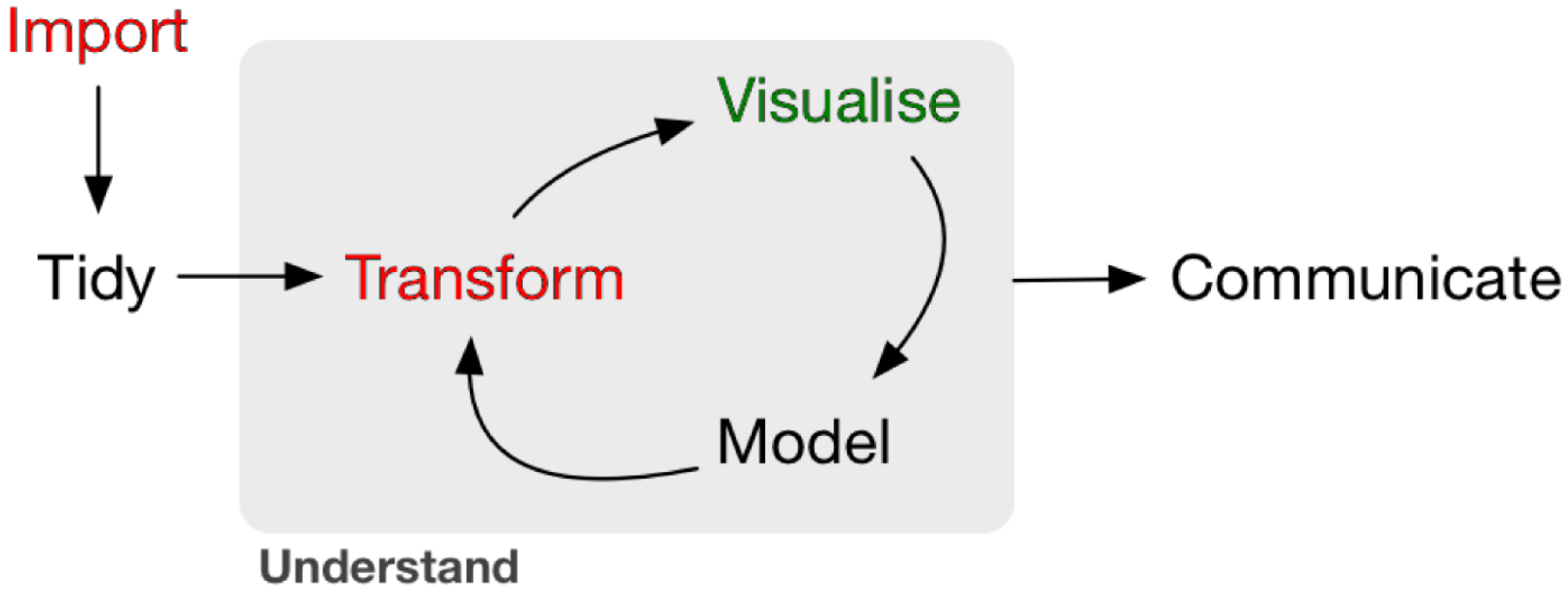
Contents

- Importing rectangular text files
- Importing other types of data
- Transforming data



Importing data with `readr`

The `readr` package

Many R packages provide examples of data.
However, sooner or later you will need to work with your own data.

`readr` is for rectangular text data.


```
library(tidyverse)
```

`readr` supports several file formats with seven `read_` functions:

- `read_csv()`: comma-separated (CSV) files
- `read_tsv()`: tab-separated files
- `read_delim()`: general delimited files
- `read_fwf()`: fixed-width files
- `read_table()`: tabular files where colums are separated by white-space
- `read_log()`: web log files

In many cases it just works: supply path to a file and get a tibble back.

Comparison with base R

Why are we learning the `readr` package?
Base R has a function `read.csv()` to read CSV files.

Some advantages of `readr`:

- it is up to 10x faster
- it produces `tibbles` instead of `data.frames`
- better parsing (e.g. does not convert strings to factors)
- more reproducible on different systems
- progress bar for large files

Reading comma-separated files

All `read_` functions have a similar syntax, so we focus on `read_csv()`.

```
readr_example("mtcars.csv")
```

```
## ~/R/x86_64-redhat-linux-gnu-library/3.4/readr/extdata/mtcars.csv
```

Fuel consumption, 10 aspects of design and performance (`?mtcars`).

```
mtcars <- read_csv(readr_example("mtcars.csv"))
```

```
## Parsed with column specification:
## cols(
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

The `read_csv` function

Also works with inline csv files (useful for experimenting).

```
read_csv("a,b,c
1,2,3
4,5,6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

First line used as column names, but this can be changed.

```
read_csv("a,b,c
1,2,3
4,5,6", col_names=FALSE)
```

```
## # A tibble: 3 x 3
##   X1    X2    X3
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 1     2     3
## 3 4     5     6
```

Other useful tweaks: skip lines (metadata); symbol for missing data (N.A.)

Now you can read most CSV files, also easily adapt to `read_tsv`, `read_fwf`. For the others, you need to know how `readr` works inside.

Parsing vectors with `readr`

Parsing a vector

Functions `parse_*()` take a char vector and return a specialised vector.

```
parse_logical(c("TRUE","FALSE"))
```

```
## [1]  TRUE FALSE
```

```
parse_integer(c("1","2","3","NA"))
```

```
## [1]  1  2  3 NA
```

These are an important building block for `readr`.

Useful functions:

- `parse_logical()`, `parse_integer()`
- `parse_double()`, `parse_number()`, for numbers from other countries
- `parse_character()` for character encodings.
- `parse_factor()`
- `parse_datetime()`, `parse_date()`, `parse_time()`

Some difficulties

Parsing is not always trivial.

