

Slides: Data Manipulation and Transformation

Actuarial Data Science Online Textbook
Fei Huang, UNSW Sydney



Table of contents

- Introduction
- Data Manipulation Functions
- Relational Data



Reading List

- R for Data Science [Online Book](#), Chapters 5, 13
- Applied predictive Modelling, 3.3 (only transformations to resolve outliers), 3.4



Introduction



Motivation

- It is rare that you get the data in exactly the right form you need.
- You'll need to create some new variables or summaries, or
- you just want to rename the variables or reorder the observations in order to make the data a little easier to work with.



Using R to manipulate data

- R package: **dplyr** (a core member of **tidyverse**) for data manipulation and transformation¹
- Data: **nycflights13** package, flights departing New York City in 2013
- We use **ggplot2** to help us understand the data.

```
1 #install.packages("nycflights13")  
2 library(nycflights13)  
3 library(tidyverse)
```



1. dplyr overwrites some functions in base R. If you want to use the base version of these functions after loading dplyr, you'll need to use their full names: `stats::filter()` and `stats::lag()`



Data

```
1 flights #tibble, tweaked data frame to work better in tidyverse
```



year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4
2013	1	1	555	600	-5
2013	1	1	557	600	-3
2013	1	1	557	600	-3
2013	1	1	558	600	-2

1-10 of 10,000 rows | 1-6 of 19 columns

Previous [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) ... [1000](#) [Next](#)

```
1 view(flights) # will open the dataset in the RStudio viewer
```



Types of variables

- `int` stands for integers.
- `dbl` stands for doubles, or real numbers.
- `chr` stands for character vectors, or strings.
- `dtm` stands for date-times (a date + a time).
- `lgl` stands for logical, vectors that contain only TRUE or FALSE.
- `fctr` stands for factors, which R uses to represent categorical variables with fixed possible values.
- `date` stands for dates.



Data Manipulation Functions



Functions for data manipulation

Functions in `dplyr` package

- `%>%` Pipe operator
- `glimpse()` A glimpse into the data and its structure
- `filter()` Pick observations by their values
- `arrange()` Reorder the rows
- `select()` Pick variables by their names
- `summarise()` Collapse many values down to a single
- `group_by()` Changes the scope of each function above from operating on the entire dataset to operating on it group-by-group



Filter: Introduction

```
1 #The first argument is the name of the data frame.
2 #The subsequent arguments are the expressions that filter the data frame.
3 filter(flights, month == 1, day == 1)
```

year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4
2013	1	1	555	600	-5
2013	1	1	557	600	-3
2013	1	1	557	600	-3
2013	1	1	558	600	-2

1-10 of 842 rows | 1-6 of 19 columns

Previous 1 2 3 4 5 6 ... 85 Next

```
1 # use the assignment operator, <- to save the result
2 #jan1 <- filter(flights, month == 1, day == 1)
3 # Save and print the result at the same time
4 #(dec25 <- filter(flights, month == 12, day == 25))
```



Filter: Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite:

- `>` bigger than
- `>=` bigger than or equal to
- `<` less than
- `<=` less than or equal to
- `!=` not equal
- `==` equal¹

1. Be cautious when using `==`: floating point numbers. Consider using `near()`



Filter: Logical Operators

- Multiple arguments to `filter()` are combined with “and”: every expression must be true in order for a row to be included in the output.

Other types of combinations using Boolean operators:

- `&` is “and”
- `|` is “or”
- `!` is “not”
- `x %in% y` select every row where `x` is one of the values in `y`

According to De Morgan’s law,

- `!(x & y)` is the same as `!x | !y`
- `!(x | y)` is the same as `!x & !y`



Exercise 1

- Find all flights that departed in November or December.

```
1 filter(flights, month %in% c(11, 12))
```

year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	11	1	5	2359	6
2013	11	1	35	2250	105
2013	11	1	455	500	-5
2013	11	1	539	545	-6
2013	11	1	542	545	-3
2013	11	1	549	600	-11
2013	11	1	550	600	-10
2013	11	1	554	600	-6
2013	11	1	554	600	-6
2013	11	1	554	600	-6

1-10 of 10,000 rows | 1-6 of 19 columns

Previous 1 2 3 4 5 6 ... 1000 Next

```
1 #Alternatively, use the following
2 #filter(flights, month == 11 | month == 12)
```



