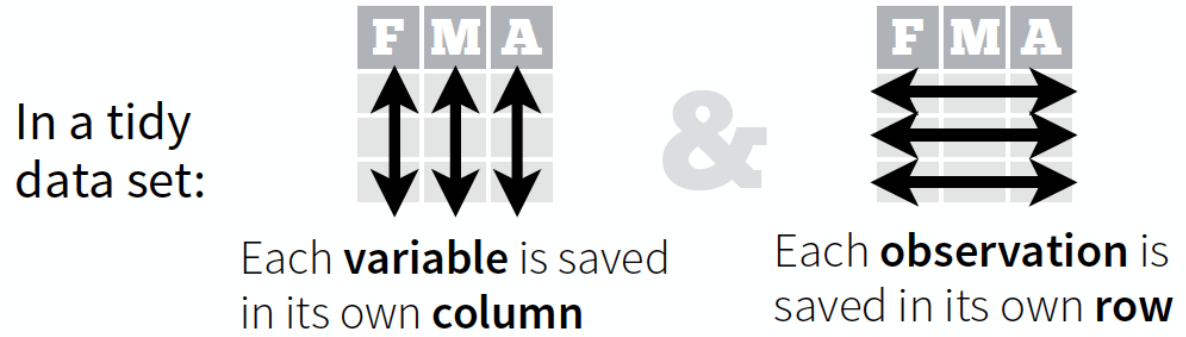
Post 01: Tidying Data Basics

Monica Wilkinson
10/31/2017

Introduction

In class, we have learned about general rules for Karl Broman's guidelines good data organization, but what do we do when we must deal with data that does not abide by these rules? In this tutorial, I will go over some of the basics for getting data into a more manageable format.

Tidy Data Concept



Many data wrangling packages in R provide functions according to a "tidy data" concept, where each column represents a *variable*, and each row represents and *observation*. Rstudio provides a data wrangling cheat sheet using `dplyr` (which we have used in this class) and `tidyr` (which we have not).

`tidyr` basics

`tidyr` has four main functions to help tidy data:

- `gather()` gathers multiple columns into fewer columns in what are called "key-value pairs".
- `spread()` takes two columns (key and value) and spreads into multiple columns
- `separate()` splits a single column into multiple columns
- `unite()` opposite of `separate()`, combines multiple columns into a single column

Another way to think about the `gather()` and `spread()` functions is that they change the form of the table from "wide" to "long" or vice versa.

`gather()`

Let's say you are a voracious coffee drinker, and, at a friend's suggestion, you have decied to keep track of how much coffee you are drinking. Perhaps if you are buying coffee at multiple coffee shops throughout the week, you might create a table that looks something like this:

```
# Create data frame where each row corresponds to the number of coffee cups purchased at each cafe that day
day <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")
elmwood <- c(0, 0, 0, 0, 0, 4, 0)
fsm <- c(1, 3, 1, 2, 0, 0, 0)
philz <- c(0, 0, 0, 0, 0, 0, 3)
strada <- c(1, 0, 1, 0, 2, 0, 0)
yalis <- c(1, 1, 1, 2, 1, 0, 0)

coffee <- data.frame(
  day,
  elmwood,
  fsm,
  philz,
  strada,
  yalis
)
coffee
```

```
##         day elmwood fsm philz strada yalis
## 1    Monday       0   1     0      1     1
## 2   Tuesday       0   3     0      0     1
## 3 Wednesday       0   1     0      1     1
## 4  Thursday       0   2     0      0     2
## 5    Friday       0   0     0      2     1
## 6  Saturday       4   0     0      0     0
## 7    Sunday       0   0     3      0     0
```

Now, this format is perfectly intuitive for a human reader, but it's a little difficult to do data analysis on it. Let's use `gather()` to get it into a better format.

