Reshaping data with the `reshape` package

Hadley Wickham.
`http://had.co.nz/reshape`

September 2006

Contents

1 Introduction

Reshaping data is a common task in practical data analysis, and it is usually tedious and unintuitive.

Data often has multiple levels of grouping (nested treatments, split plot designs, or repeated measurements) and typically requires investigation at multiple levels. For example, from a long term clinical study we may be interested in investigating relationships over time, or between times or patients or treatments. Performing these investigations fluently requires the data to be reshaped in different ways, but most software packages make it difficult to generalise these tasks and code needs to be written for each specific case.

While most practitioners are intuitively familiar with the idea of reshaping, it is useful define it a little more formally. Data reshaping is easiest to define with respect to aggregation. Aggregation is a common and familiar task where data is reduced and rearranged into a smaller, more convenient form, with a concomitant reduction in the amount of information. One commonly used aggregation procedure is Excel's Pivot tables. Reshaping involves a similar rearrangement, but preserves all original information; where aggregation reduces many cells in the original data set to one cell in the new dataset, reshaping preserves a one-to-one connection. These ideas are expanded and formalised in the next section.

In R, there are a number of general functions that can aggregate data, for example `tapply`, `by` and `aggregate`, and a function specifically for reshaping data, `reshape`. Each of these functions tends to deal well with one or two specific scenarios, and each requires slightly different input arguments. In practice, careful thought is required to piece together the correct sequence of operations to get your data into the form that you want. The `reshape` package overcomes these problems with a general conceptual framework that needs just two functions: `melt` and `cast`.

In this form it is difficult to investigate relationships between other facets of the data: between subjects, or treatments, or replicates. Reshaping the data allows us to explore these other relationships while still being able to use the familiar tools that operate on columns.

This document provides an introduction to the conceptual framework behind `reshape` with the two fundamental operations of melting and casting. I then provide a detailed description of `melt` and `cast` with plenty of examples. I discuss stamp, an extension of cast, and other useful functions in the `reshape` package. Finally, I provide some case studies using reshape in real life examples.

2 Conceptual framework

To help us think about the many ways we might rearrange a data set it is useful to think about data in a new way. Usually, we think about data in terms of a matrix or data frame, where we have observations in the rows and variables in the columns. For the purposes of reshaping, we can divide the variables into two groups: identifier and measured variables.

1. Identifier, or id, variables identify the unit that measurements take place on. Id variables are usually discrete, and are typically fixed by design. In ANOVA notation (Y_{ijk}), id variables are the indices on the variables (i, j, k).

2. Measured variables represent what is measured on that unit (Y).

It is possible to take this abstraction a step further and say there are only id variables and a value, where the id variables also identify what measured variable the value represents. For example, we could represent this data set, which has two id variables, subject and time,

	subject	time	age	weight	height
1	John Smith	1	33	90	2
2	Mary Smith	1			2

as:

	subject	time	variable	value
1	John Smith	1	age	33
2	John Smith	1	weight	90
3	John Smith	1	height	2
4	Mary Smith	1	height	2

where each row represents one observation of one variable. This operation is called melting and produces "molten" data. Compared to the original data set, it has a new id variable "variable", and a new column "value", which represents the value of that observation. We now have the data in a form in which there are only id variables and a value.

From this form, we can create new forms by specifying which variables should form the columns and rows. In the original data frame, the "variable" id variable forms the columns, and all identifiers form the rows. We don't have to specify all the original id variables in the new form. When we don't, the id variables no longer uniquely identify one row, and in this case we need a function that reduces these many numbers to one. This is called an aggregation function.

The following section describes the melting operation in detail with an implementation in R.

3 Melting data

Melting a data frame is a little trickier in practice than it is in theory. This section describes the practical use of the `melt` function in R.

