Tidy Data

Kylie Ariel Bemis

1/21/2019

Introduction to Data Wrangling

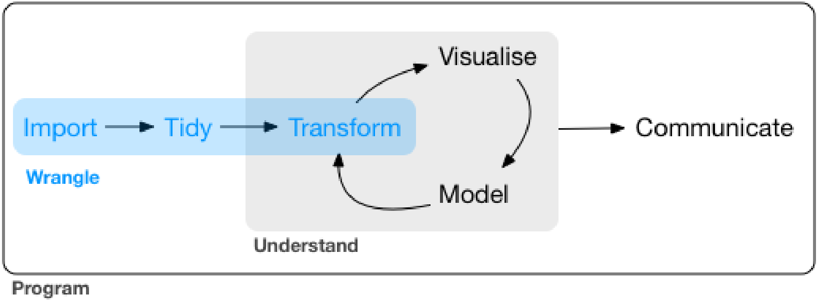


Figure 1: Wickham and Grolemund, *R for Data Science*

Introduction to Data Wrangling

Today we will discuss the often-frustrating but necessary steps that come before we can visualize or model data.

- Importing data (read it into your analysis software)
- Tidying data (put it in a tidy format for data analysis)
- Transforming data (perform any transformations necessary)

Together, these steps are often collectively refered to as data wrangling.

We will focus on importing and tidying today.

Tibbles

Tibbles are a type of lightweight data frame used by the `tidyverse`.

They inherit many behaviors from `data.frame`.

In fact, as an S3 class, they inherit from `data.frame` directly.

```
class(mpg)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The `tbl_df` part tells us that it's a tibble that is fully loaded in memory. The `data.frame` part tells it inherits from `data.frame`.

`tbl` is the tidyverse's generic notion of tabular data. They will become important again later when we discuss working with databases.

Differences versus `data.frame`

- Tibbles print only the first 10 rows
- Tibbles print only as many columns as fit on your console
- Tibbles print information about the column data type
- Tibbles don't require `row.names`
- Tibbles don't munge column names
- Tibbles don't coerce inputs (`stringsAsFactors=FALSE`)
- Tibbles always use `drop=FALSE` when subsetting with `data[,j]`
- You can always use `as.data.frame` to get an ordinary `data.frame`

Coercing tibbles with as_tibble

```r
as_tibble(iris)
```

```
## # A tibble: 150 x 5
##    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##           <dbl>       <dbl>        <dbl>       <dbl> <fct>
##  1          5.1         3.5          1.4         0.2 setosa
##  2          4.9         3            1.4         0.2 setosa
##  3          4.7         3.2          1.3         0.2 setosa
##  4          4.6         3.1          1.5         0.2 setosa
##  5          5           3.6          1.4         0.2 setosa
##  6          5.4         3.9          1.7         0.4 setosa
##  7          4.6         3.4          1.4         0.3 setosa
##  8          5           3.4          1.5         0.2 setosa
##  9          4.4         2.9          1.4         0.2 setosa
## 10          4.9         3.1          1.5         0.1 setosa
## # ... with 140 more rows
```

Creating tibbles with `tibble`

```
tibble(x=1:10, y=11:20, z=letters[1:10])
```

```
## # A tibble: 10 x 3
##        x     y z
##    <int> <int> <chr>
##  1     1    11 a
##  2     2    12 b
##  3     3    13 c
##  4     4    14 d
##  5     5    15 e
##  6     6    16 f
##  7     7    17 g
##  8     8    18 h
##  9     9    19 i
## 10    10    20 j
```

Creating tibbles with `tribble`

```
tribble(~x, ~y, ~z,
        1,   2,  'i',
        2,   4,  'j',
        3,   8,  'k')
```

```
## # A tibble: 3 x 3
##         x     y z
##     <dbl> <dbl> <chr>
## 1       1     2 i
## 2       2     4 j
## 3       3     8 k
```

A note on factors

- Categorical variables are stored in R as the `factor` data type
- Factors are stored as integers with character information about levels
  - This allows them to be smaller than `character` vectors
  - This is also useful for many statistical methods
- Many base R functions automatically coerce `character` to `factor`; most `tidyverse` functions do not
  - `data.frame()` vs `tibble()`
  - `read.csv()` vs `read_csv()`
- `ordered` is an ordered version for categorical variables with order levels
- Can change levels with `levels()<-` or `dplyr::recode()`
- Use `factor` or `character`?