Exercise 2

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

```
1 filter(flights, !(arr_delay > 120 | dep_delay > 120))
```

year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4
2013	1	1	555	600	-5
2013	1	1	557	600	-3
2013	1	1	557	600	-3
2013	1	1	558	600	-2

1-10 of 10,000 rows | 1-6 of 19 columns

Previous 1 2 3 4 5 6 ... 1000 Next

```
1 #Alternatively, use the following
2 #filter(flights, arr_delay <= 120, dep_delay <= 120)
```



Filter: Missing Values

- **NA** represents an unknown value so missing values are “contagious”: almost any operation involving an unknown value will also be unknown.
- **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:

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



	x
	<dbl>
	3
1 row	

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



	x
	<dbl>
	NA
	3
2 rows	



Arrange

- `arrange()`: Order by column names (or more complicated expressions)
- Use `desc()` to re-order by a column in descending order
- Missing values are always sorted at the end

```
1 arrange(flights, year)
```

year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	1	1	517	515	2
2013	1	1	533	529	4
2013	1	1	542	540	2
2013	1	1	544	545	-1
2013	1	1	554	600	-6
2013	1	1	554	558	-4
2013	1	1	555	600	-5
2013	1	1	557	600	-3
2013	1	1	557	600	-3
2013	1	1	558	600	-2

1-10 of 10,000 rows | 1-6 of 19 columns

Previous 1 2 3 4 5 6 ... 1000 Next

```
1 #arrange(flights, desc(dep_delay))
```



Select

- `select()` allows you to select a useful subset based on the names of the variables.

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

	year <int>	month <int>	day <int>
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1
	2013	1	1

1-10 of 10,000 rows

Previous 1 2 3 4 5 6 ... 1000 Next

```
1 #select(flights, year:day)
2 #select(flights, -(year:day))
```



Select: Useful functions

- `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.
- `everything()`

Other related functions:

- `rename()`: rename variables
- `mutate()`: create new variables with functions of existing variables



Summarise

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



		delay
		<dbl>
		12.63907
1 row		



Summarise: with group-by

- This changes the unit of analysis from the complete dataset to individual groups.

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



year <int>	month <int>	day <int>	delay <dbl>
2013	1	1	11.54892601
2013	1	2	13.85882353
2013	1	3	10.98783186
2013	1	4	8.95159516
2013	1	5	5.73221757
2013	1	6	7.14801444
2013	1	7	5.41720430
2013	1	8	2.55307263
2013	1	9	2.27647715
2013	1	10	2.84499462

1-10 of 365 rows

Previous 1 2 3 4 5 6 ... 37 Next



Summarise: Exercise with multiple operations

- Explore the relationship between the distance and average delay for each location
- There are three steps to prepare this data:
 - Group flights by destination
 - Summarise to compute distance, average delay, and number of flights
 - Filter to remove noisy points (counts below or equal to 20) and Honolulu (“HNL”) airport, which is almost twice as far away as the next closest airport

Please have a try!



