Chapter 3: Data Wrangling

Mark Andrews
Psychology Department, Nottingham Trent University

✉ mark.andrews@ntu.ac.uk

Data wrangling with the `tidyverse`

- There are many tools in R for doing data wrangling.
- Here, we will focus of a core set of inter-related `tidyverse` tools.
- These include the commands available in the `dplyr` package, particularly its so-called *verbs* such as the following:
    - select
    - rename
    - slice
    - filter
    - mutate
    - arrange
    - group_by
    - summarize

Data wrangling with the `tidyverse`

- In addition, `dplyr` provides tools for merging and joining data sets.
- Next, there are the tools in the `tidyr` package, particularly the following:
  - gather
  - spread
  - unite
  - separate
  - pivot_longer
  - pivot_wider
- Other packages such as `lubridate` and `stringr` provide essential tools for dealing with and manipulating dates and strings, respectively.
- All of these tools are can be combined together using the `%>%` pipe operator

The `dplyr` verbs

As an example data set, we will use the data contained in the file `blp-trials-short.txt`.

```r
blp <- read_csv('data/blp-trials-short.txt')
```

Select variables with `select`

The `dplyr` command `select` allows us to select columns from a data frame. For example, if we just want `participant`, `lex`, `resp`, and `rt`, then we would do the following.

```
select(blp, participant, lex, resp, rt)
#> # A tibble: 1,000 x 4
#>    participant lex   resp     rt
#>          <dbl> <chr> <chr> <dbl>
#>  1          20 N     N       977
#>  2           9 N     N       565
#>  3          47 N     N       562
#>  4         103 N     N       572
#>  5          45 W     W       659
#>  6          73 W     W       538
#>  7          24 W     W       626
#>  8          11 W     W       566
#>  9          32 W     W       922
#> 10          96 N     N       555
#> # ... with 990 more rows
```

Select variables with `select`

We can select a range of variables by specifying the first and last variables in the range with a `:` between them as follows.

```
select(blp, spell:prev.rt)
#> # A tibble: 1,000 x 4
#>    spell    resp     rt prev.rt
#>    <chr>    <chr> <dbl>   <dbl>
#>  1 staud    N       977     511
#>  2 dinbuss  N       565     765
#>  3 snilling N       562     496
#>  4 gancens  N       572     656
#>  5 filled   W       659     981
#>  6 journals W       538    1505
#>  7 apache   W       626     546
#>  8 flake    W       566     717
#>  9 reliefs  W       922    1471
#> 10 sarves   N       555     806
#> # ... with 990 more rows
```

Select variables with `select`

We can also select a range of variables using indices as in the following example.

```
select(blp, 2:5) # columns 2 to 5
#> # A tibble: 1,000 x 4
#>    lex   spell    resp     rt
#>    <chr> <chr>    <chr> <dbl>
#>  1 N     staud    N       977
#>  2 N     dinbuss  N       565
#>  3 N     snilling N       562
#>  4 N     gancens  N       572
#>  5 W     filled   W       659
#>  6 W     journals W       538
#>  7 W     apache   W       626
#>  8 W     flake    W       566
#>  9 W     reliefs  W       922
#> 10 N     sarves   N       555
#> # ... with 990 more rows
```

Select variables with `select`

We can select variables according to the character or characters that they begin with. For example, we select all variables that being with `p` as follows.

```
select(blp, starts_with('p'))
#> # A tibble: 1,000 x 2
#>    participant prev.rt
#>          <dbl>   <dbl>
#>  1          20     511
#>  2           9     765
#>  3          47     496
#>  4         103     656
#>  5          45     981
#>  6          73    1505
#>  7          24     546
#>  8          11     717
#>  9          32    1471
#> 10          96     806
#> # ... with 990 more rows
```

Select variables with `select`

Or we can select variables by the characters they end with.

```
select(blp, ends_with('t'))
#> # A tibble: 1,000 x 3
#>    participant    rt prev.rt
#>          <dbl> <dbl>   <dbl>
#>  1          20   977     511
#>  2           9   565     765
#>  3          47   562     496
#>  4         103   572     656
#>  5          45   659     981
#>  6          73   538    1505
#>  7          24   626     546
#>  8          11   566     717
#>  9          32   922    1471
#> 10          96   555     806
#> # ... with 990 more rows
```

