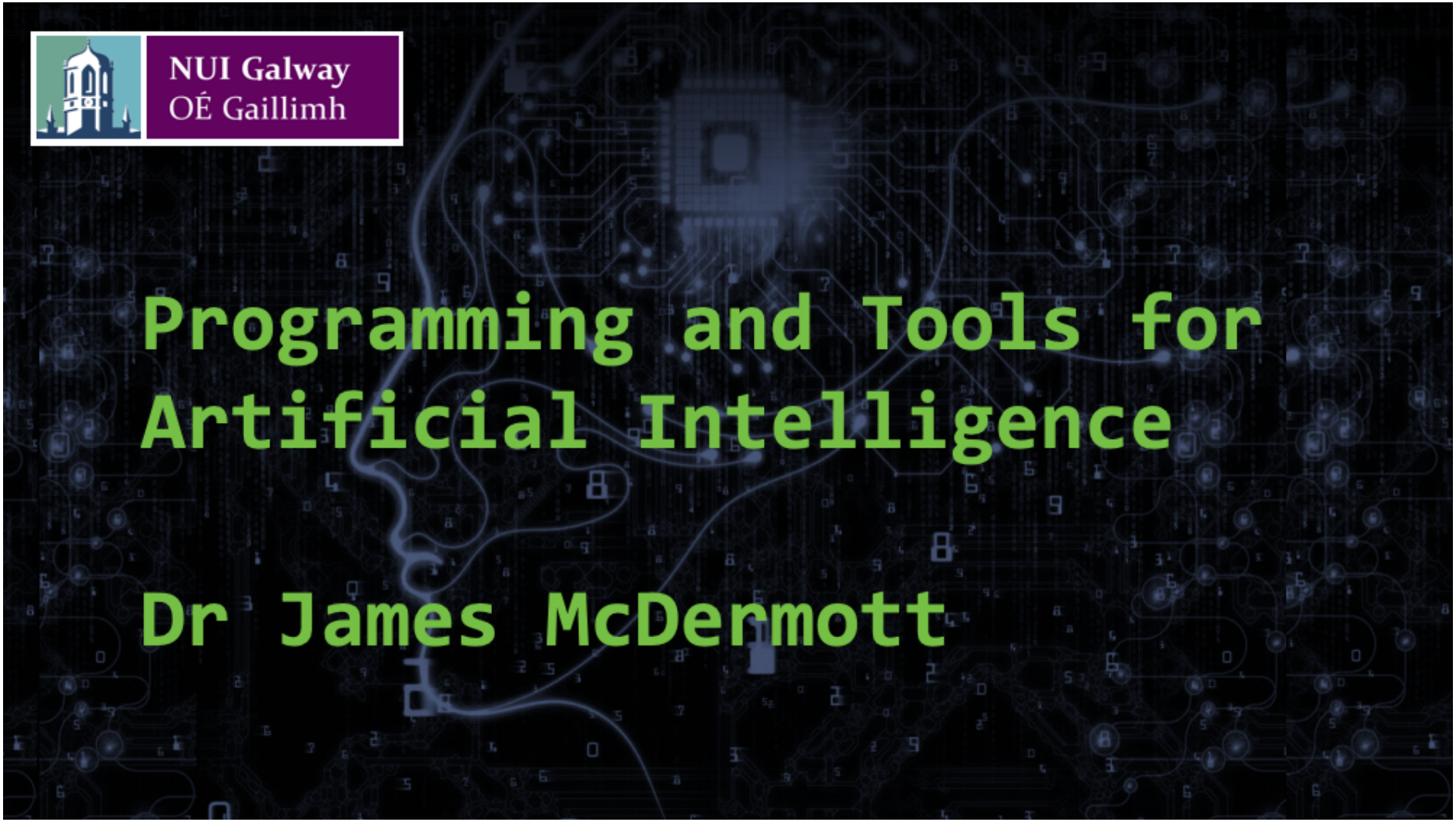
Introduction to R

James McDermott

NUI Galway



Introduction to R

R

R is a language for statistical computing. It is based on an older, commercial language S. Like most of the software studied in this MSc, R is open-source. Research statisticians develop new algorithms in R because it is high-quality open-source. Professional data scientists use it because many statistical algorithms become available in R first, and because the ecosystem, especially tools like RStudio, R Markdown, ggplot, the tidyverse, and Shiny, are excellent.

R Ecosystem

- RStudio: a nice IDE for R
- R Markdown: a text-based format for writing reports with integrated R code, code outputs, and plots
- ggplot: best-in-class plotting
- The tidyverse: a collection of packages for manipulating data according to rational principles of "tidy data"
- Shiny: web-based dashboards

Sources

- Our R lessons are based partly on Hadley Wickham's *R for Data Science* https://r4ds.had.co.nz
- We also draw on Dr Jim Duggan's NUI Galway module CT474
- The materials are written in "R Markdown". I'll distribute both the `.Rmd` source and the `.pdf` slide output.
  https://rmarkdown.rstudio.com/lesson-1.html

Further reading

- Venables, Smith and the R Core Team, *An Introduction to R* https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf
- Wickham, *Advanced R* https://adv-r.hadley.nz
- Kabacoff, *Quick-R* https://www.statmethods.net/

Cheatsheets

- https://rstudio.com/wp-content/uploads/2015/03/ggplot2-cheatsheet.pdf
- https://rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf
- https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
- https://rstudio.com/resources/cheatsheets/

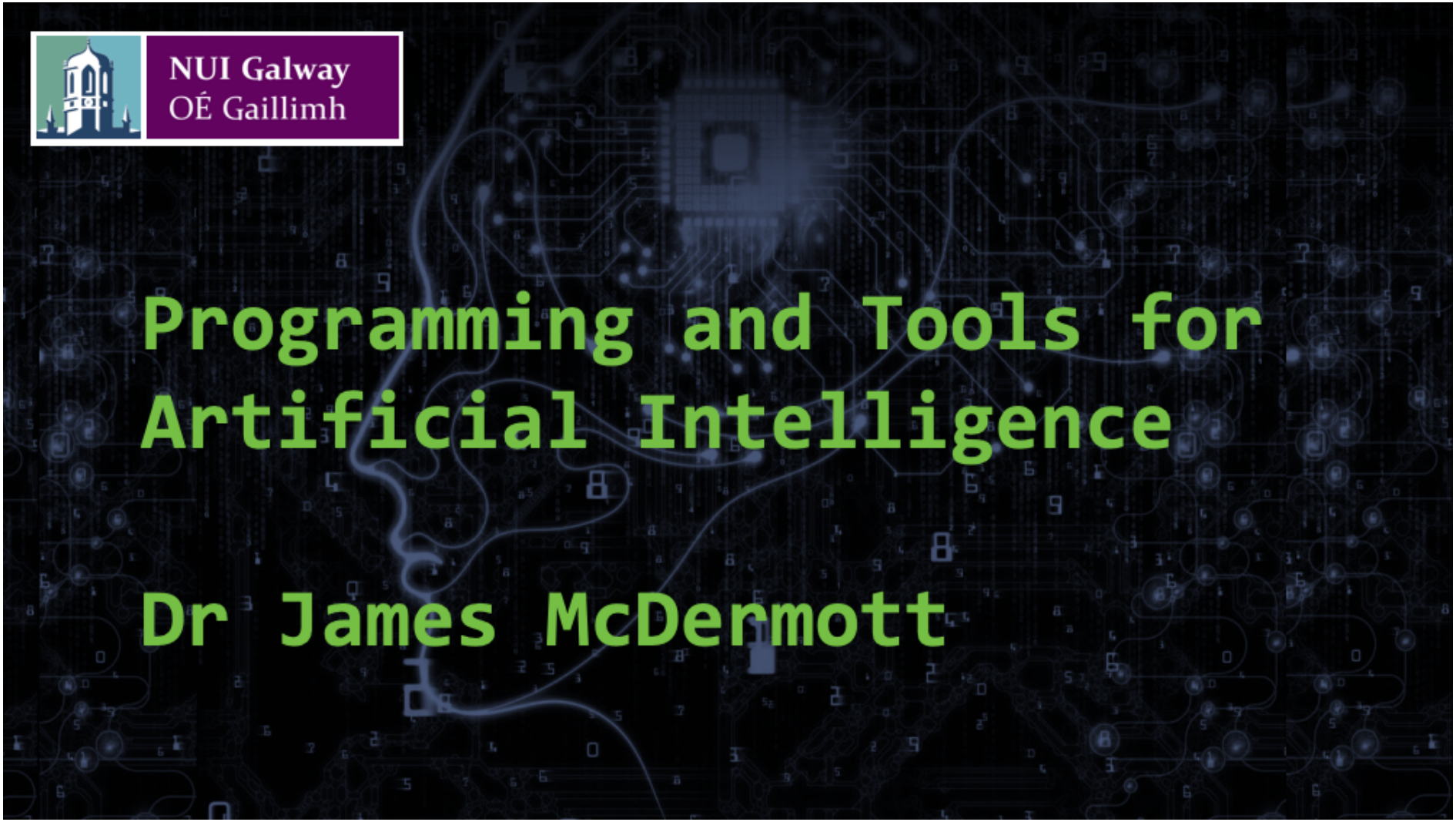
RStudio

"RStudio is an integrated development environment, or IDE, for R programming. Download and install it from http://www.rstudio.com/download." - R4DS

R Basics

James McDermott

NUI Galway


NUI Galway
OÉ Gaillimh
Programming and Tools for Artificial Intelligence
Dr James McDermott

R Basics

Basic R

R **syntax** looks a bit different to Python. Many people think it's not as clean. But the fundamental **concepts** are mostly the same: line-by-line execution, primitive data types, compound data types, assignment, function calls, iteration and conditionals, classes.

Numerical data

```
x <- 5
typeof(x) # vector of length 1 of type double!
```

```
## [1] "double"
```

```
x <- 5L
typeof(x) # using 'L', a vector of length 1 of type int
```

```
## [1] "integer"
```

```r
s <- "abcdefghi" # assignment using "<-"
x <- 5
while (x > 0) { # curly brackets
  if (x %% 2 == 0) { # operators may differ from Python
    print(c(x, substr(s, x, x))) # c() means concatenate
  }
  x = x - 1 # assignment using "="
}
```

```
## [1] "4" "d"
## [1] "2" "b"
```

Special values

- `NA`: not applicable/missing (common in data read from e.g. CSV files)
- `NaN`: "not a number", as in Python
- `-Inf`, `Inf`: infinite values, as in Python

```
c(NA, c(-1, 0, 1) / 0)
```

```
## [1]   NA -Inf  NaN  Inf
```

Special functions

`is.finite()`, `is.infinite()`, `is.na()`, `is.nan()`

What's this about <- and =?

They usually do the same thing – assignment – but there are a few places where = is not allowed. The R community tends to stick to <-.

- https://stackoverflow.com/questions/1741820/what-are-the-differences-between-and-in-r

Vectors: a compound data type

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
```

```
## [1] "one"
```

```
x[[1]]
```

```
## [1] "one"
```

What's the difference between [] and [[]]?

Both exist and sometimes do the same thing, sometimes different!

```
x <- c("one", "two", "three", "four", "five")
x[1] # BTW R indexes from 1, not from 0
```

```
## [1] "one"
```

```
x[[1]]
```

```
## [1] "one"
```

```
x[c(3, 2, 5)] # single [] for selecting a subset of elements
```

```
## [1] "three" "two"   "five"
```

```
# x[[c(3, 2, 5)]] # double [[]] doesn't work here
```

- https://stackoverflow.com/questions/1169456/the-difference-between-bracket-and-double-bracket-for-accessing-the-el
- http://adv-r.had.co.nz/Subsetting.html

Lists with named elements

A `list` with named elements is a bit like a `dict`:

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
a[["b"]] # double square brackets
```

```
## [1] "a string"
```

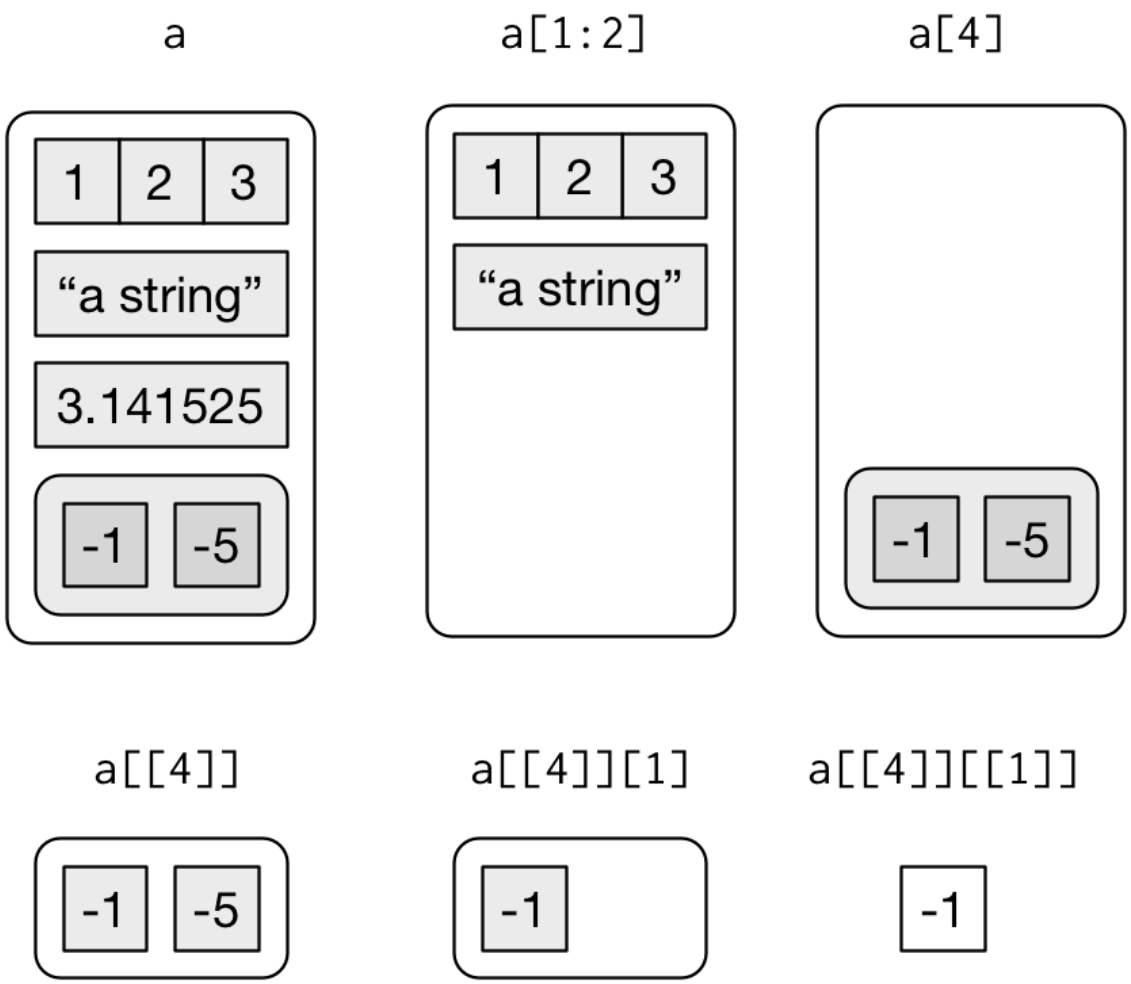
`$` is the same as `[[]]` but only for named elements:

```
a$b # notice, no quotation marks
```

```
## [1] "a string"
```

List subsetting

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```



(From R4DS)

List subsetting

See also Wickham's pepper pot: https://r4ds.had.co.nz/vectors.html (Ctrl-F pepper)

Compound data types

- R `vector` -> Python `list` or Numpy `ndarray`
- R `list` -> Python `tuple`
- R `data.frame` -> Pandas `DataFrame`
- R `tibble` -> Pandas `DataFrame`

Inspecting compound data types

The `str` function gives you the *structure* of an item:

```
str(a)
```

```
## List of 4
##  $ a: int [1:3] 1 2 3
##  $ b: chr "a string"
##  $ c: num 3.14
##  $ d:List of 2
##   ..$ : num -1
##   ..$ : num -5
```

The `typeof` and `length` functions are self-explanatory:

```
typeof(a)
```

```
## [1] "list"
```

```
length(a)
```

```
## [1] 4
```

Ranges, columns, for-in

```
xs = 1:10 # range 1, 2, ... 10
print(xs)
```

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

```
for (x in xs) {
  print(x^2 %% 2)
}
```

```
## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
## [1] 1
## [1] 0
```

But don't use for-loops!

R is vectorised, like Numpy:

```
xs = 1:10
xs = xs ^ 2
ys = xs %% 2
ys
```

```
##  [1] 1 0 1 0 1 0 1 0 1 0
```

Vectorised if-else

```
v1 = 1:5
v2 = v1 ^ 2
v3 = ifelse(v2 %% 2 == 0, "Even", "Odd")
v3
```

```
## [1] "Odd"  "Even" "Odd"  "Even" "Odd"
```

Compare to Numpy `np.where`.

Recycling

```
1:10 * 1:2
```

```
##  [1]  1  4  3  8  5 12  7 16  9 20
```

The shorter vector is recycled. But this is ugly: don't use it!

Functions

`function` is the equivalent of Python `lambda`.

```
normalise <- function(x) {
  # no "return" statement: last value is returned
  (x - min(x)) / (max(x) - min(x))
}
normalise(1:10)
```

```
##  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.55
##  [8] 0.7777778 0.8888889 1.0000000
```

Exercises

1. Write the Factorial function in R, eg `fact(5)` gives 120.
2. Given `x <- "John"`, calculate the length in characters of `x`. Use `nchar()`.
3. Given `xs <- c("John", "Paul", "George", "Ringo")`, calculate the length of each name, using vectorisation (not a for-loop).
4. Calculate whether each name is shorter than 5 characters.
5. Index `xs` to keep just the names shorter than 5 characters.
6. Write a function which unit-norms a vector, ie normalises it so that the vector length equals 1. Eg `unit_norm(c(10, 10, 10, 10))` gives `0.5 0.5 0.5 0.5`.
7. Write a function which standardises a vector, ie gets the z-score, ie maps it to have mean 0 and standard deviation 1. Eg `z_score(c(10, 6, 12, 12))` gives `0.0000000 -1.4142136  0.7071068 0.7071068`.

Solutions

```
fact <- function(n) { # Exercise 1
  if (n <= 1) {
    1 # remember, no return statement!
  } else {
    n * fact(n-1)
  }
}
fact(5)
```

```
## [1] 120
```

```r
x <- "John"
nchar(x) # Exercise 2
```

```
## [1] 4
```

```r
xs <- c("John", "Paul", "George", "Ringo")
nchar(xs) # Exercise 3
```

```
## [1] 4 4 6 5
```

```r
nchar(xs) < 5 # Exercise 4
```

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

```r
xs[nchar(xs) < 5] # Exercise 5
```

```
## [1] "John" "Paul"
```

```r
unit_norm <- function(x) { # Exercise 6
  x / sqrt(sum(x**2))
}
unit_norm(c(10, 10, 10, 10))
```

```
## [1] 0.5 0.5 0.5 0.5
```

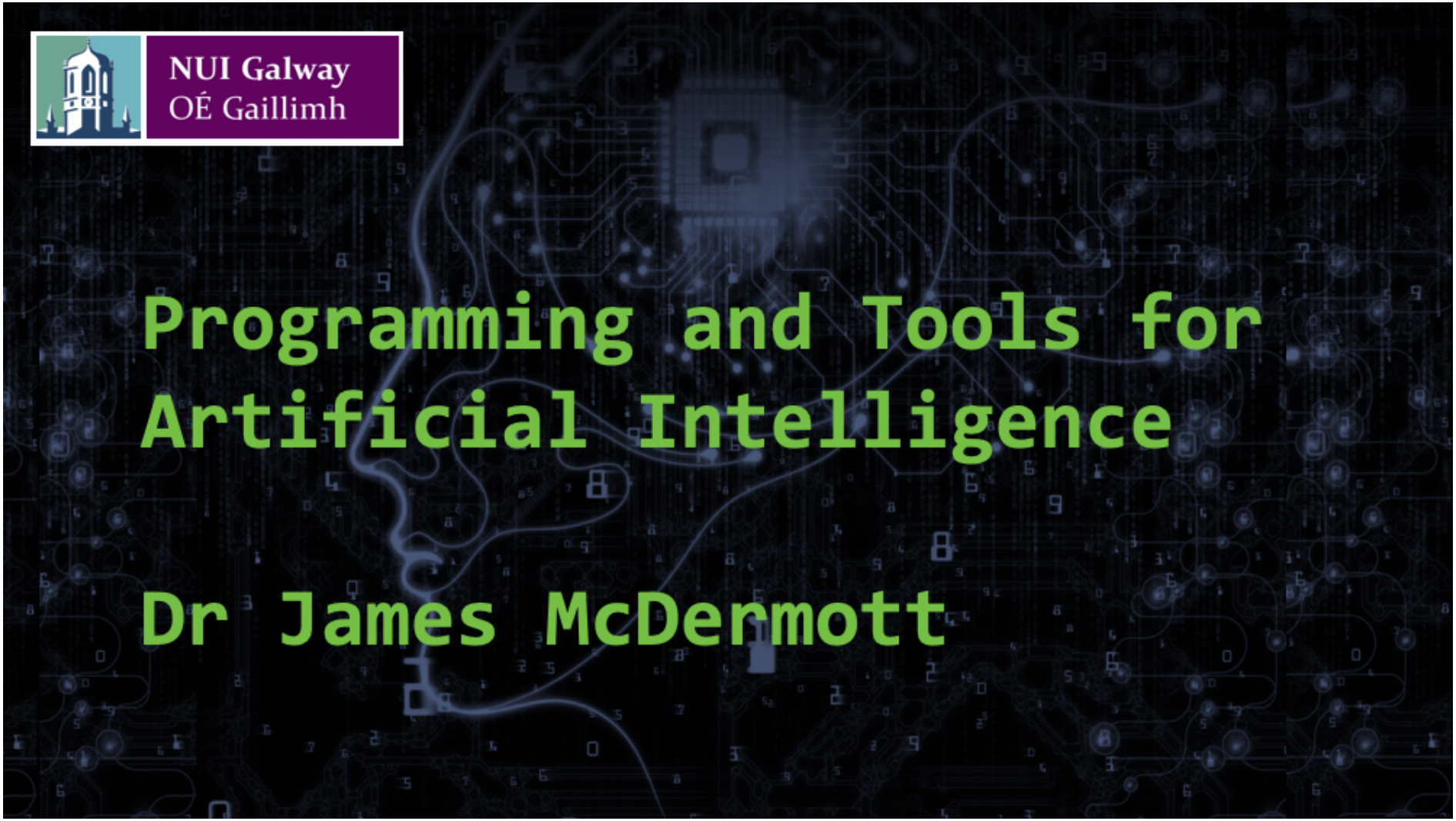
```r
z_score <- function(x) { # Exercise 7
  (x - mean(x)) / sd(x)
}
z_score(c(10, 6, 12, 12))
```

```
## [1]  0.0000000 -1.4142136  0.7071068  0.7071068
```

Tidy Data in R

James McDermott

NUI Galway



Tidy Data in R

Tidy Data

- "Tidy Data" is a set of data-organising principles common in R.
- Proposed by Hadley Wickham, author of important R packages, recently won the 2019 COPSS President's Award ("the Fields medal of statistics").
- "when your data is tidy, each column is a variable, and each row is an observation".
- Much of this and the following R notebooks draw on Wickham's *R for Data Science* https://r4ds.had.co.nz.