- Numbers are written differently in different parts of the world (“,” vs “.”)
- Numbers are often surrounded by other characters (“$1000”, “10%”)
- Numbers often contain “grouping” characters (“1,000,000”)
- There are many different ways of writing dates and times
- Times can be in different timezones
- Encodings: special characters in other languages

Locales

A `locale` specifies common options varying between languages and places

To create a new locale, you use the `locale()` function:

```
locale(date_names = "en", date_format = "%AD", time_format = "%AT",
  decimal_mark = ".", grouping_mark = ",", tz = "UTC",
  encoding = "UTF-8", asciify = FALSE)
```

```
## <locale>
## Numbers:  123,456.78
## Formats:  %AD / %AT
## Timezone: UTC
## Encoding: UTF-8
## <date_names>
## Days:   Sunday (Sun), Monday (Mon), Tuesday (Tue), Wednesday (Wed), Thursday
##         (Thu), Friday (Fri), Saturday (Sat)
## Months: January (Jan), February (Feb), March (Mar), April (Apr), May (May),
##         June (Jun), July (Jul), August (Aug), September (Sep), October
##         (Oct), November (Nov), December (Dec)
## AM/PM:  AM/PM
```

```
# To learn more:
vignette("locales")
```

Parsing dates

```
parse_date("2010-10-01")
```

```
## [1] "2010-10-01"
```

Example: French format with full name of month:

```
parse_date("1 janvier 2010")
```

```
## Warning: 1 parsing failure.
## row col   expected         actual
##   1  -- date like  1 janvier 2010
```

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

```
parse_date("1 janvier 2010", format="%d %B %Y", locale=locale("fr"))
```

```
## [1] "2010-01-01"
```

Learn more by typing `?parse_date`

Parsing times

`parse_time` expects the hour, :, the minutes,
optionally : and seconds, and an optional am/pm specifier

```
parse_time("01:10 am")
```

```
## 01:10:00
```

Parsing dates and times:

```
parse_datetime("2001-10-10 20:10", locale = locale(tz = "US/Pacific"))
```

```
## [1] "2001-10-10 20:10:00 PDT"
```

For more details, see the book R for data science or use the documentation.

Parsing real numbers

Using a different decimal mark

```
parse_double("1,23")
```

```
## Warning: 1 parsing failure.
## row col               expected actual
##   1  -- no trailing characters    ,23
```

```
## [1] NA
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected               actual
##   <int> <int> <chr>                  <chr>
## 1     1    NA no trailing characters ,23
```

```
parse_double("1,23", locale = locale(decimal_mark = ","))
```

```
## [1] 1.23
```

Parsing numbers

`parse_number` ignores non-numeric characters before and after.

```
parse_number("$100")
```

```
## [1] 100
```

```
parse_number("20%")
```

```
## [1] 20
```

```
parse_number("$123,456,789")
```

```
## [1] 123456789
```

```
parse_number("cost: $123.45")
```

```
## [1] 123.45
```

Parsing numbers with locales

```
# Separation used in Switzerland
parse_number("123'456'789", locale = locale(grouping_mark = "'"))
```

```
## [1] 123456789
```

Parsing factors

Factors are categorical variables with a known set of possible values.

```
fruits <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruits)
```

```
## Warning: 1 parsing failure.
## row col           expected   actual
##   3  -- value in level set bananana
```

```
## [1] apple  banana <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected           actual
##   <int> <int> <chr>              <chr>
## 1     3    NA value in level set bananana
## Levels: apple banana
```

If the levels are not known in advance,

```
parse_factor(c("apple", "banana", "apple"), NULL)
```

```
## [1] apple  banana apple
## Levels: apple banana
```

Parsing files with `readr`

readr's strategy

readr uses a heuristic to determine column type, using the first 1000 rows

You can emulate this process with two functions:

- `guess_parser()`: returns readr's best guess
- `parse_guess()`: uses that guess to parse the column

```
guess_parser("15:01")
```

```
## [1] "time"
```

```
guess_parser("Oct 10, 2010; 15:01")
```

```
## [1] "character"
```

```
parse_guess("12,352,561")
```

```
## [1] 12352561
```

```
parse_guess(c("TRUE", "FALSE"))
```

```
## [1]  TRUE FALSE
```

The heuristic tries a sequence of types, stopping when it finds a match.
If none of these rules apply, then the column will stay as a vector of strings.

When the default strategy fails

The default strategy does not always work for larger files.

- The first 1000 rows might be a special case
- The column might contain a lot of missing values

```
challenge <- read_csv(readr_example("challenge.csv"))
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_logical()
## )
```

```
## Warning: 1000 parsing failures.
##  row col           expected     actual
## 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16 '/home/rstudio-user/R/x86_64-pc-linux-gnu-library
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 '/home/rstudio-user/R/x86_64-pc-linux-gnu-library
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 '/home/rstudio-user/R/x86_64-pc-linux-gnu-library
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 '/home/rstudio-user/R/x86_64-pc-linux-gnu-library
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 '/home/rstudio-user/R/x86_64-pc-linux-gnu-library
## .... ... .................. .......... ..........................................
## See problems(...) for more details.
```

