Data Manipulation

Vik Gopal

Though a program be but three lines long,
someday it will have to be maintained.

1. Data Transformation
    - `filter()` Verb
    - `select()` Verb
    - `mutate()` Verb
    - `arrange()` Verb
    - `group_by()` Operator
    - `summarise()` Verb
    - Miscellanous Tasks

2. Tidy Data
    - The Concept of Tidy Data
    - Pivoting to Make Datasets Longer
    - Pivoting to Make Datasets Wider
    - Separating Columns
    - Uniting Columns

3. Relational Data

Introduction

- It is extremely rare that the data you obtain will be in precisely the right format for the analysis that you wish to do.
- Very often, we shall need to do some or all of the following:
  - Create new variables or summaries
  - Re-order the data
  - Rename the variables
  - Select only a subset of rows and/or columns.
- In this section, we shall learn about the `dplyr` package, and the data manipulation verbs that it uses.
- We shall need the following packages for this topic:

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

Flights Dataset `nycflights13`

- The data frame `flights` contains information on 336,776 flights that departed from New York City in 2013.
- More information on the dataset can be obtained using `?flights`.

```r
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
# ...
```

Tibbles

- The output shows that it is not in fact a data frame - it is a `tibble`.

```
class(flights)
```

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

- A tibble is an object that is designed by the creators of the tidyverse collection of packages.
- It is different from data frames in a few ways:
  1. When printing a data frame, it does not print all the rows and all the columns. This makes it better for inspecting a data frame.
  2. It does not do partial matching when extracting columns.
  3. If you request for a column that does not exist, it will generate a warning. In contrast, a data frame object would simply return NULL.

The Key `dplyr` Functions

single table verbs

The following five functions, and combinations of them, will allow you to accomplish the vast majority of data cleaning tasks.

1. The `filter()` function enables you to pick observations (rows) by the values in their columns.
2. The `mutate()` function is used to create new variables.
3. The `select()` function is for you to pick variables (columns) by their names.
4. The `arrange()` function enables you to reorder the rows.
5. The `summarise()` function collapses many values to a smaller set of summary values.

In conjunction with `group_by()`, which splits a dataset by values in a variable, these verbs provide a language for data manipulation.

Applying These Functions

- Each of the above functions is called in an identical manner.
- The first argument is the data frame.
- The subsequent arguments describe what to do with the data frame, using the variable names *without quotes*.
- The output is a new data frame. The original data frame is not modified.
- These operations can be daisy-chained using the pipe operator %>%, which we shall learn about soon.

Filtering Rows with `filter()`

- The output object `jan1` contains all flights that took off on January 1st.
- The `filter()` function does not modify the original data set. It has to be assigned to an object in order to save the output.

```
jan1 <- filter(flights, month==1, day == 1)
jan1
```

```
# A tibble: 842 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
# ...
```

Comparisons and Logical Operators

- In the previous call to `filter()`, there were two arguments apart from the data frame itself:
  1. `month == 1`
  2. `day == 1`
- The filter() function combines these criterion using the AND operator. In other words, it uses `month == 1 & day == 1` to subset the data frame.
- There is no limit to the number of criteria specified. They are simply tagged on as additional arguments.

Comparisons and Logical Operators

additional criteria

- If we wished to use other operations, e.g. the OR operation, then we will have to manually specify those.
- For instance, if we wished to filter all flights that departed in November or December, then we would use

```
filter(flights, month == 11 | month == 12)
```

- If we wished to filter all flights that departed in November/December that were in the air for more than 3 hours,

```
filter(flights, month == 11 | month == 12,
                air_time/60 > 3)
```

- Be careful to use just a single '|' line.

Comparisons and Logical Operators

`%in%` operator

- Grouping by levels of a factor is so common that there is a special operator in R that can be used to simplify the previous command.

```
filter(flights, month %in% c(11, 12),
                air_time/60 > 3)
```

Comparisons and Logical Operators

`na` values

- By default, `filter()` only includes those columns where the condition is TRUE.
- If a cell is missing (`NA`), then that row is dropped. To include those observations, use an expression of the form:

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)                 # NA dropped
filter(df, is.na(x) | x > 1)     # NA kept
```

Select Columns with `select()`

- When we wish to zoom in on a particular set of variables, we can use the `select()` command.
- Prior to the existence of the `select()` function, we could only choose a subset of variables using an index vector consisting of integers or names.
  - This was a huge disadvantage in instances where there were hundreds of columns, but we only needed 2 or 3 in the middle.
- Now, with the `select()` function, we have a huge variety of ways to extricate them.

Examples of `select()` Function

- To select columns by name (no need for quotes)

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

- To select all columns located between the column named year and the one named day (inclusive).

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

- To select all columns *except* those between year and day

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

Functions to Assist Selection

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

- `starts_with("abc")` matches column names that begin with "abc".
- `ends_with("xyz")` matches column names that end with "xyz".
- `contains("ijk")` matches column names that contain "ijk".
- `matches(".a.")` matches columns whose names match the provided regular expression.
- `where(fn_check)` matches columns that return `TRUE` when `fn_check( )` is applied to them.

Functions to Assist Selection

cont'd

- If we wished to select all columns consisting of a time, we could use

```
select(flights, ends_with("time"))
```

- If we wished to select all columns pertaining to departure, we would use

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

- If we wished to select all numeric columns, we would use

```
select(flights, where(is.numeric))
```

1. Data Transformation
    - `filter()` Verb
    - `select()` Verb
    - `mutate()` Verb
    - `arrange()` Verb
    - `group_by()` Operator
    - `summarise()` Verb
    - Miscellanous Tasks
2. Tidy Data
    - The Concept of Tidy Data
    - Pivoting to Make Datasets Longer
    - Pivoting to Make Datasets Wider
    - Separating Columns
    - Uniting Columns
3. Relational Data

Add New Variables with mutate()

- Besides selecting sets of existing columns, we might need to add new columns.
- By default, mutate() adds the new columns to the **end** of the dataset. However, we can change this by specifying an input to the .before or .after arguments.
- First, let us first create a new dataset with fewer columns.