Base R and the Tidyverse

- The "Tidyverse" is the "Tidy data universe", a set of packages which try to adhere to tidy data principles.
- Tidyverse packages are all about working on *tibbles* (improved data frames), rather than vectors.
- The tidyverse packages provide new functionality and cleaner interfaces to existing functionality.
- "Base R" is the name used to mean R without the tidyverse packages.

Installing and using packages

To install the Tidyverse packages, copy this code to your R Console in RStudio.

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

To actually use these packages, we have to import as follows:

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

`tibbles` are improved `data.frames`

- In Base R, a `data.frame` is (as in Pandas) a data structure for rectangular data, with column headers – a *table*
- A `tibble` is an improved dataframe for the tidyverse
- Docs: https://r4ds.had.co.nz/tibbles.html

Converting a data.frame to a tibble

```
d <- read.csv("data/mpg_extract.csv")
head(d, n=3) # just look at 3 rows
```

```
##   X..year miles.per.gallon
## 1      70               18
## 2      70               15
## 3      70               18
```

```
dt = as_tibble(d)
head(dt, n=3)
```

```
## # A tibble: 3 x 2
##   X..year miles.per.gallon
##     <dbl>            <dbl>
## 1      70               18
## 2      70               15
## 3      70               18
```

Why are tibbles named tibbles?

Making a tibble by hand: use `tribble`

```
d = tribble(
  ~x, ~y, ~z, # here ~ indicates that these are formulas
  #--/--/----
  "a", 2, 3.6,
  "b", 1, 8.5
)
```

Reading in a tibble

`read_csv` (not `read.csv`) is part of the Tidyverse `readr` package.

```
d <- read_csv("data/prices.csv")
```

```
## Parsed with column specification:
## cols(
##   Date = col_character(),
##   AAPL = col_double(),
##   MSFT = col_double(),
##   YHOO = col_double(),
##   NFLX = col_double(),
##   BMW = col_double(),
##   F = col_double(),
##   FB = col_double(),
##   GOOG = col_double(),
##   GM = col_double(),
##   GE = col_double(),
##   LULU = col_double(),
```

Was everything read in ok?

```
head(d) # take a quick look
```

```
## # A tibble: 6 x 14
##   Date      AAPL  MSFT  YHOO  NFLX   BMW     F    FB  GOOG
##   <chr>    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <
## 1 27/03/~  76.8  39.4  35.6  52.0  91.0  15.2  61.0  557.
## 2 28/03/~  76.7  40.3  35.9  51.3  92.0  15.4  60.0  558.
## 3 29/03/~  76.7  40.3  35.9  51.3  92.0  15.4  60.0  558.
## 4 30/03/~  76.7  40.3  35.9  51.3  92.0  15.4  60.0  558.
## 5 31/03/~  76.7  41.0  35.9  50.3  91.5  15.6  60.2  555.
## 6 01/04/~  77.4  41.4  36.5  52.1  92.2  16.3  62.6  566.
## # ... with 2 more variables: C <dbl>, JPM <dbl>
```

Was everything read in ok?

```
glimpse(d) # another way of taking a quick look
```

```
## Observations: 1,010
## Variables: 14
## $ Date <chr> "27/03/2014", "28/03/2014", "29/03/2014", "30/
## $ AAPL <dbl> 76.7800, 76.6943, 76.6943, 76.6943, 76.6771, 7
## $ MSFT <dbl> 39.36, 40.30, 40.30, 40.30, 40.99, 41.42, 41.3
## $ YHOO <dbl> 35.5900, 35.9000, 35.9000, 35.9000, 35.9000, 3
## $ NFLX <dbl> 52.026, 51.267, 51.267, 51.267, 50.290, 52.099
## $ BMW  <dbl> 91.05, 92.02, 92.02, 92.02, 91.49, 92.25, 92.7
## $ F    <dbl> 15.25, 15.45, 15.45, 15.45, 15.60, 16.32, 16.4
## $ FB   <dbl> 60.970, 60.010, 60.010, 60.010, 60.240, 62.620
## $ GOOG <dbl> 556.931, 558.456, 558.456, 558.456, 555.445, 5
## $ GM   <dbl> 34.51, 34.73, 34.73, 34.73, 34.42, 34.34, 34.8
## $ GE   <dbl> 25.81, 25.88, 25.88, 25.88, 25.89, 25.87, 26.0
## $ LULU <dbl> 51.20, 51.89, 51.89, 51.89, 52.59, 52.98, 54.4
## $ C    <dbl> 47.45, 47.25, 47.25, 47.25, 47.60, 47.80, 48.2
```

Manually specifying column types when reading

It looks as if Date has type <chr>. But R has a special date type.

```
d <- read_csv("data/prices.csv",
              col_types=cols(Date=col_date(
                format="%d/%m/%Y")))
head(d, n=3)
```

```
## # A tibble: 3 x 14
##   Date         AAPL  MSFT  YHOO  NFLX   BMW     F    FB  GOO
##   <date>      <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl
## 1 2014-03-27  76.8  39.4  35.6  52.0  91.0  15.2  61.0  557
## 2 2014-03-28  76.7  40.3  35.9  51.3  92.0  15.4  60.0  558
## 3 2014-03-29  76.7  40.3  35.9  51.3  92.0  15.4  60.0  558
## # ... with 3 more variables: LULU <dbl>, C <dbl>, JPM <dbl>
```

See https://r4ds.had.co.nz/data-import.html for more examples like this.

Accessing subsets of a tibble

The $ operator gets a named column as a vector:

```
d$MSFT
```

```
##     [1] 39.360 40.300 40.300 40.300 40.990 41.420 41.350 41.
##    [10] 39.870 39.870 39.820 39.820 40.470 39.360 39.190 39.
##    [19] 39.180 39.750 40.400 40.010 40.010 40.010 40.010 39.
##    [28] 39.690 39.860 39.910 39.910 39.910 40.870 40.510 40.
##    [37] 39.690 39.690 39.690 39.430 39.060 39.400 39.640 39.
##    [46] 39.540 39.970 40.420 40.240 39.600 39.830 39.830 39.
##    [55] 39.680 40.350 40.100 40.120 40.120 40.120 40.120 40.
##    [64] 40.340 40.940 40.940 40.940 40.790 40.290 40.320 41.
##    [73] 41.480 41.480 41.270 41.110 40.860 40.580 41.230 41.
##    [82] 41.500 41.680 41.650 41.510 41.670 41.670 41.670 41.
##    [91] 42.030 41.720 42.250 42.250 42.250 41.700 41.870 41.
##   [100] 41.800 41.800 41.800 42.010 41.780 41.670 41.685 42.
##   [109] 42.090 42.140 42.450 44.080 44.530 44.710 44.710 44.
##   [118] 44.830 44.870 44.400 44.500 44.500 44.500 43.970 43.
```

Accessing subsets of a tibble

Square brackets, name as string => 1-column tibble:

```
d["MSFT"]
```

```
## # A tibble: 1,010 x 1
##      MSFT
##     <dbl>
##  1  39.4
##  2  40.3
##  3  40.3
##  4  40.3
##  5  41.0
##  6  41.4
##  7  41.4
##  8  41.0
##  9  39.9
## 10  39.9
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Square brackets and column number as int, same effect:

```
d[2]
```

```
## # A tibble: 1,010 x 1
##      AAPL
##     <dbl>
##  1  76.8
##  2  76.7
##  3  76.7
##  4  76.7
##  5  76.7
##  6  77.4
##  7  77.5
##  8  77.0
##  9  76.0
## 10  76.0
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Square brackets and "slice" – select several columns of d:

```
d[2:4]
```

```
## # A tibble: 1,010 x 3
##      AAPL  MSFT  YHOO
##     <dbl> <dbl> <dbl>
##  1  76.8   39.4  35.6
##  2  76.7   40.3  35.9
##  3  76.7   40.3  35.9
##  4  76.7   40.3  35.9
##  5  76.7   41.0  35.9
##  6  77.4   41.4  36.5
##  7  77.5   41.4  36.6
##  8  77.0   41.0  35.8
##  9  76.0   39.9  34.3
## 10  76.0   39.9  34.3
## # ... with 1,000 more rows
```

Accessing subsets of a tibble

Get the first 10 rows of MSFT column:

```
d$MSFT[1:10]
```

```
##  [1] 39.36 40.30 40.30 40.30 40.99 41.42 41.35 41.01 39.87
```

Non-tidy data (1)

	treatmenta	treatmentb
John Smith	—	2
Jane Doe	16	11
Mary Johnson	3	1

Non-tidy data (2)

	John Smith	Jane Doe	Mary Johnson
treatmenta	—	16	3
treatmentb	2	11	1

Tidy Data

person	treatment	result
John Smith	a	—
Jane Doe	a	16
Mary Johnson	a	3
John Smith	b	2
Jane Doe	b	11
Mary Johnson	b	1

"In tidy data. . .

Every value belongs to a variable and an observation. [. . .]

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

This is Codd's 3rd normal form" – R4DS

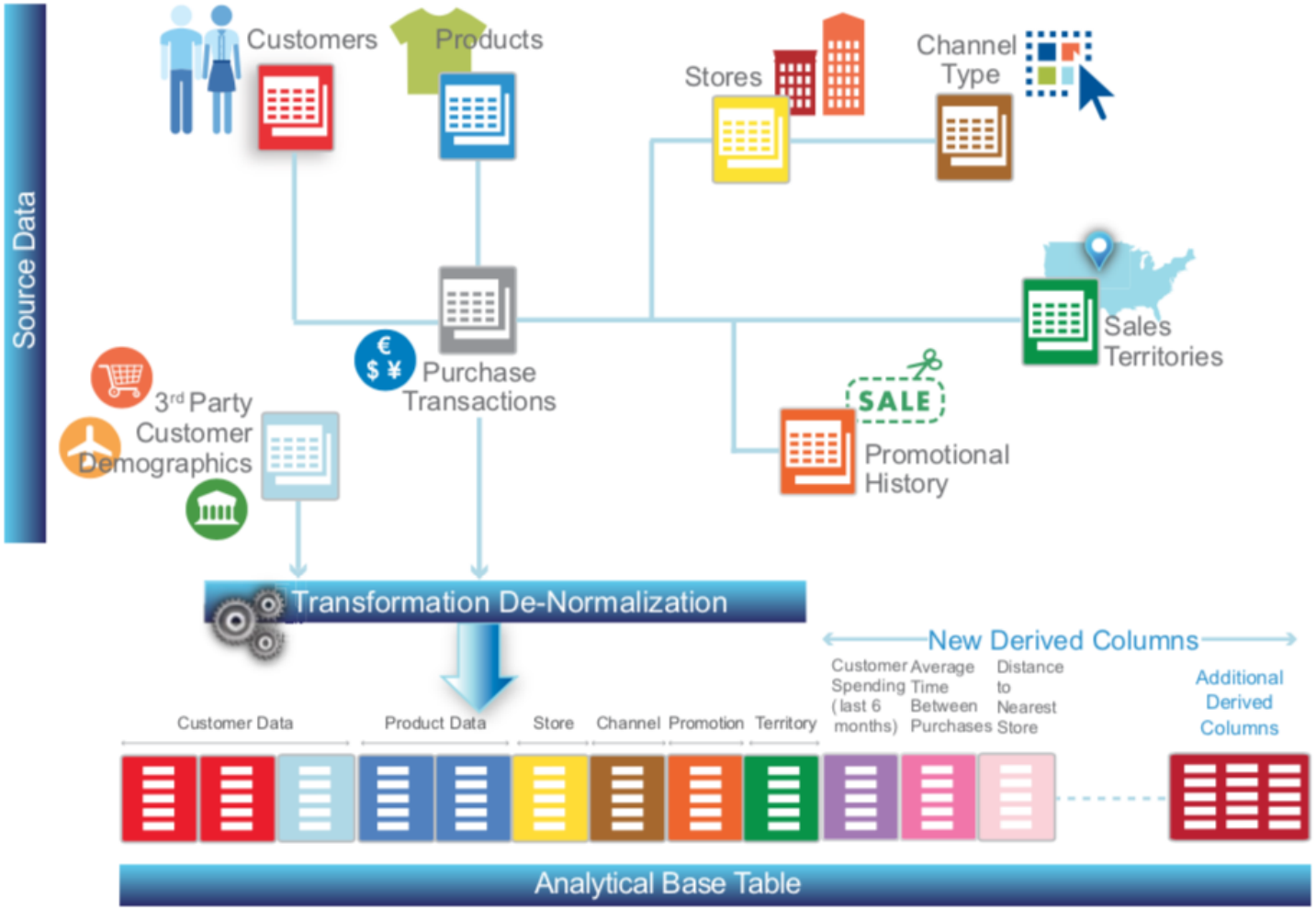
- *Is it?* (See `code/tidy_data.xlsx`)

Forms of data

- Tidy Data versus Codd's 3rd Normal Form (3NF), studied in database theory
- Tidy Data versus analytical base table (ABT)
- Some theory: http://www.jstatsoft.org/v59/i10/paper.

Analytical base table

A DB in 3NF has multiple tables and no repetition of data. An ABT is one table with repetition and derived data. (Image from SAS).



Why Tidy Data?

- Consistent underlying structure => consistent tools (`ggplot`, `dplyr`, etc.) => easier to learn
- Suits R's vectorisation

Another important motivation: if they are regularly queried together, we gain ease of querying. We may lose efficiency of storage, may re-introduce potential anomalies (relative to 3NF).

gather

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

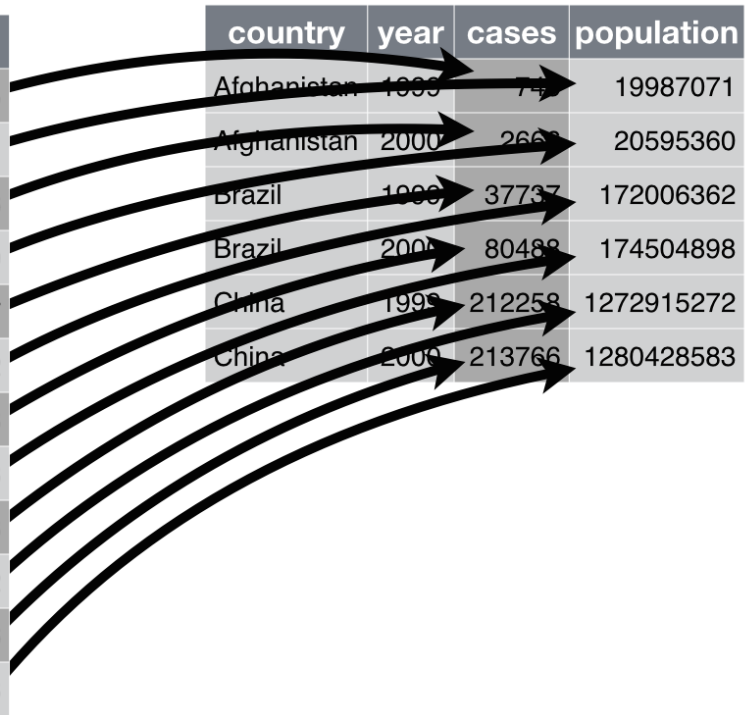
table4

```
gather(table4b, key="year", value="cases", c("1999", "2000"))
```

spread

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




```
spread(table2, key="key", value="value")
```

Some more tidying functions

- `separate`: separate a column which encodes two variables into two columns, e.g. `name` "James McDermott" -> `first_name` "James", `second_name` "McDermott"
- `unite`: the opposite
- `complete`: add missing rows (and use `NA` as the missing value)

Exercises

1. Recall our experiment on running time for sorting an array of different sizes. The original data (before we added extra columns) is available in `data/sort_times_original.csv`. Read it in to a `tibble`. (You might need to set the working directory first.)
2. Use `glimpse` to take a look. What types do the columns have?
3. In what way is this *not* tidy data? Use `gather` to fix it. Hint: the result should have shape 50 x 3 with columns `n`, `run_number`, `run_time`.
4. It would be nicer if `run_number` was just an integer, eg `0`, instead of `run0`. Use `separate` to split it into two parts. Hint: use `into=c("dummy", "run_number")`.
5. Look again at the result. We don't need that "dummy" column. Use `NA` to omit it. Hint: see `?separate` for help on `into`.
6. Look again – `run_number` is still not an integer! Fix this. Hint: `separate` can guess the correct type to convert to, but see `?separate` again to see how to ask it to.
7. Write it to a file `data/sort_times_tidy.csv` using `write_csv()`.

