Data Wrangling (Made Easy) in R Workshop

T. J. McKinley (t.mckinley@exeter.ac.uk)

Recap: 'Tidy' data

Specifically, a **tidy** data set is one in which:

- **rows** contain different **observations**;
- **columns** contain different **variables**;
- **cells** contain values.

Remember:

> *"Tidy datasets are all alike but every messy dataset is messy in its own way."—Hadley Wickham*

The `tidyverse`

In the previous session we explored the use of `ggplot2` to produce visualisations of complex data sets.

This utilised the fact that the data sets we had available were **'tidy'** (in the Wickham sense)!

However, it is estimated that data scientists spend around 50-80% of their time cleaning and manipulating data.

In this session we will explore the use of other `tidyverse` packages, such as `dplyr` and `tidyr`, that facilitate effective **data wrangling**.

Cheat sheets

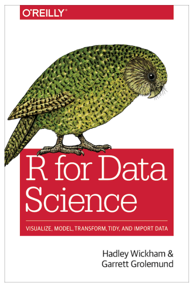
As before, useful cheat sheets can be found at:

https://www.rstudio.com/resources/cheatsheets/

I would highly recommend downloading the appropriate ones (note that they do get updated from time-to-time as the packages are further developed).

Further reading

I would highly recommend Hadley Wickham and Garrett Grolemund's **"R for Data Science"** book:



Can be bought as a hard copy, or a link to a free HTML version is here.

Structure of the workshop

Full (and more comprehensive notes) are provided at:

https://exeter-data-analytics.github.io/AdVis/

You are encouraged to go through these in more detail outside of the workshop.

Today we will discuss the main concepts, and work through some (although not all) of the examples in **Section 2** of the notes.

I would encourage you to work from the HTML here, but a PDF is available as a link in the HTML notes.

RStudio server

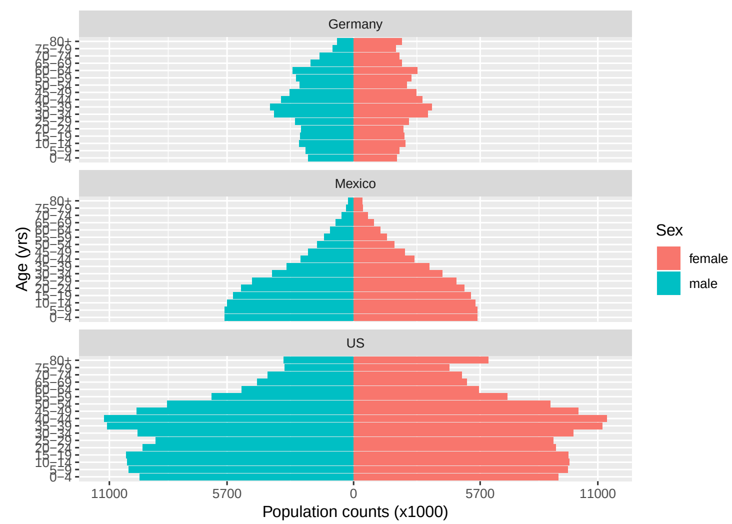
CLES have kindly offered the use of their RStudio server in case anyone needs it:

https://rstudio04.cles.ex.ac.uk

Please note that this server is only for use for this workshop, unless you otherwise have permission to use it .

You will need to log-in using your University log-in details.

What we're aiming for...



Basic operations

We will assume here that we are working with `data.frame`[1] objects[2]. Common data wrangling tasks include:

- sorting;
- **filtering**;
- **selecting** columns;
- transforming columns.

[1] or `tibble`—see later
[2] note that the `purrr` package provides functionality to wrangle different types of object, such as standard `list`s. We won't cover these here, but see Hadley's book, or the tutorials on the `tidyverse` website for more details

Basic operations

These basic operators all have an associated function:

- sorting: `arrange()`;
- **filtering**: `filter()`;
- **selecting** columns: `select()`;
- transforming columns: `mutate()`.

However, each of these operations can be done in base R. So why bother to use these functions at all?

Why bother?