```
flights_sml <- select(flights, year:day,
                      ends_with("delay"), distance,
                      air_time)
```

```
# A tibble: 336,776 x 7
    year month   day dep_delay arr_delay distance air_time
   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
 1  2013     1     1         2        11     1400      227
 2  2013     1     1         4        20     1416      227
 3  2013     1     1         2        33     1089      160
# ...
```

Create New Columns

```r
f2 <- mutate(flights_sml, gain=arr_delay - dep_delay,
                          speed = distance / air_time * 60)
```

```
   year month   day dep_delay arr_delay distance air_time  gain speed
  <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl> <dbl> <dbl>
1  2013     1     1         2        11     1400      227     9  370.
2  2013     1     1         4        20     1416      227    16  374.
```

```r
f2a <- mutate(flights_sml, gain=arr_delay - dep_delay,
                           speed = distance / air_time * 60,
                           .before=dep_delay)
```

```
   year month   day  gain speed dep_delay arr_delay distance air_time
  <int> <int> <int> <dbl> <dbl>     <dbl>     <dbl>    <dbl>    <dbl>
1  2013     1     1     9  370.         2        11     1400      227
2  2013     1     1    16  374.         4        20     1416      227
```

Creation Functions

- Just as for `select()`, the `mutate()` function comes with a set of built-in helper functions.
- These can assist in creating new variables.
- The beauty of these assistants is that they do not just work on a single row; they are aware of the row(s) above and below them!
- You can write your own functions for mutate to use.
  - The only condition is that the function must be vectorised.
- The additional arguments (i.e. the new columns) to the `mutate()` function are expected to be **vectors**.
  - If the new column is shorter than it required, it is recycled.
  - If the value given is `NULL`, that column is dropped. For instance, this drops `dep_delay`:

```
mutate(flights_sml, dep_delay=NULL)
```

Functions to Assist Creation of New Variables

1. Any of the arithmetic operations within R can be used. For instance, to get the `air_time` in hours instead of minutes, we can use

```
mutate(flights_sml, air_time_mins = air_time / 60)
```

2. The `log` or `log10` function. These functions are useful as transformations when the variable is highly skewed.
3. The `lead()` and `lag()` functions from `dplyr`. These allow you to
   - compute running differences `x - lag(x)`
   - find when a value has changed `x != lag(x)`

Lead and Lag Operations in `dplyr`

- Suppose that we have a vector `x` of length n, and we wish to apply `lag()` or `lead()` from `dplyr` to it.
- Let us call the output vector `y`. It will also be of length n.
- Then internally, the `lag()` function sets `y[1]` to be `NA`, and then sets `y[i]` to be `x[i-1]` for i from 2 to n.
- Similarly, the `lead()` function sets `y[n]` to be `NA`, and then sets `y[i]` to be `x[i+1]` for i from 1 to $n-1$.

```
x <- 1:10
lag(x)   # output y-vector
```

```
 [1] NA  1  2  3  4  5  6  7  8  9
```

```
lead(x) # output y-vector
```

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

Lag Operation in `stats`

be careful

- There is a function with the same name `lag()` in the `stats` package in R.
- This package is always automatically loaded.
- If you call `lag()` without loading `dplyr`, it will apply the one from `stats` package instead.
- The `lag()` function from `stats` will lag the **time index** (not the actual values) of a time series:

```r
y <- ts(1:10)
y
```

```
Time Series:
Start = 1
End = 10
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
lag(y)
```

```
Time Series:
Start = 0
End = 9
Frequency = 1
 [1]  1  2  3  4  5  6  7  8  9 10
```

Lag Operation in `stats`

To *lead* the time index, use a negative lag.

```r
lag(y, -1)
```

```
 Time Series:
 Start = 2
 End = 11
 Frequency = 1
  [1]  1  2  3  4  5  6  7  8  9 10
```

Distinguishing Functions With The Same Name

- There will be cases when we need to refer to the `lag()` function from one package when the other is also loaded.
- We can do so by explicitly naming the package that a function comes from.
- Thus we use `stats::lag()` and `dplyr::lag()`

```
y <- 1:10
stats::lag(y)
```

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

```
dplyr::lag(y)
```

```
 [1] NA  1  2  3  4  5  6  7  8  9
```

Functions to Assist Creation of New Variables

cont'd

4. Cumulative and rolling aggregates can be computed using
   - `cumsum`, `cumprod`, `cumin`, `cumax`.

```
cumsum(x)
```

```
 [1]  1  3  6 10 15 21 28 36 45 55
```

```
cummean(x)
```

```
 [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

Functions to Assist Creation of New Variables

cont'd

5. Ranking function `min_rank()`. This assigns rank 1 to the smallest number, rank 2 to the next, and so on.
   - Use `min_rank(desc(x))` to assign rank 1 to the largest number.

```
x <- c(1, 2, 3, NA, 3, 4)
min_rank(x)
```

```
 [1]  1  2  3 NA  3  5
```

- We shall return to see how the `mutate()` function behaves in the presence of groups after we introduce the `group_by()` operator.

Arrange Rows with `arrange()`

- This function changes the order of observations in a data set.
- It takes a set of column names (or complicated expressions of these names) to order the data set by.
- If more than one column or expression is provided, subsequent columns are used to break ties in preceding columns.

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

Arranging Flights

- To arrange a column in descending order, use the `desc` operator.

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

- Missing values will always be placed at the end, whether it is in ascending or descending order.

Heights Dataset

```
heights <- read.csv('../data/heights.csv')
filter(heights, earn > 1e5) %>%
  arrange(desc(earn))
```

- Recall the dataset on heights that we first encountered in topic 02.
- On slide 19 of that topic, we used the `filter()` verb. If we continue with the `arrange()` verb, we get a sorted data frame.

	earn	height	sex	ed	age	race
1	200000	69.66276	male	18	34	white
2	175000	70.58955	male	16	48	white
3	170000	71.01003	male	18	45	white
4	148000	66.74020	male	18	38	white
5	125000	74.34062	male	18	45	white
6	123000	61.42908	female	14	58	white
7	110000	65.96504	male	18	37	white
8	110000	66.31204	female	18	48	other
9	105000	74.58005	male	12	49	white

The `group_by()` Operator

- The `group_by()` operator changes the unit of analysis from the complete dataset to individual groups.
- The grouping operator has no effect on the `select()` verb. The whole column is returned, just like before.
- The `filter()` and `mutate()` verbs work within the scope of a group.
- There are a useful set of **window functions** that work within groups.
- It may be best to work with a dummy data frame to understand these concepts.

