Class 3 - Intro to the `tidyverse`

QMSS GR5072 Modern Data Structures

Marco Morales

February 2, 2022

Disclaimers

- Materials compiled by Thomas Brambor & Mike Parrot
- Sources - R Tidyverse Vignettes & H. Wickham

Working with Tibbles

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).

Creating Tibbles

tibble() is a nice way to create data frames. It encapsulates best practices for data frames:

```
tibble(x = letters)
```

```
## # A tibble: 26 × 1
##    x
##    <chr>
##  1 a
##  2 b
##  3 c
##  4 d
##  5 e
##  6 f
##  7 g
##  8 h
##  9 i
## 10 j
## # … with 16 more rows
```

Tibbles Vs. Data Frames

There are three key differences between tibbles and data frames:

printing, subsetting, and recycling rules.

Printing

When you print a tibble, it only shows the first ten rows and all the columns that fit on one screen.

It also prints an abbreviated description of the column type, and uses font styles and color for highlighting:

Printing

```
tibble(x = -5:1000)
```

```
## # A tibble: 1,006 × 1
##        x
##    <int>
##  1    -5
##  2    -4
##  3    -3
##  4    -2
##  5    -1
##  6     0
##  7     1
##  8     2
##  9     3
## 10     4
## # … with 996 more rows
```

Printing

You can control the default appearance with options:

options(tibble.print_max = n, tibble.print_min = m): if there are more than n rows, print only the first m rows.

Use options(tibble.print_max = Inf) to always show all rows.

options(tibble.width = Inf) will always print all columns, regardless of the width of the screen.

Subsetting

Tibbles are quite strict about subsetting.

- always returns another tibble. Contrast this with a data frame: sometimes
- returns a data frame and sometimes it just returns a vector:

```
df1 <- data.frame(x = 1:3, y = 3:1)
class(df1[, 1:2])
```

```
## [1] "data.frame"
```

```
class(df1[, 1])
```

```
## [1] "integer"
```

Subsetting

```
df2 <- tibble(x = 1:3, y = 3:1)
class(df2[, 1:2])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
class(df2[, 1])
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

To extract a single column use [[or $:

```
class(df2[[1]])
```

```
## [1] "integer"
```

```
class(df2$x)
```

```
## [1] "integer"
```

Intro to dplyr package

When working with data you must:

- Figure out what you want to do.
- Describe those tasks in the form of a computer program.
- Execute the program.

Intro to dplyr package

The dplyr package makes these steps fast and easy:

By constraining your options, it helps you think about your data manipulation challenges.

It provides simple "verbs", functions that correspond to the most common data manipulation tasks, to help you translate your thoughts into code.

It also uses efficient backends, so you spend less time waiting for the computer.

Intro to dplyr package

Example data: nycflights13

To work with data you need to install the nycflights13 package

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

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

```
flights
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
```

Single table verbs

Dplyr aims to provide a function for each basic verb of data manipulation:

- filter() to select cases based on their values.
- arrange() to reorder the cases.
- select() and rename() to select variables based on their names.
- mutate() and transmute() to add new variables that are functions of existing variables.
- summarise() to condense multiple values to a single value.
- sample_n() and sample_frac() to take random samples.

Filter rows with filter()

filter() allows you to select a subset of rows in a data frame.

Like all single verbs, the first argument is the tibble (or data frame).

The second and subsequent arguments refer to variables within that data frame, selecting rows where the expression is TRUE.

Filter rows with filter()

For example, we can select all flights on January 1st with:

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

```
## # A tibble: 842 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # … with 832 more rows, and 11 more variables: 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>
```

Filter rows with filter()

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

This is roughly equivalent to this base R code:

```
flights[flights$month == 1 & flights$day == 1, ]
```

```
## # A tibble: 842 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # … with 832 more rows, and 11 more variables: 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>
```

Arrange rows with arrange()

arrange() works similarly to filter() except that instead of filtering or selecting rows, it reorders them.

It takes a data frame, and a set of column names (or more complicated expressions) to order by.

If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

arrange(flights, year, month, day)

Arrange rows with arrange()

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

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # … with 336,766 more rows, and 11 more variables: 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>
```

Arrange rows with arrange()

Use desc() to order a column in descending order:

arrange(flights, desc(arr_delay))

