An Introduction to Data Wrangling

I found this Twitter thread on the vagaries of data wrangling killing momentum interesting, particularly the notion of how frustrating it must be to be at point A with your data, see Point B, where you'd like to go, and have no idea how to get there. Even for those with programming experience, data wrangling can be an enormous chore.

To that end, I thought I'd walk through a basic overview of how to accomplish some of the operations you might commonly encounter when you first get a data set. If you're generating the data yourself, you can try to make your life easier by saving it in the format you want.

Where possible, I'll show multiple ways to accomplish something and try to highlight packages that will make things easier.

MANDATORY DISCLAIMER: There are often at least six ways to do anything in R. If you don't like any of the methods here, rest assured that there are others (and probably better ones, too); you can almost certainly find something that suits your style.

Reading in data

CSV Files

First, I'm going to create some data to actually read in. The following code chunk will write three csv files to the current directory, each with 3 columns of random data. This is meant to simulate raw data from three different subjects.

```
#Generate some dummy data
data <- replicate(3, mapply(rnorm, c(100, 100, 100),  c(10, 100, 1), c(2, 25, .5)),
                  simplify = FALSE)
catch_output <- mapply(write.csv, data,
       mapply(paste, rep("data", times=length(data)),
              seq(1, length(data)),
              rep(".csv"), sep=""), row.names=FALSE)
```

Here's a common situation: one spreadsheet has all of a subject's raw data, and you have a few dozen spreadsheets. First, let's talk about an easy way to read that in as painlessly as possible. No for-loops needed here; we'll just use the trusty `apply()` family from base R.

```
#Note that if your data are not in your current directory, you need to either:
#Call, for ex., setwd('~/my_data_folder') to set the data folder as the current directory
#Specify the path in list.files and then have it return the full path name of each file, rather than #t
alldata <- lapply(list.files(pattern = '*.csv'), read.csv)
```

We're accomplishing a few things in one line. First, the call to `list.files` simply lists all of the files in your current directory. It has a bunch of optional arguments, too. You can specify a pattern, which is just a regex expression that specifies what you want. Here, I only want .csv files, so I specify that I want any file ("*" is a wildcard symbol allowing anything) that ends in the extension .csv. I can specify other arguments, like whether I want the full path names returned, whether I want it to also search sub-directories for files, and so on.

After we have this list of files, we simply iterate over it and call `read.csv` on each one. The end result is a list, wherein each element is one subject's data frame.

Now, a list of data frames is not the *most* useful data format to have. Fortunately, it's easy to bind this list together into one big data frame. Here's how to bring it all together in base R.

```
subjects_all <- do.call('rbind', alldata)
head(subjects_all)
```

```
##          V1        V2          V3
## 1 10.284203  74.83840  0.71990864
## 2  8.874419 160.80261  0.79568419
## 3  9.457610 118.13179  0.91566817
## 4 10.630952  84.45278  0.81225294
## 5  9.700885 131.23012 -0.02204879
## 6 11.739639  82.43163  0.31490996
```

Note that the information about which data belong to each subject is lost here. You'll need to add an identifier column, or make sure that each file has one, before reading it in this way.

How about some good old dplyr?

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
subjects_all <- bind_rows(alldata, .id='subject')
head(subjects_all)
```

```
##   subject        V1        V2          V3
## 1       1 10.284203  74.83840  0.71990864
## 2       1  8.874419 160.80261  0.79568419
## 3       1  9.457610 118.13179  0.91566817
## 4       1 10.630952  84.45278  0.81225294
## 5       1  9.700885 131.23012 -0.02204879
## 6       1 11.739639  82.43163  0.31490996
```

```
unique(subjects_all$subject)
```

```
## [1] "1" "2" "3"
```

We can use the `.id` argument to specify an ID column, which will keep track of where the data comes from.

We can also use the handy rbindlist function from the data.table/dtplyr package. This will label the data automatically for us according to which data frame it came from; we can call this new column (specified by the id argument) anything we like.

```
library(data.table)
```

```
## -------------------------------------------------------------------------
```

```
## data.table + dplyr code now lives in dtplyr.
## Please library(dtplyr)!
```

```
## -------------------------------------------------------------------------
```

```
##
## Attaching package: 'data.table'
```

```
## The following objects are masked from 'package:dplyr':
##
##     between, last
```

```
subjects_all <- rbindlist(alldata, idcol='subject')
head(subjects_all)
```

```
##    subject        V1        V2          V3
## 1:       1 10.284203  74.83840  0.71990864
## 2:       1  8.874419 160.80261  0.79568419
## 3:       1  9.457610 118.13179  0.91566817
## 4:       1 10.630952  84.45278  0.81225294
## 5:       1  9.700885 131.23012 -0.02204879
## 6:       1 11.739639  82.43163  0.31490996
```

```
unique(subjects_all$subject)
```

```
## [1] 1 2 3
```

Note also that `rbindlist()` is an order of magnitude faster than `do.call`. If you've got a lot of data, you'll probably want to go with this function. `data.tables` are extremely fast and memory efficient in general, and might be a good option if you're working with truly huge amounts of data. For most uses, though, this kind of optimization isn't really necessary.

Text files

Same process. I'll make some real quick:

```
catch_output <- mapply(write.table, data,
       mapply(paste, rep("data", times=length(data)),
              seq(1, length(data)),
              rep(".txt"), sep=""), row.names=FALSE)
```

To read it in, we just call `read.table` instead.

