and the next subsections will cover**

1. Summarize data via summarize()
2. summarize() together with group_by()
3. Combining multiple operations with the pipe: %>%
4. Treating of missing values

5. Grouping by multiple variables

**summarize() collapses a data frame to a single row**

```
> summarize(flights, delay = mean(dep_delay, na.rm = TRUE))

# A tibble: 1 x 1
  delay
  <dbl>
1  12.6

> summarize(flights, delay.sd = sd(dep_delay, na.rm = TRUE))

# A tibble: 1 x 1
  delay.sd
     <dbl>
1     40.2
```

***summarize()* is useful togather with *group_by()***

```
> by_day <- group_by(flights, year, month, day)
> summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day delay
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

## 3.7 Operations with the pipe %>%

**Combining multiple operations with the pipe: %>%**

Imagine that we want to *explore the relationship between the distance and average delay for each location.* you might write code like this:

```
> by_dest <- group_by(flights, dest)
>   delay <- summarize(by_dest,
+   count = n(),
+   dist = mean(distance, na.rm = TRUE),
+   delay = mean(arr_delay, na.rm = TRUE) )
> delay <- filter(delay, count > 20, dest != "HNL")
```

There are three steps to prepare this data:

1. Group flights by destination.
2. Summarize to compute distance, average delay, and number of flights.
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

It is frustrating, because we have to give each intermediate data frame a name, even though we don't care about it.

**The pipe operator: %>%**

The pipe operator (%>%) allows us to chain together *dplyr* data wrangling functions.
The pipe operator can be read as "then" :

group_by(dest) **then** ▷ summarize **then** ▷filter

And then you can employ *delays* to display or model.

```
> delays <- flights %>%
+    group_by(dest) %>%
+    summarize(
+      count = n(),
+      dist = mean(distance, na.rm = TRUE),
+      delay = mean(arr_delay, na.rm = TRUE)
+      ) %>%
+ filter(count > 20, dest != "HNL")
```

**Missing values?**

Too much NA, missing values!

```
> flights %>%
+   group_by(year, month, day) %>%
+   summarize(mean = mean(dep_delay))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1    NA
 2  2013     1     2    NA
 3  2013     1     3    NA
 4  2013     1     4    NA
 5  2013     1     5    NA
 6  2013     1     6    NA
 7  2013     1     7    NA
 8  2013     1     8    NA
 9  2013     1     9    NA
10  2013     1    10    NA
# ... with 355 more rows
```

### Missing values? add “na.rm = TRUE”

```
> flights %>%
+   group_by(year, month, day) %>%
+   summarize(mean = mean(dep_delay, na.rm = TRUE))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.5
 2  2013     1     2 13.9
 3  2013     1     3 11.0
 4  2013     1     4  8.95
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.55
 9  2013     1     9  2.28
10  2013     1    10  2.84
# ... with 355 more rows
```

### Removing the cancelled flights

You can remove the cancelled flights by first, and save this dataset so we can reuse it.

```
> not_cancelled <- flights %>%
+         filter(!is.na(dep_delay), !is.na(arr_delay))
```

Summarize the *not_cancelled* data.

```
> not_cancelled %>%
+   group_by(year, month, day) %>%
+   summarize(mean = mean(dep_delay))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day  mean
   <int> <int> <int> <dbl>
 1  2013     1     1 11.4
 2  2013     1     2 13.7
 3  2013     1     3 10.9
 4  2013     1     4  8.97
 5  2013     1     5  5.73
 6  2013     1     6  7.15
 7  2013     1     7  5.42
 8  2013     1     8  2.56
 9  2013     1     9  2.30
10  2013     1    10  2.84
# ... with 355 more rows
```

### How many flights left before 5am?

These usually indicate delayed flights from the previous day.

```
> not_cancelled %>%
+         group_by(year, month, day) %>%
+         summarize(n_early = sum(dep_time < 500))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day n_early
   <int> <int> <int>   <int>
 1  2013     1     1       0
 2  2013     1     2       3
 3  2013     1     3       4
 4  2013     1     4       3
 5  2013     1     5       3
 6  2013     1     6       2
 7  2013     1     7       2
 8  2013     1     8       1
 9  2013     1     9       3
10  2013     1    10       3
# ... with 355 more rows
```

### What proportion are delayed by more than an hour?