Select variables with `select`

We can select variables that contain a certain set of characters in any position. For example, the following selects variables whose names contain the string `rt`.

```
select(blp, contains('rt'))
#> # A tibble: 1,000 x 4
#>    participant    rt prev.rt rt.raw
#>          <dbl> <dbl>   <dbl>  <dbl>
#>  1          20   977     511    977
#>  2           9   565     765    565
#>  3          47   562     496    562
#>  4         103   572     656    572
#>  5          45   659     981    659
#>  6          73   538    1505    538
#>  7          24   626     546    626
#>  8          11   566     717    566
#>  9          32   922    1471    922
#> 10          96   555     806    555
#> # ... with 990 more rows
```

Select variables with `select`

We can also match by regular expressions. For example, the regular expression `^rt|rt$` will match the `rt` if it begins or ends a string. Therefore, we can select the variables that contain `rt`, where the string `rt` means reaction time, as follows.

```
select(blp, matches('^rt|rt$'))
#> # A tibble: 1,000 x 3
#>       rt prev.rt rt.raw
#>    <dbl>   <dbl>  <dbl>
#>  1   977     511    977
#>  2   565     765    565
#>  3   562     496    562
#>  4   572     656    572
#>  5   659     981    659
#>  6   538    1505    538
#>  7   626     546    626
#>  8   566     717    566
#>  9   922    1471    922
#> 10   555     806    555
#> # ... with 990 more rows
```

Remove variables with `select`

We can use `select` to *remove* variables as well as select them. To remove a variable, we precede its name with a minus sign.

```
select(blp, -participant) # remove `participant`
#> # A tibble: 1,000 x 6
#>    lex   spell    resp     rt prev.rt rt.raw
#>    <chr> <chr>    <chr> <dbl>   <dbl>  <dbl>
#>  1 N     staud    N       977     511    977
#>  2 N     dinbuss  N       565     765    565
#>  3 N     snilling N       562     496    562
#>  4 N     gancens  N       572     656    572
#>  5 W     filled   W       659     981    659
#>  6 W     journals W       538    1505    538
#>  7 W     apache   W       626     546    626
#>  8 W     flake    W       566     717    566
#>  9 W     reliefs  W       922    1471    922
#> 10 N     sarves   N       555     806    555
#> # ... with 990 more rows
```

Remove variables with `select`

Just as we selected ranges or sets of variables above, we can remove them by preceding their selection functions with minus signs.

```
select(blp, -(2:6))
#> # A tibble: 1,000 x 2
#>    participant rt.raw
#>          <dbl>  <dbl>
#>  1          20    977
#>  2           9    565
#>  3          47    562
#>  4         103    572
#>  5          45    659
#>  6          73    538
#>  7          24    626
#>  8          11    566
#>  9          32    922
#> 10          96    555
#> # ... with 990 more rows
```

Remove variables with `select`

Or, as another example, we can remove the variables that contain the string `rt` as follows.

```
select(blp, -contains('rt'))
#> # A tibble: 1,000 x 3
#>    lex   spell    resp
#>    <chr> <chr>    <chr>
#>  1 N     staud    N
#>  2 N     dinbuss  N
#>  3 N     snilling N
#>  4 N     gancens  N
#>  5 W     filled   W
#>  6 W     journals W
#>  7 W     apache   W
#>  8 W     flake    W
#>  9 W     reliefs  W
#> 10 N     sarves   N
#> # ... with 990 more rows
```

Renaming variables with select

When we select individual variables with `select`, we can rename them too, as in the following example.

```
select(blp, subject=participant, reaction_time=rt)
#> # A tibble: 1,000 x 2
#>    subject reaction_time
#>      <dbl>         <dbl>
#>  1      20           977
#>  2       9           565
#>  3      47           562
#>  4     103           572
#>  5      45           659
#>  6      73           538
#>  7      24           626
#>  8      11           566
#>  9      32           922
#> 10      96           555
#> # ... with 990 more rows
```