1. These functions are written in a **consistent** way: they all take a `data.frame`/`tibble` objects as their initial argument and return a revised `data.frame`/`tibble` object.
2. Their names are informative. In fact they are **verbs**, corresponding to us **doing something specific** to our data. This makes the code much more readable, as we will see subsequently.
3. They do not require extraneous operators: such as `$` operators to extract columns, or quotations around column names.
4. Functions adhering to these criteria can be developed and expanded to perform all sorts of other operations, such as summarising data over groups.
5. They can be used in **pipes** (see later).

Aside: `tibbles`

`tidyverse` introduces a new object known as a `tibble`. Paraphrased from the `tibble` webpage:

> *A `tibble` is an opinionated `data.frame`; keeping the bits that are effective, and throwing out what is not. Tibbles are **lazy** and **surly**: they do less (i.e. they don't change variable names or types, and don't do partial matching) and complain more (e.g. when a variable does not exist). **This forces you to confront problems earlier, typically leading to cleaner, more expressive code.***[3].

[3] `tibble`s also have an enhanced `print()` method

Aside: `tibbles`

The `readr` package (part of `tidyverse`) introduces a `read_csv()`[4] function to read .csv files in as `tibble` objects e.g.

```
gapminder <- read_csv("gapminder.csv")
gapminder
```

```
## # A tibble: 1,704 x 6
##    country     continent  year lifeExp      pop gdpPercap
##    <chr>       <chr>     <int>   <dbl>    <int>     <dbl>
##  1 Afghanistan Asia       1952    28.8  8425333      779.
##  2 Afghanistan Asia       1957    30.3  9240934      821.
##  3 Afghanistan Asia       1962    32.0 10267083      853.
##  4 Afghanistan Asia       1967    34.0 11537966      836.
##  5 Afghanistan Asia       1972    36.1 13079460      740.
##  6 Afghanistan Asia       1977    38.4 14880372      786.
##  7 Afghanistan Asia       1982    39.9 12881816      978.
##  8 Afghanistan Asia       1987    40.8 13867957      852.
##  9 Afghanistan Asia       1992    41.7 16317921      649.
## 10 Afghanistan Asia       1997    41.8 22227415      635.
## # ... with 1,694 more rows
```

[4]note the underscore (`read_csv`) **not** `read.csv()`

Aside: `tibbles`

Notice:

- `read_csv()` does not convert `characters` into `factors` automatically;
- the `print()` method includes information about the type of each variable (e.g. `integer`, `logical`, `character` etc.), as well as information on the number of rows and columns.

There is also an `as.tibble()` function that will convert standard `data.frame` objects into `tibble`s.

In almost all of the `tidyverse` functions, you can use `data.frame` or `tibble` objects interchangeably.

Example: Superheroes

These data have been extracted from some data scraped by FiveThirtyEight, and available here.

We will assume the complete data consist of three tables:

- `comics`: a table of characters and characteristics;
- `publisher`: a table of characters and who publishes them (Marvel or DC);
- `year_published`: characters against the year they were first published.

Example: Superheroes

Let's have a look at the `comics` data frame:

```
comics
```

```
##                name         EYE       HAIR APPEARANCES
## 1            Batman   Blue Eyes Black Hair        3093
## 2      Wonder Woman   Blue Eyes Black Hair        1231
## 3 Jakeem Williams  Brown Eyes       <NA>          79
## 4        Spider-Man  Hazel Eyes Brown Hair        4043
## 5       Susan Storm   Blue Eyes Blond Hair        1713
## 6     Namor McKenzie  Green Eyes Black Hair        1528
```

Example: Superheroes

To extract a subset of these data, we can use the `filter()` function e.g.

```
filter(comics, HAIR == "Black Hair")
```

```
##              name        EYE       HAIR APPEARANCES
## 1          Batman  Blue Eyes Black Hair        3093
## 2    Wonder Woman  Blue Eyes Black Hair        1231
## 3 Namor McKenzie  Green Eyes Black Hair        1528
```