```
> not_cancelled %>%
+         group_by(year, month, day) %>%
+         summarize(hour_perc = mean(arr_delay > 60))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day hour_perc
   <int> <int> <int>     <dbl>
 1  2013     1     1    0.0722
 2  2013     1     2    0.0851
 3  2013     1     3    0.0567
 4  2013     1     4    0.0396
 5  2013     1     5    0.0349
 6  2013     1     6    0.0470
 7  2013     1     7    0.0333
 8  2013     1     8    0.0213
 9  2013     1     9    0.0202
10  2013     1    10    0.0183
# ... with 355 more rows
```

### Grouping by Multiple Variables: per_day

```
> daily <- group_by(flights, year, month, day)
> (per_day <- summarize(daily, flights = n()))

# A tibble: 365 x 4
# Groups:   year, month [12]
    year month   day flights
   <int> <int> <int>   <int>
 1  2013     1     1     842
 2  2013     1     2     943
 3  2013     1     3     914
```

```
 4  2013     1     4     915
 5  2013     1     5     720
 6  2013     1     6     832
 7  2013     1     7     933
 8  2013     1     8     899
 9  2013     1     9     902
10  2013     1    10     932
# ... with 355 more rows
```

### Grouping by Multiple Variables: per_month

```
> daily <- group_by(flights, year, month, day)
> (per_month <- summarize(per_day, flights = sum(flights)))

# A tibble: 12 x 3
# Groups:   year [1]
    year month flights
   <int> <int>   <int>
 1  2013     1   27004
 2  2013     2   24951
 3  2013     3   28834
 4  2013     4   28330
 5  2013     5   28796
 6  2013     6   28243
 7  2013     7   29425
 8  2013     8   29327
 9  2013     9   27574
10  2013    10   28889
11  2013    11   27268
12  2013    12   28135
```

### Grouped Mutates and Filters

Grouping is most useful in conjunction with *summarize()*, but you can also do convenient operations with *mutate()* and *filter()*.

```
> # Find the worst members of each group
> flights_sml %>%
+         group_by(year, month, day) %>%
+         filter(rank(desc(arr_delay)) < 10)

# A tibble: 3,306 x 7
# Groups:   year, month, day [365]
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1       853       851      184       41
 2  2013     1     1       290       338     1134      213
 3  2013     1     1       260       263      266       46
 4  2013     1     1       157       174      213       60
 5  2013     1     1       216       222      708      121
 6  2013     1     1       255       250      589      115
 7  2013     1     1       285       246     1085      146
 8  2013     1     1       192       191      199       44
 9  2013     1     1       379       456     1092      222
10  2013     1     2       224       207      550       94
# ... with 3,296 more rows
```

### Find all groups bigger than a threshold

```
> popular_dests <- flights %>%
+         group_by(dest) %>%
+         filter(n() > 365)
> popular_dests

# A tibble: 332,577 x 19
# Groups:   dest [77]
    year month   day dep_time sched_dep_time dep_delay arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>
 1  2013     1     1      517            515         2      830
 2  2013     1     1      533            529         4      850
 3  2013     1     1      542            540         2      923
 4  2013     1     1      544            545        -1     1004
 5  2013     1     1      554            600        -6      812
 6  2013     1     1      554            558        -4      740
 7  2013     1     1      555            600        -5      913
 8  2013     1     1      557            600        -3      709
 9  2013     1     1      557            600        -3      838
10  2013     1     1      558            600        -2      753
# ... with 332,567 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 <dttm>
```

### Standardize to compute per group metrics

```
> popular_dests %>%
+         filter(arr_delay > 0) %>%
+         mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
+         select(year:day, dest, arr_delay, prop_delay)

# A tibble: 131,106 x 6
# Groups:   dest [77]
    year month   day dest  arr_delay prop_delay
   <int> <int> <int> <chr>     <dbl>      <dbl>
 1  2013     1     1 IAH          11  0.000111
 2  2013     1     1 IAH          20  0.000201
 3  2013     1     1 MIA          33  0.000235
 4  2013     1     1 ORD          12  0.0000424
 5  2013     1     1 FLL          19  0.0000938
 6  2013     1     1 ORD           8  0.0000283
 7  2013     1     1 LAX           7  0.0000344
 8  2013     1     1 DFW          31  0.000282
 9  2013     1     1 ATL          12  0.0000400
10  2013     1     1 DTW          16  0.000116
# ... with 131,096 more rows
```

# 4 Data Tidy with tidyr

### Aims

This section will cover:

1. Tibbles with tibble
2. Import Data with readr
3. Tidy data with tidyr
   (a) What is tidy data?
   (b) A modern data frame of *tibble*
   (c) Tidy data with tidyr

## 4.1 Tibbles with tibble

### Creating Tibbles

Tibbles are data frames, but they tweak some older behaviors to make life a little easier.

**as_tibble():** change a data frame to a tibble, e.g. *as_tibble(iris)*

**tibble():** create a new tibble from individual vectors

**tribble():** is customized for data entry in code: column headings are defined by formulas (i.e., they start with ~), and entries are separated by commas.

### tibble()