Examining what went wrong

See `problems(...)` for more details.

```
problems(challenge)
```

```
## # A tibble: 1,000 x 5
##      row col   expected       actual   file
##    <int> <chr> <chr>          <chr>    <chr>
##  1  1001 y     1/0/T/F/TRUE/… 2015-01… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  2  1002 y     1/0/T/F/TRUE/… 2018-05… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  3  1003 y     1/0/T/F/TRUE/… 2015-09… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  4  1004 y     1/0/T/F/TRUE/… 2012-11… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  5  1005 y     1/0/T/F/TRUE/… 2020-01… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  6  1006 y     1/0/T/F/TRUE/… 2016-04… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  7  1007 y     1/0/T/F/TRUE/… 2011-05… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  8  1008 y     1/0/T/F/TRUE/… 2020-07… '/home/rstudio-user/R/x86_64-pc-linux-gn…
##  9  1009 y     1/0/T/F/TRUE/… 2011-04… '/home/rstudio-user/R/x86_64-pc-linux-gn…
## 10  1010 y     1/0/T/F/TRUE/… 2010-05… '/home/rstudio-user/R/x86_64-pc-linux-gn…
## # … with 990 more rows
```

It seems that first column should be double

Now work column-by-column until there are no problems remaining.

Fixing the column specifications

Automatic colomn specifications are

```
challenge <- read_csv(readr_example("challenge.csv"),
  col_types = cols(x = col_integer(), y = col_character()) )
```

First column should be real

```
( challenge <- read_csv(readr_example("challenge.csv"),
    col_types = cols(x = col_double(), y = col_character()) ) )
```

```
## # A tibble: 2,000 x 2
##        x y
##    <dbl> <chr>
##  1   404 <NA>
##  2  4172 <NA>
##  3  3004 <NA>
##  4   787 <NA>
##  5    37 <NA>
##  6  2332 <NA>
##  7  2489 <NA>
##  8  1449 <NA>
##  9  3665 <NA>
## 10  3863 <NA>
## # … with 1,990 more rows
```

Fixing the column specifications (2)

Are we done? Check the "y" column

```
tail(challenge)
```

```
## # A tibble: 6 x 2
##       x y
##   <dbl> <chr>
## 1 0.805 2019-11-21
## 2 0.164 2018-03-29
## 3 0.472 2014-08-04
## 4 0.718 2015-08-16
## 5 0.270 2020-02-04
## 6 0.608 2019-01-06
```

Not yet: dates are stored as strings. To fix this, we use:

```
challenge <- read_csv(readr_example("challenge.csv"),
  col_types = cols(x = col_double(), y = col_date() ) )
```

Every `parse_xyz()` function has a corresponding `col_xyz()` function. `col_xyz()` tells `readr` how to load the data.

Diagnosing problems

Maybe easier to diagnose problems if all columns are read as characters:

```
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character()) )
```

Then use `type_convert()`, which applies the parsing heuristics to the character columns.

```
type_convert(challenge2)
```

```
## # A tibble: 2,000 x 2
##        x y
##    <dbl> <date>
##  1   404 NA
##  2  4172 NA
##  3  3004 NA
##  4   787 NA
##  5    37 NA
##  6  2332 NA
##  7  2489 NA
##  8  1449 NA
##  9  3665 NA
## 10  3863 NA
## # … with 1,990 more rows
```

Importing other types of data

Other packages for importing data

We will not go into the details in this course.
We only list a few other useful packages for importing data.

Rectangular data:

- Package `haven` reads SPSS, Stata, and SAS files.
- Package `readxl` reads excel files (both .xls and .xlsx).
- Package `DBI`, along with a database specific backend (e.g. `RMySQL`, `RSQLite`, `RPostgreSQL` etc) allows you to run SQL queries against a database and return a data frame.

Hierarchical data:

- `jsonlite` for json (common for browser-server communications)
- `xml2` for XML (common for textual data in web services)

And many more are available.

