Data management with dplyr, tidyr, and reshape2

Shane T. Mueller shanem@mtu.edu

2021-01-14

Data Management Libraries

In recent years, RStudio has spearheaded development of a series of libraries that make data refactoring, selecting, and management simple and fast for large data sets. Many of these tools are equiva lent to what you can do using selection, sorting, aggregate, and tapply of normal data frames. Some of them offer very useful capabilities that are otherwise very difficult to manage. Most of these are developed by Hadley Wickham, who also created ggplot2. Part of the reason for the proliferation of libraries is the philosophy to not break what people rely on, and so when improved functionality is made, a new library is created so that compatibility can be broken without harming anyone relying on certain functionality.

Some relevant libraries include:

plyr and dplyr

These libraries are sets of tools for splitting, applying, and combining data. The goal is to have a coherent set of tools for breaking down data into smaller pieces, operating on each chunk of data and reassembling them–an idiom called "split-apply-combine".

dplyr is a successor to plyr, written to be much faster, to integrae with remote databases, but it works only with data frames. The dplyr library seems to be better supported, and tests show it can be more than a hundred times faster than plyr.

reshape, reshape2 and tidyr

The reshape2 library is a 'reboot' of reshape, that is faster and better. These libraries allow easily transforming a data set from 'long' to 'wide' format and back again. That is, you can take a data set with multiple columns you are treating as distinct DVs, and reframe the data set so they are both in a single DV column, with a separate column specifiying which level of IV a row belongs to. The tidyr library is the newest entry into data management libraries, also by Wickham, and is described as an "evolution" of reshape2.

Magrittr and forward-piping

Although most of the functions in these libraries can be used like normal functions, it is common to compose a set of operations that are applied in sequence, where the output of one function is used as the input of another function. Most of the tidyverse work together with a library called magrittr ("Ceci n'est pas un pipe") to support a code-efficient way of doing this. It works by putting the output of one function into the first argument of the next function using the `%>%` operator.

Forward-piping with Magrittr

I will not often use forward-piping in this class, but you should recognize it when it does get used. Piping can be useful when you think about a series of operations you want to perform on a single data set. Magrittr

supports this mainly through an operator %>%, but there are a few others supported by the library. The operator basically replaces nested function calls. The following two ways of applying f1 and f2 are equivalent:

```
library(magrittr)

f1 <- function(x) {
    abs(x)
}
f2 <- function(x) {
    sqrt(x)
}

a <- f1(-33.2)
b <- f2(a)

f2(f1(-33.2))
```

```
[1] 5.761944
```

```
(-33.2) %>% f1 %>% f2
```

```
[1] 5.761944
```

Here, these seem about equivalent in complexity of writing, but when you have 5 or 6 operations, having to either make intermediate variables or make sure all your parentheses are properly matched can get a bit tedious.

You can specify other arguments of a multi-argument function by using . to denote the piped-in value. Here, we round 10 random values to 3 decimal places:

```
rnorm(10) %>% round(., 3)
```

```
 [1] -0.091  0.328 -0.370  0.905  0.113 -0.619  1.281  0.771  2.090  0.646
```

```
rnorm(10) %>% round(3)
```

```
 [1]  0.573 -1.265 -0.051  0.161 -0.317 -0.074 -0.040 -1.971 -0.018  0.789
```

```
sample(1:5, replace = T, size = 10) %>% round(runif(10), .)
```

```
 [1] 0.38790 0.20000 0.11727 0.37030 0.46432 0.28078 0.01003 0.23900 0.25230
[10] 0.40000
```

Finally, R can have its assignment arrow go in either direction, like below:

```
value <- runif(100) %>% sd %>% sqrt %>% log
value <- runif(100) %>% sd %>% sqrt %>% log
value
```

```
[1] -0.6060166
```

Overview of dplyr

The following creates a couple data sets for use in these examples:

```
dat0 <- data.frame(sub = c(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4), question = c("a",
    "b", "c", "a", "b", "c", "a", "b", "c", "a", "b", "c"), dv = c(5, 3, 1, 2, 3,
    6, 4, 2, 3, 1, 3, 5))
```

```
dat <- data.frame(sub = sample(letters, 100, replace = T), cond = sample(c("A", "B",
    "C"), 100, replace = T), group = sample(1:10, 100, replace = T), dv1 = runif(100) *
    5)
```

The dplyr library implements a number of functions that are available in one form or another within R, but may be difficult to use, inconsistent, or slow.

The dplyr library does not create side-effects. That is, it always makes a copy of your original data and returns it, rather than altering the form of your original data. Consequently, you need to usually assign the outcome to a new variable. Sometimes, it is acceptable to assign it to its old name, as in the following:

```
library(dplyr)
data <- dat
dplyr::filter(data, sub == "b")
```

```
  sub cond group       dv1
1   b    C     8 4.6416069
2   b    A    10 2.9888271
3   b    A     6 1.3821269
4   b    A     8 0.3403095
5   b    A     8 3.9516492
6   b    B     4 3.1677131
```

```
data <- filter(data, sub == "b")
head(data)
```

```
  sub cond group       dv1
1   b    C     8 4.6416069
2   b    A    10 2.9888271
3   b    A     6 1.3821269
4   b    A     8 0.3403095
5   b    A     8 3.9516492
6   b    B     4 3.1677131
```

However, this is often not the best practice, because it means that the data variable depends on whether you have run some code or not.

