gcplyr-workflow

Mike Blazanin

Contents

Getting started

`gcplyr` is a package that implements a number of functions to make it easier to import, manipulate, and analyze bacterial growth from data collected in multiwell plate readers ("growth curves"). This document gives a walkthrough of how to use `gcplyr`'s most common functions.

To get started, all you need is the data file with the growth curves measures saved in a tabular format (.csv, .xls, or .xlsx) to your computer.

Users often want to combine their data with some information on experimental design elements of their growth curves plate(s). For instance, this might include which strains went into which wells. You can save

this information into a tabular file as well, or you can just keep it handy to enter it directly through a function later on.

Let's get started by loading `gcplyr`

```
library(gcplyr)
```

Data layouts

Growth curves data and design elements can be organized in one of three different tabular layouts: block-shaped, wide-shaped, and tidy-shaped, described below.

Tidy-shaped data is the best layout for analyses, but most plate readers output block-shaped or wide-shaped data, and most user-created design files will be block-shaped. Thus, `gcplyr` works by reshaping block-shaped into wide-shaped data, and wide-shaped data into tidy-shaped data, then running any analyses.

So, what are these three data layouts, and how can you tell which of them your data is in?

Block-shaped

In block-shaped data, the organization of the data corresponds directly with the layout of the physical multi-well plate it was generated from. For instance, a data point from the third row and fourth column of the `data.frame` will be from the well in the third row and fourth column in the physical plate. Because of this, a timeseries of growth curve data that is block-shaped will consist of many separate block-shaped data.frames, each corresponding to a single timepoint.

For example, here is a block-shaped `data.frame` of a 96-well plate (with "..." indicating Columns 4 - 10, not shown). In this example, all the data shown would be from a single timepoint.

	Column 1	Column 2	Column 3	...	Column 11	Column 12
Row A	0.060	0.083	0.086	...	0.082	0.085
Row B	0.099	0.069	0.065	...	0.066	0.078
Row C	0.081	0.071	0.070	...	0.064	0.084
Row D	0.094	0.075	0.065	...	0.067	0.087
Row E	0.052	0.054	0.072	...	0.079	0.065
Row F	0.087	0.095	0.091	...	0.075	0.058
Row G	0.095	0.079	0.099	...	0.063	0.075
Row H	0.056	0.069	0.070	...	0.053	0.078

Wide-shaped

In wide-shaped data, each column of the dataframe corresponds to a single well from the plate, and each row of the dataframe corresponds to a single timepoint. Typically, headers contain the well names.

For example, here is a wide-shaped dataframe of a 96-well plate (here, "..." indicates the 91 columns A4 - H10, not shown). Each row of this dataframe corresponds to a single timepoint.

Time	A1	A2	A3	...	H11	H12
0	0.060	0.083	0.086	...	0.053	0.078
1	0.012	0.166	0.172	...	0.106	0.156
2	0.024	0.332	0.344	...	0.212	0.312

Time	A1	A2	A3	...	H11	H12
3	0.048	0.664	0.688	...	0.424	0.624
4	0.096	1.128	0.976	...	0.848	1.148
5	0.162	1.256	1.152	...	1.096	1.296
6	0.181	1.292	1.204	...	1.192	1.352
7	0.197	1.324	1.288	...	1.234	1.394

Tidy-shaped

In tidy-shaped data, there is a single column that contains all the plate reader measurements, with each unique measurement having its own row. Additional columns specify the timepoint, which well the data comes from, and any other design elements.

Note that, in tidy-shaped data, the number of rows equals the number of wells times the number of timepoints. For instance, with a 96 well plate and 100 timepoints, that will be 9600 rows. (Yes, that's a lot of rows! But don't worry, tidy-shaped data is the best format for downstream analyses.) Tidy-shaped data is common in a number of R packages, including ggplot where it's sometimes called a "long" format. If you want to read more about tidy-shaped data and why it's ideal for analyses, see: Wickham, Hadley. Tidy data. The Journal of Statistical Software, vol. 59, 2014. for more details.

Timepoint	Well	Measurement
1	A1	0.060
1	A2	0.083
1	A3	0.086
...	...	...
7	H10	1.113
7	H11	1.234
7	H12	1.394

Importing data

Once you've determined what format your data is in, you can begin importing it using the `read_` functions of `gcplyr`. If your data is block-shaped, you'll use `read_blocks`. If your data is wide-shaped, you'll use `read_wides`. In the unlikely event your data is already tidy, you can simply read it using the built-in R function `read.table`.

Importing block-shaped data

To import block-shaped data, use the `read_blocks` function. `read_blocks` only requires a list of filenames (or relative file paths) and will return a list of data.frames, with each `data.frame` corresponding to a single block.