Exercises

Exercise 1: Parse dates

Generate the correct format string to parse each of the following dates and times:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015) - 3:04PM", "July 1 (2015) - 4:04PM")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"     # 5:05PM
```

Exercise 2: Read a CSV file

Import into R this NCHS dataset on leading Causes of death in the United States, from 1999 to 2015:

https://data.cdc.gov/api/views/bi63-dtpu/rows.csv

Hint: you can do everything inside R.

Use the most appropriate column type for each variable.

How many different values does the variable “State” take? What are they?

Exercise 3: Read a text file

Import into R this dataset on New York City school level College Board SAT results for the graduating seniors of 2010.

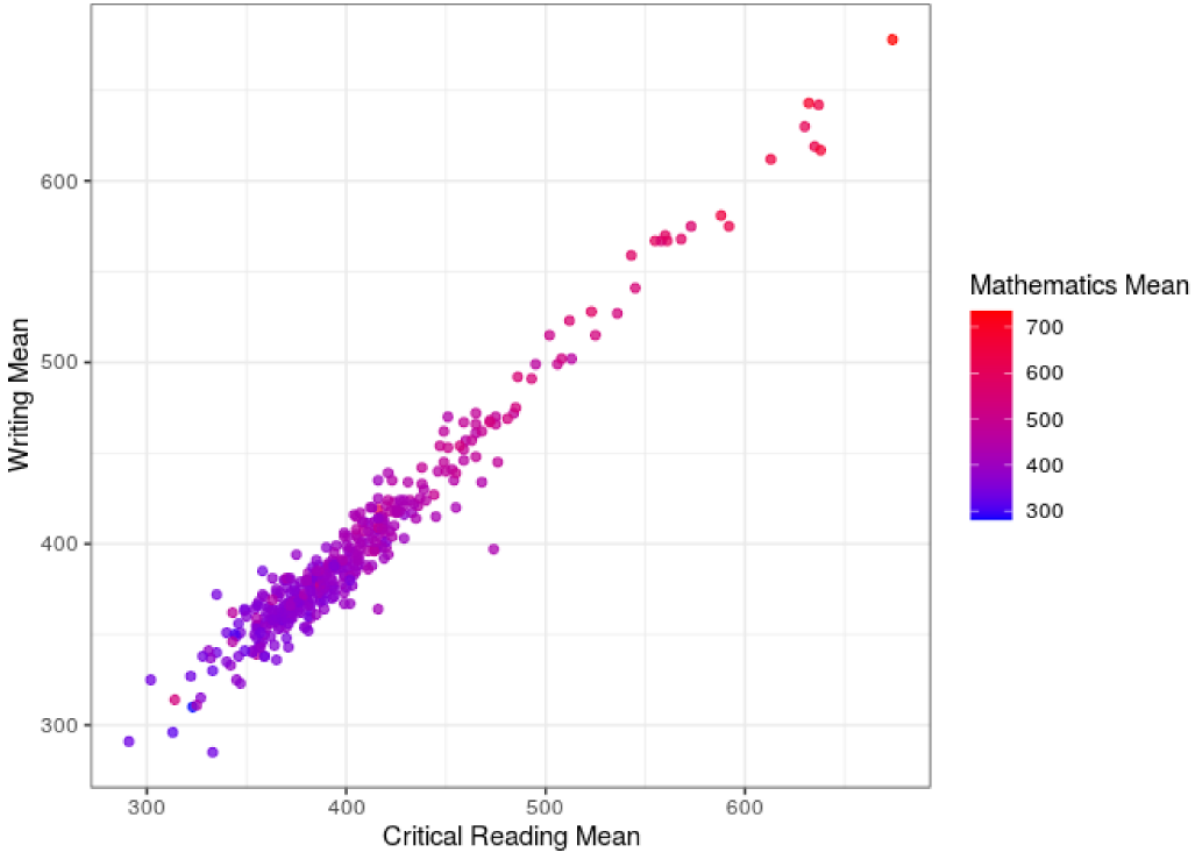
https://data.cityofnewyork.us/api/views/zt9s-n5aj/rows.csv

Use the function `read_delim` and the most appropriate column type for each variable.

Hint: you can do everything inside R.

Exercise 4: Visualizing data

Investigate the relation between “reading”, “writing”, and “math” SAT scores by producing the following scatter plot.



Transforming data (I)

Data trasnformation

At this point, we have learned how to

- Import data into R
- Visualize it

However, rarely do you get the data in the form you need.

For example, you may need to:

- create new variables or summaries
- rename the variables
- reorder the observations

We will now learn how to do this and more.

New York City flights data

Dataset on flights departing New York City in 2013.

```
install.packages("nycflights13")
```

```
library(nycflights13)
flights
```

```
## # A tibble: 336,776 x 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>
```

The `dplyr` package

The `dplyr` package is part of the core `tidyverse`.

```
library(tidyverse)
```

It solves the vast majority of your data manipulation challenges:

- Pick observations by their values: `filter()`
- Reorder the rows: `arrange()`
- Pick variables by their names: `select()`
- Create new variables with functions of existing variables: `mutate()`
- Collapse many values down to a single summary: `summarise()`

All verbs work similarly:

- The first argument is a tibble (or data frame)
- The subsequent ones describe what to do, using the variable names
- The result is a new tibble

Filter rows with `filter()`

`filter()` allows you to subset observations based on their values.

```
filter(flights, month == 1, day == 2)
```

```
## # A tibble: 943 x 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     2       42           2359        43      518            442
##  2  2013     1     2      126           2250       156      233           2359
##  3  2013     1     2      458            500        -2      703            650
##  4  2013     1     2      512            515        -3      809            819
##  5  2013     1     2      535            540        -5      831            850
##  6  2013     1     2      536            529         7      840            828
##  7  2013     1     2      539            545        -6      959           1022
##  8  2013     1     2      554            600        -6      845            901
##  9  2013     1     2      554            600        -6      841            851
## 10  2013     1     2      554            600        -6      909            858
## # … with 933 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>
```

Package `dplyr` executes the filtering and returns a new data frame.
It never modifies the original one.

Comparison operators in R

In order to use `filter()`, you need to use comparison operators.

R provides the standard suite: >, >=, <, <=, != (not equal), and == (equal).

The == operator can be tricky with floating point numbers.

