





Data Manipulation

 Files	
 Notes	
 Status	
 Topics	Week 3

Lecture Notes

General Notes:

▼ What to do when processing data

- Keep thinking of what are the other possible associations that we can look into and any other underlying causes that could cause the deviation to the data
- Always plot the scatterplot to see if there are any other associations other than a linear relationship
- Find the correlation between the columns to see if there is any correlation.
 - Can be found using `cor()` function in R
 - However, note that if we are only looking at the correlation coefficient, it could be affected by outliers and it may not be a linear relationship
- There is always more information within a data and it is really up to us to see if we can unravel them
- If there are **clustering**:
 - Can think of either taking the
 - Square Root (Make bigger values smaller)
 - Normalize the graph (Make the axis be of the same length)
 - Make the axis longer
 - Taking the log
 - Note that if there are 0 values then we will have errors because we cannot take log0
 - Way to counter 0's is that we can just add a very small value to it before taking the log

▼ General Tips

- Look into other packages that could be useful
- `Hmisc` - Can look into this package if there is time, can provide good
- Plan before doing a chain of steps
- Work with 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.

Data Transformation

▼ Package

`dplyr` - It makes use of data manipulation verbs that allows us to make the manipulation of data much simpler. It is a Grammar of Data Manipulation

`tidyverse` - Set of packages that work in harmony because they share common data representations and 'API' design

▼ Loads the following packages

`ggplot2`
`tibble`
`tidyr`
`readr`
`purrr`
`dplyr`
`stringr`

▼ Warning

When we load in `tidyverse` it will cause a conflict between

- `dplyr::filter()` and `stats::filter()`
- `dplyr::lag()` and `stats::lag()`

The `dplyr` functions will mask the `stats` functions and if we want to use the `stats` functions, we will need to use `::` in order to specify the package that we need to use

`search()` - We can make use of this to see what are the packages that are loaded into the environment and the ranking that it is loaded. If they are higher and if there are conflicts in the names of the functions, the functions in the higher ranking will mask the lower ranking one

`tibble` - Package that allows for creation of a tibble

Tibble is an object that is similar to a data frame but has some different properties:

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. **(Pretty Printing)**
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.

▼ Usage

- Create new variables or summaries
- Re-order the data
- Rename the variables
- Select only a subset of rows and/or columns

▼ Pipe Operator `%>%`

- Under `magrittr` package. It is automatically loaded by `tidyverse`
- It allows for us to pipe the dataframe result of a verb as the first argument of another verb
- **(Argument Position)** By default, the object on the left is piped to the first argument on the right. But we can specify the position that we supply the object with the period `.` symbol on the right

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

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

▼ Good Practices

- Put at most one pipe operator per line and just use a line break if we are going to move on to the next operator
- This makes the code easier to read, easier to debug and more team-friendly

▼ Behind the scenes

```
# These are the conversions that will occur
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)
```

▼ Benefits:

- Makes reading much easier
- Reduces typing
- Easier to modify our code
- But we will need to plan before we write this out
- This is what makes the `dplyr` package so good

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

▼ Functions (Verbs) Using `dplyr`

▼ Note

- All the verbs are called in a similar manner whereby the first argument is the data frame.
- Subsequent arguments describe what to do with the data frame
- Variable names can be stated without quotations
- Output is a new data frame and original data frame is not modified

▼ `filter(data.frame, ...clauses)`

- Pick observations (rows) by the values in their columns
- For the `clauses` we can pass in any number of logical expressions for the columns of the data frame and it will filter out the rows that satisfies all the conditions
- Note that for `NA` values, they are automatically omitted from the filter as we will only take the values that are true for the clauses

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

# | - Or operator
filter(flights, month == 11 | month == 12)

# left <= x <= right
filter(flights, between(day, 13, 17)) # in this case, the column vector is x, left boundary: 13, right boundary: 17

filter(flights, day >= 13, day <= 17) # using between is a shortcut

# grouping by levels
filter(flights, month %in% c(11, 12),
       air_time/60 > 3) # months will be either 11 or 12

# filtering of NA values
df <- tibble(x = c(1, NA, 3))
filter(df, is.na(x) | x > 1) # if we want to include NA values
```

- `between(variable, lower, upper)` - Between function is useful when we want to have a lower and upper bound for the values that we want to filter. We can just pass this under the filter function
 - This is the same as `filter(var >= lower, var <= upper)`

▼ `slice`

- Subset rows using their positions

`slice_head(data.frame, n=1)` - Selects the first row by default.

We can specify `n =` to specify how many rows we want to splice from the top

`slice_tail(data.frame, n=1)` - Selects the last row by default.

We can specify `n =` to specify how many rows we want to splice from the back

`slice_max(data.frame, order_by, n, with_ties = TRUE)` - Select the `n` rows with the highest values based on the `order_by` variable

We can specify `with_ties` to be `TRUE` or `FALSE` to see whether we want to include ties

`slice_min(data.frame, order_by, n, with_ties = TRUE)` - Select the `n` rows with the lowest values based on the `order_by` variable

We can specify `with_ties` to be `TRUE` or `FALSE` to see whether we want to include ties

`slice_sample(data.frame, n)` - Select `n` random samples from the data frame

```
slice_max(flights, air_time, n=3) %>% View # if there are ties, it will return all of it
slice_min(flights, air_time, n=3) %>% View # if there are ties, it will return all of it
slice_head(flights) # default is n = 1
slice_tail(flights) # default is n = 1
slice_sample(flights, n = 5)
```

▼ `select(data.frame, ...names)`

- Pick variables (columns) by their names

▼ Ways to Select

1. Columns by name
2. Columns located between a starting column and an ending column (inclusive)
3. Select all columns except those stated

```
select(flights, year, month, day)
select(flights, 1:3) # we can use the indices as well
select(flights, year:day)
select(flights, !(year:day)) # all columns except the ones stated
select(flights, carrier:origin, year:day) # we can use the colon when they are consecutive and the rest will all be combined together
```

▼ Helper Functions to Assist in Selection

`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.

`last_col()` - Selects the last variable and it has an offset

`lastcol(2)` - It is the 3rd last column because it offsets for the last column because `lastcol()` is the last, `lastcol(1)` is the second last