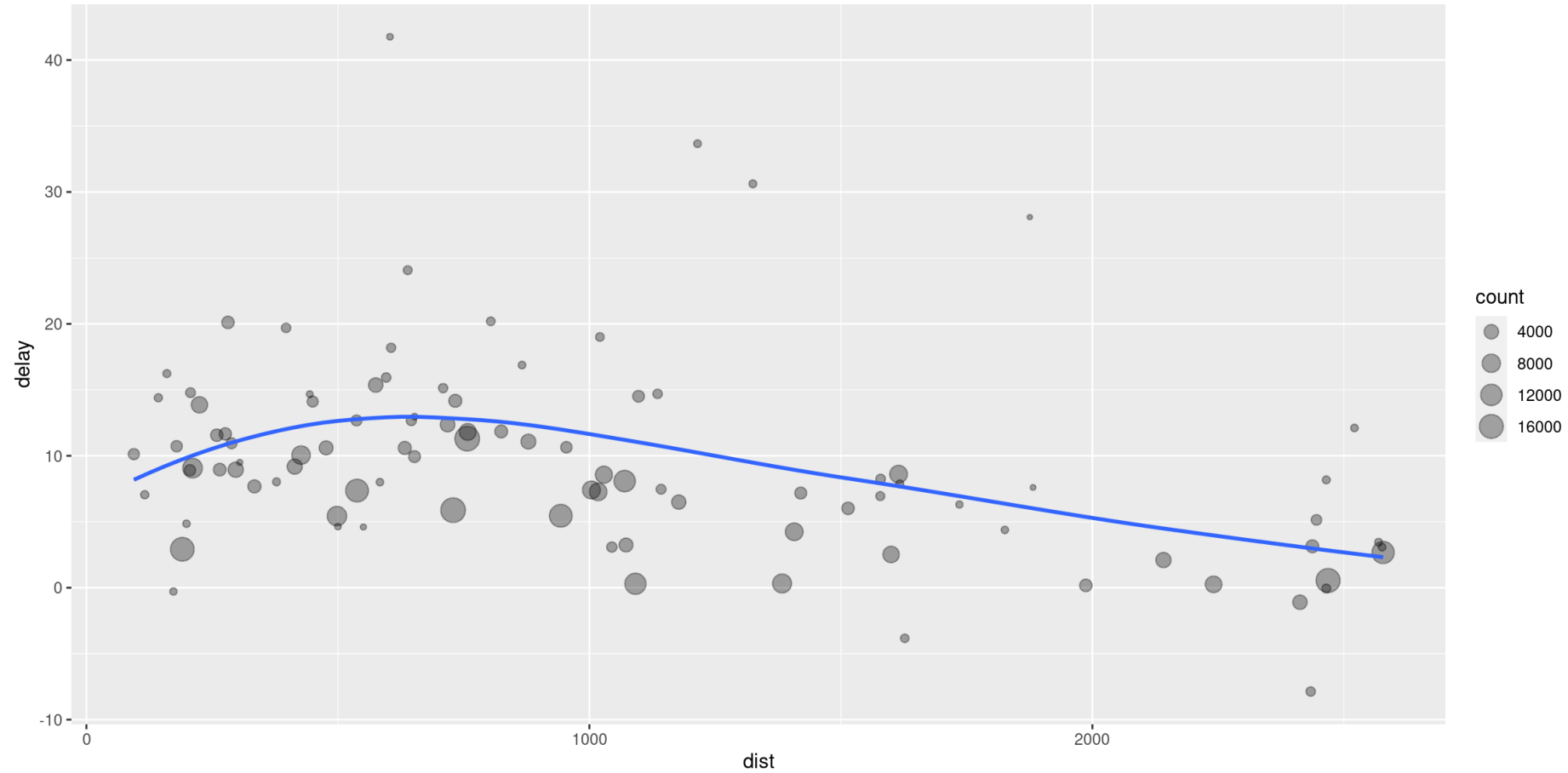
Exercise: code

```
1 by_dest <- group_by(flights, dest)
2 delay <- summarise(by_dest,
3   count = n(),
4   dist = mean(distance, na.rm = TRUE),
5   delay = mean(arr_delay, na.rm = TRUE)
6 )
7 delay <- filter(delay, count > 20, dest != "HNL")
8
9 # It looks like delays increase with distance up to ~750 miles
10 # and then decrease. Maybe as flights get longer there's more
11 # ability to make up delays in the air?
12 plot=ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
13   geom_point(aes(size = count), alpha = 1/3) +
14   geom_smooth(se = FALSE)
15 #> `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Exercise: plot