```
sqrt(2) ^ 2 == 2
```

```
## [1] FALSE
```

```
1/49 * 49 == 1
```

```
## [1] FALSE
```

```
near(sqrt(2) ^ 2, 2)
```

```
## [1] TRUE
```

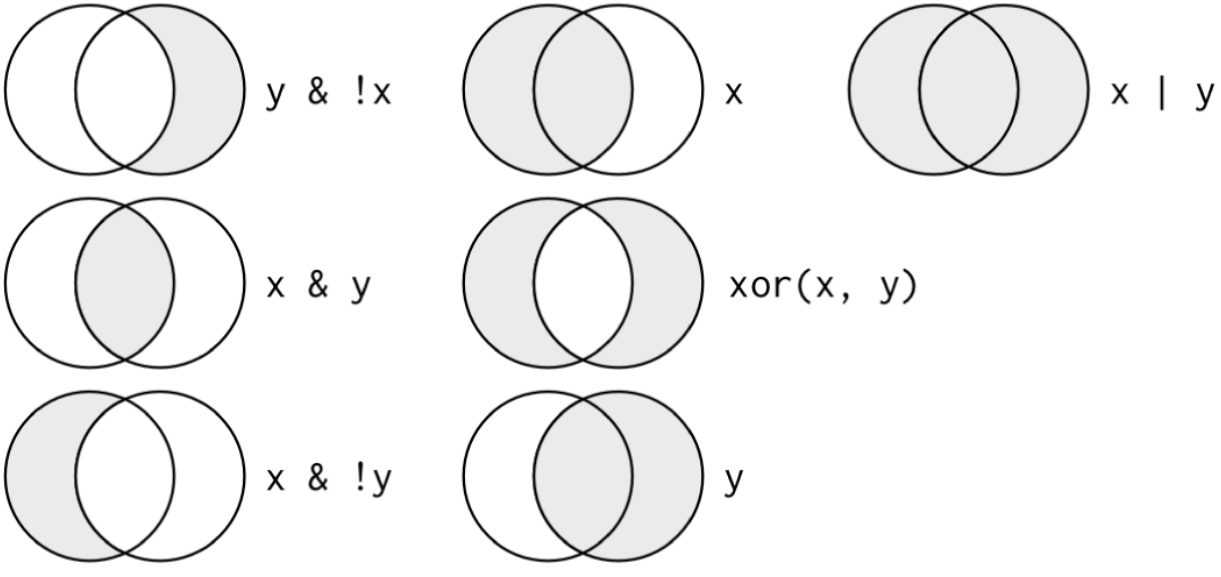
```
near(1/49 * 49, 1)
```

```
## [1] TRUE
```

`near()` is the safe way of checking if two vectors of floating point numbers are (pairwise) equal.

Logical operators in R

By default, `filter()` combines multiple arguments with a logical AND.



Credit to http://r4ds.had.co.nz

For example, this finds all flights that departed in November or December:

```
filter(flights, month == 11 | month == 12)
filter(flights, month %in% c(11,12) )        # Equivalent short-hand
```

The logic of missing values in R

Missing values are contagious.

```
NA > 5
```

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

```
NA == 10
```

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

```
NA + 2
```

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

```
NA == NA
```

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

```
is.na(NA)
```

```
## [1] TRUE
```

```
is.na(2)
```

```
## [1] FALSE
```

`filter()` includes rows where condition is TRUE, neither FALSE nor NA.

```
filter(flights, is.na(arr_delay) | arr_delay > 10) # Example
```

Arrange rows with `arrange()`

Similar to `filter()`, but it only changes the order of rows.

It takes a data frame and a set of column names to order by.
Additional column names are used to break ties.
For descending order, use the function `desc()` around the column name.

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

```
## # A tibble: 336,776 x 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    12     1       13           2359        14      446            445
##  2  2013    12     1       17           2359        18      443            437
##  3  2013    12     1      453            500        -7      636            651
##  4  2013    12     1      520            515         5      749            808
##  5  2013    12     1      536            540        -4      845            850
##  6  2013    12     1      540            550       -10     1005           1027
##  7  2013    12     1      541            545        -4      734            755
##  8  2013    12     1      546            545         1      826            835
##  9  2013    12     1      549            600       -11      648            659
## 10  2013    12     1      550            600       -10      825            854
## # … 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()`

`select()` zooms in on a subset of variables, specified by name.

This is most useful for large datasets with many variables.

```
# Columns year and month
select(flights, year, month)
```

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

```
# All except those from day to hour
select(flights, -(day:hour))
```

```
## # A tibble: 336,776 x 4
##     year month minute time_hour
##    <int> <int>  <dbl> <dttm>
##  1  2013     1     15 2013-01-01 05:00:00
##  2  2013     1     29 2013-01-01 05:00:00
##  3  2013     1     40 2013-01-01 05:00:00
##  4  2013     1     45 2013-01-01 05:00:00
##  5  2013     1      0 2013-01-01 06:00:00
##  6  2013     1     58 2013-01-01 05:00:00
##  7  2013     1      0 2013-01-01 06:00:00
##  8  2013     1      0 2013-01-01 06:00:00
##  9  2013     1      0 2013-01-01 06:00:00
## 10  2013     1      0 2013-01-01 06:00:00
## # … with 336,766 more rows
```

Helper functions for `select()`

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

- `starts_with("abc")`: matches names that begin with “abc”
- `ends_with("xyz")`: matches names that end with “xyz”
- `contains("ijk")`: matches names that contain “ijk”
- `matches`: selects variables that match a regular expression
- `num_range("x", 1:3)`: matches x1, x2 and x3.

To move some columns to the beginning, use the `everything()` helper.

```
select(flights, time_hour, air_time, everything())
```

See `?select` for more details.

```
# Example: smaller dataset that we will use later
flights_sml <- select(flights, year:day, ends_with("delay"),
                      air_time, distance)
```

Add new variables with `mutate()`

`mutate()` adds new columns that are a function of the existing ones.

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

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

To discard old variables, use `transmute()` instead of `mutate()`.

Helper functions for `mutate()`

Many functions can be used together with `mutate()`.

Some examples:

- Arithmetic operators (+, -, *, /)
- Modular arithmetic: integer division (%), remainder (%%)
- Logarithms (`log`, `log10`) and exponentials (`exp`)
- Offsets (`lead`, `lag`)
- Cumulative and rolling aggregates (`cumsum`, `cumprod`, `cummin`, `cummax`)
- Logical comparisons (<, <=, >, >=, ==, !=)
- Ranking (`min_rank`, `percent_rank`)

For details: type `?function_name`.

Basically any vectorized function can be used here.

