NBIS

Tidy work in Tidyverse

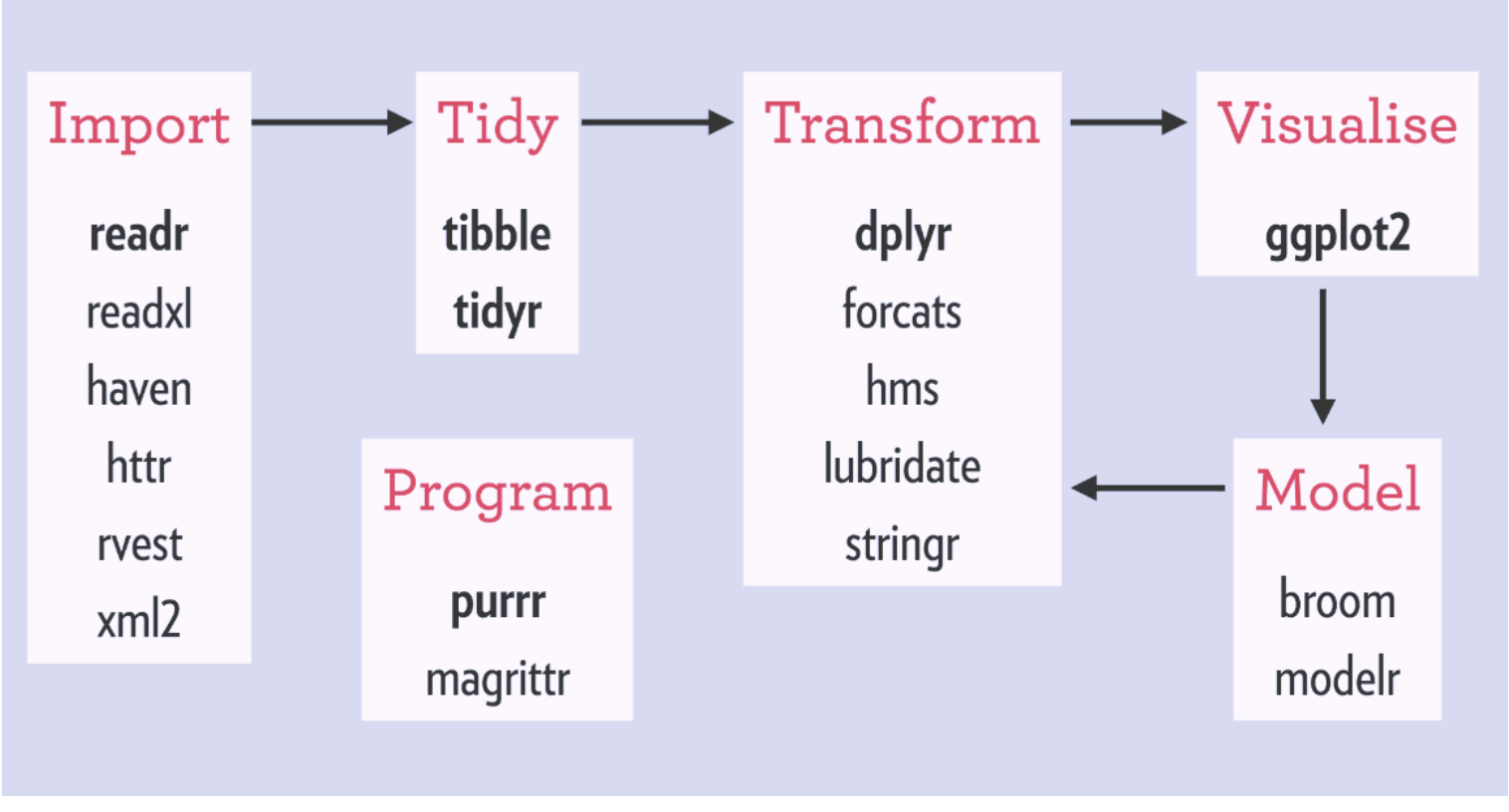
R Foundation for Life Scientists

Marcin Kierczak

Typical Tidyverse Workflow

The tidyverse curse?