Example: Superheroes

We can also filter by multiple variables and with negation e.g.

```
filter(comics, HAIR == "Black Hair" & EYE != "Blue Eyes")
```

```
##              name        EYE       HAIR APPEARANCES
## 1 Namor McKenzie  Green Eyes Black Hair        1528
```


Example: Superheroes

To sort these data, we can use the `arrange()` function e.g.

```
arrange(comics, APPEARANCES)
```

```
##               name        EYE       HAIR APPEARANCES
## 1 Jakeem Williams  Brown Eyes       <NA>          79
## 2    Wonder Woman   Blue Eyes Black Hair        1231
## 3  Namor McKenzie  Green Eyes Black Hair        1528
## 4     Susan Storm   Blue Eyes Blond Hair        1713
## 5          Batman   Blue Eyes Black Hair        3093
## 6      Spider-Man  Hazel Eyes Brown Hair        4043
```


Example: Superheroes

We can prefix with a - sign to sort is descending order, and can sort by multiple variables e.g.

```
arrange(comics, HAIR, -APPEARANCES)
```

```
##               name        EYE       HAIR APPEARANCES
## 1          Batman   Blue Eyes Black Hair        3093
## 2  Namor McKenzie  Green Eyes Black Hair        1528
## 3    Wonder Woman   Blue Eyes Black Hair        1231
## 4     Susan Storm   Blue Eyes Blond Hair        1713
## 5      Spider-Man  Hazel Eyes Brown Hair        4043
## 6 Jakeem Williams  Brown Eyes       <NA>          79
```

Example: Superheroes

To extract a subset of ***columns*** of these data, we can use the `select()` function e.g.

```
select(comics, name, HAIR, APPEARANCES)
```

```
##                name       HAIR APPEARANCES
## 1            Batman Black Hair        3093
## 2      Wonder Woman Black Hair        1231
## 3 Jakeem Williams         <NA>          79
## 4        Spider-Man Brown Hair        4043
## 5       Susan Storm Blond Hair        1713
## 6    Namor McKenzie Black Hair        1528
```


Example: Superheroes

A `-` prefix *removes* a column e.g.

```
select(comics, -APPEARANCES)
```

```
##                name        EYE       HAIR
## 1            Batman  Blue Eyes Black Hair
## 2      Wonder Woman  Blue Eyes Black Hair
## 3 Jakeem Williams   Brown Eyes       <NA>
## 4        Spider-Man  Hazel Eyes Brown Hair
## 5       Susan Storm  Blue Eyes Blond Hair
## 6    Namor McKenzie  Green Eyes Black Hair
```


Example: Superheroes

To *transform* or *add* columns, we can use the `mutate()` function[5] e.g.

```
mutate(comics, logApp = log(APPEARANCES))
```

```
##                name        EYE       HAIR APPEARANCES   logApp
## 1            Batman  Blue Eyes Black Hair        3093 8.036897
## 2      Wonder Woman  Blue Eyes Black Hair        1231 7.115582
## 3 Jakeem Williams   Brown Eyes       <NA>          79 4.369448
## 4        Spider-Man  Hazel Eyes Brown Hair        4043 8.304742
## 5       Susan Storm  Blue Eyes Blond Hair        1713 7.446001
## 6    Namor McKenzie  Green Eyes Black Hair        1528 7.331715
```

[5] see also `?transmute`

Pipes

One of the most useful[6] features of `tidyverse` is the ability to use **pipes**.

Piping comes from Unix scripting, and simply allows you to run a chain of commands, such that the results from each command feed into the next one.

`tidyverse` does this using the `%>%` operator[7].

[6] in my opinion
[7] note that the fantastic `magrittr` package does this more generally in R

Pipes

The pipe operator in R works by passing the **result** of the *left-hand side* function into the **first** argument of the *right-hand side* function.