`gather()`'s main arguments are:

- `data`: A data frame.
- `key`: A column name representing the new variable by which we are gathering the data. You can also think of it as the variable which encompasses the column names that are being gathered. In this case, we are setting it to `cafe`.
- `value`: A column name that describes the observations we are gathering into a new column. In this case, it is the `cups` of coffee bought at each cafe for each day.
- `...`: The columns you wish to apply the `gather()` function to. If left empty, all columns will be selected.
- `na.rm`: If `TRUE`, removes rows where the value column is `NA`. Default is `FALSE`.
- `convert`: If `TRUE`, will automatically convert character values to logical, integer, numeric, complex, or factor. Default is `FALSE`.

```
# Load tidyr package
library(tidyr)

# Gather coffee df by day and cafe
coffee_gathered <- gather(
  data = coffee,
  key = cafe,
  value = cups,
  elmwood:yalis,
)
head(coffee_gathered, 7)
```

```
##         day    cafe cups
## 1    Monday elmwood    0
## 2   Tuesday elmwood    0
## 3 Wednesday elmwood    0
## 4  Thursday elmwood    0
## 5    Friday elmwood    0
## 6  Saturday elmwood    4
## 7    Sunday elmwood    0
```

Our new data frame now has one row for each day-cafe pair. This format of data is more in line with the tidy data concept.

`spread()`

The `spread()` function is effectually the inverse of gather. It takes similar arguments:

- `data`: A data frame.
- `key`: The column containing the values to be converted to new columns. Same `key` as in `gather()`.
- `value`: The column containing values to be converted to the new columns' values. Same `value` as in `gather()`.
- `fill`: If there isn't a value for every key-value pair, this value will be substituted. This concept is similar to `na.rm` in `gather()`. Default is `NA`.
- `convert`: If `TRUE`, will automatically convert character values to logical, integer, numeric, complex, or factor. Default is `FALSE`.

Let's test it out on our `coffee_gathered` data.frame.

```
coffee_spread <- spread(
  data = coffee_gathered,
  key = cafe,
  value = cups
)
coffee_spread
```

```
##         day elmwood fsm philz strada yalis
## 1    Friday       0   0     0      2     1
## 2    Monday       0   1     0      1     1
## 3  Saturday       4   0     0      0     0
## 4    Sunday       0   0     3      0     0
## 5  Thursday       0   2     0      0     2
## 6   Tuesday       0   3     0      0     1
## 7 Wednesday       0   1     0      1     1
```

Because the `day` column is in a differnt order from the original `coffee` data.frame, the table looks a little different, but it is essentially the same table.

`separate()`

The `separate()` function can be used to separate columns containing multiple variables into multiple columns each containing a single variable.

For instance, let's say you come from a big family, and you've come up with the following data.frame to keep track of where all your relatives live so you know who to call if you ever need a place to crash:

```
who <- c("Aunt Marge", "Aunt Patricia", "Weird Uncle Frank", "Grandma #1", "Grandma #2", "Cousin Dan")
where <- c("Portland, OR", "Little Rock, AR", "Richmond, VA", "Santa Cruz, CA", "Richmond, CA", "Salem, OR"

relatives <- data.frame(who, where)
relatives
```

```
##                who           where
## 1        Aunt Marge    Portland, OR
## 2     Aunt Patricia Little Rock, AR
## 3 Weird Uncle Frank    Richmond, VA
## 4        Grandma #1  Santa Cruz, CA
## 5        Grandma #2    Richmond, CA
## 6        Cousin Dan       Salem, OR
```

In this format, we wouldn't be able to separate our relatives out by state! However, we can change this with the `separate()` function with these main arguments:

- `data` : A data frame.
- `col` : Name or position of the column you wish to split into multiple columns.
- `into` : Names of new variables to create *as a character vector*.
- `sep` : A character or numeric separator between columns. If numeric, it is interpreted as the position to split at starting from the far left of the string.
- `remove` : If `TRUE` , removes original column. Default is `TRUE` .
- `convert` : If `TRUE` , will automatically convert character values to logical, integer, numeric, complex, or factor. Default is `FALSE` .

```
relatives_separate <- separate(
  data = relatives,
  col = where,
  into = c('city', 'state'),
  sep = ", "
)
relatives_separate
```

```
##                who        city state
## 1        Aunt Marge    Portland    OR
## 2     Aunt Patricia Little Rock    AR
## 3 Weird Uncle Frank    Richmond    VA
## 4        Grandma #1  Santa Cruz    CA
## 5        Grandma #2    Richmond    CA
## 6        Cousin Dan       Salem    OR
```

`unite()`

`unite()` can be considered the inverse function of `separate()` . Perhaps you realize that once you have separated you `relatives` , cities with the same name in different states now look the same in your new `city` column. You decide that

you like the original format of your table better. We can do that with `unite()` !

