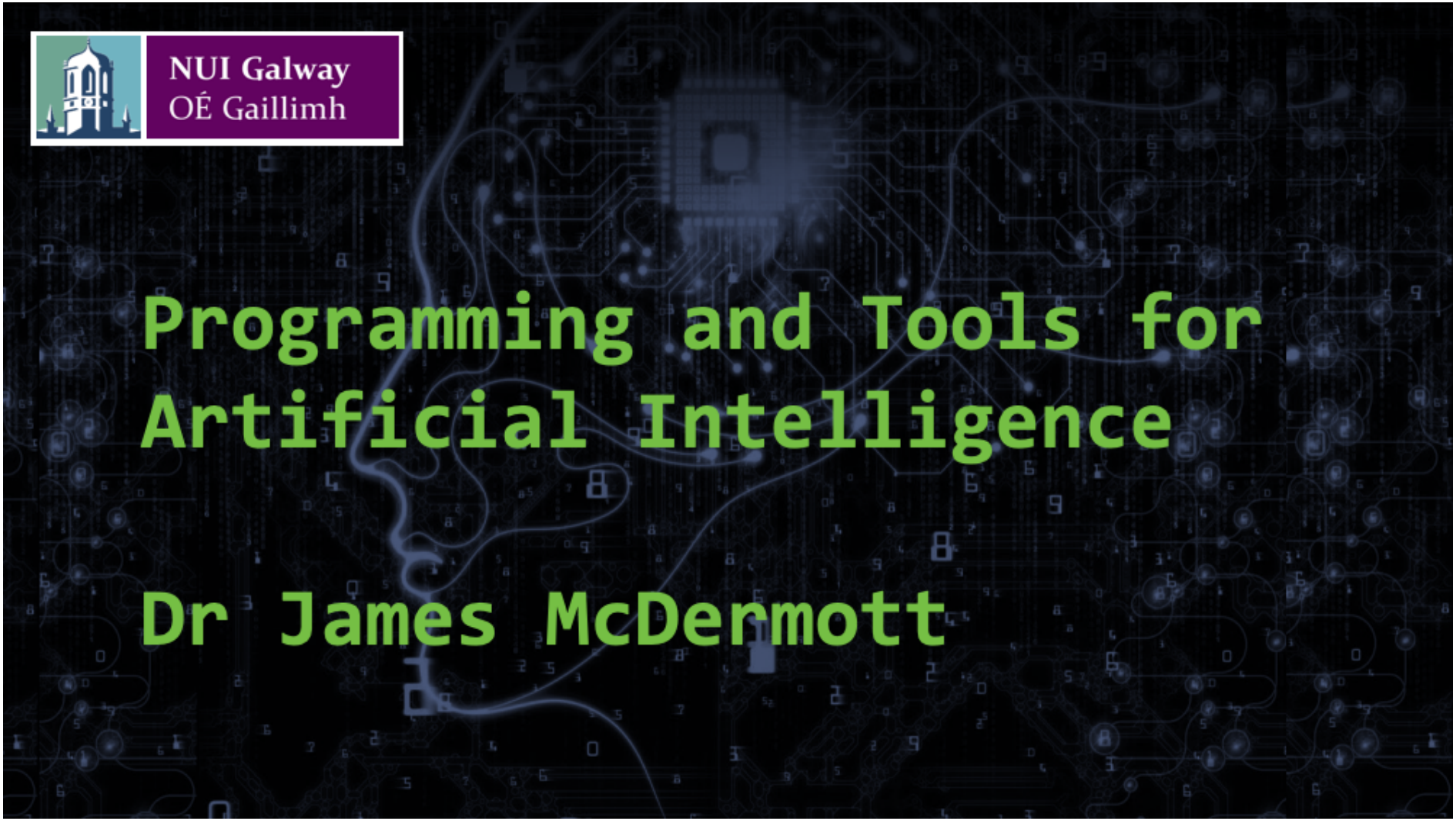
Tidy Data in R

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

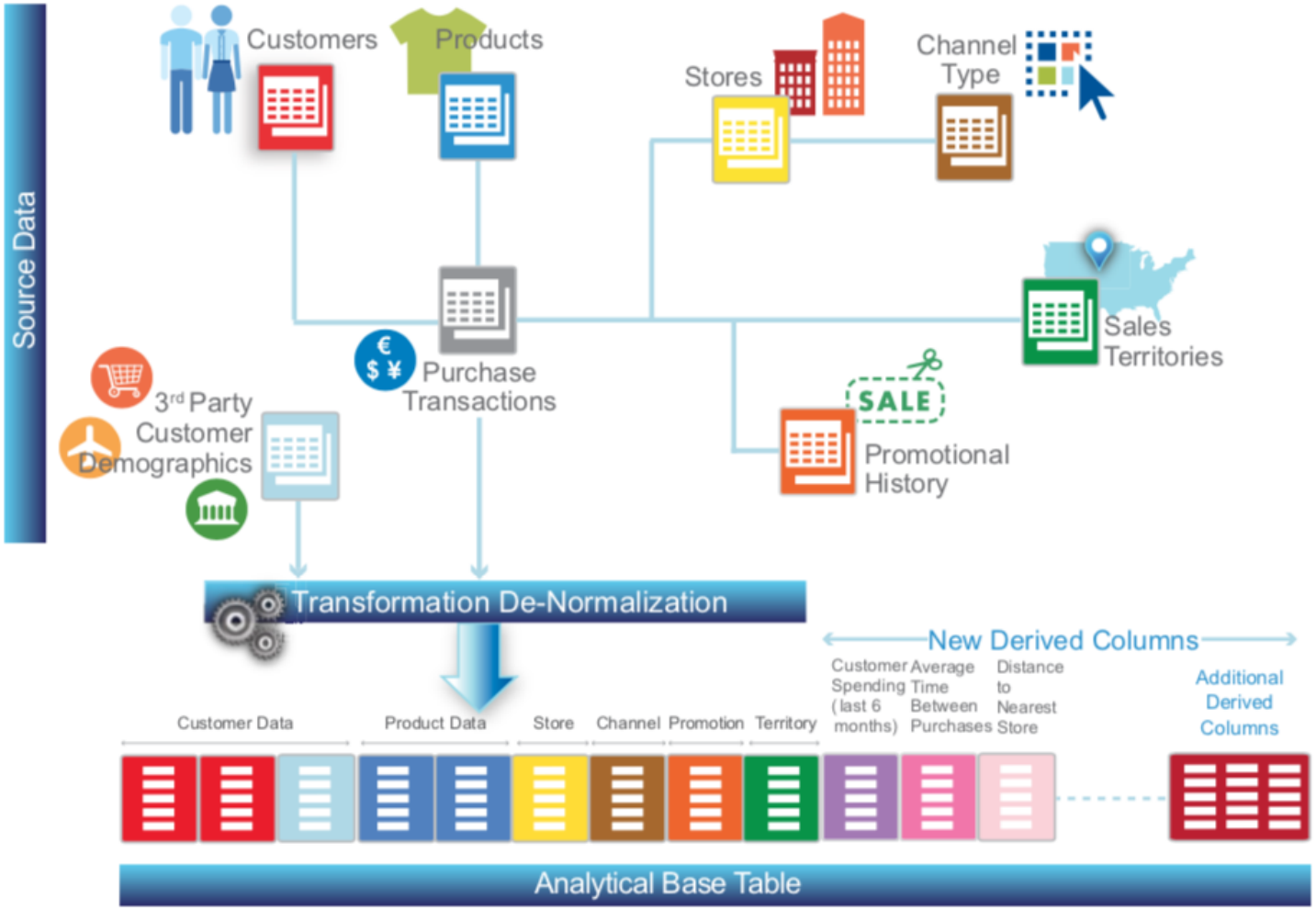
- *Is it?* (See `code/tidy_data.xlsx`)