Since all the functions we've seen so far take a `data.frame` as their first argument, and return a `data.frame`, then we can chain these together e.g.

```
comics %>%
    select(name, APPEARANCES) %>%
    arrange(-APPEARANCES) %>%
    mutate(logApp = log(APPEARANCES))
```


Pipes

```
comics %>%
    select(name, APPEARANCES) %>%
    arrange(-APPEARANCES) %>%
    mutate(logApp = log(APPEARANCES))
```

```
##                name APPEARANCES   logApp
## 1       Spider-Man        4043 8.304742
## 2           Batman        3093 8.036897
## 3      Susan Storm        1713 7.446001
## 4    Namor McKenzie        1528 7.331715
## 5      Wonder Woman        1231 7.115582
## 6 Jakeem Williams          79 4.369448
```

Notice:

- No need for *temporary* variables;
- less verbose;
- can be read like prose (easier to understand)

Note: if splitting over multiple lines, the pipe operator must be at the end of the previous line.

Your turn

Have a read through Sections 2.1 and 2.2 of the notes, and have a go at the tasks.

Aside: `*_if` and `*_all`

There are some useful shortcut functions, notably:

- `mutate_if()`;
- `mutate_all()`;
- `summarise_if()`;
- `summarise_all()`.

The `*_if()` functions apply a transformation or summary to a column **if** it adheres to some criteria. The `*_all()` functions apply the transformation or summary to **all** columns[8].

[8] you will see the `summarise()` function shortly...

Aside: `*_if` and `*_all`

As a simple example, let's summarise the data:

```
summary(comics)
```

```
##      name               EYE                HAIR            APPEARANCES
##  Length:6           Length:6           Length:6           Min.   :  79
##  Class :character   Class :character   Class :character   1st Qu.:1305
##  Mode  :character   Mode  :character   Mode  :character   Median :1620
##                                                           Mean   :1948
##                                                           3rd Qu.:2748
##                                                           Max.   :4043
```

Here the `character` columns do not provide a helpful summary. We could temporarily convert each `character` column to a `factor` to produce a better summary.

Aside: `*_if` and `*_all`

Instead let's try:

```
comics %>%
    mutate_if(is.character, as.factor) %>%
    summary()
```

```
##                 name            EYE                HAIR     APPEARANCES
##  Batman           :1   Blue Eyes :3   Black Hair:3   Min.   :  79
##  Jakeem Williams :1   Brown Eyes:1   Blond Hair:1   1st Qu.:1305
##  Namor McKenzie  :1   Green Eyes:1   Brown Hair:1   Median :1620
##  Spider-Man      :1   Hazel Eyes:1   NA's      :1   Mean   :1948
##  Susan Storm     :1                                 3rd Qu.:2748
##  Wonder Woman    :1                                 Max.   :4043
```

This is much neater, and doesn't change the original data frame.

Grouping and summarising

We may also want to produce summaries for different **subsets** of the data.

For example, let's say we want to produce a mean number of appearances for superheroes with different eye colours[9]. We do this using the `group_by()` and `summarise()` functions e.g.

```
comics %>%
    group_by(EYE) %>%
    summarise(
        meanApp = mean(APPEARANCES))
```

```
##          EYE  meanApp
## 1  Blue Eyes 2012.333
## 2 Brown Eyes   79.000
## 3 Green Eyes 1528.000
## 4 Hazel Eyes 4043.000
```

[9] not very interesting I know...

Grouping and summarising

A particularly useful function is `count()`, which tabulates the numbers of observations. This is particularly useful when combined with `group_by()` e.g.

```
comics %>%
    group_by(EYE) %>%
    count()
```

```
##          EYE n
## 1  Blue Eyes 3
## 2 Brown Eyes 1
## 3 Green Eyes 1
## 4 Hazel Eyes 1
```

Your turn

Have a crack at Section 2.3 of the workshop.

Gather and spread

Other really important functions are `gather()` and `spread()`.

These functions are used to **manipulate** `data.frame` objects into different forms.

They are often key to wrangling 'messy' data sets into 'tidy' data sets.

Example: Senate predictions 2018

Let's look at an example from the FiveThirtyEight website.

These data show the predicted probability of each party winning each seat, based on a statistical model fitted on 30th October 2018.

I have filtered and wrangled these data to illustrate these methods, the original data were in fact 'tidy'!