> Navigating the balance between base R and the tidyverse is a challenge to learn. -Robert A. Muenchen



source: http://www.storybench.org/getting-started-with-tidyverse-in-r/

Introduction to Pipes

Rene Magritt, *La trahison des images*, Wikimedia Commons

- Let the data flow.
- *Ceci n'est pas une pipe* -- `magrittr`
- The `%>%` pipe:
  - `x %>% f` ≡ `f(x)`
  - `x %>% f(y)` ≡ `f(x, y)`
  - `x %>% f %>% g %>% h` ≡ `h(g(f(x)))`

instead of writing this:

```
data <- iris
data <- head(data, n=3)
```

write this:

```
iris %>% head(n=3)
```

```
##   Sepal.Length Sepal.Width Petal.Length P
## 1          5.1         3.5          1.4
## 2          4.9         3.0          1.4
## 3          4.7         3.2          1.3
```

Tibbles

- `tibble` is one of the unifying features of tidyverse,
- it is a *better* `data.frame` realization,
- objects `data.frame` can be coerced to `tibble` using `as_tibble()`

```
head(as_tibble(iris))
```

```
## # A tibble: 6 × 5
##   Sepal.Length Sepal.Width Petal.Length P
##          <dbl>       <dbl>        <dbl>
## 1          5.1         3.5          1.4
## 2          4.9         3            1.4
## 3          4.7         3.2          1.3
## 4          4.6         3.1          1.5
## 5          5           3.6          1.4
## 6          5.4         3.9          1.7
```

```
tibble(
  x = 1,           # recycling
  y = runif(8),
  z = x + y^2,
  outcome = rnorm(8)
)
```

```
## # A tibble: 8 × 4
##       x     y     z outcome
##   <dbl> <dbl> <dbl>   <dbl>
## 1     1 0.734  1.54  -1.71
## 2     1 0.738  1.54  -0.925
## 3     1 0.340  1.12   0.876
## 4     1 0.340  1.12   0.407
## 5     1 0.751  1.56   0.488
## 6     1 0.926  1.86  -1.41
## 7     1 0.983  1.97   0.957
## 8     1 0.713  1.51  -0.945
```

More on Tibbles

- When you print a `tibble`:
    - all columns that fit the screen are shown,
    - first 10 rows are shown,
    - data type for each column is shown.

```
as_tibble(cars) %>% print(n = 5)
```

```
## # A tibble: 50 × 2
##   speed  dist
##   <dbl> <dbl>
## 1     4     2
## 2     4    10
## 3     7     4
## 4     7    22
## 5     8    16
## # ℹ 45 more rows
```

- `my_tibble %>% print(n = 50, width = Inf)`,
- `options(tibble.print_min = 15, tibble.print_max = 25)`,
- `options(dplyr.print_min = Inf)`,
- `options(tibble.width = Inf)`

Subsetting Tibbles

```
vehicles <- as_tibble(cars[1:5,])

vehicles[['speed']]
vehicles[[1]]
vehicles$speed

# Using placeholders

vehicles %>% .$dist
vehicles %>% .[['dist']]
vehicles %>% .[[2]]
```

```
## [1] 4 4 7 7 8
## [1] 4 4 7 7 8
## [1] 4 4 7 7 8
## [1]  2 10  4 22 16
## [1]  2 10  4 22 16
## [1]  2 10  4 22 16
```

Note! Not all old R functions work with tibbles, than you have to use `as.data.frame(my_tibble)`.