Solutions

Exercise 1

```
d <- read_csv("data/sort_times_original.csv")
```

```
## Parsed with column specification:
## cols(
##   n = col_double(),
##   run0 = col_double(),
##   run1 = col_double(),
##   run2 = col_double(),
##   run3 = col_double(),
##   run4 = col_double()
## )
```

Exercise 2

```
glimpse(d) # All columns of type `dbl`, which is ok
```

```
## Observations: 10
## Variables: 6
## $ n    <dbl> 1e+06, 2e+06, 3e+06, 4e+06, 5e+06, 6e+06, 7e+0
## $ run0 <dbl> 0.09924603, 0.19706607, 0.30280304, 0.4454867
## $ run1 <dbl> 0.1099961, 0.1945050, 0.3008888, 0.5314040, 0.
## $ run2 <dbl> 0.1018548, 0.2033260, 0.3165970, 0.4161611, 0.
## $ run3 <dbl> 0.1004527, 0.1933441, 0.3653409, 0.4889781, 0.
## $ run4 <dbl> 0.1140921, 0.2565329, 0.3853610, 0.4850140, 0.
```

Exercise 3

```
d <- gather(d, key="run_number", value="run_time",
            run0, run1, run2, run3, run4)
```

Exercise 4

```
separate(d, run_number,
         into=c("dummy", "run_number"), sep=3)
```

```
## # A tibble: 50 x 4
##            n dummy run_number run_time
##        <dbl> <chr> <chr>          <dbl>
##  1   1000000 run   0              0.0992
##  2   2000000 run   0              0.197
##  3   3000000 run   0              0.303
##  4   4000000 run   0              0.445
##  5   5000000 run   0              0.584
##  6   6000000 run   0              0.771
##  7   7000000 run   0              1.54
##  8   8000000 run   0              0.982
##  9   9000000 run   0              1.24
## 10  10000000 run   0              1.38
## # ... with 40 more rows
```

Exercise 5

```
separate(d, run_number,
         into=c(NA, "run_number"), sep=3)
```

```
## # A tibble: 50 x 3
##            n run_number run_time
##        <dbl> <chr>          <dbl>
##  1   1000000 0              0.0992
##  2   2000000 0              0.197
##  3   3000000 0              0.303
##  4   4000000 0              0.445
##  5   5000000 0              0.584
##  6   6000000 0              0.771
##  7   7000000 0              1.54
##  8   8000000 0              0.982
##  9   9000000 0              1.24
## 10  10000000 0              1.38
## # ... with 40 more rows
```

Exercise 6

```
d <- separate(d, run_number,
              into=c(NA, "run_number"), sep=3,
              convert=TRUE)
d
```

```
## # A tibble: 50 x 3
##            n run_number run_time
##        <dbl>      <int>    <dbl>
##  1   1000000          0   0.0992
##  2   2000000          0   0.197
##  3   3000000          0   0.303
##  4   4000000          0   0.445
##  5   5000000          0   0.584
##  6   6000000          0   0.771
##  7   7000000          0   1.54
##  8   8000000          0   0.982
##  9   9000000          0   1.24
## 10  10000000          0   1.38
```

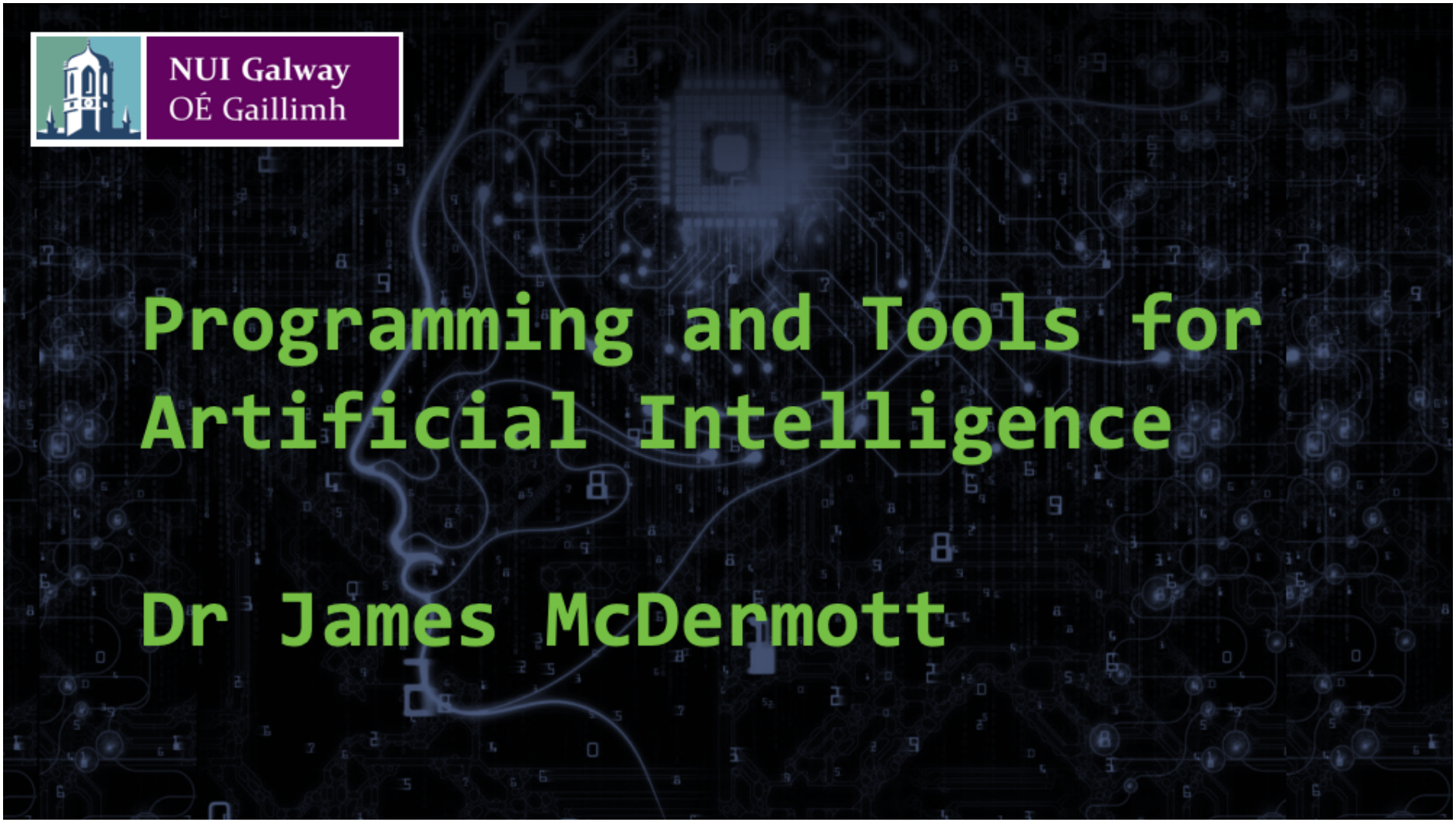
Exercise 7

```
write_csv(d, "data/sort_times_tidy.csv")
```

R dplyr

James McDermott

NUI Galway


NUI Galway
OÉ Gaillimh
Programming and Tools for Artificial Intelligence
Dr James McDermott

dplyr

dplyr

dplyr is a package for relational operations on data. That is, it does stuff similar to SQL, which many students will be familiar with (also comparable to Excel and Pandas). In particular:

- `filter` (choose rows, like SQL `where`)
- `arrange` (sort rows)
- `select` (choose columns, like SQL `select`)
- `mutate` (add columns)
- `summarise` (condense multiple values)
- `sample_n`, `sample_frac` (for taking a quick look at a sub-sample, see also `head`)
- `inner_join`, `left_join`, `right_join`, `full_join` (join two tables, like SQL `join`)

`dplyr` and the pipe operator

All the `dplyr` verbs (`select`, etc.) have three things in common (from https://cran.r-project.org/web/packages/dplyr/vignettes/dplyr.html):

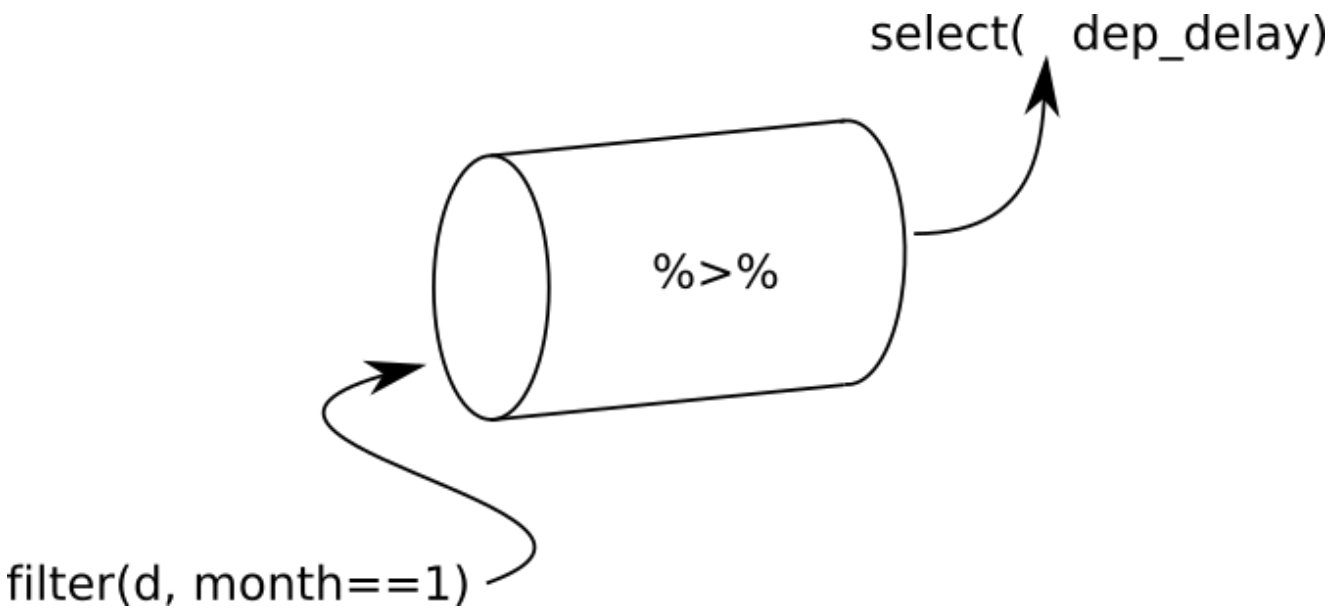
1. The first argument is a data frame [actually, a tibble].
2. The subsequent arguments describe what to do with the data frame. You can refer to columns in the data frame directly without using $.
3. The result is a new data frame

Therefore, it is natural to *chain* operations. That is where the *pipe* operator comes in.

The pipe %>%

- In Unix, the *pipe* symbol | is used to pass data from one command to another, e.g. `ls | grep -v "#"`
- In `magrittr`, the pipe %>% passes the output of its left-hand side to become the first argument of its right-hand side.

The pipe %>%




```
filter(d, month==1) %>% select(dep_delay)
```

The pipe %>%

In R, special operators are often named with double % symbols. The *pipe* operator %>% is an example.

`t %>% f()` means precisely `f(t)`, i.e. the output of the LHS becomes the first argument of the RHS.

It is useful to avoid complicated nested expressions:

`t %>% f("abc") %>% g("x", "y")` is easier to read than

```
g(f(t, "abc"), "x", "y")
```

(isn't it?)

Example: NYC Flights

```
# uncomment if not already installed
# install.packages("nycflights13")
library(nycflights13)
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------

## v ggplot2 3.2.1       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0

## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Take a quick look at data

```
flights
```

```
## # A tibble: 336,776 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_
##     <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## # ... with 336,766 more rows, and 12 more variables: sched_
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <
```

filter

Let's see just flights on Jan 1:

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_
##     <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## #     with 832 more rows, and 12 more variables: sched_arr
```

filter

Remember, this is internally translated to:

```
filter(flights, month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_
##     <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## #     with 832 more rows, and 12 more variables: sched_arr
```

filter

Which flight departure was delayed the longest? We can use `filter` again:

```
flights %>% filter(dep_delay == max(dep_delay, na.rm=TRUE))
```

```
## # A tibble: 1 x 19
##    year month   day dep_time sched_dep_time dep_delay arr_t
##   <int> <int> <int>    <int>          <int>     <dbl>    <i
## 1  2013     1     9      641            900      1301     1
## # ... with 12 more variables: sched_arr_time <int>, arr_del
## #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr
## #   air_time <dbl>, distance <dbl>, hour <dbl>, minute <dbl
## #   time_hour <dttm>
```

Notice that `na.rm=TRUE` *inside* `max()` is essential: consider `max(flights$dep_delay)` to see why.

Syntax notes

1. `na.rm` argument
2. We can refer to columns with no special syntax (not even quotes)
3. Remember == for equality (I put spaces), but = for passing keyword arguments (I don't put spaces, as in Python).

filter examples

Boolean AND: use comma as we already saw

```
flights %>% filter(month == 1, day == 1)
```

```
## # A tibble: 842 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_
##     <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## #     with 832 more rows, and 12 more variables: sched_arr
```

filter examples

Boolean OR: use |

```
flights %>% filter(month == 1 | month == 12)
```

```
## # A tibble: 55,139 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_
##    <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## #     with 55,129 more rows, and 12 more variables: sched_a
```

filter examples

%in% operator does the same as above:

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

```
## # A tibble: 55,139 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_
##    <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     1      517            515         2
##  2  2013     1     1      533            529         4
##  3  2013     1     1      542            540         2
##  4  2013     1     1      544            545        -1
##  5  2013     1     1      554            600        -6
##  6  2013     1     1      554            558        -4
##  7  2013     1     1      555            600        -5
##  8  2013     1     1      557            600        -3
##  9  2013     1     1      557            600        -3
## 10  2013     1     1      558            600        -2
## #     with 55,129 more rows, and 12 more variables: sched_a
```

arrange

dplyr becomes like a programmatic interface to Excel,
e.g. sort-by-column:

```
arrange(flights, dep_delay) # sort-by-column
```

```
## # A tibble: 336,776 x 19
##      year month   day dep_time sched_dep_time dep_delay arr_
##     <int> <int> <int>    <int>          <int>     <dbl>  <
##  1  2013    12     7     2040           2123       -43
##  2  2013     2     3     2022           2055       -33
##  3  2013    11    10     1408           1440       -32
##  4  2013     1    11     1900           1930       -30
##  5  2013     1    29     1703           1730       -27
##  6  2013     8     9      729            755       -26
##  7  2013    10    23     1907           1932       -25
##  8  2013     3    30     2030           2055       -25
##  9  2013     3     2     1431           1455       -24
## 10  2013     5     5      934            958       -24
```

arrange

Descending order:

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

```
## # A tibble: 336,776 x 19
##     year month   day dep_time sched_dep_time dep_delay arr_
##    <int> <int> <int>    <int>          <int>     <dbl>    <
##  1  2013     1     9      641            900      1301
##  2  2013     6    15     1432           1935      1137
##  3  2013     1    10     1121           1635      1126
##  4  2013     9    20     1139           1845      1014
##  5  2013     7    22      845           1600      1005
##  6  2013     4    10     1100           1900       960
##  7  2013     3    17     2321            810       911
##  8  2013     6    27      959           1900       899
##  9  2013     7    22     2257            759       898
## 10  2013    12     5      756           1700       896
## # ... with 336,766 more rows, and 12 more variables: sched
```

`select` chooses columns

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

```
## # A tibble: 336,776 x 3
##      day month  year
##    <int> <int> <int>
##  1     1     1  2013
##  2     1     1  2013
##  3     1     1  2013
##  4     1     1  2013
##  5     1     1  2013
##  6     1     1  2013
##  7     1     1  2013
##  8     1     1  2013
##  9     1     1  2013
## 10     1     1  2013
## # ... with 336,766 more rows
```

`select` chooses columns

Select all columns from year to day (behind the scenes, the columns are *numbers*):

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

```
## # A tibble: 336,776 x 3
##     year month   day
##    <int> <int> <int>
##  1  2013     1     1
##  2  2013     1     1
##  3  2013     1     1
##  4  2013     1     1
##  5  2013     1     1
##  6  2013     1     1
##  7  2013     1     1
##  8  2013     1     1
##  9  2013     1     1
## 10  2013     1     1
```

Using `select` to deselect

```
select(flights, -(month:minute))
```

```
## # A tibble: 336,776 x 2
##      year time_hour
##     <int> <dttm>
##  1  2013 2013-01-01 05:00:00
##  2  2013 2013-01-01 05:00:00
##  3  2013 2013-01-01 05:00:00
##  4  2013 2013-01-01 05:00:00
##  5  2013 2013-01-01 06:00:00
##  6  2013 2013-01-01 05:00:00
##  7  2013 2013-01-01 06:00:00
##  8  2013 2013-01-01 06:00:00
##  9  2013 2013-01-01 06:00:00
## 10  2013 2013-01-01 06:00:00
## # ... with 336,766 more rows
```

More ways to select columns

```
select(flights, starts_with("d"))
```

```
## # A tibble: 336,776 x 5
##      day dep_time dep_delay dest  distance
##    <int>    <int>     <dbl> <chr>    <dbl>
##  1     1      517         2 IAH       1400
##  2     1      533         4 IAH       1416
##  3     1      542         2 MIA       1089
##  4     1      544        -1 BQN       1576
##  5     1      554        -6 ATL        762
##  6     1      554        -4 ORD        719
##  7     1      555        -5 FLL       1065
##  8     1      557        -3 IAD        229
##  9     1      557        -3 MCO        944
## 10     1      558        -2 ORD        733
## # ... with 336,766 more rows
```

mutate

```
select(flights, year:day, ends_with("delay"),
               distance, air_time) %>%
  mutate(gain=dep_delay - arr_delay) %>%
  mutate(speed=distance / air_time * 60)
```

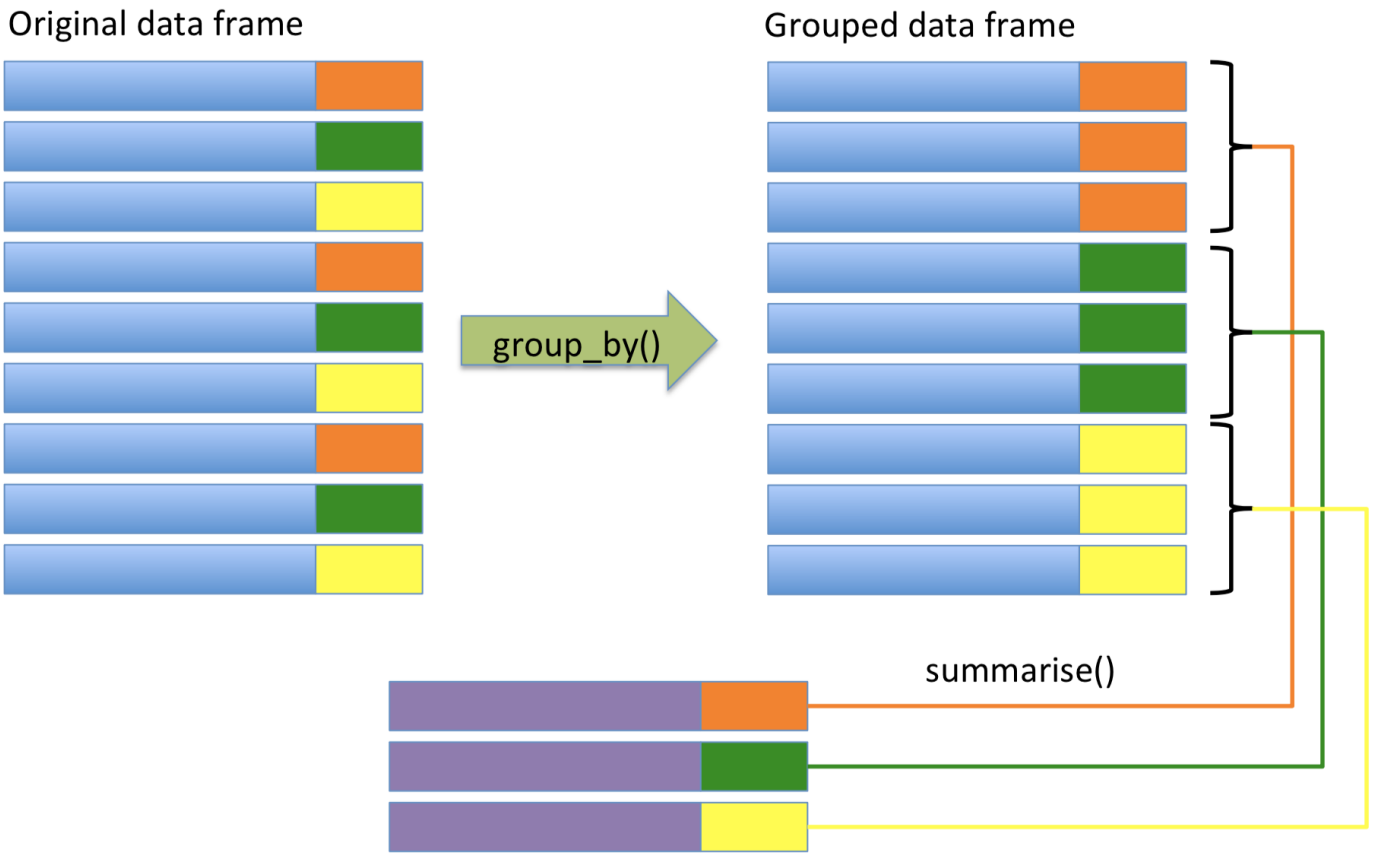
```
## # A tibble: 336,776 x 9
##      year month   day dep_delay arr_delay distance air_time
##     <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
##  1  2013     1     1         2        11     1400      227
##  2  2013     1     1         4        20     1416      227
##  3  2013     1     1         2        33     1089      160
##  4  2013     1     1        -1       -18     1576      183
##  5  2013     1     1        -6       -25      762      116
##  6  2013     1     1        -4        12      719      150
##  7  2013     1     1        -5        19     1065      158
##  8  2013     1     1        -3       -14      229       53
##  9  2013     1     1        -3        -8      944      140
```

mutate

Remember none of these operations change the tibble itself, just return a new one. So we may decide to save the result in a new variable.

```
sml <- select(flights, year:day, ends_with("delay"),
              distance, air_time) %>%
  mutate(gain=dep_delay - arr_delay) %>%
  mutate(speed=distance / air_time * 60)
```

group_by



(R4DS)

group_by and summarise go well together

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

```
## # A tibble: 16 x 2
##    carrier dep_delay
##    <chr>       <dbl>
##  1 9E          16.7
##  2 AA           8.59
##  3 AS           5.80
##  4 B6          13.0
##  5 DL           9.26
##  6 EV          20.0
##  7 F9          20.2
##  8 FL          18.7
##  9 HA           4.90
## 10 MQ          10.6
## 11 OO          12.6
```

A bigger example: `who`

`who` is a dataset from the World Health Organisation which needs a lot of cleaning:

```
who %>%
  # gather many columns to 1
  gather(key, value, new_sp_m014:newrel_f65,
         na.rm = TRUE) %>%
  # fix inconsistent spelling
  mutate(key = stringr::str_replace(key, "newrel", "new_rel")) %>%
  # split eg "new_sp_m014" -> "new", "sp", "m014"
  separate(key, c("new", "var", "sexage")) %>%
  # remove redundant/unneeded columns
  select(-new, -iso2, -iso3) %>%
  # split eg "m014" -> "m", "014"
  separate(sexage, c("sex", "age"), sep = 1)
```

```
## # A tibble: 76,046 x 6
```

A bigger example: `who`

See https://r4ds.had.co.nz/tidy-data.html for detailed explanation.
Exercise: look at the output of each step of this transformation, starting at `who` itself, to understand the need for the next.