slice and filter

The following use dplyr to rearrange and filter rows of a data frame. `filter` picks out rows based on a boolean vector of the same size (number of rows)

```
head((dat$sub == "b"))  ##shows the first 6 elements of the boolean
```

```
[1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

```
filter(dat, sub == "b")  ##use filter to pick out just tho subject B rows
```

```
  sub cond group       dv1
1   b    C     8 4.6416069
2   b    A    10 2.9888271
3   b    A     6 1.3821269
4   b    A     8 0.3403095
5   b    A     8 3.9516492
6   b    B     4 3.1677131
```

Similarly, slice allows you to do this based on the row index (number)

```
slice(dat, 1)  ##first row
```

```
  sub cond group      dv1
1   b    C     8 4.641607
```

```
slice(dat, 2:10)  ##9 rows after the first
```

```
  sub cond group      dv1
1   g    A     2 2.526877
2   u    B     9 3.745631
3   v    A     1 1.630290
4   t    C     9 1.296970
5   n    B     7 3.959520
6   q    A     5 4.922692
7   q    C     6 4.679532
8   b    A    10 2.988827
9   q    B     9 4.596968
```

```
slice(dat, 1:20 * 2)  ##even rows 2..40
```

```
   sub cond group       dv1
1    g    A     2 2.5268769
2    v    A     1 1.6302901
3    n    B     7 3.9595198
4    q    C     6 4.6795324
5    q    B     9 4.5969681
6    s    A     1 4.4623555
7    p    A     7 1.6781015
8    l    B    10 0.2785777
9    e    C     2 4.9400149
10   w    B    10 0.4030141
11   x    C     3 3.4626379
12   s    B     5 3.0545895
13   k    A     5 0.5882065
14   g    B     7 0.3046925
15   d    C     7 1.5462854
16   z    C     8 1.3228325
17   y    A     3 2.5684641
18   b    A     6 1.3821269
 [ reached 'max' / getOption("max.print") -- omitted 2 rows ]
```

```
slice(dat, -1)
```

```
   sub cond group         dv1
1    g    A     2 2.526876862
2    u    B     9 3.745630565
3    v    A     1 1.630290060
4    t    C     9 1.296969750
5    n    B     7 3.959519783
6    q    A     5 4.922691828
7    q    C     6 4.679532404
8    b    A    10 2.988827146
9    q    B     9 4.596968099
10   x    A     9 2.466899226
11   s    A     1 4.462355511
12   y    B     9 2.600431531
```

```
13   p    A     7 1.678101493
14   p    B     4 0.001471081
15   l    B    10 0.278577667
16   e    C     9 1.162890925
17   e    C     2 4.940014919
18   x    B     4 0.318043083
 [ reached 'max' / getOption("max.print") -- omitted 81 rows ]
```

---

arrange()

The `arrange` function reorders the rows by the levels of a specific factor

```
arrange(dat, sub)
```

```
   sub cond group       dv1
1    a    B     5 1.0404293
2    a    C     7 2.2799050
3    a    A     7 2.7496770
4    a    C     9 1.0445277
5    b    C     8 4.6416069
6    b    A    10 2.9888271
7    b    A     6 1.3821269
8    b    A     8 0.3403095
9    b    A     8 3.9516492
10   b    B     4 3.1677131
11   c    B     1 4.3777077
12   c    A     9 4.2183914
13   c    C     9 1.6489995
14   d    C     7 1.5462854
15   d    A     2 0.6767127
16   d    C     2 1.9933575
17   d    C     7 4.0561043
18   e    C     9 1.1628909
 [ reached 'max' / getOption("max.print") -- omitted 82 rows ]
```

```
arrange(dat, sub, group)
```

```
   sub cond group       dv1
1    a    B     5 1.0404293
2    a    C     7 2.2799050
3    a    A     7 2.7496770
4    a    C     9 1.0445277
5    b    B     4 3.1677131
6    b    A     6 1.3821269
7    b    C     8 4.6416069
8    b    A     8 0.3403095
9    b    A     8 3.9516492
10   b    A    10 2.9888271
11   c    B     1 4.3777077
12   c    A     9 4.2183914
13   c    C     9 1.6489995
14   d    A     2 0.6767127
15   d    C     2 1.9933575
16   d    C     7 1.5462854
```

```
17   d    C     7 4.0561043
18   e    C     2 4.9400149
 [ reached 'max' / getOption("max.print") -- omitted 82 rows ]
```

```
dat %>% arrange(sub, cond) %>% filter(dv1 > 1)
```

```
   sub cond group      dv1
1    a    A     7 2.749677
2    a    B     5 1.040429
3    a    C     7 2.279905
4    a    C     9 1.044528
5    b    A    10 2.988827
6    b    A     6 1.382127
7    b    A     8 3.951649
8    b    B     4 3.167713
9    b    C     8 4.641607
10   c    A     9 4.218391
11   c    B     1 4.377708
12   c    C     9 1.649000
13   d    C     7 1.546285
14   d    C     2 1.993357
15   d    C     7 4.056104
16   e    A     2 2.299132
17   e    B     6 1.389595
18   e    B     2 3.582787
 [ reached 'max' / getOption("max.print") -- omitted 62 rows ]
```

---

select()

- The `select` function picks out **columns** by name

```
select(dat0, sub, dv)
```

```
   sub dv
1    1  5
2    1  3
3    1  1
4    2  2
5    2  3
6    2  6
7    3  4
8    3  2
9    3  3
10   4  1
11   4  3
12   4  5
```