The simplest example

Here's a simple example. First, we need to create a series of example block-shaped .csv files. **Don't worry how this code works**. When working with real growth curves data, these files would be output by the plate reader. All you need to do is put the file names in R in a vector, here we've stored the file names in `temp_filenames`.

```
#This code just creates a series of block-shaped example files
#Don't worry about how it works - when working with real growth
#curves data, all these files would be created by the plate reader
temp_filenames <-
  tempfile(pattern = paste(as.character(example_widedata$Time), "_", sep = ""),
                           fileext = ".csv")
for (i in 1:length(temp_filenames)) {
  write.table(
    cbind(matrix(c("", "A", "B", "C", "D", "E", "F", "G", "H"), nrow = 9),
          rbind(
            matrix(1:12, ncol = 12),
            matrix(
                (example_widedata[i, 2:ncol(example_widedata)]/(5*10**8)),
              ncol = 12)
            )
          ),
    file = temp_filenames[i], quote = FALSE, row.names = FALSE, sep = ",",
    col.names = FALSE)
}
```

Here's what one of the files looks like (where the values are absorbance/optical density):

```
print_df(read.csv(temp_filenames[1], header = FALSE))
#>   1 2 3 4 5 6 7 8 9 10 11 12
#> A 0 0 0 0 0 0 0 0 0  0  0  0
#> B 0 0 0 0 0 0 0 0 0  0  0  0
#> C 0 0 0 0 0 0 0 0 0  0  0  0
#> D 0 0 0 0 0 0 0 0 0  0  0  0
#> E 0 0 0 0 0 0 0 0 0  0  0  0
#> F 0 0 0 0 0 0 0 0 0  0  0  0
#> G 0 0 0 0 0 0 0 0 0  0  0  0
#> H 0 0 0 0 0 0 0 0 0  0  0  0
```

This would correspond to all the reads for a single plate taken at the very first timepoint. We can see that the first row contains column headers, and the first column contains row names. Moreover, we can see that at this timepoint the wells on the left-hand-side of the plate have a different density than on the right-hand-side.

If we want to read these files into R, we simply provide `read_blocks` with the vector of file names.

```
imported_blockdata <- read_blocks(files = temp_filenames)
```

Specifying the location of your block-shaped data

However, running `read_blocks` with only the filenames only works if the data in your block-shaped files starts in the first row and column (or has column names in the first row and/or rownames in the first column). If your data starts elsewhere, `read_blocks` needs to know what row/column to start reading on (if your data isn't the last thing in the file, `read_blocks` also needs to know where your data ends).

To show how this works, first let's create some example files where the data doesn't begin in the first row/column. In these example files, the plate reader saved the time that each plate was read in the 2nd row of the file, and started saving the data itself with a header in the 4th row.

Again, **don't worry how this code works**. When working with real growth curves data, these files would be output by the plate reader. All you need to do is put the file names in R in a vector, here we've stored the file names in `temp_filenames2`.

```
#This code just creates a series of block-shaped example files
#Don't worry about how it works - when working with real growth
#curves data, all these files would be created by the plate reader
temp_filenames2 <-
  tempfile(pattern = paste(as.character(example_widedata$Time), "_2_", sep = ""),
                           fileext = ".csv")
for (i in 1:length(temp_filenames2)) {
  write.table(
    cbind(
      matrix(c("", "", "", "", "A", "B", "C", "D", "E", "F", "G", "H"),
             nrow = 12),
      rbind(
        rep("", 12),
        matrix(c("Time", example_widedata$Time[i], rep("", 10)), ncol = 12),
        rep("", 12),
        matrix(1:12, ncol = 12),
        matrix(
              (example_widedata[i, 2:ncol(example_widedata)]/(5*10**8)),
                ncol = 12)
      )
    ),
    file = temp_filenames2[i], quote = FALSE, row.names = FALSE, sep = ",",
    col.names = FALSE)
}
```

Let's take a look at one of the files:

```
print_df(read.csv(temp_filenames2[1], header = FALSE))
#>        NA NA NA NA NA NA NA NA NA NA NA
#>   Time  0 NA NA NA NA NA NA NA NA NA NA
#>        NA NA NA NA NA NA NA NA NA NA NA
#>      1  2  3  4  5  6  7  8  9 10 11 12
#> A    0  0  0  0  0  0  0  0  0  0  0  0
#> B    0  0  0  0  0  0  0  0  0  0  0  0
#> C    0  0  0  0  0  0  0  0  0  0  0  0
#> D    0  0  0  0  0  0  0  0  0  0  0  0
#> E    0  0  0  0  0  0  0  0  0  0  0  0
#> F    0  0  0  0  0  0  0  0  0  0  0  0
#> G    0  0  0  0  0  0  0  0  0  0  0  0
#> H    0  0  0  0  0  0  0  0  0  0  0  0
```