Tibbles are Stricter than `data.frames`

```
cars <- cars[1:5,]
```

```
cars$spe      # partial matching
```

```
## [1] 4 4 7 7 8
```

```
vehicles$spe  # no partial matching
```

```
## Warning: Unknown or uninitialised column: `spe`.
```

```
## NULL
```

```
cars$gear
```

```
## NULL
```

```
vehicles$gear
```

```
## Warning: Unknown or uninitialised column: `gear`.
```

```
## NULL
```

Loading Data

In `tidyverse` you import data using `readr` package that provides a number of useful data import functions:

- `read_delim()` a generic function for reading *-delimited files. There are a number of convenience wrappers:
  - `read_csv()` used to read comma-delimited files,
  - `read_csv2()` reads semicolon-delimited files, `read_tsv()` that reads tab-delimited files.
- `read_fwf` for reading fixed-width files with its wrappers:
  - fwf_widths() for width-based reading,
  - fwf_positions() for positions-based reading and
  - read_table() for reading white space-delimited fixed-width files.
- `read_log()` for reading Apache-style logs. The most commonly used `read_csv()` has some familiar arguments like:
- `skip` -- to specify the number of rows to skip (headers),
- `col_names` -- to supply a vector of column names,
- `comment` -- to specify what character designates a comment,
- `na` -- to specify how missing values are represented.

Importing Data Using `readr`

When reading and parsing a file, `readr` attempts to guess proper parser for each column by looking at the 1000 first rows.

```
tricky_dataset <- read_csv(readr_example('challenge.csv'))
```

```
## Rows: 2000 Columns: 2
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl  (1): x
## date (1): y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

OK, so there are some parsing failures. We can examine them more closely using `problems()` as suggested in the above output.

Looking at Problematic Columns

```
(p <- problems(tricky_dataset))
```

```
## # A tibble: 0 × 5
## # ℹ 5 variables: row <int>, col <int>, expected <chr>, actual <chr>, file <chr>
```

OK, let's see which columns cause trouble:

```
p %$% table(col)
```

```
## < table of extent 0 >
```

Looks like the problem occurs only in the `x` column.

Fixing Problematic Columns

So, how can we fix the problematic columns?

1. We can explicitely tell what parser to use:

```
tricky_dataset <- read_csv(readr_example('challenge.csv'),
                           col_types = cols(x = col_double(),
                                            y = col_character()))
tricky_dataset %>% tail(n = 5)
```

```
## # A tibble: 5 × 2
##       x y
##   <dbl> <chr>
## 1 0.164 2018-03-29
## 2 0.472 2014-08-04
## 3 0.718 2015-08-16
## 4 0.270 2020-02-04
## 5 0.608 2019-01-06
```

As you can see, we can still do better by parsing the `y` column as *date*, not as *character*.

Fixing Problematic Columns cted.

But knowing that the parser is guessed based on the first 1000 lines, we can see what sits past the 1000-th line in the data:

```
tricky_dataset %>% head(n = 1002) %>% tail(n = 4)
```

```
## # A tibble: 4 × 2
##          x y
##      <dbl> <chr>
## 1 4569     <NA>
## 2 4548     <NA>
## 3    0.238 2015-01-16
## 4    0.412 2018-05-18
```

It seems, we were very unlucky, because up till 1000-th line there are only integers in the x column and `NA` s in the y column so the parser cannot be guessed correctly. To fix this:

```
tricky_dataset <- read_csv(readr_example('challenge.csv'),
                           guess_max = 1001)
```

```
## Rows: 2000 Columns: 2
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## dbl  (1): x
## date (1): y
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Writing to a File

The `readr` package also provides functions useful for writing tibbled data into a file:

- `write_csv()`
- `write_tsv()`
- `write_excel_csv()`

They **always** save:

- text in UTF-8,
- dates in ISO8601

But saving in csv (or tsv) does mean you loose information about the type of data in particular columns. You can avoid this by using:

- `write_rds()` and `read_rds()` to read/write objects in R binary rds format,
- use `write_feather()` and `read_feather()` from package `feather` to read/write objects in a fast binary format that other programming languages can access.