The `melt` function needs to know which variables are measured and which are identifiers. This distinction should be obvious from your design: if you fixed the value, it is an id variable. If you don't specify them explicitly, `melt` will assume that any factor or integer column is an id variable. If you specify only one of measured and identifier variables, `melt` assumes that all the other variables are the other sort. For example, with the `smiths` dataset, as shown below, all the following calls have the same effect:

```
> str(smiths)
'data.frame': 2 obs. of  5 variables:
 $ subject: Factor w/ 2 levels "John Smith","Mary Smith": 1 2
 $ time   : int  1 1
 $ age    : num  33 NA
 $ weight : num  90 NA
 $ height : num  1.87 1.54


melt(smiths, id=c("subject","time"), measured=c("age","weight","height"))
melt(smiths, id=c("subject","time"))
melt(smiths, id=1:2)
melt(smiths, measured=c("age","weight","height"))
melt(smiths)

> melt(smiths)
      subject time variable value
1 John Smith    1      age  33.0
2 Mary Smith    1      age    NA
3 John Smith    1   weight  90.0
4 Mary Smith    1   weight    NA
5 John Smith    1   height   1.9
6 Mary Smith    1   height   1.5
```

Melt doesn't make many assumptions about your measured and id variables: there can be any number, in any order, and the values within the columns can be in any order too. There is only one assumption that `melt` makes: all measured values must be numeric. This is usually ok, because most of the time measured variables are numeric, but unfortunately if you are working with categorical or date measured variables, `reshape` isn't going to be much help. If you need this functionality, please let me know and I'll make it more of a priority.

3.1 Melting data with id variables encoded in column names

A more complicated case is where the variable names contain information about more than one variable. For example, here we have an experiment with two treatments (A and B) with data

4

recorded on two time points (1 and 2), and the column names represent both treatment and time.

```
> trial <- data.frame(id = factor(1:4), A1 = c(1, 2, 1, 2), A2 = c(2,
+     1, 2, 1), B1 = c(3, 3, 3, 3))
> (trialm <- melt(trial))
   id variable value
1   1       A1     1
2   2       A1     2
3   3       A1     1
4   4       A1     2
5   1       A2     2
6   2       A2     1
7   3       A2     2
8   4       A2     1
9   1       B1     3
10  2       B1     3
11  3       B1     3
12  4       B1     3
```

To fix this we need to create a time and treatment column after reshaping:

```
> (trialm <- cbind(trialm, colsplit(trialm$variable, names = c("treatment",
+     "time"))))
   id variable value treatment time
1   1       A1     1         A    1
2   2       A1     2         A    1
3   3       A1     1         A    1
4   4       A1     2         A    1
5   1       A2     2         A    2
6   2       A2     1         A    2
7   3       A2     2         A    2
8   4       A2     1         A    2
9   1       B1     3         B    1
10  2       B1     3         B    1
11  3       B1     3         B    1
12  4       B1     3         B    1
```

I'm not aware of any general way to do this, so you may need to modify the code in `colsplit` depending on your situation.

3.2 Melting arrays

Sometimes, especially if your data is highly balanced or crossed, the data you want to reshape may be stored in an `array`. In this case, each array index acts as an id variable, and the value in the cell

is the measured value. The melt method uses the `dimnames` component to determine the names and values of the id variables, as shown in this example:

```
> (a <- array(sample(1:6), c(3, 2, 1)))
, , 1

     [,1] [,2]
[1,]    6    3
[2,]    2    4
[3,]    1    5


> melt(a)
  X1 X2 X3 value
1  1  1  1     6
2  2  1  1     2
3  3  1  1     1
4  1  2  1     3
5  2  2  1     4
6  3  2  1     5

> dimnames(a) <- lapply(dim(a), function(x) LETTERS[1:x])
> melt(a)
  X1 X2 X3 value
1  A  A  A     6
2  B  A  A     2
3  C  A  A     1
4  A  B  A     3
5  B  B  A     4
6  C  B  A     5

> names(dimnames(a)) <- c("trt", "loc", "time")
> melt(a)
  trt loc time value
1   A   A    A     6
2   B   A    A     2
3   C   A    A     1
4   A   B    A     3
5   B   B    A     4
6   C   B    A     5
```