In the above example, the column names are in row 4 and the rownames are in column 1. To specify that to `read_blocks`, we simply do:

```
#Now let's read it with read_blocks
imported_blockdata <- read_blocks(
  files = temp_filenames,
  startrow = 4, startcol = 1)
```

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns aren't nicely numbered. Instead, they're coded by letter. Rather than have to count by hand what columns

your data starts and ends on, just specify the column by letter and `read_blocks` will translate that to a number for you!

```
#Now let's read it with read_blocks
imported_blockdata <- read_blocks(
  files = temp_filenames,
  startrow = 4, startcol = "A")
```

Additionally, some plate readers might output growth curves data in a block shape but in a single file. For instance, the file may contain the block from lines 1 - 8, then an empty line, then the next block from lines 10 - 17, etc. Since `read_blocks` is vectorized on most of its input arguments, including `startrow`, `startcol`, `endrow`, and `endcol`, such a layout can be specified by passing a vector of startrows and endrows to `read_blocks`:

```
imported_blockdata <- read_blocks(
  files = "example_file.csv",
  startrow = c(1, 10, 19, 28, 37, 46, 55),
  endrow = c(8, 17, 26, 35, 44, 53, 62))
```

Specifying metadata

Sometimes, your input files will have information you want to import that's not included in the main block of data. For instance, with block-shaped data the timepoint is nearly always specified somewhere in the input file. `read_blocks` can include that information as well via the `metadata` argument.

For example, let's return to our most-recent example files:

```
print_df(read.csv(temp_filenames2[1], header = FALSE))
#>         NA NA NA NA NA NA NA NA NA NA NA
#>   Time   0 NA NA NA NA NA NA NA NA NA NA
#>         NA NA NA NA NA NA NA NA NA NA NA
#>       1  2  3  4  5  6  7  8  9 10 11 12
#> A     0  0  0  0  0  0  0  0  0  0  0  0
#> B     0  0  0  0  0  0  0  0  0  0  0  0
#> C     0  0  0  0  0  0  0  0  0  0  0  0
#> D     0  0  0  0  0  0  0  0  0  0  0  0
#> E     0  0  0  0  0  0  0  0  0  0  0  0
#> F     0  0  0  0  0  0  0  0  0  0  0  0
#> G     0  0  0  0  0  0  0  0  0  0  0  0
#> H     0  0  0  0  0  0  0  0  0  0  0  0
```

In these files, the timepoint information was located in the 2nd row and 3rd column. Here's how we could specify that metadata in our `read_blocks` command:

```
#Reading the blockcurves files with metadata included
imported_blockdata <- read_blocks(
  files = temp_filenames2,
  startrow = 4, startcol = "A",
  metadata = list("time" = c(2, 3)))
```

You can see that the metadata argument must be a list of named vectors. Each vector should have two elements specifying the location of the metadata in the input files: the first element is the row, the second element is the column.

And just like how you can specify `startrow`, `startcol`, etc. with Excel-style lettering, the location of metadata can also be specified with Excel-style lettering.

```
#Reading the blockcurves files with metadata included
# imported_blockdata <- read_blocks(
#   files = temp_filenames2,
#   startrow = 4, startcol = "A",
#   metadata = list("time" = c(2, "C")))
```

Importing wide-shaped data

To import wide-shaped data, use the `read_wides` function. `read_wides` only requires a filename (or vector of filenames, or relative file paths) and will return a `data.frame` (or list of data.frames).

The simplest example

Here's a simple example. First, we need to create an example wide-shaped .csv file. **Don't worry how this code works**. when working with real growth curves data, these files would be output by the plate reader. All you need to do is put the file name(s) in R, here we've stored the file name in `temp_filename`.

```
#This code just creates a wide-shaped example file
#Don't worry about how it works - when working with real growth
#curves data, this file would be created by the plate reader
temp_filename <- paste(tempfile(), ".csv", sep = "")
write.csv(example_widedata, file = temp_filename, row.names = FALSE)
```

Here's what the start of the file looks like (where the values are absorbance/optical density):

```
print_df(head(read.csv(temp_filename, header = FALSE),
              c(10, 4), row.names = FALSE))
#> Time A1    B1    C1
#>    0  0     0     0
#>  900  0     0     0
#> 1800  0     0     0
#> 2700  0     0     0
#> 3600  0     0     0
#> 4500  0 0.001     0
#> 5400  0 0.001     0
#> 6300  0 0.001     0
#> 7200  0 0.001 0.001
```

This would correspond to all the reads for a single plate taken across all timepoints. For instance, we can see that the first column contains the timepoint information, and each subsequent column corresponds to a well in the plate.