Some more handy functions (some from `dplyr`)

- Offset a vector of values: `lead` and `lag`
- Cumulative calculations: `cumsum`, `cummax`, etc.
- Where does each value come in a sort? `min_rank`
- Counts: `n`, `n_distinct`

Functional programming in R

- `dplyr` and the pipe are already examples of functional programming! E.g. all these operations don't change their input, just return a new version.
- Our friend `map` also exists in R. It makes a list.
- `map_dbl` and friends may be more useful since they return `vectors`. Compare `map` and `map_dbl` in the following.

```
d = 1:5
# R uses the opposite argument ordering, compared to Python
map(d, sqrt)
```

```
d = 1:5
map_dbl(d, sqrt)
```

Functional programming in R

```
d %>% map_dbl(sqrt) # equivalent, using pipe
```

Functional programming in R

`map` and friends come from the `purrr` package, well-documented here: https://r4ds.had.co.nz/iteration.html#the-map-functions

- The Joy of Functional Programming ACM Tech Talk webcast with Hadley Wickham can be viewed here: https://learning.acm.org/techtalks/functionalprogramming (Prerequisites: basic R, tibbles, distinction between lists, vectors, dataframes)

Summary

- tidy data: columns are variables, rows are observations
- tibbles
- pipe %>%
- verbs including `select`, `filter`, `mutate`, `arrange`, `rename`, `gather`, `spread`

Let's look at a cheatsheet for `dplyr`:
https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf

Exercises

- Exercise 1: Our sort times data is available in tidy format as `sort_times_tidy.csv`. Use `group_by` and `summarise` to get the mean and the standard deviation for each `n`, and then for each `run_number`.
- A dataset of characters in *Star Wars* is available as `dplyr::starwars`. Exercise 2: Find all the human females. Exercise 3: Find the characters who are human *or* Wookiee. Exercise 4: Find the shortest character. Hint: recall we might need `na.rm`. Exercise 5: Add a new column called `BMI` giving the body mass index, where the formula is BMI $= m/h^2$ for mass m in kg and height h in metres. https://en.wikipedia.org/wiki/Body_mass_index. Exercise 6: Which character has the highest BMI?

Solutions

Exercise 1

```
d <- read_csv("data/sort_times_tidy.csv")
```

```
## Parsed with column specification:
## cols(
##   n = col_double(),
##   run_number = col_double(),
##   run_time = col_double()
## )
```

```
d %>% group_by(n) %>% summarise(mean_run_time=mean(run_time),
```

```
## # A tibble: 10 x 3
##           n mean_run_time sd_run_time
##       <dbl>         <dbl>       <dbl>
##  1   1000000         0.105     0.00654
##  2   2000000         0.209     0.0269
##  3   3000000         0.334     0.0387
```

Exercise 1

Notice that the mean and stddev for $n = 7$ million are anomalously high. One way this could occur is if our computer had a spike in CPU usage during the experiment, e.g. due to a browser loading a video.

Exercise 1

```
d %>% group_by(run_number) %>%
  summarise(mean_run_time=mean(run_time),
            sd_run_time=sd(run_time))
```

```
## # A tibble: 5 x 3
##   run_number mean_run_time sd_run_time
##        <dbl>         <dbl>       <dbl>
## 1          0         0.754       0.512
## 2          1         0.644       0.368
## 3          2         0.604       0.353
## 4          3         0.648       0.369
## 5          4         0.678       0.416
```

No major anomalies this time.

Exercise 2

```
sw <- dplyr::starwars
sw %>% filter(species == "Human", gender == "female") # human
```

```
## # A tibble: 9 x 13
##   name  height  mass hair_color skin_color eye_color birth_
##   <chr>  <int> <dbl> <chr>      <chr>       <chr>           <
## 1 Leia~    150    49 brown      light       brown
## 2 Beru~    165    75 brown      light       blue
## 3 Mon ~    150    NA auburn     fair        blue
## 4 Shmi~    163    NA black      fair        brown
## 5 Cordé    157    NA brown      light       brown
## 6 Dormé    165    NA brown      light       brown
## 7 Joca~    167    NA white      fair        blue
## 8 Rey       NA    NA brown      light       hazel
## 9 Padm~    165    45 brown      light       brown
## # ... with 5 more variables: homeworld <chr>, species <chr>
## #   vehicles <list>, starships <list>
```

Exercise 3

```
sw %>% filter(species == "Human" | species == "Wookiee") # hu
```

```
## # A tibble: 37 x 13
##    name  height  mass hair_color skin_color eye_color birth
##    <chr>  <int> <dbl> <chr>      <chr>      <chr>
##  1 Luke~    172    77 blond      fair       blue
##  2 Dart~    202   136 none       white      yellow
##  3 Leia~    150    49 brown      light      brown
##  4 Owen~    178   120 brown, gr~ light      blue
##  5 Beru~    165    75 brown      light      blue
##  6 Bigg~    183    84 black      light      brown
##  7 Obi-~    182    77 auburn, w~ fair       blue-gray
##  8 Anak~    188    84 blond      fair       blue
##  9 Wilh~    180    NA auburn, g~ fair       blue
## 10 Chew~    228   112 brown      unknown    blue
## # ... with 27 more rows, and 5 more variables: homeworld <c
## #   species <chr>  films <list>  vehicles <list>  starships
```

Exercise 4

```
sw %>% filter(height == max(height, na.rm=TRUE))
```

```
## # A tibble: 1 x 13
##   name  height  mass hair_color skin_color eye_color birth_
##   <chr>  <int> <dbl> <chr>      <chr>       <chr>           <
## 1 Yara~    264    NA none       white       yellow
## # ... with 5 more variables: homeworld <chr>, species <chr>
## #   vehicles <list>, starships <list>
```

Exercise 5

```
# NB convert height from cm to metres before squaring
BMI <- function(h, m) {m / (h / 100)^2}
sw <- sw %>% mutate(bmi=BMI(height, mass))
```

Exercise 6

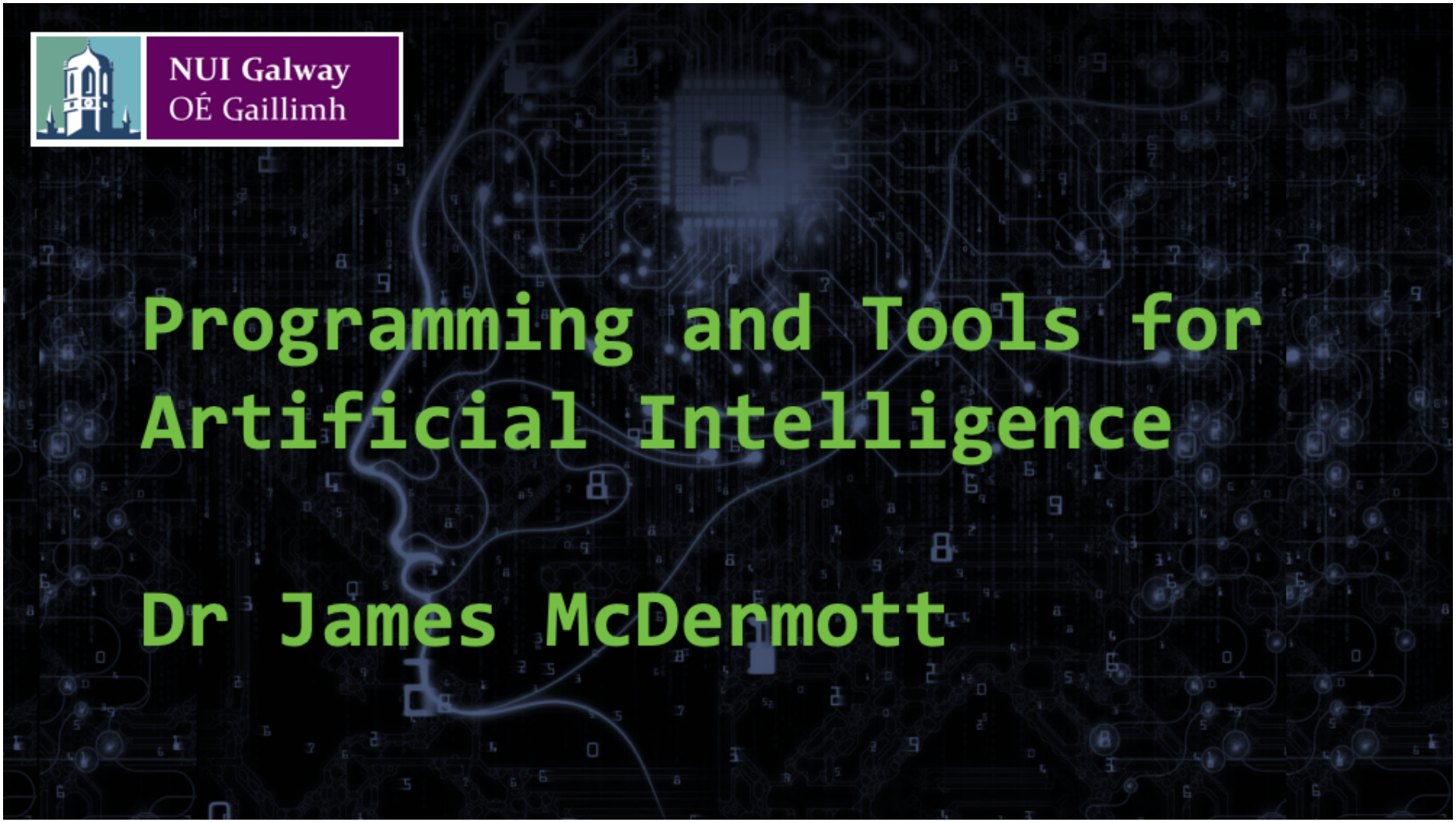
```
sw %>% filter(bmi == max(bmi, na.rm=TRUE))
```

```
## # A tibble: 1 x 14
##   name  height  mass hair_color skin_color eye_color birth_
##   <chr>  <int> <dbl> <chr>      <chr>       <chr>           <
## 1 Jabb~    175  1358 <NA>       green-tan~ orange
## # ... with 6 more variables: homeworld <chr>, species <chr>
## #   vehicles <list>, starships <list>, bmi <dbl>
```

dplyr joins

James McDermott

NUI Galway


NUI Galway
OÉ Gaillimh
Programming and Tools for Artificial Intelligence
Dr James McDermott

`dplyr` joins

Relational databases

The main ideas of relational databases (SQL) are probably familiar to all:

- A database consists of tables
- A table consists of a set of columns
- A column has a type, and maybe some constraints (e.g. positive integer)
- Some column(s) may be designated as a key for the table

Joins

- As we know, in Relational Databases, it is good practice to use *normalisation*: splitting a table up into multiple tables, to avoid duplication of information and the possibility of *update anomalies*. 3NF is the result of normalisation.
- Doing ML/stats/analytics may require *de-normalisation* – re-joining – eventually to export to our ML/stats/analytics system.

Before normalisation

Movie rental DB

Date	Movie	Genre	Customer	Address
01-Jan	Amelie	Romance	Bob	11, Haight St
02-Jan	The Matrix	Sci-fi	Frida	Oxford Circus
02-Jan	Amelie	Romance	Carrie	99, Fifth Ave
05-Jan	Skyfall	Adventure	Bob	11, Haight St
05-Jan	Avengers	Sci-fi	Frida	Oxford Circus

After normalisation: 3rd Normal Form (3NF)

Rentals table

Date	Movie ID	Customer ID
01-Jan	102	1
02-Jan	101	2
02-Jan	102	3
05-Jan	103	1
05-Jan	104	2

Customer table

Customer ID	Name	Address
1	Bob	11, Haight St
2	Frida	Oxford Circus
3	Carrie	99, Fifth Ave

Movie table

Movie ID	Name	Genre
101	The Matrix	Sci-fi
102	Amelie	Romance
103	Skyfall	Adventure
104	Avengers	Sci-fi

Key columns

After normalisation, the link between data is via key columns – in this case, the Customer ID and Movie ID columns. It is possible to put the original table back together using a **join**. We say that we join **on** the key column.

SQL

In SQL, a JOIN might be something like this. This is an *implicit join*:

```
SELECT * FROM RENTALS, CUSTOMER
WHERE RENTALS.CustomerID = CUSTOMER.CustomerID;
```

This is an equivalent *explicit join*:

```
SELECT * FROM RENTALS JOIN CUSTOMER
ON RENTALS.CustomerID = CUSTOMER.CustomerID;
```

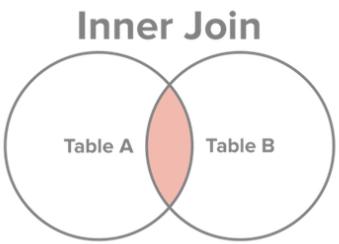
(This is not examinable.)

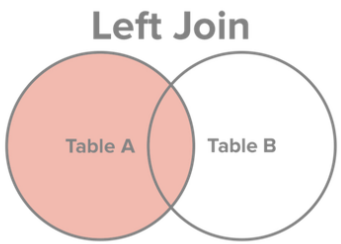
What is a join, really?

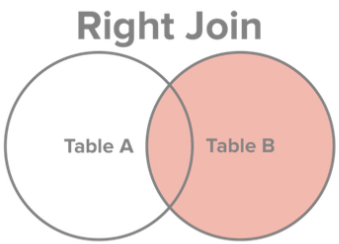
- Think of join as an *operator* whose left and right operands are tables, and whose result is a table formed as the union of their columns
- The *Cross join* is a good place to start. Conceptually, a cross join is a *Cartesian product of rows*. For every row in T1, we put it side by side with every row in T2. Think of that as a new joined table. Now we can select columns from it and filter rows using ON. In particular, we'll probably filter for rows where a key column in one table matches a key column in the other, discarding the large majority of this cross product.
- Other joins just restrict the "every row in T1" and "every row in T2" parts depending on which matches actually exist.

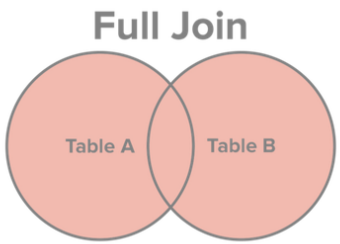
Different types of joins

There are a few types of joins. To distinguish them, many textbooks and cheatsheets proceed to Venn diagrams,
e.g. http://www.sql-join.com/sql-join-types (below). These are helpful as mnemonics but the language of Venn diagrams is not sufficient to define the different joins.

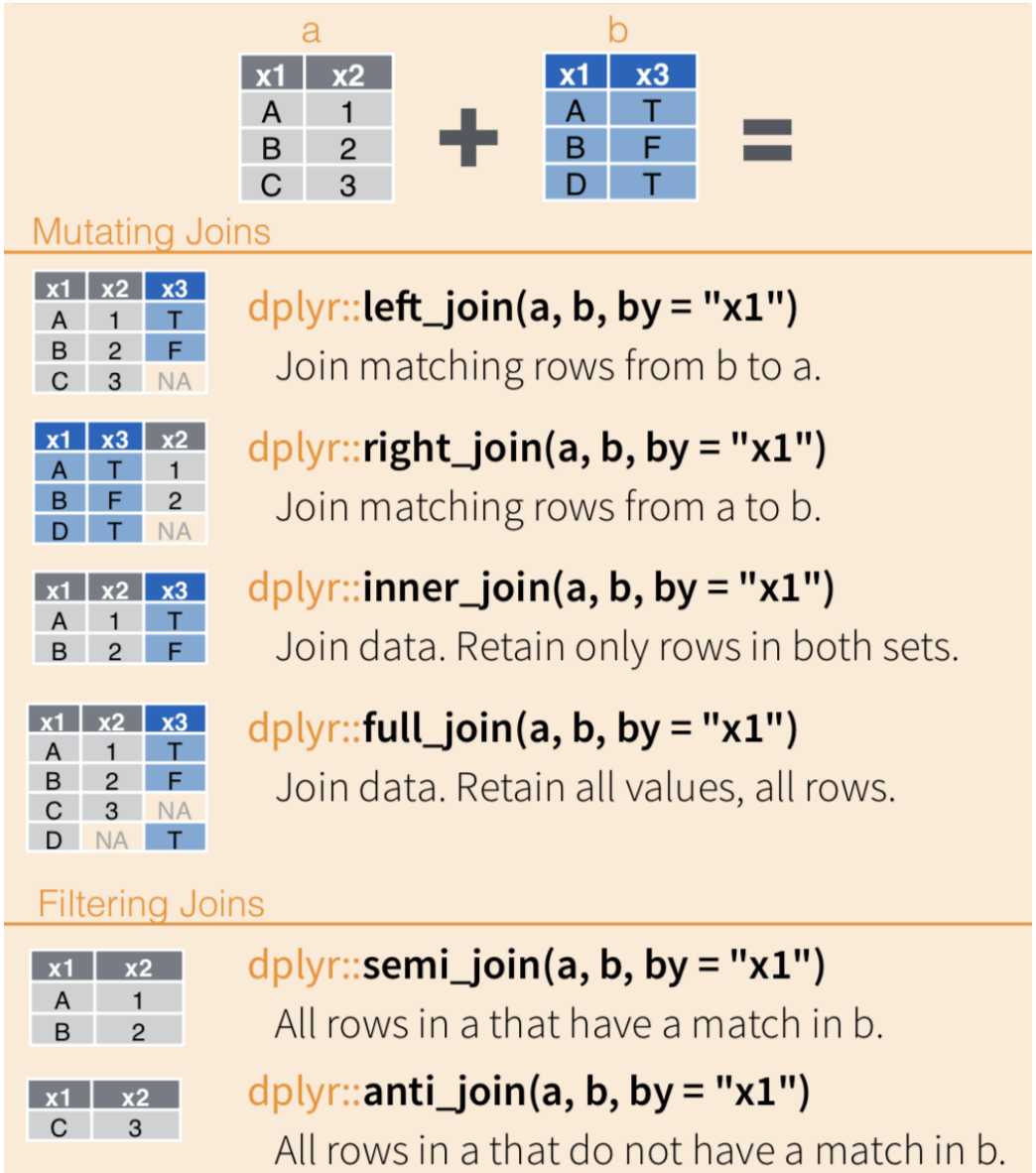








Different types of joins (Data Wrangling Cheatsheet)



a

x1	x2
A	1
B	2
C	3

+

b

x1	x3
A	T
B	F
D	T

=

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**

Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**

Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**

Join data. Retain all values, all rows.

Filtering Joins

x1	x2
A	1
B	2

dplyr::**semi_join(a, b, by = "x1")**

All rows in a that have a match in b.

x1	x2
C	3

dplyr::**anti_join(a, b, by = "x1")**

All rows in a that do not have a match in b.

Further reading

- Most people working in industry in the fields of AI, ML, Data Science, Statistics, etc., use relational databases and SQL a lot.
- We don't teach it, because it is usually seen as a topic for undergrad level. This MOOC is recommended as an optional catch-up or refresher:
  - Stanford Databases https://lagunita.stanford.edu/courses/DB/2014/SelfPaced/about
  - (The following topics in the MOOC are recommended for a "short version": Introduction, JSON, Relational Algebra (Section 1), SQL, Relational Design Theory (Section 1), Unified Modelling Language, Online Analytical Processing)

Exercises

1. Read the three data files `rentals.csv`, `movies.csv`, `customers.csv`, all in the `data/` directory, as tibbles.
2. Optional: get R to read the Date column correctly. Hint: https://readr.tidyverse.org/reference/parse_datetime.html
3. Using a `dplyr` join command, create a table showing the customer name and address for every rental.
4. Piping the result into another join command, recreate the full original table as shown under "Before Normalisation" above.
5. Notice the columns `Name.x` and `Name.y` which appear because there is a `Name` column in each of the Movies and Customers tables. Rename them.
6. Calculate the number of movies Frida watched of the Sci-fi genre.