```
> tibble(
+ x = 1:5,
+ y = 1,
+ z = x ^ 2 + y
+ )

# A tibble: 5 x 3
      x     y     z
  <int> <dbl> <dbl>
1     1     1     2
2     2     1     5
3     3     1    10
4     4     1    17
5     5     1    26
```

### tribble()

```
> tribble(
+ ~x, ~y, ~z,
+ #--/--/----
+ "a", 2, 3.6,
+ "b", 1, 8.5
+ )

# A tibble: 2 x 3
  x         y     z
  <chr> <dbl> <dbl>
1 a         2   3.6
2 b         1   8.5
```

### Main differences in *tibble* versus a classic *data.frame*

There are two main differences in the usage of a tibble versus a classic data.frame:

- ▷ printing: shows only the first 10 rows, and all the columns that fit on screen.
- ▷ and subsetting: to pull out a single variable, $ and [["name"]] or [[position]].

### Extract from tibble

```
> df <- tibble( x = runif(5), y = rnorm(5) )
> # Extract by name
> df$x

[1] 0.3830567 0.3180658 0.7901315 0.8191459 0.2007465

> df[["x"]]

[1] 0.3830567 0.3180658 0.7901315 0.8191459 0.2007465

> # Extract by position
> df[[1]]

[1] 0.3830567 0.3180658 0.7901315 0.8191459 0.2007465
```

## 4.2 Import Data with readr

### readr's functions

- ▷ *read_csv()* reads comma-delimited files, *read_tsv()* reads tab-delimited files, and *read_delim()* reads in files with any delimiter.
- ▷ *read_fwf()* reads fixed-width files. You can specify fields either by their widths with *fwf_widths()* or their position with *fwf_positions()*.
- ▷ *read_table()* reads a common variation of fixed-width files where columns are separated by white space.

### Compared to base R

why we prefer to use *read_csv()* rather than *read.csv()*?

- ▷ They are typically much faster (~10x) than their base equivalents.
- ▷ They produce tibbles, and they don't convert character vectors to factors, use row names, or munge the column names.
- ▷ They are more reproducible. Base R functions inherit some behavior from your operating system and environment variables.

### Writing to a File

*readr* also comes with two useful functions for writing data back to disk:

- ▷ write_csv()
- ▷ write_tsv()

Export a CSV file to Excel: *write_excel_csv()*

## 4.3 Tidy Data with tidyr

### What is tidy data?

### What is a dataset?

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

### An example

A dataset includes the same values of four variables *country*, *year*, *population*, and *cases*, but organized in a different way:

- ▷ table1
- ▷ table2
- ▷ table3
- ▷ table4a (cases )
- ▷ table4b (population)

### table1

```
> table1

# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

### table2

```
> table2

# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

### table3

```
> table3

# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

### table4a (cases)

```
> table4a  # cases

# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

### table4b (population)

```
> table4b # population

# A tibble: 3 x 3
  country         `1999`     `2000`
* <chr>            <int>      <int>
```

```
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

In this example, only “table1” is tidy.

### What is tidy data?

*There are three interrelated rules which make a dataset tidy:*

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

### Why ensure that your data is tidy?

There are two main advantages:

1. There’s a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it’s easier to learn the tools that work with it because they have an underlying uniformity.
2. There’s a specific advantage to placing variables in columns because it allows R’s vectorised nature to shine.

*dplyr*, *ggplot2*, and all the other packages in the *tidyverse* are designed to work with tidy data.

