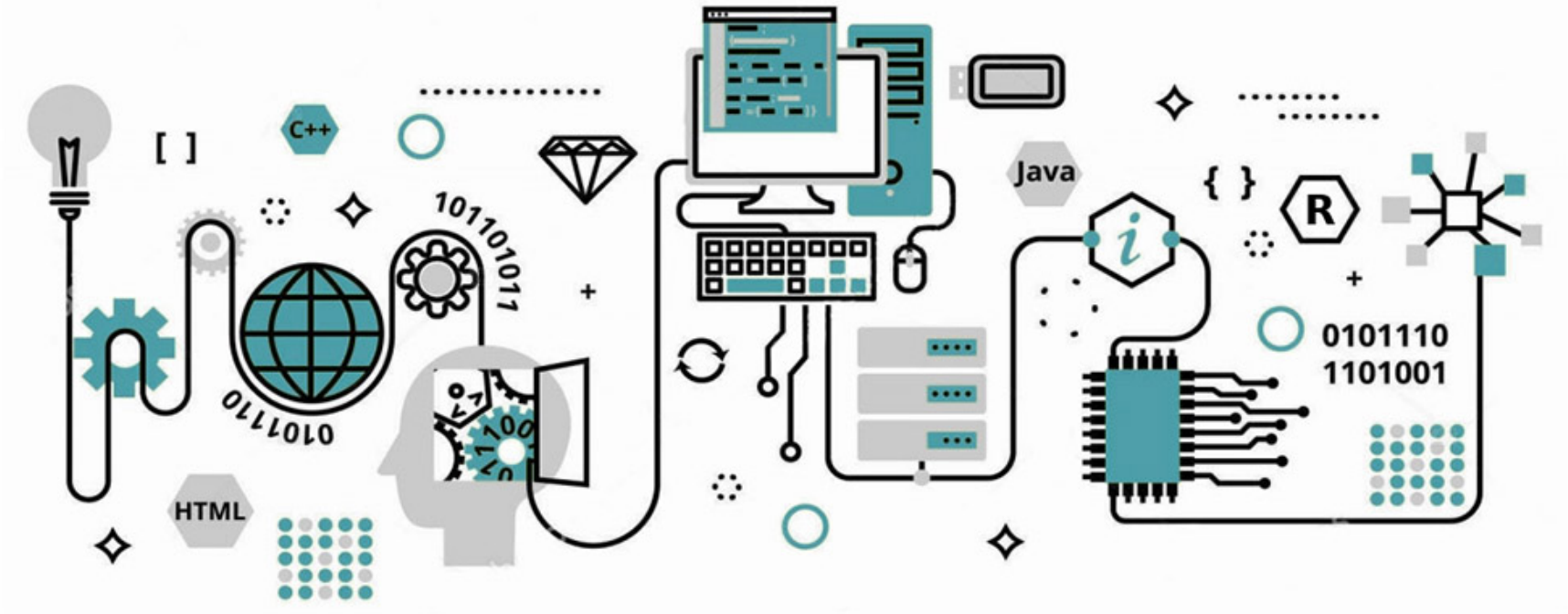


Tidy work in Tidyverse

R Programming Foundation for Life Scientists

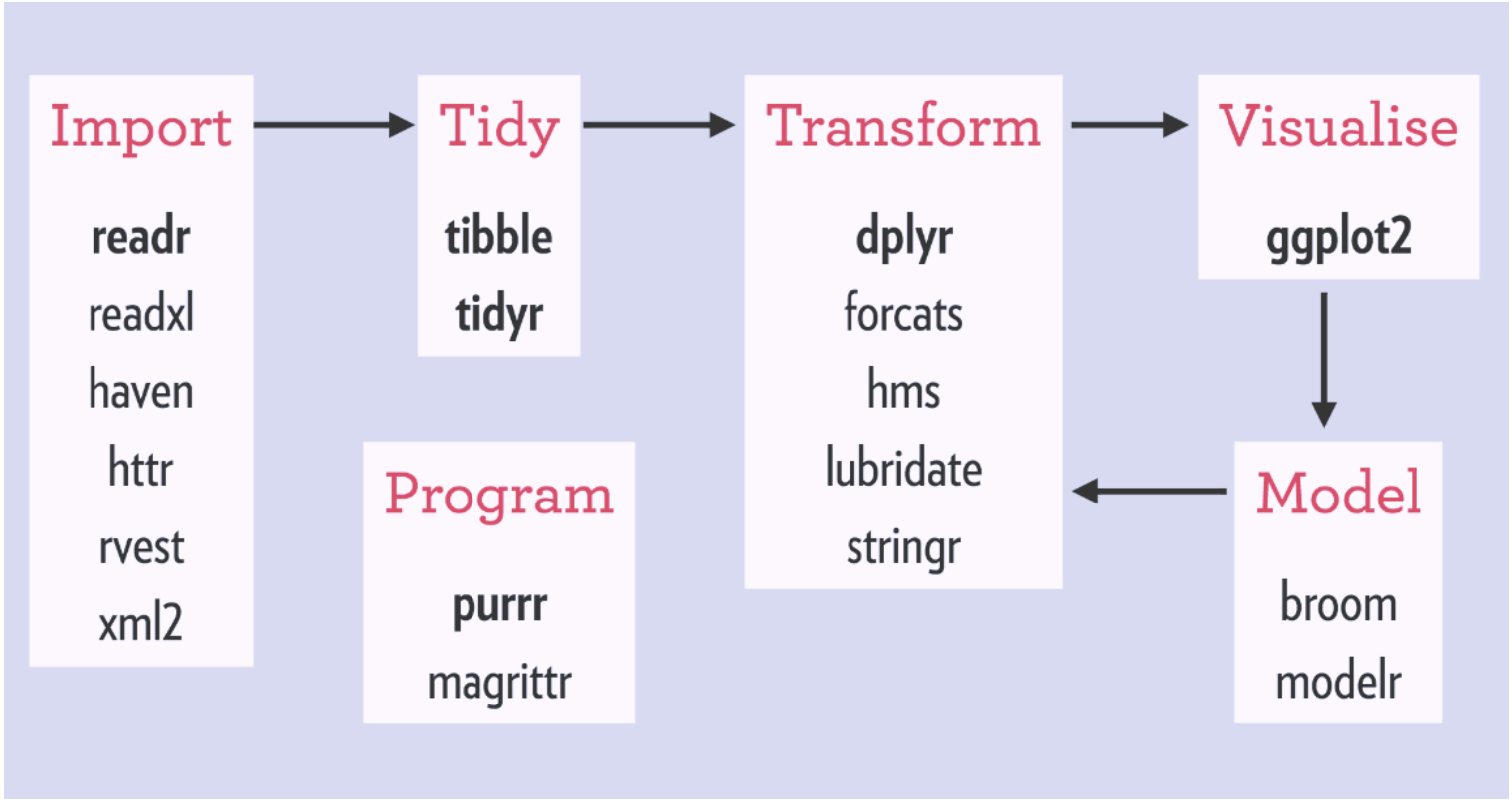
06-Nov-2020

NBIS

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
## 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()`

```
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length
##           <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
##  7          4.6         3.4          1.4
##  8          5           3.4          1.5
##  9          4.4         2.9          1.4
## 10          4.9         3.1          1.5
## 11          5.4         3.7          1.5
## 12          4.8         3.4          1.6
## 13          4.8         3            1.4
## 14          4.3         3            1.1
```

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

```
## # A tibble: 8 x 4
##       x      y     z outcome
##   <dbl>  <dbl> <dbl>   <dbl>
## 1     1 0.598   1.36  -0.741
## 2     1 0.924   1.85   0.963
## 3     1 0.767   1.59  -2.24
## 4     1 0.0402  1.00  -0.873
## 5     1 0.338   1.11  -0.210
## 6     1 0.922   1.85  -0.296
## 7     1 0.490   1.24   1.16
## 8     1 0.0561  1.00   1.83
```

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: 5 x 2
##   speed  dist
##   <dbl> <dbl>
## 1     4     2
## 2     4    10
## 3     7     4
## 4     7    22
## 5     8    16
```

- `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'))
```

```
##
## ── Column specification ────────────────────────────────────────────────────────
## cols(
##   x = col_double(),
##   y = col_logical()
## )
```

```
## Warning: 1000 parsing failures.
##  row col           expected     actual
## 1001   y 1/0/T/F/TRUE/FALSE 2015-01-16 '/home/runner/work/_temp/Library/readr/extdata/
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 '/home/runner/work/_temp/Library/readr/extdata/
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 '/home/runner/work/_temp/Library/readr/extdata/
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 '/home/runner/work/_temp/Library/readr/extdata/
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 '/home/runner/work/_temp/Library/readr/extdata/
## .... ... .................. .......... ..............................................
## See problems(...) for more details.
```