`unite()` takes very similar arguments to `separate()` :

- `data` : A data frame.
- `col` : The nam of the new column.
- `...` : The columns to be merged.
- `sep` : Separator to use between values.
- `remove` : If `TRUE` , removes input column from new data frame. Default is `TRUE` .

```
relatives_unite <- unite(
  data = relatives_separate,
  col = where,
  city, state,
  sep = ", "
)
relatives_unite
```

```
##                 who           where
## 1        Aunt Marge    Portland, OR
## 2     Aunt Patricia Little Rock, AR
## 3 Weird Uncle Frank    Richmond, VA
## 4        Grandma #1  Santa Cruz, CA
## 5        Grandma #2    Richmond, CA
## 6        Cousin Dan       Salem, OR
```

More `dplyr`

While we covered the basics of `dplyr` in class, there are a few useful aspects that were not covered. For the following sections, let us use the following tables.

```
# Create 10 x 10 data.frame counting from 1 to 100, by row
hundred <- data.frame(
  one = seq(1, 91, 10),
  two = seq(2, 92, 10),
  three = seq(3, 93, 10),
  four = seq(4, 94, 10),
  five = seq(5, 95, 10),
  six = seq(6, 96, 10),
  seven = seq(7, 97, 10),
  eight = seq(8, 98, 10),
  nine = seq(9, 99, 10),
  oh = seq(10, 100, 10)
)
names(hundred) <- c("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "ten")
hundred
```

```
##    one two three four five six seven eight nine ten
## 1    1   2     3    4    5   6     7     8    9  10
## 2   11  12    13   14   15  16    17    18   19  20
## 3   21  22    23   24   25  26    27    28   29  30
## 4   31  32    33   34   35  36    37    38   39  40
## 5   41  42    43   44   45  46    47    48   49  50
## 6   51  52    53   54   55  56    57    58   59  60
## 7   61  62    63   64   65  66    67    68   69  70
## 8   71  72    73   74   75  76    77    78   79  80
## 9   81  82    83   84   85  86    87    88   89  90
## 10  91  92    93   94   95  96    97    98   99 100
```

```
# Create data.frame with a column containing letters of the Roman alphabet and
# a column for each letter's corresponding value

letter_vals <- data.frame(
  LETTER = c(LETTERS, ".", ",", "?", "!"),
  value = 1:30
)
letter_vals
```

```
##    LETTER value
## 1       A     1
## 2       B     2
## 3       C     3
## 4       D     4
## 5       E     5
## 6       F     6
## 7       G     7
## 8       H     8
## 9       I     9
## 10      J    10
## 11      K    11
## 12      L    12
## 13      M    13
## 14      N    14
## 15      O    15
## 16      P    16
## 17      Q    17
## 18      R    18
## 19      S    19
## 20      T    20
## 21      U    21
## 22      V    22
## 23      W    23
## 24      X    24
## 25      Y    25
## 26      Z    26
## 27      .    27
## 28      ,    28
## 29      ?    29
## 30      !    30
```

```r
# Create a data.frame with a column containing capital Roman letters, a column
# containing lowercase Roman letters, and a column containing two examples of
# words starting with each letter

ex_list1 <- c("Alpha", "Bravo", "Charlie", "Delta", "Echo", "Foxtrot", "Golf", "Hotel", "India", "Juliett",

# Does not contain I, U, V, X, Z
ex_list2 <- c("Apple", "Banana", "Cherry", "Date", "Elderberry", "Fig", "Grape", "Honeydew", "Jackfruit", '

letter_ex <- data.frame(
  LETTER = c(LETTERS, LETTERS[c(1:8, 10:20, 23, 25)]),
  letter = c(letters, letters[c(1:8, 10:20, 23, 25)]),
  example = c(ex_list1, ex_list2)
)
letter_ex
```

```
##    LETTER letter    example
## 1       A      a      Alpha
## 2       B      b      Bravo
## 3       C      c    Charlie
## 4       D      d      Delta
## 5       E      e       Echo
## 6       F      f    Foxtrot
## 7       G      g       Golf
## 8       H      h      Hotel
## 9       I      i      India
## 10      J      j    Juliett
## 11      K      k       Kilo
## 12      L      l       Lima
## 13      M      m       Mike
## 14      N      n   November
## 15      O      o      Oscar
## 16      P      p       Papa
## 17      Q      q     Quebec
## 18      R      r      Romeo
## 19      S      s     Sierra
## 20      T      t      Tango
## 21      U      u    Uniform
## 22      V      v     Victor
## 23      W      w    Whiskey
## 24      X      x      X-ray
## 25      Y      y     Yankee
## 26      Z      z       Zulu
## 27      A      a      Apple
## 28      B      b     Banana
## 29      C      c     Cherry
## 30      D      d       Date
## 31      E      e Elderberry
## 32      F      f        Fig
## 33      G      g      Grape
## 34      H      h   Honeydew
## 35      J      j  Jackfruit
## 36      K      k       Kiwi
## 37      L      l     Lychee
## 38      M      m      Mango
## 39      N      n  Nectarine
## 40      O      o     Orange
## 41      P      p  Persimmon
## 42      Q      q     Quince
## 43      R      r  Raspberry
## 44      S      s Strawberry
## 45      T      t   Tamarind
## 46      W      w Watermelon
## 47      Y      y       Yuzu
```