Arrange rows with arrange()

```
arrange(flights, desc(arr_delay))
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     9      641            900      1301     1242           1530
##  2  2013     6    15     1432           1935      1137     1607           2120
##  3  2013     1    10     1121           1635      1126     1239           1810
##  4  2013     9    20     1139           1845      1014     1457           2210
##  5  2013     7    22      845           1600      1005     1044           1815
##  6  2013     4    10     1100           1900       960     1342           2211
##  7  2013     3    17     2321            810       911      135           1020
##  8  2013     7    22     2257            759       898      121           1026
##  9  2013    12     5      756           1700       896     1058           2020
## 10  2013     5     3     1133           2055       878     1250           2215
## # … with 336,766 more rows, and 11 more variables: 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>
```

Select columns with select()

Often you work with large datasets with many columns but only a few are actually of interest to you.

select() allows you to rapidly zoom in on a useful subset of columns

Select columns by name

```
select(flights, year, month, day)
```

```
## # A tibble: 336,776 × 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 (inclusive)

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

```
## # A tibble: 336,776 × 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 (inclusive)

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

```
## # A tibble: 336,776 × 16
##    dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay carrier
##       <int>          <int>     <dbl>    <int>          <int>     <dbl> <chr>
##  1      517            515         2      830            819        11 UA
##  2      533            529         4      850            830        20 UA
##  3      542            540         2      923            850        33 AA
##  4      544            545        -1     1004           1022       -18 B6
##  5      554            600        -6      812            837       -25 DL
##  6      554            558        -4      740            728        12 UA
##  7      555            600        -5      913            854        19 B6
##  8      557            600        -3      709            723       -14 EV
##  9      557            600        -3      838            846        -8 B6
## 10      558            600        -2      753            745         8 AA
## # … with 336,766 more rows, and 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>
```

Helper functions

There are a number of helper functions you can use within select(),

- like starts_with(),
- ends_with(),
- matches()
- and contains().

These let you quickly match larger blocks of variables that meet some criterion. See ?select for more details.

Use rename() to change variable names

Example: For original column name “tailnum”, change it to “tail_num”…

```
rename(flights, tail_num = tailnum)
```

```
## # A tibble: 336,776 × 19
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
## 10  2013     1     1      558            600        -2      753            745
## # … with 336,766 more rows, and 11 more variables: arr_delay <dbl>,
## #   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>,
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl>, time_hour <dttm>
```

```
colnames(flights)
```

Add new columns with mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns.

This is the job of mutate():

```
mutate(flights,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60
)
```

```
## # A tibble: 336,776 × 21
##     year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
##    <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
##  1  2013     1     1      517            515         2      830            819
##  2  2013     1     1      533            529         4      850            830
##  3  2013     1     1      542            540         2      923            850
##  4  2013     1     1      544            545        -1     1004           1022
##  5  2013     1     1      554            600        -6      812            837
##  6  2013     1     1      554            558        -4      740            728
##  7  2013     1     1      555            600        -5      913            854
##  8  2013     1     1      557            600        -3      709            723
##  9  2013     1     1      557            600        -3      838            846
```

More with mutate()

case_when is particularly useful inside mutate when you want to create a new variable that relies on a complex combination of existing variables:

More with mutate()

```
starwars %>%
  select(name:mass, gender, species) %>%
  mutate(
    type = case_when(
      height > 200 | mass > 200 ~ "large",
      species == "Droid"        ~ "robot",
      TRUE                      ~ "other"
      # this "TRUE" code defines what to define everything
    )
  )
```

```
## # A tibble: 87 × 6
##    name               height  mass gender    species type
##    <chr>               <int> <dbl> <chr>     <chr>   <chr>
##  1 Luke Skywalker        172    77 masculine Human   other
##  2 C-3PO                 167    75 masculine Droid   robot
##  3 R2-D2                  96    32 masculine Droid   robot
##  4 Darth Vader           202   136 masculine Human   large
##  5 Leia Organa           150    49 feminine  Human   other
##  6 Owen Lars             178   120 masculine Human   other
##  7 Beru Whitesun lars    165    75 feminine  Human   other
##  8 R5-D4                  97    32 masculine Droid   robot
##  9 Biggs Darklighter     183    84 masculine Human   other
```

Summarise values with summarise()

The last verb is summarise(). It collapses a data frame to a single row.

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

```
## # A tibble: 1 × 1
##   delay
##   <dbl>
## 1  12.6
```

It's not that useful until we learn the group_by() verb below.