If we want to read these files into R, we simply provide `read_wides` with the file name.

```
#Now let's use read_wides to import our wide-shaped data
imported_widedata <- read_wides(files = temp_filename)
```

The resulting `data.frame` looks like this:

```
print_df(head(imported_widedata, c(6, 3)))
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3    0 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3  900 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3 1800 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3 2700 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3 3600 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a458084ec3 4500 0
```

Note that read_wides automatically saves the filename the data was imported from into the first column of the output `data.frame`. This is done to ensure that later on, `data.frames` from multiple plates can be combined without fear of losing the identity of each plate.

Note that if you have multiple files you'd like to read in, you can do so directly with a single `read_wides` command. In this case, read_wides will return a list containing all the data.frames:

```
#If we had multiple wide-shaped data files to import
imported_widedata <- read_wides(files = c(temp_filename, temp_filename))
```

Specifying the location of your wide-shaped data

However, running `read_wides` with only the filename(s) only works if the data in your wide-shaped files starts in the first row and column (or has column names in the first row and/or rownames in the first column). If your data starts elsewhere, `read_wides` needs to know what row/column to start reading on (if your data isn't the last thing in the file, `read_wides` also needs to know where your data ends).

To show how this works, first let's create an example file where the data doesn't begin in the first row/column. In this example file, the plate reader started saving the data itself with a header in the 5th row.

Again, **don't worry how this code works**. When working with real growth curves data, these files would be output by the plate reader. All you need to do is put the file names in R in a vector, here we've stored the file name in `temp_filename2`.

```
#This code just creates a wide-shaped example file where the data doesn't
#start on the first row.
#Don't worry about how it works - when working with real growth
#curves data, this file would be created by the plate reader
temp_filename2 <- tempfile(fileext = ".csv")
temp_example_widedata <- example_widedata
colnames(temp_example_widedata) <- paste("V", 1:ncol(temp_example_widedata),
                                         sep = "")
modified_example_widedata <-
  rbind(
    as.data.frame(matrix("", nrow = 4, ncol = ncol(example_widedata))),
    colnames(example_widedata),
    temp_example_widedata)
modified_example_widedata[1:2, 1:2] <-
  c("Experiment name", "Start date", "Experiment_1", as.character(Sys.Date()))

write.table(modified_example_widedata, file = temp_filename2,
          row.names = FALSE, col.names = FALSE, sep = ",")
```

Let's take a look at the file:

```
#Let's take a peek at what this file looks like
print_df(head(read.csv(temp_filename2, header = FALSE), c(10, 4)))
#> Experiment name Experiment_1
#>      Start date   2022-03-01
#>
#>
#>            Time           A1 B1 C1
#>               0            0  0  0
#>             900            0  0  0
#>            1800            0  0  0
#>            2700            0  0  0
#>            3600            0  0  0
```

Thus, we can see the data header is in row 5, and the data begins in row 6. To specify that to `read_wides`, we simply do (note that `header = TRUE` by default):

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5)
print_df(head(imported_widedata, c(6, 3)))
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623    0 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623  900 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 1800 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 2700 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 3600 0
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 4500 0
```

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns aren't nicely numbered. Instead, they're coded by letter. Rather than have to count by hand what columns your data starts and ends on, just specify the column by letter and `read_wides` will translate that to a number for you! (in this example we don't have to specify a start column, since the data starts in the first column, but we do so just to show this letter-style functionality).

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5, startcol = "A")
```

Specifying metadata

Sometimes, your input files will have information you want to import that's not included in the main block of data. For instance, many readers will output information like the experiment name and date into a header in the file. `read_wides` can include that information as well via the `metadata` argument.

The `metadata` argument should be a list of named vectors. Each vector should be of length 2, with the first entry specifying the row and the second entry specifying the column where the metadata is located.

For example, in our previous example files, the experiment name was located in the 2nd row, 2nd column, and the start date was located in the 3rd row, 2nd column. Here's how we could specify that metadata:

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, 2),
                                                "start_date" = c(2, 2)))
print_df(head(imported_widedata, c(6, 3)))
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
```

```
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
#> C:\Users\mikeb\AppData\Local\Temp\RtmpmSKEhb\file40a43cf64623 Experiment_1 2022-03-01
```

And just like how you can specify `startrow`, `startcol`, etc. with Excel-style lettering, the location of metadata can also be specified with Excel-style lettering.