`filter_all()`, `filter_if()`, `filter_at()`

These functions use predicate expressions and functions to filter across multiple columns in a table.

These functions take some combination of the following arguments:

- `tbl`: A table.
- `.vars`: A character vector of column names, a numeric vector of column positions, or the output of `vars()`. You can pass additional functions into the `vars()` function to select columns that start with, end with, or contain certain phrases.
- `.predicate`: A predicate function which selects columns for which the function returns `TRUE`, after which the filter function is actually applied.
- `.vars_predicate`: The filter criteria which is then applied to the table. This argument is in the form of the functions `all_vars()` or `any_vars()`, which take a logical comparison using `.` as a pronoun for all variables.

While this sounds quite complicated, the functions actually read quite like prose:

- `filter_all(tbl, .vars_predicate)`: "Filter over **all** columns where x is true"
- `filter_if(tbl, .predicate, .vars_predicate)`: "Filter **if** this column criteria is met where x is true"
- `filter_at(tbl, .vars, .vars_predicate)`: "Filter **at** these column names where x is true"

```
# Load dplyr package
library(dplyr)

# Filter over all columns where x > 24
# Note that we must use any_vars() here instead of all_vars() because we have
# one column that is not numeric and would therefore return an empty table.
filter_all(hundred, any_vars(. > 24))
```

```
##   one two three four five six seven eight nine ten
## 1  21  22    23   24   25  26    27    28   29  30
## 2  31  32    33   34   35  36    37    38   39  40
## 3  41  42    43   44   45  46    47    48   49  50
## 4  51  52    53   54   55  56    57    58   59  60
## 5  61  62    63   64   65  66    67    68   69  70
## 6  71  72    73   74   75  76    77    78   79  80
## 7  81  82    83   84   85  86    87    88   89  90
## 8  91  92    93   94   95  96    97    98   99 100
```

```
# Filter if the column is numeric where x > 50
filter_if(hundred, is.numeric, all_vars(. > 50))
```

```
##   one two three four five six seven eight nine ten
## 1  51  52    53   54   55  56    57    58   59  60
## 2  61  62    63   64   65  66    67    68   69  70
## 3  71  72    73   74   75  76    77    78   79  80
## 4  81  82    83   84   85  86    87    88   89  90
## 5  91  92    93   94   95  96    97    98   99 100
```

```
# Filter at columns starting with "f" containing a multiple of 3
filter_at(hundred, vars(starts_with("f")), any_vars(. %% 3 == 0))
```

```
##   one two three four five six seven eight nine ten
## 1  11  12    13   14   15  16    17    18   19  20
## 2  21  22    23   24   25  26    27    28   29  30
## 3  41  42    43   44   45  46    47    48   49  50
## 4  51  52    53   54   55  56    57    58   59  60
## 5  71  72    73   74   75  76    77    78   79  80
## 6  81  82    83   84   85  86    87    88   89  90
```

Note that all rows of the table are returned, even the rows which are not part of the filtering criteria.