Example: Senate predictions 2018

Let's have a look at the data.

```
head(senate)
```

```
## # A tibble: 6 x 4
##   state      D     O       R
##   <chr>  <dbl> <dbl>   <dbl>
## 1 AZ     0.644    NA  NA
## 2 AZ    NA         0  NA
## 3 AZ    NA        NA   0.356
## 4 CA     1        NA  NA
## 5 CT     0.991    NA  NA
## 6 CT    NA        NA   0.009
```

Key:

- **D**: Democrat
- **O**: Other
- **R**: Republican

These are **not** in 'tidy' format!

Gather

To coerce these into 'tidy' format we can use the `gather()` function, which takes multiple columns, and gathers them into key-value pairs.

It takes the form:

```
gather(data, key, value, ...)
```

where `...` is replaced with the names of the columns we wish to gather together (or the ones we wish to exclude from gathering).

This is best illustrated by an example.

Example: Senate predictions 2018

```
##   state      D  O  R
## 1    AZ 0.6442 NA NA
```

Here we want to collapse the columns labelled D, O and R into a new column called `party` (the **key**), with the predicted proportions in a column called `prop` (the **value**). We do not want `state` to be gathered.

```
senate %>%
    gather(party, prop, -state)
```

```
##   state party   prop
## 1    AZ     D 0.6442
## 2    AZ     D     NA
## 3    AZ     D     NA
## 4    CA     D 1.0000
## 5    CT     D 0.9910
## 6    CT     D     NA
```


Example: Senate predictions 2018

Note that the following are **equivalent**:

```
senate %>%
    gather(party, prop, -state)
```

```
senate %>%
    gather(party, prop, D, O, R)
```

You can chose whichever option is the most sensible.

You can also pipe together to remove the extraneous NAs (and overwrite the original `senate` object):

```
senate <- senate %>%
    gather(party, prop, -state) %>%
    filter(!is.na(prop))
```

```
##   state party   prop
## 1    AZ     D 0.6442
## 2    CA     D 1.0000
## 3    CT     D 0.9910
## 4    DE     D 0.9987
## 5    FL     D 0.7005
## 6    HI     D 1.0000
```


Spread

`spread()` does the opposite of `gather()`: it takes two columns (`key` and `value`) and spreads these into multiple columns e.g.

```
senate
```

```
##   state party   prop
## 1    AZ     D 0.6442
## 2    CA     D 1.0000
## 3    CT     D 0.9910
## 4    DE     D 0.9987
## 5    FL     D 0.7005
## 6    HI     D 1.0000
```

```
senate %>%
    spread(party, prop)
```

```
##   state      D  O      R
## 1    AZ 0.6442  0 0.3558
## 2    CA 1.0000 NA     NA
## 3    CT 0.9910 NA 0.0090
## 4    DE 0.9987 NA 0.0013
## 5    FL 0.7005 NA 0.2995
## 6    HI 1.0000 NA 0.0000
```

Example: Senate predictions 2018

We can now do some more complex analyses. For example, to produce a table of binary predictions based on $p > 0.5$ (using the 'tidy' version of the data):

```
senate %>%
    mutate(outcome = ifelse(prop > 0.5, 1, 0)) %>%
    group_by(party, outcome) %>%
    count() %>%
    spread(party, n)
```

```
##   outcome  D  O  R
## 1       0  9 10 24
## 2       1 23  2  8
```


Unite and separate

Other useful functions are `unite()` and `separate()`, the former takes multiple columns and binds them together, and the latter takes a single column and splits it apart. For example:

```
senate <- senate %>%
    mutate(outcome =
      ifelse(prop > 0.5, 1, 0)) %>%
    group_by(party, outcome) %>%
    count()
senate
```

```
##   party outcome  n
## 1     D       0  9
## 2     D       1 23
## 3     O       0 10
## 4     O       1  2
## 5     R       0 24
## 6     R       1  8
```

```
senate <- senate %>%
    unite(outcome,
          party, outcome, sep = "_")
senate
```

```
##   outcome  n
## 1     D_0  9
## 2     D_1 23
## 3     O_0 10
## 4     O_1  2
## 5     R_0 24
## 6     R_1  8
```