```
1 print(plot)
```



Exercise: Another way to do it using pipe %>%

```
1 delays <- flights %>%
2   group_by(dest) %>%
3   summarise(
4     count = n(),
5     dist = mean(distance, na.rm = TRUE),
6     delay = mean(arr_delay, na.rm = TRUE)
7   ) %>%
8   filter(count > 20, dest != "HNL")
```



Missing values

- `na.rm=TRUE` removes the missing values

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

year <int>	month <int>	day <int>	mean <dbl>
2013	1	1	11.54892601
2013	1	2	13.85882353
2013	1	3	10.98783186
2013	1	4	8.95159516
2013	1	5	5.73221757
2013	1	6	7.14801444
2013	1	7	5.41720430
2013	1	8	2.55307263
2013	1	9	2.27647715
2013	1	10	2.84499462

1-10 of 365 rows

Previous 1 2 3 4 5 6 ... 37 Next

Useful Summary functions

- Measures of location (central tendency): `mean(x)`, `median(x)`
- Measures of spread (variability): `sd(x)`, `IQR(x)`¹
- Measures of rank: `min(x)`, `quantile(x, 0.25)`, `max(x)`
- Measures of position: `first(x)`, `nth(x, 2)`, `last(x)`
- Counts: `n()`, `sum(!is.na(x))`, `n_distinct(x)`
 - `count(tailnum, wt = distance)`, “count” (sum) the total number of miles a plane flew
- Counts and proportions of logical values: `sum(x > 10)`, `mean(y == 0)`

1. The interquartile range (IQR) is a measure of variability, based on dividing a data set into quartiles. Q1 is the “middle” value in the first half of the rank-ordered data set. Q2 is the median value in the set. Q3 is the “middle” value in the second half of the rank-ordered data set. The interquartile range is equal to Q3 minus Q1.



Grouping by multiple variables

```
1 daily <- group_by(flights, year, month, day)
2 #n() returns the size of the current group
3 (per_day <- summarise(daily, flights = n()))
```

year <int>	month <int>	day <int>	flights <int>
2013	1	1	842
2013	1	2	943
2013	1	3	914
2013	1	4	915
2013	1	5	720
2013	1	6	832
2013	1	7	933
2013	1	8	899
2013	1	9	902
2013	1	10	932

1-10 of 365 rows

Previous 1 2 3 4 5 6 ... 37 Next



Ungrouping

```
1 daily %>%  
2   ungroup() %>%           # no longer grouped by date  
3   summarise(flights = n()) # all flights
```

**flights**

<int>

336776

1 row



Relational Data



Relational Data

- **Relational data:** multiple tables of data that are related.

Three families of verbs designed to work with relational data:

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



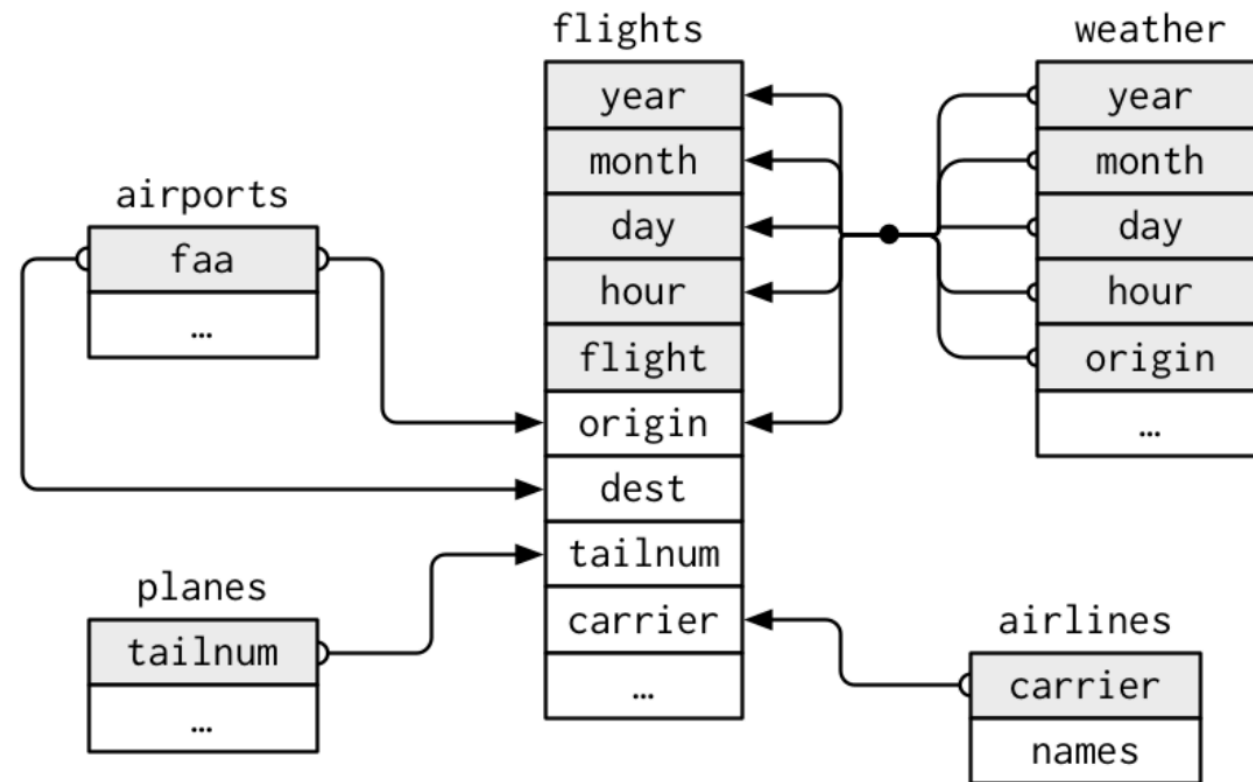
Data set

- `nycflights13` contains five tibbles that are related to each other:
 - `flights`: gives information about each flight
 - `airlines`: lets you look up the full carrier name
 - `airports`: gives information about each airport, identified by the `faa` airport code
 - `planes`: gives information about each plane, identified by its `tailnum`
 - `weather`: gives the weather at each NYC airport for each hour



Table Relations

- Each relation always concerns a pair of tables
- Understand the chain of relations between the tables that you are interested in.



Relations of Tibbles



Keys

- Primary key: uniquely identifies an observation in its own table.
 - For example, `planes$tailnum` is a primary key because it uniquely identifies each plane in the planes table.
- Foreign key: uniquely identifies an observation in another table.
 - For example, `flights$tailnum` is a foreign key because it appears in the flights table where it matches each flight to a unique plane.
- A variable can be both a primary key and a foreign key. For example, origin is part of the weather primary key, and is also a foreign key for the airport table.



Identify the primary keys

- `count()` the primary keys and look for entries where `n` is greater than one.

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



0 rows



Add a primary key

- What's the primary key in the `flights` table?
 - None
- Surrogate key: add one primary key with `mutate()` and `row_number()`
- A primary key and the corresponding foreign key in another table form a **relation**.
- Relations are typically one-to-many. You can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation.



Exercise: Add a surrogate key to flights