3.3 Missing values in molten data

Finally, it's important to discuss what happens to missing values when you melt your data. Explicitly coded missing values usually denote sampling zeros rather than structural missings, which are usually implicit in the data. Clearly a structural missing depends on the structure of the data and as we are changing the structure of the data, we might expect some changes to structural missings. Structural missings change from implicit to explicit when we change from a nested to a crossed structure. For example, imagine a dataset with two id variables, sex (male or female) and pregnant (yes or no). When the variables are nested (ie. both on the same dimension) then the missing value "pregnant male" is encoded by its absence. However, in a crossed view, we need to add an explicit missing as there will now be a cell which must be filled with something. This is illustrated below.

	sex	pregnant	value
1	male	no	10.00
2	female	no	14.00
3	female	yes	4.00

	sex	no	yes
1	female	14.00	4.00
2	male	10.00	

Continuing along this path, the molten form is a perfectly nested form: there are no crossings. For this reason, it is possible to encode all missing values implicitly (by omitting that combination of id variables) rather than explicitly (with an NA value).

However, you may expect these to be in the data frame, and it is a bad idea for a function to throw data away by default, so you need to explicitly state that implicit missing values are ok. In most cases it is safe to get rid of them, which you can do by using preserve.na = FALSE in the call to melt. The two different results are illustrated below.

```
> melt(smiths)
     subject time variable value
1 John Smith    1      age  33.0
2 Mary Smith    1      age    NA
3 John Smith    1   weight  90.0
4 Mary Smith    1   weight    NA
5 John Smith    1   height   1.9
6 Mary Smith    1   height   1.5

> melt(smiths, preserve.na = FALSE)
     subject time variable value
1 John Smith    1      age  33.0
2 John Smith    1   weight  90.0
3 John Smith    1   height   1.9
```

```
4 Mary Smith    1    height   1.5
```

If you don't use `preserve.na = FALSE` you will need to make sure to account for possible missing values when aggregating (§4.6, pg. 17), for example, by supplying `na.rm = TRUE` to `mean`, `sum` and `var`.

4 Casting molten data

Once you have your data in the molten form, you can use `cast` to create the form you want. `Cast` has two arguments that you will always supply:

- `data`: the molten data set to cast

- `formula`: the casting formula which describes the shape of the output format (if you omit this argument, cast will return the data frame to its pre-molten form)

Most of this section explains the different casting formulas you can use. It also explains the use of the other optional arguments to cast:

- `fun.aggregate`: aggregation function to use (if necessary)

- `margins`: what marginal values should be computed

- `subset`: only operate on a subset of the original data.

4.1 Basic use

The casting formula has the following basic form: `col_var_1 + col_var_2` $\sim$ `row_var_1 + row_var_2`. This describes which variables you want to appear in the columns and which in the rows. These variables need to come from the molten data frame or be one of the following special variables:

- `.` corresponds to no variable, useful when creating formulas of the form `.` $\sim$ `x` or `x` $\sim$ `.`

- `...` represents all variables not previously included in the casting formula. Including this in your formula will guarantee that no aggregation occurs. There can be only one `...` in a cast formula.

- `result_variable` is used when your aggregation formula returns multiple results. See §4.3, pg. 13 for more details.

The first set of examples illustrated reshaping: all the original variables are used.

```
> cast(smithsm, time + subject ~ variable)
  time    subject age weight height
1    1 John Smith  33     90    1.9
2    1 Mary Smith  NA     NA    1.5

> cast(smithsm, ... ~ variable)
     subject time age weight height
1 John Smith    1  33     90    1.9
2 Mary Smith    1  NA     NA    1.5

> cast(smithsm, ... ~ subject)
  time variable John.Smith Mary.Smith
```

```
1   1      age       33.0         NA
2   1   weight       90.0         NA
3   1   height        1.9        1.5


> cast(smithsm, ... ~ time)
    subject variable   X1
1 John Smith      age 33.0
2 John Smith   weight 90.0
3 John Smith   height  1.9
4 Mary Smith   height  1.5
```