```r
fc <- factor(c("red", "red", "blue"))
fc
```

```
## [1] red  red  blue
## Levels: blue red
```

```r
levels(fc) <- c("blue2", "red1")
fc
```

```
## [1] red1  red1  blue2
## Levels: blue2 red1
```

```r
dplyr::recode(fc, red1="one", blue2="two")
```

```
## [1] one one two
## Levels: two one
```

Importing data

At some point, it is necessary to import outside datasets into your data analysis software (R in our case).

Sometimes this can be easy, but sometimes this can be the most tedious and frustrating step in data science.

Data files can be:

- Messy
- Have errors
- An unknown file format
- Text or binary
- Structured or unstructured

Today, we will focus on ways of importing tabular data in a flat text file.

Next week, we will discuss importing other types of data.

Importing data with `readr`

The `readr` package is the part of the tidyverse responsible for importing data.

It provides multiple functions for the importing of tabular data.

- `read_csv()` and family read delimited files
  - `read_csv()` and `read_csv2()` read in comma or semicolon separated files, respectively
  - `read_tsv()` reads in tab-delimited files
  - `read_delim()` allows the user to specify the delimiter
- `read_fwf()` reads fixed-width files
- `read_file()` and `read_lines()` simply read in lines or full files as `character` data or `raw` (byte) data

We will primarily discuss `read_csv()`.

Differences with `read.csv()` and related functions

`read.csv()` and similar functions are also provided in any default R installation (package `utils`, loaded automatically in most R sessions).

The `readr` versions such as `read_csv()` have certain advantages:

- They are typically faster (up to 10x)
- They typically use less memory
- They output data as tibbles
  - `character` vectors aren't coerced to `factor`
  - `row.names` are not added
  - Column names are not munged

Reading csv files with `read_csv`

First argument is the path to the file.

This may be a relative path or the full path.

R understands typicaly *nix shortcuts.

```
output1 <- read_csv("path/to/file.csv")
output1 <- read_csv("/Users/username/data/path/to/file.csv")
output2 <- read_csv("~/path/to/other/file.csv")
output1 <- read_csv("../data/path/to/file.csv")
```

```r
mtcars2 <- read_csv(readr_example("mtcars.csv"))
```

```
## Parsed with column specification:
## cols(
##   mpg = col_double(),
##   cyl = col_double(),
##   disp = col_double(),
##   hp = col_double(),
##   drat = col_double(),
##   wt = col_double(),
##   qsec = col_double(),
##   vs = col_double(),
##   am = col_double(),
##   gear = col_double(),
##   carb = col_double()
## )
```

```r
mtcars2
```

```
## # A tibble: 32 x 11
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gea
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl
##  1  21       6  160    110  3.9   2.62  16.5     0     1
##  2  21       6  160    110  3.9   2.88  17.0     0     1
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0
##  8  24.4     4  147.    62  3.69  3.19  20       1     0
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0
## # ... with 22 more rows
```

Inline csv input is also accepted.

```
read_csv("a,b,c
          1,2,3
          4,5,6")
```

```
## # A tibble: 2 x 3
##         a     b     c
##     <dbl> <dbl> <dbl>
## 1       1     2     3
## 2       4     5     6
```

```r
read_csv("a,b,c\n1,2,3\n4,5,6")
```

```
## # A tibble: 2 x 3
##       a     b     c
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Skip lines with `skip`.

```r
read_csv("The first line of metadata
          The second line of metadata
          x,y,z
          1,2,3", skip = 2)
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

Specify comments with `comment`.

```
read_csv("# A comment I want to skip
          x,y,z
          1,2,3", comment = "#")
```

```
## # A tibble: 1 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
```

`read_csv()` assumes the first line gives column names.

Set `col_names` to `FALSE` if this is not the case.

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```

```
## # A tibble: 2 x 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Or use `col_names` to set your own column names.

```r
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```

```
## # A tibble: 2 x 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

Use `na` to tell `read_csv()` how missing values are specified

```r
read_csv("a,b,c
          1,2,.", na = ".")
```

```
## # A tibble: 1 x 3
##       a     b c
##   <dbl> <dbl> <lgl>
## 1     1     2 NA
```

`read_csv()` attempts to guess the correct data type for each column.