### Compute rate per 10,000

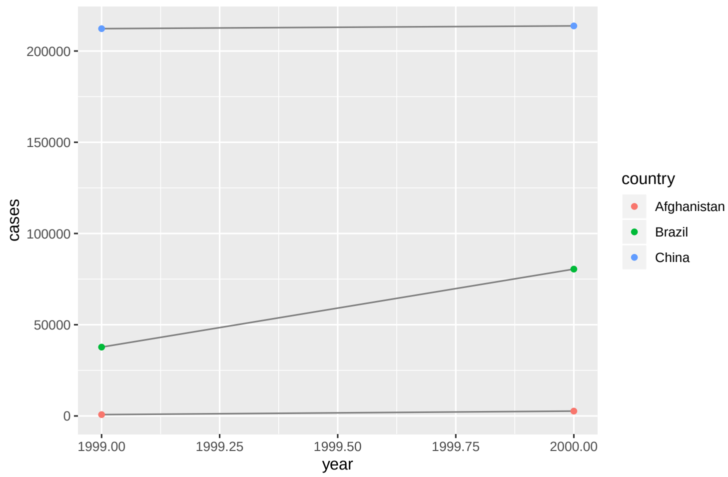
```
> table1 %>%
+        mutate(rate = cases / population * 10000)

# A tibble: 6 x 5
  country      year  cases population  rate
  <chr>       <int>  <int>      <int> <dbl>
1 Afghanistan  1999    745   19987071 0.373
2 Afghanistan  2000   2666   20595360 1.29
3 Brazil       1999  37737  172006362 2.19
4 Brazil       2000  80488  174504898 4.61
5 China        1999 212258 1272915272 1.67
6 China        2000 213766 1280428583 1.67
```

### Compute cases per year

```
> table1 %>%
+ count(year, wt = cases)

# A tibble: 2 x 2
   year      n
  <int>  <int>
1  1999 250740
2  2000 296920
```

### Visualise changes over time

```
> ggplot(table1, aes(year, cases)) +
+   geom_line(aes(group = country), colour = "grey50") +
+   geom_point(aes(colour = country))
```




### Tidy data with tidyr

### How to tidy a dataset?

- ▷ The first step is always to figure out what the variables and observations are.
- ▷ The second step is to resolve one of two common problems:
  - One variable might be spread across multiple columns.
  - One observation might be scattered across multiple rows.
- ▷ Most important functions in **tidyr**:
  - *gather()* and *spread()*
  - separate() and *unite()*

### Untidy: column names are not names of variables

A common problem is a dataset where some of the column names are not names of variables, but values of a variable.

```
> table4a

# A tibble: 3 x 3
  country     `1999` `2000`
* <chr>        <int>  <int>
1 Afghanistan    745   2666
2 Brazil       37737  80488
3 China       212258 213766
```

To tidy a dataset like this, we need to *gather* those columns into a new pair of variables.

### gather() to "year" and "cases"

```
> table4a %>%
+ gather(`1999`, `2000`, key = "year", value = "cases")

# A tibble: 6 x 3
  country     year   cases
  <chr>       <chr>  <int>
1 Afghanistan 1999     745
2 Brazil      1999   37737
3 China       1999  212258
4 Afghanistan 2000    2666
5 Brazil      2000   80488
6 China       2000  213766
```

- ▷ The name of the variable whose values form the column names. It is called the **key**, here it is *year*.
- ▷ The name of the variable whose values are spread over the cells. It is called the **value**, here is the number of *cases*.

### table4b: gather() to "year" and "population"

```
> table4b %>%
+ gather(`1999`, `2000`, key = "year", value = "population")

# A tibble: 6 x 3
  country     year  population
  <chr>       <chr>      <int>
1 Afghanistan 1999    19987071
2 Brazil      1999   172006362
3 China       1999  1272915272
4 Afghanistan 2000    20595360
5 Brazil      2000   174504898
6 China       2000  1280428583
```

### Combine two tidy tible into one: *left_join*()

```
> tidy4a <- table4a %>%
+ gather(`1999`, `2000`, key = "year", value = "cases")
> tidy4b <- table4b %>%
+ gather(`1999`, `2000`, key = "year", value = "population")
> left_join(tidy4a, tidy4b)

# A tibble: 6 x 4
  country     year   cases population
  <chr>       <chr>  <int>      <int>
1 Afghanistan 1999     745   19987071
2 Brazil      1999   37737  172006362
3 China       1999  212258 1272915272
4 Afghanistan 2000    2666   20595360
5 Brazil      2000   80488  174504898
6 China       2000  213766 1280428583
```

### Spreading

Spreading is the opposite of gathering, used when an observation is scattered across multiple rows.