```
allsubjs <- lapply(list.files(pattern = '*.txt'), read.table, header=TRUE, colClasses=c('double'))
head(allsubjs[[1]])
```

```
##          V1        V2          V3
## 1 10.284203  74.83840  0.71990864
## 2  8.874419 160.80261  0.79568419
## 3  9.457610 118.13179  0.91566817
## 4 10.630952  84.45278  0.81225294
## 5  9.700885 131.23012 -0.02204879
## 6 11.739639  82.43163  0.31490996
```

Just like before, we end up with a list of data frames, one for each subject. In `read.table()`, unlike `read.csv`, the `header` argument defaults to `FALSE`, so be sure to change that. I also specify `colClasses` here to tell R what type of data the content of the columns is. Without that, these doubles get read in as factors; doing it now saves a little work later.

We can then bind these together with `rbindlist` just like we did when we used `read.csv`.

XLS(X)

Need to read in Excel files, or read in each sheet in one file as a separate set of data?

Wickham's got your back.

Some general notes

There are a lot of arguments you can specify in the `read.csv` function call that can save you work down the line–I used some of them when I was reading in the text files, but there are many more. You can even tell the function what strings should be read as NA values! This is really handy if you have NULL and what R to treat that as NA. You can also read in only part of the file, which is useful if you have a monster file and want to read it in in chunks.

Reshaping Data

It's helpful to be able to switch at will between data in wide format, where each row is a subject and each column contains a variable, to long format, where each row is a value at a given time, or measure (for repeated measures).

Here's a very simple data set. Each row is a subject's scores. Column 1 is their subject number, followed by their scores in the control, treatment 1, and treatment 2 conditions. We tend to be most accustomed to seeing data this way. This is "wide" format.

```
traits <- data.frame('id'=seq(1, 10),
                     'control'=floor(rnorm(10, 30, 5)),
                     'treat1'=floor(rnorm(10, 10, 2)),
                     'treat2'=floor(rnorm(10, 15, 3)))
print(traits)
```

```
##    id control treat1 treat2
## 1   1      38     12     11
## 2   2      31     10     16
## 3   3      32     10     15
## 4   4      35      8      9
## 5   5      42      8     16
## 6   6      39     10     14
## 7   7      17     10     14
## 8   8      19      9     15
## 9   9      29     10     16
## 10 10      31      9     17
```

Wide to Long

Now, when we cast this to "long" format, we will have the id column (the subject number), a variable column (in this case, which test was taken), and the value column (the score on each test). Here it is, melted two ways. In base:

```
traits_long_base <- reshape(traits, idvar="id", direction='long', v.names=c('score'),
                      timevar='test',times=c('control', 'treat1', 'treat2'), varying=seq(2, 4))
print(traits_long_base)
```

```
##            id    test score
## 1.control   1 control    38
## 2.control   2 control    31
## 3.control   3 control    32
## 4.control   4 control    35
## 5.control   5 control    42
## 6.control   6 control    39
## 7.control   7 control    17
```

```
## 8.control   8 control    19
## 9.control   9 control    29
## 10.control 10 control    31
## 1.treat1    1  treat1    12
## 2.treat1    2  treat1    10
## 3.treat1    3  treat1    10
## 4.treat1    4  treat1     8
## 5.treat1    5  treat1     8
## 6.treat1    6  treat1    10
## 7.treat1    7  treat1    10
## 8.treat1    8  treat1     9
## 9.treat1    9  treat1    10
## 10.treat1  10  treat1     9
## 1.treat2    1  treat2    11
## 2.treat2    2  treat2    16
## 3.treat2    3  treat2    15
## 4.treat2    4  treat2     9
## 5.treat2    5  treat2    16
## 6.treat2    6  treat2    14
## 7.treat2    7  treat2    14
## 8.treat2    8  treat2    15
## 9.treat2    9  treat2    16
## 10.treat2  10  treat2    17
```

We have to be pretty careful about specifying our arguments here. The `idvar` indicates which column we want to map over the data. Here, we want the subject number; we want each subject's score on each test labeled with their unique ID. Direction is fairly self-explanatory; we're going to long form here. `v.names` is the name (or names) of the new columns. Here, we're collapsing everybody's scores into a single column, so we call it `'score'`. `timevar` is the variable that changes over time, or over repeated measures. Here it's which test they took, so we call the new column `'test'`. Then we tell it which values to use in this new times column with `times`; we want the name of the test. Then we tell it which columns of our data are varying over time/are our repeated measures; here it's the final three columns (you can also specify a vector of strings).

Here are the same results with the `melt()` function from data.table or reshape2. We specify again which column represents our data labels, and then we tell it which columns we want it to treat as our "measures," which in our case is our three tests (if unspecified, it just uses all non-id variables, so we could have left it out here):