```
select(dat0, sub:dv)
```

```
   sub question dv
1    1        a  5
2    1        b  3
3    1        c  1
4    2        a  2
5    2        b  3
```

```
6    2        c  6
7    3        a  4
8    3        b  2
9    3        c  3
10   4        a  1
11   4        b  3
12   4        c  5
```

```
select(dat0, -question)
```

```
   sub dv
1    1  5
2    1  3
3    1  1
4    2  2
5    2  3
6    2  6
7    3  4
8    3  2
9    3  3
10   4  1
11   4  3
12   4  5
```

```
# pping example: filter sub 4 and select just dv value.
dat0 %>% filter(sub == 4) %>% select(dv)
```

```
  dv
1  1
2  3
3  5
```

There are a lot of matching functions that can be used within select:

```
select(dat0, starts_with("s"))
```

```
   sub
1    1
2    1
3    1
4    2
5    2
6    2
7    3
8    3
9    3
10   4
11   4
12   4
```

This function can be very handy for situations like survey data where you have dozens or hundreds of columns/variables. You may be interested in just a few of these, and select will pick these out.

---

rename()

- The `rename` function renames columns.

```
rename(dat0, participant = sub)
```

```
   participant question dv
1            1        a  5
2            1        b  3
3            1        c  1
4            2        a  2
5            2        b  3
6            2        c  6
7            3        a  4
8            3        b  2
9            3        c  3
10           4        a  1
11           4        b  3
12           4        c  5
```

---

distinct()

- The `distinct` function finds distinct combinations of values (typically IVs). This is similar to doing a table, or identifying the `levels` of a factor.

```
dat2 <- data.frame(a = sample(1:10, 20, replace = T), b = sample(c(100, 200, 300),
    20, replace = T))
distinct(dat2)
```

```
    a   b
1   4 200
2   5 300
3   1 300
4   6 100
5   6 300
6   7 300
7   8 100
8   2 200
9   1 200
10  2 300
11  7 100
12 10 200
13  2 100
14  3 300
15  1 100
```

You can also specify specific variables you wish to use:

```
distinct(dat, sub)
```

```
   sub
1    b
2    g
3    u
4    v
5    t
6    n
7    q
```

```
8    x
9    s
10   y
11   p
12   l
13   e
14   w
15   k
16   h
17   d
18   z
19   j
20   f
21   r
22   m
23   a
24   c
25   o
```

Retain all columns of distinct data:

```
distinct(dat, sub, .keep_all = T)
```

```
   sub cond group       dv1
1    b    C     8 4.6416069
2    g    A     2 2.5268769
3    u    B     9 3.7456306
4    v    A     1 1.6302901
5    t    C     9 1.2969697
6    n    B     7 3.9595198
7    q    A     5 4.9226918
8    x    A     9 2.4668992
9    s    A     1 4.4623555
10   y    B     9 2.6004315
11   p    A     7 1.6781015
12   l    B    10 0.2785777
13   e    C     9 1.1628909
14   w    B    10 0.4030141
15   k    A     5 0.5882065
16   h    C     4 4.1019006
17   d    C     7 1.5462854
18   z    C     8 1.3228325
 [ reached 'max' / getOption("max.print") -- omitted 7 rows ]
```

mutate() and transmute()

- The `mutate` function adds a column that is a function of other columns. Transmute does the same thing, but returns only the new variable. This can be really useful for creating summarized data, composite values of ratings scales, and the like.

```
## reverse code a scale
dat1 <- mutate(dat0, newdv = 6 - dv)
```

More complex mutations are possible:

```
dat1$newdv2 = dat1$dv * dat1$newdv
```

```
mutate(dat1, newdv2 = dv * newdv)
```

```
   sub question dv newdv newdv2
1    1        a  5     1      5
2    1        b  3     3      9
3    1        c  1     5      5
4    2        a  2     4      8
5    2        b  3     3      9
6    2        c  6     0      0
7    3        a  4     2      8
8    3        b  2     4      8
9    3        c  3     3      9
10   4        a  1     5      5
11   4        b  3     3      9
12   4        c  5     1      5
```

```
transmute(dat1, newdv2 = dv * newdv)
```

```
   newdv2
1       5
2       9
3       5
4       8
5       9
6       0
7       8
8       8
9       9
10      5
11      9
12      5
```

---

merging and joining

dplyr has a lot of functions to merge data frames, and these are especially useful when you may not have an exact match between the levels (so you cant just do a cbind)

```
A <- data.frame(sub = c("A", "B", "C", "E"), data1 = 1:4)
B <- data.frame(sub = c("A", "B", "D", "F"), data2 = 11:14)
```

- `left_join(A,B)` Joins everything into A that is in B

```
left_join(A, B, by = "sub")
```

```
  sub data1 data2
1   A     1    11
2   B     2    12
3   C     3    NA
4   E     4    NA
```

- `right_join(A,B)`

```
right_join(A, B, by = "sub")
```

```
  sub data1 data2
1   A     1    11
```

```
2   B     2    12
3   D    NA    13
4   F    NA    14
```

*`inner_join(A,B)`

```
inner_join(A, B, by = "sub")
```

```
  sub data1 data2
1   A     1    11
2   B     2    12
```

*`full_join(A,B)` adds all data, incorporating NAs when one or the other are missing.

