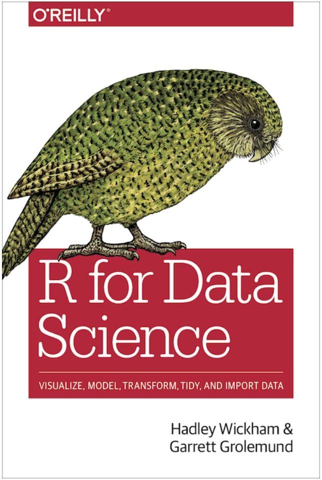
URPP tutorial on dplyr/tidyverse

Stefan Wyder - URPP Evolution in Action

June 1, 2018

R for Data Science



http://r4ds.had.co.nz

Installation

You can install the complete tidyverse with a single line of code:

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

install also the data package nycflights13

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

Load packages

```
library(tidyverse)
library(nycflights13)
```

This data frame contains all 336,776 flights that departed from New York City in 2013 flights
see `?flights`

Data Science cycle

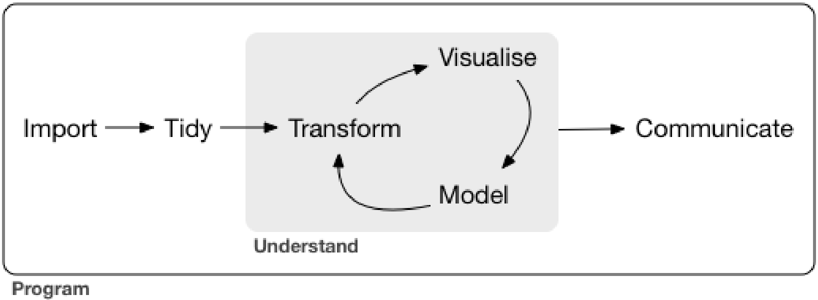


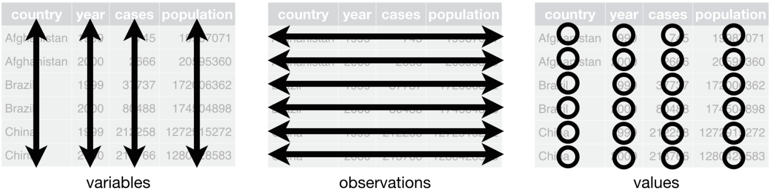
Table of Contents

- Data transformation (Chapter 5)
- Tibbles (Chapter 10)
- Relational data (Chapter 13)

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.



Tibble (Chapter 10)

Tibbles are data frames, but slightly tweaked to work better in the tidyverse
- better printing
- never changes the type of the inputs (strings to factors!)
- never changes the names of variables

```
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>
```

5 key dplyr functions

- `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
- `summarise()`: Collapse many values down to a single summary

1 example

```r
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>
```

filter()

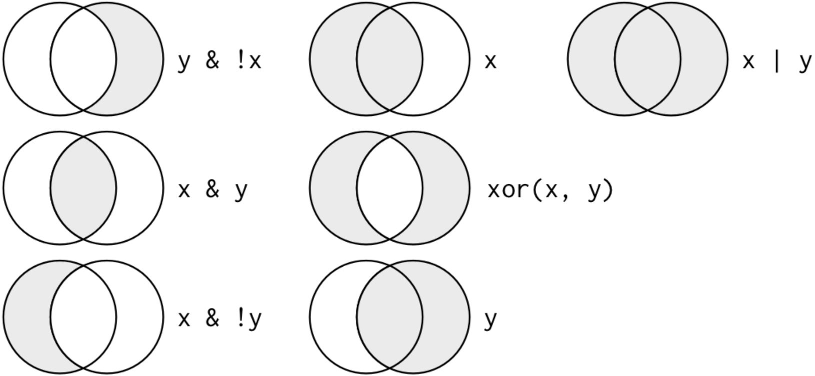
Guess which ones are correct. What does it do?

```
filter(flights, month == 1, day == 1)
filter(flights, month == 11 & month == 12)
filter(flights, month == 11 | month == 12)
filter(flights, month == 11 | 12)
filter(flights, month %in% c(11, 12))
```

filter()

```r
# finds all flights that departed on Jan 1
filter(flights, month == 1, day == 1)
filter(flights, month == 1 & day == 1)
# finds all flights that departed in Nov or Dec:
filter(flights, month == 11 | month == 12)
# doesn't work:
filter(flights, month == 11 | 12)
# %in% also works:
filter(flights, month %in% c(11, 12))
```

Boolean operations



Be careful with arithmetic comparisons

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

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

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

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

Use `near` with a tolerance:

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

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

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

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

Exercise 1

Find flights that weren't delayed (on arrival or departure) by more than two hours

Solution 1

You could use either of the following two filters:

```r
filter(flights, arr_delay <= 120, dep_delay <= 120)
filter(flights, !(arr_delay > 120 | dep_delay > 120))
```

Missing values

`filter()` only includes rows where the condition is TRUE; it excludes both FALSE and NA values.