```
select(flights, ends_with("time")) # select the columns that ends with a certain text
select(flights, ends_with("time") | starts_with("time"))
select(flights, contains("time"))

select(flights, contains("dep"))
select(flights, where(is.numeric)) # select columns based on the type

select(flights, last_col(2):last_col()) # gives us the last 3 columns
select(flights, 1:last_col(3)) # all except the last 3 columns
```

▼ `mutate(data.frame, ...column = func, .before, .after)`

- Used to create new variables by passing in functions for the variables that we want to create
- By default the columns will be added to the end of the data frame
- `.before` - We will put all the newly created columns before the stated column
- `.after` - We will put all the newly created columns after the stated column
- `...column = func` - The stated columns will be created under the functions that are supplied
 - Dropping columns - `column = NULL` this will drop the stated column

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

▼ Helper Functions to Create new variables

▼ Note:

- We can write our own functions to mutate but they must be vectorised
- The columns that are to be added 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
 - If the creation function creates a `data.frame` just make sure that the output `n` rows and the number of columns will be separated
 - If the new column is longer than the original number of rows, it will return an error

1. Arithmetic Operations between the variables

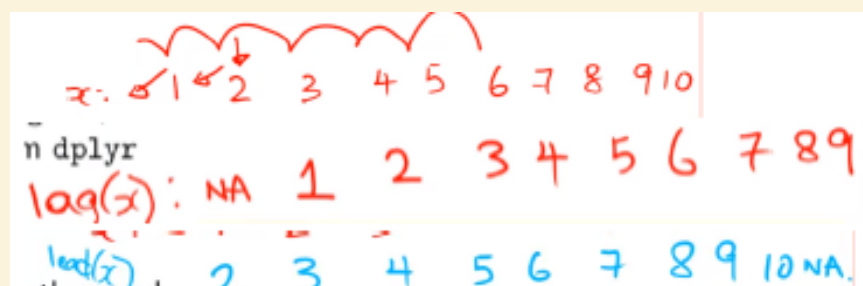
2. `log` or `log10` functions

- Useful as transformations when the variable is highly skewed

3. `lead()` and `lag()` functions from `dplyr`.

They allow for

- Computing running differences `x - lag(x)`
- Find when a value has changed `x != lag(x)`



`lag` takes the values before the current index and if it is the first element, there are no values before and it is `NA`

`lead` takes the values after the current index and if it is the last element, there are no values after and it is `NA`

▼ Note

- There is also `lag()` from the `stats` package whereby it will lag the time index (not the actual values) of a time series
- Therefore, we can use the `::` operation to state the package that we are using instead

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

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

lag(y)

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

4. Cumulative and rolling aggregates

- a. `cumsum`
 - This gives the cumulative sum of all the elements that are before the current index
- b. `cumprod`
 - This gives the cumulative product of all the elements that are before the current index
- c. `cumin`
 - This gives the cumulative min of all the elements that are before the current index
- d. `cumax`
 - This gives the cumulative max of all the elements that are before the current index
- e. `cummean`
 - This gives the cumulative mean of all the elements that are before the current index
- f. `cume_dist`
 - This gives a cumulative distribution function. Proportion of all values less than or equal to the current rank.

5. Ranking Functions

- a. `min_rank()` - Assigns rank 1 to the smallest number, rank 2 to the next, and so on. Note that this will skip to the next rank if there are ties
- b. `dense_rank()` - like `min_rank()`, but with no gaps between ranks

6. Applying a function (or functions) across multiple columns

- a. `across(.cols = everything(), .fns = NULL)` - Makes it easy to apply the same transformation to multiple columns
 - `.cols` - States the columns that we want to operate on
 - `.fns` - States the functions that we want to execute on the columns

```
flights %>%
  select(arr_delay, dep_delay, air_time) %>%
  mutate(across(.fns = ~.x/60)) # the ~ takes the variable x as a placeholder and it will take in each of the columns as an argument each time
```

7. Renaming values/columns

- a. `recode(.x, ..., .missing, .ordered)`

Used for changing values in columns

Helps to recode the values in the vector `.x`. We will supply the `original = new_text` to the `...`

- `...` - We can also pass in a named vector and the name will be the original value and the values in the named vector will be the new text. We will have to use `!!!named_vector` as the notation with the `!!!` when we are passing in the values
- `.missing` - If supplied, any missing values in `.x` will be replaced by this value. Must be either length 1 or the same length as `.x`.
- `.ordered` - If TRUE, `recode_factor()` creates an ordered factor

b. `rename(.data, ...)`

Used for changing the names of columns

- `.data` - A data frame, data frame extension (e.g. a tibble), or a lazy data frame (e.g. from dbplyr or dtplyr). See Methods, below, for more details.
- `...` - Use `new_name = old_name` to rename selected variables.

c. `reorder(x, X)`

Used for reordering of the data

- `x` - Vector that we want the levels to be reordered
 - This is usually a `factor` and it is treated as a categorical variable which we want to order
- `X` - Vector that is of the same length and it will be used to determine the ordering of the levels
 - Note that the default is in ascending order and we can add in `-` before the variable to denote that it is in descending order

8. Changing the type for columns

a. `as.____()`

Pass in the type that we want inside `____`

```
flights %>%
  mutate(air_time = as.numeric(air_time))
```

9. Getting the row number

a. `row_number()` - Gives the row number for the group

▼ `arrange()`

- Reorder the rows
- Changes the order of observation in a data set, if more than one column or expression is provided, subsequent columns are used to break ties in preceding columns
- `desc()` - Can be placed around a variable to indicate that it should be descending. If not the default will be ascending and missing values will be placed at the end no matter what

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

▼ `summarise(data.frame, ..., .groups = NULL)`

- Collapses many values to a smaller set of summary values
 - `...` - Name-value pairs of summary functions. The name will be the name of the variable in the result. We can state the function that we want to use for summarising the values
 - The value can be:
 - A vector of length 1, e.g. `min(x)`, `n()`, or `sum(is.na(y))`.
 - A vector of length n, e.g. `quantile()`.
 - A data frame, to add multiple columns from a single expression.
 - `.groups` - Grouping structure of the result.
 - `"drop_last"` : dropping the last level of grouping. This was the only supported option before version 1.0.0.

- `"drop"` : All levels of grouping are dropped.
- `"keep"` : Same grouping structure as `.data`.
- `"rowwise"` : Each row is its own group.

When `.groups` is not specified, it is chosen based on the number of rows of the results:

- If all the results have 1 row, you get `"drop_last"`.
- If the number of rows varies, you get `"keep"`.

▼ Useful Functions

▼ Measures of location

`mean()`

`median()`

▼ Measures of spread

`sd()`

`IQR()`

`mad()`

▼ Measures of rank

`min()`

`quantile()` - Note that this returns a vector of length 5 which will create 5 rows for the group rather than just 1

If we want to create a column for each of the quantiles