```
full_join(A, B, by = "sub")
```

```
  sub data1 data2
1   A     1    11
2   B     2    12
3   C     3    NA
4   E     4    NA
5   D    NA    13
6   F    NA    14
```

*"semi_join picks out just the first argument for variables where both exist; `anti_join` picks out the first argument for those where the second doesn't exist. These can be useful for imputing data and the like–you can choose the values for which the other value is missing.

```
semi_join(A, B, by = "sub")
```

```
  sub data1
1   A     1
2   B     2
```

```
anti_join(A, B, by = "sub")
```

```
  sub data1
1   C     3
2   E     4
```

Combining data frames row-wise

The bind_rows acts like rbind, stacking two data frames on top of one another.

```
## This doesn't make any sense, but it works:
bind_rows(left_join(A, B, by = "sub"), right_join(A, B, by = "sub"))
```

```
  sub data1 data2
1   A     1    11
2   B     2    12
3   C     3    NA
4   E     4    NA
5   A     1    11
6   B     2    12
7   D    NA    13
8   F    NA    14
```

summarize/summarise and related pipeline functions

The `summarize` function is a replacement for aggregate, but very flexible and powerful. Because it fits into a maggrittr pipeline, it is also useful for ggplot, to aggregate down from raw data to cell means in one step.

The summarize function is a bit difficult to use at first, because the separation into groups is done prior to piping to summarize, with a grouping function. But when you get accustomed to it, it can be easier to use than aggregate and sometimes simpler because it is easier to create multiple new variables or a single variable with multiple input variables.

We will start with a simple summarize, which does no aggregation.

```
summarize(dat1, mean = mean(as.numeric(dv)), sd = sd(as.numeric(dv)), total = mean(dv +
    newdv))
```

```
      mean       sd total
1 3.166667 1.585923     6
```

The above can use pipes as well, which is more typical:

```
dat1 %>% summarize(mean = mean(as.numeric(dv)), sd = sd(as.numeric(dv)), total = mean(dv +
    newdv))
```

```
      mean       sd total
1 3.166667 1.585923     6
```

Sometimes, we want to calculate a value for each condition/group/etc, and just embed it back in the full data. We can add normal variables into a summarize too:

```
newdat1 <- dat1 %>% summarize(sub = sub, question = question, dv = dv, absdv = sqrt(abs(dv)),
    mean = mean(as.numeric(dv)), sd = sd(as.numeric(dv)), total = mean(dv + newdv))

newdat1[1:10, ]
```

```
   sub question dv    absdv     mean       sd total
1    1        a  5 2.236068 3.166667 1.585923     6
2    1        b  3 1.732051 3.166667 1.585923     6
3    1        c  1 1.000000 3.166667 1.585923     6
4    2        a  2 1.414214 3.166667 1.585923     6
5    2        b  3 1.732051 3.166667 1.585923     6
6    2        c  6 2.449490 3.166667 1.585923     6
7    3        a  4 2.000000 3.166667 1.585923     6
8    3        b  2 1.414214 3.166667 1.585923     6
9    3        c  3 1.732051 3.166667 1.585923     6
10   4        a  1 1.000000 3.166667 1.585923     6
```

This would let us create z-scores easily, but in our case all the means and sd values are the same because we are using the summarized values of the entire data set.

```
newdat1$z <- (newdat1$dv - newdat1$mean)/newdat1$sd
```

Notice we can create two DVs in one command, but it applies it to the entire data set. Also, for all of these functions, the data is automatically fed into the first .data argument, so we don't need to specify it.

Suppose we want to organize by participant code subcode. The group_by function creates a special data structure of tibbles that separates a tibble into separate groups. You might not even be able to see it, but the tibble now contains property that says "Groups: sub [4]".

```
dat1 %>% group_by(sub)
```

```
# A tibble: 12 x 4
```

```
# Groups:   sub [4]
     sub question    dv newdv
   <dbl> <chr>    <dbl> <dbl>
 1     1 a            5     1
 2     1 b            3     3
 3     1 c            1     5
 4     2 a            2     4
 5     2 b            3     3
 6     2 c            6     0
 7     3 a            4     2
 8     3 b            2     4
 9     3 c            3     3
10     4 a            1     5
11     4 b            3     3
12     4 c            5     1
```

Now, we can just compose these together within the pipeline. I will aggregate age by the q3 answer (language)

```
dat1 %>% group_by(sub) %>% summarize(mean = mean(as.numeric(dv)), sd = sd(as.numeric(dv)))
```

```
# A tibble: 4 x 3
    sub  mean    sd
  <dbl> <dbl> <dbl>
1     1  3     2
2     2  3.67  2.08
3     3  3     1
4     4  3     2
```

Calculating z-scores for each participant

```
dat1 %>% group_by(sub) %>% summarize(sub = sub, dv = dv, mean = mean(as.numeric(dv)),
    sd = sd(as.numeric(dv))) %>%
mutate(zdv = (dv - mean)/sd)  ##Compute a z-score
```

```
# A tibble: 12 x 5
# Groups:   sub [4]
     sub    dv  mean    sd    zdv
   <dbl> <dbl> <dbl> <dbl>  <dbl>
 1     1     5  3     2     1
 2     1     3  3     2     0
 3     1     1  3     2    -1
 4     2     2  3.67  2.08 -0.801
 5     2     3  3.67  2.08 -0.320
 6     2     6  3.67  2.08  1.12
 7     3     4  3     1     1
 8     3     2  3     1    -1
 9     3     3  3     1     0
10     4     1  3     2    -1
11     4     3  3     2     0
12     4     5  3     2     1
```

group_by can take multiple variables. Note that if you want to do counts, in aggregate we usually did `length()`, but dplyr provides the more expressive `n()` function, which does not get applied to any particular data value:

```
dat1 %>% group_by(sub) %>% summarize(mean = mean(as.numeric(dv)), N = n())
```

```
# A tibble: 4 x 3
    sub  mean     N
  <dbl> <dbl> <int>
1     1  3         3
2     2  3.67      3
3     3  3         3
4     4  3         3
```

These can become vary powerful when you include filtering and selection before and after a summarize operation, and the pipeline makes this a bit easier to manage the syntax for.

Advanced exercises

suppose every other item was reverse coded

```
dat0$coding <- rep(c(-1, 1), 6)
```

Recode using mutate and filter:

```
d1 <- mutate(filter(dat0, coding == 1), newdv = dv)
d2 <- mutate(filter(dat0, coding == -1), newdv = 6 - dv)
dat0b <- bind_rows(d1, d2)
arrange(dat0b, sub, question)
```

```
   sub question dv coding newdv
1    1        a  5     -1     1
2    1        b  3      1     3
3    1        c  1     -1     5
4    2        a  2      1     2
5    2        b  3     -1     3
6    2        c  6      1     6
7    3        a  4     -1     2
8    3        b  2      1     2
9    3        c  3     -1     3
10   4        a  1      1     1
11   4        b  3     -1     3
12   4        c  5      1     5
```

Big five coding

Load the data set using the big five personality questionnaire.

- The Q1..Q44 are the personality questions. Some are reverse coded, so that the proper coding is 6-X instead of X.
- The questions alternate between 5 factors, but at the end they are a bit off.
- Some of them are reverse coded.

```
big5 <- read.csv("bigfive.csv")
qtype <- c("E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N",
    "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O",
    "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "O", "A", "C", "O")
valence <- c(1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
    1, -1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, -1, 1)
```

Exercise:

Use the above data and dplyr to recode the responses by valence, and then select out each of five personality variables as sums of the proper dimension.

The reshape2 library

The following gives instructions for using the (older) reshape2 library. The tidyr library is its successor, and can also be used (different function names, different arguments) for doing much of the same thing. Some examples of similar reorganization is covered below.

Load the library and a survey for examples:

```
library(reshape2)
dat1 <- read.csv("pooled-survey.csv")
head(dat1)
```

```
  subcode question               timestamp  type  time  answer
1     207        1 Fri Oct 24 14:27:59 2014  inst 88803
2     207        2 Fri Oct 24 14:28:04 2014 short  5172      20
3     207        3 Fri Oct 24 14:28:11 2014 short  6582 english
4     207        4 Fri Oct 24 14:28:29 2014 short 18461      na
5     207        5 Fri Oct 24 14:28:49 2014 multi 19452       1
6     201        1 Mon Oct 20 17:55:59 2014  inst 29450
```

Notice that here, we have five questions of different types in a survey, across a bunch of respondents. This is 'long' format (what Wickham calls 'tidy'). What if we want "wide"? We can use dcast to reorganize into a data frame (d= data frame):

```
dat2 <- dcast(dat1, subcode ~ question, value.var = "answer")
dat2
```

```
   subcode 1  2       3    4 5
1      101   20 english   na 1
2      102   19 english <NA> 1
3      103   20 English <NA> 1
4      104   18 English <NA> 1
5      201   19 english <NA> 1
6      202   19 english   na 1
7      203   19 english   na 1
8      204   20 English <NA> 1
9      206   19 english <NA> 1
10     207   20 english   na 1
11     209   16 english   na 3
12     210   22 english <NA> 1
 [ reached 'max' / getOption("max.print") -- omitted 12 rows ]
```

This is good, but the variable names are a bit inconvenient.

```
colnames(dat2) <- c("subcode", "q1", "q2", "q3", "q4", "q5")
```

or, use `acast` for a vector/matrix. This is not appropriate in this case:

```
dat3 <- acast(dat1, subcode ~ question, value.var = "answer")
dat3[1:5, ]
```

```
    1  2    3         4    5
101 "" "20" "english" "na" "1"
102 "" "19" "english" NA   "1"
```

```
103 "" "20" "English" NA   "1"
104 "" "18" "English" NA   "1"
201 "" "19" "english" NA   "1"
```

What if we want a table of timestamps for each question–maybe to look at how long each one took? Specify this as value.var.

```
dat4 <- dcast(dat1, subcode ~ question, value.var = "timestamp")
dat4[1:10, ]
```