Solutions

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Exercises 1 and 2:

```
rentals <- read_csv("data/rentals.csv",
                    col_types=cols(Date=col_date(
                      format="%d-%b-%Y")))
movies <- read_csv("data/movies.csv")
```

```
## Parsed with column specification:
## cols(
##   MovieID = col_double(),
##   Name = col_character(),
##   Genre = col_character()
## )
```

```
customers <- read_csv("data/customers.csv")
```

```
## Parsed with column specification:
## cols(
##   CustomerID = col_double(),
##   Name = col_character(),
##   Address = col_character()
```

Customer name and address for each rental

```
inner_join(rentals, customers, by="CustomerID")
```

```
## # A tibble: 5 x 5
##   Date       MovieID CustomerID Name   Address
##   <date>       <dbl>      <dbl> <chr>  <chr>
## 1 2018-01-01     102          1 Bob    11, Haight St
## 2 2018-01-02     101          2 Frida  Oxford Circus
## 3 2018-01-02     102          3 Carrie 99, Fifth Ave
## 4 2018-01-05     103          1 Bob    11, Haight St
## 5 2018-01-05     104          2 Frida  Oxford Circus
```

Recreate original table

```
inner_join(rentals, customers, by="CustomerID") %>%
  inner_join(movies, by="MovieID")
```

```
## # A tibble: 5 x 7
##   Date       MovieID CustomerID Name.x Address        Name.y
##   <date>       <dbl>      <dbl> <chr>  <chr>          <chr>
## 1 2018-01-01     102          1 Bob    11, Haight St  Amelie
## 2 2018-01-02     101          2 Frida  Oxford Circus  The Ma
## 3 2018-01-02     102          3 Carrie 99, Fifth Ave  Amelie
## 4 2018-01-05     103          1 Bob    11, Haight St  Skyfal
## 5 2018-01-05     104          2 Frida  Oxford Circus  Avenge
```

Rename columns

```
t = inner_join(rentals, customers, by="CustomerID") %>%
  inner_join(movies, by="MovieID") %>%
  rename(CustomerName=Name.x, MovieTitle=Name.y)
t
```

```
## # A tibble: 5 x 7
##   Date       MovieID CustomerID CustomerName Address      Mo
##   <date>       <dbl>      <dbl> <chr>        <chr>        <c
## 1 2018-01-01     102          1 Bob          11, Haight~  An
## 2 2018-01-02     101          2 Frida        Oxford Cir~  Th
## 3 2018-01-02     102          3 Carrie       99, Fifth ~  An
## 4 2018-01-05     103          1 Bob          11, Haight~  Sk
## 5 2018-01-05     104          2 Frida        Oxford Cir~  Av
```

Filter and count

```
t %>% filter(CustomerName=="Frida", Genre=="Sci-fi") %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     2
```

Filter and count

The following is a solution to the problem, but it requires the programmer to do all the work in their head. That's not scalable or flexible and it's error-prone, so don't do this.

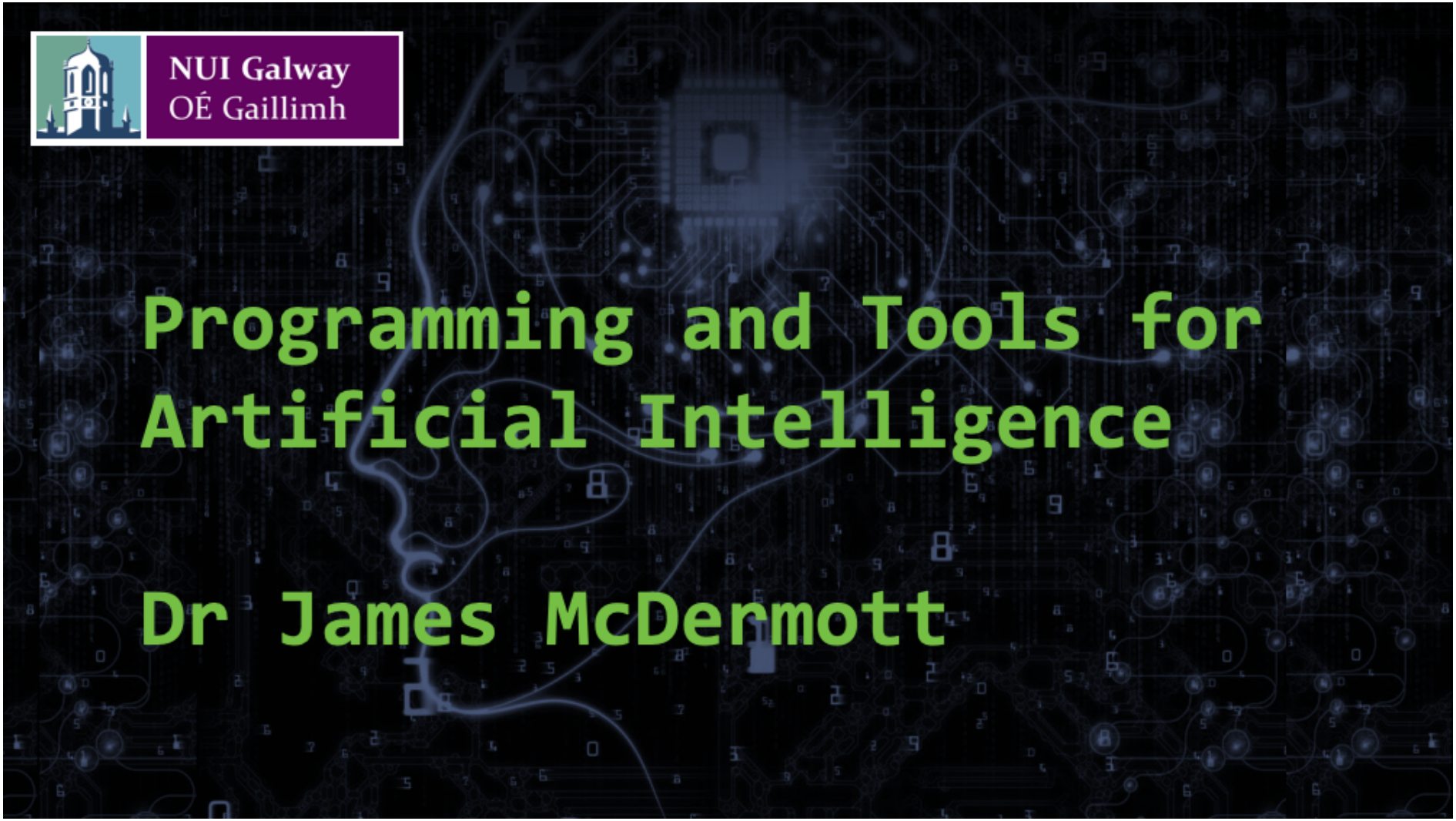
```
# Frida is CustomerID 2
# Movies 101 and 104 are Sci-fi
rentals %>% filter(CustomerID == 2,
                   MovieID %in% c(101, 104)) %>%
  count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     2
```

Plotting in R

James McDermott

NUI Galway



Plotting in R

Plotting in R

Plotting is a central part of the data analysis loop. `ggplot2` is a great library that works well with `dplyr` and the rest of the Tidyverse.

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------

## v ggplot2 3.2.1       v purrr   0.3.2
## v tibble  2.1.1       v dplyr   0.8.0.1
## v tidyr   0.8.3       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0

## -- Conflicts ------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Example (mpg dataset)

We previously saw a small extract of this dataset data/mpg_extract.csv. The full dataset is built-in to the ggplot2 library.

```
dt <- ggplot2::mpg
dt
```

```
## # A tibble: 234 x 11
##    manufacturer model displ  year   cyl trans drv     cty
##    <chr>        <chr> <dbl> <int> <int> <chr> <chr> <int> <
##  1 audi         a4      1.8  1999     4 auto~ f        18
##  2 audi         a4      1.8  1999     4 manu~ f        21
##  3 audi         a4      2    2008     4 manu~ f        20
##  4 audi         a4      2    2008     4 auto~ f        21
##  5 audi         a4      2.8  1999     6 auto~ f        16
##  6 audi         a4      2.8  1999     6 manu~ f        18
##  7 audi         a4      3.1  2008     6 auto~ f        18
##  8 audi         a4 q~   1.8  1999     4 manu~ 4        18
##  9 audi         a4 q~   1.8  1999     4 auto~ 4        16
```

Example (`mpg` dataset)

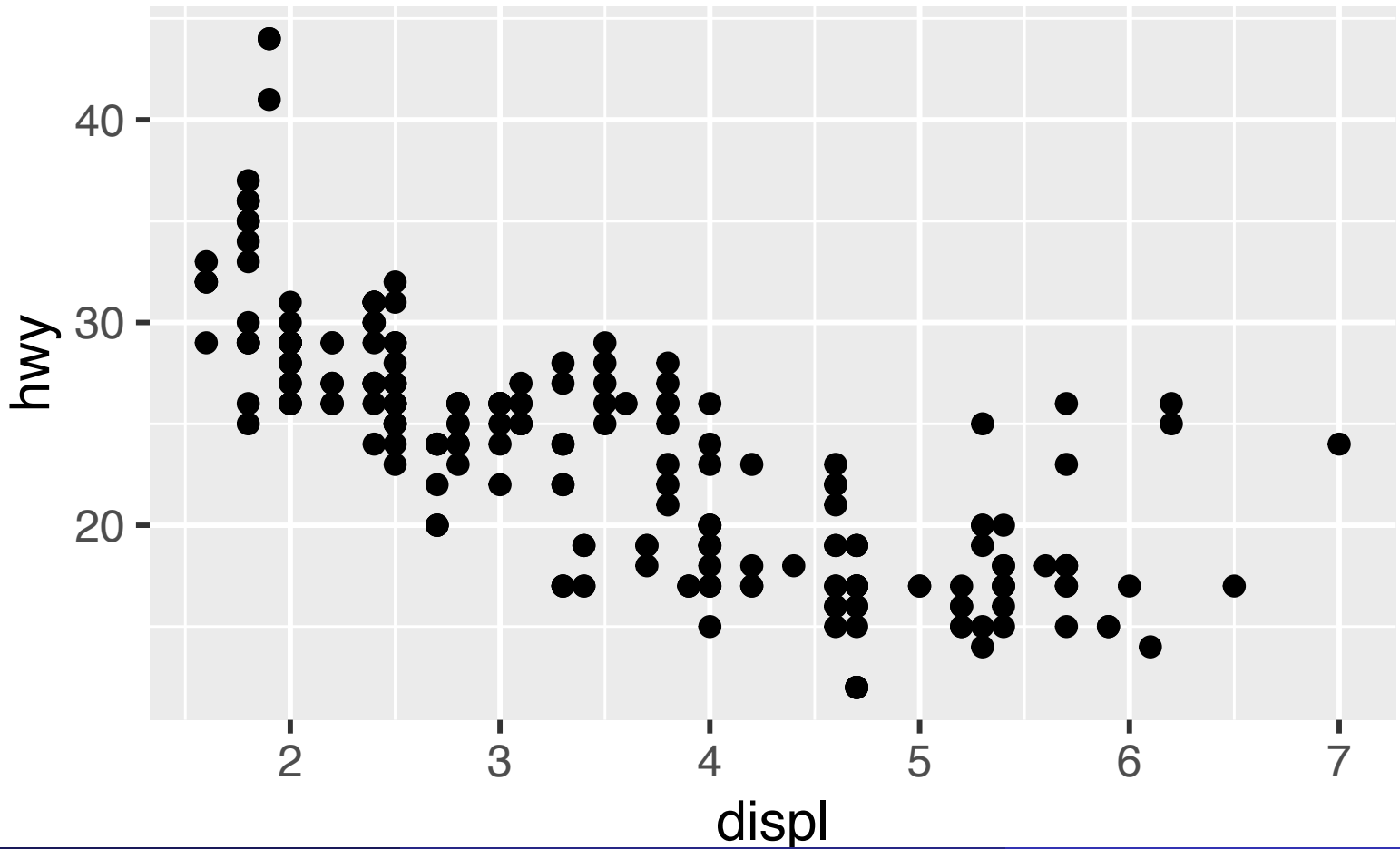
Do cars with big engines use more fuel than cars with small engines? Among the variables are:

- `displ`, a car's engine volume ("displacement") in litres
- `hwy`, a car's fuel efficiency on the highway in miles per gallon

Scatterplot using geom_point

The library is called ggplot2, but the function is called ggplot.

```
ggplot(data = dt) +
  geom_point(mapping = aes(x=displ, y=hwy))
```



Seeing/saving

- In R Studio, the plot appears in the bottom-right panel.
- If editing R Markdown, it appears inline.
- If we want to save:

```
ggsave("img/mpg_test.png")
```

```
## Saving 4 x 2.5 in image
```

General recipe for ggplot

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

The Grammar of Graphics

- Wilkinson, L. (2005), The Grammar of Graphics (2nd ed.). Statistics and Computing, New York: Springer.
- Wickham calls it "The most important modern work in graphical grammars"
- Every graph: a data set, a coordinate system, and visual marks representing data
- Wickham wrote the `ggplot2` package, an implementation of the grammar of graphics, which is used by most R practitioners.

ggplot2 concepts

- Layers
- Aesthetic mappings
- Geometric objects

Layers

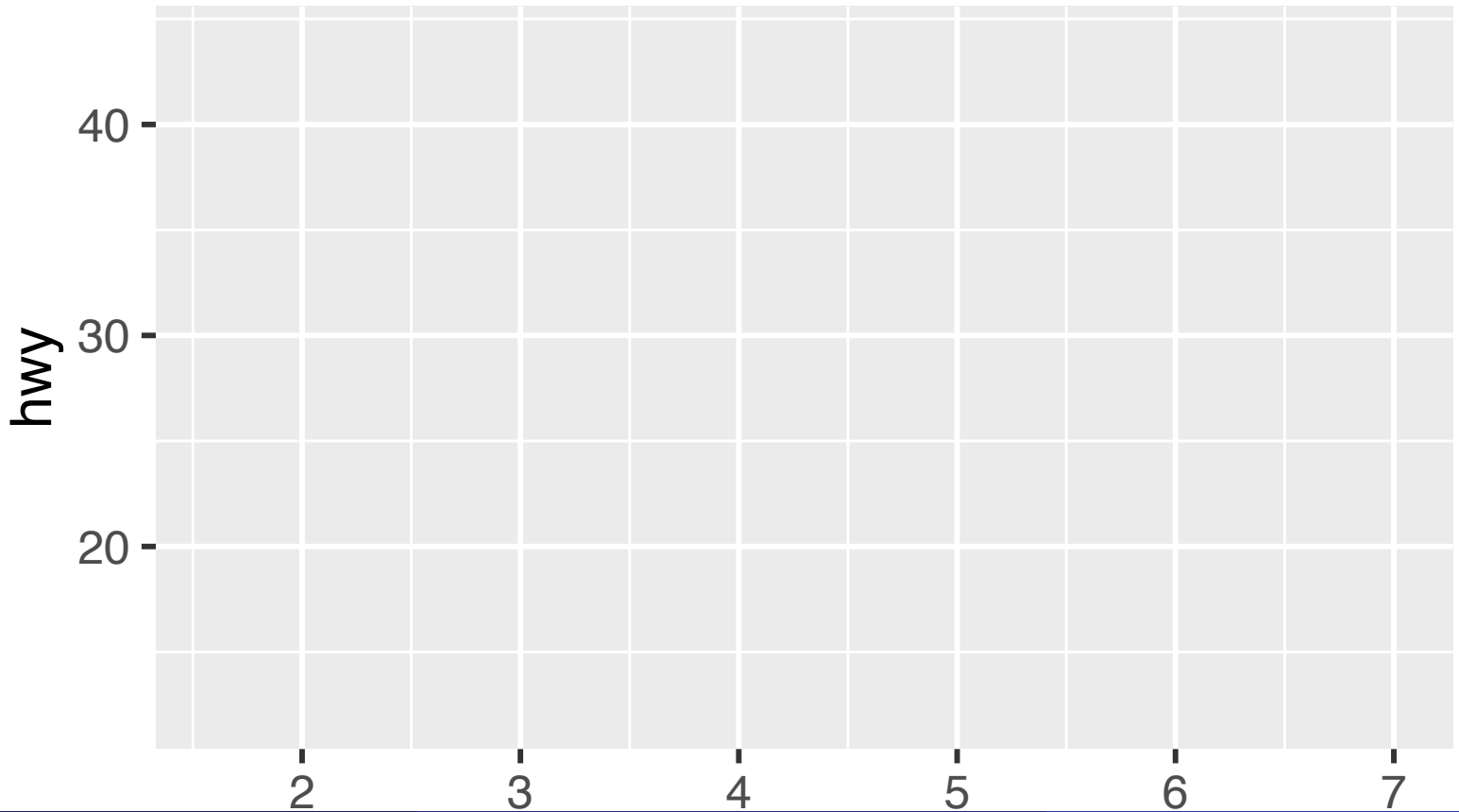
We can create a blank plot with something like this.

```
ggplot(mpg)
```

Aesthetics

aes says what variable maps to what aesthetic property e.g. colour, or position on an axis. It still doesn't have any *layers*. (Notice it is allowed to put aes inside ggplot.)

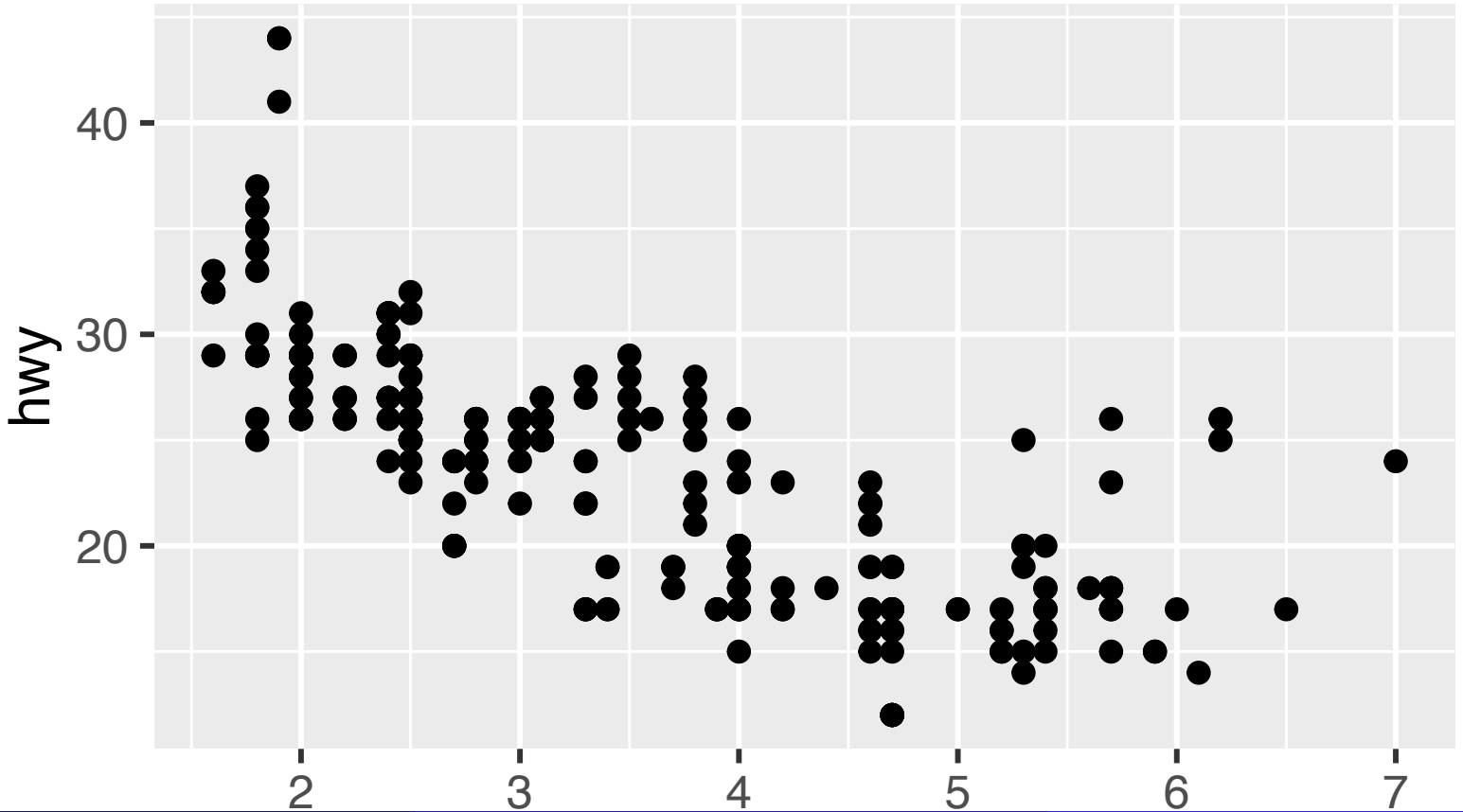
```
ggplot(mpg, aes(x=displ, y=hwy))
```



geoms

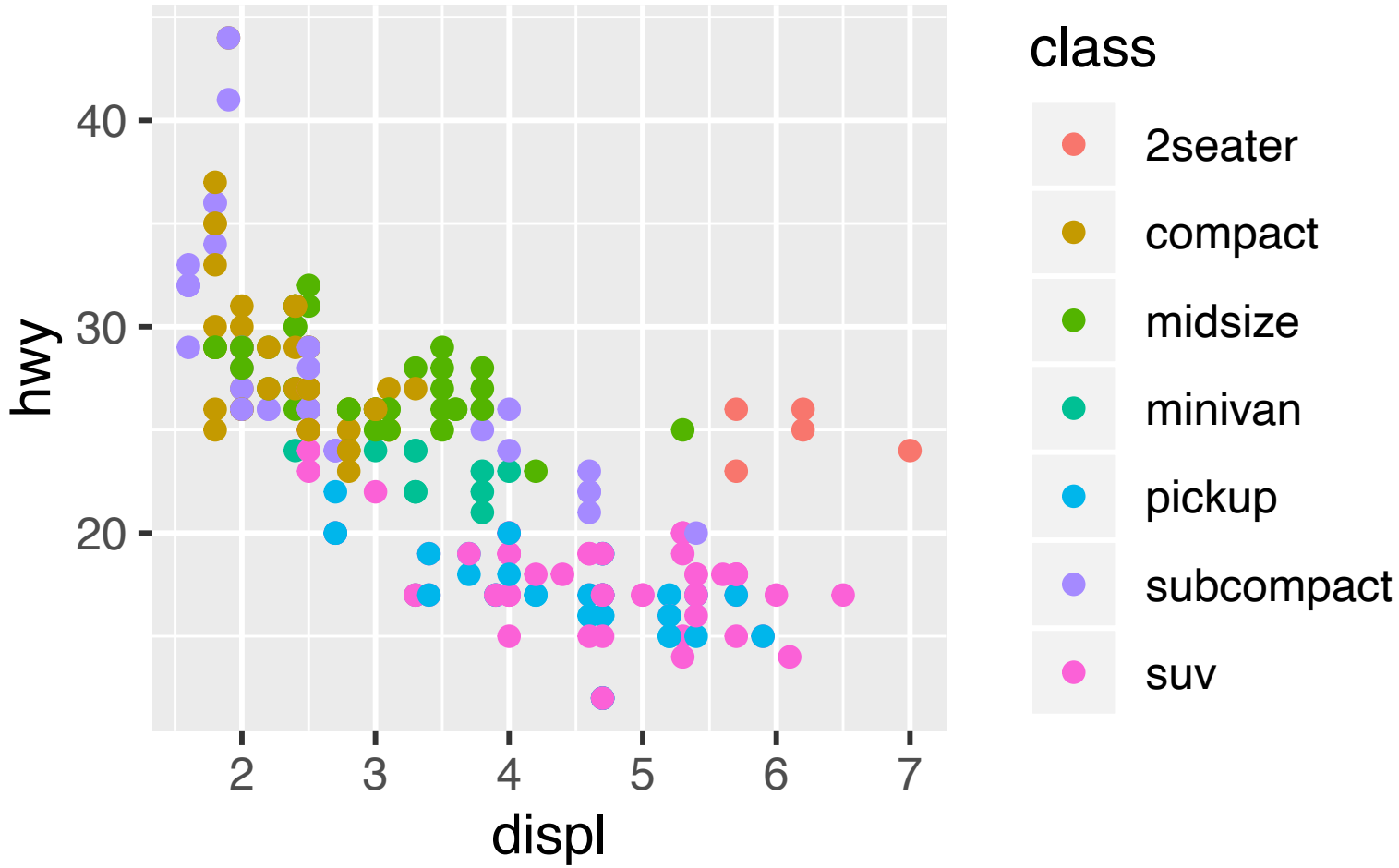
A *geom* is a way of translating a value to marks in the plot. We write + geom, and **that adds a layer**.

```
ggplot(data = dt) +
  geom_point(mapping = aes(x=displ, y=hwy))
```