Calculate stats by variable category

Grouped operations with group_by():

The dplyr verbs are useful on their own, but they become even more powerful when you apply them to groups of observations within a dataset.

In dplyr, you do this with the group_by() function.

group_by() examples:

Group flights data by values in tailnum column:

by_tailnum <- group_by(flights, tailnum)

Group flights data by values in dest column:

destinations <- group_by(flights, dest)

group_by() examples:

We often use group_by() with the summarise function.

You use summarise() with aggregate functions, which take a vector of values and return a single number.

```
destinations <- group_by(flights, dest)
summarise(destinations,
  planes = n_distinct(tailnum),
  flights = n()
)
```

```
## # A tibble: 105 × 3
##    dest  planes flights
##    <chr>  <int>   <int>
##  1 ABQ      108     254
##  2 ACK       58     265
##  3 ALB      172     439
##  4 ANC        6       8
##  5 ATL     1180   17215
##  6 AUS      993    2439
##  7 AVL      159     275
##  8 BDL      186     443
```

summarise aggregate functions:

You use summarise() with aggregate functions, which take a vector of values and return a single number.

There are many useful examples of such functions in base R like ***min(), max(), mean(), sum(), sd(), median(), and IQR().***

dplyr provides a handful of others:

n(): the number of observations in the current group

n_distinct(x):the number of unique values in x.

Others include - first(x), last(x) and nth(x, n) - these work similarly to x[1], x[length(x)], and x[n] but give you more control over the result if the value is missing.

Use ungroup() to stop grouped operations

```
ungroup(flights)
```

Pipe Operators and the Tidyverse

You can use Tidyverse functions with the pipe operator

The %>% operator is used to make code easier to read

Different ways to organize code

Option 1: Use multiple objects/variables

```
a <- filter(mtcars, carb > 1)
b <- group_by(a, cyl)
c <- summarise(b, Avg_mpg = mean(mpg))
d <- arrange(c, desc(Avg_mpg))
print(d)
```

```
## # A tibble: 3 × 2
##     cyl Avg_mpg
##   <dbl>   <dbl>
## 1     4    25.9
## 2     6    19.7
## 3     8    15.1
```

Different ways to organize code

Option 2: Nested Option

```
arrange(
   summarize(
       group_by(
           filter(mtcars, carb > 1),
           cyl
          ),
       Avg_mpg = mean(mpg)
      ),
   desc(Avg_mpg)
 )
```

```
## # A tibble: 3 × 2
##     cyl Avg_mpg
##   <dbl>   <dbl>
## 1     4    25.9
## 2     6    19.7
## 3     8    15.1
```

Different ways to organize code

Option 3: Use Pipe Operators (Not always available, but you can do it with tidyverse functions)

```
mtcars %>%
        filter(carb > 1) %>%
        group_by(cyl) %>%
        summarise(Avg_mpg = mean(mpg)) %>%
        arrange(desc(Avg_mpg))
```

```
## # A tibble: 3 × 2
##     cyl Avg_mpg
##   <dbl>   <dbl>
## 1     4    25.9
## 2     6    19.7
## 3     8    15.1
```