The following examples demonstrate aggregation. See §4.6, pg. 17 for more details. These examples use the french fries dataset included in the **reshape** package. It is data from a sensory experiment on french fries, where different types of frier oil, **treatment**, were tested by different people, **subject**, over ten weeks **time**.

```
> ffm <- melt(french_fries, id = 1:4, preserve = FALSE)
> cast(ffm, . ~ ., length)
  value value.1
1 value    3471

> cast(ffm, treatment ~ ., length)
  treatment value
1         1  1159
2         2  1157
3         3  1155

> cast(ffm, . ~ treatment, length)
  value   X1   X2   X3
1 value 1159 1157 1155

> cast(ffm, rep ~ treatment, length)
  rep  X1  X2  X3
1   1 579 578 575
2   2 580 579 580

> cast(ffm, treatment ~ rep, length)
  treatment  X1  X2
1         1 579 580
2         2 578 579
3         3 575 580

> cast(ffm, treatment + rep ~ ., length)
  treatment rep value
```

```
1            1    1    579
2            1    2    580
3            2    1    578
4            2    2    579
5            3    1    575
6            3    2    580

> cast(ffm, rep + treatment ~ ., length)
  rep treatment value
1   1          1   579
2   1          2   578
3   1          3   575
4   2          1   580
5   2          2   579
6   2          3   580

> cast(ffm, . ~ treatment + rep, length)
  value X1_1 X1_2 X2_1 X2_2 X3_1 X3_2
1 value  579  580  578  579  575  580
```

As illustrated above, the order in which the row and column variables are specified in is very important. As with a contingency table there are many possible ways of displaying the same variables, and the way they are organised reveals different patterns in the data. Variables specified first vary slowest, and those specified last vary fastest. Because comparisons are made most easily between adjacent cells, the variable you are most interested in should be specified last, and the early variables should be thought of as conditioning variables. An additional constraint is that displays have limited width but essentially infinite length, so variables with many levels must be specified as row variables.

4.2 Margins

It's often useful to be able to add margins to your tables. What is a margin? It is marginal in the statistical sense: we have averaged over the other the variables. You can tell cast to display all margins with margins = TRUE, or list individual variables in a character vector, margins=c("subject","day"). There are two special margins, "grand_col" and "grand_row", which display margins for the overall columns and rows respectively. Margins are displayed with a "." instead of the value of the variable.

These examples illustrate some of the possible ways to use margins. I've used sum as the aggregation function so that you can check the results yourself. Note that changing the order and position of the variables in the cast formula affects the margins that appear.

```
> cast(ffm, treatment ~ ., sum, margins = TRUE)
  treatment value
1         1  3702
```

```
2       2 3640
3       3 3640

> cast(ffm, treatment ~ ., sum, margins = "grand_row")
  treatment value
1         1  3702
2         2  3640
3         3  3640

> cast(ffm, treatment ~ rep, sum, margins = TRUE)
  treatment   X1   X2     .
1         1 1857 1845  3702
2         2 1836 1804  3640
3         3 1739 1901  3640
4      <NA> 5433 5550 10983

> cast(ffm, treatment + rep ~ ., sum, margins = TRUE)
  treatment rep value
1         1   1  1857
2         1   2  1845
3         1  NA  3702
4         2   1  1836
5         2   2  1804
6         2  NA  3640
7         3   1  1739
8         3   2  1901
9         3  NA  3640

> cast(ffm, treatment + rep ~ time, sum, margins = TRUE)
   treatment rep   X1   X2   X3   X4   X5   X6   X7   X8  X9  X10     .
1          1   1  156  213  206  181  208  182  156  185 176  194  1857
2          1   2  216  195  194  154  204  185  158  216 122  201  1845
3          1  NA  373  408  400  335  412  366  314  402 298  396  3702
4          2   1  187  213  172  193  157  183  175  173 185  199  1836
5          2   2  168  157  186  187  173  215  172  189 145  212  1804
6          2  NA  355  370  358  380  330  398  347  362 330  411  3640
7          3   1  189  212  172  190  151  161  165  150 173  175  1739
8          3   2  217  180  199  192  183  192  218  175 164  182  1901
9          3  NA  406  392  372  382  334  353  384  325 337  357  3640
10      <NA>  NA 1134 1170 1129 1097 1076 1117 1045 1088 965 1163 10983

> cast(ffm, treatment + rep ~ time, sum, margins = "treatment")
  treatment rep  X1  X2  X3  X4  X5  X6  X7  X8  X9 X10
1         1   1 156 213 206 181 208 182 156 185 176 194
```

```
2          1   2 216 195 194 154 204 185 158 216 122 201
3          1  NA 373 408 400 335 412 366 314 402 298 396
4          2   1 187 213 172 193 157 183 175 173 185 199
5          2   2 168 157 186 187 173 215 172 189 145 212
6          2  NA 355 370 358 380 330 398 347 362 330 411
7          3   1 189 212 172 190 151 161 165 150 173 175
8          3   2 217 180 199 192 183 192 218 175 164 182
9          3  NA 406 392 372 382 334 353 384 325 337 357

> cast(ffm, rep + treatment ~ time, sum, margins = "rep")
  rep treatment   X1  X2  X3  X4  X5  X6  X7  X8  X9 X10
1   1           1 156 213 206 181 208 182 156 185 176 194
2   1           2 187 213 172 193 157 183 175 173 185 199
3   1           3 189 212 172 190 151 161 165 150 173 175
4   1        <NA> 532 637 550 564 515 526 497 509 534 568
5   2           1 216 195 194 154 204 185 158 216 122 201
6   2           2 168 157 186 187 173 215 172 189 145 212
7   2           3 217 180 199 192 183 192 218 175 164 182
8   2        <NA> 601 533 579 533 560 591 548 580 430 595
```