```
   subcode                        1                        2
1      101 Fri Oct 24 11:28:24 2014 Fri Oct 24 11:28:33 2014
2      102 Fri Oct 24 13:03:34 2014 Fri Oct 24 13:03:41 2014
3      103 Fri Nov 07 09:53:40 2014 Fri Nov 07 09:54:06 2014
4      104 Fri Nov 07 12:59:11 2014 Fri Nov 07 12:59:23 2014
5      201 Mon Oct 20 17:55:59 2014 Mon Oct 20 17:56:05 2014
6      202 Thu Oct 23 15:58:06 2014 Thu Oct 23 15:58:13 2014
7      203 Fri Oct 24 09:57:43 2014 Fri Oct 24 09:57:51 2014
8      204 Fri Oct 24 11:36:44 2014 Fri Oct 24 11:37:07 2014
9      206 Fri Oct 24 13:04:24 2014 Fri Oct 24 13:04:28 2014
10     207 Fri Oct 24 14:27:59 2014 Fri Oct 24 14:28:04 2014
                          3                        4                        5
1  Fri Oct 24 11:28:40 2014 Fri Oct 24 11:28:54 2014 Fri Oct 24 11:28:57 2014
2  Fri Oct 24 13:03:45 2014 Fri Oct 24 13:03:54 2014 Fri Oct 24 13:03:57 2014
3  Fri Nov 07 09:54:18 2014 Fri Nov 07 09:54:26 2014 Fri Nov 07 09:54:30 2014
4  Fri Nov 07 12:59:31 2014 Fri Nov 07 12:59:37 2014 Fri Nov 07 12:59:41 2014
5  Mon Oct 20 17:56:12 2014 Mon Oct 20 17:56:19 2014 Mon Oct 20 17:56:22 2014
6  Thu Oct 23 15:58:19 2014 Thu Oct 23 15:58:26 2014 Thu Oct 23 15:58:32 2014
7  Fri Oct 24 09:58:02 2014 Fri Oct 24 09:58:13 2014 Fri Oct 24 09:58:18 2014
8  Fri Oct 24 11:37:11 2014 Fri Oct 24 11:37:17 2014 Fri Oct 24 11:37:22 2014
9  Fri Oct 24 13:04:31 2014 Fri Oct 24 13:04:37 2014 Fri Oct 24 13:04:40 2014
10 Fri Oct 24 14:28:11 2014 Fri Oct 24 14:28:29 2014 Fri Oct 24 14:28:49 2014
```

Now, do the same for time:

```
dat4 <- dcast(dat1, subcode ~ question, value.var = "time")
dat4[1:10, ]
```

```
   subcode     1     2     3     4     5
1      101 32764  9226  6762 13743  3104
2      102 20689  7266  4396  8204  2891
3      103 38236 25939 12205  7573  4403
4      104 45862 12164  7875  5612  4136
5      201 29450  5183  7235  6557  3187
6      202 74307  6757  6266  7033  5502
7      203 34879  7859 11528 10525  5120
8      204 37176 22599  4510  5656  5098
9      206 31742  3629  3055  5933  3415
10     207 88803  5172  6582 18461 19452
```

Using melt to re-form wide data frames

The *cast function take long (tidy) format and make data frames based on a category label. We can do the opposite too, a process referred to as 'melting' (in tidyr, you can use 'gather'). Before, question was used as

the label.

The following doesn't work right. It uses q1..q5 as id variables, because they are non-numeric.

```
melt(dat2)[1:10, ]
```

```
   q1 q2      q3   q4 q5 variable value
1     20 english   na  1  subcode   101
2     19 english <NA>  1  subcode   102
3     20 English <NA>  1  subcode   103
4     18 English <NA>  1  subcode   104
5     19 english <NA>  1  subcode   201
6     19 english   na  1  subcode   202
7     19 english   na  1  subcode   203
8     20 English <NA>  1  subcode   204
9     19 english <NA>  1  subcode   206
10    20 english   na  1  subcode   207
```

Instead, we can specify id.vars, which gets us closer

```
melt(dat2, id.vars = c("subcode"))[1:10, ]
```

```
   subcode variable value
1      101       q1
2      102       q1
3      103       q1
4      104       q1
5      201       q1
6      202       q1
7      203       q1
8      204       q1
9      206       q1
10     207       q1
```

It is a bit puzzling why this works. It uses only subcode as the id variable. Any variable we want to use to tag each row we can move out of the variable set and into the id set, for example, language:

```
melt(dat2, id.vars = c("subcode", "q3"))[1:10, ]
```

```
   subcode      q3 variable value
1      101 english       q1
2      102 english       q1
3      103 English       q1
4      104 English       q1
5      201 english       q1
6      202 english       q1
7      203 english       q1
8      204 English       q1
9      206 english       q1
10     207 english       q1
```

id.vars specify the variables you want to keep and not split on. These appear several times in the new data . Notice that value.name names the value that the matrix is being unfolded to.

we can name the response like this:

```
melt(dat2, id.vars = c("subcode", "q3"), value.name = "response", variable.name = "Question")
```

```
   subcode      q3 Question response
```

```
1      101 english       q1
2      102 english       q1
3      103 English       q1
4      104 English       q1
5      201 english       q1
6      202 english       q1
7      203 english       q1
8      204 English       q1
9      206 english       q1
10     207 english       q1
11     209 english       q1
12     210 english       q1
13     211 English       q1
14     212 English       q1
15     301 english       q1
16     302 English       q1
17     303 English       q1
18     304 english       q1
 [ reached 'max' / getOption("max.print") -- omitted 78 rows ]
```

Notice that q1 was empty, so we can specify just the measure variables we care about:

```
melt(dat2, id.vars = c("subcode", "q3"), measure.vars = c("q2", "q4", "q5"), value.name = "response",
    variable.name = "Question")
```

```
   subcode      q3 Question response
1      101 english       q2       20
2      102 english       q2       19
3      103 English       q2       20
4      104 English       q2       18
5      201 english       q2       19
6      202 english       q2       19
7      203 english       q2       19
8      204 English       q2       20
9      206 english       q2       19
10     207 english       q2       20
11     209 english       q2       16
12     210 english       q2       22
13     211 English       q2       20
14     212 English       q2       20
15     301 english       q2       20
16     302 English       q2       20
17     303 English       q2       19
18     304 english       q2       19
 [ reached 'max' / getOption("max.print") -- omitted 54 rows ]
```

Using `tidyr`

The tidyr library replaces melt and cast with `gather` and `spread`, and more recently replaces `gather` and `spread` with `pivot_wider` and `pivot_longer`.

For gather, you specify the key and value names, and then a selection of columns to 'gather'.

Using gather and pivot_longer