Adding an extra aesthetic

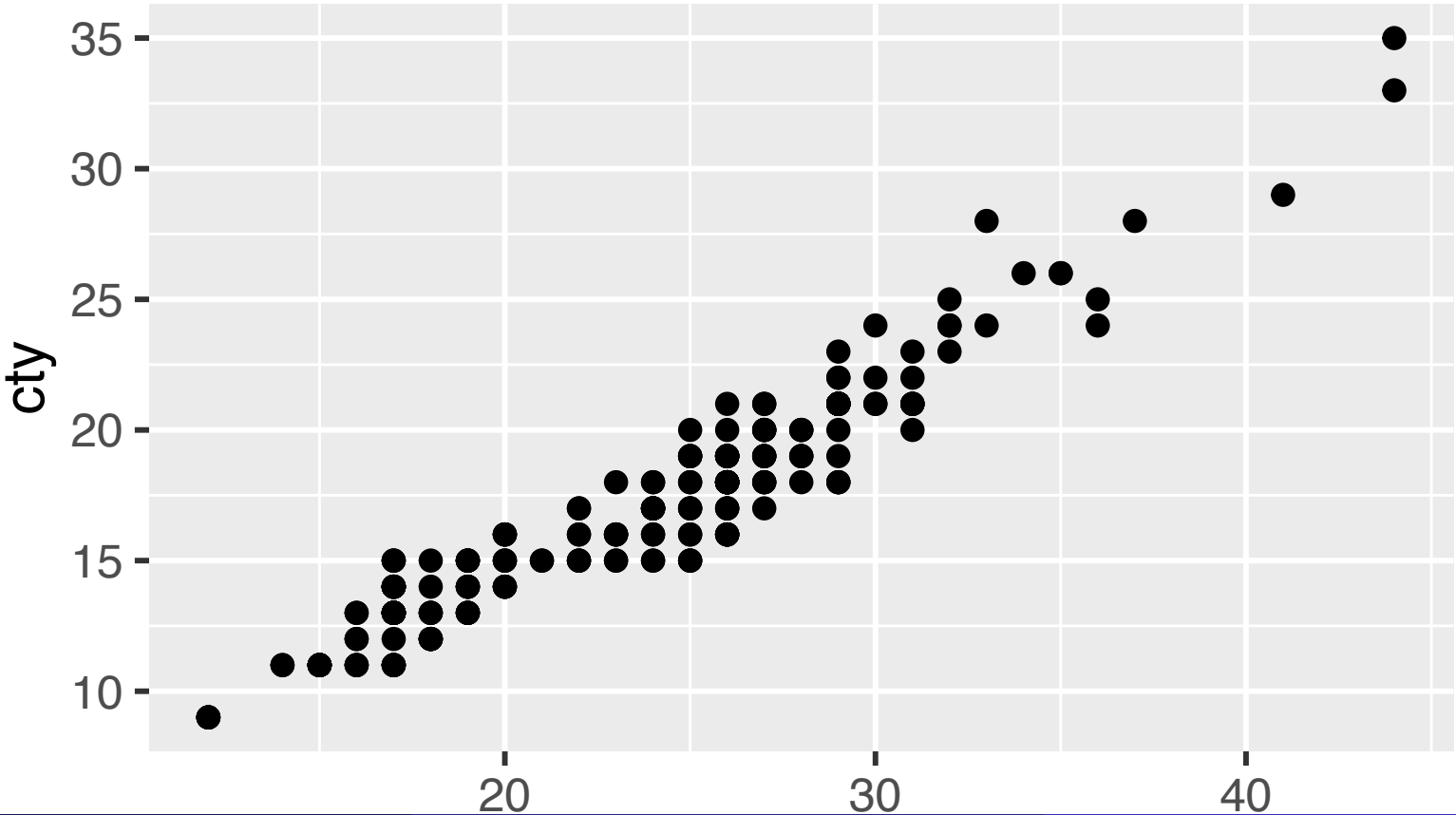
```
ggplot(data = dt) +
  geom_point(mapping = aes(x=displ, y=hwy, colour=class))
```



More geoms

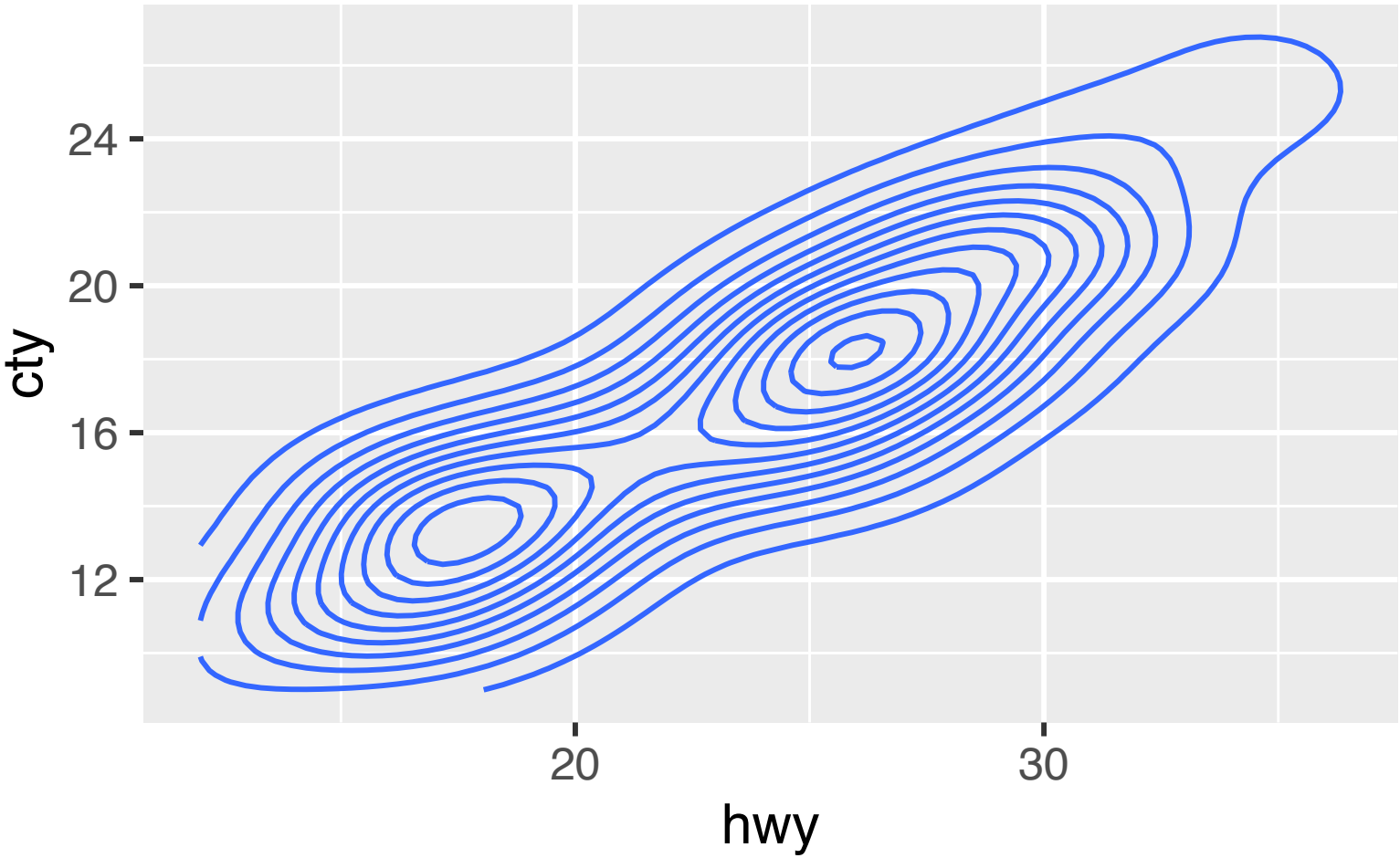
There are several different geoms. We already saw `geom_point`. Next, we'll look at how we can change appearance.

```
ggplot(mpg, aes(x=hwy, y=cty)) +
  geom_point()
```



More geoms

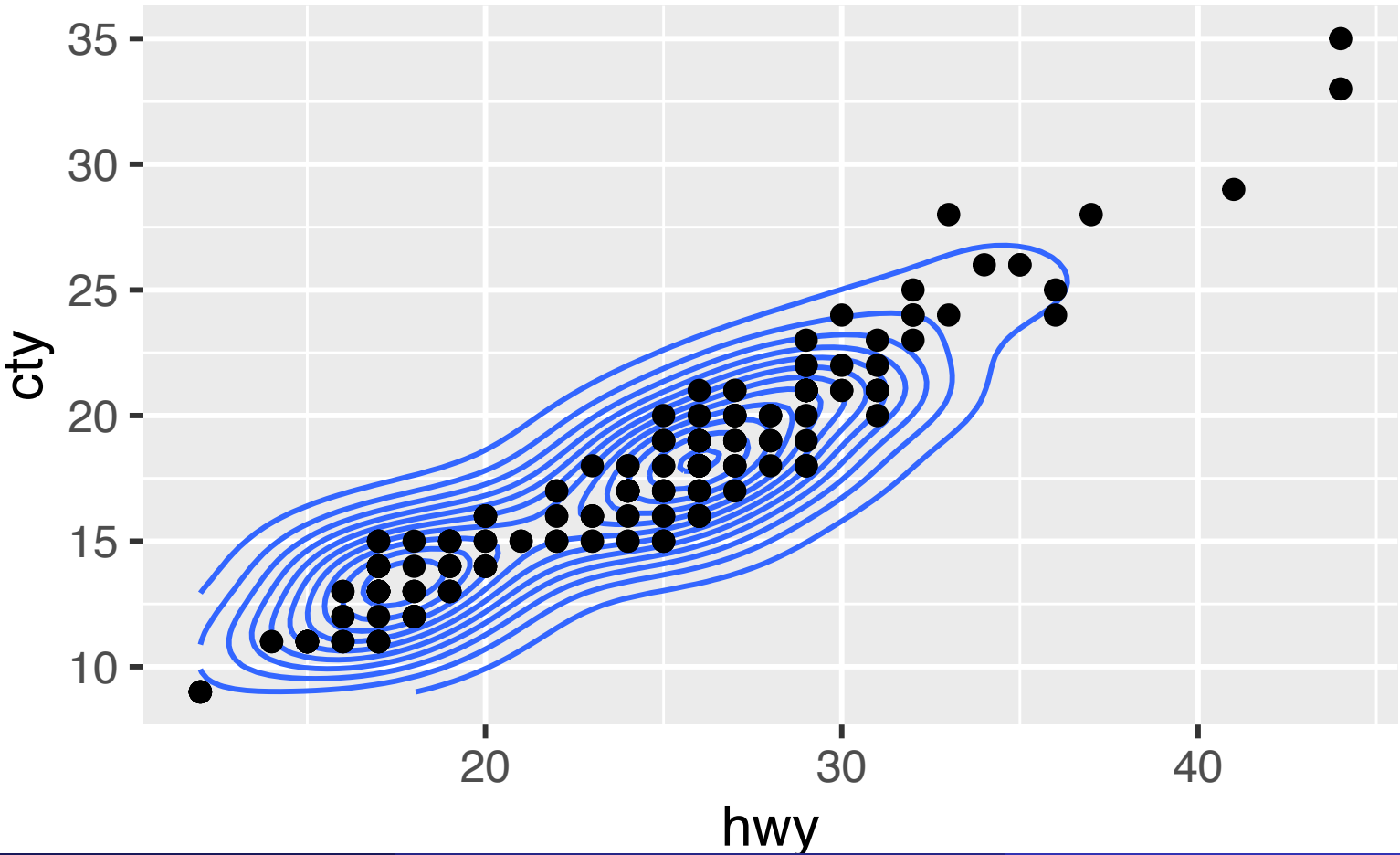
```
ggplot(mpg, aes(x=hwy, y=cty)) +
  geom_density2d()
```



Multiple geoms

We can just write + to add multiple geoms to a plot (i.e. multiple layers).

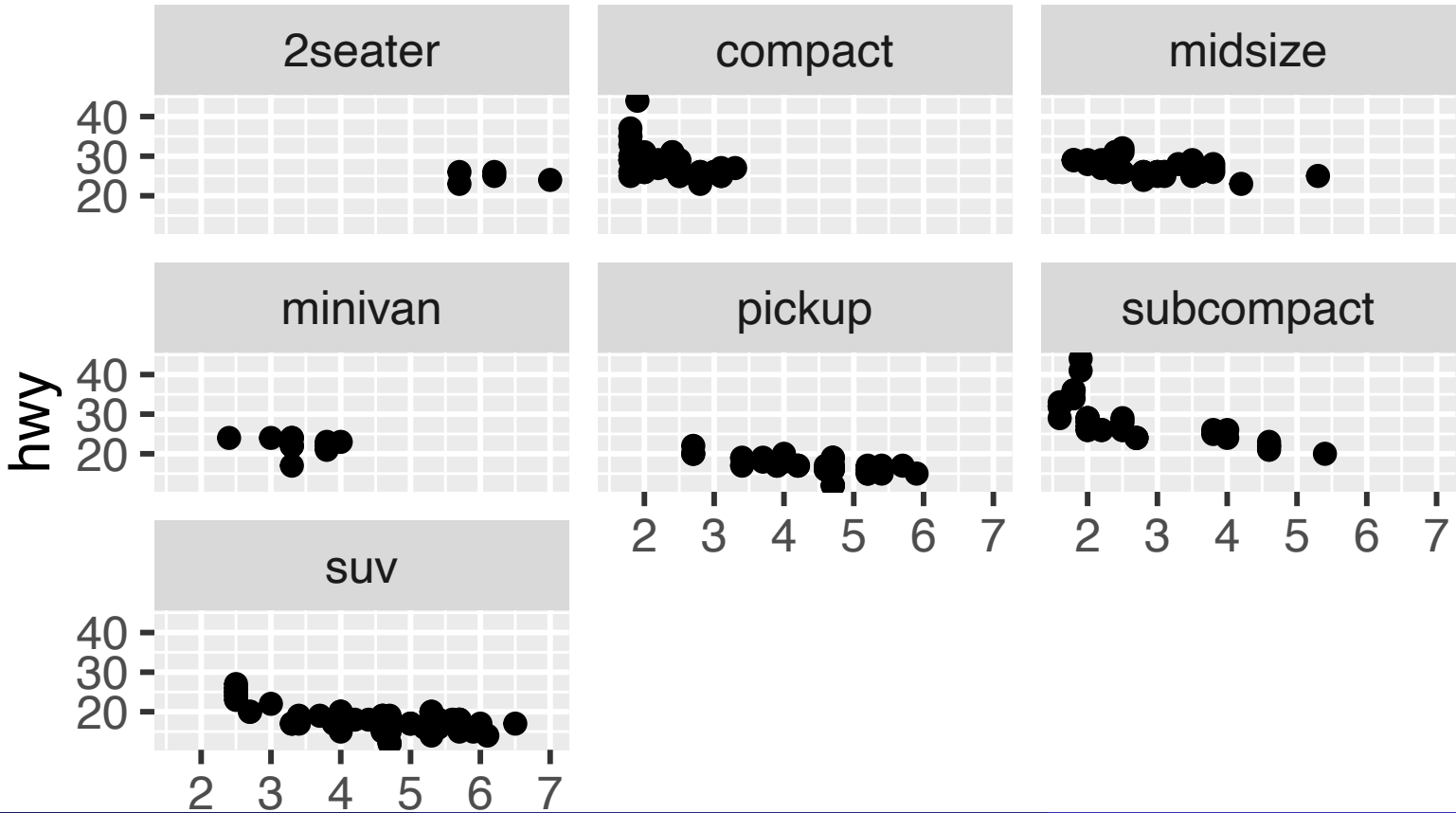
```
ggplot(mpg, aes(x=hwy, y=cty)) +
  geom_density2d() + geom_point()
```



Faceting

Maybe a nicer way is instead to split the data into one graph per class.