Renaming variables with `rename`

If we want to rename some variables, and get a data frame with all variables, including the renamed ones, we should use `rename`.

```
rename(blp, subject=participant, reaction_time=rt)
#> # A tibble: 1,000 x 7
#>    subject lex   spell    resp  reaction_time prev.rt rt.raw
#>      <dbl> <chr> <chr>    <chr>         <dbl>   <dbl>  <dbl>
#>  1      20 N     staud    N               977     511    977
#>  2       9 N     dinbuss  N               565     765    565
#>  3      47 N     snilling N               562     496    562
#>  4     103 N     gancens  N               572     656    572
#>  5      45 W     filled   W               659     981    659
#>  6      73 W     journals W               538    1505    538
#>  7      24 W     apache   W               626     546    626
#>  8      11 W     flake    W               566     717    566
#>  9      32 W     reliefs  W               922    1471    922
#> 10      96 N     sarves   N               555     806    555
#> # ... with 990 more rows
```

Selecting observations by indices with `slice`

We use `slice` to select observations by their indices. For example, to select rows 10, 20, 50, 100, 500, we would simply do the following.

```
slice(blp, c(10, 20, 50, 100, 500))
#> # A tibble: 5 x 7
#>   participant lex   spell  resp     rt prev.rt rt.raw
#>         <dbl> <chr> <chr>  <chr> <dbl>   <dbl>  <dbl>
#> 1          96 N     sarves N       555     806    555
#> 2          46 W     mirage W       778     571    778
#> 3          72 N     gright N       430     675    430
#> 4           3 W     gleam  W       361     370    361
#> 5          92 W     coaxes W       699     990    699
```

Selecting observations by indices with `slice`

Given that, for example, `10:100` would list the integers 10 to 100 inclusive, we can select just these observations as follows.

```
slice(blp, 10:100)
#> # A tibble: 91 x 7
#>    participant lex   spell     resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>     <chr> <dbl>   <dbl>  <dbl>
#>  1          96 N     sarves    N       555     806    555
#>  2          82 W     deceits   W       657     728    657
#>  3          37 W     nothings  N        NA     552    712
#>  4          52 N     chuespies N       427     539    427
#>  5          96 N     mowny     N      1352    1020   1352
#>  6          96 N     cranned   N       907     573    907
#>  7          89 N     flud      N       742     834    742
#>  8           3 N     bromble   N       523     502    523
#>  9           7 N     trubbles  N       782     458    782
#> 10          35 N     playfound N       643     663    643
#> # ... with 81 more rows
```

De-selecting observations by indices with `slice`

Just as we did with `select`, we can precede the indices with a minus sign to drop the corresponding observations. Thus, for example, we can drop the first 10 observations as follows.

```
slice(blp, -(1:10))
#> # A tibble: 990 x 7
#>    participant lex   spell     resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>     <chr> <dbl>   <dbl>  <dbl>
#>  1          82 W     deceits   W       657     728    657
#>  2          37 W     nothings  N        NA     552    712
#>  3          52 N     chuespies N       427     539    427
#>  4          96 N     mowny     N      1352    1020   1352
#>  5          96 N     cranned   N       907     573    907
#>  6          89 N     flud      N       742     834    742
#>  7           3 N     bromble   N       523     502    523
#>  8           7 N     trubbles  N       782     458    782
#>  9          35 N     playfound N       643     663    643
#> 10          46 W     mirage    W       778     571    778
#> # ... with 980 more rows
```

Selecting observations by condition with `filter`

The `filter` command is a powerful means to filter observations according to their values. For example, we can select all the observations where the `lex` variable is `N` as follows.

```
filter(blp, lex == 'N')
#> # A tibble: 502 x 7
#>    participant lex   spell     resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>     <chr> <dbl>   <dbl>  <dbl>
#>  1          20 N     staud     N       977     511    977
#>  2           9 N     dinbuss   N       565     765    565
#>  3          47 N     snilling  N       562     496    562
#>  4         103 N     gancens   N       572     656    572
#>  5          96 N     sarves    N       555     806    555
#>  6          52 N     chuespies N       427     539    427
#>  7          96 N     mowny     N      1352    1020   1352
#>  8          96 N     cranned   N       907     573    907
#>  9          89 N     flud      N       742     834    742
#> 10           3 N     bromble   N       523     502    523
#> # ... with 492 more rows
```

Selecting observations by condition with `filter`

We can also filter by multiple conditions by listing each one with commas between them. For example, the following gives us the observations where `lex` has the value of `N` and `resp` has the value of `W`.

```
filter(blp, lex == 'N', resp=='W')
#> # A tibble: 35 x 7
#>    participant lex   spell    resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl>
#>  1          73 N     bunding  W        NA     978   1279
#>  2          63 N     gallays  W        NA     589    923
#>  3          50 N     droper   W        NA     741    573
#>  4           6 N     flooder  W        NA     524    557
#>  5          73 N     khantum  W        NA     623   1355
#>  6          81 N     seaped   W        NA     765    691
#>  7          43 N     gafers   W        NA     556    812
#>  8         101 N     winchers W        NA     632    852
#>  9          81 N     flaged   W        NA     674    609
#> 10          11 N     frocker  W        NA     653    665
#> # ... with 25 more rows
```