Window Functions in dplyr

```r
dummy <- data.frame(grp=c('a','a','b','b','c','c','c'),
                    x = c(1,2,9,2,4,12,15),
                    stringsAsFactors=FALSE)
d2 <- group_by(dummy, grp)
d2
```

```
# A tibble: 7 x 2
# Groups:   grp [3]
  grp       x
  <chr> <dbl>
1 a         1
2 a         2
3 b         9
# ...
```

- The group_by operation does not modify the values of the data frame.
- It simply adds an attribute so that future operations on it are modified.

Window Functions in `dplyr`

`filter()`

```
filter(d2, row_number() > 2)
```

```
# A tibble: 1 x 2
# Groups:   grp [1]
  grp       x
  <chr> <dbl>
1 c        15
```

```
filter(d2, x <= median(x))
```

```
# A tibble: 4 x 2
# Groups:   grp [3]
  grp       x
  <chr> <dbl>
1 a         1
2 b         2
3 c         4
4 c        12
```

Window Functions in `dplyr`

`mutate()`

```
mutate(d2, y = cumsum(x))
```

```
# A tibble: 7 x 3
# Groups:   grp [3]
  grp       x     y
  <chr> <dbl> <dbl>
1 a         1     1
2 a         2     3
3 b         9     9
4 b         2    11
5 c         4     4
6 c        12    16
7 c        15    31
```

- Window functions work differently when your data frame is grouped, as compared to when the data frame is not grouped.
- Window functions take in *n* values and return *n* values.

1. Data Transformation
    - `filter()` Verb
    - `select()` Verb
    - `mutate()` Verb
    - `arrange()` Verb
    - `group_by()` Operator
    - `summarise()` Verb
    - Miscellanous Tasks
2. Tidy Data
    - The Concept of Tidy Data
    - Pivoting to Make Datasets Longer
    - Pivoting to Make Datasets Wider
    - Separating Columns
    - Uniting Columns
3. Relational Data

Grouped Summaries with summarise()

- The summarise() function collapses a data frame into a single row:

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

```
# A tibble: 1 x 1
  mean_dep
     <dbl>
1 12.63907
```

Combining `group_by()` with `summarise()`

- `summarise()` is not useful on its own.
- However, when paired with `group_by()`, it changes the unit of analysis from the complete dataset to individual groups.
- When the `dplyr` verbs are used on a grouped dataset, they will be automatically applied ''by group''.

Mean Delay On Each Day

- The first line below groups the flights by calendar day.
- The second line computes the mean departure delay within each group, thus returning the mean departure delay on each day.

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

```
# A tibble: 365 x 4
   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
 ...
```

.groups argument

- The .groups argument can be used to retain or drop the groupings after the summarisation.
- It is good practice to include it.

Pipe Operator %>%

- Notice what we did on the previous slide.
  1. Introduce a grouping in the data set.
  2. Apply the mean function to the `dep_delay` variable within each group.
- This required us to name the grouped dataset, and then supply that to `summarise()`.
- It would have been neater to *pipe* the output from `group_by()` into `summarise()`.

Pipe Operator %>%

cont'd

- That is precisely what the pipe operator %>% does.
- This is an operator defined by the `magrittr` package. It is loaded automatically by the tidyverse.
- Behind the scenes, it converts
  - `x %>% f(y)` into `f(x, y)`
  - `x %>% f(y) %>% g(z)` into `f(x,y) %>% g(z)`, which is just `g(f(x,y), z)`

Argument Position of Piped Object

- By default, the object on the left is piped to the first argument on the right.
- This can be modified by specifying the object with the period symbol on the right.
- The following code extracts all even rows from the dummy data frame earlier in slide 36.

```
dummy %>% subset(1:nrow(.) %%2 == 0)
```

```
# A tibble: 3 x 2
  grp       x
  <chr> <dbl>
1 a         2
2 b         2
3 c        12
```

Preparing IMDA Data

piped versus non-piped versions

- Recall slide 80 of the topic 02 notes, where we used `dplyr` verbs to prepare the data before plotting.

```
filter(media_data, age == "20-29", year==2015) %>%
  mutate(pct = as.numeric(ever_used)) %>%
  arrange(desc(pct))
```

- Reading from left to right is so much easier than reading a nested set of functions from the inside out!

```
arrange(mutate(filter(media_data,
                      age == "20-29",
                      year==2015),
               pct=as.numeric(ever_used)), desc(pct))
```

Mean Delay On Each Day (Piped)

- We can now rewrite the earlier set of two commands on slide 42

```
mn_dep_day <- group_by(flights, year, month, day) %>%
              summarise(delay=mean(dep_delay, na.rm=TRUE),
                        .groups="drop")
mn_dep_day
```

```
# A tibble: 365 x 4
   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
 ...
```

Remember:

- The output of each `dplyr` verb is a data frame.
- The first input of each `dplyr` verb is a data frame.

How Does Delay Vary With Distance?

- Suppose that we wished to study how delay varies with distance (from New York).

```
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), .groups="drop")
delay <- filter(delay, count > 20, dest != "HNL")
```

- The `n()` function is a `dplyr` window function, that counts the number of observations in each group.
- The last clause in the `filter` verb removes Hawaii from consideration.

How Does Delay Vary With Distance?

with pipes

- With pipes, the goal becomes clearer and the code is easier to read.

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

- There is no longer a need to think about intermediate names!
- This reduces the chances for typographical errors.