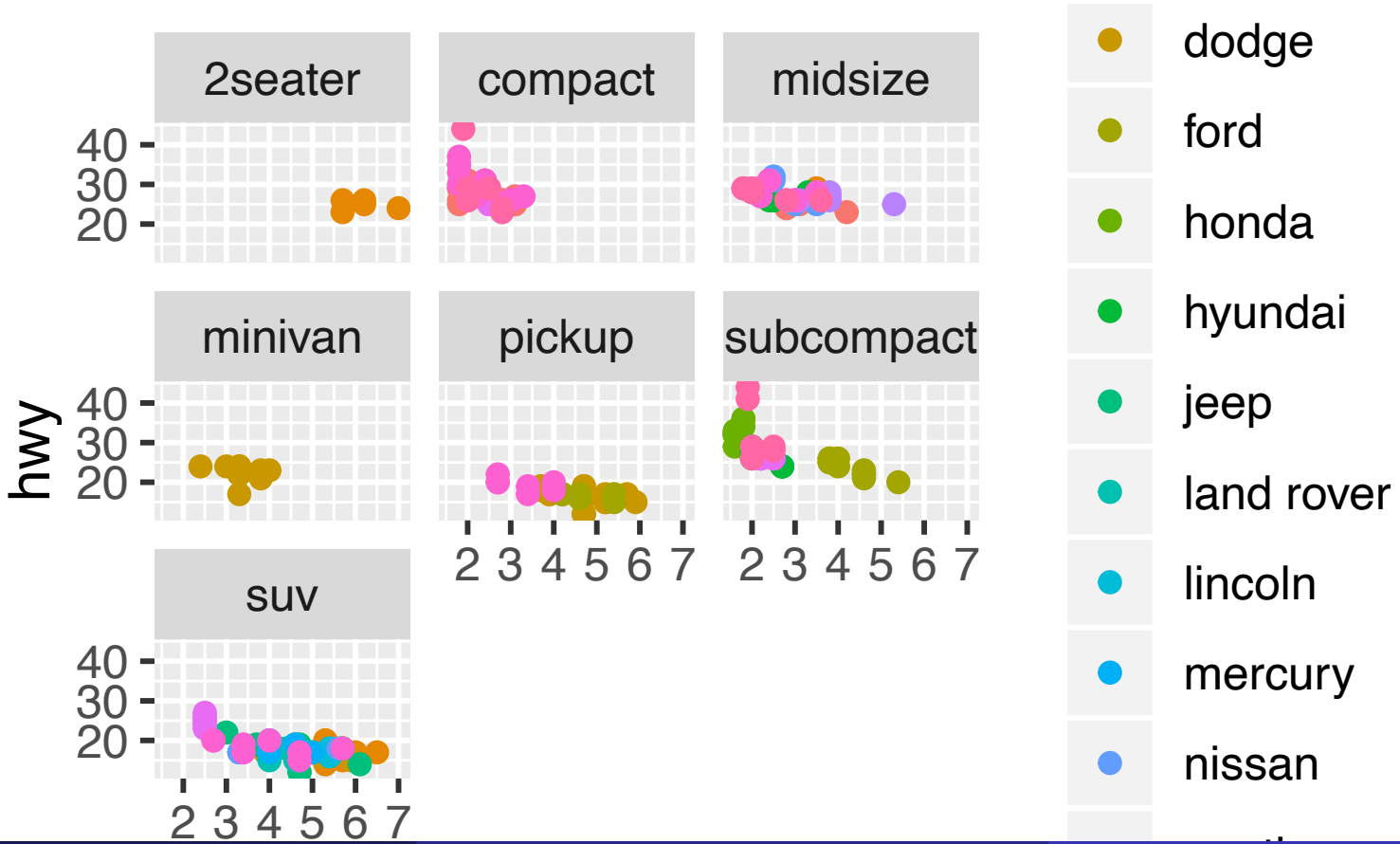
```
ggplot(data = dt) +
  geom_point(mapping = aes(x=displ, y=hwy)) +
  facet_wrap(~class) # notice tilde ~ for a *formula*
```



Adding another variable

Another variable is `manufacturer`. We are now showing 4 variables.

```
ggplot(data = dt) +
  geom_point(mapping=aes(x=displ, y=hwy, colour=manufacturer)) +
  facet_wrap(~class)
```



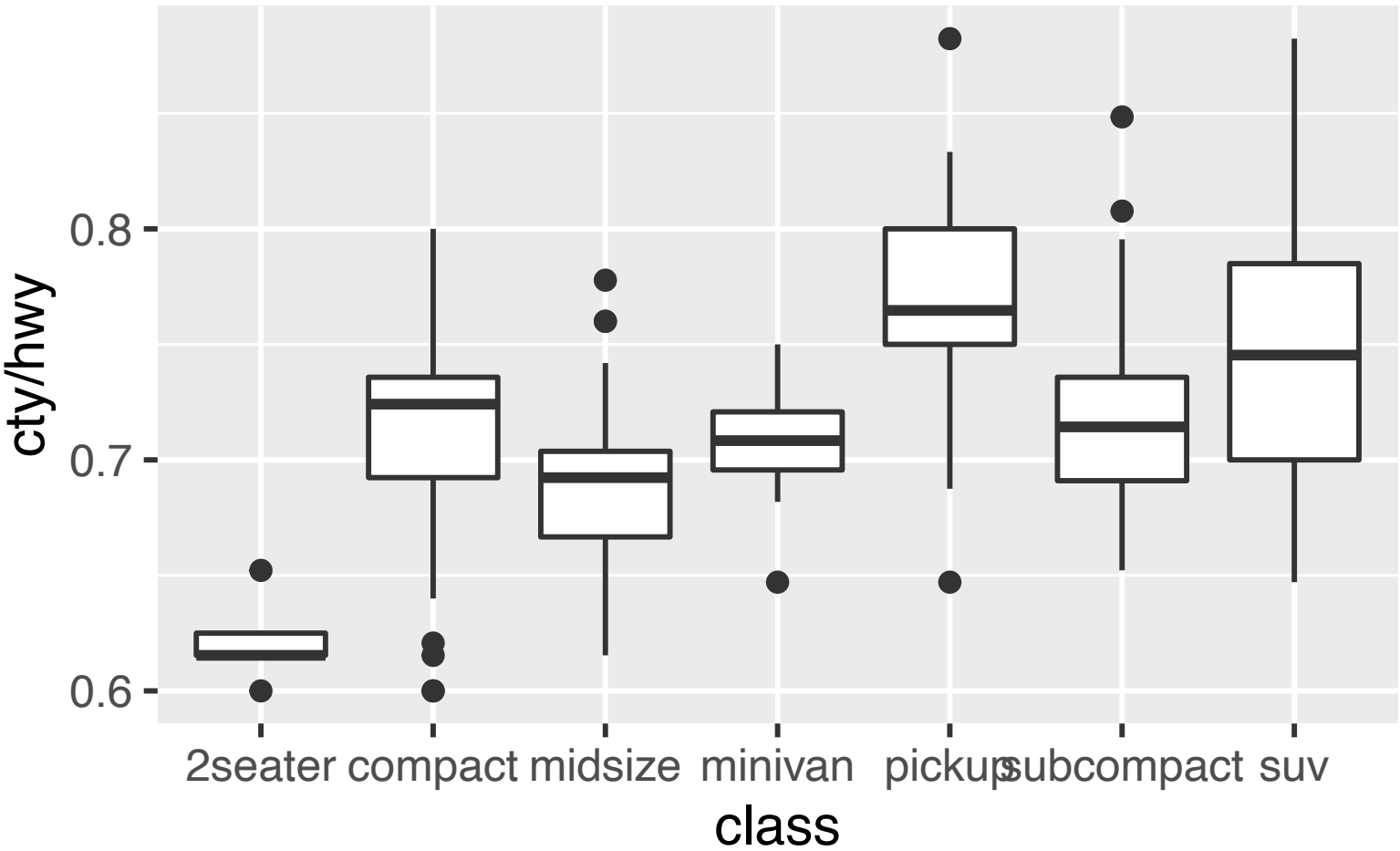
New variables

We can create a new variable inside the `aes` call.

New variables

Which car types emphasise city fuel efficiency over highway fuel efficiency?

```
ggplot(mpg, aes(x=class, y=cty/hwy)) +
  geom_boxplot()
```

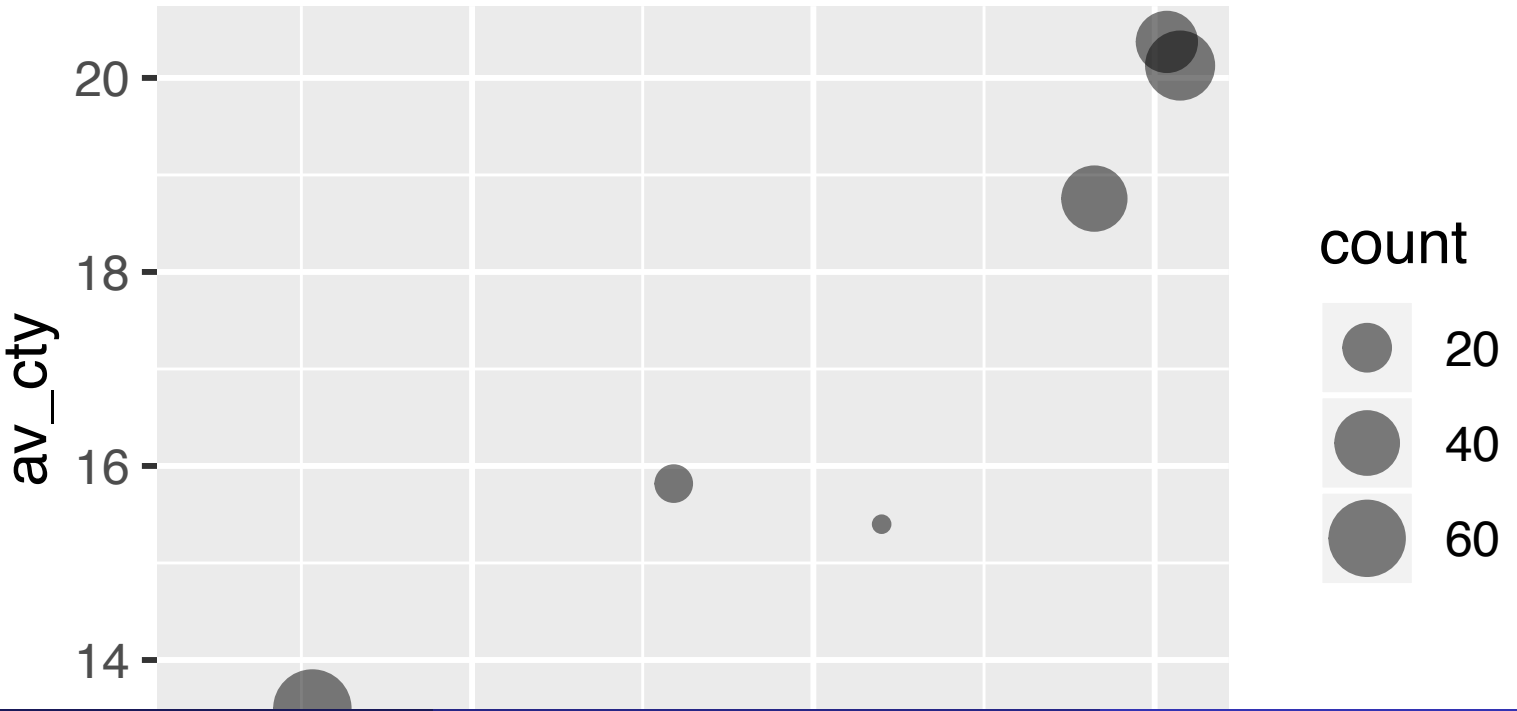


dplyr with ggplot

We can pipe the output of some dplyr manipulation straight into ggplot and then use aesthetics and geom commands to refine the plot.

dplyr with ggplot

```
mpg %>% group_by(class) %>%
  summarise(count=n(),
    av_hwy=mean(hwy),
    av_cty=mean(cty)) %>%
  ggplot(mapping = aes(x=av_hwy, y=av_cty)) +
  geom_point(aes(size=count), alpha=0.5)
```



Collision modifiers

These are methods of preventing marks (e.g. dots) from overlapping with each other.

- `dodge` ("smart" displacement of dots)
- `jitter` (random displacement of dots)
- `nudge` (manual displacement of dots)

There's an example with `nudge` in Assignment 2.

More references

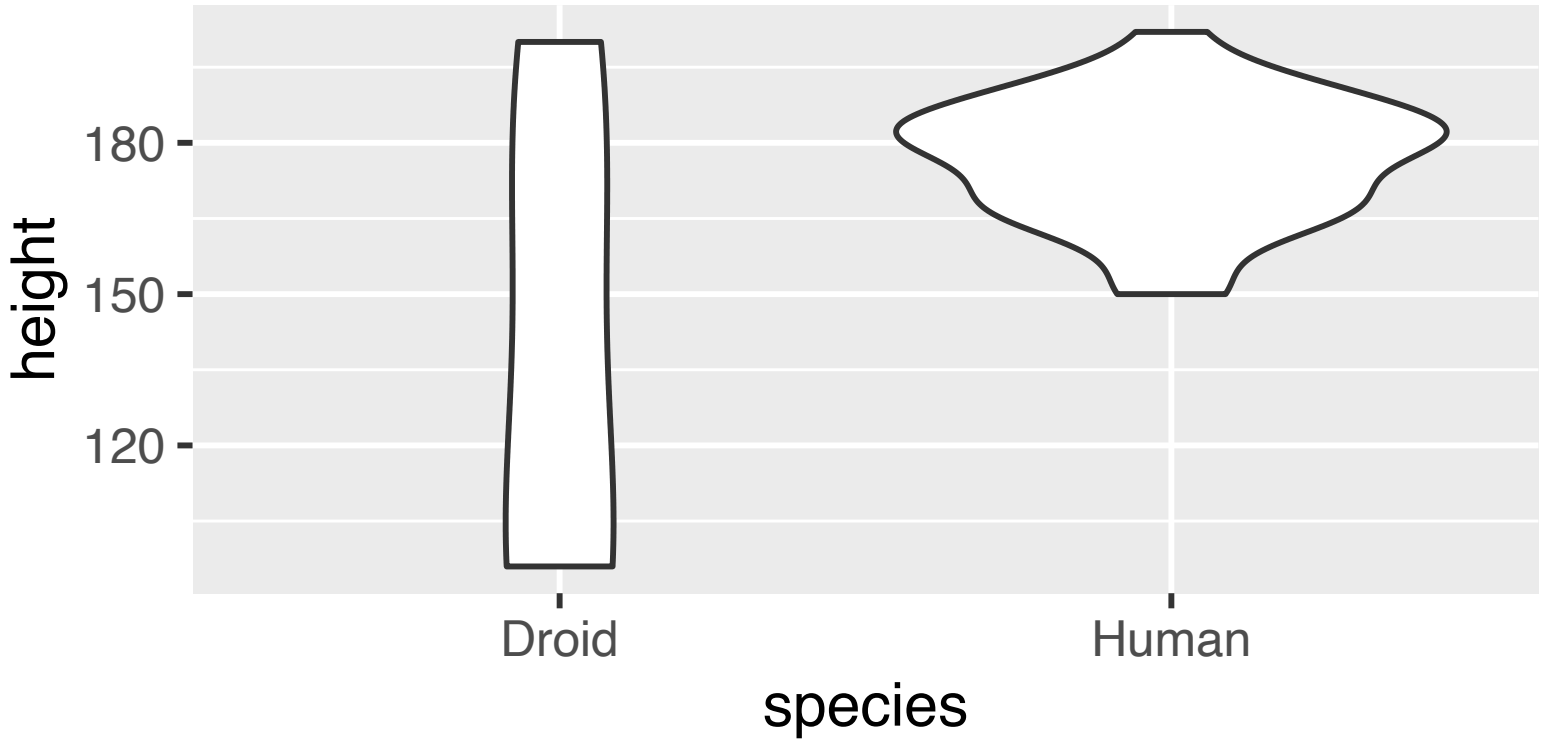
- Manual https://ggplot2.tidyverse.org/reference/
- Cheatsheet https://github.com/rstudio/cheatsheets/blob/master/data-visualization-2.1.pdf
- BBC using R for data journalism with a "house style" https://medium.com/bbc-visual-and-data-journalism/how-the-bbc-visual-and-data-journalism-team-works-with-graphics-in-r-ed0b35693535

Exercises

Here are a few plots from well-known datasets. The exercise is to reproduce them using `ggplot`

Exercise 1

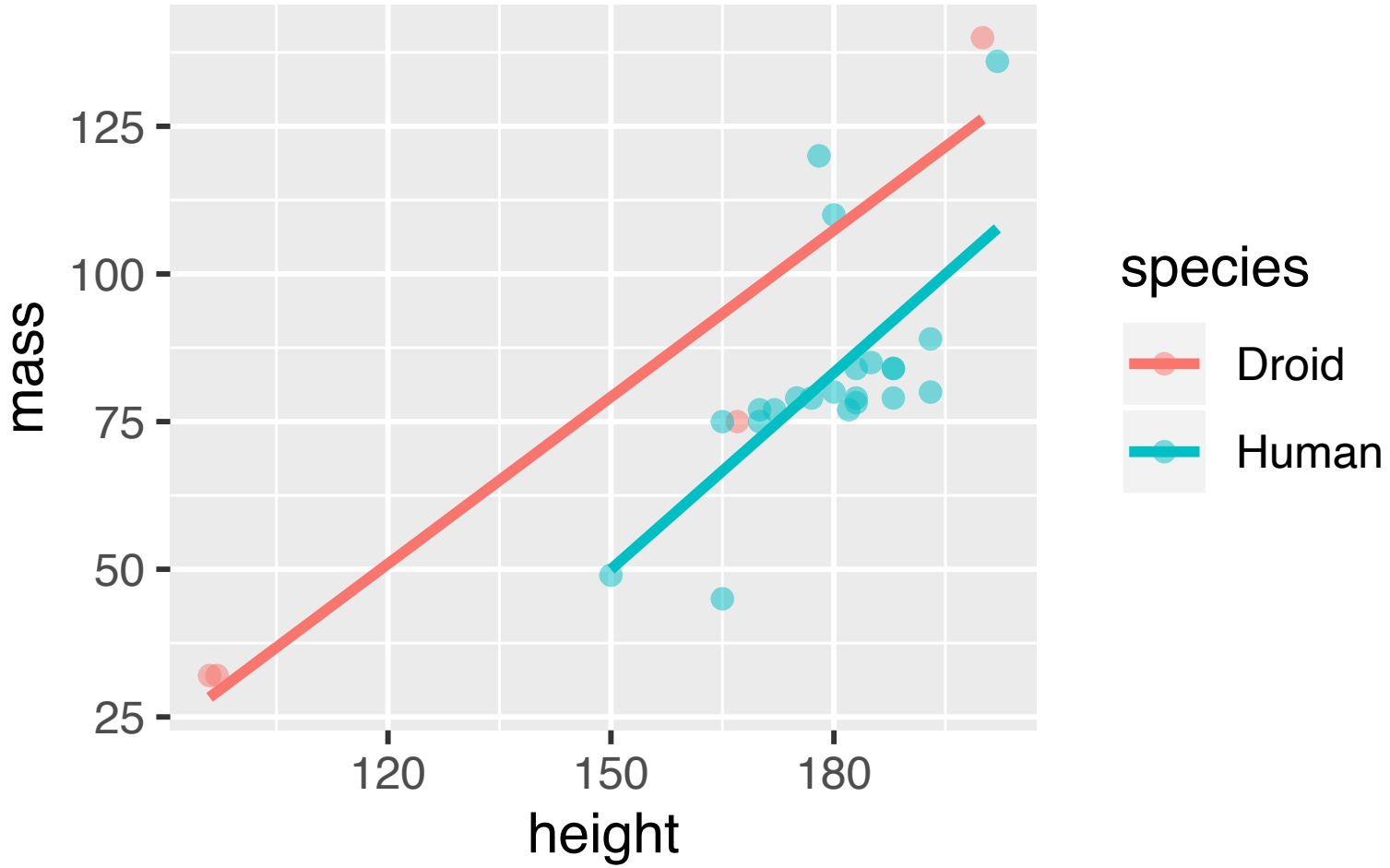
Character height in Star Wars using a "violin plot". A violin plot is like a sideways, smoothed histogram, or like a box plot.
The dataset is `dplyr::starwars`.



Exercise 2

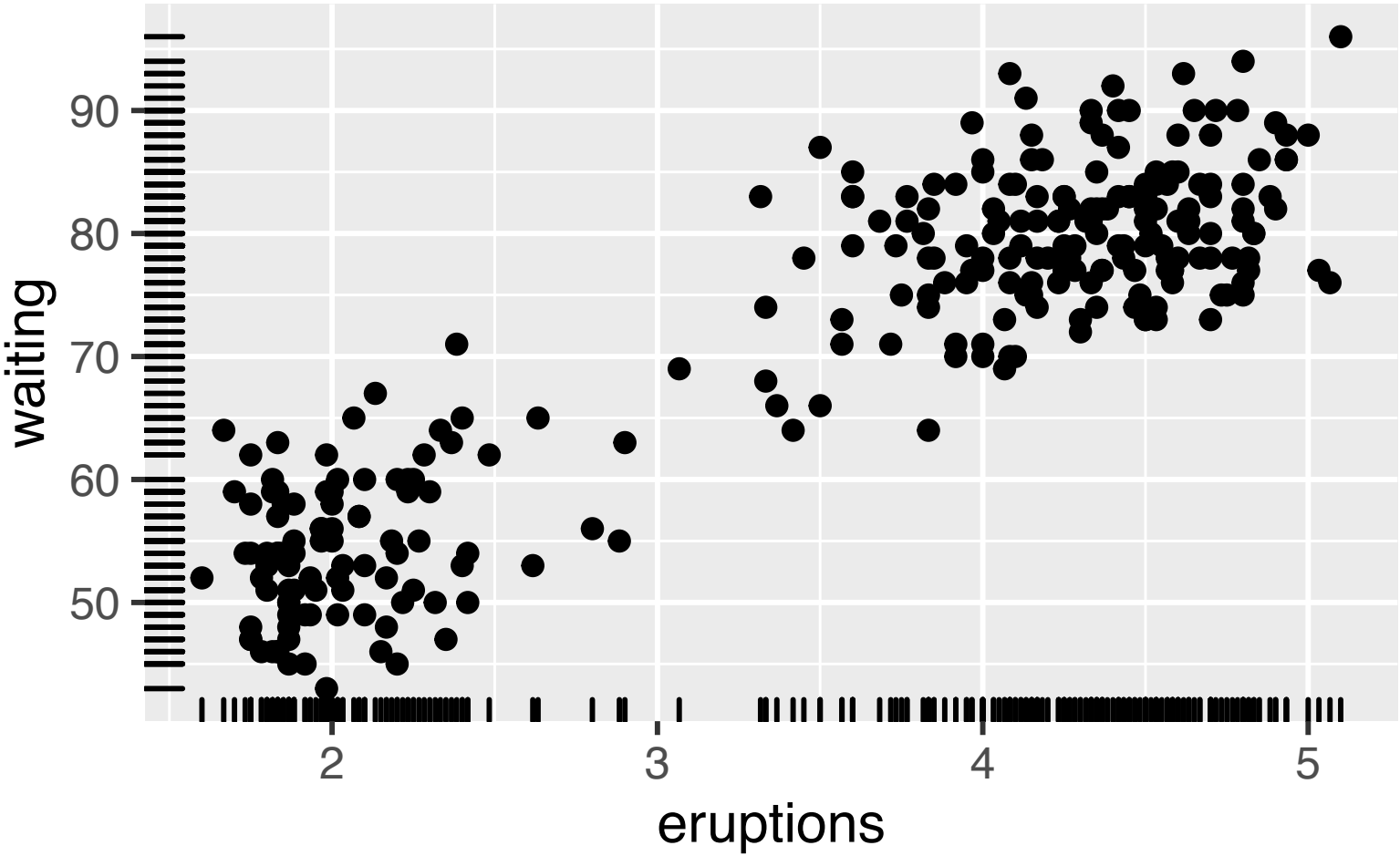
Character height and mass in Star Wars, with trendlines.

```
## Warning: Removed 14 rows containing non-finite values (stat
## Warning: Removed 14 rows containing missing values (geom_po
```