```
# we can supply this under the summarise function instead and it will supply a data frame instead will will return
multiple columns
as.data.frame(t(quantile(long_delay_prop)))
```

`max()`

▼ Measures of position

`first(x)` - Extracts the first element of a vector

`nth(x, n)` - Extracts the nth element of a vector

`last(x)` - Extracts the last element of a vector

▼ Counts

`n()` - Gives the current group size, we can allocate it to a column when we are doing `summarise()` or `mutate()`

`count()` - Creates a column in the data frame that has the count of the groups (if there are groups), else it will be the count of the whole data frame. This also summarises the data and collapses them into the groups

- Note that we can pass in the variables that we want it to count together as and we dont have to group the variables

`n_distinct(x)` - This is a faster and more concise equivalent of `length(unique(x))`. It gives the number of distinct elements in a given vector

`add_tally()` - Adds a new column containing the counts, without summarising the data and does not collapse the data like `count()`

▼ `group_by(data.frame, ..., .add = FALSE)`

- Splits a dataset by values in a variable, all the other verbs will operate on the groups instead

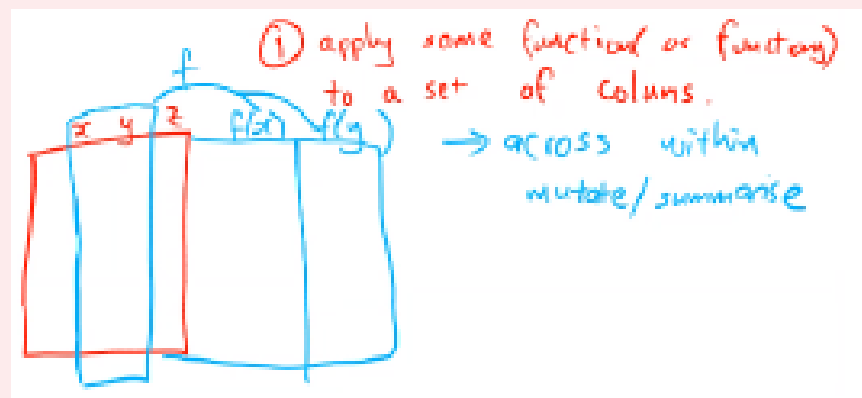
▼ Note

- The grouping operator has no effect on `select()`
- For `arrange()` if we want to use the groups to arrange, we need to set `.by_groups = TRUE` because we can technically just sort by naming the names too

- We just have to supply the columns that we want to be grouped under `...` and it will group the columns together
- `.add` can be set to `TRUE` if we want to add on more groupings on top of the current grouping. Else, the previous groupings will be overwritten by default
- `ungroup()` - Can be used to ungroup the data

▼ Miscellaneous Tasks

▼ Performing the **same function(s)** to a set of columns



`across(.cols, .fns)` - We can make use of across to apply the functions across the set of columns

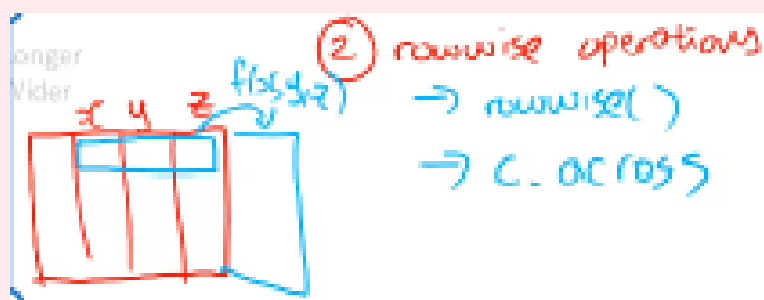
- Has to be called **within** `mutate()` or `summarise()`
- `.cols` - The columns to apply to, selected using `select` syntax
- `.fns` - The functions to apply to them. We can put it as a `name:function` pair

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

# if we want to create new columns
dummy <- mutate(dummy,
  y = rnorm(7),
  z = runif(7))
mutate(dummy, across(x:y, .fns = list(new_names = abs))) # it will add the names as a suffix to the original column

# multiple functions
dummy <- mutate(dummy,
  y = rnorm(7),
  z = runif(7))
mutate(dummy, across(x:y, .fns = list(absolute = abs, square = ~.x^2)))
```

▼ Rowwise Operations



- Make use of `rowwise()` to define row-wise groups, and then we use `mutate()` as usual, with functions that return scalars

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

▼ Column wise Operations

- If we wish to perform an operation on a group of columns
 - E.g. Add the columns of x, y, z

`c_across()` - Allows to work with a row as though it is a vector

- Note that we need to make sure that we are using `rowwise()` first
- We just need to state the columns that we are doing the operation on and we can operate under `mutate`

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

Tidy Data

▼ Package

`tidyr` - It contains the tools that can help us to make our data go from messy to tidy format (This will be loaded when we use `tidyverse`)