While we have just explored these additional forms of `filter()`, there are also similar forms of `select()` and `mutate()`.

`select()` helper functions

In addition to what we have learned, `select()` also takes a few helper functions as arguments:

- `contains("x")`: Select columns whose name contains this character string.
- `starts_with("x")`: Select columns whose name starts with this character string.
- `ends_with("x")`: Select columns whose name ends with this character string.
- `matches("/.")`: Select columns whose name matches this *regular expression*.
- `num_range("prefix", range)`: Select columns whose names are in the form of a prefix and number, i.e. "x1", "x2", "x3".
- `one_of(c("Column", "Names"))`:
- `everything()`: Selects every column.

All of these functions also take these additional arguments:

- `ignore.case`: If `TRUE`, ignores case when matching names. Default is `TRUE`.
- `vars`: A character vector of the column names on which to apply the helper function. Default is all columns.

Here are some examples.

```
# ------------------------------------------------------------------------------
# Select columns whose name contains "tt"
select(letter_vals, contains("tt"))
```

```
##    LETTER
## 1       A
## 2       B
## 3       C
## 4       D
## 5       E
## 6       F
## 7       G
## 8       H
## 9       I
## 10      J
## 11      K
## 12      L
## 13      M
## 14      N
## 15      O
## 16      P
## 17      Q
## 18      R
## 19      S
## 20      T
## 21      U
## 22      V
## 23      W
## 24      X
## 25      Y
## 26      Z
## 27      .
## 28      ,
## 29      ?
## 30      !
```

```
# Select columns starting with "let", ignoring case
select(letter_ex, starts_with("let"))
```

```
##    LETTER letter
## 1       A      a
## 2       B      b
## 3       C      c
## 4       D      d
## 5       E      e
## 6       F      f
## 7       G      g
## 8       H      h
## 9       I      i
## 10      J      j
## 11      K      k
## 12      L      l
## 13      M      m
## 14      N      n
## 15      O      o
## 16      P      p
## 17      Q      q
## 18      R      r
## 19      S      s
## 20      T      t
## 21      U      u
## 22      V      v
## 23      W      w
## 24      X      x
## 25      Y      y
## 26      Z      z
## 27      A      a
## 28      B      b
## 29      C      c
## 30      D      d
## 31      E      e
## 32      F      f
## 33      G      g
## 34      H      h
## 35      J      j
## 36      K      k
## 37      L      l
## 38      M      m
## 39      N      n
## 40      O      o
## 41      P      p
## 42      Q      q
## 43      R      r
## 44      S      s
## 45      T      t
## 46      W      w
## 47      Y      y
```

```
# Selects columns whose names are contained in the cols vector
# This should give an warning because "something else" is not the name of a
# column in this data frame.
cols <- c("letter", "LETTER", "something else")
select(letter_ex, one_of(cols))
```

```
##    letter LETTER
## 1       a      A
## 2       b      B
## 3       c      C
## 4       d      D
## 5       e      E
## 6       f      F
## 7       g      G
## 8       h      H
## 9       i      I
## 10      j      J
## 11      k      K
## 12      l      L
## 13      m      M
## 14      n      N
## 15      o      O
## 16      p      P
## 17      q      Q
## 18      r      R
## 19      s      S
## 20      t      T
## 21      u      U
## 22      v      V
## 23      w      W
## 24      x      X
## 25      y      Y
## 26      z      Z
## 27      a      A
## 28      b      B
## 29      c      C
## 30      d      D
## 31      e      E
## 32      f      F
## 33      g      G
## 34      h      H
## 35      j      J
## 36      k      K
## 37      l      L
## 38      m      M
## 39      n      N
## 40      o      O
## 41      p      P
## 42      q      Q
## 43      r      R
## 44      s      S
## 45      t      T
## 46      w      W
## 47      y      Y
```

Different types of joins

We have learned the `base::merge()` and `dplyr::join()` functions already, which work just fine when joining two tables where the columns by which we wish to join contain all the same values, but `dplyr` provides a few more functions for when this is not the case. This may be useful when your data is in multiple files.