4.3 Returning multiple values

Occasionally it is useful to aggregate with a function that returns multiple values, e.g. `range` or `summary`. This can be thought of as combining multiple `cast`s each with an aggregation function that returns one variable. To display this we need to add an extra variable, `result_variable` that differentiates the multiple return values. By default, this new id variable will be shown as the last column variable, but you can specify the position manually by including `result_variable` in the casting formula.

```
> cast(ffm, treatment ~ ., summary)
  treatment Min. X1st.Qu. Median Mean X3rd.Qu. Max.
1         1    0        0    1.6  3.2      5.4   15
2         2    0        0    1.4  3.1      5.4   15
3         3    0        0    1.5  3.1      5.7   14

> cast(ffm, treatment ~ ., quantile, c(0.05, 0.5, 0.95))
  treatment X5. X50. X95.
1         1   0  1.6   11
2         2   0  1.4   11
3         3   0  1.5   11

> cast(ffm, treatment ~ rep, range)
  treatment X1_X1 X1_X2 X2_X1 X2_X2
1         1     0    15     0    14
```

```
2           2    0    15    0    14
3           3    0    14    0    14

> named.range <- function(x) c(min = min(x), max = max(x))
> cast(ffm, treatment ~ rep, named.range)
  treatment X1_min X1_max X2_min X2_max
1         1      0     15      0     14
2         2      0     15      0     14
3         3      0     14      0     14

> cast(ffm, treatment ~ result_variable + rep, named.range)
  treatment min_1 min_2 max_1 max_2
1         1     0     0    15    14
2         2     0     0    15    14
3         3     0     0    14    14

> cast(ffm, treatment ~ rep ~ result_variable, named.range)
, , 1

  [,1] [,2]
1    0    0
2    0    0
3    0    0

, , 2

  [,1] [,2]
1   15   14
2   15   14
3   14   14
```

Returning multidimensional objects (eg. matrices or arrays) from an aggregation doesn't currently work very well. However, you can probably work around this deficiency by creating a high-D array, and then using `iapply`.

4.4 High-dimensional arrays

You can use more than one $\sim$ to create structures with more than two dimensions. For example, a cast formula of x $\sim$ y $\sim$ z will create a 3D array with x, y, and z dimensions. You can also still use multiple variables in each dimension: x + a $\sim$ y + b $\sim$ z + c. The following example shows the resulting dimensionality of various casting formulas (I don't show the actual output because it would be too large. You may want to verify the results for yourself):

14

```
> dim(cast(ffm, time ~ variable ~ treatment, mean))
[1] 10  5  3

> dim(cast(ffm, time ~ variable ~ treatment + rep, mean))
[1] 10  5  6

> dim(cast(ffm, time ~ variable ~ treatment ~ rep, mean))
[1] 10  5  3  2

> dim(cast(ffm, time ~ variable ~ subject ~ treatment ~ rep))
[1] 10  5 12  3  2

> dim(cast(ffm, time ~ variable ~ subject ~ treatment ~ result_variable,
+     range))
[1] 10  5 12  3  2
```

The high-dimensional array form is useful for sweeping out margins with `sweep`, or modifying with `iapply` (§**??**, pg. **??**). See the case studies for examples.