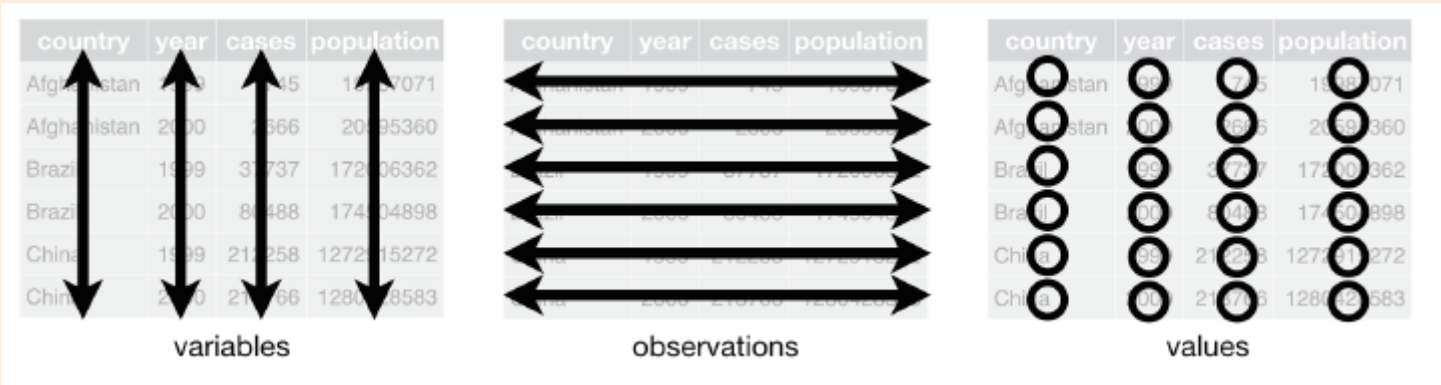
▼ Structure of Tidy Data

- A dataset is a collection of values

▼ Things to consider:

- Observational Unit (**Rows**)
- Variables (**Columns**)

▼ Components



- **Variable** - Contains all the values that measures the same underlying attribute (like height, temperature, duration) across units
- **Observation** - Contains all the values measured on the same unit (like a person, or a day) across attributes
- **Values** - Belongs to a observation variable pair

▼ Ordering Variables

- 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.

▼ Usage

▼ Reasons

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.

▼ Common Issues

▼ Help Page

```
help(package = "tidyr")
```

▼ Too Many Columns

- Column headers are values, not actually variable names
- One variable might be spread across multiple columns

▼ Solution

- We can make use of `pivot_longer()` which is under the `tidyr` library
- The dataset will become taller and narrower, some columns will remain; other columns will be rearranged

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

<ul style="list-style-type: none"> • The columns 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	

Example from lecture

```
pivot_longer(df, cols, names_to, values_to, names_prefix, names_sep, names_transform, values_drop_na)
```

- `df` - Data frame that we want to work on, we can make use of the pipe operator for this
- `cols` - The columns that are values and we want to make them into a single column
- `names_to` - A string specifying the name of the column to create from the data stored in the column names of data.
 - A string specifying the name of the column to create from the data stored in the column names of data.
 - Can be a character vector, creating multiple columns, if `names_sep` or `names_pattern` is provided. In this case, there are two special values you can take advantage of:
 - `NA` will discard that component of the name.
 - `.value` indicates that component of the name defines the name of the column containing the cell values, overriding `values_to`.
- `values_to` - The name of the column that we want to put the values under each of the previous columns
- `names_prefix` - A regular expression used to remove matching text from the start of each variable name.
- `names_sep` - We can specify how we want to split the names of the columns that we have taken. But note that we will have to put in multiple columns under `names_to`
- `names_transform` - A list of column name-function pairs. Use these arguments if you need to change the types of specific columns. For example, `names_transform = list(week = as.integer)` would convert a character variable called week to an integer.
- `values_drop_na` - If `TRUE`, will drop rows that contain only `NA`s in the `value_to` column. This effectively converts explicit missing values to implicit missing values, and should generally be used only when missing values in data were created by its structure.

▼ Too Many Rows

- Multiple variables are stored in one column
- Single observation scattered across multiple rows

▼ Solution

- We can make use of `pivot_wider()` from `tidyr` package
- The dataset becomes shorter and wider, some columns will remain; others will be rearranged

```
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.

Example from lecture