Pipes and Readable Code

- The goal of the pipe syntax was to make code more readable, not to make code shorter.
- Refrain from code such as this:

```
flights %>% group_by( ...  ) %>% mutate( ... ) %>% arrange( ... ) %>
```

- Instead, put at most one pipe operator per line:

```
flights %>%
  group_by( ...  ) %>%
  mutate( ... ) %>%
  arrange( ... ) %>%
  select( ... )
```

- That makes your code much more team-friendly.

Useful Summary Functions

- The basic criteria for a function to be used as a summary is that it should return a vector of length 1 (i.e. a scalar).
- Here are some such functions that come with `dplyr`. We shall work with a subset of flights - those that have not been cancelled for this section.

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

- Note that these functions are similar to window functions in the sense that they work within the groups, but they return a single scalar value.

Useful Summary Functions

cont'd

1. Measures of location, such as `mean()` and `median()`.
2. Measures of spread, such as `sd()`, `IQR()` and `mad()`.
3. Measures of rank such as `min()`, `quantile()` and `max()`.

```
# Shortest and longest delays on each day
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(first = min(dep_time),
            last = max(dep_time), .groups="drop")
```

Useful Summary Functions

cont'd

4. Measures of position such as `first(x)`, `nth(x, 2)` and `last(x)`

```
# First and last destinations each day
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(earliest_flight=first(dest, order_by = dep_time),
            latest_flight=last(dest, order_by = dep_time),
            .groups="drop")
```

Useful Summary Functions

cont'd

5. Counts are essential and should almost always be computed, because they reflect the size of each group. The important functions to note are `n()`, and `n_distinct()`.

```
not_cancelled %>%
  group_by(dest) %>%
  summarise(carriers = n_distinct(carrier), .groups="drop") %>%
  arrange(desc(carriers))
```

```
# A tibble: 104 x 2
    dest carriers
   <chr>    <int>
 1   ATL        7
 2   BOS        7
 ...
```

add_tally

`add_tally()` is a dplyr verb that adds a new column containing the counts, **without summarising the data**:

```
not_cancelled %>%
  group_by(dest) %>%
  add_tally()
```

Scoped Variants of dplyr Verbs

- When we need to perform the **same function(s)** to a set of columns, we can use `across( )`.
- The arguments to `across( )` are

  `.cols` The colums to apply to, selected using `select` syntax.

  `.fns` The functions to apply to them.
- `across( )` has to be called **within** `mutate()` or `summarise()`.

```
set.seed(2101)
dummy <- mutate(dummy,
                y = rnorm(7),
                z= runif(7))
mutate(dummy, across(x:y, abs))
```

```
  grp  x          y         z
1   a  1 1.28999261 0.4903550
2   a  2 0.22848498 0.5727893
3   b  9 0.75647036 0.4819479
4   b  2 0.09096765 0.6731821
5   c  4 0.05442133 0.6831719
6   c 12 0.06114213 0.3081298
7   c 15 0.28652271 0.8987088
```

Scoped Variants of dplyr Verbs

cont'd

- To apply several functions at the same time, we use the `.fns` argument.

```
mutate(dummy,
       across(x:y, .fns = list(a1 = abs, a2 = function(x) x^2)))
```

```
  grp  x           y         z x_a1 x_a2       y_a1        y_a2
1   a  1  1.28999261 0.4903550    1    1 1.28999261 1.664080931
2   a  2 -0.22848498 0.5727893    2    4 0.22848498 0.052205385
3   b  9  0.75647036 0.4819479    9   81 0.75647036 0.572247411
4   b  2  0.09096765 0.6731821    2    4 0.09096765 0.008275113
5   c  4  0.05442133 0.6831719    4   16 0.05442133 0.002961681
6   c 12 -0.06114213 0.3081298   12  144 0.06114213 0.003738360
7   c 15 -0.28652271 0.8987088   15  225 0.28652271 0.082095265
```

Rowwise Operations

- How would you perform a row-wise operation?
- Use `rowwise()` to define row-wise groups, and then use `mutate()` as usual, with functions that return scalars!
- Each such function serves as a window function for each row.
- In the code below, the new variable `x2` is a random choice between the letter in column `grp` and the integer in column `x`. *Your output may be different from below.*

```r
dummy %>%
  rowwise() %>%
  mutate(x2 = sample(as.character(c(grp,x)), size=1))
```

```
# A tibble: 7 x 5
# Rowwise:
  grp       x      y     z x2
  <chr> <dbl>  <dbl> <dbl> <chr>
1 a         1  1.29   0.490 1.28999260900253
2 a         2 -0.228  0.573 a
# ...
```

Columnwise Operations

- Sometimes, we wish to perform an operation on a group of columns.
- For instance, we may want to add up the columns `x`, `y` and `z`.
- The `c_across( )` function allows you to work with a row as though it is a vector.

```
dummy %>%
  rowwise() %>%
  mutate(sum = sum(c_across(x:z)))
```

```
# A tibble: 7 x 5
# Rowwise:
  grp       x       y     z   sum
  <chr> <dbl>   <dbl> <dbl> <dbl>
1 a         1  1.29   0.490  2.78
2 a         2 -0.228  0.573  2.34
3 b         9  0.756  0.482 10.2
4 b         2  0.0910 0.673  2.76
5 c         4  0.0544 0.683  4.74
6 c        12 -0.0611 0.308 12.2
7 c        15 -0.287  0.899 15.6
```

1. Data Transformation
   - `filter()` Verb
   - `select()` Verb
   - `mutate()` Verb
   - `arrange()` Verb
   - `group_by()` Operator
   - `summarise()` Verb
   - Miscellanous Tasks

2. Tidy Data
   - The Concept of Tidy Data
   - Pivoting to Make Datasets Longer
   - Pivoting to Make Datasets Wider
   - Separating Columns
   - Uniting Columns

3. Relational Data

Introduction

- In this section, we shall study a consistent way of organising data.
- Getting our data into this format requires some work in the beginning, but the payoff is in the ease with which we will be able to manipulate it afterwards.
- Note that **tidy data**, as defined here, work best with the **tidy tools** that we have been working with and will continue to work with here.
- A different paradigm might be best suited for a different set of tools.