Use `col_types` and `cols()` to manually specify column data types.

```
tmp <- read_csv("a,b,c\n1,2,3\n4,5,6",
         col_types = cols(b=col_character(),
                          c=col_character()))
```

You can also set a default column data type.

```r
read_csv("a,b,c\n1,2,3\n4,5,6",
         col_types = cols(.default=col_character()))
```

```
## # A tibble: 2 x 3
##   a     b     c
##   <chr> <chr> <chr>
## 1 1     2     3
## 2 4     5     6
```

Sometimes `read_csv()` guesses wrong.

```
challenge <- read_csv(readr_example("challenge.csv"))
```

```
## Parsed with 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 '/Users/kuwisdelu/Libr
## 1002   y 1/0/T/F/TRUE/FALSE 2018-05-18 '/Users/kuwisdelu/Libr
## 1003   y 1/0/T/F/TRUE/FALSE 2015-09-05 '/Users/kuwisdelu/Libr
## 1004   y 1/0/T/F/TRUE/FALSE 2012-11-28 '/Users/kuwisdelu/Libr
## 1005   y 1/0/T/F/TRUE/FALSE 2020-01-13 '/Users/kuwisdelu/Libr
## .... ... .................. .......... ......................
## See problems(...) for more details.
```

```r
problems(challenge)
```

```
## # A tibble: 1,000 x 5
##      row col   expected        actual    file
##    <int> <chr> <chr>           <chr>     <chr>
##  1  1001 y     1/0/T/F/TRUE/~ 2015-01~ '/Users/kuwisdelu/Libr
##  2  1002 y     1/0/T/F/TRUE/~ 2018-05~ '/Users/kuwisdelu/Libr
##  3  1003 y     1/0/T/F/TRUE/~ 2015-09~ '/Users/kuwisdelu/Libr
##  4  1004 y     1/0/T/F/TRUE/~ 2012-11~ '/Users/kuwisdelu/Libr
##  5  1005 y     1/0/T/F/TRUE/~ 2020-01~ '/Users/kuwisdelu/Libr
##  6  1006 y     1/0/T/F/TRUE/~ 2016-04~ '/Users/kuwisdelu/Libr
##  7  1007 y     1/0/T/F/TRUE/~ 2011-05~ '/Users/kuwisdelu/Libr
##  8  1008 y     1/0/T/F/TRUE/~ 2020-07~ '/Users/kuwisdelu/Libr
##  9  1009 y     1/0/T/F/TRUE/~ 2011-04~ '/Users/kuwisdelu/Libr
## 10  1010 y     1/0/T/F/TRUE/~ 2010-05~ '/Users/kuwisdelu/Libr
## # ... with 990 more rows
```

x should be a double

```
challenge <- read_csv(readr_example("challenge.csv"),
                      col_types=cols(x=col_double()))
```

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

```
challenge
```

```
## # A tibble: 2,000 x 2
##         x y
##     <dbl> <lgl>
##  1    404 NA
##  2   4172 NA
##  3   3004 NA
##  4    787 NA
##  5     37 NA
##  6   2332 NA
##  7   2489 NA
##  8   1449 NA
##  9   3665 NA
## 10   3863 NA
## # ... with 1,990 more rows
```

Did `read_csv()` guess correctly for `y`?

```r
challenge[which(!is.na(challenge$y)),]
```

```
## # A tibble: 0 x 2
## # ... with 2 variables: x <dbl>, y <lgl>
```

y should be a date.

```
challenge <- read_csv(readr_example("challenge.csv"),
                      col_types=cols(x=col_double(),
                                     y=col_date()))
```

By default, `read_csv()` guesses based on the first 1000 rows.

We can tell `read_csv()` to look at more rows before guessing.

```
challenge2 <- read_csv(readr_example("challenge.csv"),
                       guess_max = 1001)
```

```
## Parsed with column specification:
## cols(
##   x = col_double(),
##   y = col_date(format = "")
## )
```

This also fixes the problem in this case.

Writing csv files with `write_csv`

We can also write out files.

