Data management with dplyr, tidyr, and reshape2

Shane T. Mueller shanem@mtu.edu

2025-01-13

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 integrate 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. However, R now includes the slightly-different pipe operator "'|>'' as a first-class part of the language. It mostly works the same as the magrittr version, but there are some differences in how it works with functions that have multiple arguments, so you may need to use the magrittr version in some cases.

Forward-piping

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, while |> is the built-in equivalent. These operators are supported widely by a number of libraries–all of the tidyverse libraries in fact. We will tend to use the |> version, but older versions of R or special applications may require magrittr. 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
```

```
## Without magrittr--requires ()
(-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:

```
## magrittr lets you specify the first argument of a function with a dot
rnorm(10) %>%
    round(., 3)
```

```
 [1]  1.336  0.206 -0.903 -0.600  1.002 -1.743 -1.456  1.445  0.198 -0.426
```

```
# rnorm(10) |> round(.,3) ##This won't work


## Piping automatically replaces the first argument and any other arguments are
## bound to later slots
set.seed(100)
rnorm(10) |>
    round(3)
```

```
 [1] -0.502  0.132 -0.079  0.887  0.117  0.319 -0.582  0.715 -0.825 -0.360
```

```
## standard |> works the same here:
set.seed(100)
rnorm(10) |>
    round(3)
```

```
 [1] -0.502  0.132 -0.079  0.887  0.117  0.319 -0.582  0.715 -0.825 -0.360
```

```
## put the argument into the second slot:
sample(1:5, replace = T, size = 10) %>%
    round(runif(10), .)
```

```
 [1] 0.54000 0.71100 0.53830 0.74900 0.42010 0.17142 0.77000 0.90000 0.54900
[10] 0.27770
```

Finally, R can have its assignment arrow go in either direction, like below, which makes a pipe stream make more sense:

```
runif(100) %>%
    sd %>%
    sqrt %>%
    log -> value
value <- runif(100) %>%
    sd %>%
    sqrt %>%
    log  #Notice no parens are needed when using magrittr pipes.
value
```

```
[1] -0.6216435
```

Exercise 1

Rewrite the following code using piping:

```
set.seed(100)
dat.A <- sample(1:10, 100, replace = T)
dat.B <- table(dat.A)
sd <- sd(dat.B)
```

These are fairly simple examples but not that much of an improvement over just using functions. However, it can transform how you use the libraries in tidyverse for data management, because it allows you to create data processing 'pipelines'. We will see this in the next section.

Overview of dplyr

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

```
dat0 <- as_tibble(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 <- tibble(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")  ##this just returns the data for use, but does not save
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 b     B        10  2.90
```

```
data2 <- filter(data, sub == "b")  ## re-assign to data

head(data)
```

```
# A tibble: 6 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
2 p     B         1  4.40
3 z     B         4  4.56
4 f     A         9  1.95
5 x     B         2  4.33
6 k     B         8  2.57
```

```
dim(data)
```

```
[1] 100   4
```

```
dim(data2)
```

```
[1] 1 4
```

Notice how data2 is only 3 rows long. 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. You can use magrittr pipes here alternately:

```
data %>%
    filter(sub == "b")
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 b     B        10  2.90
```

```
data |>
    filter(sub == "b") -> data3  ##use a pipe, then assign to data at the end

data3
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 b     B        10  2.90
```

In this version, data doesn't get overwritten and you can use this expression directly within another function, or pipe it to later processing steps or graphing. Note that the <- assignment symbol can be reversed as -> and assigned to a variable name at the end of the pipeline.

dplyr operations on rows

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] FALSE FALSE FALSE FALSE FALSE FALSE
```

```
filter(dat, sub == "b")  ##use filter to pick out just the sub == 'b' rows
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 b     B        10  2.90
```

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

```
dat |>
    slice(1)  ##first row
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
```

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

```
# A tibble: 9 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 p     B         1  4.40
2 z     B         4  4.56
3 f     A         9  1.95
4 x     B         2  4.33
5 k     B         8  2.57
6 s     C         5  3.71
7 d     C         3  4.80
8 q     A         5  2.85
9 j     C         3  1.56
```

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

```
# A tibble: 20 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 p     B         1 4.40
 2 f     A         9 1.95
 3 k     B         8 2.57
 4 d     C         3 4.80
 5 j     C         3 1.56
 6 r     B         2 2.25
 7 m     C         7 2.37
 8 e     B         5 0.536
 9 d     C         2 2.11
10 i     C        10 3.08
```

```
11 n     C         1 3.41
12 u     C         5 4.46
13 i     C         5 4.69
14 p     C         4 4.86
15 h     B         4 1.19
16 e     C         6 3.74
17 r     A         9 4.66
18 g     A         6 1.56
19 q     A         8 1.04
20 b     B        10 2.90
```

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

```
# A tibble: 99 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 p     B         1  4.40
 2 z     B         4  4.56
 3 f     A         9  1.95
 4 x     B         2  4.33
 5 k     B         8  2.57
 6 s     C         5  3.71
 7 d     C         3  4.80
 8 q     A         5  2.85
 9 j     C         3  1.56
10 n     B        10  3.72
# i 89 more rows
```

There are several versions of slice for particular common tasks. These include:

- `slice_head` and `slice_tail` for the first or last n rows
- `slice_min` and `slice_max` for the rows with the minimum or maximum value of a variable
- `slice_sample` for a random sample of rows