Creating summaries with `summarise()`

The `summarize` function collapses a data frame to a single row.

```
summarise(flights,
          mean_distance = mean(distance, na.rm = TRUE),
          mean_arr_delay = mean(arr_delay, na.rm = TRUE) )
```

```
## # A tibble: 1 x 2
##   mean_distance mean_arr_delay
##           <dbl>          <dbl>
## 1         1040.           6.90
```

The argument "na.rm = TRUE" in the `mean()` function is important.

```
summarise(flights,
          mean_distance = mean(distance),
          mean_arr_delay = mean(arr_delay) )
```

```
## # A tibble: 1 x 2
##   mean_distance mean_arr_delay
##           <dbl>          <dbl>
## 1         1040.             NA
```

Grouped summaries with `summarise()`

`summarise()` is more useful when used with `group_by()`.

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, mean_distance = mean(distance, na.rm = TRUE),
          mean_arr_delay = mean(arr_delay, na.rm = TRUE))
```

```
## # A tibble: 365 x 5
## # Groups:   year, month [12]
##     year month   day mean_distance mean_arr_delay
##    <int> <int> <int>         <dbl>          <dbl>
##  1  2013     1     1         1077.         12.7
##  2  2013     1     2         1053.         12.7
##  3  2013     1     3         1037.          5.73
##  4  2013     1     4         1032.         -1.93
##  5  2013     1     5         1068.         -1.53
##  6  2013     1     6         1052.          4.24
##  7  2013     1     7          998.         -4.95
##  8  2013     1     8          986.         -3.23
##  9  2013     1     9          981.         -0.264
## 10  2013     1    10          993.         -5.90
## # … with 355 more rows
```

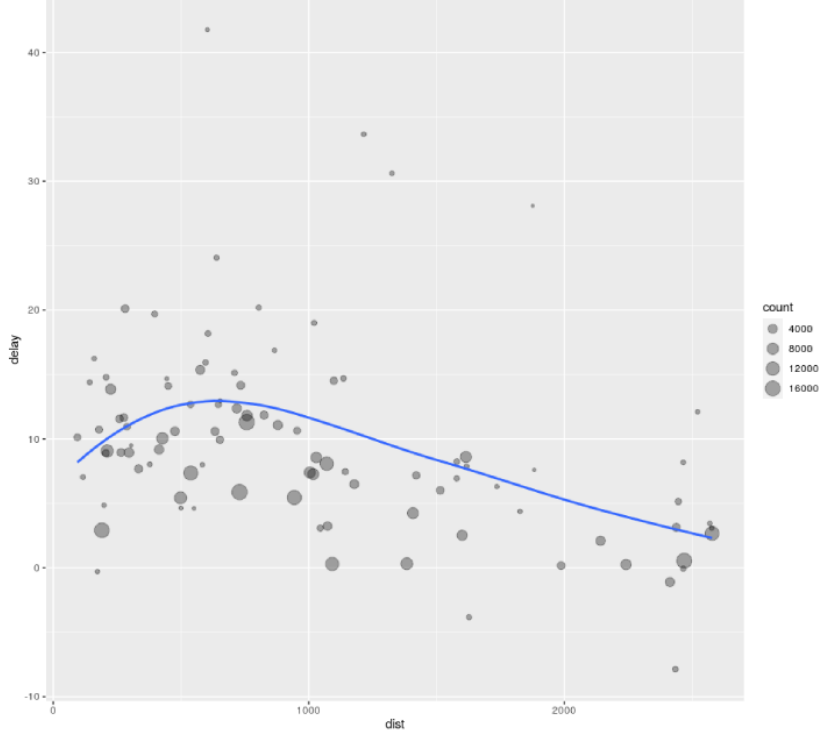
Transforming data (II)

A data analysis example

Task: relationship between distance and average delay for each location.

```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
  count = n(),                    # Count the number of occurrences
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE) )
delay <- filter(delay, count > 20, dest != "HNL")
```

```
ggplot(delay,
  aes(x=dist, y=delay)) +
  geom_point(aes(size=count),
             alpha=1/3) +
  geom_smooth(se = FALSE)
```



It looks like delays increase with distance up to ~750 miles and then decrease.

Combining multiple operations with the pipe

Our analysis had three steps:

- Group flights by destination
- Summarise to compute distance, average delay, and number of flights
- Filter to remove noisy points and Honolulu airport

Every step creates a new dataset, which we do not need.

Solution: the pipe operator "%>%" (general for `tidyverse`).

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

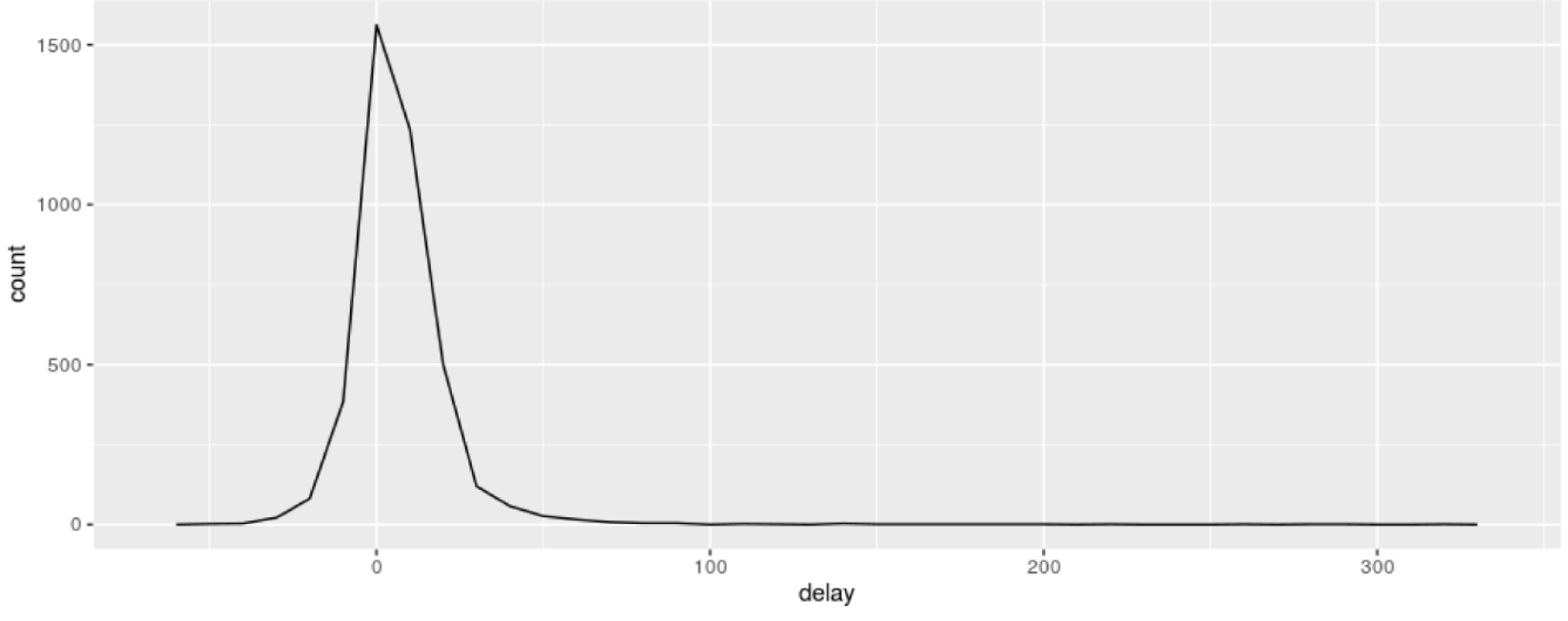
Focus on the transformations, not what's being transformed. Easier to read.

`x %>% f(y)` turns into: `f(x, y)`

`x %>% f(y) %>% g(z)` turns into: `g(f(x, y), z)`

Counting values