Basic Data Transformations with `dplyr`

Let us create a tibble:

```
(bijou <- as_tibble(diamonds) %>% head(n = 10))
```

```
## # A tibble: 10 × 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
##  2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
##  5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  7  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  8  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  9  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
## 10  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
```

Picking Observations using `filter()`

```
bijou %>% filter(cut == 'Ideal' | cut == 'Premium', carat >= 0.23) %>% head(n = 5)
```

```
## # A tibble: 2 × 10
##   carat cut     color clarity depth table price     x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.29 Premium I     VS2      62.4    58   334  4.2   4.23  2.63
```

Be careful with floating point comparisons! Also, rows with comparison resulting in `NA` are skipped by default!

```
bijou %>% filter(near(0.23, carat) | is.na(carat)) %>% head(n = 5)
```

```
## # A tibble: 3 × 10
##   carat cut       color clarity depth table price     x     y     z
##   <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
## 3  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
```

Rearranging Observations using `arrange()`

```
bijou %>% arrange(cut, carat, desc(price))
```

```
## # A tibble: 10 × 10
##    carat cut       color clarity depth table price     x     y     z
##    <dbl> <ord>     <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
##  1  0.22 Fair      E     VS2      65.1    61   337  3.87  3.78  2.49
##  2  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
##  3  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
##  4  0.23 Very Good H     VS1      59.4    61   338  4     4.05  2.39
##  5  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
##  6  0.24 Very Good I     VVS1     62.3    57   336  3.95  3.98  2.47
##  7  0.26 Very Good H     SI1      61.9    55   337  4.07  4.11  2.53
##  8  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
##  9  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
## 10  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
```

The `NA`s always end up at the end of the rearranged tibble.

Selecting Variables with `select()`

Simple `select` with a range:

```
bijou %>% select(color, clarity, x:z) %>% head(n = 4)
```

```
## # A tibble: 4 × 5
##   color clarity     x     y     z
##   <ord> <ord>   <dbl> <dbl> <dbl>
## 1 E     SI2      3.95  3.98  2.43
## 2 E     SI1      3.89  3.84  2.31
## 3 E     VS1      4.05  4.07  2.31
## 4 I     VS2      4.2   4.23  2.63
```

Exclusive `select`:

```
bijou %>% select(-(x:z)) %>% head(n = 4)
```

```
## # A tibble: 4 × 7
##   carat cut     color clarity depth table price
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int>
## 1  0.23 Ideal   E     SI2      61.5    55   326
## 2  0.21 Premium E     SI1      59.8    61   326
## 3  0.23 Good    E     VS1      56.9    65   327
## 4  0.29 Premium I     VS2      62.4    58   334
```

Selecting Variables with `select()` cted.

`rename` is a variant of `select`, here used with `everything()` to move `x` to the beginning and rename it to `var_x`

```
bijou %>% rename(var_x = x) %>% head(n = 5)
```

```
## # A tibble: 5 × 10
##   carat cut     color clarity depth table price var_x     y     z
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      61.5    55   326  3.95  3.98  2.43
## 2  0.21 Premium E     SI1      59.8    61   326  3.89  3.84  2.31
## 3  0.23 Good    E     VS1      56.9    65   327  4.05  4.07  2.31
## 4  0.29 Premium I     VS2      62.4    58   334  4.2   4.23  2.63
## 5  0.31 Good    J     SI2      63.3    58   335  4.34  4.35  2.75
```

use `everything()` to bring some columns to the front:

```
bijou %>% select(x:z, everything()) %>% head(n = 4)
```

```
## # A tibble: 4 × 10
##       x     y     z carat cut     color clarity depth table price
##   <dbl> <dbl> <dbl> <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <int>
## 1  3.95  3.98  2.43  0.23 Ideal   E     SI2      61.5    55   326
## 2  3.89  3.84  2.31  0.21 Premium E     SI1      59.8    61   326
## 3  4.05  4.07  2.31  0.23 Good    E     VS1      56.9    65   327
## 4  4.2   4.23  2.63  0.29 Premium I     VS2      62.4    58   334
```