```
imported_widedata <- read_wides(files = temp_filename2,
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, "B"),
                                                "start_date" = c(2, "B")))
```

Transforming data

Now that you've gotten your data into the R environment, we need to transform it before we can do analyses. To reiterate, this is necessary because most plate readers that generate growth curves data outputs it in block-shaped or wide-shaped files, but tidy-shaped `data.frame`s are the best shape for analyses and required by gcplyr.

You can transform your `data.frame`s using the `trans_\*` functions in `gcplyr`.

Transforming from block-shaped to wide-shaped

If the data you've read into the R environment is block-shaped, you'll need to transform it from block-shaped to wide-shaped, and then wide-shaped to tidy-shaped. For the first step, you'll use `trans_block_to_wide`. All you need to do is provide `trans_block_to_wide` with the R object created by `read_blocks`.

```
imported_blocks_now_wide <- trans_block_to_wide(imported_blockdata)
#> Warning in trans_block_to_wide(imported_blockdata): Inferring nested_metadata to be
#> TRUE
```

Note that `trans_block_to_wide` automatically detected the metadata that `read_blocks` had pulled from our files, and has stored each piece of metadata as a column in our output file.

```
print(head(imported_blocks_now_wide, c(1, 5)), row.names = FALSE)
#>                                                          block_name time A_1 A_2
#>  C:\\Users\\mikeb\\AppData\\Local\\Temp\\RtmpmSKEhb\\0_2_40a479835eb3    0   0   0
#>  A_3
#>    0
```

Now that your block-shaped data has been transformed to wide-shaped data, you can use `trans_wide_to_tidy` (below) to further transform it into the tidy-shaped data we need for our analyses.

Transforming from wide-shaped to tidy-shaped

If the data you've read into the R environment is wide-shaped (or you've gotten wide-shaped data by transforming your originally block-shaped data), you'll transform it to tidy-shaped using `trans_wide_to_tidy`.

First, you need to provide `trans_wide_to_tidy` with the R object created by `read_wides` or by `trans_block_to_wide`. Then, you have to specify either the columns your data (the spectrophotometric measures) are in via `data_cols`, or what columns your non-data (e.g. time and other information) are in via `id_cols`.

```
imported_blocks_now_tidy <- trans_wide_to_tidy(
  wides = imported_blocks_now_wide,
  id_cols = c("block_name", "time"))

imported_wides_now_tidy <- trans_wide_to_tidy(
  wides = imported_widedata,
  id_cols = c("file", "experiment_name", "start_date", "Time"))
```

Including design elements

Often during analysis of growth curves data, we'd like to incorporate information on the experimental design. For example, which bacteria are present in which wells, or which wells have received some treatment. `gcplyr` enables incorporation of design elements in two ways: 1. Design elements can be imported from tidy-shaped files using `read_table` functions and merged with previously-imported data 2. Design elements can be generated programmatically using `make_tidydesign`

Reading design elements from files

Just like spectrophotometric data, design elements that are saved in tidy-shaped tabular data files can be read using the `read_table` function.

Once these design elements have been read into the R environment, you can merge them with your data. See the next section for details.

Generating tidy-shaped design elements programmatically

If you don't have your experimental design information saved in a file, you can directly create such a `data.frame` using the `gcplyr` function `make_tidydesign`. `make_tidydesign` uses the spatial location of design elements in a multiwell plate as input arguments, but outputs a tidy-shaped `data.frame` that can be easily merged with your tidy-shaped data.

An example with a single design

For example, let's imagine a growth curves experiment where a 96 well plate (12 columns and 8 rows) has a different bacterial strain in each row, but the first and last columns and first and last rows were left empty.

Row names	Column 1	Column 2	Column 3	...	Column 11	Column 12
Row A	Blank	Blank	Blank	...	Blank	Blank
Row B	Blank	Strain #1	Strain #1	...	Strain #1	Blank
Row B	Blank	Strain #2	Strain #2	...	Strain #2	Blank
...	...	...	...	...	...	...
Row G	Blank	Strain #5	Strain #5	...	Strain #5	Blank
Row G	Blank	Strain #6	Strain #6	...	Strain #6	Blank
Row H	Blank	Blank	Blank	...	Blank	Blank

To generate a tidy-shaped design dataframe representing this information, we can use `make_tidydesign`:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12,
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "123456",
    FALSE)
)
```

Now, what are each of the things we've specified for our "Bacteria" design component?

Well, make_tidydesign expects give things for each design component: * a vector containing the possible values * a vector containing all the rows these values should be applied to * a vector containing all the columns these values should be applied to * a string of the pattern itself within those rows and columns * a Boolean for whether this pattern should be filled byrow (defaults to TRUE)

So for our example above, we can see: * the possible values are `c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6")` * the rows these values should be applied to are rows `2:7` * the columns these values should be applied to are columns `2:11` * the pattern these values should be filled in by is `"123456"` * and these values should *not* be filled byrow

This entire list is passed with a name (here, "Bacteria"), that will be used as the resulting column header.