Selecting observations by condition with `filter`

The following gives us those observations where where `lex` has the value of `N` and `resp` has the value of `W` and `rt.raw` is less than or equal to 500.

```
filter(blp, lex == 'N', resp=='W', rt.raw <= 500)
#> # A tibble: 5 x 7
#>   participant lex   spell    resp     rt prev.rt rt.raw
#>         <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl>
#> 1          28 N     cown     W        NA     680    498
#> 2          17 N     beeched  W        NA     450    469
#> 3          29 N     conforn  W        NA     495    497
#> 4          35 N     blear    W        NA     592    461
#> 5          89 N     stumming W        NA     571    442
```

Selecting observations by condition with `filter`

The previous command is equivalent to making a conjunction of conditions using `&` as follows.

```
filter(blp, lex == 'N' & resp=='W' & rt.raw <= 500)
#> # A tibble: 5 x 7
#>   participant lex   spell    resp     rt prev.rt rt.raw
#>         <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl>
#> 1          28 N     cown     W        NA     680    498
#> 2          17 N     beeched  W        NA     450    469
#> 3          29 N     conforn  W        NA     495    497
#> 4          35 N     blear    W        NA     592    461
#> 5          89 N     stumming W        NA     571    442
```

Selecting observations by condition with `filter`

We can make a *disjunction* of conditions for filtering using the logical-or symbol `|`. For example, to filter observation where the `rt.raw` was either less than 500 or greater than 1000, we can do the following.

```
filter(blp, rt.raw < 500 | rt.raw > 1000)
#> # A tibble: 296 x 7
#>    participant lex   spell      resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>      <chr> <dbl>   <dbl>  <dbl>
#>  1          52 N     chuespies  N       427     539    427
#>  2          96 N     mowny      N      1352    1020   1352
#>  3          28 W     stelae     N        NA     678    497
#>  4          85 W     forewarned N        NA     525    350
#>  5          24 W     owl        W       470     535    470
#>  6          97 W     soda       W       436     447    436
#>  7          81 N     fugate     N       425     403    425
#>  8         105 N     pamps      N        NA     884   1494
#>  9          27 W     outgrowth  N        NA     633   1014
#> 10          82 W     kitty      W       431     476    431
#> # ... with 286 more rows
```

Changing variables and values with `mutate`

To create a new variable `is_accurate` that takes the value of `TRUE` whenever `lex` and `resp` have the same value, we can do the following:

```
mutate(blp, acc = lex == resp)
#> # A tibble: 1,000 x 8
#>    participant lex   spell    resp     rt prev.rt rt.raw acc
#>          <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl> <lgl
#>  1          20 N     staud    N       977     511    977 TRUE
#>  2           9 N     dinbuss  N       565     765    565 TRUE
#>  3          47 N     snilling N       562     496    562 TRUE
#>  4         103 N     gancens  N       572     656    572 TRUE
#>  5          45 W     filled   W       659     981    659 TRUE
#>  6          73 W     journals W       538    1505    538 TRUE
#>  7          24 W     apache   W       626     546    626 TRUE
#>  8          11 W     flake    W       566     717    566 TRUE
#>  9          32 W     reliefs  W       922    1471    922 TRUE
#> 10          96 N     sarves   N       555     806    555 TRUE
#> # ... with 990 more rows
```

Changing variables and values with `mutate`

As another example, we can create a new variable that gives the length of the word given by the `spell` variable.

```
mutate(blp, len = str_length(spell))
#> # A tibble: 1,000 x 8
#>    participant lex   spell    resp     rt prev.rt rt.raw   le
#>          <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl> <int
#>  1          20 N     staud    N       977     511    977
#>  2           9 N     dinbuss  N       565     765    565
#>  3          47 N     snilling N       562     496    562
#>  4         103 N     gancens  N       572     656    572
#>  5          45 W     filled   W       659     981    659
#>  6          73 W     journals W       538    1505    538
#>  7          24 W     apache   W       626     546    626
#>  8          11 W     flake    W       566     717    566
#>  9          32 W     reliefs  W       922    1471    922
#> 10          96 N     sarves   N       555     806    555
#> # ... with 990 more rows
```