Create/alter new Variables with `mutate`

```
bijou %>% mutate(p = x + z, q = p + y) %>% select(-(depth:price)) %>% head(n = 5)
```

```
## # A tibble: 5 × 9
##   carat cut     color clarity     x     y     z     p     q
##   <dbl> <ord>   <ord> <ord>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  0.23 Ideal   E     SI2      3.95  3.98  2.43  6.38  10.4
## 2  0.21 Premium E     SI1      3.89  3.84  2.31  6.2   10.0
## 3  0.23 Good    E     VS1      4.05  4.07  2.31  6.36  10.4
## 4  0.29 Premium I     VS2      4.2   4.23  2.63  6.83  11.1
## 5  0.31 Good    J     SI2      4.34  4.35  2.75  7.09  11.4
```

or with `transmute` (only the transformed variables will be retained)

```
bijou %>% transmute(carat, cut, sum = x + y + z) %>% head(n = 5)
```

```
## # A tibble: 5 × 3
##   carat cut       sum
##   <dbl> <ord>   <dbl>
## 1  0.23 Ideal    10.4
## 2  0.21 Premium  10.0
## 3  0.23 Good     10.4
## 4  0.29 Premium  11.1
## 5  0.31 Good     11.4
```

Group and Summarize

```
bijou %>% group_by(cut) %>% summarize(max_price = max(price),
                                      mean_price = mean(price),
                                      min_price = min(price))
```

```
## # A tibble: 5 × 4
##   cut       max_price mean_price min_price
##   <ord>         <int>      <dbl>     <int>
## 1 Fair            337       337         337
## 2 Good            335       331         327
## 3 Very Good       338       337.        336
## 4 Premium         334       330         326
## 5 Ideal           326       326         326
```

```
bijou %>% group_by(cut, color) %>%
  summarize(max_price = max(price),
            mean_price = mean(price),
            min_price = min(price)) %>% head(n = 4)
```

```
## # A tibble: 4 × 5
## # Groups:   cut [3]
##   cut       color max_price mean_price min_price
##   <ord>     <ord>     <int>      <dbl>     <int>
## 1 Fair      E           337       337         337
## 2 Good      E           327       327         327
## 3 Good      J           335       335         335
## 4 Very Good H           338       338.        337
```

Other data manipulation tips

```
bijou %>% group_by(cut) %>% summarize(count = n())
```

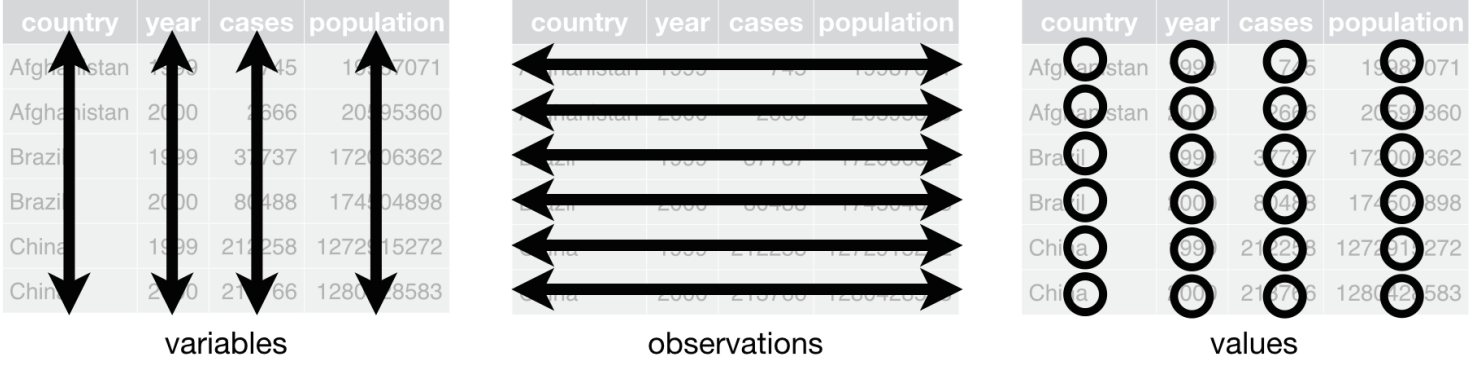
```
## # A tibble: 5 × 2
##   cut       count
##   <ord>     <int>
## 1 Fair          1
## 2 Good          2
## 3 Very Good     4
## 4 Premium       2
## 5 Ideal         1
```

When you need to regroup within the same pipe, use `ungroup()`.