- Mutating joins:
  - `left_join(df1, df2, by = "col")` : Join only rows in `df2` that have a match in `df1` in column `"col"`. Non-matching values will result in an `NA` in the new column(s).
  - `right_join(df1, df2, by = "col")` : Join only rows in `df1` that have a match in `df2` in column `"col"`. Non-matching values will result in an `NA` in the new column(s).
  - `inner_join(df1, df2, by = "col")` : Join only rows that are in both `df1` and `df2` in column `"col"`. Rows without a match will be eliminated.
  - `full_join(df1, df2, by = "col")` : Join all rows. Non-matching values will result in an `NA` in both the new and old column(s).
- Filtering Joins:
  - `semi_join(df1, df2, by = "col")` : Filters rows in `df1` that have a match in `df2`. Similar to `inner_join()`, but does not add columns from `df2` to `df1`.
  - `anti_join(df1, df2, by = "col")` : Filters rows from `df1` that do not have a match in `df2`.

Note that by default, these functions will automatically find all variables with matching names in the two tables if the `by` argument is not specified. If the column by which you are joining does not have the same name in the two tables, you can use `by = c("col1" = "col2")` where `"col"` is the column name in `df1` and `"col2"` is the column name in `df2`.

Here are some examples:

```
# Join letter_ex and letter_vals, keeping only values represented in letter_vals.
left_join(letter_ex, letter_vals, by = "LETTER")
```

```
##    LETTER letter    example value
## 1       A      a      Alpha     1
## 2       B      b      Bravo     2
## 3       C      c    Charlie     3
## 4       D      d      Delta     4
## 5       E      e       Echo     5
## 6       F      f    Foxtrot     6
## 7       G      g       Golf     7
## 8       H      h      Hotel     8
## 9       I      i      India     9
## 10      J      j    Juliett    10
## 11      K      k       Kilo    11
## 12      L      l       Lima    12
## 13      M      m       Mike    13
## 14      N      n   November    14
## 15      O      o      Oscar    15
## 16      P      p       Papa    16
## 17      Q      q     Quebec    17
## 18      R      r      Romeo    18
## 19      S      s     Sierra    19
## 20      T      t      Tango    20
## 21      U      u    Uniform    21
## 22      V      v     Victor    22
## 23      W      w    Whiskey    23
## 24      X      x      X-ray    24
## 25      Y      y     Yankee    25
## 26      Z      z       Zulu    26
## 27      A      a      Apple     1
## 28      B      b     Banana     2
## 29      C      c     Cherry     3
## 30      D      d       Date     4
## 31      E      e Elderberry     5
## 32      F      f        Fig     6
## 33      G      g      Grape     7
## 34      H      h   Honeydew     8
## 35      J      j  Jackfruit    10
## 36      K      k       Kiwi    11
## 37      L      l     Lychee    12
## 38      M      m      Mango    13
## 39      N      n  Nectarine    14
## 40      O      o     Orange    15
## 41      P      p  Persimmon    16
## 42      Q      q     Quince    17
## 43      R      r  Raspberry    18
## 44      S      s Strawberry    19
## 45      T      t   Tamarind    20
## 46      W      w Watermelon    23
## 47      Y      y       Yuzu    25
```

```
# Full join of letter_ex and letter_vals. Note where there are NA values.
full_join(letter_ex, letter_vals)
```

```
##    LETTER letter    example value
## 1       A      a      Alpha     1
## 2       B      b      Bravo     2
## 3       C      c    Charlie     3
## 4       D      d      Delta     4
## 5       E      e       Echo     5
## 6       F      f    Foxtrot     6
## 7       G      g       Golf     7
## 8       H      h      Hotel     8
## 9       I      i      India     9
## 10      J      j    Juliett    10
## 11      K      k       Kilo    11
## 12      L      l       Lima    12
## 13      M      m       Mike    13
## 14      N      n   November    14
## 15      O      o      Oscar    15
## 16      P      p       Papa    16
## 17      Q      q     Quebec    17
## 18      R      r      Romeo    18
## 19      S      s     Sierra    19
## 20      T      t      Tango    20
## 21      U      u    Uniform    21
## 22      V      v     Victor    22
## 23      W      w    Whiskey    23
## 24      X      x      X-ray    24
## 25      Y      y     Yankee    25
## 26      Z      z       Zulu    26
## 27      A      a      Apple     1
## 28      B      b     Banana     2
## 29      C      c     Cherry     3
## 30      D      d       Date     4
## 31      E      e Elderberry     5
## 32      F      f        Fig     6
## 33      G      g      Grape     7
## 34      H      h   Honeydew     8
## 35      J      j  Jackfruit    10
## 36      K      k       Kiwi    11
## 37      L      l     Lychee    12
## 38      M      m      Mango    13
## 39      N      n  Nectarine    14
## 40      O      o     Orange    15
## 41      P      p  Persimmon    16
## 42      Q      q     Quince    17
## 43      R      r  Raspberry    18
## 44      S      s Strawberry    19
## 45      T      t   Tamarind    20
## 46      W      w Watermelon    23
## 47      Y      y       Yuzu    25
## 48      .   <NA>       <NA>    27
## 49      ,   <NA>       <NA>    28
## 50      ?   <NA>       <NA>    29
## 51      !   <NA>       <NA>    30
```

```
# Select only the rows in letter_vals that are not in letter_ex (punctuation)
anti_join(letter_vals, letter_ex)
```

```
##   LETTER value
## 1      .    27
## 2      ,    28
## 3      ?    29
## 4      !    30
```