table2: an *observation* is a country in a year, but each observation is *spread across two rows*.

```
> table2

# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999 cases           37737
 6 Brazil       1999 population  172006362
 7 Brazil       2000 cases           80488
 8 Brazil       2000 population  174504898
 9 China        1999 cases          212258
10 China        1999 population 1272915272
11 China        2000 cases          213766
12 China        2000 population 1280428583
```

### Tidy up using spread()

```
> spread(table2, key = type, value = count)

# A tibble: 6 x 4
  country      year   cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

### table3: the rate column contains two variables

```
> table3

# A tibble: 6 x 3
  country      year rate
* <chr>       <int> <chr>
1 Afghanistan  1999 745/19987071
2 Afghanistan  2000 2666/20595360
3 Brazil       1999 37737/172006362
4 Brazil       2000 80488/174504898
5 China        1999 212258/1272915272
6 China        2000 213766/1280428583
```

### separate()

separate() pulls apart one column into multiple columns, by splitting wherever a separator character appears.

```
> table3 %>%
+ separate(rate, into = c("cases", "population"))

# A tibble: 6 x 4
  country      year cases  population
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

### separate(), by sep argument

.

```
> table3 %>%
+ separate(rate, into = c("cases", "population"), sep = "/")

# A tibble: 6 x 4
  country      year cases  population
  <chr>       <int> <chr>  <chr>
1 Afghanistan  1999 745    19987071
2 Afghanistan  2000 2666   20595360
3 Brazil       1999 37737  172006362
4 Brazil       2000 80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

It should be noticed that cases and population are *character*.

### separate(): convert the variable type to numeric

```
> table3 %>%
+ separate( rate, into = c("cases", "population"),
+           convert = TRUE )

# A tibble: 6 x 4
  country      year  cases population
  <chr>       <int>  <int>      <int>
1 Afghanistan  1999    745   19987071
2 Afghanistan  2000   2666   20595360
3 Brazil       1999  37737  172006362
4 Brazil       2000  80488  174504898
5 China        1999 212258 1272915272
6 China        2000 213766 1280428583
```

### How to unite the *century* and *year* columns?

```
> table5

# A tibble: 6 x 4
  country     century year  rate
* <chr>       <chr>   <chr> <chr>
1 Afghanistan 19      99    745/19987071
2 Afghanistan 20      00    2666/20595360
3 Brazil      19      99    37737/172006362
4 Brazil      20      00    80488/174504898
5 China       19      99    212258/1272915272
6 China       20      00    213766/1280428583
```

### unite()...but with underscore (_)

*unite()* is the inverse of separate(): it combines multiple columns into a single column.

```
> table5 %>%
+ unite(new, century, year)

# A tibble: 6 x 3
  country     new   rate
  <chr>       <chr> <chr>
1 Afghanistan 19_99 745/19987071
2 Afghanistan 20_00 2666/20595360
3 Brazil      19_99 37737/172006362
4 Brazil      20_00 80488/174504898
5 China       19_99 212258/1272915272
6 China       20_00 213766/1280428583
```

### We don't want the underscore (_)

```
> table5 %>%
+ unite(new, century, year, sep = "")

# A tibble: 6 x 3
  country     new   rate
```

```
  <chr>       <chr> <chr>
1 Afghanistan 1999  745/19987071
2 Afghanistan 2000  2666/20595360
3 Brazil      1999  37737/172006362
4 Brazil      2000  80488/174504898
5 China       1999  212258/1272915272
6 China       2000  213766/1280428583
```

### Tidy missing values

### Missing values

A value can be missing in one of two possible ways:

- ▷ Explicitly, i.e., flagged with NA.
- ▷ Implicitly, i.e., simply not present in the data.

```
> stocks <- tibble(
+   year = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
+   qtr  = c(   1,    2,    3,    4,    2,    3,    4),
+   return=c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
+ )
```

### Tidy missing data with *complete()*

An important tool for making missing values explicit in tidy data is *complete().*

```
> stocks %>%
+ complete(year, qtr)

# A tibble: 8 x 3
   year   qtr return
  <dbl> <dbl>  <dbl>
1  2015     1   1.88
2  2015     2   0.59
3  2015     3   0.35
4  2015     4  NA
5  2016     1  NA
6  2016     2   0.92
7  2016     3   0.17
8  2016     4   2.66
```

### Make missing values explicit: change a represent way

The way that a dataset is represented can make implicit values explicit.