```
library(tidyr)
d1 <- gather(dat2, key = "question", value = "answer", q1, q2, q3, q4, q5)
d2 <- gather(dat2, key = "question", value = "answer", q1:q4)  #only q1 to q5
d3 <- gather(dat2, key = "question", value = "answer", -subcode, -q3)
d1
```

```
   subcode question answer
1      101       q1
2      102       q1
3      103       q1
4      104       q1
5      201       q1
6      202       q1
7      203       q1
8      204       q1
9      206       q1
10     207       q1
11     209       q1
12     210       q1
13     211       q1
14     212       q1
15     301       q1
16     302       q1
17     303       q1
18     304       q1
19     305       q1
20     306       q1
21     307       q1
22     308       q1
23     309       q1
24     310       q1
25     101       q2     20
 [ reached 'max' / getOption("max.print") -- omitted 95 rows ]
```

```
d2
```

```
   subcode q5 question answer
1      101  1       q1
2      102  1       q1
3      103  1       q1
4      104  1       q1
5      201  1       q1
6      202  1       q1
7      203  1       q1
8      204  1       q1
9      206  1       q1
10     207  1       q1
11     209  3       q1
12     210  1       q1
13     211  1       q1
14     212  1       q1
15     301  1       q1
16     302  1       q1
```

```
17     303  1       q1
18     304  1       q1
 [ reached 'max' / getOption("max.print") -- omitted 78 rows ]
```

```
d3
```

```
   subcode      q3 question answer
1      101 english       q1
2      102 english       q1
3      103 English       q1
4      104 English       q1
5      201 english       q1
6      202 english       q1
7      203 english       q1
8      204 English       q1
9      206 english       q1
10     207 english       q1
11     209 english       q1
12     210 english       q1
13     211 English       q1
14     212 English       q1
15     301 english       q1
16     302 English       q1
17     303 English       q1
18     304 english       q1
 [ reached 'max' / getOption("max.print") -- omitted 78 rows ]
```

```
d3 %>% arrange(subcode, question)
```

```
   subcode      q3 question answer
1      101 english       q1
2      101 english       q2     20
3      101 english       q4     na
4      101 english       q5      1
5      102 english       q1
6      102 english       q2     19
7      102 english       q4   <NA>
8      102 english       q5      1
9      103 English       q1
10     103 English       q2     20
11     103 English       q4   <NA>
12     103 English       q5      1
13     104 English       q1
14     104 English       q2     18
15     104 English       q4   <NA>
16     104 English       q5      1
17     201 english       q1
18     201 english       q2     19
 [ reached 'max' / getOption("max.print") -- omitted 78 rows ]
```

Notice that anything excluded from the columns you want to gather is replicated on each row–in these cases subcode, q5, and q3. Thus, it attempts to include all the original data in one form or another. The parameterization of pivot_longer is similar, but slightly different. Most importantly, the variables you want to gather are specified in a `c()` vector, and key becomes names_to and value becomes values_to. Here are the same operations. Notice that the outcome data are organized in a different order, so you might want to pipe this into a an `arrange` function.

```r
d1 <- pivot_longer(dat2, names_to = "question", values_to = "answer", cols = c(q1,
    q2, q3, q4, q5))
d2 <- pivot_longer(dat2, names_to = "question", values_to = "answer", cols = q1:q4)  #only q1 to q5
d3 <- pivot_longer(dat2, names_to = "question", values_to = "answer", cols = c(-subcode,
    -q3))

d1
```

```
# A tibble: 120 x 3
   subcode question answer
     <int> <chr>    <chr>
 1     101 q1       ""
 2     101 q2       "20"
 3     101 q3       "english"
 4     101 q4       "na"
 5     101 q5       "1"
 6     102 q1       ""
 7     102 q2       "19"
 8     102 q3       "english"
 9     102 q4        <NA>
10     102 q5       "1"
# ... with 110 more rows
```

```r
d2
```

```
# A tibble: 96 x 4
   subcode q5    question answer
     <int> <chr> <chr>    <chr>
 1     101 1     q1       ""
 2     101 1     q2       "20"
 3     101 1     q3       "english"
 4     101 1     q4       "na"
 5     102 1     q1       ""
 6     102 1     q2       "19"
 7     102 1     q3       "english"
 8     102 1     q4        <NA>
 9     103 1     q1       ""
10     103 1     q2       "20"
# ... with 86 more rows
```

```r
d3
```

```
# A tibble: 96 x 4
   subcode q3      question answer
     <int> <chr>   <chr>    <chr>
 1     101 english q1       ""
 2     101 english q2       "20"
 3     101 english q4       "na"
 4     101 english q5       "1"
 5     102 english q1       ""
 6     102 english q2       "19"
 7     102 english q4        <NA>
 8     102 english q5       "1"
 9     103 English q1       ""
10     103 English q2       "20"
# ... with 86 more rows
```

```
d3 %>% arrange(subcode, question)
```

```
# A tibble: 96 x 4
   subcode q3      question answer
     <int> <chr>   <chr>    <chr>
 1     101 english q1       ""
 2     101 english q2       "20"
 3     101 english q4       "na"
 4     101 english q5       "1"
 5     102 english q1       ""
 6     102 english q2       "19"
 7     102 english q4        <NA>
 8     102 english q5       "1"
 9     103 English q1       ""
10     103 English q2       "20"
# ... with 86 more rows
```

Using spread

Spread reverses the gathering.