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: 1,000 x 5
##      row col   expected        actual   file
##    <int> <chr> <chr>           <chr>    <chr>
##  1  1001 y     1/0/T/F/TRUE/F… 2015-01… '/home/runner/work/_temp/Library/readr/…
##  2  1002 y     1/0/T/F/TRUE/F… 2018-05… '/home/runner/work/_temp/Library/readr/…
##  3  1003 y     1/0/T/F/TRUE/F… 2015-09… '/home/runner/work/_temp/Library/readr/…
##  4  1004 y     1/0/T/F/TRUE/F… 2012-11… '/home/runner/work/_temp/Library/readr/…
##  5  1005 y     1/0/T/F/TRUE/F… 2020-01… '/home/runner/work/_temp/Library/readr/…
##  6  1006 y     1/0/T/F/TRUE/F… 2016-04… '/home/runner/work/_temp/Library/readr/…
##  7  1007 y     1/0/T/F/TRUE/F… 2011-05… '/home/runner/work/_temp/Library/readr/…
##  8  1008 y     1/0/T/F/TRUE/F… 2020-07… '/home/runner/work/_temp/Library/readr/…
##  9  1009 y     1/0/T/F/TRUE/F… 2011-04… '/home/runner/work/_temp/Library/readr/…
## 10  1010 y     1/0/T/F/TRUE/F… 2010-05… '/home/runner/work/_temp/Library/readr/…
## 11  1011 y     1/0/T/F/TRUE/F… 2014-11… '/home/runner/work/_temp/Library/readr/…
## 12  1012 y     1/0/T/F/TRUE/F… 2014-06… '/home/runner/work/_temp/Library/readr/…
## 13  1013 y     1/0/T/F/TRUE/F… 2017-05… '/home/runner/work/_temp/Library/readr/…
## 14  1014 y     1/0/T/F/TRUE/F… 2017-11… '/home/runner/work/_temp/Library/readr/…
## 15  1015 y     1/0/T/F/TRUE/F… 2013-04… '/home/runner/work/_temp/Library/readr/…
## # … with 985 more rows
```

OK, let's see which columns cause trouble:

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

```
## col
```

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 x 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 x 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)
```

```
##
## ── Column specification ────────────────────────────────────────────────────────
## cols(
##   x = col_double(),
##   y = col_date(format = "")
## )
```

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 = 100))
```

```
## # A tibble: 100 x 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 0.290 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
## 11 0.3   Good      J     SI1      64      55   339  4.25  4.28  2.73
## 12 0.23  Ideal     J     VS1      62.8    56   340  3.93  3.9   2.46
## 13 0.22  Premium   F     SI1      60.4    61   342  3.88  3.84  2.33
## 14 0.31  Ideal     J     SI2      62.2    54   344  4.35  4.37  2.71
## 15 0.2   Premium   E     SI2      60.2    62   345  3.79  3.75  2.27
## # … with 85 more rows
```

Picking Observations using `filter()`

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

```
## # A tibble: 5 x 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 0.290 Premium I     VS2      62.4    58   334  4.2   4.23  2.63
## 3 0.23  Ideal   J     VS1      62.8    56   340  3.93  3.9   2.46
## 4 0.31  Ideal   J     SI2      62.2    54   344  4.35  4.37  2.71
## 5 0.32  Premium E     I1       60.9    58   345  4.38  4.42  2.68
```

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: 5 x 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
## 4  0.23 Ideal     J     VS1      62.8    56   340  3.93  3.9   2.46
## 5  0.23 Very Good E     VS2      63.8    55   352  3.85  3.92  2.48
```

Rearranging Observations using `arrange()`

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

```
## # A tibble: 100 x 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  0.86 Fair  E     SI2      55.1    69  2757  6.45  6.33  3.52
##  3  0.96 Fair  F     SI2      66.3    62  2759  6.27  5.95  4.07
##  4  0.23 Good  F     VS1      58.2    59   402  4.06  4.08  2.37
##  5  0.23 Good  E     VS1      64.1    59   402  3.83  3.85  2.46
##  6  0.23 Good  E     VS1      56.9    65   327  4.05  4.07  2.31
##  7  0.26 Good  E     VVS1     57.9    60   554  4.22  4.25  2.45
##  8  0.26 Good  D     VS2      65.2    56   403  3.99  4.02  2.61
##  9  0.26 Good  D     VS1      58.4    63   403  4.19  4.24  2.46
## 10  0.3  Good  H     SI1      63.7    57   554  4.28  4.26  2.72
## 11  0.3  Good  I     SI1      63.2    55   405  4.25  4.29  2.7
## 12  0.3  Good  J     SI1      63.4    54   351  4.23  4.29  2.7
## 13  0.3  Good  J     SI1      63.8    56   351  4.23  4.26  2.71
## 14  0.3  Good  I     SI2      63.3    56   351  4.26  4.3   2.71
## 15  0.3  Good  J     SI1      64      55   339  4.25  4.28  2.73
## # … with 85 more rows
```