If you want to preserve missing values, ask for them explicitly:

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

Exercises 2

(from 5.2.4 Exercises) Find all flights that

1. Had an arrival delay of two or more hours
2. Flew to Houston (IAH or HOU)
3. Departed in summer (July, August, and September)
4. Arrived more than two hours late, but didn't leave late
5. Were delayed by at least an hour, but made up over 30 minutes in flight
6. How many flights have a missing dep_time? What other variables are missing? What might these rows represent?

`arrange()`: order rows

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order.

```
arrange(flights, year)
arrange(flights, year, month, day)
arrange(flights, desc(arr_delay))  # order in decreasing order
```

missing values are always sorted at the end

Exercises 3

(from (5.3.1 Exercises))

1. Sort flights to find the most delayed flights. Find the flights that left earliest.
2. How could you use arrange() to sort all missing values to the start? (Hint: use is.na())
3. Which flights traveled the longest? Which traveled the shortest?

Select columns with `select()`

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

```
# Select all columns between year and day (inclusive)
select(flights, year:day)
```

`everything()` is useful if you have a handful of variables you'd like to move to the start of the data frame

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

`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("(.)\\1")`: selects variables that match a regular expression. Here: any variables that contain repeated characters.
- `num_range("x", 1:3)` matches x1, x2 and x3.

See `?select` for more details.

Exercises 4

(from 5.4.1 Exercises)

1. Brainstorm as many ways as possible to select dep_time, dep_delay, arr_time, and arr_delay from flights.
2. What happens if you include the name of a variable multiple times in a select() call?
3. Does the result of running the following code surprise you? How do the select helpers deal with case by default? How can you change that default?

```
select(flights, contains("TIME"))
```

Add new variables with `mutate()`

`mutate()` always adds new columns at the end of your dataset, as function of the existing columns.

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

Note that you can refer to columns that you've just created

`mutate()` works with all vectorized functions

The key property is that the function must be vectorised: it must take a vector of values as input, return a vector with the same number of values as output.

- Arithmetic operators: `+ - * / ^`
- Modular arithmetic: %/% (integer division) and %% (remainder)
- Logs: `log()`, `log2()`, `log10()`
- Offsets: `lead()` and `lag()`
- Logical comparisons: `<`, `<=`, `>`, `>=`, `!=`
- Ranking: `min_rank()`, `min_rank(desc())` also its variants `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`
- Cumulative and rolling aggregates: running sums, products, mins, maxes and means: `cumsum()`, `cumprod()`, `cummin()`, `cummax()` and `cummean()`
- conjuction of functions: i.e. `x / sum(x)`: proportion of a total, and `y - mean(y)`: diff from the mean

transmute()

If you only want to keep the new variables, use transmute():

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
```

%/% (integer division) handy to because it allows you to break integers up into pieces (here: hours)

Exercises 5

(from 5.5.2 Exercises)

1. Currently dep_time and sched_dep_time are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.
2. Find the 10 most delayed flights using a ranking function. How do you want to handle ties? Carefully read the documentation for min_rank().

Grouped summaries with summarise()

summarise() is not terribly useful unless we pair it with group_by():

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

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     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
```

Combining multiple operations with the pipe

Explore the relationship between the distance and average delay for each location

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

a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard and slow.

the pipe %>%

There's another way to tackle the same problem with the pipe, %>%:

```
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")
```

- pronounced as "then"
- much more readable (left-right,top-bottom)
- key criteria for belonging to the tidyverse (only exception: ggplot2)

Missing values

```r
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     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
```

We get a lot of missing values! Usual rule of missing values: if there's any missing value in the input, the output will be a missing value.

Missing values cont.

Fortunately, all aggregation functions have an na.rm argument which removes the missing values prior to computation:

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

```
## # A tibble: 365 x 4
## # Groups:   year, month [?]
##     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
```

Missing values are due to cancelled flights. In this case we could also create a new data frame without cancelled flights.