The ∼ operator is a type of crossing operator, as all combinations of the variables will appear in the output table. Compare this to the + operator, where only combinations that appear in the data will appear in the output. For this reason, increasing the dimensionality of the output, i.e. using more ∼s, will generally increase the number of (structural) missings. This is illustrated in the next example:

```
> sum(is.na(cast(ffm, ... ~ .)))
[1] 0

> sum(is.na(cast(ffm, ... ~ rep)))
[1] 9

> sum(is.na(cast(ffm, ... ~ subject)))
[1] 129

> sum(is.na(cast(ffm, ... ~ time ~ subject ~ variable ~ rep)))
[1] 129
```

Unfortunately, margins currently don't work with high-dimensional arrays. If you need this functionality, please let me know and I'll make it more a priority. Bribes always help too.

4.5 Lists

You can also use `cast` to produce lists. This is done with the | operator. Using multiple variables after | will create multiple levels of nesting.

```
> cast(ffm, treatment ~ rep | variable, mean)
$potato
  treatment  X1  X2
1         1 6.8 7.0
2         2 7.2 6.8
3         3 6.9 7.0

$buttery
  treatment  X1  X2
1         1 1.8 1.8
2         2 2.0 2.0
3         3 1.8 1.6

$grassy
  treatment   X1   X2
1         1 0.45 0.85
2         2 0.69 0.64
3         3 0.59 0.77

$rancid
  treatment  X1  X2
1         1 4.3 3.8
2         2 3.7 3.5
3         3 3.8 4.0

$painty
  treatment  X1  X2
1         1 2.7 2.4
2         2 2.3 2.6
3         3 2.0 3.0


> length(cast(ffm, treatment ~ rep | variable, mean))
[1] 5

> length(cast(ffm, treatment ~ rep | subject, mean))
[1] 12

> length(cast(ffm, treatment ~ rep | time, mean))
[1] 10

> sapply(cast(ffm, treatment ~ rep | time + variable, mean), length)
 1  2  3  4  5  6  7  8  9 10
 5  5  5  5  5  5  5  5  5  5
```

This form is useful for input to `lapply` and `sapply`, and completes the discussion of the different types of output you can create with reshape. The rest of the section discusses the other options available for output.

4.6 Aggregation

Whenever there are fewer cells in the cast form than there were in the original data format, an aggregation function is necessary. This formula reduces multiple cells into one, and is supplied in the `fun.aggregate` argument, which defaults (with a warning) to `length`. Aggregation is a very common and useful operation and the case studies section (§6, pg. 19) contains many other examples of aggregation.

The aggregation function will be passed the vector of a values for one cell. It may take other arguments, passed in through `...` in `cast`. Here are a few examples:

```
> cast(ffm, . ~ treatment)
Warning:  Aggregation requires fun.aggregate: length used as default
  value   X1   X2   X3
1 value 1159 1157 1155

> cast(ffm, . ~ treatment, function(x) length(x))
  value   X1   X2   X3
1 value 1159 1157 1155

> cast(ffm, . ~ treatment, length)
  value   X1   X2   X3
1 value 1159 1157 1155

> cast(ffm, . ~ treatment, sum)
  value   X1   X2   X3
1 value 3702 3640 3640

> cast(ffm, . ~ treatment, mean)
  value  X1  X2  X3
1 value 3.2 3.1 3.2

> cast(ffm, . ~ treatment, mean, trim = 0.1)
  value  X1  X2  X3
1 value 2.6 2.5 2.6
```

You can also display margins and use functions that return multiple results. See the next two sections for details.

5 Other convenience functions

There are many other problems encountered in practical analysis that can be painful to overcome with some handy functions. This section describes some of the functions that reshape provides to make dealing with data a little bit easier.

5.1 Factors

- `combine_factor` combines levels in a factor. For example, if you have many small levels you can combine them together into an "other" level.