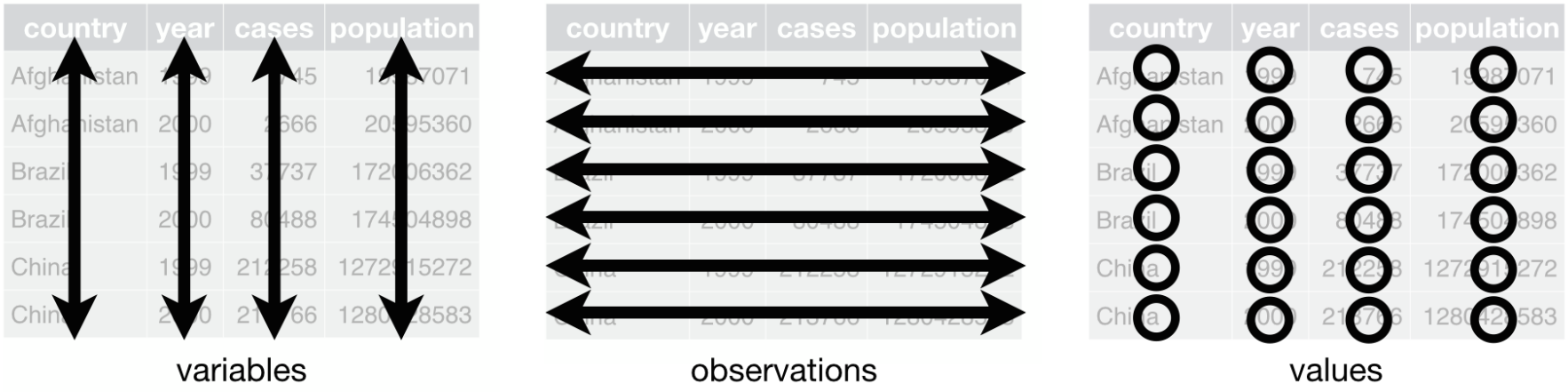
Tidy Data

There are three interrelated rules which make a dataset tidy:

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

We often want data to be in this format for data analysis

Tidy Data


country
year
cases
population
variables
observations
values

Separating and uniting

separate() pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take table3:

```
table3
```

```
## # A tibble: 6 × 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
```

Separating

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

```
## # A tibble: 6 × 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
```

Separating

```
#Separate year values after second element of value

table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
```

```
## # A tibble: 6 × 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
```

Uniting

Unite combines two variables into one

```
table5[1:2,]
```

```
## # A tibble: 2 × 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
```

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

```
## # A tibble: 6 × 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
```

Joining (merging) data in the Tidyverse

- **Mutating joins**, which add new variables to one data frame from matching observations in another.
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other table.
- **Set operations**, which treat observations as if they were set elements.

We will focus on the most common joins, *Mutating Joins*.

Keys

The variables used to connect each pair of tables are called keys.

- A *key is a variable (or set of variables) that uniquely identifies an observation*.
- In simple cases, a single variable is sufficient to identify an observation.

For example, each plane is uniquely identified by its tailnum.

- In other cases, multiple variables may be needed.
- For example, to identify an observation in weather you need five variables: year, month, day, hour, and origin.

There are two types of keys:

- A primary key uniquely identifies an observation in its own table.

For example, planes$tailnum is a primary key because it uniquely identifies each plane in the planes table.

- A foreign key uniquely identifies an observation in another table.

A variable can be both a primary key and a foreign key.

Best Practices: Ensure data has unique values on key

- It's good practice to verify that they do indeed uniquely identify each observation.
- One way to do that is to count() the primary keys and look for entries where n is greater than one (we want none greater than one):

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
```

```
## # A tibble: 0 × 2
## # … with 2 variables: tailnum <chr>, n <int>
```

Mutating Joins

Lets say we want to add a column from the airlines dataset to the following data:

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
```

```
## # A tibble: 336,776 × 8
##     year month   day  hour origin dest  tailnum carrier
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>
##  1  2013     1     1     5 EWR    IAH   N14228  UA
##  2  2013     1     1     5 LGA    IAH   N24211  UA
##  3  2013     1     1     5 JFK    MIA   N619AA  AA
##  4  2013     1     1     5 JFK    BQN   N804JB  B6
##  5  2013     1     1     6 LGA    ATL   N668DN  DL
##  6  2013     1     1     5 EWR    ORD   N39463  UA
##  7  2013     1     1     6 EWR    FLL   N516JB  B6
##  8  2013     1     1     6 LGA    IAD   N829AS  EV
##  9  2013     1     1     6 JFK    MCO   N593JB  B6
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA
## # … with 336,766 more rows
```

Mutating Joins

Lets say we want to add a column from the airlines dataset to the following data:

```
# the airlines dataset looks like:
head(airlines)
```

```
## # A tibble: 6 × 2
##   carrier name
##   <chr>   <chr>
## 1 9E      Endeavor Air Inc.
## 2 AA      American Airlines Inc.
## 3 AS      Alaska Airlines Inc.
## 4 B6      JetBlue Airways
## 5 DL      Delta Air Lines Inc.
## 6 EV      ExpressJet Airlines Inc.
```

Mutating Joins

To do so, we would use left_join():