```
> stocks %>%
+ spread(year, return)

# A tibble: 4 x 3
    qtr `2015` `2016`
  <dbl>  <dbl>  <dbl>
```

```
1     1   1.88  NA
2     2   0.59   0.92
3     3   0.35   0.17
4     4  NA      2.66
```

### Fill in missing values with fill()

```
> treatment <- tribble(
+ ~ person, ~ treatment, ~response,
+ "Derrick Whitmore", 1, 7,
+                     NA, 2, 10,
+                 NA, 3, 9,
+ "Katherine Burke", 1, 4
+ )
```

### Fill in missing values with fill()

You can fill in these missing values with fill(). It takes a set of columns where you want missing values to be replaced by the most recent nonmissing value.

```
> treatment %>%
+ fill(person)

# A tibble: 4 x 3
  person           treatment response
  <chr>                <dbl>    <dbl>
1 Derrick Whitmore         1        7
2 Derrick Whitmore         2       10
3 Derrick Whitmore         3        9
4 Katherine Burke          1        4
```

# 5 Descriptive Statistics

### Descriptive Statistics

Numerical summaries of the *population* are called **parameters**, while numerical summaries of the *sample* are called **statistics**.

▷ Summary Measures of Location

- mean, median, mode, quantiles,

▷ Summary Measures of Spread

- range, interquartile-range(IQR), variance, standard-deviation(sd),The Median Absolute Deviation (MAD)

▷ Summary Measures of Shape

- skewness, kurtosis

## 5.1 Summary Measures of Location

### R functions for location

- ▷ Population mean: $\mu$
- ▷ Sample mean: $\bar{x}$
- ▷ R functions: mean(x), median(x), mode(x)
- ▷ Quantiles: the $x_p$ is called a **$p$-quantile** of a distribution, if $\mathrm{P}(X \leq x_p) \geq p$ and $\mathrm{P}(X \geq x_p) \leq 1 - p$
  - for continuous r.v., $\mathrm{P}(X \leq x_p) = p$
- ▷ quantile(x, probs=c(0.25, 0.5, 0.75)): $Q_1, Q_2, Q_3$

### mpg data

```
> mean(mpg$hwy)

[1] 23.44017

> median(mpg$hwy)

[1] 24

> quantile(mpg$hwy,probs=c(0.25,0.5,0.75))

25% 50% 75%
 18  24  27
```

### flights data

```
> not_cancelled %>%
+  group_by(year, month, day) %>%
+  summarize(
+  # average delay:
+  avg_delay1 = mean(arr_delay),
+  # average positive delay:
+  avg_delay2 = mean(arr_delay[arr_delay > 0])
+ )

# A tibble: 365 x 5
# Groups:   year, month [12]
    year month   day avg_delay1 avg_delay2
   <int> <int> <int>      <dbl>      <dbl>
 1  2013     1     1     12.7         32.5
 2  2013     1     2     12.7         32.0
 3  2013     1     3      5.73        27.7
 4  2013     1     4     -1.93        28.3
 5  2013     1     5     -1.53        22.6
 6  2013     1     6      4.24        24.4
 7  2013     1     7     -4.95        27.8
```

```
 8  2013     1     8     -3.23        20.8
 9  2013     1     9     -0.264       25.6
10  2013     1    10     -5.90        27.3
# ... with 355 more rows
```

## 5.2 Summary Measures of Spread

### functions

- range(x): returns the smallest and largest values in x
- IQR(x): Interquartile Range, IQR= Q3 - Q1
- var(x): $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$
- sd(x): $s = \sqrt{s^2}$
- Sample Coeffcient of Variation: $CV = S/\bar{X}$
- Relative Standard Deviation: $RSD = |S/\bar{X}| \times 100$
- The Median Absolute Deviation (MAD): is a robust measure of spread, often used when the median is reported to describe the center of a *skewed data set*.

  $$MAD = \text{median}\{|x_i - m|\}$$

  where $m$ is the median of $x$.

### mpg data

```
> IQR(mpg$hwy)

[1] 9

> var(mpg$hwy)

[1] 35.45778

> sd(mpg$hwy)

[1] 5.954643

> mad(mpg$hwy)

[1] 7.413
```

### flights data

```
> not_cancelled %>%
+         group_by(dest) %>%
+         summarize(
+                 distance_sd = sd(distance),
+                 distance_var = var(distance)
+         ) %>%
+         arrange(desc(distance_sd))