```
pivot_wider(data, id_cols, names_from, names_prefix, names_sep, names_sort, values_from)
```

- `data` - Data frame that we want to work on, we can make use of the pipe operator for this'
- `id_cols` - A set of columns that uniquely identifies each observation. It is also the columns that are not involved in the rearrangement
- `names_from` - The columns in which we will be getting the name of the columns that we are going to create
- `values_from` - The columns in which we will get getting the value of the columns that we are going to create
- `names_prefix` - A regular expression used to remove matching text from the start of each variable name.
- `names_sep` - If `names_from` or `values_from` contains multiple variables, this will be used to join their values together into a single string to use as a column name.
- `names_sort` - Should the column names be sorted? If `FALSE`, the default, column names are ordered by first appearance.

▼ Separating Columns

- We could have data that are in a column and it needs to be split into 2 different columns instead

▼ Solution

- We can make use of `separate()` from `tidyr` package

`separate(data, col, into, sep, remove, convert)` - Pulls apart one column into multiple columns, by splitting wherever a separator character appears

- `data` - Data frame that we want to work on, we can make use of the pipe operator for this
- `col` - The column in which we want to separate
- `into` - Names of new variables to create as character vector. Use `NA` to omit the variable in the output.
- `sep` - If character, `sep` is interpreted as a regular expression. The default value is a regular expression that matches any sequence of non-alphanumeric values.

If numeric, `sep` is interpreted as character positions to split at. Positive values start at 1 at the far-left of the string; negative value start at -1 at the far-right of the string. The length of `sep` should be one less than `into`.

- `remove` - If `TRUE`, remove input column from output data frame
- `convert` - If `TRUE`, will run `type.convert()` with `as.is = TRUE` on new columns. This is useful if the component columns are integer, numeric or logical.