```
traits_long_m <- melt(traits, id.vars="id", measure.vars=c('control', 'treat1', 'treat2'),
                      variable.name='test', value.name='score')
print(traits_long_m)
```

```
##    id    test score
## 1   1 control    38
## 2   2 control    31
## 3   3 control    32
## 4   4 control    35
## 5   5 control    42
## 6   6 control    39
## 7   7 control    17
## 8   8 control    19
## 9   9 control    29
## 10 10 control    31
## 11  1  treat1    12
## 12  2  treat1    10
## 13  3  treat1    10
```

```
## 14  4  treat1     8
## 15  5  treat1     8
## 16  6  treat1    10
## 17  7  treat1    10
## 18  8  treat1     9
## 19  9  treat1    10
## 20 10  treat1     9
## 21  1  treat2    11
## 22  2  treat2    16
## 23  3  treat2    15
## 24  4  treat2     9
## 25  5  treat2    16
## 26  6  treat2    14
## 27  7  treat2    14
## 28  8  treat2    15
## 29  9  treat2    16
## 30 10  treat2    17
```

And let's not leave out `tidyr`:

```
library(tidyr)
traits_long_t <- gather(traits, key=test, value=score, control, treat1, treat2)
print(traits_long_t)
```

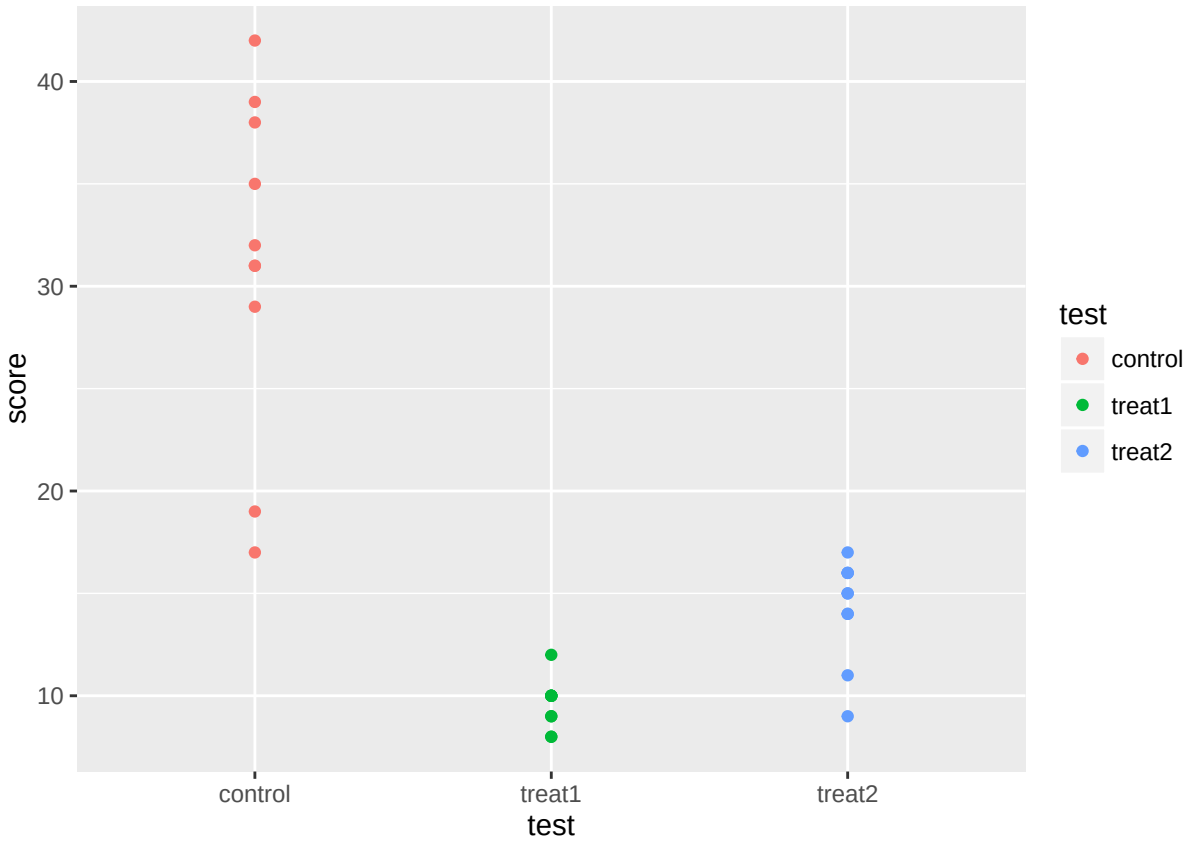
```
##    id    test score
## 1   1 control    38
## 2   2 control    31
## 3   3 control    32
## 4   4 control    35
## 5   5 control    42
## 6   6 control    39
## 7   7 control    17
## 8   8 control    19
## 9   9 control    29
## 10 10 control    31
## 11  1  treat1    12
## 12  2  treat1    10
## 13  3  treat1    10
## 14  4  treat1     8
## 15  5  treat1     8
## 16  6  treat1    10
## 17  7  treat1    10
## 18  8  treat1     9
## 19  9  treat1    10
## 20 10  treat1     9
## 21  1  treat2    11
## 22  2  treat2    16
## 23  3  treat2    15
## 24  4  treat2     9
## 25  5  treat2    16
## 26  6  treat2    14
## 27  7  treat2    14
## 28  8  treat2    15
## 29  9  treat2    16
## 30 10  treat2    17
```

Here, the key/value pairing tells us about our outcome columns, and then we just list the columns to gather up.

Three roads, same destination. I find `melt` and `gather` both much more intuitive than `reshape`, with `gather` the easiest of them all to use, but your mileage may vary.

Data is *really* easy to plot this way:

```
library(ggplot2)
plot <- ggplot(traits_long_t, aes(x=test, y=score, color=test)) +
        geom_point()
print(plot)
```



Simplicity itself.

Long to Wide

Now, if we want to go the other direction (long to wide), in base R, we call the same function with different arguments:

```
traits_wide_base <- reshape(traits_long_base, direction='wide', timevar='test', idvar='id')
print(traits_wide_base)
```

```
##            id score.control score.treat1 score.treat2
## 1.control   1            38           12           11
## 2.control   2            31           10           16
## 3.control   3            32           10           15
## 4.control   4            35            8            9
## 5.control   5            42            8           16
## 6.control   6            39           10           14
```

```
## 7.control   7            17           10           14
## 8.control   8            19            9           15
## 9.control   9            29           10           16
## 10.control 10            31            9           17
```

Now we have the original structure of our data back.

The inverse of `melt()` is `dcast`:

```
traits_wide_m <- dcast(traits_long_m, id ~ test, value.var='score')
print(traits_wide_m)
```

```
##    id control treat1 treat2
## 1   1      38     12     11
## 2   2      31     10     16
## 3   3      32     10     15
## 4   4      35      8      9
## 5   5      42      8     16
## 6   6      39     10     14
## 7   7      17     10     14
## 8   8      19      9     15
## 9   9      29     10     16
## 10 10      31      9     17
```

Right back to where we were.

And to undo `gather`, we `spread`:

```
traits_wide_t <- spread(traits_long_t, test, score)
print(traits_wide_t)
```

```
##    id control treat1 treat2
## 1   1      38     12     11
## 2   2      31     10     16
## 3   3      32     10     15
## 4   4      35      8      9
## 5   5      42      8     16
## 6   6      39     10     14
## 7   7      17     10     14
## 8   8      19      9     15
## 9   9      29     10     16
## 10 10      31      9     17
```

Reshaping with more variables

Here's a more complex example.

```
traittest <- data.frame('traitA'=factor(rep(c('high', 'med', 'low'), each=4)),
                        'traitB'=factor(rep(c('positive', 'negative'), times=6)),
                        'test1'=floor(rnorm(12, 10, 2)), 'test2'=floor(rnorm(12, 15, 2)))
head(traittest)
```

```
##   traitA   traitB test1 test2
## 1   high positive    10    11
## 2   high negative     8    15
## 3   high positive    11    14
## 4   high negative     4    12
## 5    med positive    10    15
```

```
## 6    med negative    11    13
```

There are a lot of ways to melt this data. Maybe we want to collpase the tests into a single column–in this case the traits are the identifier variables.

In base:

```
tt_bytrait_base <- reshape(traittest, direction='long', v.names='score',
                           timevar='test', times=c('test1', 'test2'), varying=c('test1','test2'))
print(tt_bytrait_base)
```

```
##          traitA   traitB  test score id
## 1.test1    high positive test1    10  1
## 2.test1    high negative test1     8  2
## 3.test1    high positive test1    11  3
## 4.test1    high negative test1     4  4
## 5.test1     med positive test1    10  5
## 6.test1     med negative test1    11  6
## 7.test1     med positive test1    10  7
## 8.test1     med negative test1    11  8
## 9.test1     low positive test1    10  9
## 10.test1    low negative test1     9 10
## 11.test1    low positive test1    11 11
## 12.test1    low negative test1     9 12
## 1.test2    high positive test2    11  1
## 2.test2    high negative test2    15  2
## 3.test2    high positive test2    14  3
## 4.test2    high negative test2    12  4
## 5.test2     med positive test2    15  5
## 6.test2     med negative test2    13  6
## 7.test2     med positive test2    15  7
## 8.test2     med negative test2    18  8
## 9.test2     low positive test2    15  9
## 10.test2    low negative test2    14 10
## 11.test2    low positive test2    15 11
## 12.test2    low negative test2    15 12
```

With `melt()`:

```
tt_bytrait_m <- melt(traittest, measure.vars=c('test1', 'test2'), variable.name='test',
                     value.name='score')
head(tt_bytrait_m)
```

```
##   traitA   traitB  test score
## 1   high positive test1    10
## 2   high negative test1     8
## 3   high positive test1    11
## 4   high negative test1     4
## 5    med positive test1    10
## 6    med negative test1    11
```

With `gather`:

```
tt_bytrait_t <- gather(traittest, test, score, test1, test2)
head(tt_bytrait_t)
```

```
##   traitA   traitB  test score
## 1   high positive test1    10
```

```
## 2   high negative test1     8
## 3   high positive test1    11
## 4   high negative test1     4
## 5    med positive test1    10
## 6    med negative test1    11
```

Or, we can let the tests be the identifiers, and collapse the traits into a single column.

Base:

```
tt_bytest_base <- reshape(traittest, direction='long', v.names='rating',
                          timevar='trait', times=c('traitA', 'traitB'),
                          varying=c('traitA','traitB'))
print(tt_bytest_base)
```

```
##           test1 test2  trait   rating id
## 1.traitA     10    11 traitA     high  1
## 2.traitA      8    15 traitA     high  2
## 3.traitA     11    14 traitA     high  3
## 4.traitA      4    12 traitA     high  4
## 5.traitA     10    15 traitA      med  5
## 6.traitA     11    13 traitA      med  6
## 7.traitA     10    15 traitA      med  7
## 8.traitA     11    18 traitA      med  8
## 9.traitA     10    15 traitA      low  9
## 10.traitA     9    14 traitA      low 10
## 11.traitA    11    15 traitA      low 11
## 12.traitA     9    15 traitA      low 12
## 1.traitB     10    11 traitB positive  1
## 2.traitB      8    15 traitB negative  2
## 3.traitB     11    14 traitB positive  3
## 4.traitB      4    12 traitB negative  4
## 5.traitB     10    15 traitB positive  5
## 6.traitB     11    13 traitB negative  6
## 7.traitB     10    15 traitB positive  7
## 8.traitB     11    18 traitB negative  8
## 9.traitB     10    15 traitB positive  9
## 10.traitB     9    14 traitB negative 10
## 11.traitB    11    15 traitB positive 11
## 12.traitB     9    15 traitB negative 12
```

melt:

```
tt_bytest_m <- melt(traittest, measure.vars=c('traitA', 'traitB'),
                    variable.name='trait', value.name='rating')
```

```
## Warning: attributes are not identical across measure variables; they will
## be dropped
```

```
head(tt_bytest_m)
```

```
##   test1 test2  trait rating
## 1    10    11 traitA   high
## 2     8    15 traitA   high
## 3    11    14 traitA   high
## 4     4    12 traitA   high
## 5    10    15 traitA    med
## 6    11    13 traitA    med
```

(We can ignore the warning; it's warning us about the fact that we're combining two factors that don't share levels, so it's coercing them all to characters.)

With `gather`:

```
tt_bytest_t <- gather(traittest, trait, rating, traitA, traitB)
```

```
## Warning: attributes are not identical across measure variables; they will
## be dropped
```

```
head(tt_bytest_t)
```

```
##   test1 test2  trait rating
## 1    10    11 traitA   high
## 2     8    15 traitA   high
## 3    11    14 traitA   high
## 4     4    12 traitA   high
## 5    10    15 traitA    med
## 6    11    13 traitA    med
```

(Same warning as above.)

Reformatting Data

So we've read data in, and can flip it between long and wide at will. Great, but what if the data itself needs to be fixed?

Recoding values

Let's say you have some data that look like this:

```
yesno <- data.frame('subj'=seq(1,10), 'resp'=rep(c('Y','N'), each=5))
print(yesno)
```

```
##    subj resp
## 1     1    Y
## 2     2    Y
## 3     3    Y
## 4     4    Y
## 5     5    Y
## 6     6    N
## 7     7    N
## 8     8    N
## 9     9    N
## 10   10    N
```

So we have 10 subjects, and each one responded either yes (Y) or no (N) to... something. But maybe we don't like the way this is coded; Y and N are hard to work with if we want to find average accuracy, for example. Maybe we want 1's and 0's instead, with which it is easy to do calculations.

If we want to recode these values, we have a few options. We can use indexing, of course, but there are also some functions that will save you some work.

Base has the `ifelse` function, which performs a logical comparison, and if true, returns the first value; else, the second:

```
yesno$resp <- ifelse(yesno$resp == 'Y', 1, 0)
print(yesno)
```

```
##    subj resp
## 1     1    1
## 2     2    1
## 3     3    1
## 4     4    1
## 5     5    1
## 6     6    0
## 7     7    0
## 8     8    0
## 9     9    0
## 10   10    0
```

If we have more than two alternatives, you'll have to use something like a `switch` statement:

```
yesnomaybe <- data.frame('subj'=seq(1,15), 'resp'=rep(c('Y','N','M'), each=5))
print(yesnomaybe)
```

```
##    subj resp
## 1     1    Y
## 2     2    Y
## 3     3    Y
## 4     4    Y
## 5     5    Y
## 6     6    N
## 7     7    N
## 8     8    N
## 9     9    N
## 10   10    N
## 11   11    M
## 12   12    M
## 13   13    M
## 14   14    M
## 15   15    M
```

Now we have three options. Maybe we want 'yes' to be 1, 'no' to be -1, and 'maybe' to be 0. Here's how you can do it with a `switch` statement and `sapply` to call it on each element:

```
yesnomaybe$resp <- sapply(yesnomaybe$resp, function(x) switch(as.character(x), 'Y'=1, 'N'=-1,
                                                              'M'=0))
print(yesnomaybe)
```

```
##    subj resp
## 1     1    1
## 2     2    1
## 3     3    1
## 4     4    1
## 5     5    1
## 6     6   -1
## 7     7   -1
## 8     8   -1
## 9     9   -1
## 10   10   -1
## 11   11    0
```

```
## 12   12    0
## 13   13    0
## 14   14    0
## 15   15    0
```

In `dplyr`, we have the `recode` function:

```
yesnomaybe$dplyr_recode <- recode(yesnomaybe$resp, `1`='yes', `-1`='no', `0`='maybe')
print(yesnomaybe)
```

```
##    subj resp dplyr_recode
## 1     1    1          yes
## 2     2    1          yes
## 3     3    1          yes
## 4     4    1          yes
## 5     5    1          yes
## 6     6   -1           no
## 7     7   -1           no
## 8     8   -1           no
## 9     9   -1           no
## 10   10   -1           no
## 11   11    0        maybe
## 12   12    0        maybe
## 13   13    0        maybe
## 14   14    0        maybe
## 15   15    0        maybe
```

Recoding, assuming you don't have to do it for a huge number of possibilities, goes pretty fast.

Adding variables

Variables can be added to an existing data frame just with the `$` operator:

```
df <- data.frame('x'=rnorm(20, 6), 'y'=rnorm(20))
print(df)
```

```
##           x           y
## 1  5.229902  1.25138733
## 2  6.364856  0.25513680
## 3  6.218524 -0.67475809
## 4  5.493366 -0.33847134
## 5  6.889623  2.56279875
## 6  7.536360  0.82758656
## 7  5.956727 -0.41493104
## 8  8.053959 -0.32020810
## 9  4.027838  0.66987603
## 10 6.917170  0.65721673
## 11 5.497130 -0.80898926
## 12 6.979512  0.57793781
## 13 6.206916  0.08844484
## 14 5.135846  0.38136344
## 15 6.111180  1.09808321
## 16 5.431460 -0.57164340
## 17 4.658127  0.96395191
## 18 4.560746 -0.10684925
## 19 4.357571 -1.02110087
```

```
## 20 5.313170 -0.41761461
```

```
df$z <- rnorm(20, 10)
print(df)
```

```
##           x           y         z
## 1  5.229902  1.25138733 10.744771
## 2  6.364856  0.25513680  9.601092
## 3  6.218524 -0.67475809 10.588714
## 4  5.493366 -0.33847134 11.283666
## 5  6.889623  2.56279875  9.121234
## 6  7.536360  0.82758656  9.306545
## 7  5.956727 -0.41493104  7.146117
## 8  8.053959 -0.32020810  8.718532
## 9  4.027838  0.66987603 10.158956
## 10 6.917170  0.65721673 10.623187
## 11 5.497130 -0.80898926  7.506135
## 12 6.979512  0.57793781  9.191577
## 13 6.206916  0.08844484  9.233813
## 14 5.135846  0.38136344 11.109850
## 15 6.111180  1.09808321 10.337297
## 16 5.431460 -0.57164340  9.803402
## 17 4.658127  0.96395191  8.189160
## 18 4.560746 -0.10684925  9.355298
## 19 4.357571 -1.02110087  9.672607
## 20 5.313170 -0.41761461  9.365573
```

If you need to manipulate two data vectors that are numeric, you can just add, multiply, etc. your columns together to perform these operations elementwise:

```
df$total <- with(df, x + y + z)
head(df)
```

```
##          x          y         z    total
## 1 5.229902  1.2513873 10.744771 17.22606
## 2 6.364856  0.2551368  9.601092 16.22108
## 3 6.218524 -0.6747581 10.588714 16.13248
## 4 5.493366 -0.3384713 11.283666 16.43856
## 5 6.889623  2.5627988  9.121234 18.57366
## 6 7.536360  0.8275866  9.306545 17.67049
```

You also have a lot of options in the `dplyr` library, notably `transform`:

```
df <- transform(df, x = -x)
head(df)
```

```
##           x          y         z    total
## 1 -5.229902  1.2513873 10.744771 17.22606
## 2 -6.364856  0.2551368  9.601092 16.22108
## 3 -6.218524 -0.6747581 10.588714 16.13248
## 4 -5.493366 -0.3384713 11.283666 16.43856
## 5 -6.889623  2.5627988  9.121234 18.57366
## 6 -7.536360  0.8275866  9.306545 17.67049
```

But now that we've updated a column, our total is wrong. Let's fix it with `transmute`:

```
df <- mutate(df, corrected_total = x + y + z)
head(df)
```

```
##           x          y         z    total corrected_total
## 1 -5.229902  1.2513873 10.744771 17.22606        6.766256
## 2 -6.364856  0.2551368  9.601092 16.22108        3.491373
## 3 -6.218524 -0.6747581 10.588714 16.13248        3.695432
## 4 -5.493366 -0.3384713 11.283666 16.43856        5.451828
## 5 -6.889623  2.5627988  9.121234 18.57366        4.794410
## 6 -7.536360  0.8275866  9.306545 17.67049        2.597771
```

Maybe I now want a dataframe just of the even numbers in the x column, and the residuals from total and corrected total (for… reasons). `transmute` is like mutate, but it throws away all the extra:

```
df_even <- transmute(df, x_ev=floor(x)%%2==0, residuals=total-corrected_total)
head(df_even)
```

```
##    x_ev residuals
## 1  TRUE  10.45980
## 2 FALSE  12.72971
## 3 FALSE  12.43705
## 4  TRUE  10.98673
## 5 FALSE  13.77925
## 6  TRUE  15.07272
```

If none of these methods fit the bill, you can call `apply` along all the columns or rows of your data frame and write a custom function to do whatever processing you need.

Factor levels as column labels

Let's take the unfortunate case of levels-as-columns, in which all the levels of a factor are columns, and people get a 1 or a 0 for each level instead of their value. Here's some example data:

```
levs <- data.frame('subj'=seq(1, 4), 'a'=c(0, 0, 1, 0), 'b'=c(1, 0, 0, 1), 'c'=c(0, 1, 0, 0))
print(levs)
```

```
##   subj a b c
## 1    1 0 1 0
## 2    2 0 0 1
## 3    3 1 0 0
## 4    4 0 1 0
```

So, what we have are three subjects, and a factor with three possible levels: A, B, and C. What we want is the subject and the actual level of their factor, so we need a 2-column matrix.

Here's one way we might do that (there are others) that uses some procedures I've already shown. First, we'll reshape the dataframe so that the factors end up in one column. This has the advantage of putting the actual values of the factors we want all in one place. Then we filter out the 0s, leaving behind only the levels the subject actually selected, drop the redundant ones colum, then put the subjects back in the right order.

For these examples, I'll print out each intermediate stage of manipulation so that you can see what's happening.

All about that base:

```
(lev_long <- reshape(levs, idvar='subj', direction='long', v.names='value', timevar='trait',
                     times=c('a', 'b', 'c'), varying=c('a', 'b', 'c')))
```

```
##     subj trait value
## 1.a    1     a     0
## 2.a    2     a     0
```

```
## 3.a    3     a     1
## 4.a    4     a     0
## 1.b    1     b     1
## 2.b    2     b     0
## 3.b    3     b     0
## 4.b    4     b     1
## 1.c    1     c     0
## 2.c    2     c     1
## 3.c    3     c     0
## 4.c    4     c     0
```

First, we reshape the data. We need all of the factor-related pieces of information in a single column. We have a column with the possible factor levels, and a column indicating 0 (not the subject's level) or 1 (the subject's level).

```
(lev_filtered <- with(lev_long, lev_long[value == 1, 1:2]))
```

```
##     subj trait
## 3.a    3     a
## 1.b    1     b
## 4.b    4     b
## 2.c    2     c
```

The second step just uses good old-fashioned indexing to keep all rows where the value is 1 (aka, the subject has that level), and to keep only the useful subject and trait columns; what the `with` function does is tell R to perform all operations with the supplied data set, so we can reference columns by isolated names rather than having to do the verbose `data_frame$column` syntax.

```
(lev_reformed_base <- lev_filtered[order(lev_filtered$subj),])
```

```
##     subj trait
## 1.b    1     b
## 2.c    2     c
## 3.a    3     a
## 4.b    4     b
```

The final step is reordering the data according to the subject column in ascending order. Now we've got our data in a much more sensible format.

Tidyr and dplyr make quick work of this. First, we gather:

```
(lev_g <- gather(levs, trait, value, a, b, c))
```

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

Filter out the 0s:

```
(lev_f <- filter(lev_g, value != 0))
```

```
##   subj trait value
## 1    3     a     1
## 2    1     b     1
## 3    4     b     1
## 4    2     c     1
```

Retain only the useful columns:

```
(lev_s <- select(lev_f, subj, trait))
```

```
##   subj trait
## 1    3     a
## 2    1     b
## 3    4     b
## 4    2     c
```

Finally, put the subjects back in order:

```
(lev_reform <- arrange(lev_s, subj))
```

```
##   subj trait
## 1    1     b
## 2    2     c
## 3    3     a
## 4    4     b
```

Here are those steps strung together with piping and thus obviating the need for all those separate variable assignments:

```
levs_reformed <- gather(levs, trait, value, a, b, c) %>%
                 filter(value != 0) %>%
                 select(subj, trait) %>%
                 arrange(subj)
print(levs_reformed)
```

```
##   subj trait
## 1    1     b
## 2    2     c
## 3    3     a
## 4    4     b
```

Charming!

What about if we have multiple factors? Here we have a test and a report, each of which has three possible levels: ABC and XYZ, respectively.