Changing variables and values with mutate

We can also create multiple new variable at the same time as in the following example.

```
mutate(blp,
       acc = lex == resp,
       fast = rt.raw < mean(rt.raw, na.rm=TRUE))
#> # A tibble: 1,000 x 9
#>    participant lex   spell    resp     rt prev.rt rt.raw acc
#>          <dbl> <chr> <chr>    <chr> <dbl>   <dbl>  <dbl> <lgl
#>  1          20 N     staud    N       977     511    977 TRUE
#>  2           9 N     dinbuss  N       565     765    565 TRUE
#>  3          47 N     snilling N       562     496    562 TRUE
#>  4         103 N     gancens  N       572     656    572 TRUE
#>  5          45 W     filled   W       659     981    659 TRUE
#>  6          73 W     journals W       538    1505    538 TRUE
#>  7          24 W     apache   W       626     546    626 TRUE
#>  8          11 W     flake    W       566     717    566 TRUE
#>  9          32 W     reliefs  W       922    1471    922 TRUE
#> 10          96 N     sarves   N       555     806    555 TRUE
#> # ... with 990 more rows
```

Sorting observations with `arrange`

Sorting observations in a data frame is easily accomplished with `arrange`. For example to sort by `participant` and then by `spell`, we would do the following.

```
arrange(blp, participant, spell)
#> # A tibble: 1,000 x 7
#>    participant lex   spell     resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>     <chr> <dbl>   <dbl>  <dbl>
#>  1           1 W     abyss     W       629     683    629
#>  2           1 N     baisees   N       524     574    524
#>  3           1 W     carport   W       779     605    779
#>  4           1 N     cellies   N       792     652    792
#>  5           1 W     chafing   W       601     720    601
#>  6           1 N     dametails N       694     635    694
#>  7           1 N     foother   N       789     566    789
#>  8           1 W     gantries  W       644     581    644
#>  9           1 N     hogtush   N       679     568    679
#> 10           1 N     lisedess  N       679     619    679
#> # ... with 990 more rows
```

Sorting observations with `arrange`

We can sort by the reverse order of any variable by using the `desc` command on the variable. In the following example, we sort by `participant`, and then by `spell` in reverse order.

```
arrange(blp, participant, desc(spell))
#> # A tibble: 1,000 x 7
#>    participant lex   spell     resp     rt prev.rt rt.raw
#>          <dbl> <chr> <chr>     <chr> <dbl>   <dbl>  <dbl>
#>  1           1 N     wintes    N       545     629    545
#>  2           1 N     treeps    N       607     610    607
#>  3           1 W     squashes  W       494     491    494
#>  4           1 N     sinkhicks N       536     519    536
#>  5           1 W     shafting  W       553     571    553
#>  6           1 W     month     W       500     498    500
#>  7           1 N     lisedess  N       679     619    679
#>  8           1 N     hogtush   N       679     568    679
#>  9           1 W     gantries  W       644     581    644
#> 10           1 N     foother   N       789     566    789
#> # ... with 990 more rows
```

Reducing data with `summary`

The `dplyr` package has a function `summarize` (or, equivalently, `summarise`) that applies summarizing functions to variables.

For example, we may calculate some summary statistics of the particular variables as in the following example.

```
summarize(blp,
          mean_rt = mean(rt, na.rm = T),
          median_rt = median(rt, na.rm = T),
          sd_rt.raw = sd(rt.raw, na.rm = T)
)
#> # A tibble: 1 x 3
#>   mean_rt median_rt sd_rt.raw
#>     <dbl>     <dbl>     <dbl>
#> 1    638.       588      474.
```

(Note that here it is necessary to use `na.rm = T` to remove the `NA` values in the variables.)