Set Operations and Binding

Lastly, `dplyr` provides set operation functions for filtering rows between tables, and binding functions for combining tables.

- Set Operations
  - `intersect(df1, df2)` : Selects rows that appear in both `df1` and `df2`. Similar to `semi_join()`, but across all rows.
  - `union(df1, df2)` : Appends rows in `df2` that are not already represented in `df1`
  - `setdiff(df1, df2)` : Selects rows that appear in `df1` but not `df2`. Similar to `anti_join()`, but across all

rows.

- Binding
  - `bind_rows(df1, df2)` : Appends `df2` as new rows in `df1` . Missing columns will be filled with NA.
  - `bind_cols(df1, df2)` : Adds `df2` as new columns in `df1` . Both data frames must have the same number of rows.

Other packages

`reshape2` basics

The `reshape2` package is similar to the `tidyr` package in that its main functionality is to change a table from a "long" format to a "short" format or vice versa. `reshape2::melt()` is similar to `tidyr::gather()` , and `reshape2::cast()` is similar to `tidyr::spread()` . The idea behind the names of these functions is akin to the visuals of melting and casting metal: melting metal drips and becomes long, casting this molten metal makes it wide again.

`melt()`

`melt()` takes the following arguments:

- `data` : A data set. Melt goes not need any further arguments by default, and can deduce the `key` and `value` arguments that needed to be specified in `gather()` .
- `variable.name` : A character string that will be the column name for the column containing the variables which were column names in the long format. Similar to the `key` argument of the `gather()` function.
- `value.name` : A character string that will be the name of the column containing the values. Similar to the `value` argument of the `gather()` function.
- `id.vars` : A character vector containing the columns which will serve as the identifier for each row in our melted table.
- `na.rm` : If `TRUE` , removes rows where the value column is `NA` . Default is `FALSE` .