The Concept of Tidy Data

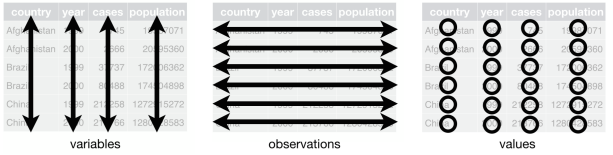
Data are tidy *sensu Wickham* if:

- each and every observation is represented as exactly one row,
- each and every variable is represented by exactly one column,
- thus each data table cell contains only one value.



Usually data are untidy in only one way. However, if you are unlucky, they are really untidy and thus a pain to work with...

Tidy Data



Are these data tidy?

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa

Species	variable	value
setosa	Sepal.Length	5.1
setosa	Sepal.Length	4.9
setosa	Sepal.Length	4.7

Sepal.L.W	Petal.L.W	Species
5.1/3.5	1.4/0.2	setosa
4.9/3	1.4/0.2	setosa
4.7/3.2	1.3/0.2	setosa

Sepal.Length	5.1	4.9	4.7	4.6
Sepal.Width	3.5	3.0	3.2	3.1
Petal.Length	1.4	1.4	1.3	1.5
Petal.Width	0.2	0.2	0.2	0.2
Species	setosa	setosa	setosa	setosa

Tidying Data with `tidyr::pivot_longer`

If some of your column names are actually values of a variable, use `pivot_longer` (replaces `gather`):

```
bijou2 %>% head(n = 5)
```

```
## # A tibble: 5 × 3
##   cut     `2008` `2009`
##   <ord>    <int>  <dbl>
## 1 Ideal      326   330.
## 2 Premium    326   330.
## 3 Good       327   331.
## 4 Premium    334   338.
## 5 Good       335   339.
```

```
bijou2 %>%
  pivot_longer(c(`2008`, `2009`), names_to = 'year', values_to = 'price') %>%
  head(n = 5)
```

```
## # A tibble: 5 × 3
##   cut     year  price
##   <ord>   <chr> <dbl>
## 1 Ideal   2008   326
## 2 Ideal   2009   330.
## 3 Premium 2008   326
## 4 Premium 2009   330.
## 5 Good    2008   327
```

Tidying Data with `tidyr::pivot_wider`

If some of your observations are scattered across many rows, use `pivot_wider` (replaces `gather`):

```
bijou3
```

```
## # A tibble: 9 × 5
##   cut     price clarity dimension measurement
##   <ord>   <int> <ord>   <chr>           <dbl>
## 1 Ideal     326 SI2     x                3.95
## 2 Premium   326 SI1     x                3.89
## 3 Good      327 VS1     x                4.05
## 4 Ideal     326 SI2     y                3.98
## 5 Premium   326 SI1     y                3.84
## 6 Good      327 VS1     y                4.07
## 7 Ideal     326 SI2     z                2.43
## 8 Premium   326 SI1     z                2.31
## 9 Good      327 VS1     z                2.31
```

```
bijou3 %>%
  pivot_wider(names_from=dimension, values_from=measurement) %>%
  head(n = 4)
```

```
## # A tibble: 3 × 6
##   cut     price clarity     x     y     z
##   <ord>   <int> <ord>   <dbl> <dbl> <dbl>
## 1 Ideal     326 SI2      3.95  3.98  2.43
## 2 Premium   326 SI1      3.89  3.84  2.31
## 3 Good      327 VS1      4.05  4.07  2.31
```