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

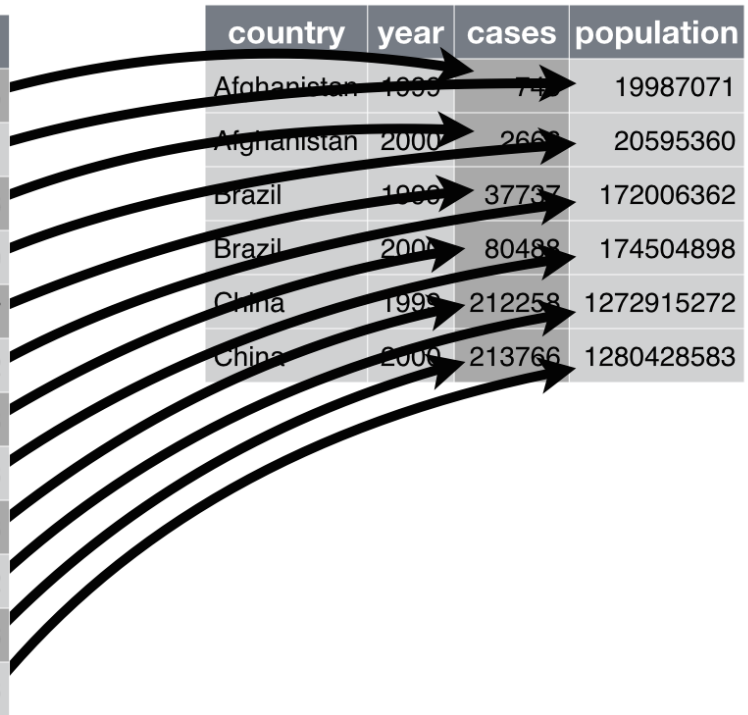
table4

```
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")
```