```
mfac <- data.frame('subj'=seq(1, 4), 'test.A'=c(0, 1, 0, 1), 'test.B'=c(1, 0, 0, 0),
                   'test.C'=c(0, 0, 1, 0), 'report.X'=c(1, 0, 0, 0),
                   'report.Y'=c(0, 1, 1, 0), 'report.Z'=c(0, 0, 0, 1))
print(mfac)
```

```
##   subj test.A test.B test.C report.X report.Y report.Z
## 1    1      0      1      0        1        0        0
## 2    2      1      0      0        0        1        0
## 3    3      0      0      1        0        1        0
## 4    4      1      0      0        0        0        1
```

So what we want is a dataframe with three columns: subject number, test, and report. Subject 1 picked test A and report X, subject 2 picked test A and report Y, and so on.

This gets a little more complicated. If we collapse everything into one column, we're going to have to then spread it back out to separate the factors. We've also got the item label merged to its type, which is a problem if we only want the letter designation.

Let's try with base. Here's the reshape-filter method:

```
mfac_long <- reshape(mfac, idvar='subj', direction='long', v.names='value', timevar='measure',
                     times=colnames(mfac)[-1], varying=colnames(mfac)[-1])
mfac_filtered <- with(mfac_long, mfac_long[value == 1, 1:2])
type_splits <- do.call(rbind, strsplit(mfac_filtered$measure, '.', fixed=TRUE))
mfac_sep <- data.frame('subj'=mfac_filtered$subj,
                       'type'=type_splits[,1],
                       'version'=type_splits[,2])
mfac_wide <- reshape(mfac_sep, idvar='subj', direction='wide', timevar='type')
(mfac_reformed_base <- mfac_wide[order(mfac_wide$subj),])
```

```
##   subj version.test version.report
## 3    1            B              X
## 1    2            A              Y
## 4    3            C              Y
## 2    4            A              Z
```

Pulling this off takes more finagling. Things are fine when we reshape and filter (note the trick used to save some verbage in reshape(); indexing with a negative excludes that item, so we're saying we want all column names except the first), but then we have to recover whether our factor was a test or a report *separately* of its type. This means we have to split the string using `strsplit`, bind the results into a matrix (because they automatically come out as a list), and then take those newly-made factors and reshape it wide again with the test type and report type as their own columns. One nice thing about this approach, in spite of its many steps, is that it's totally blind to the content of the labels (provided they are consistently delimited). If they're labeled in a cooperative way, you don't need to know how many labels there are or what they say, and they can be in any order.

Here's another base approach, from my BFF Kelly Chang. This one uses the `apply` function to sweep a filter down the dataframe, then repackage the results:

```
labels <- c('A', 'B', 'C', 'X', 'Y', 'Z')
filtered <- t(apply(mfac[,2:ncol(mfac)], 1, function(x) labels[x==1]))
mfac_kc <- data.frame(mfac$subj, filtered)
colnames(mfac_kc) <- c('subj', 'test', 'report')
print(mfac_kc)
```

```
##   subj test report
## 1    1    B      X
## 2    2    A      Y
## 3    3    C      Y
## 4    4    A      Z
```

Here, you would supply the labels, rather than recovering them from the data itself (as was done in the previous approach). Here, order is important; the labels need to be in the same order as the corresponding columns for the filter to work.

With `tidyr` and `dplyr`, this approach can look something like this (still agnostic to the label content):

```
mfac_reformed <- gather(mfac, measure, value, -subj) %>%
                 filter(value != 0) %>%
                 select(subj, measure) %>%
```

```
                separate(measure, c('test', 'type')) %>%
                spread(test, type) %>%
                arrange(subj)
print(mfac_reformed)
```

```
##   subj report test
## 1    1      X    B
## 2    2      Y    A
## 3    3      Y    C
## 4    4      Z    A
```

The first few steps are the same; melt everything down and toss the zero values. Then, we need a step to yank apart the measure's letter designation and its type. Fortunately, `tidyr` has a handy `separate` function that does just this; it pulls apart the joined values into two columns that we can label right away. Then, we need to spread our now distinct factor types back into columns–one for the test and one for the report–and sort by subject.

Note also that the intermediate steps in this last example, when we had to separate the two types of factors and get two separate ones back from the `report.X` format, which involved splitting the string and reshaping the data, can also be useful if you have data in this form, or if you have one big code for a condition or trial and at some point want to split it into its components. You can also use the `colsplit()` function from the `reshape2` package for this purpose.

Parting Thoughts

And there you have it–a brief introduction to some common data manipulation tasks, and a few ways to handle them. This is only the thinnest of samples of methods. There are lots of different ways to accomplish things, and packages to help you do it. Many of these methods will undoubtedly have my fingerprints all over them; one of the reasons I approached these problems the way I did is to show how learning a skill in one context–reshaping data for plotting, for example–can be useful in other contexts, like changing a data frame's fundamental structure. Many roads lead to the same place, and if you don't like this one, another will get you there just as comfortably, if not more so.