NB: this will cause string "NA"s to be converted to `NA`.

- `extra` - This controls what happens when they are too many pieces after we do the separation:
 - `"warn"` (the default): emit a warning and drop extra values.
 - `"drop"`: drop any extra values without a warning.

- `"merge"`: only splits at most length(into) times
- `fill` - This controls what happens when they are not enough pieces after we do the separation: (Mostly to deal with the NA values that we will have since we do not have enough columns)
 - `"warn"` (the default): emit a warning and fill from the right
 - `"right"`: fill with missing values on the right (fills from the rightmost column)
 - `"left"`: fill with missing values on the left (fills from the leftmost column)

▼ Uniting Columns

- We could have data in 2 separate columns and we want to combine the data into a single column

▼ Solution

- We can make use of `unite()` in the `tidyr` package

`unite(data, col, ... sep, remove, na.rm)` - Convenience function to paste together multiple columns into one.

- `data` - Data frame that we want to work on, we can make use of the pipe operator for this
- `col` - Name of the new column that we want to store the combined data
- `...` - Columns to unite (Uses `tidy-select`)
- `sep` - Separator to use between values
- `remove` - If `TRUE`, remove input column from output data frame
- `na.rm` - If `TRUE`, missing values will be remove prior to uniting each value.

Relational Data

▼ Usage

- Normally our data is not only stored in one table but in multiple tables, therefore, we may need to combine columns together

▼ Keys

- Variables that connect each pair of tables
- Variable that uniquely identifies an observation

▼ Primary Keys

- Uniquely identifies an observation in its own table

▼ Foreign Key

- Uniquely identifies an observation in another table
- Primary key and corresponding foreign key forms a relation (usually a one-to-many) relation

▼ Joins

- Way to connect each row in x to zero, one or more rows in y

▼ 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 the foreign keys match primary keys in another table