```
1 flights %>%
2   arrange(year, month, day, sched_dep_time, carrier, flight) %>%
3   mutate(flight_id = row_number()) %>%
4   #TThis makes it possible to see every column in a data frame.
5   glimpse()
```

Rows: 336,776

Columns: 20

```
$ year      <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2...
$ month     <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
$ day       <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
$ dep_time  <int> 517, 533, 542, 544, 554, 559, 558, 559, 558, 558, 557, ...
$ sched_dep_time <int> 515, 529, 540, 545, 558, 559, 600, 600, 600, 600, 600, ...
$ dep_delay <dbl> 2, 4, 2, -1, -4, 0, -2, -1, -2, -2, -3, NA, 1, 0, -5, -...
$ arr_time  <int> 830, 850, 923, 1004, 740, 702, 753, 941, 849, 853, 838,...
$ sched_arr_time <int> 819, 830, 850, 1022, 728, 706, 745, 910, 851, 856, 846,...
$ arr_delay <dbl> 11, 20, 33, -18, 12, -4, 8, 31, -2, -3, -8, NA, -6, -7,...
$ carrier   <chr> "UA", "UA", "AA", "B6", "UA", "B6", "AA", "AA", "B6", "...
$ flight    <int> 1545, 1714, 1141, 725, 1696, 1806, 301, 707, 49, 71, 79...
$ tailnum   <chr> "N14228", "N24211", "N619AA", "N804JB", "N39463", "N708...
$ origin    <chr> "EWR", "LGA", "JFK", "JFK", "EWR", "JFK", "LGA", "LGA", ...
$ dest      <chr> "IAH", "IAH", "MIA", "BQN", "ORD", "BOS", "ORD", "DFW", ...
$ air_time  <dbl> 227, 227, 160, 183, 150, 44, 138, 257, 149, 158, 140, N...
$ distance  <dbl> 1400, 1416, 1089, 1576, 719, 187, 733, 1389, 1028, 1005...
$ hour      <dbl> 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6...
$ minute    <dbl> 15, 29, 40, 45, 58, 59, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...
$ time_hour <dtm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-01 0...
$ flight_id <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ...
```



Mutating Joins

- Mutating join allows you to combine variables from two tables. It first matches observations by their keys, then copies across variables from one table to the other.
- Add columns from y to x:
 - `inner_join()`: keeps observations that appear in both tables.
 - `left_join()`: keeps all observations in x.
 - `right_join()`: keeps all observations in y.
 - `full_join()`: keeps all observations in x and y.