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 × 7
##     year month   day  hour tailnum carrier name
##    <int> <int> <int> <dbl> <chr>   <chr>   <chr>
##  1  2013     1     1     5 N14228  UA      United Air Lines Inc.
##  2  2013     1     1     5 N24211  UA      United Air Lines Inc.
##  3  2013     1     1     5 N619AA  AA      American Airlines Inc.
##  4  2013     1     1     5 N804JB  B6      JetBlue Airways
##  5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
##  6  2013     1     1     5 N39463  UA      United Air Lines Inc.
##  7  2013     1     1     6 N516JB  B6      JetBlue Airways
##  8  2013     1     1     6 N829AS  EV      ExpressJet Airlines Inc.
##  9  2013     1     1     6 N593JB  B6      JetBlue Airways
## 10  2013     1     1     6 N3ALAA  AA      American Airlines Inc.
## # … with 336,766 more rows
```

Mutating Joins

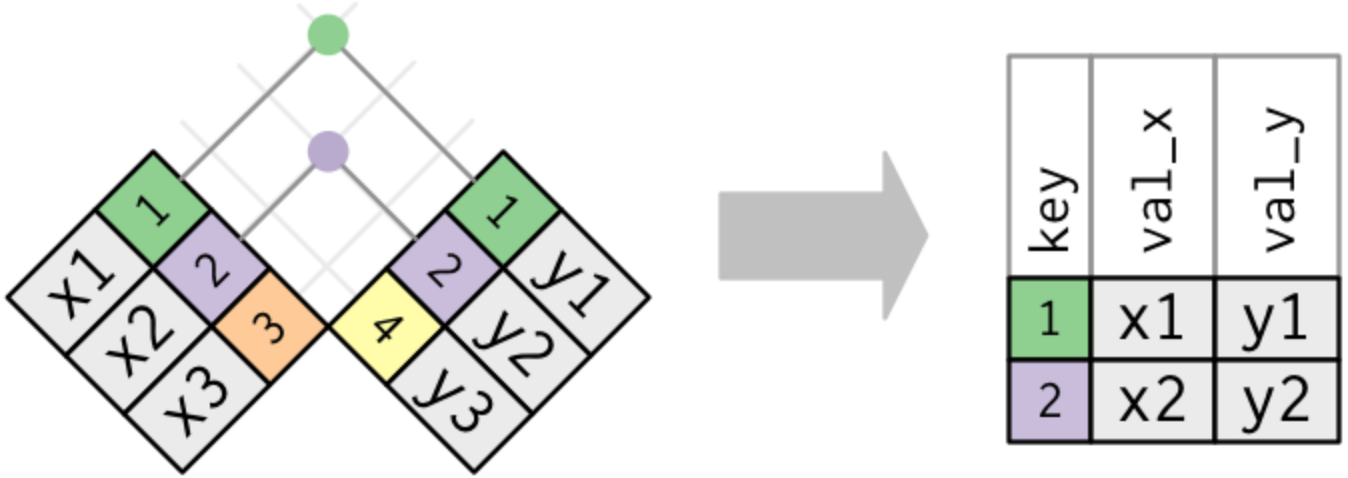
The result of joining airlines to flights2 is an additional variable: name

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
```

```
## # A tibble: 336,776 × 7
##     year month   day  hour tailnum carrier name
##    <int> <int> <int> <dbl> <chr>   <chr>   <chr>
##  1  2013     1     1     5 N14228  UA      United Air Lines Inc.
##  2  2013     1     1     5 N24211  UA      United Air Lines Inc.
##  3  2013     1     1     5 N619AA  AA      American Airlines Inc.
##  4  2013     1     1     5 N804JB  B6      JetBlue Airways
##  5  2013     1     1     6 N668DN  DL      Delta Air Lines Inc.
##  6  2013     1     1     5 N39463  UA      United Air Lines Inc.
##  7  2013     1     1     6 N516JB  B6      JetBlue Airways
##  8  2013     1     1     6 N829AS  EV      ExpressJet Airlines Inc.
##  9  2013     1     1     6 N593JB  B6      JetBlue Airways
## 10  2013     1     1     6 N3ALAA  AA      American Airlines Inc.
## # … with 336,766 more rows
```

Types of Mutating Joins

The simplest type of join is the ***inner join***. An inner join matches pairs of observations whenever their keys are equal:



Inner Join

Inner Join

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

Inner Join

```
x %>%
  inner_join(y, by = "key")
```

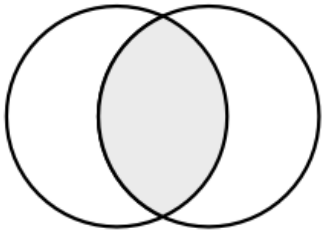
```
## # A tibble: 2 × 3
##     key val_x val_y
##   <dbl> <chr> <chr>
## 1     1 x1    y1
## 2     2 x2    y2
```

Outer joins

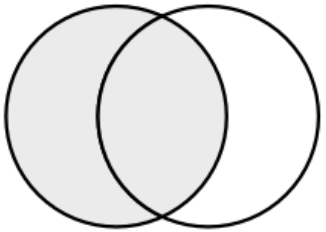
An inner join keeps observations that appear in both tables. An **outer join keeps observations that appear in at least one of the tables**. There are three types of outer joins:

- A *left join* keeps all observations in x.
- A *right join* keeps all observations in y.
- A *full join* keeps all observations in x and y.