Tidying Data with `separate`

If some of your columns contain more than one value, use `separate`:

```
bijou4
```

```
## # A tibble: 5 × 4
##   cut     price clarity dim
##   <ord>   <int> <ord>   <chr>
## 1 Ideal     326 SI2     3.95/3.98/2.43
## 2 Premium   326 SI1     3.89/3.84/2.31
## 3 Good      327 VS1     4.05/4.07/2.31
## 4 Premium   334 VS2     4.2/4.23/2.63
## 5 Good      335 SI2     4.34/4.35/2.75
```

```
bijou4 %>%
  separate(dim, into = c("x", "y", "z"), sep = "/", convert = T)
```

```
## # A tibble: 5 × 6
##   cut     price clarity     x     y     z
##   <ord>   <int> <ord>   <dbl> <dbl> <dbl>
## 1 Ideal     326 SI2      3.95  3.98  2.43
## 2 Premium   326 SI1      3.89  3.84  2.31
## 3 Good      327 VS1      4.05  4.07  2.31
## 4 Premium   334 VS2      4.2   4.23  2.63
## 5 Good      335 SI2      4.34  4.35  2.75
```

Tidying Data with `unite`

If some of your columns contain more than one value, use `separate`:

```
bijou5
```

```
## # A tibble: 5 × 7
##   cut     price clarity_prefix clarity_suffix     x     y     z
##   <ord>   <int> <chr>          <chr>          <dbl> <dbl> <dbl>
## 1 Ideal     326 SI             2               3.95  3.98  2.43
## 2 Premium   326 SI             1               3.89  3.84  2.31
## 3 Good      327 VS             1               4.05  4.07  2.31
## 4 Premium   334 VS             2               4.2   4.23  2.63
## 5 Good      335 SI             2               4.34  4.35  2.75
```

```
bijou5 %>% unite(clarity, clarity_prefix, clarity_suffix, sep='')
```

```
## # A tibble: 5 × 6
##   cut     price clarity     x     y     z
##   <ord>   <int> <chr>   <dbl> <dbl> <dbl>
## 1 Ideal     326 SI2      3.95  3.98  2.43
## 2 Premium   326 SI1      3.89  3.84  2.31
## 3 Good      327 VS1      4.05  4.07  2.31
## 4 Premium   334 VS2      4.2   4.23  2.63
## 5 Good      335 SI2      4.34  4.35  2.75
```

Note: that `sep` is here interpreted as the position to split on. It can also be a *regular expression* or a delimiting string/character. Pretty flexible approach!

Completing Missing Values Using `complete`

```
bijou %>% head(n = 10) %>%
  select(cut, clarity, price) %>%
  mutate(continent = sample(c('AusOce', 'Eur'),
                            size = 10,
                            replace = T)) -> missing_stones
```

```
missing_stones %>% complete(cut, continent)
```

```
## # A tibble: 13 × 4
##    cut       continent clarity price
##    <ord>     <chr>     <ord>   <int>
##  1 Fair      AusOce    <NA>       NA
##  2 Fair      Eur       VS2       337
##  3 Good      AusOce    <NA>       NA
##  4 Good      Eur       VS1       327
##  5 Good      Eur       SI2       335
##  6 Very Good AusOce    SI1       337
##  7 Very Good AusOce    VS1       338
##  8 Very Good Eur       VVS2      336
##  9 Very Good Eur       VVS1      336
## 10 Premium   AusOce    SI1       326
## 11 Premium   Eur       VS2       334
## 12 Ideal     AusOce    SI2       326
## 13 Ideal     Eur       <NA>       NA
```

Some Other Friends

- `stringr` for string manipulation and regular expressions,
- `forcats` for working with factors,
- `lubridate` for working with dates.

Thank you. Questions?

Graphics from freepik.com

Created: 09-Aug-2023 • Roy Francis • SciLifeLab • NBIS