Data Structure

same data, two ways

- Most statistical datasets are rectangular tables made up of *rows* and *columns*.
- However, the same dataset can be presented in different ways.
- We need a more accurate way of describing the information in a table.

	treatmentA	treatmentB
John Smith	-	2
Jane Doe	16	11
Mary Johnson	3	1

	John Smith	Jane Doe	Mary Johnson
treatmentA	-	16	3
treatmentB	2	11	1

Data Semantics

- A dataset is a collection of *values*.
- Values are organised in two ways: Every value belongs to a **variable** and an **observation**.
- A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units.
- An observation contains all values measured on the same unit (like a person, or a day) across attributes.

Variables and Observations

In the example on slide 62, there are three variables:

1. `person`, with three possible values.
2. `treatment` with two possible values.
3. `result` with six possible values, one of which is missing.

There are six observation units. Each one is identified by the combination of person and treatment values.

Variables and Observations

cont'd

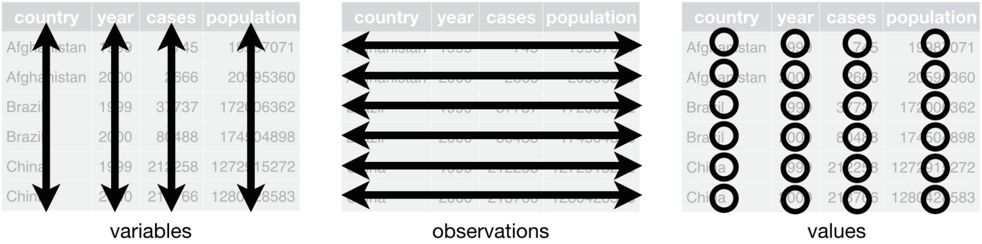
The following is a **tidy** version of the same dataset.

	person	treatment	result
1	John Smith	A	-
2	Jane Doe	A	16
3	Mary Johnson	A	3
4	John Smith	B	2
5	Jane Doe	B	11
6	Mary Johnson	B	1

Tidy Data

Tidy data is a standard way of structuring a data. It requires that

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.



Tidy Data

cont'd

Why Tidy Data?

1. Having a consistent data structure means that we do not have to re-learn the tools to work with data.
2. Placing variables in columns allows the vectorised nature of R functions to take precedence.

Ordering Variables

A good ordering of variables makes it easier to scan the raw values.

- **Fixed variables** refer to those that describe the experimental design. These are typically known in advance. These should come first.
- **Measured variables** are what we actually measure in the study. These should come later.

Untidy Data

Data can be *untidy* in many different ways, but these are the two most common ones:

1. Column headers are values, not actually variable names. In other words, one variable might be spread across multiple columns
   - The dataset is wider than needed to be tidy.
   - Solve this problem using `pivot_longer()`.
2. Multiple variables are stored in one column. This leads to a single observation being scattered across multiple rows.
   - The dataset is longer than needed in order to be tidy.
   - Solve this using `pivot_wider()`.

`pivot_wider` and `pivot_longer` have replaced `spread` and `gather` respectively.

1. Data Transformation
   - `filter()` Verb
   - `select()` Verb
   - `mutate()` Verb
   - `arrange()` Verb
   - `group_by()` Operator
   - `summarise()` Verb
   - Miscellanous Tasks
2. Tidy Data
   - The Concept of Tidy Data
   - Pivoting to Make Datasets Longer
   - Pivoting to Make Datasets Wider
   - Separating Columns
   - Uniting Columns
3. Relational Data

TB Cases

- The following tibble contains TB counts for 3 countries.

```
table4a
```

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

- There are three variables:
  1. Country, with 3 values.
  2. Year, with 2 values.
  3. Number of TB cases, with 6 distinct values.
- One of the variables, Year, is stored in the column names. While this is good for display, it is not tidy.
- Number of TB cases is spread across columns!

How to Pivot Longer

- When we pivot longer, the dataset becomes taller and narrower.
- Some columns will remain; other columns will be rearranged. Most of the time the number of columns will be reduced. (*Not in this small example though.*)
  - The ones that are rearranged are the columns named 1999 and 2000.
  - The column that remains unchanged is `country`.
- The *names* of these rearranged columns from the original (wide) dataset will go into a new column in the new (long) dataset.
  - The name of the new column is specified with the `names_to` argument of `pivot_longer`.
- The cell values in these rearranged colums go into a new single column.
  - The name of this new column is specified with the `values_to` argument of `pivot_longer`.

How to Pivot Longer

cont'd

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

How to Pivot Longer

code

```
table4a %>%
  pivot_longer(!country, names_to="year", values_to="cases")
```

- The colums to pivot are specified using `select` notation.
- Column **names** from the original data go **to** the `year` column in the new data.
- Column **values** from the original data go **to** the `cases` column in the new data.

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

Multiple Variables in a Single Column

- The following tibble contains an observation unit scattered across rows:

```
table2
```

- The observation unit is a country in a year.
- The type column contains two different measurements on each unit.

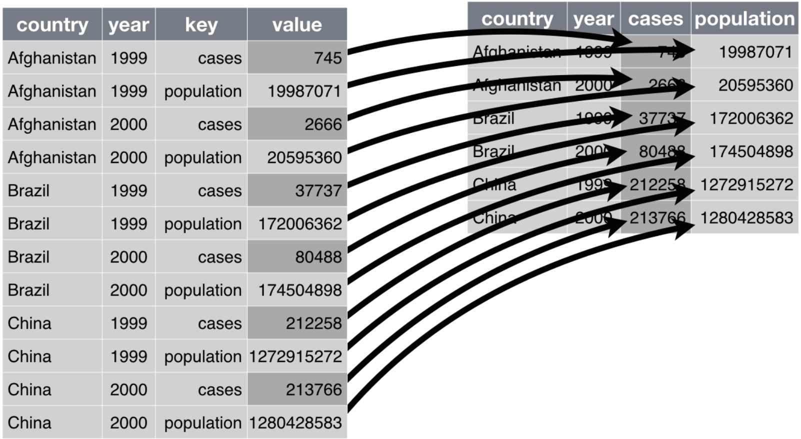
```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <int> <chr>           <int>
 1 Afghanistan  1999 cases             745
 2 Afghanistan  1999 population   19987071
 3 Afghanistan  2000 cases            2666
 4 Afghanistan  2000 population   20595360
 5 Brazil       1999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

How to Pivot Wider

- When we pivot wider, the dataset becomes shorter and wider.
- Some columns will remain; others will be rearranged.
- A subset of columns will uniquely identify an observation.
  - These will be called `id_cols`. In this example, the `country` and `year` columns uniquely identify an observation.
- Most of the time, the number of columns will increase and the number of rows will decrease.
- The *names* of the new columns have to come from the cells of a column in the old dataset.
  - In this example, they come from the `type` column.
- The corresponding *values* in the other rearranged columns will go into new columns in the new dataset.
  - In this example, they come from the `count` column.

How to Pivot Wider

cont'd



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

How to Pivot Wider

code