What does the resulting `data.frame` look like?

```
head(my_design, 20)
#>    Well Bacteria
#> 1    A1     <NA>
#> 2    A2     <NA>
#> 3    A3     <NA>
#> 4    A4     <NA>
#> 5    A5     <NA>
#> 6    A6     <NA>
#> 7    A7     <NA>
#> 8    A8     <NA>
#> 9    A9     <NA>
#> 10  A10     <NA>
#> 11  A11     <NA>
#> 12  A12     <NA>
#> 13   B1     <NA>
#> 14   B2 Strain 1
#> 15   B3 Strain 1
#> 16   B4 Strain 1
#> 17   B5 Strain 1
#> 18   B6 Strain 1
#> 19   B7 Strain 1
#> 20   B8 Strain 1
```

A few notes on the pattern string

The fourth element of every argument passed to `make_tidydesign` is the string specifying the pattern of values.

Oftentimes, it will be most convenient to simply use single-characters to correspond to the values. This is the default behavior of `make_tidydesign`, which splits the pattern string into individual characters, and then uses those characters to correspond to the indices of the values you provided.

For instance, in our example above, we used the numbers 1 through 6 to correspond to the values `"Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"`.

It's important to **note that the "0" character is reserved for `NA` values.** There is an example of this later.

If you have more than 9 values, you can use letters (uppercase and/or lowercase) and specify to `make_tidydesign` what letter you'd like the indices to start with. By default, the order goes from 1 to 9, then A to Z (uppercase), then a to z (lowercase). For instance, in the previous example, we could have done:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "A",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "ABCDEF",
    FALSE)
)
```

Or we could have done:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "abcdef",
    FALSE)
)
```

Alternatively, you can use a separating character like a comma to delineate your indices. If you are doing so in order to use multicharacter indices (like numbers with more than one digit), all your indices will have to be numeric.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, pattern_split = ",",
  Bacteria = list(
    c("Strain 1", "Strain 2", "Strain 3", "Strain 4", "Strain 5", "Strain 6"),
    2:7,
    2:11,
    "1,2,3,4,5,6",
    FALSE)
)
```

Continuing with the example: multiple designs

Now let's return to our example growth curves experiment. Imagine that now, in addition to having a different bacterial strain in each row, we also have a different media in each column in the plate.

Row names	Column 1	Column 2	Column 3	...	Column 11	Column 12
Row A	Blank	Blank	Blank	...	Blank	Blank
Row B	Blank	Media #1	Media #2	...	Media #10	Blank
...	...	...	...	...	...	...
Row G	Blank	Media #1	Media #2	...	Media #10	Blank
Row H	Blank	Blank	Blank	...	Blank	Blank

We can generate that design by adding an additional argument to our make_tidydesign call.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Bacteria = list(c("Strain 1", "Strain 2", "Strain 3",
                    "Strain 4", "Strain 5", "Strain 6"),
                  2:7,
                  2:11,
                  "abcdef",
                  FALSE),
  Media = list(c("Media 1", "Media 2", "Media 3",
                 "Media 4", "Media 5", "Media 6",
                 "Media 7", "Media 8", "Media 9",
                 "Media 10", "Media 11", "Media 12"),
               2:7,
               2:11,
               "abcdefghij")
  )
head(my_design, 20)
#>    Well Bacteria   Media
#> 1    A1     <NA>    <NA>
#> 2    A2     <NA>    <NA>
#> 3    A3     <NA>    <NA>
#> 4    A4     <NA>    <NA>
#> 5    A5     <NA>    <NA>
#> 6    A6     <NA>    <NA>
#> 7    A7     <NA>    <NA>
#> 8    A8     <NA>    <NA>
#> 9    A9     <NA>    <NA>
#> 10  A10     <NA>    <NA>
#> 11  A11     <NA>    <NA>
#> 12  A12     <NA>    <NA>
#> 13   B1     <NA>    <NA>
#> 14   B2 Strain 1 Media 1
#> 15   B3 Strain 1 Media 2
#> 16   B4 Strain 1 Media 3
#> 17   B5 Strain 1 Media 4
#> 18   B6 Strain 1 Media 5
#> 19   B7 Strain 1 Media 6
#> 20   B8 Strain 1 Media 7
```