Unite and separate

To reverse this, we can use `separate()`:

```
senate
```

```
##   outcome  n
## 1     D_0  9
## 2     D_1 23
## 3     O_0 10
## 4     O_1  2
## 5     R_0 24
## 6     R_1  8
```

```
senate <- senate %>%
    separate(outcome,
      c("party", "outcome"), sep = "_")
senate
```

```
##   party outcome  n
## 1     D       0  9
## 2     D       1 23
## 3     O       0 10
## 4     O       1  2
## 5     R       0 24
## 6     R       1  8
```


Your turn

Have a crack at Section 2.4 of the workshop.

Joins

A key data analytics skill is to be able to **join** different tables together. This can be done using `*_join()` functions. Key types of join are:

- `inner_join()`
- `left_join()` / `right_join()`
- `full_join()`
- `semi_join()` / `anti_join()`

You can join tables by **cross-referencing** against **key variables**. As an example, let's join two tables relating to information on superheroes...

Joins

```
comics
```

```
##              name        EYE       HAIR APPEARANCES
## 1          Batman  Blue Eyes Black Hair        3093
## 2    Wonder Woman  Blue Eyes Black Hair        1231
## 3 Jakeem Williams Brown Eyes       <NA>          79
## 4      Spider-Man Hazel Eyes Brown Hair        4043
## 5     Susan Storm  Blue Eyes Blond Hair        1713
## 6  Namor McKenzie Green Eyes Black Hair        1528
```

```
year_published
```

```
##           name Year
## 1       Batman 1939
## 2 Wonder Woman 1941
## 3   Spider-Man 1962
## 4  Susan Storm 1961
```

Here we will join the two tables by **name**.

`inner_join()`

The simplest type of join is an **inner join**. This joins two data frames and *retains only those rows in each data frame that can be matched* e.g.

```
inner_join(comics, year_published, by = "name")
```

```
##           name        EYE       HAIR APPEARANCES Year
## 1       Batman  Blue Eyes Black Hair        3093 1939
## 2 Wonder Woman  Blue Eyes Black Hair        1231 1941
## 3   Spider-Man Hazel Eyes Brown Hair        4043 1962
## 4  Susan Storm  Blue Eyes Blond Hair        1713 1961
```


`left_join()`

A **left join** retains *all rows* in the **left** data frame, but *only* rows in the **right** data frame that *can be matched* e.g.

```
left_join(comics, year_published, by = "name")
```

```
##              name        EYE       HAIR APPEARANCES Year
## 1          Batman  Blue Eyes Black Hair        3093 1939
## 2    Wonder Woman  Blue Eyes Black Hair        1231 1941
## 3 Jakeem Williams Brown Eyes       <NA>          79   NA
## 4      Spider-Man Hazel Eyes Brown Hair        4043 1962
## 5     Susan Storm  Blue Eyes Blond Hair        1713 1961
## 6  Namor McKenzie Green Eyes Black Hair        1528   NA
```

Here R replaces elements it can't match with NA.

right_join()

A **right join** retains *all rows* in the **right** data frame, but *only* rows in the **left** data frame that *can be matched* e.g.

```
right_join(comics, year_published, by = "name")
```

```
##           name       EYE       HAIR APPEARANCES Year
## 1       Batman  Blue Eyes Black Hair        3093 1939
## 2 Wonder Woman  Blue Eyes Black Hair        1231 1941
## 3   Spider-Man Hazel Eyes Brown Hair        4043 1962
## 4  Susan Storm  Blue Eyes Blond Hair        1713 1961
```

This is the same as the `inner_join()` in this case. Why?

full_join()

A **full join** retains *all rows* in the **both** data frames e.g.

```
full_join(comics, year_published, by = "name")
```

```
##                name        EYE       HAIR APPEARANCES Year
## 1            Batman  Blue Eyes Black Hair        3093 1939
## 2      Wonder Woman  Blue Eyes Black Hair        1231 1941
## 3 Jakeem Williams  Brown Eyes       <NA>          79   NA
## 4        Spider-Man Hazel Eyes Brown Hair        4043 1962
## 5       Susan Storm  Blue Eyes Blond Hair        1713 1961
## 6    Namor McKenzie Green Eyes Black Hair        1528   NA
```