```
table2 %>%
  pivot_wider(id_cols=country:year, names_from="type",
              values_from="count")
```

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

- Column **names** in the reshaped data come **from** the `type` column in the original data.
- Column **values** in the reshaped data come **from** the `count` column in the original data.

EPL Data

- Recall the EPL data on goal timings and goal scorers (Home or Away) that we imported in the previous topic.
- Here is a reminder of what the data looks like:

```
epl <- readRDS("../data/epl_topic_03.rds") %>%
       as_tibble()
epl
```

```
# A tibble: 760 x 22
   Date       HomeTeam AwayTeam Referee    FGT   SGT   TGT  FOGT ...
   <date>     <chr>    <chr>    <chr>    <int> <int> <int> <int>
 1 2014-08-16 Arsenal  Crystal  J Moss      35    45    90    NA
 2 2014-08-16 Leicest  Everton  M Jones     20    22    45    86
 3 2014-08-16 Man Uni  Swansea  M Dean      28    53    72    NA
 4 2014-08-16 QPR      Hull     C Paws      52    NA    NA    NA
# ...
```

EPL Data

columns

- The columns `FGT:NGT` (9 of them, of integer type) contain the **timings** of goals.
- The columns `FGW:NGW` (9 of them, of factor type) contain the **scorers** of goals.
- These column names contain variables. We need to gather these columns together.
- Note that there is redundant information in these two sets. For instance, `FGT` will always appear with `FGW`, and so on.
- Here is the plan (that uses what we have covered):
  1. convert the columns `FGT:NGW` to character.
  2. pivot longer to collapse these 18 columns into three: goal order, information (T/W), information (timing or scorer).
     - ⋆ At this point, timing and scorer are in the same column.
  3. pivot wider to have separate columns for timing and scorer.
  4. convert the column types to be integer and character once more.

EPL Data

code

```
epl %>% mutate(across(FGT:NGW, as.character)) %>%
  #slice_head(n = 3) %>%
  pivot_longer(FGT:NGW, names_to=c("goal", "info_type"),
               values_to="info",  names_sep=-1) %>%
  pivot_wider(id_cols=Date:goal, names_from=info_type,
              values_from=info)  %>%
  mutate(timing = as.integer(T), scorer = as.character(W),
         T = NULL, W=NULL)
```

```
# A tibble: 6,840 x 7
   Date       HomeTeam  AwayTeam       Referee goal  timing scorer
   <date>     <chr>     <chr>          <chr>   <chr>  <int> <chr>
 1 2014-08-16 Arsenal   Crystal Palace J Moss  FG        35 A
 2 2014-08-16 Arsenal   Crystal Palace J Moss  SG        45 H
 3 2014-08-16 Arsenal   Crystal Palace J Moss  TG        90 H
 ...
```

EPL Data

improving

- When we manipulate data, I find it useful to experiment with a subset of data before trusting the final output.
- The names_sep argument splits the column names, creating two new columns. That is why the names_to needed to be a vector.
- The pivot_xxxx functions are actually much more powerful than we have touched on. We can in fact reshape this data with just this:

```
epl %>%
  pivot_longer(FGT:NGW, names_to=c("goal", ".value"),
                 names_sep=-1)
```

- The pivot vignette and the help page have a lot more examples on the full functionality of these two functions.

1 Data Transformation
- `filter()` Verb
- `select()` Verb
- `mutate()` Verb
- `arrange()` Verb
- `group_by()` Operator
- `summarise()` Verb
- Miscellanous Tasks

2 Tidy Data
- The Concept of Tidy Data
- Pivoting to Make Datasets Longer
- Pivoting to Make Datasets Wider
- Separating Columns
- Uniting Columns

3 Relational Data

Separating A Column

```r
table3
```

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

- The same TB data from slide 75 could have also been stored like this.
- There is little that can be done with the data until the values in the `rate` column are teased apart.

Separating A Column

cont'd

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

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

Separating A Column

cont'd

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

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

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

Separating A Column

cont'd

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

- Notice that the output columns are "character"
- We could ask separate to convert the columns as well.

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

Uniting Columns

- The opposite of separating columns is uniting them.
- `unite()` combines multiple columns into one, using a separator character.

NEA Weather Data

uniting columns