Counts

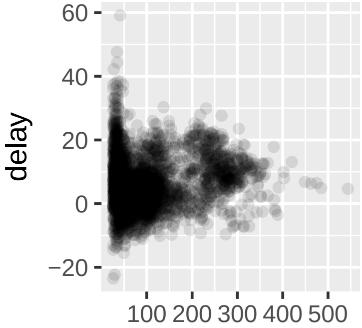
Whenever you do any aggregation, it's always a good idea to include either a count `n()`, or a count of non-missing values `sum(!is.na(x))`. Not to draw conclusions based on very small amounts of data

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

Plot

It's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```



Variation

- Measures of location: `mean(x)`, `median(x)`
- Measures of spread: `sd(x)`, `IQR(x)`, `mad(x)`
- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`
- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`: allow to set a default position
- Counts: `n()` returns the size of the current group.
  `sum(!is.na(x))` returns the number of non-missing values
  `n_distinct(x)` counts the number of distinct (unique) values

Counts

Counts are so useful that dplyr provides a simple helper if all you want is a count:

```
not_cancelled %>%
  count(dest)
```

```
## # A tibble: 104 x 2
##    dest      n
##    <chr> <int>
##  1 ABQ     254
##  2 ACK     264
##  3 ALB     418
##  4 ANC       8
##  5 ATL   16837
##  6 AUS    2411
##  7 AVL     261
##  8 BDL     412
##  9 BGR     358
## 10 BHM     269
## # ... with 94 more rows
```

Counts cont.

You can optionally provide a weight variable. For example, you could use this to "count" (sum) the total number of miles a plane flew:

```
not_cancelled %>%
  count(tailnum, wt = distance)
```

```
## # A tibble: 4,037 x 2
##    tailnum       n
##    <chr>     <dbl>
##  1 D942DN    3418.
##  2 N0EGMQ  239143.
##  3 N10156  109664.
##  4 N102UW   25722.
##  5 N103US   24619.
##  6 N104UW   24616.
##  7 N10575  139903.
##  8 N105UW   23618.
##  9 N107US   21677.
## 10 N108UW   32070.
## # ... with 4,027 more rows
```

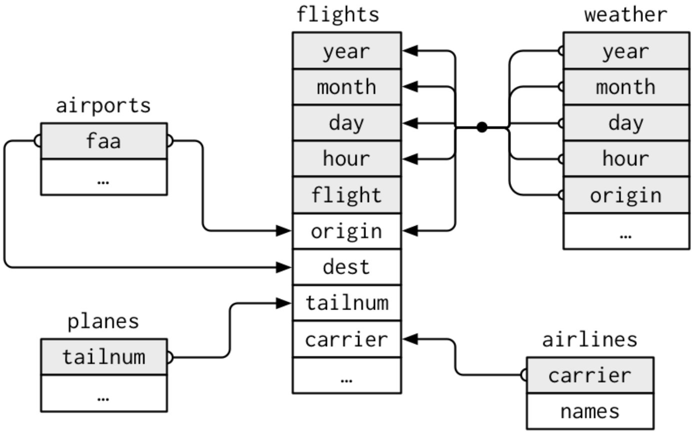
Exercises 6

(from 5.6.7 Exercises)

1. Look at the number of canceled flights per day. Is there a pattern? Is the proportion of canceled flights related to the average delay?
2. Which carrier has the worst delays? Challenge: can you disentangle the effects of bad airports vs. bad carriers? Why/why not? (Hint: think about flights %>% group_by(carrier, dest) %>% summarise(n()))
3. (advanced) For each plane, count the number of flights before the first delay of greater than 1 hour.

Relational data (Chapter 13)

Collectively, multiple tables of data are called **relational data** (relations between a pair of tables)



Relational data cont.

- 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.

Generally, dplyr is a little easier to use than SQL because dplyr is specialised to do data analysis.

Other packages of the tidyverse

- readr: import data (flat files)

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data
- stringr: functions to work with strings

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data
- stringr: functions to work with strings
- forcats: functions to work with factors

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data
- stringr: functions to work with strings
- forcats: functions to work with factors
- purrr: iteration, replacement of apply functions

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data
- stringr: functions to work with strings
- forcats: functions to work with factors
- purrr: iteration, replacement of apply functions
- ggplot2: plot data

Other packages of the tidyverse

- readr: import data (flat files)
- tidyr: tidy data
- dplyr: wrangle data
- stringr: functions to work with strings
- forcats: functions to work with factors
- purrr: iteration, replacement of apply functions
- ggplot2: plot data

- even more packages: e.g. import .xls and .xlsx sheets (readxl), for dates and date-times (lubridate), ...