▼ Operations

▼ Defining Key Columns

1. Leaving the `by` argument empty. The join functions will use all variables that are common across both of the tables to treat them as the primary keys

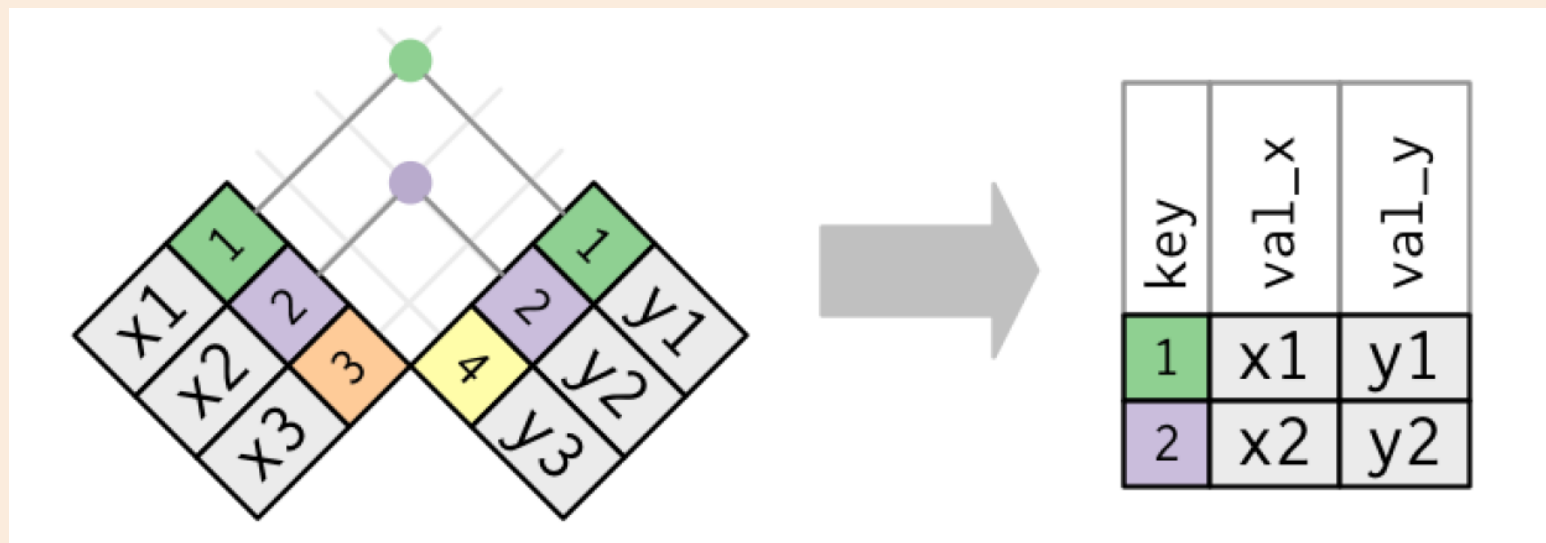
2. Pass in a character vector. This limits the number of variables that is used to match the observations. We can pass a character vector of one of the column names into `by` to specify that it is the key that we are combining by
3. Named character vector. This is when we have the same type of keys in both of the tables but they have different column names. `c("a" = "b")` can be passed into `by` to specify the columns from `x = y`

▼ Mutating Joins

- Adds new variables to a data frame from matching observations in another data frame

▼ Inner Join

- Matches pairs of observations whenever their keys are equal
- Keeps observations that appears in both tables, unmatched rows are dropped



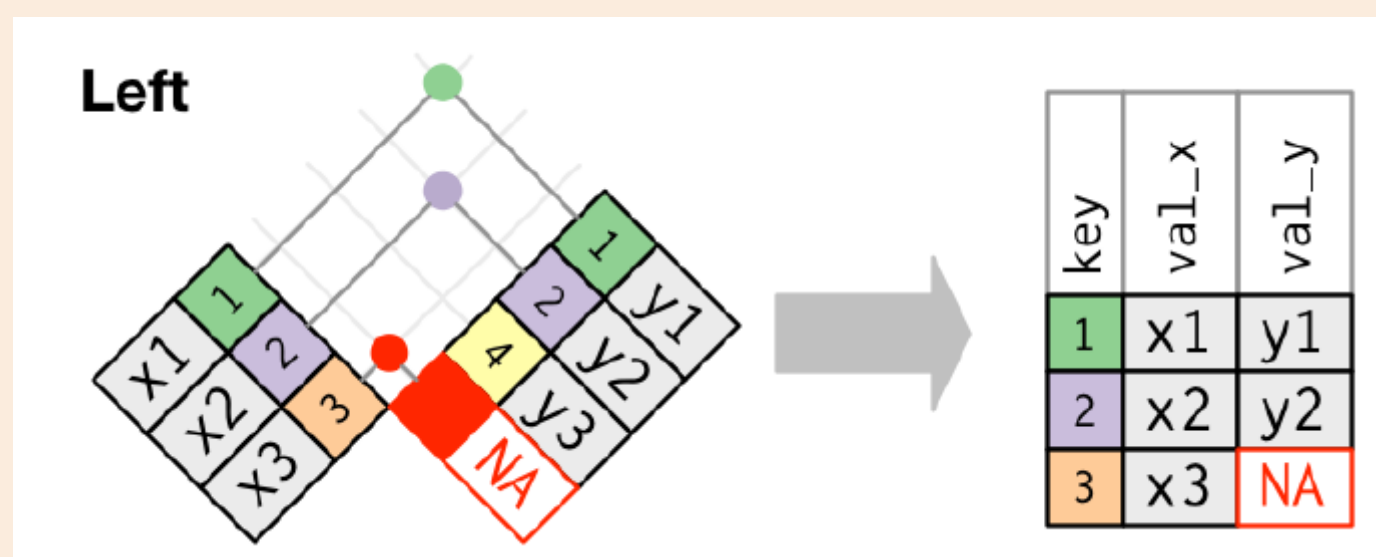
As seen above, we have a match for the keys 1, 2 but 3 and 4 does not have any match. therefore they are dropped

▼ Function

- `inner_join(x, y, by = column)`
 - `x` and `y` are either tibbles or data frames that we want to combine together
 - `by` - Specify which are the columns that is our primary key

▼ Left Join

- Mostly commonly used because it allows for us to add variables to our existing data frame from another table
- Keeps all observations in `x`, unmatched rows are dropped



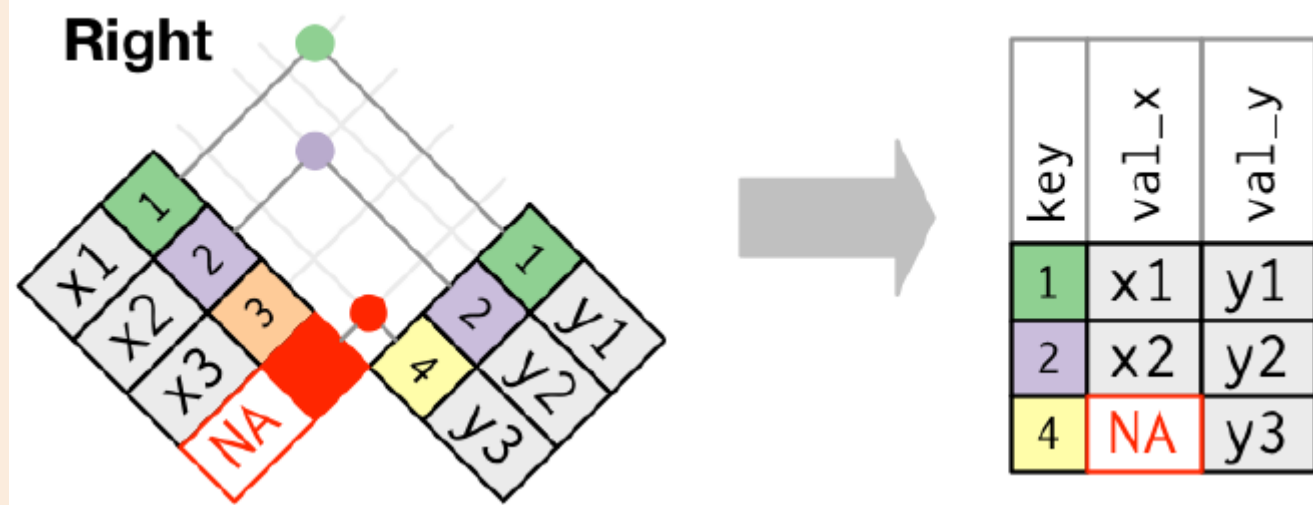
As seen above, we will keep all the keys that are in x and those that are in y will be dropped if they do not match up with the keys in x

▼ Function

- `left_join(x, y, by = column)`
 - `x` and `y` are either tibbles or data frames that we want to combine together
 - `by` - Specify which are the columns that is our primary key

▼ Right Join

- Keeps all observations in `y`, unmatched rows are dropped



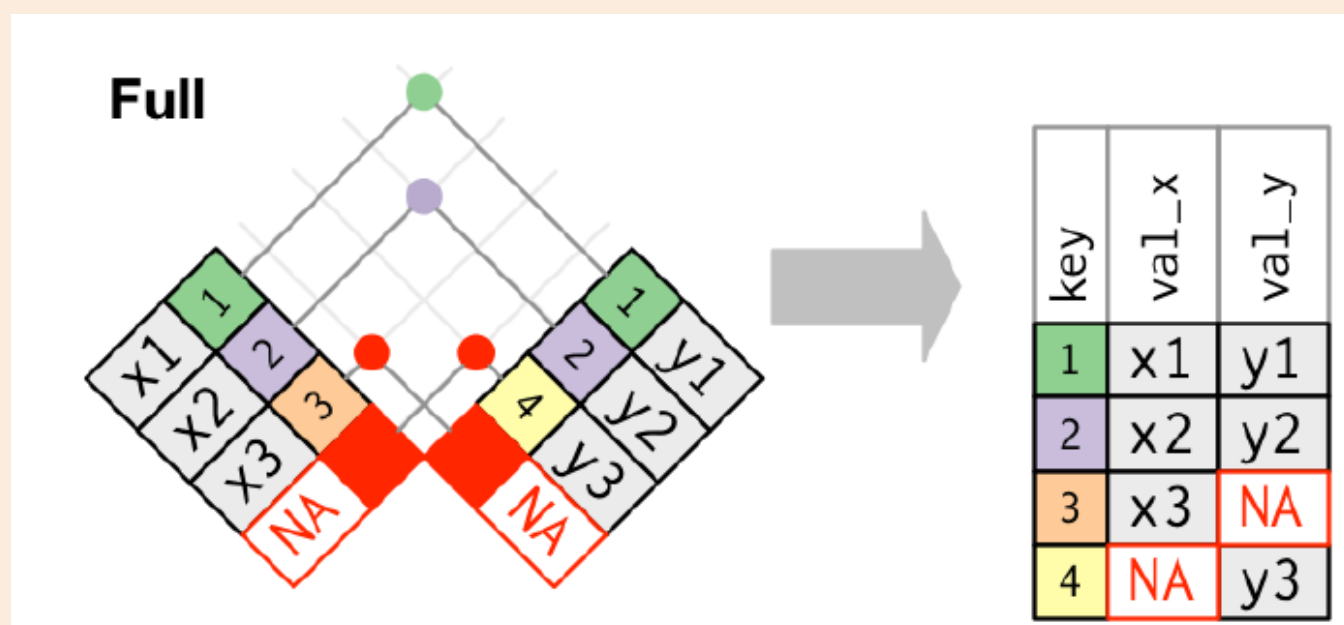
As seen above, we will keep all the keys that are in y and those that are in x will be dropped if they do not match up with the keys in y