```
delays <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%   # Not canceled
  group_by(tailnum) %>%                              # By flight number
  summarise(delay = mean(arr_delay) )                # Average delay

ggplot(data=delays, mapping=aes(x=delay)) + geom_freqpoly(binwidth=10)
```

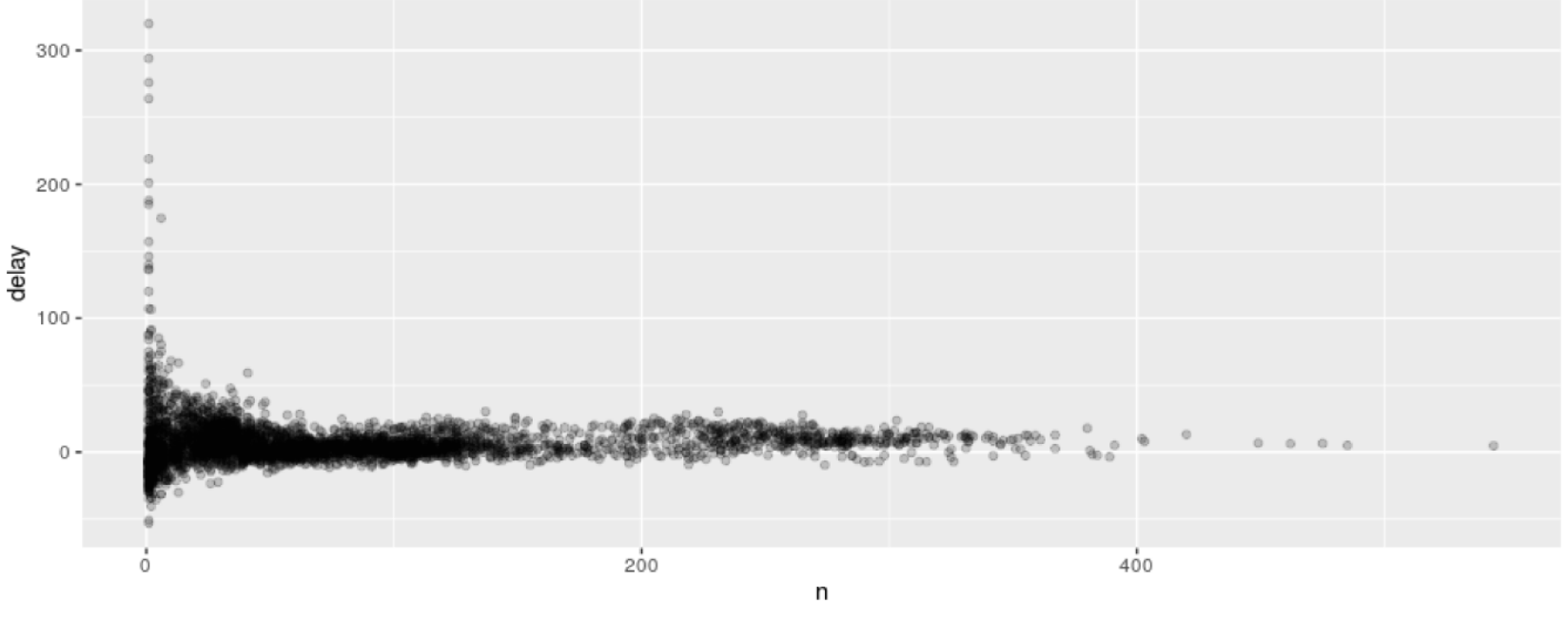


Whenever you aggregate, it's useful to include either a count, `n()`, or a count of non-missing values, `sum(!is.na(x))`.

Counting values (II)

More insight with scatterplot of number of flights vs. average delay.

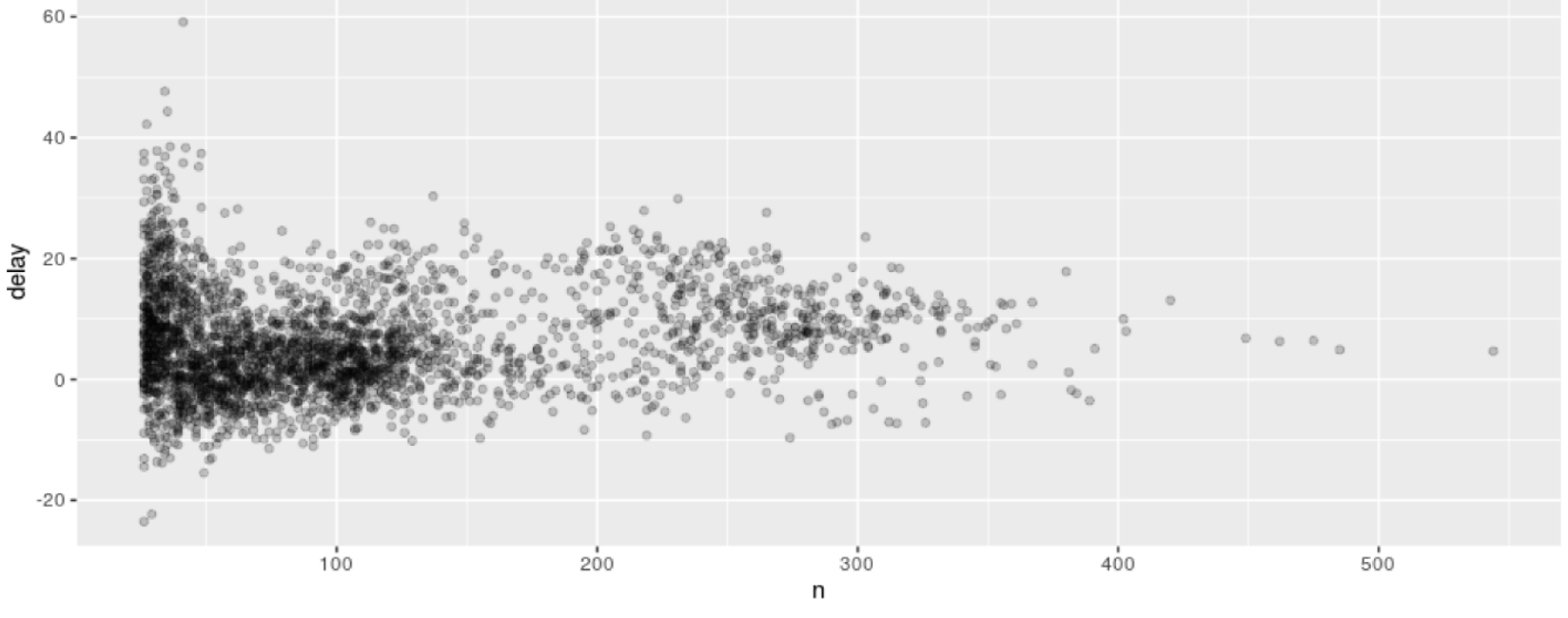
```
delays <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%   # Not canceled
  group_by(tailnum) %>%                              # By flight number
  summarise(n = n(), delay = mean(arr_delay) )       # Count and delay