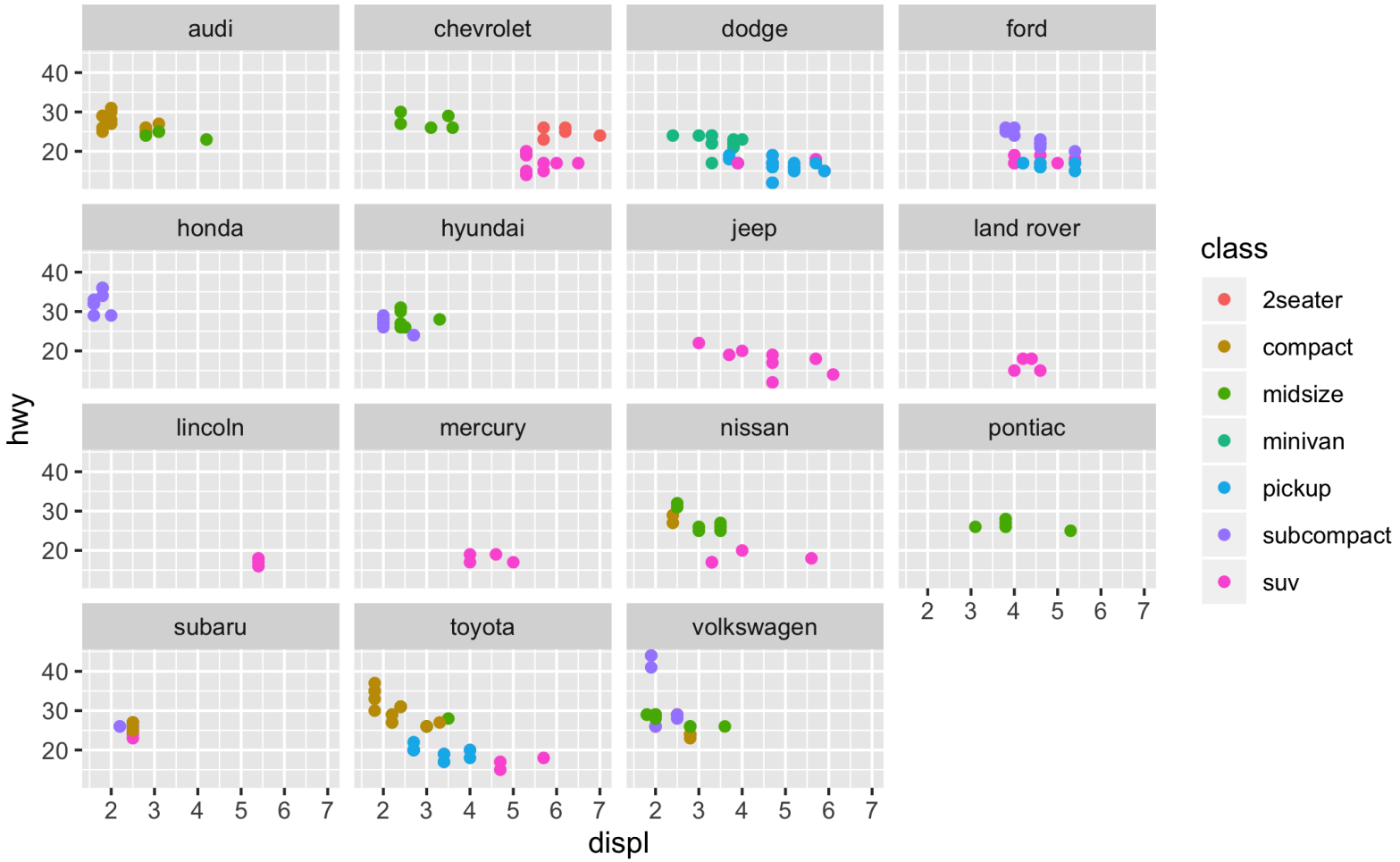
Exercise 3

Eruption length versus waiting time at the Old Faithful geyser. The dataset is `faithful`, built-in to R as a `data.frame`. Recall we can use `as_tibble` to convert to a tibble. Hint: this is called a "rug plot".



Exercise 4

Like a previous plot, but now faceting `manufacturer` and showing `class` as colour.



Solutions

Exercise 1

```
s = dplyr::starwars
s %>% select(height, mass, species) %>%
  filter(height != "na.rm") %>%
  filter(species %in% c("Human", "Droid")) %>%
  ggplot(mapping=aes(x=species, y=height)) + geom_violin()
```

Exercise 2

```
s %>% filter(species == "Human" | species == "Droid") %>%
  ggplot(aes(x=height, y=mass, color=species)) +
  geom_point(alpha=0.5) + geom_smooth(method=lm, se=FALSE)
```

Exercise 3

```
f = as_tibble(faithful)
f %>% ggplot(aes(x=eruptions, y=waiting))
  + geom_point() + geom_rug()
```

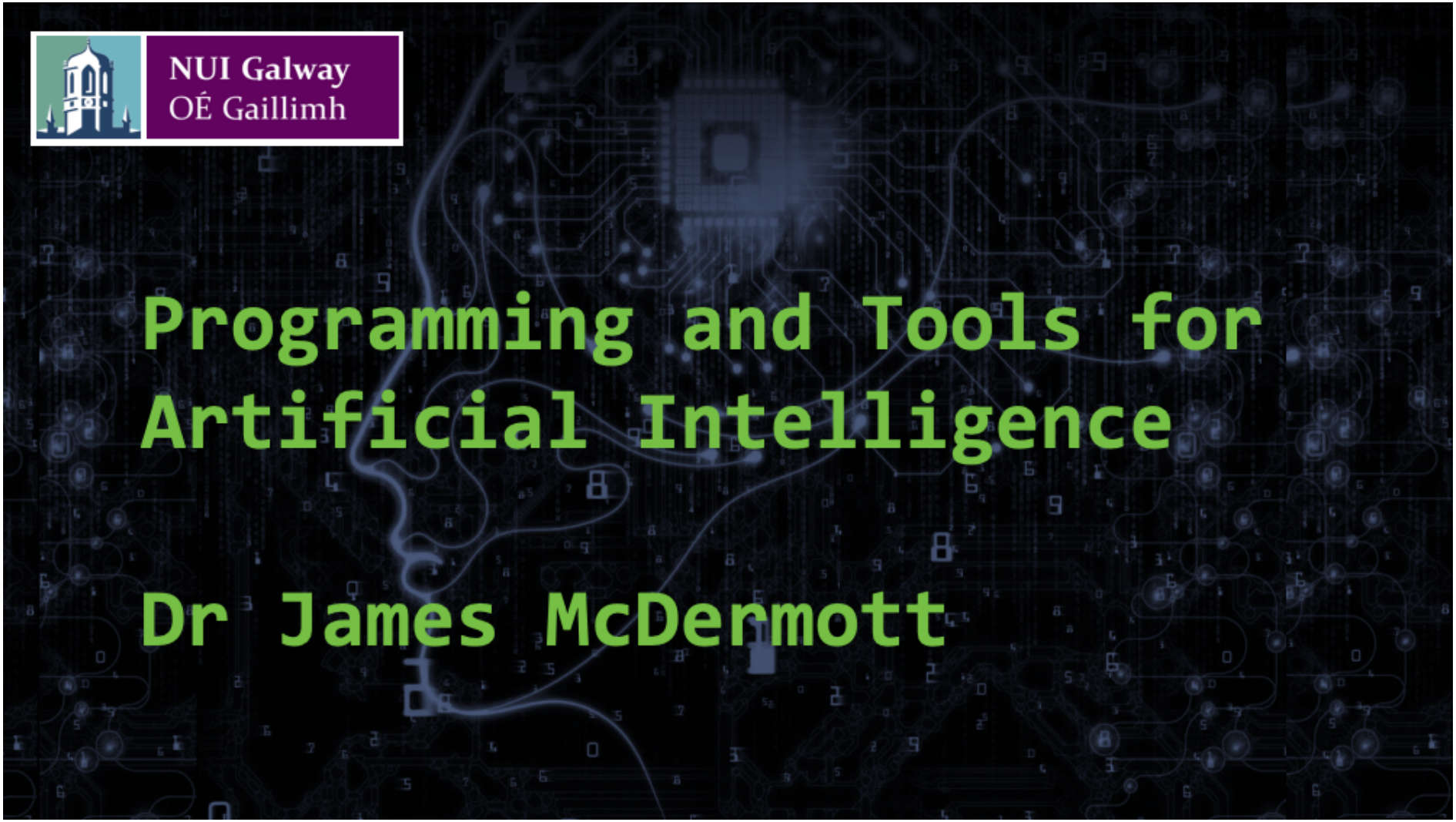
Exercise 4

```
dt <- ggplot2::mpg
ggplot(data = dt) +
  geom_point(mapping = aes(x=displ, y=hwy, colour=class)) +
  facet_wrap(~manufacturer)
ggsave("img/R_mpg_displ_hwy_manu_class.png")
```

Statistics in R

James McDermott

NUI Galway



Statistics in R

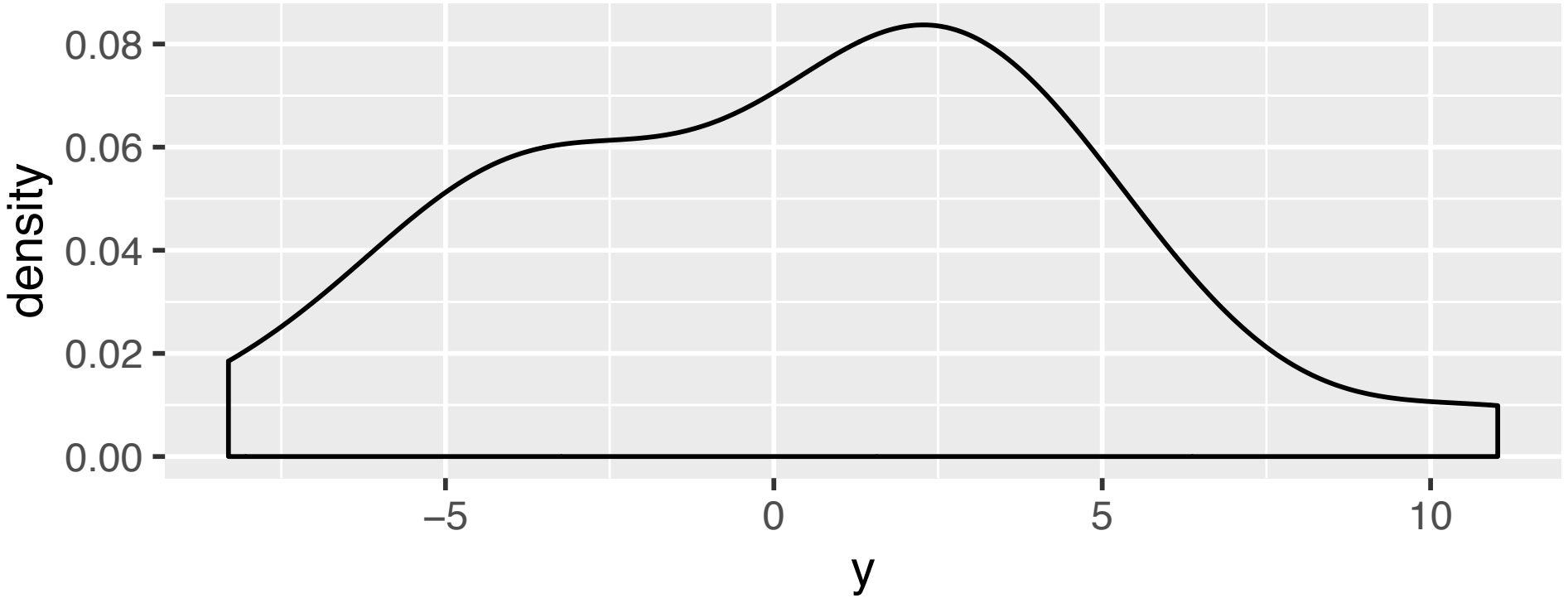
Load Tidyverse as usual

```
library(tidyverse)
```

```
## -- Attaching packages ---------------------------------------
## v ggplot2 3.2.1     v purrr   0.3.2
## v tibble  2.1.3     v dplyr   0.8.3
## v tidyr   1.0.0     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0
## -- Conflicts ------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Random numbers

```
x1 <- runif(20, min=0, max=2) # random uniform with bounds
x2 <- rnorm(20) # random normal with mean 0, sd 1
y <- x1 + x2 * rnorm(20, mean=5, sd=2)
ggplot(tibble(y), aes(x=y)) + geom_density()
```



Basic statistics

```
for (f in c(min, max, mean, median, sd, var, IQR, mad)) {
  print(f(y))
}
```

```
## [1] -8.304924
## [1] 11.02246
## [1] 0.4374315
## [1] 1.325795
## [1] 4.580337
## [1] 20.97949
## [1] 5.971987
## [1] 4.859902
```

More data summaries

```
for (f in c(range, quantile, summary, fivenum)) {
  print(f(y))
}
```

```
## [1] -8.304924 11.022464
##         0%        25%        50%        75%       100%
## -8.304924 -3.058389  1.325795  2.913598 11.022464
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -8.3049 -3.0584  1.3258  0.4374  2.9136 11.0225
## [1] -8.304924 -3.309386  1.325795  2.951253 11.022464
```

Correlations

```
cor(x1, y) # get the correlation
```

```
## [1] 0.295735
```

Correlations: statistical test

```r
cor.test(x1, y) # run a test
```

```
##
##  Pearson's product-moment correlation
##
## data:  x1 and y
## t = 1.3134, df = 18, p-value = 0.2055
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  -0.1688880  0.6528217
## sample estimates:
##      cor
## 0.295735
```

Correlations: using results

```
res = cor.test(x1, y) # save the result
names(res) # see result structure
```

```
## [1] "statistic"   "parameter"   "p.value"     "estimate"
## [6] "alternative" "method"      "data.name"   "conf.int"
```

```
R = res['statistic'] # extract values...
p = res['p.value'] # ...from the result
```

Null hypothesis significance testing

Independent 2-sample 2-sided t-test

Test whether difference in means is different from 0

```
t.test(x1, y)
```

```
##
##  Welch Two Sample t-test
##
## data:  x1 and y
## t = 0.61544, df = 19.607, p-value = 0.5453
## alternative hypothesis: true difference in means is not equ
## 95 percent confidence interval:
##  -1.520858  2.791563
## sample estimates:
## mean of x mean of y
## 1.0727840 0.4374315
```

More t-tests

The `t.test` function also has options for:

- 1-sided tests
- paired tests
- 1-sample tests.

Regression models

The `lm` (linear model) function and variants are used for regression.

```
df = tibble(x1, x2, y)
head(df)
```

```
## # A tibble: 6 x 3
##       x1      x2      y
##    <dbl>   <dbl> <dbl>
## 1 1.94   -0.220   1.07
## 2 1.62    0.702   5.72
## 3 0.514   0.505   2.88
## 4 1.27    0.384   4.29
## 5 0.475  -0.964  -4.36
## 6 0.132  -0.512  -2.81
```

Formulas

R provides a special formula syntax involving the tilde `~`. It's used to specify a regression model. The left-hand side is the dependent variable, `y`. The right-hand side gives the independent variables, interactions, and transformations. So, `~` means something like "is modelled as".

```
y ~ x1 + x2
```

This says: run the formula $y = a + b_1 x_1 + b_2 x_2$

Using a formula in a regression

```
res <- lm(y ~ x1 + x2, data=df)
summary(res) # show results
```

```
##
## Call:
## lm(formula = y ~ x1 + x2, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0111 -0.6745  0.2500  0.6999  2.7058
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.7509     0.7119  -1.055   0.3063
## x1            1.2890     0.5867   2.197   0.0422 *
## x2            4.5939     0.3722  12.343 6.53e-10 ***
## ---
```

Formulas with interaction

If we changed + to $*$, we would add the interaction effect, ie we would run the formula

$y = a + b_1 x_1 + b_2 x_2 + b_{12} x_1 x_2$

Use `?formula` for more on this special syntax.

Formulas with interaction

```
res <- lm(y ~ x1 * x2, data=df)
summary(res) # show results
```

```
##
## Call:
## lm(formula = y ~ x1 * x2, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.6828 -0.4217  0.2195  0.7291  2.2952
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.6443     0.7051  -0.914   0.3744
## x1            1.2612     0.5774   2.184   0.0442 *
## x2            5.7138     0.9635   5.930 2.11e-05 ***
## x1:x2        -1.0724     0.8535  -1.257   0.2270
```

Formulas with transformation

We could also use transformations. For example:

```
res <- lm(y ~ x1 + log(x2), data=df)
```

```
## Warning in log(x2): NaNs produced
```

```
summary(res) # show results
```

```
##
## Call:
## lm(formula = y ~ x1 + log(x2), data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9823 -1.7985 -0.0445  0.8793  4.2214
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   4.4794     2.0797   2.154   0.0747 .
## x1            0.8535     1.6607   0.514   0.6257
```

One-way analysis of variance (ANOVA)

Like t-test for multiple groups, again using a formula.

```
res = aov(height ~ gender * species, data=dplyr::starwars)
summary(res)
```

```
##                 Df Sum Sq Mean Sq F value   Pr(>F)
## gender           3   1674   558.1   8.421 0.000254 ***
## species         34  73457  2160.5  32.599  < 2e-16 ***
## gender:species   2    196    97.9   1.477 0.242539
## Residuals       34   2253    66.3
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
## 13 observations deleted due to missingness
```

Beyond Base R: the `caret` package

- k-nearest neighbours
- Linear regression
- Support vector machines
- Classification/regression trees
- Perceptrons
- Ensembles, including forests, bagging, boosting

https://topepo.github.io/caret

The `caret` package

The main Python competitor is `scikit-learn` which we will study later.

We won't go into detail on ML algorithms in this class.

Further reading

- https://www.statmethods.net/stats/ttest.html
- https://www.statmethods.net/stats/regression.html
- https://www.statmethods.net/stats/anova.html

Exercises

1. In the `mpg` dataset (part of the tidyverse), calculate the mean and standard deviation of the highway fuel efficiency.
2. Using `group_by`, calculate the mean and standard deviation of the highway fuel efficiency per manufacturer.
3. Calculate the correlation between highway fuel efficiency and engine size.
4. What was the average highway fuel efficiency in 1999 and in 2008?
5. Carry out a two-sample independent t-test between highway fuel efficiency in 1999 and 2008 and interpret the result.
6. Carry out a regression on highway fuel efficency by displacement.

Solution 1

```
library(tidyverse)
```

```
mean(mpg$hwy)
```

```
## [1] 23.44017
```

```
sd(mpg$hwy)
```

```
## [1] 5.954643
```

Solution 2

```
mpg %>% group_by(manufacturer) %>%
  summarise(mean=mean(hwy), sd=sd(hwy))
```

```
## # A tibble: 15 x 3
##    manufacturer  mean    sd
##    <chr>        <dbl> <dbl>
##  1 audi          26.4  2.18
##  2 chevrolet     21.9  5.11
##  3 dodge         17.9  3.57
##  4 ford          19.4  3.33
##  5 honda         32.6  2.55
##  6 hyundai       26.9  2.18
##  7 jeep          17.6  3.25
##  8 land rover    16.5  1.73
##  9 lincoln       17    1
## 10 mercury       18    1.15
## 11 nissan        24.6  5.09
```

Solution 3

```
cor(mpg$hwy, mpg$displ)
```

```
## [1] -0.76602
```

Solution 4

```
mpg %>% group_by(year) %>%
  summarise(mean=mean(hwy), sd=sd(hwy))
```

```
## # A tibble: 2 x 3
##    year  mean    sd
##   <int> <dbl> <dbl>
## 1  1999  23.4  6.08
## 2  2008  23.5  5.85
```

Solution 5

```r
mpg1999 <- mpg %>% filter(year == 1999)
mpg2008 <- mpg %>% filter(year == 2008)
t.test(mpg1999$hwy, mpg2008$hwy)
```

```
##
##  Welch Two Sample t-test
##
## data:  mpg1999$hwy and mpg2008$hwy
## t = -0.032864, df = 231.64, p-value = 0.9738
## alternative hypothesis: true difference in means is not equ
## 95 percent confidence interval:
##  -1.562854  1.511572
## sample estimates:
## mean of x mean of y
##  23.42735  23.45299
```

Solution 6

```
res = lm(hwy ~ displ, data=mpg)
summary(res)
```

```
##
## Call:
## lm(formula = hwy ~ displ, data = mpg)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -7.1039 -2.1646 -0.2242  2.0589 15.0105
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  35.6977     0.7204   49.55   <2e-16 ***
## displ        -3.5306     0.1945  -18.15   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 '
```