▼ Function

- `right_join(x, y, by = column)`
 - `x` and `y` are either tibbles or data frames that we want to combine together
 - `by` - Specify which are the columns that is our primary key

▼ Full Join

- Keeps all observations from `x` and `y`



As seen above, all the observations that are in both x and y are included in the final output

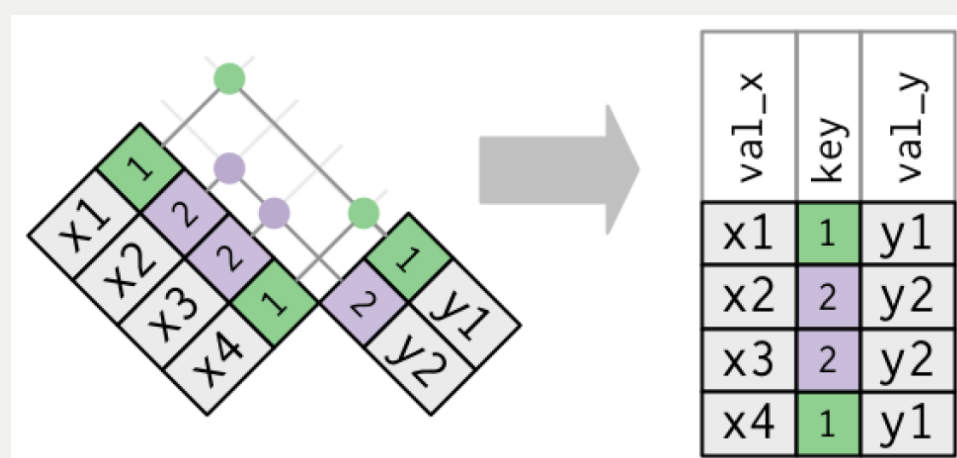
▼ Function

`full_join(x, y, by = column)`

- `x` and `y` are either tibbles or data frames that we want to combine together
- `by` - Specify which are the columns that is our primary key

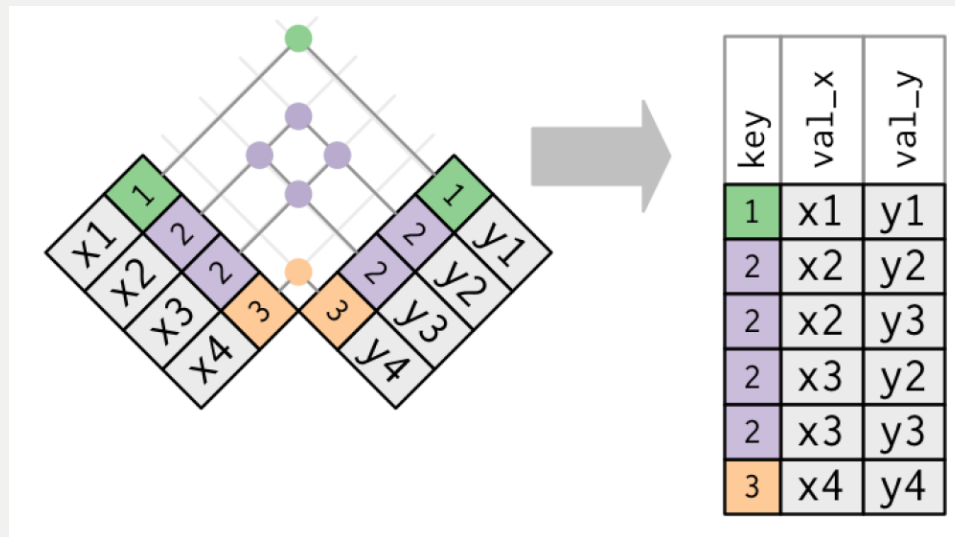
▼ General Points

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



As we can see, since there is a duplicate for the key "2", the join will be duplicated from y

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



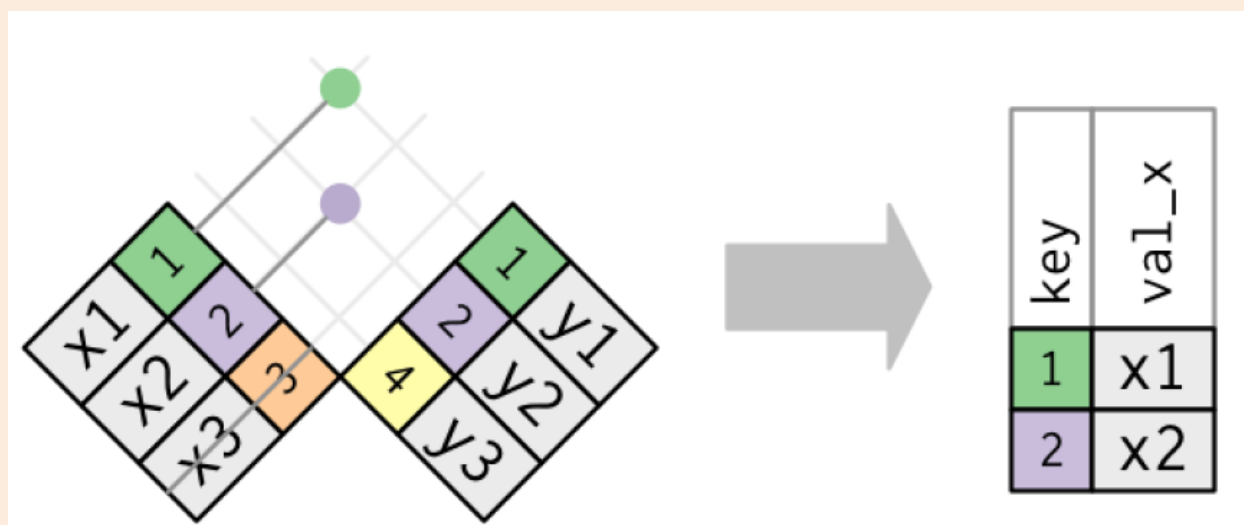
As we can see above, since there are duplicated keys for "2" in both x and y, the resultant table will have all the combination of both of them

▼ Filtering Joins

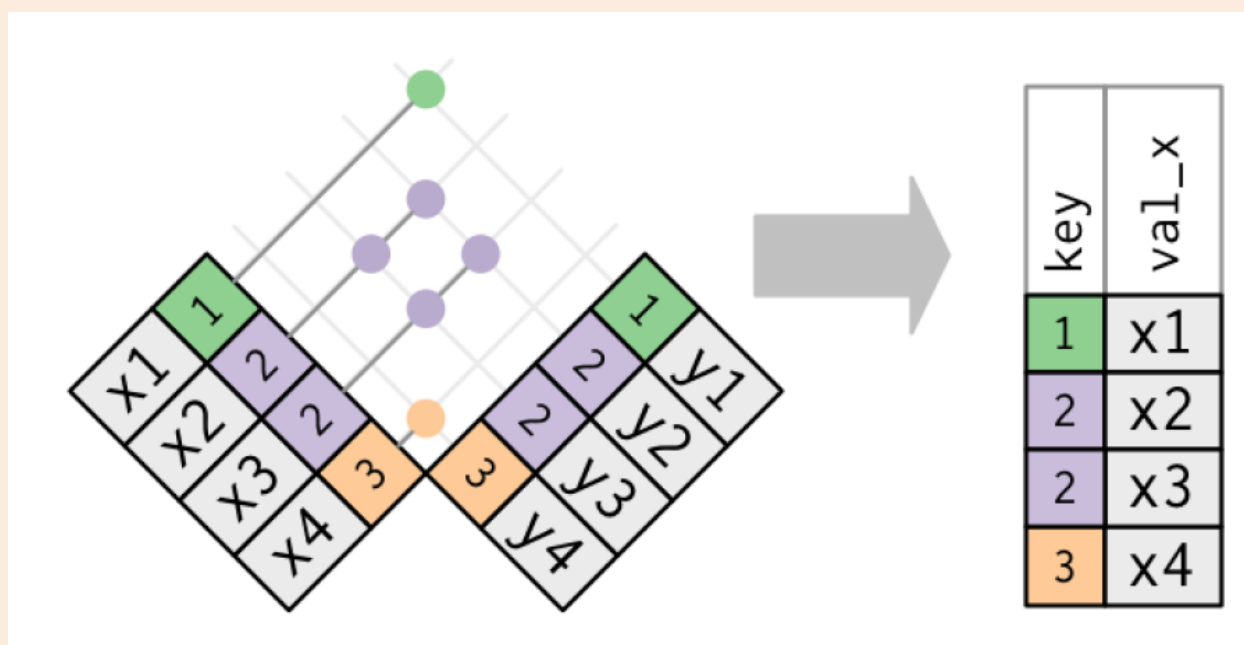
- Filter observations from one data based on whether or not they match an observation in the other table

▼ Semi Join

- Keeps all observations in x that have a match in y



As we can see above, if there is a match in the keeps, we keep the rows in x but we do not make any combination



If there are duplicate keys, the duplicated rows will also be kept so long as there is a match in the other table

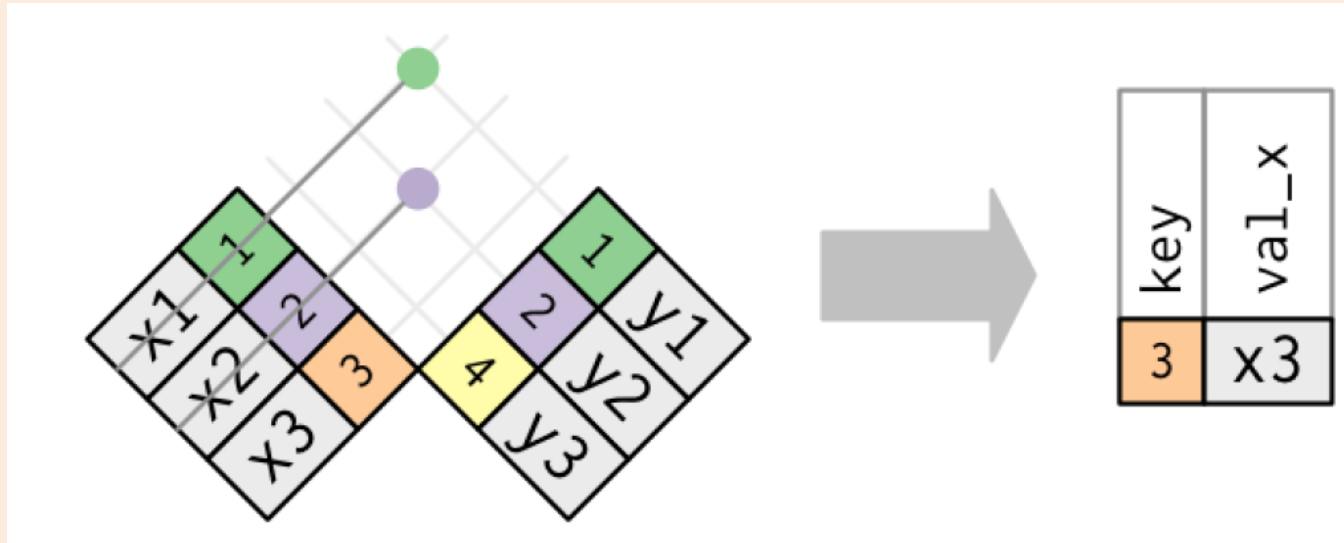
▼ Function

```
semi_join(x, y, by =)
```

- `x` and `y` are either tibbles or data frames that we want to combine together
- `by` - Specify which are the columns that is our primary key

▼ Anti Join

- Drops all observations in x that have a match in y



As we can see, we will only keep the observations that does not have a match with y

▼ Function

```
anti_join(x, y, by =)
```

- `x` and `y` are either tibbles or data frames that we want to combine together
- `by` - Specify which are the columns that is our primary key

▼ Set operations

- Treat observations as if they were set elements