Reducing data with `summary` and `group_by`

The `summarize` command, and its variants, become considerably more powerful when combined with the `group_by` command. Effectively, `group_by` groups the observations within a data frame according to the values of specified variables. For example, the following command groups `blp` into groups of observations according to value of the `lex` variable.

```
blp_by_lex <- group_by(blp, lex)
```

Reducing data with `summary` and `group_by`

If we now apply `summarize` to this grouped data frame, we will obtain summary statistics for each group, as in the following example.

```
summarize(blp_by_lex, mean = mean(rt, na.rm=T))
#> # A tibble: 2 x 2
#>   lex    mean
#>   <chr> <dbl>
#> 1 N      638.
#> 2 W      637.
```

The %>% operator

The `%>%` operator in R is known as the *pipe*. It available from the `magrittr` package, which is part of the `tidyverse`. In RStudio, the keyboard shortcut Ctrl+Shift+M types `%>%`.

To understand pipes, let us begin with a very simple example. The following `primes` variable is a vector of the first 10 prime numbers.

```
primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
```

We can calculate the sum of `primes` as follows.

```
sum(primes)
#> [1] 129
```

We may then calculate the square root of this sum.

```
sqrt(sum(primes))
#> [1] 11.35782
```

We may then calculate the logarithm of this square root.

```
log(sqrt(sum(primes)))
#> [1] 2.429906
```

The %>% operator

The `%>%` is *syntactic sugar* that reexpresses nested functions as sequences.

Returning to some of our examples above, we will see how they can be rewritten with pipes. In each case, we will precede the piped version with a comment showing its original version.

```
# sum(primes)
primes %>% sum()
#> [1] 129
```

```
# sum(primes, na.rm=T)
primes %>% sum(na.rm=T)
#> [1] 129
```

```
# log(sqrt(sum(primes)))
primes %>% sum() %>% sqrt() %>% log()
#> [1] 2.429906
```

Reshaping with `pivot_longer` and `pivot_wider`

A so-called *tidy* data set is a data set where all rows are observations, all columns are variables, and each variable is a single value.

Consider the following data frame.

```
recall_df <- read_csv('data/repeated_measured_a.csv')
recall_df
#> # A tibble: 5 x 4
#>   Subject   Neg   Neu   Pos
#>   <chr>   <dbl> <dbl> <dbl>
#> 1 Faye       26    12    42
#> 2 Jason      29     8    35
#> 3 Jim        32    15    45
#> 4 Ron        22    10    38
#> 5 Victor     30    13    40
```

In this data frame, for each subject, we have three values, which are their scores on a memory test in three different conditions of an experiment. The conditions are `Neg` (negative), `Neu` (neutral), `Pos` (positive). However, each column is not a variable. The `Neg`, `Neu`, `Pos` are, in fact, *values* of a variable, namely the condition of the experiment.

Reshaping with `pivot_longer`

To tidy this data frame, we need a variable for the subject, another for the experiment's condition, and another for the memory score for the corresponding subject in the corresponding condition. To do so, we perform what is sometimes known as a *wide to long* transformation. The `tidyr` package has a function `pivot_longer` for this transformation.

To use `pivot_longer`, we must specify the variables (using the `cols` argument) that we want to pivot from wide to long. Next, we must provide a name for the column that will indicate the experimental condition. Finally, we must provide a name for the column that will indicate the memory scores.

Reshaping with `pivot_longer`

Here is the necessary code:

```
recall_long <- pivot_longer(
  recall_df, cols = -Subject, names_to = 'condition',
  values_to = 'score')
recall_long
#> # A tibble: 15 x 3
#>    Subject condition score
#>    <chr>   <chr>     <dbl>
#>  1 Faye    Neg          26
#>  2 Faye    Neu          12
#>  3 Faye    Pos          42
#>  4 Jason   Neg          29
#>  5 Jason   Neu           8
#>  6 Jason   Pos          35
#>  7 Jim     Neg          32
#>  8 Jim     Neu          15
#>  9 Jim     Pos          45
#> 10 Ron     Neg          22
#> 11 Ron     Neu          10
#> 12 Ron     Pos          38
```

Reshaping with `pivot_wider`

The inverse of a `pivot_longer` is a `pivot_wider`. It is very similar to `pivot_longer` and we use `names_from` and `values_from` in the opposite sense to `names_to` and `values_to`.

```
pivot_wider(recall_long,
            names_from = 'condition', values_from = 'score')
#> # A tibble: 5 x 4
#>   Subject   Neg   Neu   Pos
#>   <chr>   <dbl> <dbl> <dbl>
#> 1 Faye       26    12    42
#> 2 Jason      29     8    35
#> 3 Jim        32    15    45
#> 4 Ron        22    10    38
#> 5 Victor     30    13    40
```