ggplot(data=delays, aes(x=n, y=delay)) + geom_point(alpha=0.2)
```



Counting values (III)

We can filter out the groups with the smallest numbers of observations.

```
delays %>%
  filter(n > 25) %>%
  ggplot(aes(x=n, y=delay)) + geom_point(alpha=0.2)
```



Note that we can also pipe data into `ggplot2`.

Useful summary functions

- Measures of location: `mean()`, `median()` or spread: `sd()`
- Measures of rank: `min()`, `quantile()`, `max()`
- Measures of position: `first()`, `nth()`, `last()`
- Counts: `n()`, `sum(!is.na(x))`, `n_distinct()`
- Counts of logical values: `sum(x>10)`

```
# What proportion of flights are delayed by more than an hour?
flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay)) %>%
  group_by(year,month,day) %>%
  summarise(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
```

Ungrouping

If you need to undo the grouping, use `ungroup()`.

```
flights %>%
  group_by(year,month,day) %>%   # grouped by date
  ungroup()  %>%                 # no longer grouped
  summarise(flights = n())       # summary of all flights
```

```
## # A tibble: 1 x 1
##   flights
##     <int>
## 1  336776
```

This makes sense if you do some operation on groups before ungrouping.

Other uses of grouping

Grouping can also do convenient operations with `mutate()` and `filter()`.

```
# Find the worst members of each group
flights %>%
  group_by(year,month,day) %>%       # grouped by date
  filter(rank(desc(arr_delay)) < 10) # largest 10 delays
```

```
# Find all groups larger than a threshold
flights %>%
  group_by(dest) %>%                 # grouped by destination
  filter(n() > 365)                  # more than 365 flights
```

```
# Standardise to compute per group metrics:
# proportion of delays for each popular destination
flights %>%
  group_by(dest) %>%                              # group by destination
  filter(n() > 365, arr_delay > 0) %>%            # popular and delayed
  mutate(prop_delay=arr_delay/sum(arr_delay))%>% # proportion of delays
  select(year:day, dest, arr_delay, prop_delay)  # select columns
```

Functions that work most naturally in grouped mutates and filters are called window functions; see `vignette("window-functions")`.