This is the same as the `left_join()` in this case. Why?

semi_join()

A **semi join** return *all rows* from the **left** data frame where there *are matching* values in the **right** data frame. It returns just columns in the **left** data frame, and does not duplicate rows (i.e. it is a *filtering* join):

```
semi_join(comics, year_published, by = "name")
```

```
##           name       EYE       HAIR APPEARANCES
## 1       Batman  Blue Eyes Black Hair        3093
## 2 Wonder Woman  Blue Eyes Black Hair        1231
## 3   Spider-Man Hazel Eyes Brown Hair        4043
## 4  Susan Storm  Blue Eyes Blond Hair        1713
```


anti_join()

An **anti join** return *all rows* from the **left** data frame where there *are **not** matching* values in the **right** data frame. It returns just columns in the **left** data frame, and does not duplicate rows (i.e. it is a *filtering* join):

```
anti_join(comics, year_published, by = "name")
```

```
##                name        EYE       HAIR APPEARANCES
## 1 Jakeem Williams  Brown Eyes       <NA>          79
## 2    Namor McKenzie Green Eyes Black Hair        1528
```

Joins

We can also join multiple tables together using e.g. **pipes** (or similar)

```
comics
```

```
##                name         EYE       HAIR APPEARANCES
## 1            Batman   Blue Eyes Black Hair        3093
## 2      Wonder Woman   Blue Eyes Black Hair        1231
## 3 Jakeem Williams  Brown Eyes       <NA>          79
## 4        Spider-Man  Hazel Eyes Brown Hair        4043
## 5       Susan Storm   Blue Eyes Blond Hair        1713
## 6    Namor McKenzie  Green Eyes Black Hair        1528
```

```
year_published
```

```
##             name Year
## 1         Batman 1939
## 2 Wonder Woman 1941
## 3     Spider-Man 1962
## 4    Susan Storm 1961
```

```
publisher
```

```
##                name publisher
## 1            Batman        DC
## 2      Wonder Woman        DC
## 3 Jakeem Williams        DC
## 4        Spider-Man    Marvel
## 5       Susan Storm    Marvel
## 6    Namor McKenzie    Marvel
```


Joins

```
comics %>%
    full_join(year_published, by = "name") %>%
    full_join(publisher, by = "name")
```

```
##                name         EYE       HAIR APPEARANCES Year publisher
## 1            Batman   Blue Eyes Black Hair        3093 1939        DC
## 2      Wonder Woman   Blue Eyes Black Hair        1231 1941        DC
## 3 Jakeem Williams  Brown Eyes       <NA>          79   NA        DC
## 4        Spider-Man  Hazel Eyes Brown Hair        4043 1962    Marvel
## 5       Susan Storm   Blue Eyes Blond Hair        1713 1961    Marvel
## 6    Namor McKenzie  Green Eyes Black Hair        1528   NA    Marvel
```


Joins

Note: you can also join by more than one variable e.g.

```
inner_join(x, y, by = c("variable1", "variable2"))
```


Your turn

Have a crack at Section 2.5 of the workshop.

Epilogue

You should now be ready to work through the final (more comprehensive) example in Section 2.6 of the workshop notes.

This brings together various aspects of the last two-days. We take multiple 'messy' data sets, join them together, wrangle them into the correct format and then plot them using `ggplot2`.

Along the way we use a few features of `tidyverse` that we haven't introduced, so I wouldn't expect you to be able to recreate this plot from scratch, but I want you to go through the code and understand what is happening.

Epilogue

Hopefully these workshops have given you a flavour of the power of `tidyverse`.

I for one do most of my data analyses using `tidyverse` now, although remember that it may not be suitable for all types of data / analysis method, so you should view it as one tool in your data science arsenal.

If this has whetted your appetite, I can thoroughly recommend Hadley Wickham and Garrett Grolemund's **"R for Data Science"** book!

Please feel free to e-mail me if you have any further questions.