The first argument is the data and the second argument is the path.

```
write_csv(mtcars2, "mtcars2.csv")
```

Because the data is written as text, all type information is lost.

Reading tabular and non-tabular data

There are many other packages for reading other types of data formats.

Some packages for reading other formats of tabular data include:

- `haven` for reading SPSS, Stata, and SAS files
- `readxl` for reading Excel files
- `DBI` and a database backend allow working with databases

We will also discuss more on importing and tidying non-tabular data next week.

Tidy data

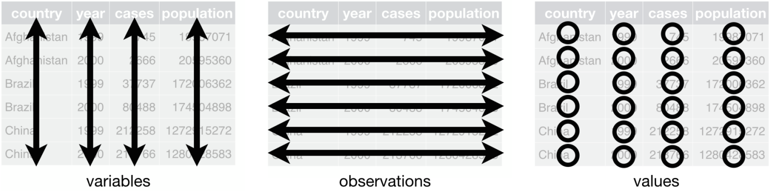


Figure 2: Wickham and Grolemund, *R for Data Science*

Review of tidy data rules

- Each variable must have its own column.
- Each observation must have its own row.
- Each value must have its own cell.

Is it tidy?

```
## # A tibble: 12 x 4
##    country      year type             count
##    <chr>       <int> <chr>            <int>
##  1 Afghanistan  1999 cases              745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases             2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases            37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases            80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases           212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases           213766
## 12 China        2000 population 1280428583
```

Is it tidy?

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

Is it tidy?

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
## * <chr>            <int>      <int>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

Is it tidy?

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Why tidy data?

- Consistent format allows us to work with many datasets with a single set of tools
- Vectorized operations on variables are intuitive and computationally efficient

Tidying data with `tidyr`

The `tidyr` package is the part of the tidyverse responsible for helping you make data tidy.

It is primarily designed around solving two common problems:

- One variable is spread across multiple columns.
- One observation is scattered across multiple rows.

The `spread()` and `gather()` functions are designed to fix these problems.

Gathering

Column names are values rather than variables.

Figure 3: Wickham and Grolemund, *R for Data Science*

1999 and 2000 are values of an omitted variable `year`.

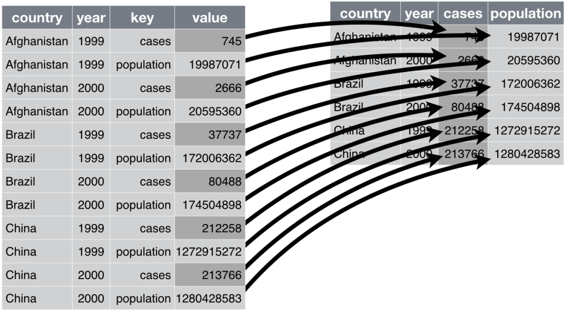
```
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
## * <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```r
gather(table4a, `1999`, `2000`, key = "year", value = "cases")
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <int>
## 1 Afghanistan 1999     745
## 2 Brazil      1999   37737
## 3 China       1999  212258
## 4 Afghanistan 2000    2666
## 5 Brazil      2000   80488
## 6 China       2000  213766
```

Spreading



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

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

Figure 4: Wickham and Grolemund, *R for Data Science*

Observation is scattered in multiple rows.

The values of `type` should be their own variables.

```
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <int> <chr>           <int>
##  1 Afghanistan  1999 cases             745
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```r
spread(table2, key = type, value = count)
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <int>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Separating

Sometimes character strings are used to encode values for more than one variable.

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Figure 5: Wickham and Grolemund, *R for Data Science*

`rate` encodes both `cases` and `population` as a single string.

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```r
separate(table3, rate, into = c("cases", "population"))
```

```
## # A tibble: 6 x 4
##   country      year cases  population
## * <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```r
pets <- tribble(~name, ~description,
                "Daisy", "CAT_F",
                "Johnny", "CAT_M",
                "Patsy", "CAT_F",
                "Yuma", "DOG_F")
```

Sometimes you want to separate at a specific character.

```
separate(pets, description, into=c("species", "sex"), sep="_")
```

```
## # A tibble: 4 x 3
##   name   species sex
##   <chr>  <chr>   <chr>
## 1 Daisy  CAT     F
## 2 Johnny CAT     M
## 3 Patsy  CAT     F
## 4 Yuma   DOG     F
```

```
separate(pets, description, into=c("species", "sex"), sep=4)
```

```
## # A tibble: 4 x 3
##   name   species sex
##   <chr>  <chr>   <chr>
## 1 Daisy  CAT_    F
## 2 Johnny CAT_    M
## 3 Patsy  CAT_    F
## 4 Yuma   DOG_    F
```

Uniting

Sometimes you want to create a variable that combines character encodings from multiple rows.