- `reorder_factor` reorders a factor based on another variable. For example, you can order a factor by the average value of a variable for each level.

5.2 Data frames

- `rescaler` performs column-wise rescaling of data frames, with a variety of different scaling options including rank, common range and common variance. It automatically preserves non-numeric variables.

- `merge.all` merges multiple data frames together, an extension of `merge` in base R. It assumes that all columns with the same name should be equated.

- `rbind.fill` `rbind`s two data frames together, filling in any missing columns in the second data frame with missing values.

5.3 Miscellaneous

- `round_any` allows you to round a number to any degree of accuracy, e.g. to the nearest 1, 10, or any other number.

- `iapply` is an idempotent version of the `apply` function. This is useful when dealing with high-dimensional arrays as it will return the array in the same shape that you sent it. It also supports functions that return matrices or arrays in a sensible manner.

6 Case studies

These case studies provide fuller exposition of using reshape for specific tasks.

6.1 Investigating balance

This data is from a sensory experiment investigating the effect of different frying oils on the taste of french fries over time. There are three different types of frying oils (treatment), each in two different fryers (rep), tested by 12 people (subject) on 10 different days (time). The sensory attributes recorded, in order of desirability, are potato, buttery, grassy, rancid, painty flavours. The first few rows of the data look like:

	time	treatment	subject	rep	potato	buttery	grassy	rancid	painty
61	1	1	3	1.00	2.90	0.00	0.00	0.00	5.50
25	1	1	3	2.00	14.00	0.00	0.00	1.10	0.00
62	1	1	10	1.00	11.00	6.40	0.00	0.00	0.00
26	1	1	10	2.00	9.90	5.90	2.90	2.20	0.00
63	1	1	15	1.00	1.20	0.10	0.00	1.10	5.10
27	1	1	15	2.00	8.80	3.00	3.60	1.50	2.30

We first melt the data to use in subsequent analyses.

```
> ffm <- melt(french_fries, id = 1:4, preserve.na = FALSE)
> head(ffm)
  time treatment subject rep variable value
1    1         1       3   1   potato   2.9
2    1         1       3   2   potato  14.0
3    1         1      10   1   potato  11.0
4    1         1      10   2   potato   9.9
5    1         1      15   1   potato   1.2
6    1         1      15   2   potato   8.8
```

One of the first things we might be interested in is how balanced this design is, and whether there are many different missing values. We are interested in missingness, so remove missings to put structural and non-structural on an equal footing with `preserve.na = FALSE`. We can investigate balance using length as our aggregation function:

```
> cast(ffm, subject ~ time, length, margins = TRUE)
  subject X1 X2 X3 X4 X5 X6 X7 X8 X9 X10   .
1       3 30 30 30 30 30 30 30 30 30  NA 270
2      10 30 30 30 30 30 30 30 30 30  30 300
3      15 30 30 30 30 25 30 30 30 30  30 295
4      16 30 30 30 30 30 30 30 29 30  30 299
5      19 30 30 30 30 30 30 30 30 30  30 300
```

```
6        31   30  30  30  30  30  30  30  30   NA  30  270
7        51   30  30  30  30  30  30  30  30   30  30  300
8        52   30  30  30  30  30  30  30  30   30  30  300
9        63   30  30  30  30  30  30  30  30   30  30  300
10       78   30  30  30  30  30  30  30  30   30  30  300
11       79   30  30  30  30  30  30  29  28   30  NA  267
12       86   30  30  30  30  30  30  30  30   NA  30  270
13     <NA>  360 360 360 360 355 360 359 357  300 300 3471
```

Of course we can also use our own aggregation function. Each subject should have had 30 observations at each time, so by displaying the difference we can more easily see where the data is missing.

```
> cast(ffm, subject ~ time, function(x) 30 - length(x))
   subject X1 X2 X3 X4 X5 X6 X7 X8 X9 X10
1        3  0  0  0  0  0  0  0  0  0  NA
2       10  0  0  0  0  0  0  0  0  0   0
3       15  0  0  0  0  5  0  0  0  0   0
4       16  0  0  0  0  0  0  0  1  0   0
5       19  0  0  0  0  0  0  0  0  0   0
6       31  0  0  0  0  0  0  0  0 NA   0
7       51  0  0  0  0  0  0  0  0  0   0
8       52  0  0  0  0  0  0  0  0  0   0
9       63  0  0  0  0  0  0  0  0  0   0
10      78  0  0  0  0  0  0  0  0  0   0
11      79  0  0  0  0  0  0  1  2  0  NA
12      86  0  0  0  0  0  0  0  0 NA   0
```