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 x 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 x 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 0.290 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 x 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 0.290 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 x 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 0.290 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 x 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 0.290 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 x 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 0.290 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 x 4
##   cut       max_price mean_price min_price
##   <ord>         <int>      <dbl>     <int>
## 1 Fair           2759      1951        337
## 2 Good           2759       661.       327
## 3 Very Good      2760       610.       336
## 4 Premium        2760       569.       326
## 5 Ideal          2757       693.       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 x 5
## # Groups:   cut [2]
##   cut   color max_price mean_price min_price
##   <ord> <ord>     <int>      <dbl>     <int>
## 1 Fair  E          2757      1547        337
## 2 Fair  F          2759      2759       2759
## 3 Good  D           403       403        403
## 4 Good  E          2759      1010.       327
```

Other data manipulation tips

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

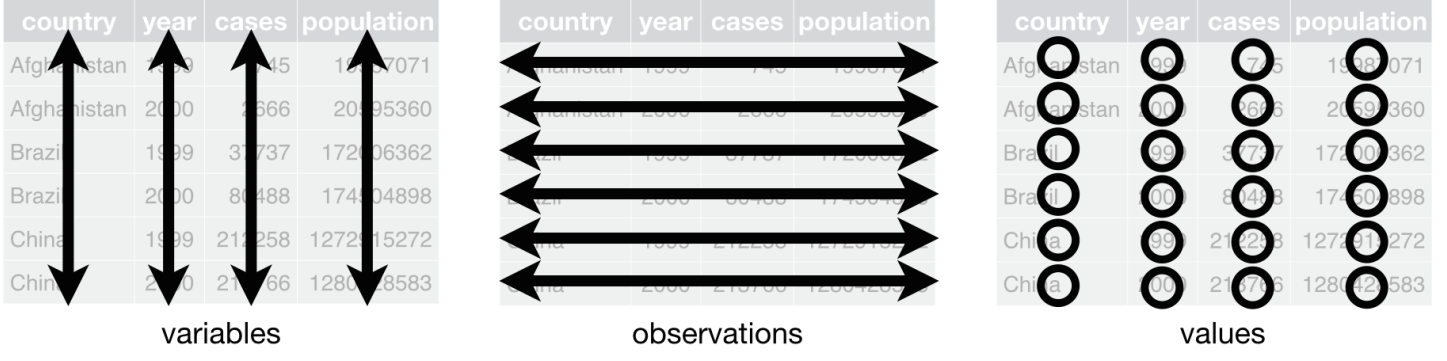
```
## # A tibble: 5 x 2
##   cut       count
##   <ord>     <int>
## 1 Fair          3
## 2 Good         18
## 3 Very Good    38
## 4 Premium      22
## 5 Ideal        19
```

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

The Concept of Tidy Data

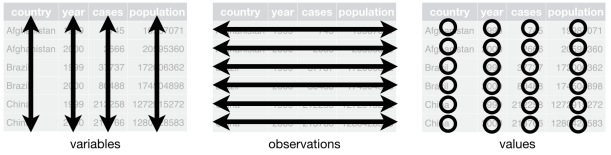
Data are tidy *sensu Wickham* if:

- 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 x 3
##   cut     `2008` `2009`
##   <ord>    <int>  <dbl>
## 1 Ideal      326   333.
## 2 Premium    326   333.
## 3 Good       327   334.
## 4 Premium    334   341.
## 5 Good       335   342.
```

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

```
## # A tibble: 5 x 3
##   cut     year  price
##   <ord>   <chr> <dbl>
## 1 Ideal   2008   326
## 2 Ideal   2009   333.
## 3 Premium 2008   326
## 4 Premium 2009   333.
## 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 x 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 x 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 x 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 x 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 x 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 x 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: 14 x 4
##    cut       continent clarity price
##    <ord>     <chr>     <ord>   <int>
##  1 Fair      AusOce    VS2       337
##  2 Fair      Eur       <NA>       NA
##  3 Good      AusOce    <NA>       NA
##  4 Good      Eur       VS1       327
##  5 Good      Eur       SI2       335
##  6 Very Good AusOce    VVS2      336
##  7 Very Good AusOce    VVS1      336
##  8 Very Good AusOce    SI1       337
##  9 Very Good Eur       VS1       338
## 10 Premium   AusOce    SI1       326
## 11 Premium   AusOce    VS2       334
## 12 Premium   Eur       <NA>       NA
## 13 Ideal     AusOce    SI2       326
## 14 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?

R version 4.0.3 (2020-10-10)

Platform: x86_64-pc-linux-gnu (64-bit)

OS: Ubuntu 18.04.5 LTS

Built on : 06-Nov-2020 at 22:21:03

2020 • SciLifeLab • NBIS