```
1 flights %>%
2   select(year:day, hour, tailnum, carrier) %>%
3   left_join(airlines, by = "carrier") #by = "key"
```

year <int>	month <int>	day <int>	hour <dbl>	tailnum <chr>	carrier <chr>
2013	1	1	5	N14228	UA
2013	1	1	5	N24211	UA
2013	1	1	5	N619AA	AA
2013	1	1	5	N804JB	B6
2013	1	1	6	N668DN	DL
2013	1	1	5	N39463	UA



Mutating Joins: The Key Columns

- When you join duplicated keys, you get all possible combinations.
- Defining the key columns
 - `by=NULL`: uses all variables that appear in both tables, the so called natural join
 - `by = "x"`: uses only some of the common variables.
 - `by = c("a" = "b")`: match variable a in table x to variable b in table y



Mutating Joins: `base::merge()`

dplyr	merge
<code>inner_join(x, y)</code>	<code>merge(x, y)</code>
<code>left_join(x, y)</code>	<code>merge(x, y, all.x = TRUE)</code>
<code>right_join(x, y)</code>	<code>merge(x, y, all.y = TRUE)</code>
<code>full_join(x, y)</code>	<code>merge(x, y, all.x = TRUE, all.y = TRUE)</code>

- dplyr's joins are considerably faster and don't mess with the order of the rows.



Filtering Joins

- Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables.
- There are two types:
 - `semi_join(x, y)` keeps all observations in x that have a match in y.
 - `anti_join(x, y)` drops all observations in x that have a match in y.



Exercise 1

- Questions:
 - find the top ten most popular destinations
 - match it back to `flights`

```
1 top_dest <- flights %>%
2   count(dest, sort = TRUE) %>%
3   head(10)
4
5 flights %>%
6   semi_join(top_dest)
```

year <int>	month <int>	day <int>	dep_time <int>	sched_dep_time <int>	dep_delay <dbl>
2013	1	1	542	540	2
2013	1	1	554	600	-6
2013	1	1	554	558	-4
2013	1	1	555	600	-5
2013	1	1	557	600	-3
2013	1	1	558	600	-2
2013	1	1	558	600	-2
2013	1	1	558	600	-2
2013	1	1	559	559	0
2013	1	1	600	600	0



Exercise 2

- Question: when connecting flights and planes, what are the flights that don't have a match in planes?

```
1 flights %>%
2   anti_join(planes, by = "tailnum") %>%
3   count(tailnum, sort = TRUE)
```

tailnum	n
<chr>	<int>
NA	2512
N725MQ	575
N722MQ	513
N723MQ	507
N713MQ	483
N735MQ	396
N0EGMQ	371
N534MQ	364
N542MQ	363
N531MQ	349

1-10 of 722 rows

Previous 1 2 3 4 5 6 ... 73 Next



Join problems

- Start by identifying the variables that form the primary key in each table.
- Check that none of the variables in the primary key are missing. If a value is missing then it can't identify an observation!
- Check that your foreign keys match primary keys in another table. The best way to do this is with an `anti_join()`.



Set Operations

- `intersect(x, y)`: return only observations in both x and y.
- `union(x, y)`: return unique observations in x and y.
- `setdiff(x, y)`: return observations in x, but not in y.



Examples

```

1 df1 <- tribble(
2   ~x, ~y,
3   1,  1,
4   2,  1
5 )
6 df2 <- tribble(
7   ~x, ~y,
8   1,  1,
9   1,  2
10 )
11 intersect(df1, df2)

```

x	y
<dbl>	<dbl>
1	1

1 row

```
1 union(df1, df2)
```

x	y
<dbl>	<dbl>
1	1
2	1
1	2

3 rows

```
1 setdiff(df1, df2) #setdiff(df2, df1)
```