Let's go back to our `coffee` table.

```
# Load reshape2 package
library(reshape2)

coffee_melted <- melt(
  data = coffee,
  variable.name = "cafe",
  value.name = "cups",
  id.vars = "day"
)
coffee_melted
```

```
##          day    cafe cups
## 1     Monday elmwood    0
## 2    Tuesday elmwood    0
## 3  Wednesday elmwood    0
## 4   Thursday elmwood    0
## 5     Friday elmwood    0
## 6   Saturday elmwood    4
## 7     Sunday elmwood    0
## 8     Monday     fsm    1
## 9    Tuesday     fsm    3
## 10 Wednesday     fsm    1
## 11  Thursday     fsm    2
## 12    Friday     fsm    0
## 13  Saturday     fsm    0
## 14    Sunday     fsm    0
## 15    Monday   philz    0
## 16   Tuesday   philz    0
## 17 Wednesday   philz    0
## 18  Thursday   philz    0
## 19    Friday   philz    0
## 20  Saturday   philz    0
## 21    Sunday   philz    3
## 22    Monday  strada    1
## 23   Tuesday  strada    0
## 24 Wednesday  strada    1
## 25  Thursday  strada    0
## 26    Friday  strada    2
## 27  Saturday  strada    0
## 28    Sunday  strada    0
## 29    Monday   yalis    1
## 30   Tuesday   yalis    1
## 31 Wednesday   yalis    1
## 32  Thursday   yalis    2
## 33    Friday   yalis    1
## 34  Saturday   yalis    0
## 35    Sunday   yalis    0
```

```
# Note that this is identical to our coffee_gathered table, except that the cafe
# column is a factor in this one
coffee_gathered
```

```
##          day    cafe cups
## 1     Monday elmwood    0
## 2    Tuesday elmwood    0
## 3  Wednesday elmwood    0
## 4   Thursday elmwood    0
## 5     Friday elmwood    0
## 6   Saturday elmwood    4
## 7     Sunday elmwood    0
## 8     Monday     fsm    1
## 9    Tuesday     fsm    3
## 10 Wednesday     fsm    1
## 11  Thursday     fsm    2
## 12    Friday     fsm    0
## 13  Saturday     fsm    0
## 14    Sunday     fsm    0
## 15    Monday   philz    0
## 16   Tuesday   philz    0
## 17 Wednesday   philz    0
## 18  Thursday   philz    0
## 19    Friday   philz    0
## 20  Saturday   philz    0
## 21    Sunday   philz    3
## 22    Monday  strada    1
## 23   Tuesday  strada    0
## 24 Wednesday  strada    1
## 25  Thursday  strada    0
## 26    Friday  strada    2
## 27  Saturday  strada    0
## 28    Sunday  strada    0
## 29    Monday   yalis    1
## 30   Tuesday   yalis    1
## 31 Wednesday   yalis    1
## 32  Thursday   yalis    2
## 33    Friday   yalis    1
## 34  Saturday   yalis    0
## 35    Sunday   yalis    0
```

`cast()`

Actually, `cast` is a group of functions, `acast()`, which is for vector/matrix/array output and `dcast()`, which is for data frame output. Since we have been working primarily with data frames, let's work with `dcast()`

Rather than "traditional" arguments, the `cast` functions take a "casting formula" in the form of `xvar1 + xvar2 ~ yvar1 + yvar2 ~ zvar ~ ...`.

```
# Cast coffee_melted table back into the coffee table.
dcast(coffee_melted, day ~ cafe)
```

```
##         day elmwood fsm philz strada yalis
## 1    Friday       0   0     0      2     1
## 2    Monday       0   1     0      1     1
## 3  Saturday       4   0     0      0     0
## 4    Sunday       0   0     3      0     0
## 5  Thursday       0   2     0      0     2
## 6   Tuesday       0   3     0      0     1
## 7 Wednesday       0   1     0      1     1
```

References

1. http://kbroman.org/dataorg/
2. https://www.rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
3. https://rpubs.com/bradleyboehmke/data_wrangling
4. http://r4ds.had.co.nz/relational-data.html
5. https://www.rdocumentation.org/packages/dplyr/versions/0.7.3/topics/filter_all
6. https://b-rodrigues.github.io/fput/tidyverse.html
7. http://seananderson.ca/2013/10/19/reshape.html
8. R documentation from the help pane of Rstudio