# A tibble: 104 x 3
   dest  distance_sd distance_var
   <chr>       <dbl>        <dbl>
 1 EGE         10.5         111.
 2 SAN         10.4         107.
 3 SFO         10.2         104.
 4 HNL         10.0         100.
 5 SEA          9.98         99.6
 6 LAS          9.91         98.2
 7 PDX          9.87         97.5
 8 PHX          9.86         97.3
 9 LAX          9.66         93.3
10 IND          9.46         89.5
# ... with 94 more rows
```

## 5.3 Summary Measures of Shape

**Skewness**

The base R doesn't provide functions for skew and kurtosis.

**Skewness** is a measure of the asymmetry of the probability distribution of a real-valued random variable about its mean.

▷ Pearson's moment coefficient of skewness

$$\text{Skew}_p = E\big[\big(\frac{X-\mu}{\sigma}\big)\big]^3$$

▷ Sample skewness

$$\text{Skew}_S = \frac{1}{n}\sum_{i=1}^{n}\big(\frac{x_i-\bar{x}}{s}\big)^3$$

Or

$$\text{Skew}_S = \frac{n^2}{(n-1)(n-2)}\frac{1}{n}\sum_{i=1}^{n}\big(\frac{x_i-\bar{x}}{s}\big)^3$$

where $s^2$ is the observed sample variance.

### Kurtosis

- ▷ **Kurtosis** is a measure of the "tailedness" of the probability distribution of a real-valued random variable.

$$\text{Kurt}_p = E\big[\big(\frac{X-\mu}{\sigma}\big)\big]^4$$

- ▷ Sample Kurtosis

$$\text{Kurt}_S = \frac{1}{n}\sum_{i=1}^{n}\big(\frac{x_i-\bar{x}}{s}\big)^4$$

- ▷ The ***excess kurtosis*** = *Kurt - 3* (Kurt=3 for Normal)

### mpg data

```
> library(moments)
> skewness(mpg$hwy,na.rm = TRUE)

[1] 0.366865

> kurtosis(mpg$hwy,na.rm = TRUE)

[1] 3.163929
```

### Descriptive Statistics via *summarize()*

```
> library(moments)
> mpg %>%
+   summarize(
+   mean = mean(hwy),
+   sd = sd(hwy),
+   skew = skewness(hwy, na.rm = FALSE),
+   kurt = kurtosis(hwy, na.rm = FALSE)
+ )

# A tibble: 1 x 4
   mean    sd  skew  kurt
  <dbl> <dbl> <dbl> <dbl>
1  23.4  5.95 0.367  3.16
```

### Descriptive Statistics via Own-written Function

```
> dec_stats<-function(x,na.omit=FALSE){
+   if(na.omit)
+      x<-x[!is.na(x)]
+   m<-mean(x)
+   n<-length(x)
+   s<-sd(x)
```

```
+   skew<-sum((x-m)^3/s^3)/n
+   kurt<-sum((x-m)^4/s^4)/n - 3
+ return(c(n=n,mean=m,stdev=s,skew=skew,kurtosis=kurt))
+ }
> round(dec_stats(mpg$hwy),3)

       n     mean    stdev     skew kurtosis
 234.000   23.440    5.955    0.365    0.137
```

**Recap**

1. Data Structure
   - ▷ vector, matrix, array, data frame, factor, list
2. Data visualisation with ggplot2
3. Data Tranformation with dplyr
   - ▷ filter, arrange, select, mutate, summarize, group_by.
4. Data Wrangle
   (a) Tibbles with tibble
   (b) Import Data with readr
   (c) Tidy data with tidyr
      i. What is tidy data?
      ii. Tidy data with tidyr
      iii. Tidy missing values
5. Descriptive Statistics
   - ▷ Location, Spread,Shape

## References

Casella, George and Roger Berger (2002). *Statistical Inference*. 2nd. Duxbury: Wadsworth Group.

Cohen, Yosef and Jeremiah Y. Cohen (2008). *Statistics and data with R: an applied approach through examples*. Chichester, U.K: Wiley.

Kabacoff, Robert (2015). *R in action: data analysis and graphics with R*. 2nd. Shelter Island, NY: Manning.

Ugarte, María Dolores, Ana F. Militino, and Alan T. Arnholt (2016). *Probability and Statistics with R (Text book)*. 2nd. Boca Raton, FL: CRC Press.

Wickham, Hadley and Garrett Grolemund (2016). *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. O'Reilly Media, Inc.