```
dat |>
    slice_head()  ## show the first rows
```

```
# A tibble: 1 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
```

```
dat |>
    slice_tail(n = 3)  ## show the last 3 rows
```

```
# A tibble: 3 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 g     B         9 1.83
2 j     A         5 0.395
3 e     C         8 1.03
```

```
dat |>
    slice_min(dv1, n = 2)  ## show the two rows with the smallest dv1
```

```
# A tibble: 2 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
```

```
1 j     A         7 0.148
2 d     B         5 0.159
```

```
dat |>
    slice_max(dv1, n = 3)  ## show the two rows with the largest dv1
```

```
# A tibble: 3 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 p     A         1  4.97
2 p     C         4  4.86
3 w     C         8  4.82
```

```
dat |>
    slice_sample(n = 5)  ## show 5 random rows
```

```
# A tibble: 5 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 e     B         5 0.536
2 r     A         6 3.87
3 g     B         8 3.29
4 n     A         6 3.23
5 s     C         5 3.71
```

---

arrange()

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

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

```
# A tibble: 100 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 a     C         5 0.671
 2 a     B         4 3.96
 3 a     C        10 1.29
 4 b     B        10 2.90
 5 d     C         3 4.80
 6 d     C         2 2.11
 7 d     B         9 4.00
 8 d     B         5 0.159
 9 d     C        10 2.76
10 e     B         5 0.536
# i 90 more rows
```

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

```
# A tibble: 100 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 a     B         4 3.96
 2 a     C         5 0.671
 3 a     C        10 1.29
 4 b     B        10 2.90
 5 d     C         2 2.11
```

```
 6 d     C         3 4.80
 7 d     B         5 0.159
 8 d     B         9 4.00
 9 d     C        10 2.76
10 e     A         1 4.66
# i 90 more rows
```

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

```
# A tibble: 84 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 a     B         4  3.96
 2 a     C        10  1.29
 3 b     B        10  2.90
 4 d     B         9  4.00
 5 d     C         3  4.80
 6 d     C         2  2.11
 7 d     C        10  2.76
 8 e     A         1  4.66
 9 e     A         8  2.48
10 e     B         6  4.44
# i 74 more rows
```

---

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   2 200
2  10 300
3   6 100
4   8 100
5   3 200
6   9 300
7   4 200
8   3 100
9   5 100
10  8 200
11 10 200
12  9 100
13  1 100
14  9 200
15  2 100
16  6 200
```

You can also specify specific variables you wish to use:

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

```
# A tibble: 25 x 1
   sub
   <chr>
 1 f
 2 p
 3 z
 4 x
 5 k
 6 s
 7 d
 8 q
 9 j
10 n
# i 15 more rows
```

Retain all columns of distinct data:

here, we use the strange-looking hidden argument .keep_all=T to retain all the columns

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

```
# A tibble: 25 x 1
   sub
   <chr>
 1 f
 2 p
 3 z
 4 x
 5 k
 6 s
 7 d
 8 q
 9 j
10 n
# i 15 more rows
```

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

```
# A tibble: 25 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 f     C         3  3.32
 2 p     B         1  4.40
 3 z     B         4  4.56
 4 x     B         2  4.33
 5 k     B         8  2.57
 6 s     C         5  3.71
 7 d     C         3  4.80
 8 q     A         5  2.85
 9 j     C         3  1.56
10 n     B        10  3.72
# i 15 more rows
```

```
## distinct pairs of columns:
dat |>
```

```
    distinct(cond, sub)
```

```
# A tibble: 53 x 2
   cond  sub
   <chr> <chr>
 1 C     f
 2 B     p
 3 B     z
 4 A     f
 5 B     x
 6 B     k
 7 C     s
 8 C     d
 9 A     q
10 C     j
# i 43 more rows
```

dplyr operations on columns

select()

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

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

```
# A tibble: 12 x 2
     sub    dv
   <dbl> <dbl>
 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)
```

```
# A tibble: 12 x 3
     sub question    dv
   <dbl> <chr>    <dbl>
 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)
```

```
# A tibble: 12 x 2
     sub    dv
   <dbl> <dbl>
 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
```

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

```
# A tibble: 3 x 1
     dv
  <dbl>
1     1
2     3
3     5
```

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

```
# select columns that start with s
dat0 |>
    select(starts_with("s"))
```

```
# A tibble: 12 x 1
     sub
   <dbl>
 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) |>
    head()
```

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

```
dat0 |>
    rename(participant2 = sub) |>
    head()
```

```
# A tibble: 6 x 3
  participant2 question    dv
         <dbl> <chr>    <dbl>
1            1 a            5
2            1 b            3
3            1 c            1
4            2 a            2
5            2 b            3
6            2 c            6
```

The related `rename_with` function allows you to rename columns based on a function. Here, we add a prefix each variable with a 'V' and transform to capital letters:

```
dat0 |>
    rename_with(function(x) {
        paste("V", toupper(x), sep = "")
    }) |>
    slice_head()
```

```
# A tibble: 1 x 3
   VSUB VQUESTION   VDV
  <dbl> <chr>     <dbl>
1     1 a             5
```

other dplyr column functions

Less used but handy functions include:

- `glimpse()`: gives a quick overview of the data frame, showing the first few rows and the data types of each column.
- `pull()`: extracts a single column as a vector.
- `relocate()`: changes the order of columns

```
dat0 |>
    glimpse()
```

```
Rows: 12
Columns: 3
$ sub      <dbl> 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4
$ question <chr> "a", "b", "c", "a", "b", "c", "a", "b", "c", "a", "b", "c"
$ dv       <dbl> 5, 3, 1, 2, 3, 6, 4, 2, 3, 1, 3, 5
```

```
dat0 |>
    pull(sub)
```

```
 [1] 1 1 1 2 2 2 3 3 3 4 4 4
```

```
## This is basically the same as select:
dat0 |>
    relocate(question, sub, dv) |>
    slice_head()
```

```
# A tibble: 1 x 3
  question   sub    dv
  <chr>    <dbl> <dbl>
1 a            1     5
```

```
## Without using all the variable names, it is different than select:
dat0 |>
    relocate(question, sub) |>
    slice_head()
```

```
# A tibble: 1 x 3
  question   sub    dv
  <chr>    <dbl> <dbl>
1 a            1     5
```

```
dat0 |>
    select(question, sub) |>
    slice_head()
```

```
# A tibble: 1 x 2
  question   sub
  <chr>    <dbl>
1 a            1
```

Exercise 2:

rewrite the following old-style examples with dplyr functions and pipes where appropriate.

```
dat[dat$cond == "C", ]
```

```
# A tibble: 40 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 f     C         3  3.32
 2 s     C         5  3.71
 3 d     C         3  4.80
 4 j     C         3  1.56
 5 m     C         7  2.37
 6 j     C         3  3.21
 7 d     C         2  2.11
 8 i     C        10  3.08
 9 n     C         1  3.41
```

```
10 u     C         5  4.46
# i 30 more rows
```

```
dat[dat$dv1 > 2.5, ]
```

```
# A tibble: 51 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 f     C         3  3.32
 2 p     B         1  4.40
 3 z     B         4  4.56
 4 x     B         2  4.33
 5 k     B         8  2.57
 6 s     C         5  3.71
 7 d     C         3  4.80
 8 q     A         5  2.85
 9 n     B        10  3.72
10 l     B         1  3.70
# i 41 more rows
```

```
dat[1:10, ]
```

```
# A tibble: 10 x 4
   sub   cond  group   dv1
   <chr> <chr> <int> <dbl>
 1 f     C         3  3.32
 2 p     B         1  4.40
 3 z     B         4  4.56
 4 f     A         9  1.95
 5 x     B         2  4.33
 6 k     B         8  2.57
 7 s     C         5  3.71
 8 d     C         3  4.80
 9 q     A         5  2.85
10 j     C         3  1.56
```

```
dat[, 2:3]
```

```
# A tibble: 100 x 2
   cond  group
   <chr> <int>
 1 C         3
 2 B         1
 3 B         4
 4 A         9
 5 B         2
 6 B         8
 7 C         5
 8 C         3
 9 A         5
10 C         3
# i 90 more rows
```

```
dat[, c(3, 1, 2)]
```

```
# A tibble: 100 x 3
   group sub   cond
```

```
   <int> <chr> <chr>
 1     3 f     C
 2     1 p     B
 3     4 z     B
 4     9 f     A
 5     2 x     B
 6     8 k     B
 7     5 s     C
 8     3 d     C
 9     5 q     A
10     3 j     C
# i 90 more rows
```

```
datx <- dat
colnames(datx) <- c("participant", "condition", "counterbalance", "score")


dat0[dat0$question == "a", ]
```

```
# A tibble: 4 x 3
    sub question    dv
  <dbl> <chr>    <dbl>
1     1 a            5
2     2 a            2
3     3 a            4
4     4 a            1
```

```
dat0[order(dat0$question), ]
```

```
# A tibble: 12 x 3
     sub question    dv
   <dbl> <chr>    <dbl>
 1     1 a            5
 2     2 a            2
 3     3 a            4
 4     4 a            1
 5     1 b            3
 6     2 b            3
 7     3 b            2
 8     4 b            3
 9     1 c            1
10     2 c            6
11     3 c            3
12     4 c            5
```

dplyr Recoding functions

mutate() and transmute()

The `mutate` function is a workhorse that you use frequently to do calculation and recoding in a dplyr pipeline. It 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)
```

```
## alternative using pipes:
dat0 |>
    mutate(newdv = 6 - dv) -> dat1  ##rewrites new data set to dat1

dat0 |>
    mutate(dv3 = dv * 2)  #does not add to dat0
```

```
# A tibble: 12 x 4
     sub question    dv   dv3
   <dbl> <chr>    <dbl> <dbl>
 1     1 a            5    10
 2     1 b            3     6
 3     1 c            1     2
 4     2 a            2     4
 5     2 b            3     6
 6     2 c            6    12
 7     3 a            4     8
 8     3 b            2     4
 9     3 c            3     6
10     4 a            1     2
11     4 b            3     6
12     4 c            5    10
```

More complex mutations are possible:

```
dat1$newdv2 = dat1$dv * dat1$newdv
mutate(dat1, newdv3 = dv * newdv)
```

```
# A tibble: 12 x 6
     sub question    dv newdv newdv2 newdv3
   <dbl> <chr>    <dbl> <dbl>  <dbl>  <dbl>
 1     1 a            5     1      5      5
 2     1 b            3     3      9      9
 3     1 c            1     5      5      5
 4     2 a            2     4      8      8
 5     2 b            3     3      9      9
 6     2 c            6     0      0      0
 7     3 a            4     2      8      8
 8     3 b            2     4      8      8
 9     3 c            3     3      9      9
10     4 a            1     5      5      5
11     4 b            3     3      9      9
12     4 c            5     1      5      5
```

```
dat1[1:5, ]
```

```
# A tibble: 5 x 5
    sub question    dv newdv newdv2
  <dbl> <chr>    <dbl> <dbl>  <dbl>
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
```

Notice that like all the tidyverse functions, there is no side effect on the input–newdv3 doesn't get added do

dat1 unless you do it explicitly.

Transmute returns only the new variable, which can be handy if you want to do a set of separate analysis and add them to an existing data set.

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

```
# A tibble: 12 x 1
   newdv2
    <dbl>
 1      5
 2      9
 3      5
 4      8
 5      9
 6      0
 7      8
 8      8
 9      9
10      5
11      9
12      5
```

```
dat1$newdv3 <- dat1 |>
    transmute(newdv2 = dv * newdv)
```

Using mutate to recode categorical values

Generally, if you want to recode the levels of a variable, you would use a mutate and reassign the new variable name to the original using code within the mutate to do the recoding. There are a few options, including `ifelse()`, `recode()`, `which`, `case_when`, and some others. Let's say we want to capitalize question variable. Here are a few approaches:

Recode using `toupper`

Because we are just capitalizing, we can just use `toupper`. This is not a general approach, but it works here:

```
dat1 |>
    mutate(questionUC = toupper(question)) |>
    head()
```

```
# A tibble: 6 x 7
    sub question    dv newdv newdv2 newdv3$newdv2 questionUC
  <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl> <chr>
1     1 a            5     1      5             5 A
2     1 b            3     3      9             9 B
3     1 c            1     5      5             5 C
4     2 a            2     4      8             8 A
5     2 b            3     3      9             9 B
6     2 c            6     0      0             0 C
```

Recode using `ifelse`

The ifelse function is useful for recoding into a binary set, but we can nest two to recode our three levels:

```
dat1 |>
    mutate(question = ifelse(question == "a", "A", ifelse(question == "b", "B", "C"))) |>
    head()
```

```
# A tibble: 6 x 6
    sub question    dv newdv newdv2 newdv3$newdv2
  <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
1     1 A            5     1      5             5
2     1 B            3     3      9             9
3     1 C            1     5      5             5
4     2 A            2     4      8             8
5     2 B            3     3      9             9
6     2 C            6     0      0             0
```

Recode using `case_when`

Within mutate, `case_when` can be used to recode based on a testing a boolean on the left of ~ and providing the recoded value on the right. The tests `question=='a'` can be anything, so you can do arbitrary tests of multiple different expressions there. This is fine, but you have to write a test for each case, so it is a bit complicated if you are just doing recoding.

```
dat1 |>
    mutate(question = case_when(question == "a" ~ "A", question == "b" ~ "B", question ==
        "c" ~ "C"))
```

```
# A tibble: 12 x 6
     sub question    dv newdv newdv2 newdv3$newdv2
   <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
 1     1 A            5     1      5             5
 2     1 B            3     3      9             9
 3     1 C            1     5      5             5
 4     2 A            2     4      8             8
 5     2 B            3     3      9             9
 6     2 C            6     0      0             0
 7     3 A            4     2      8             8
 8     3 B            2     4      8             8
 9     3 C            3     3      9             9
10     4 A            1     5      5             5
11     4 B            3     3      9             9
12     4 C            5     1      5             5
```

Recode using `switch`

Using `switch` can means avoiding writing multiple tests. However, you need to preface it with 'rowwise' for it to work:

```
dat1 |>
    rowwise() |>
    mutate(question = switch((question), a = "A", b = "B", c = "C"))
```

```
# A tibble: 12 x 6
# Rowwise:
     sub question    dv newdv newdv2 newdv3$newdv2
   <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
 1     1 A            5     1      5             5
 2     1 B            3     3      9             9
 3     1 C            1     5      5             5
 4     2 A            2     4      8             8
 5     2 B            3     3      9             9
 6     2 C            6     0      0             0
```

```
 7     3 A            4     2      8             8
 8     3 B            2     4      8             8
 9     3 C            3     3      9             9
10     4 A            1     5      5             5
11     4 B            3     3      9             9
12     4 C            5     1      5             5
```

An advantage of switch is that you could put a complex expression within the first argument of switch, giving you a lot of flexibility. By default, the N+1th argument of switch will be selected based on what the first argument computes to, but you can also use it like above to pick out particular mappings. Here, we could recode dv to be 'odd'/'even'. The argument `dv %% 2+1` turns into either 1 or 2, and that picks out the 'even' and 'odd' labels.