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Figure 6: Wickham and Grolemund, *R for Data Science*

```r
unite(pets, id, name, description)
```

```
## # A tibble: 4 x 1
##   id
##   <chr>
## 1 Daisy_CAT_F
## 2 Johnny_CAT_M
## 3 Patsy_CAT_F
## 4 Yuma_DOG_F
```

```r
addressbook <- tribble(~name,   ~city,          ~state,
                       "Kylie", "Jamaica Plain", "MA",
                       "Olga",  "Brookline",     "MA")
```

```r
unite(addressbook, address, city, state, sep=", ")
```

```
## # A tibble: 2 x 2
##   name  address
##   <chr> <chr>
## 1 Kylie Jamaica Plain, MA
## 2 Olga  Brookline, MA
```

When to make data ‘untidy’?

Sometimes it can be helpful to transform a dataset into an ‘untidy’ format for a particular purpose.

For example, consider a survey that allows you to select more than one option for a particular question.

The following tibble contains information on students and the classes in which they are enrolled.

```
classes <- tibble(student = c(0, 1, 2, 3, 4, 5, 6, 7, 8, 9),
                  year =    c(1, 2, 1, 1, 2, 3, 3, 4, 4, 3),
                  algrthm = c(1, 1, 1, 0, 0, 0, 1, 1, 1, 1),
                  datastr = c(1, 0, 0, 1, 0, 1, 0, 0, 0, 0),
                  opersys = c(0, 0, 0, 0, 1, 1, 1, 0, 0, 1),
                  prglang = c(0, 1, 0, 0, 1, 1, 1, 0, 0, 0))
```

What if we want to calculate summaries based on each class?

We would like to `group_by(class)` using `dplyr`.

To do that, we need to create a `class` variable.

```
classes2 <- classes %>%
  gather(key="class", value="is_enrolled",
       algrthm, datastr, opersys, prglang) %>%
  filter(is_enrolled == 1) %>%
  select(-is_enrolled)
```

A student may now appear in more than one row.

```
classes2
```

```
## # A tibble: 18 x 3
##    student  year class
##      <dbl> <dbl> <chr>
##  1       0     1 algrthm
##  2       1     2 algrthm
##  3       2     1 algrthm
##  4       6     3 algrthm
##  5       7     4 algrthm
##  6       8     4 algrthm
##  7       9     3 algrthm
##  8       0     1 datastr
##  9       3     1 datastr
## 10       5     3 datastr
## 11       4     2 opersys
## 12       5     3 opersys
## 13       6     3 opersys
## 14       9     3 opersys
## 15       1     2 prglang
## 16       4     2 prglang
```

Get the number of students in each class.

```
classes2 %>% group_by(class) %>% count()
```

```
## # A tibble: 4 x 2
## # Groups:   class [4]
##   class        n
##   <chr>    <int>
## 1 algrthm      7
## 2 datastr      3
## 3 opersys      4
## 4 prglang      4
```

We can use a similar technique for when a categorical variable is spread across multiple rows.

```
pets2 <- tribble(~name, ~is_cat, ~is_dog,
                 "Daisy", "yes", "no",
                 "Johnny", "yes", "no",
                 "Patsy", "yes", "no",
                 "Yuma", "no", "yes")
```

```r
pets2 %>%
  gather(key="species", value="is_it", is_cat, is_dog) %>%
  filter(is_it == "yes") %>%
  select(-is_it) %>%
  mutate(species=recode(species,
                        is_cat="cat",
                        is_dog="dog"))
```

```
## # A tibble: 4 x 2
##   name   species
##   <chr>  <chr>
## 1 Daisy  cat
## 2 Johnny cat
## 3 Patsy  cat
## 4 Yuma   dog
```