Now, imagine after the experiment we discover that Bacterial Strain 4 and Media #6 were contaminated, and we'd like to exclude them from our analyses by marking them as `NA` in the design. We can simply modify our pattern string, placing a 0 anywhere we would like an `NA` to be filled in.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = list(c("Media 1", "Media 2", "Media 3",
                 "Media 4", "Media 5", "Media 6",
                 "Media 7", "Media 8", "Media 9",
                 "Media 10", "Media 11", "Media 12"),
               2:7,
               2:11,
               "abcde0ghij"),
  Bacteria = list(c("Strain 1", "Strain 2", "Strain 3",
                    "Strain 4", "Strain 5", "Strain 6"),
                  2:7,
                  2:11,
                  "abc0ef",
                  FALSE))
head(my_design, 20)
#>    Well   Media Bacteria
#> 1    A1    <NA>     <NA>
#> 2    A2    <NA>     <NA>
#> 3    A3    <NA>     <NA>
#> 4    A4    <NA>     <NA>
#> 5    A5    <NA>     <NA>
#> 6    A6    <NA>     <NA>
#> 7    A7    <NA>     <NA>
#> 8    A8    <NA>     <NA>
#> 9    A9    <NA>     <NA>
#> 10  A10    <NA>     <NA>
#> 11  A11    <NA>     <NA>
#> 12  A12    <NA>     <NA>
#> 13   B1    <NA>     <NA>
#> 14   B2 Media 1 Strain 1
#> 15   B3 Media 2 Strain 1
#> 16   B4 Media 3 Strain 1
#> 17   B5 Media 4 Strain 1
#> 18   B6 Media 5 Strain 1
#> 19   B7    <NA> Strain 1
#> 20   B8 Media 7 Strain 1
```

Note that `make_tidydesign` is not limited to simple alternating patterns. The pattern string specified can be any pattern, which `make_tidydesign` will replicate sufficient times to cover the entire set of listed wells.

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = list(c("Media 1", "Media 2", "Media 3"),
               2:7,
               2:11,
               "aabbbc000abc"),
  Bacteria = list(c("Strain 1", "Strain 2"),
                  2:7,
                  2:11,
                  "abaaabbbab",
                  FALSE))
head(my_design, 20)
```

```
#>    Well   Media Bacteria
#> 1    A1    <NA>     <NA>
#> 2    A2    <NA>     <NA>
#> 3    A3    <NA>     <NA>
#> 4    A4    <NA>     <NA>
#> 5    A5    <NA>     <NA>
#> 6    A6    <NA>     <NA>
#> 7    A7    <NA>     <NA>
#> 8    A8    <NA>     <NA>
#> 9    A9    <NA>     <NA>
#> 10  A10    <NA>     <NA>
#> 11  A11    <NA>     <NA>
#> 12  A12    <NA>     <NA>
#> 13   B1    <NA>     <NA>
#> 14   B2 Media 1 Strain 1
#> 15   B3 Media 1 Strain 2
#> 16   B4 Media 2 Strain 1
#> 17   B5 Media 2 Strain 1
#> 18   B6 Media 2 Strain 1
#> 19   B7 Media 3 Strain 1
#> 20   B8    <NA> Strain 2
```

`gcplyr` also includes an optional helper function for `make_tidydesign` called `make_designpattern`. `make_designpattern` just helps by reminding the user what arguments are necessary for each design and ensuring they're in the correct order. For example, the following produces the same `data.frame` as the above code:

```
my_design <- make_tidydesign(
  nrows = 8, ncols = 12, lookup_tbl_start = "a",
  Media = make_designpattern(
    values = c("Media 1", "Media 2", "Media 3",
               "Media 4", "Media 5", "Media 6",
               "Media 7", "Media 8", "Media 9",
               "Media 10", "Media 11", "Media 12"),
    rows = 2:7, cols = 2:11, pattern = "abcde0ghij"),
  Bacteria = make_designpattern(
    values = c("Strain 1", "Strain 2", "Strain 3",
               "Strain 4", "Strain 5", "Strain 6"),
    rows = 2:7, cols = 2:11, pattern = "abc0ef",
    byrow = FALSE))
head(my_design, 20)
#>    Well   Media Bacteria
#> 1    A1    <NA>     <NA>
#> 2    A2    <NA>     <NA>
#> 3    A3    <NA>     <NA>
#> 4    A4    <NA>     <NA>
#> 5    A5    <NA>     <NA>
#> 6    A6    <NA>     <NA>
#> 7    A7    <NA>     <NA>
#> 8    A8    <NA>     <NA>
#> 9    A9    <NA>     <NA>
#> 10  A10    <NA>     <NA>
#> 11  A11    <NA>     <NA>
#> 12  A12    <NA>     <NA>
```

```
#> 13   B1    <NA>     <NA>
#> 14   B2 Media 1 Strain 1
#> 15   B3 Media 2 Strain 1
#> 16   B4 Media 3 Strain 1
#> 17   B5 Media 4 Strain 1
#> 18   B6 Media 5 Strain 1
#> 19   B7    <NA> Strain 1
#> 20   B8 Media 7 Strain 1
```

Merging spectrophotometric and design data

Once we have both our design and data in the R environment, we can merge them using `merge_dfs`.