We can also easily see the range of values that each variable takes:

```
> cast(ffm, variable ~ ., function(x) c(min = min(x), max = max(x)))
  variable min max
1   potato   0  15
2  buttery   0  11
3   grassy   0  11
4   rancid   0  15
5   painty   0  13
```

6.2 Tables of means

When creating these tables, it is a good idea to restrict the number of digits displayed. You can do this globally, by setting options(digits=2), or locally, by using round_any.

Since the data is fairly well balanced, we can do some (crude) investigation as to the effects of the different treatments. For example, we can calculate the overall means for each sensory attribute for each treatment:

```
> options(digits = 2)
> cast(ffm, treatment ~ variable, mean, margins = c("grand_col",
+     "grand_row"))
  treatment potato buttery grassy rancid painty    .
1         1    6.9     1.8   0.65    4.1    2.6 3.2
2         2    7.0     2.0   0.66    3.6    2.5 3.1
3         3    7.0     1.7   0.68    3.9    2.5 3.2
4      <NA>    7.0     1.8   0.66    3.9    2.5 3.2
```

It doesn't look like there is any effect of treatment. This can be confirmed using a more formal analysis of variance.

6.3 Investigating inter-rep reliability

Since we have a repetition over treatments, we might be interested in how reliable each subject is: are the scores for the two reps highly correlated? We can explore this graphically by reshaping the data and plotting the data. Our graphical tools work best when the things we want to compare are in different columns, so we'll cast the data to have a column for each rep.

```
> qplot(X1, X2, . ~ variable, data = cast(ffm, ... ~ rep))
```
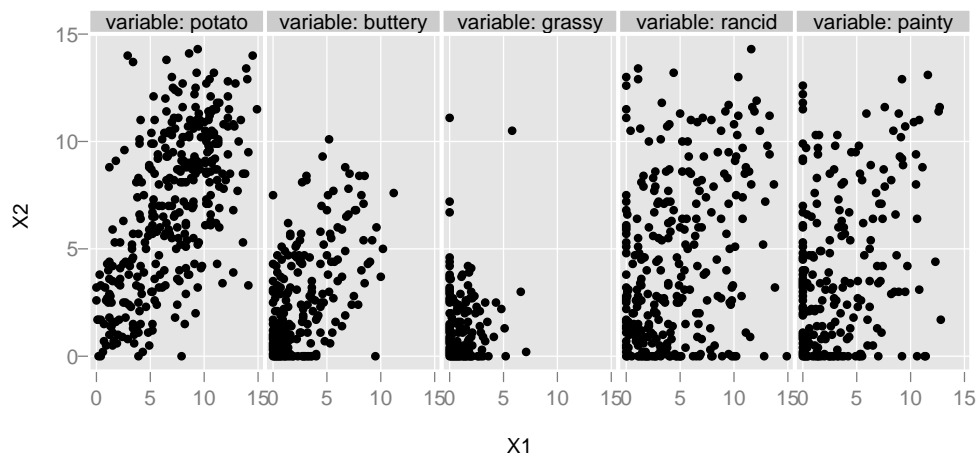

The correlation looks strong for potatoey, weak for buttery and grassy, and particularly poor for painty.

If we wanted to explore the relationships between subjects or times or treatments we could follow similar steps.

7 Where to go next

Now that you've read this introduction, you should be able to get started using the `reshape` package. You can find a quick reference and more examples in `?melt` and `?cast`. You can find some additional information on the reshape website `http://had.co.nz/reshape`, including the latest version of this document, as well as copies of presentations and papers related to reshape.

I would like to include more case studies of reshape in use. If you have an interesting example, or there is something you are struggling with please let me know: h.wickham@gmail.com.