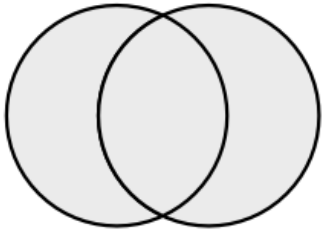
Outer joins



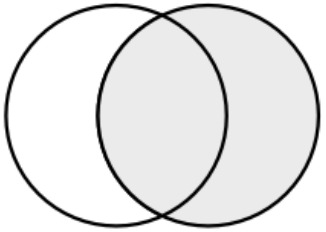
`inner_join(x, y)`



`left_join(x, y)`



`full_join(x, y)`



`right_join(x, y)`

Outer joins

```
#leftjoin() with same variable name in two datasets
flights2 %>%
  left_join(planes, by = "tailnum")
```

```
## # A tibble: 336,776 × 16
##    year.x month   day  hour origin dest  tailnum carrier year.y type
##     <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>    <int> <chr>
##  1   2013     1     1     5 EWR    IAH   N14228  UA        1999 Fixed wing mult…
##  2   2013     1     1     5 LGA    IAH   N24211  UA        1998 Fixed wing mult…
##  3   2013     1     1     5 JFK    MIA   N619AA  AA        1990 Fixed wing mult…
##  4   2013     1     1     5 JFK    BQN   N804JB  B6        2012 Fixed wing mult…
##  5   2013     1     1     6 LGA    ATL   N668DN  DL        1991 Fixed wing mult…
##  6   2013     1     1     5 EWR    ORD   N39463  UA        2012 Fixed wing mult…
##  7   2013     1     1     6 EWR    FLL   N516JB  B6        2000 Fixed wing mult…
##  8   2013     1     1     6 LGA    IAD   N829AS  EV        1998 Fixed wing mult…
##  9   2013     1     1     6 JFK    MCO   N593JB  B6        2004 Fixed wing mult…
## 10   2013     1     1     6 LGA    ORD   N3ALAA  AA          NA <NA>
## # … with 336,766 more rows, and 6 more variables: manufacturer <chr>,
## #   model <chr>, engines <int>, seats <int>, speed <int>, engine <chr>
```

Outer joins with difft key names

Use a named character vector: by = c("a" = "b").

- This will match variable a in table x to variable b in table y. The variables from x will be used in the output.

```
#leftjoin() with diff't variable name in two datasets
flights2 %>%
  left_join(airports, c("dest" = "faa"))
```

```
## # A tibble: 336,776 × 15
##     year month   day  hour origin dest  tailnum carrier name     lat   lon   alt
##    <int> <int> <int> <dbl> <chr>  <chr> <chr>   <chr>   <chr>  <dbl> <dbl> <dbl>
##  1  2013     1     1     5 EWR    IAH   N14228  UA      Georg…  30.0 -95.3    97
##  2  2013     1     1     5 LGA    IAH   N24211  UA      Georg…  30.0 -95.3    97
##  3  2013     1     1     5 JFK    MIA   N619AA  AA      Miami…  25.8 -80.3     8
##  4  2013     1     1     5 JFK    BQN   N804JB  B6      <NA>    NA    NA      NA
##  5  2013     1     1     6 LGA    ATL   N668DN  DL      Harts…  33.6 -84.4  1026
##  6  2013     1     1     5 EWR    ORD   N39463  UA      Chica…  42.0 -87.9   668
##  7  2013     1     1     6 EWR    FLL   N516JB  B6      Fort …  26.1 -80.2     9
##  8  2013     1     1     6 LGA    IAD   N829AS  EV      Washi…  38.9 -77.5   313
##  9  2013     1     1     6 JFK    MCO   N593JB  B6      Orlan…  28.4 -81.3    96
## 10  2013     1     1     6 LGA    ORD   N3ALAA  AA      Chica…  42.0 -87.9   668
```

Base R versus Tidyverse Joins

base::merge() can perform all four types of mutating join:

dplyr versus merge - inner_join(x, y) is equal to merge(x, y) - left_join(x, y) is equal to merge(x, y, all.x = TRUE) - right_join(x, y) is equal to merge(x, y, all.y = TRUE), - full_join(x, y) is equal to merge(x, y, all.x = TRUE, all.y = TRUE)

Tidy data using Spread() and Gather

Data is often organised to facilitate some use other than analysis.