```
dat1 |>
    rowwise() |>
    mutate(odd = switch((dv%%2 + 1), "even", "odd")) |>
    select(sub, question, dv, odd)
```

```
# A tibble: 12 x 4
# Rowwise:
     sub question    dv odd
   <dbl> <chr>    <dbl> <chr>
 1     1 a            5 odd
 2     1 b            3 odd
 3     1 c            1 odd
 4     2 a            2 even
 5     2 b            3 odd
 6     2 c            6 even
 7     3 a            4 even
 8     3 b            2 even
 9     3 c            3 odd
10     4 a            1 odd
11     4 b            3 odd
12     4 c            5 odd
```

Recode using `recode()`

Probably the most natural approach is to use the `recode` function within dplyr. Be careful though—the `car` library also has a `recode` function but it won't work the same way, and if you have loaded `car` this might fail unless you specify the dplyr one specifically. Here, the new value goes on the right and the old value goes on the left. The kind of quotes you use don't really matter:

```
dat1 |>
    mutate(question = dplyr::recode(question, a = "A", b = "B", c = "C"))
```

```
# A tibble: 12 x 6
     sub question    dv newdv newdv2 newdv3$newdv2
   <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
 1     1 A            5     1      5             5
 2     1 B            3     3      9             9
 3     1 C            1     5      5             5
 4     2 A            2     4      8             8
 5     2 B            3     3      9             9
 6     2 C            6     0      0             0
 7     3 A            4     2      8             8
 8     3 B            2     4      8             8
```

```
 9     3 C            3     3      9             9
10     4 A            1     5      5             5
11     4 B            3     3      9             9
12     4 C            5     1      5             5
```

Exercise 2

For dat1, in a single pipeline: * select just sub, question, and newdv2 * remove subject 2 * recode question to be "Arsenic" if it is "a", "Borox" if it is "b", and "Carbon" if it is "c" * rename the variables to be "participant", "element", and "happiness" * compute a new variable dvnormed that is the new dv variable minus 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
```

dplyr functions to aggregation and and calculation:

summarize or summarise

The `summarize` function is a replacement for aggregate, but very flexible and powerful. Because it fits into a 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. Without a group_by function, using summarize is basically like mutate, but it does not keep the variables we don't specify.

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

```
# A tibble: 1 x 3
   mean    sd total
  <dbl> <dbl> <dbl>
1  3.17  1.59     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))
```

```
# A tibble: 1 x 3
   mean    sd total
  <dbl> <dbl> <dbl>
1  3.17  1.59     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:

```
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

newdat1[1:10, ]
```

```
# A tibble: 10 x 7
     sub question    dv absdv  mean    sd total
   <dbl> <chr>    <dbl> <dbl> <dbl> <dbl> <dbl>
 1     1 a            5  2.24  3.17  1.59     6
 2     1 b            3  1.73  3.17  1.59     6
 3     1 c            1  1     3.17  1.59     6
 4     2 a            2  1.41  3.17  1.59     6
 5     2 b            3  1.73  3.17  1.59     6
 6     2 c            6  2.45  3.17  1.59     6
 7     3 a            4  2     3.17  1.59     6
 8     3 b            2  1.41  3.17  1.59     6
 9     3 c            3  1.73  3.17  1.59     6
10     4 a            1  1     3.17  1.59     6
```

Notice that this no longer works like mutate, because the functions get applied to the whole data column–not to each observation. This is probably a bit inefficient, because it is calculating the mean and sd for every row of the data. But this lets 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, which is simpler than adding a bunch of variables one at a time like we would have to for the z value above.

Summarize by groups–a modern analog to aggregate.

Suppose we want to organize by participant code subcode and calculate values on each group. This is the same thing we use aggregate for, and the same thing pivot tables do in spreadsheets. The group_by function creates a special data structure of tibbles that separates a tibble into separate groups behind the scenes. You might not even be able to see it, but the tibble now contains property that says "Groups: sub [4]".

```
dat1
```

```
# A tibble: 12 x 6
     sub question    dv newdv newdv2 newdv3$newdv2
   <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
 1     1 a            5     1      5             5
 2     1 b            3     3      9             9
 3     1 c            1     5      5             5
 4     2 a            2     4      8             8
```

```
 5     2 b            3     3      9             9
 6     2 c            6     0      0             0
 7     3 a            4     2      8             8
 8     3 b            2     4      8             8
 9     3 c            3     3      9             9
10     4 a            1     5      5             5
11     4 b            3     3      9             9
12     4 c            5     1      5             5
```

```
dat1 |>
    group_by(sub)
```

```
# A tibble: 12 x 6
# Groups:   sub [4]
     sub question    dv newdv newdv2 newdv3$newdv2
   <dbl> <chr>    <dbl> <dbl>  <dbl>         <dbl>
 1     1 a            5     1      5             5
 2     1 b            3     3      9             9
 3     1 c            1     5      5             5
 4     2 a            2     4      8             8
 5     2 b            3     3      9             9
 6     2 c            6     0      0             0
 7     3 a            4     2      8             8
 8     3 b            2     4      8             8
 9     3 c            3     3      9             9
10     4 a            1     5      5             5
11     4 b            3     3      9             9
12     4 c            5     1      5             5
```

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

We can use group_by + summarize to calculate means and standard deviations in each group. If we want z-scores for each group, we need to pipe this to mutate to make a function on this new data frame:

```
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.