```
d1.wide <- d1 %>% spread(question, answer)
d1.wide[1:10, ]
```

```
# A tibble: 10 x 6
   subcode q1    q2    q3      q4    q5
     <int> <chr> <chr> <chr>   <chr> <chr>
 1     101 ""    20    english na    1
 2     102 ""    19    english <NA>  1
 3     103 ""    20    English <NA>  1
 4     104 ""    18    English <NA>  1
 5     201 ""    19    english <NA>  1
 6     202 ""    19    english na    1
 7     203 ""    19    english na    1
 8     204 ""    20    English <NA>  1
 9     206 ""    19    english <NA>  1
10     207 ""    20    english na    1
```

Similarly, `spread` has been replaced with `pivot_wider`

```
d1.wide2 <- d1 %>% pivot_wider(names_from = question, values_from = answer)
d1.wide2[1:10, ]
```

```
# A tibble: 10 x 6
   subcode q1    q2    q3      q4    q5
     <int> <chr> <chr> <chr>   <chr> <chr>
 1     101 ""    20    english na    1
 2     102 ""    19    english <NA>  1
 3     103 ""    20    English <NA>  1
 4     104 ""    18    English <NA>  1
 5     201 ""    19    english <NA>  1
 6     202 ""    19    english na    1
 7     203 ""    19    english na    1
 8     204 ""    20    English <NA>  1
 9     206 ""    19    english <NA>  1
10     207 ""    20    english na    1
```

Each of these functions have a number of additional arguments, including sorting variables, ways of creating new variable names, and how to deal with missing values.

Exercises

- Using the `big5` data set, add a unique subject code to each row. Then, use "melt" to create a data frame that has the following columns: subject code, gender, question and answer.

```
big5 <- read.csv("bigfive.csv")
qtype <- c("E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N",
    "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O",
    "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "O", "A", "C", "O")
valence <- c(1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
    1, -1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, -1, 1)
varnames <- colnames(big5)[2:45]

## first, recode the negative codings.
answers <- select(big5, contains("Q"))

## mutate the columns with -1 valence:
recoded <- answers %>% mutate_if(valence == -1, function(x) {
    6 - x
})

melted <- melt(mutate(recoded, sub = 1:nrow(recoded)), id.vars = c("sub"))

arrange(melted, sub, variable)
```

```
   sub variable value
1    1       Q1     3
2    1       Q2     2
3    1       Q3     4
4    1       Q4     2
5    1       Q5     3
6    1       Q6     2
7    1       Q7     5
8    1       Q8     2
9    1       Q9     1
10   1      Q10     5
11   1      Q11     3
12   1      Q12     4
13   1      Q13     2
14   1      Q14     4
15   1      Q15     4
16   1      Q16     2
17   1      Q17     5
18   1      Q18     2
19   1      Q19     1
20   1      Q20     4
21   1      Q21     4
22   1      Q22     5
23   1      Q23     3
24   1      Q24     1
25   1      Q25     4
```

```
 [ reached 'max' / getOption("max.print") -- omitted 5563 rows ]
```

Solution to Exercise 1.

```
big5 <- read.csv("bigfive.csv")
qtype <- c("E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N",
    "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "E", "A", "C", "N", "O",
    "E", "A", "C", "N", "O", "E", "A", "C", "N", "O", "O", "A", "C", "O")
valence <- c(1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, -1,
    1, -1, -1, 1, 1, -1, 1, 1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, -1, 1)
varnames <- colnames(big5)[2:45]

## first, recode the negative codings.
answers <- select(big5, contains("Q"))

## mutate the columns with -1 valence:
recoded <- answers %>% mutate_if(valence == -1, function(x) {
    6 - x
})


## check this. For negative valence, 2 becomes 4 etc.
bind_rows(recoded[1, ], answers[1, ])
```

```
  Q1 Q2 Q3 Q4 Q5 Q6 Q7 Q8 Q9 Q10 Q11 Q12 Q13 Q14 Q15 Q16 Q17 Q18 Q19 Q20 Q21
1  3  2  4  2  3  2  5  2  1   5   3   4   2   4   4   2   5   2   1   4   4
  Q22 Q23 Q24 Q25 Q26 Q27 Q28 Q29 Q30 Q31 Q32 Q33 Q34 Q35 Q36 Q37 Q38 Q39 Q40
1   5   3   1   4   4   2   3   4   2   1   3   5   2   1   3   3   2   3   1
  Q41 Q42 Q43 Q44
1   2   2   2   4
 [ reached 'max' / getOption("max.print") -- omitted 1 rows ]
```

```
## create composite subsets
b5.e <- select(recoded, one_of(varnames[qtype == "E"]))
b5.a <- select(recoded, one_of(varnames[qtype == "A"]))
b5.c <- select(recoded, one_of(varnames[qtype == "C"]))
b5.n <- select(recoded, one_of(varnames[qtype == "N"]))
b5.o <- select(recoded, one_of(varnames[qtype == "O"]))




composites1 <- data.frame(e = rowMeans(b5.e, na.rm = T), a = rowMeans(b5.a, na.rm = T),
    c = rowMeans(b5.c, na.rm = T), n = rowMeans(b5.n, na.rm = T), o = rowMeans(b5.o,
        na.rm = T))
```