- For example, data is often organised to make entry as easy as possible.
- This means for most real analyses, you'll need to do some tidying.

Tidy data using Spread() and Gather

- The first step is always to figure out what the variables and observations are.
- Sometimes this is easy; other times you'll need to consult with the people who originally generated the data.

Tidy data using Spread() and Gather

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.
- Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky!

Tidy data using Spread() and Gather

To fix these problems, you'll need the two most important functions in tidyr according to Hadley w.:

gather() and spread().

Gathering

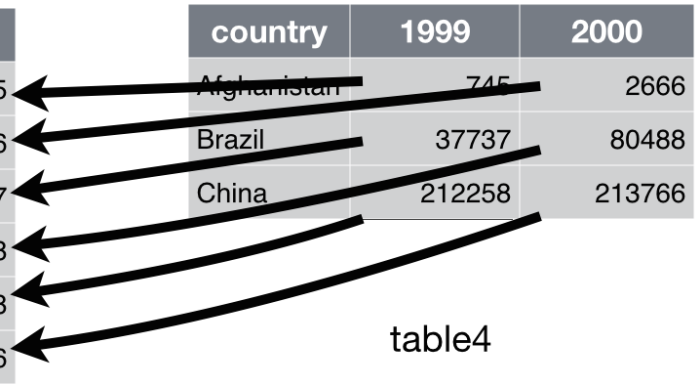
To make the following data “tidy” we need to convert our observations to represent country-years. (Unique row = unique country and year values) Example data:

```
table4a
```

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

Gathering

country	year	cases
Afghanistan	1999	745
Afghanistan	2000	2666
Brazil	1999	37737
Brazil	2000	80488
China	1999	212258
China	2000	213766



country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

table4

Gathering

```
# Note that columns are numeric, so we use backticks
# More typically we can just refer to column names.
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

```
## # A tibble: 6 × 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
```

Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows.

How would you make the following data tidy?

```
table2
```

```
## # A tibble: 12 × 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
```

Spreading

- The variable we want to create new columns with is the key.
- The variable we want to specify as our values is the value

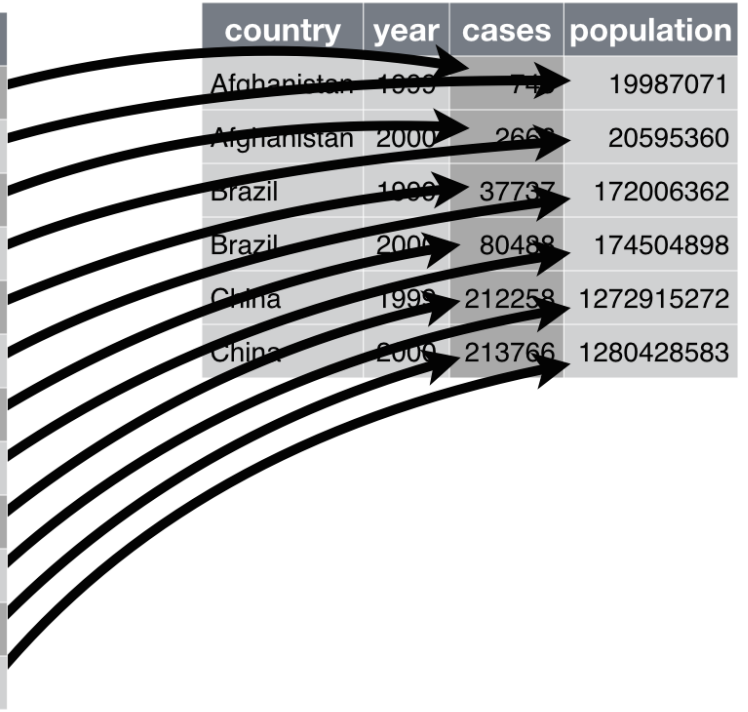
```
table2 %>%
    spread(key = type, value = count)
```

```
## # A tibble: 6 × 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
```

Spreading

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

table2



country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583