Aggregating with a function returning multiple values with `reframe`

In some cases (like the z-score calculation), the `summarize` function returns a warning that its use is depracated when returning more than 1 row per group, and suggests using reframe() instead. In many cases, this warning can be ignored. But if we have a number of related new statistics calculated on the same grouping, we can use reframe to add all the variables (which summarize cannot do.)

```
dostats <- function(x) {
    mu <- mean(x)
    sd <- sd(x)
    z <- (mu)/sd
    n <- length(x)
    se <- sd/sqrt(n)

    data.frame(mu, sd, z, n, se)
}
dat1 |>
    group_by(sub) |>
    reframe(dostats(dv))
```

```
# A tibble: 4 x 6
    sub    mu    sd     z     n    se
  <dbl> <dbl> <dbl> <dbl> <int> <dbl>
1     1  3     2     1.5       3 1.15
2     2  3.67  2.08  1.76      3 1.20
```

```
3     3  3     1     3         3 0.577
4     4  3     2     1.5       3 1.15
```

Other dplyr capabilities

This unit is not intended to completely cover all the capabilities of dplyr, but rather the ones you are likely to use frequently and will benefit from knowing off the top of your head. There are many additional functions and helper functions that support more complex data management, and that work seamlessly with the functions you are likely to use most often like mutate, select, filter, and summarize. There are a number of functions that support connecting to databases, ways of dealing with multiple columns at once when recoding or summarizing.

Advanced exercises

Suppose every other item was reverse coded. We can specify a column that identifies this coding:

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

Now, to recode, we can use using mutate and filter. Note that for a 5-point scale, reverse coding involves just subtracting the value from 6. The simplest way to do this is to divide the data into two groups using filter, create a new variable separately in each one, and then re-join using bind_rows. Finally, I can re-order them using arrange.

```
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)
```

```
# A tibble: 12 x 5
     sub question    dv coding newdv
   <dbl> <chr>    <dbl>  <dbl> <dbl>
 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
```

You could recode using a single mutate command along with ifelse:

```
dat0 |>
    mutate(newdv = ifelse(coding == 1, dv, 6 - dv))
```

```
# A tibble: 12 x 5
     sub question    dv coding newdv
   <dbl> <chr>    <dbl>  <dbl> <dbl>
 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 3:

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.

Reshaping data

One common task in data programming is to change the shape of your data table. The two common forms are called 'wide' and 'long'. In wide format, each row is a single observation, and each column is a variable. In long format, each row is a single observation, and each column is a variable. The long format is often called 'tidy' data, and each DV is tagged (often redundantly) with each piece of information we have about it. For example, consider the following data:

```
sub stage question answer
1    pre     "age"    23
1    pre     "color" "blue"
1    pre     "height"  5'10"
1    test1    Q1       3
1    test1    Q2       1
1    test1    Q3       5
1    test1    Q4       3
1    post     P1       "yes"
1    post     P2       "no"
2    pre     "age"    21
2    pre     "color" "orange"
2    pre     "height"  5'7"
2    test1    Q1       4
2    test1    Q2       2
2    test1    Q3       5
2    test1    Q4       2
```

```
2    post     P1       "no"
2    post     P2       "no"
```

Here, we have asked each participant nine questions. In wide format, it would look like this:

```
sub age color height Q1 Q2 Q3 Q4 P1 P2
1    23  blue  5'10"  3  1  5  3 yes no
2    21  orange 5'7"  4  2  5  2 no  no
```

It is a bit ironic, but the long format is sometimes called 'tidy' while the second format is not (maybe 'messy'). These contain the same information, but many libraries (especially those in the tidyverse) expect long data format, and so it is useful to be able to convert between the two. Without software to do this, you will be painstakingly cutting, pasting, and sorting in excel, which is time-consuming and error-prone.

There have been several libraries that help do this. 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.

The `reshape2` library

Load the library and a survey for examples:

```
library(reshape2)
dat2 <- read.csv("pooled-survey.csv")
head(dat2)
```

```
  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):

```
dat3 <- dcast(dat2, subcode ~ question, value.var = "answer")
head(dat3)
```

```
  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
```

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

```
colnames(dat3) <- c("subcode", "q1", "q2", "q3", "q4", "q5")
dat3$q1 <- "Missing"  ##all the data are empty so we will fill it in
```

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

```
acast(dat2, subcode ~ question, value.var = "answer") |>
    head()
```

```
    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"
202 "" "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.

```
dcast(dat2, subcode ~ question, value.var = "timestamp") |>
    head()
```

```
  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
                           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
```

Now, do the same for time:

```
dcast(dat2, subcode ~ question, value.var = "time") |>
    head()
```

```
  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
```

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.

So let's try to use `melt` to create a long data frame that looks like dat1. Try just using melt():

```
melt(dat3) |>
    head()
```

```
       q1 q2      q3   q4 q5 variable value
1 Missing 20 english   na  1  subcode   101
2 Missing 19 english <NA>  1  subcode   102
3 Missing 20 English <NA>  1  subcode   103
4 Missing 18 English <NA>  1  subcode   104
```

```
5 Missing 19 english <NA>  1  subcode   201
6 Missing 19 english   na  1  subcode   202
```

It didn't quite work right. It uses q1..q5 as id variables, because they are non-numeric. We'd like q1..q5 to appear as an entry in a single column, and subcode should be the id variable. We can specify id.vars directly, which actually works:

```
melt(dat3, id.vars = c("subcode")) |>
    head()
```

```
  subcode variable   value
1     101       q1 Missing
2     102       q1 Missing
3     103       q1 Missing
4     104       q1 Missing
5     201       q1 Missing
6     202       q1 Missing
```

This magically works, but it is a bit puzzling why. It uses only subcode as the id variable.

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(dat3, id.vars = c("subcode"), value.name = "response", variable.name = "Question") |>
    head()
```

```
  subcode Question response
1     101       q1  Missing
2     102       q1  Missing
3     103       q1  Missing
4     104       q1  Missing
5     201       q1  Missing
6     202       q1  Missing
```

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

```
dat3 |>
    melt(id.vars = c("subcode"), measure.vars = c("q2", "q4", "q5"), value.name = "response",
        variable.name = "Question") |>
    head()
```

```
  subcode Question response
1     101       q2       20
2     102       q2       19
3     103       q2       20
4     104       q2       18
5     201       q2       19
6     202       q2       19
```

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)
gather(dat3, key = "question", value = "answer", q1, q2, q3, q4, q5) |>
    head()
```

```
  subcode question  answer
1     101       q1 Missing
2     102       q1 Missing
3     103       q1 Missing
4     104       q1 Missing
5     201       q1 Missing
6     202       q1 Missing
```

```
gather(dat3, key = "question", value = "answer", q2:q5) |>
    head()  #only q1 to q5
```

```
  subcode      q1 question answer
1     101 Missing       q2     20
2     102 Missing       q2     19
3     103 Missing       q2     20
4     104 Missing       q2     18
5     201 Missing       q2     19
6     202 Missing       q2     19
```

```
d3 <- gather(dat3, key = "question", value = "answer", -subcode, -q3)
head(d3)
```

```
  subcode      q3 question  answer
1     101 english       q1 Missing
2     102 english       q1 Missing
3     103 English       q1 Missing
4     104 English       q1 Missing
5     201 english       q1 Missing
6     202 english       q1 Missing
```

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

```
   subcode      q3 question  answer
1      101 english       q1 Missing
2      101 english       q2      20
3      101 english       q4      na
4      101 english       q5       1
5      102 english       q1 Missing
6      102 english       q2      19
7      102 english       q4    <NA>
8      102 english       q5       1
9      103 English       q1 Missing
10     103 English       q2      20
11     103 English       q4    <NA>
12     103 English       q5       1
13     104 English       q1 Missing
14     104 English       q2      18
15     104 English       q4    <NA>
16     104 English       q5       1
17     201 english       q1 Missing
```

```
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.

```
pivot_longer(dat3, names_to = "question", values_to = "answer", cols = c(q1, q2,
    q3, q4, q5)) |>
    head()
```

```
# A tibble: 6 x 3
  subcode question answer
    <int> <chr>    <chr>
1     101 q1       Missing
2     101 q2       20
3     101 q3       english
4     101 q4       na
5     101 q5       1
6     102 q1       Missing
```

```
pivot_longer(dat3, names_to = "question", values_to = "answer", cols = q1:q4) |>
    head()  #only q1 to q4; q5 gets put in the ID variables
```

```
# A tibble: 6 x 4
  subcode q5    question answer
    <int> <chr> <chr>    <chr>
1     101 1     q1       Missing
2     101 1     q2       20
3     101 1     q3       english
4     101 1     q4       na
5     102 1     q1       Missing
6     102 1     q2       19
```

```
pivot_longer(dat3, names_to = "question", values_to = "answer", cols = c(-subcode,
    -q1)) |>
    head()
```

```
# A tibble: 6 x 4
  subcode q1      question answer
    <int> <chr>   <chr>    <chr>
1     101 Missing q2       20
2     101 Missing q3       english
3     101 Missing q4       na
4     101 Missing q5       1
5     102 Missing q2       19
6     102 Missing q3       english
```

Using `spread` or `pivot_wider`

Spread reverses the gathering.

```
d3 |>
    spread(question, answer) |>
    head()
```

```
  subcode      q3      q1 q2   q4 q5
1     101 english Missing 20   na  1
2     102 english Missing 19 <NA>  1
3     103 English Missing 20 <NA>  1
4     104 English Missing 18 <NA>  1
5     201 english Missing 19 <NA>  1
6     202 english Missing 19   na  1
```

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

```
d3 |>
    pivot_wider(names_from = question, values_from = answer)
```

```
# A tibble: 24 x 6
   subcode q3      q1      q2    q4    q5
     <int> <chr>   <chr>   <chr> <chr> <chr>
 1     101 english Missing 20    na    1
 2     102 english Missing 19    <NA>  1
 3     103 English Missing 20    <NA>  1
 4     104 English Missing 18    <NA>  1
 5     201 english Missing 19    <NA>  1
 6     202 english Missing 19    na    1
 7     203 english Missing 19    na    1
 8     204 English Missing 20    <NA>  1
 9     206 english Missing 19    <NA>  1
10     207 english Missing 20    na    1
# i 14 more rows
```

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.