```
pu_2009_12 <- read.csv("../data/nea_200912.csv", header=FALSE,
                       skip=1, na.strings="\x97",
                       stringsAsFactors = FALSE) %>%
              select(1:5) %>%
              rename(station=V1, year=V2, month=V3, day=V4,
                     rainfall=V5) %>%
              unite(date_c, year:day, sep="/") %>%
              mutate(date = as.Date(date_c), date_c=NULL)
```

Introduction

- It is rare that a data analysis involves only a single table.
- Typically, these tables have to be combined to answer the questions we are interested in.
- Multiple tables of data are called **relational data**.
- Relations are always defined between a pair of tables.

New Verbs for Manipulation

Just as `dplyr` introduced us to verbs that worked on single tables, we have a new set of verbs that work with relational data:

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

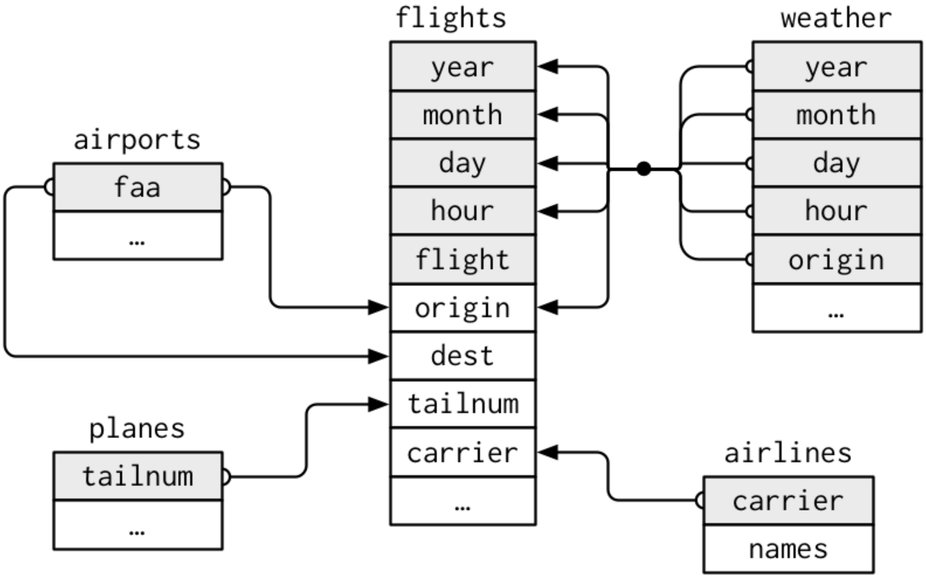
Tables from `nycflights13`

We shall work with 4 tables from this package for this section:

1. `airlines` contains the carrier name and its abbreviated code.
2. `airports` contains information about airports.
3. `planes` contains information about each plan, identified by its `tailnum`.
4. `weather` gives the weather at each airport in New York for each hour.

Take a look at these datasets and their documentation before going on.

Relations Between Tables



Relations Between Tables

cont'd

- `flights` is related to `planes` via a single variable, `tailnum`.
- `flights` connects to `airlines` through the `carrier` variable.
- `flights` connects to `airports` via the `origin` and `dest` variables.
- `flights` connects to `weather` via `origin`, `year`, `month`, `day` and `hour`.

Keys

- The variables that connect each pair of tables are called **keys**.
- A key is a variable (or a set of variables) that uniquely identifies an observation.
- In `planes`, each observation unit (a plane) is uniquely identified by its `tailnum`. Hence `tailnum` is a key.
- In `weather`, each observation unit (an airport in New York at a time) is uniquely identifed by `year`, `month`, `day`, `hour` and `origin`.

Primary Keys and Foreign Keys

- A **primary key** uniquely identifies an observation in its own table. For example, in the `planes` table, `tailnum` is a primary key.
- A **foreign key** uniquely identifies an observation in *another* table. For example, `planes$tailnum` appears in the `flights` table, where it identifies a unique plane.
- A variable can be both a primary key and a foreign key at the same time.
- Sometimes, the best identifier of an observation is still not unique.
- Once you have identified the keys for your tables, it is good to double-check if they are indeed unique.

Relations

- A primary key and the corresponding foreign key form a relation.
- Relations are typically one-to-many.

Verifying Uniqueness of Keys

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

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

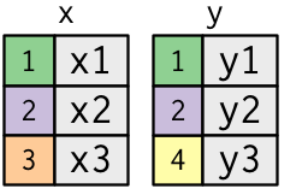
```
weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
```

```
# A tibble: 0 x 6
# ... with 6 variables: year <dbl>, month <dbl>, day <int>,
#   origin <chr>, n <int>
```

Joining Data

- To understand how the different types of mutating joins work, we work with a simple, stripped down data frame.

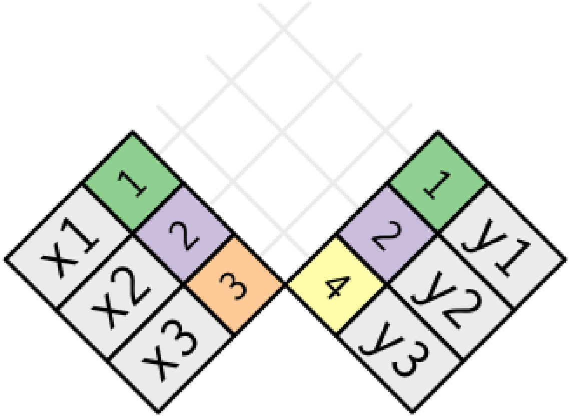
```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```



- The coloured column represents the primary key in each table.
- The gray column represents variables that are not the primary key.
- Notice that there is one value of the primary key that is in `x` but not in `y`, and vice versa.

Defining a Join

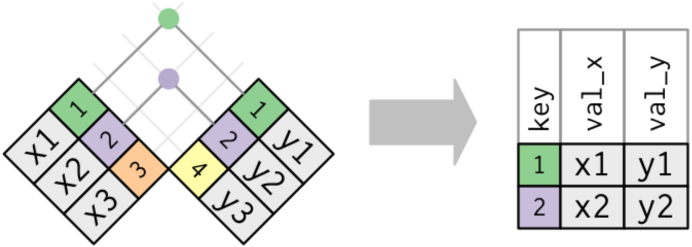
- A join is a way of connecting each row in `x` to zero, one or more rows in `y`.



Defining a Join

cont'd

- In an actual join, we will indicate the matches with dots.

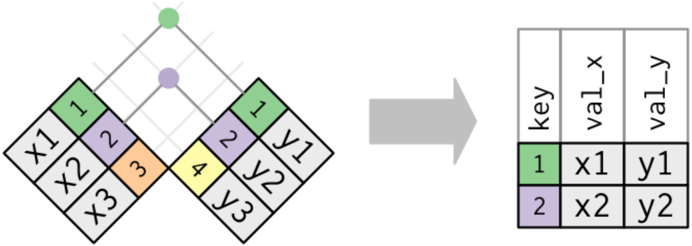


- The different types of joins have to do with which rows to keep if one or the other data frame does not have a match.