Exercise 3:

- 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 ]
```

In-class exercises

Look at the 'student performance' data set, which has five variables. https://www.kaggle.com/datasets/stealthtechnologies/predict-student-performance-dataset

The data are available here https://www.kaggle.com/datasets/stealthtechnologies/predict-student-performance-dataset?select=data.csv

We may not be able to download the data from kaggle directly, but once you download it, use ONLY tidyverse functions and a pipeline to do the following:

1. read it into a tibble
2. look at the tibble using a pipeline
3. make a new variable to bin SES into 5 levels
4. remove the middle level of SES from the data set
5. remove anyone with fewer than 1.0 study hours
6. select grades and SES variables
7. summarize to show the mean, SD, and N of grades by SES
8. do the same for attendance
9. Make a wide data frame out each of these, making SES the column variable, and rows indicating mean, sd, and N for grades. Do the same for attendance.
10. Bind the two table together row-wise. Be sure to add a column that indicates whether the data are for grades or attendance.
11. Make a long data frame out of this, with SES as the id variable, and the other variables as the measure variables. This should look like a little like the original data set.

Exercise Solutions

Exercise 1:

Rewrite the following code using piping:

```
# set.seed(100) dat.A <- sample(1:10,100,replace=T) dat.B <- table(dat.A) sd <-
# sd(dat.B)


sample(1:10, 100, replace = T) |>
    table() |>
    sd()
```

```
[1] 4.2947
```

Exercise 2

Rewrite the following old-style examples with dplyr functions and pipes where appropriate.

```
# dat[dat$cond=='C',]

dat |>
    filter(cond == "C") |>
    head()
```

```
# A tibble: 6 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
2 s     C         5  3.71
3 d     C         3  4.80
4 j     C         3  1.56
5 m     C         7  2.37
6 j     C         3  3.21
```

```
# dat[dat$dv1>2.5,]
dat |>
    filter(dv1 > 2.5) |>
    head()
```

```
# A tibble: 6 x 4
  sub   cond  group   dv1
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
2 p     B         1  4.40
3 z     B         4  4.56
4 x     B         2  4.33
5 k     B         8  2.57
6 s     C         5  3.71
```

```
# dat[1:10,]
dat |>
    slice(1:10) |>
    head()
```

```
# A tibble: 6 x 4
  sub   cond  group   dv1
```

```
  <chr> <chr> <int> <dbl>
1 f     C         3  3.32
2 p     B         1  4.40
3 z     B         4  4.56
4 f     A         9  1.95
5 x     B         2  4.33
6 k     B         8  2.57
```

```
# dat[,2:3]
dat |>
    select(2:3) |>
    head()
```

```
# A tibble: 6 x 2
  cond  group
  <chr> <int>
1 C         3
2 B         1
3 B         4
4 A         9
5 B         2
6 B         8
```

```
# dat[,c(3,1,2)]
dat |>
    select(3, 1, 2) |>
    head()
```

```
# A tibble: 6 x 3
  group sub   cond
  <int> <chr> <chr>
1     3 f     C
2     1 p     B
3     4 z     B
4     9 f     A
5     2 x     B
6     8 k     B
```

```
# datx <- dat colnames(datx) <-
# c('participant','condition','counterbalance','score')

dat |>
    rename(participant = sub, condition = cond, counterbalance = group, score = dv1)
```

```
# A tibble: 100 x 4
   participant condition counterbalance score
   <chr>       <chr>              <int> <dbl>
 1 f           C                      3  3.32
 2 p           B                      1  4.40
 3 z           B                      4  4.56
 4 f           A                      9  1.95
 5 x           B                      2  4.33
 6 k           B                      8  2.57
 7 s           C                      5  3.71
 8 d           C                      3  4.80
 9 q           A                      5  2.85
```

```
10 j           C                       3  1.56
# i 90 more rows
```

```
# dat0[dat0$question=='a',]
dat0 |>
    filter(question == "a") |>
    head()
```

```
# A tibble: 4 x 4
    sub question    dv coding
  <dbl> <chr>    <dbl>  <dbl>
1     1 a            5     -1
2     2 a            2      1
3     3 a            4     -1
4     4 a            1      1
```

```
# dat0[order(dat0$question),]
dat0 |>
    arrange(question) |>
    head()
```

```
# A tibble: 6 x 4
    sub question    dv coding
  <dbl> <chr>    <dbl>  <dbl>
1     1 a            5     -1
2     2 a            2      1
3     3 a            4     -1
4     4 a            1      1
5     1 b            3      1
6     2 b            3     -1
```

Exercise 3:

```
dat1 |>
    select(sub, question, newdv2) |>
    filter(sub != 2) |>
    mutate(question = dplyr::recode(question, a = "Arsenic", b = "Borox", c = "Carbon")) |>
    rename(participant = sub, element = question, happiness = newdv2) |>
    mutate(dvnormed = happiness - 5)
```

```
# A tibble: 9 x 4
  participant element happiness dvnormed
        <dbl> <chr>       <dbl>    <dbl>
1           1 Arsenic         5        0
2           1 Borox           9        4
3           1 Carbon          5        0
4           3 Arsenic         8        3
5           3 Borox           8        3
6           3 Carbon          9        4
7           4 Arsenic         5        0
8           4 Borox           9        4
9           4 Carbon          5        0
```

Exercise 4.

```
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))
```