Inner Joins

- An **inner join** matches pairs of observations whenever their keys are equal.

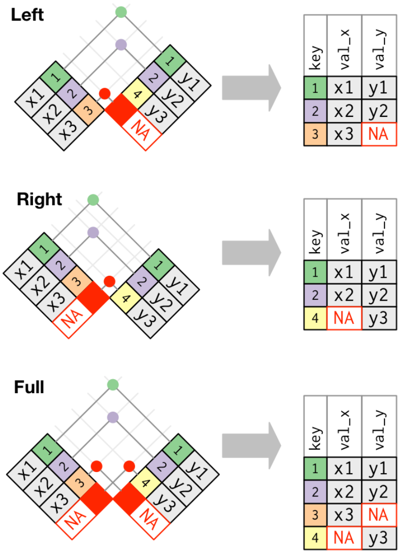
```
x %>% inner_join(y, by="key")
```



- Unmatched rows are dropped.

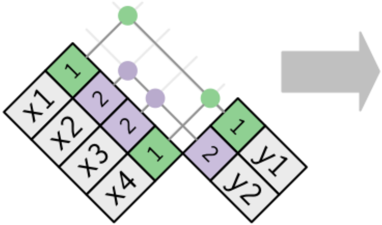
Outer Joins

- 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:
  1. A **left join** keeps all observations in `x`.
  2. A **right join** keeps all observations in `y`.
  3. A **full join** keeps all observations in `x`, and all observations in `y`.
- The most common join is the left join. It allows us to add variables to our existing data frame from another table.



Duplicate Keys

- If one table has duplicate keys, then the matching row is duplicated as well.

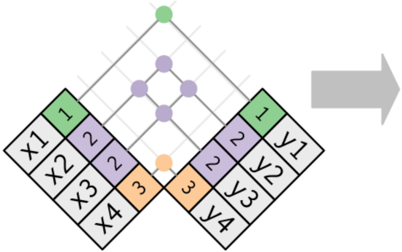


val_x	key	val_y
x1	1	y1
x2	2	y2
x3	2	y2
x4	1	y1

Duplicate Keys

cont'd

- If both tables have duplicate keys, then the cartesian product of keys is created.



key	val_x	val_y
1	x1	y1
2	x2	y2
2	x2	y3
2	x3	y2
2	x3	y3
3	x4	y4

Defining Key Columns

- Let us return to the 4 tables on flight data.

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

- There are several ways in which we can specify the primary/foreign keys.

1. The default is to leave this argument empty. Then the function uses all variables that appear in both tables.

```
flights2 %>% left_join(weather)
```

Defining Key Columns

cont'd

2. We could also use a character vector. Thus we can limit the number of variables used to match the observations.

```
flights2 %>% left_join(planes, by="tailnum")
```

3. A named character vector, of the form `by = c("a" = "b")`. This will match variable `a` in table `x` to variable `b` in table `y`.

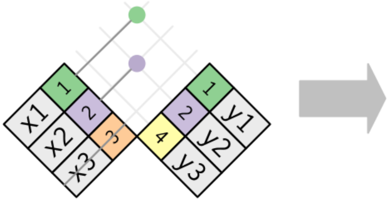
```
flights2 %>% left_join(airports, by=c("dest" = "faa"))
```

Filtering Joins

- Filtering joins match observations in the same way as mutating joins, but affect the observations.
- There are two kinds of filtering joins:
  1. `semi_join(x,y)` **keeps** all observations in `x` that have a match in `y`.
  2. `anti_join(x,y)` **drops** all observations in `x` that have a match in `y`.

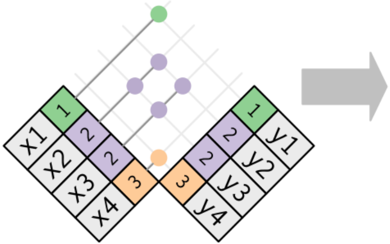
Graphical Semi-Join

Graphically, this is what a semi-join looks like:



key	val_x
1	x1
2	x2

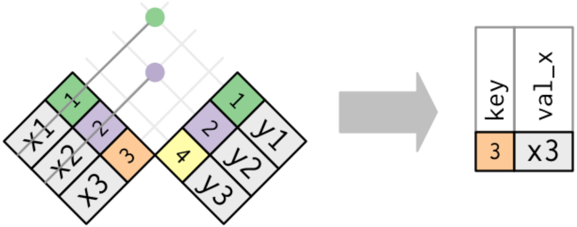
If there are duplicate keys in `x`, then all those rows are kept:



key	val_x
1	x1
2	x2
2	x3
3	x4

Graphical Anti-Join

Graphically, this is what an anti-join looks like:



Flights to Top Destinations

- Suppose we wished to find all flights that flew to the top 10 most popular destinations:

```
top_dest <- flights %>%
  count(dest, sort=TRUE) %>%
  head(10)
flights %>% semi_join(top_dest)
```

Looking for Mismatches

- Anti-joins are useful for looking for mismatches.
- Suppose we are interested in checking if there are flights without planes:

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

A Rough Guide

1. Identify the primary keys in each table.
2. Check that none of the variables in the primary key are missing.
3. Check that foreign keys match primary keys in another table.

Doing the above before starting your analysis could prevent nasty surprises or long debugging hours.

Some Practical Tips for Data Manipulation

- The `tidyverse` provides us with several tools for manipulating data.
- Before you execute a chain of operations, plan the steps.
- Work with a smaller subset of data to ensure things work before executing on the entire dataset.
- Check the output at each stage of the operation, especially with respect to the number of rows / columns you should obtain.
- If you have the luxury of time, perform the reshaping in two different ways. If they don't agree, find out why.
- It is not necessary to memorise the arguments for each function. Get comfortable checking the help pages.
- New functions are added quite regularly. Refer to the vignettes for help and extensive examples on these.
- Keep the `dplyr` cheatsheet close by:
  `https://www.rstudio.org/links/data_transformation_cheat_sheet`