For this, we'll use the data in the `example_widedata` dataset that is included with `gcplyr`, and which was the source for our previous examples with `read_blocks` and `read_wides`.

In the `example_widedata` dataset, we have 48 different bacterial strains. The left side of the plate has all 48 strains in a single well each, and the right side of the plate also has all 48 strains in a single well each:

Row names	Column 1	...	Column 6	Column 7	...	Column 12
Row A	Strain #1	...	Strain #6	Strain #1	...	Strain #6
Row B	Strain #7	...	Strain #12	Strain #7	...	Strain #12
...	...	...	...	...	...	...
Row G	Strain #37	...	Strain #42	Strain #37	...	Strain #42
Row H	Strain #43	...	Strain #48	Strain #43	...	Strain #48

Then, on the right hand side of the plate a phage was also inoculated (while the left hand side remained bacteria-only):

Row names	Column 1	...	Column 6	Column 7	...	Column 12
Row A	No Phage	...	No Phage	Phage Added	...	Phage Added
Row B	No Phage	...	No Phage	Phage Added	...	Phage Added
...	...	...	...	...	...	...
Row G	No Phage	...	No Phage	Phage Added	...	Phage Added
Row H	No Phage	...	No Phage	Phage Added	...	Phage Added

Let's generate our design:

```
example_design <- make_tidydesign(
  pattern_split = ",", nrows = 8, ncols = 12,
  "Bacteria_strain" = make_designpattern(
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 1:6,
    pattern = paste(1:48, collapse = ","),
    byrow = TRUE),
  "Bacteria_strain" = make_designpattern(
    values = paste("Strain", 1:48),
    rows = 1:8, cols = 7:12,
    pattern = paste(1:48, collapse = ","),
    byrow = TRUE),
```

```
  "Phage" = make_designpattern(
    values = c("No Phage"),
    rows = 1:8, cols = 1:6,
    pattern = "1"),
  "Phage" = make_designpattern(
    values = c("Phage Added"),
    rows = 1:8, cols = 7:12,
    pattern = "1"))
```

Now let's transform the `example_widedata` to tidy-shaped.

```
example_tidydata <- trans_wide_to_tidy(example_widedata,
                                       id_cols = "Time")
```

And finally, we merge the two using `merge_dfs`:

```
example_data_and_designs <-
  merge_dfs(example_tidydata,
            example_design)
#> Joining, by = "Well"

print(example_data_and_designs)
#>     Time Well Measurements Bacteria_strain       Phage
#> 1      0   A1            0        Strain 1    No Phage
#> 2      0   B1            0        Strain 7    No Phage
#> 3      0   C1            0       Strain 13    No Phage
#> 4      0   D1            0       Strain 19    No Phage
#> 5      0   E1            0       Strain 25    No Phage
#> 6      0   F1            0       Strain 31    No Phage
#> 7      0   G1            0       Strain 37    No Phage
#> 8      0   H1            0       Strain 43    No Phage
#> 9      0   A2            0        Strain 2    No Phage
#> 10     0   B2            0        Strain 8    No Phage
#> 11     0   C2            0       Strain 14    No Phage
#> 12     0   D2            0       Strain 20    No Phage
#> 13     0   E2            0       Strain 26    No Phage
#> 14     0   F2            0       Strain 32    No Phage
#> 15     0   G2            0       Strain 38    No Phage
#> 16     0   H2            0       Strain 44    No Phage
#> 17     0   A3            0        Strain 3    No Phage
#> 18     0   B3            0        Strain 9    No Phage
#> 19     0   C3            0       Strain 15    No Phage
#> 20     0   D3            0       Strain 21    No Phage
#> 21     0   E3            0       Strain 27    No Phage
#> 22     0   F3            0       Strain 33    No Phage
#> 23     0   G3            0       Strain 39    No Phage
#> 24     0   H3            0       Strain 45    No Phage
#> 25     0   A4            0        Strain 4    No Phage
#> 26     0   B4            0       Strain 10    No Phage
#> 27     0   C4            0       Strain 16    No Phage
#> 28     0   D4            0       Strain 22    No Phage
#> 29     0   E4            0       Strain 28    No Phage
#> 30     0   F4            0       Strain 34    No Phage
```

```
#> 31     0   G4            0       Strain 40    No Phage
#> 32     0   H4            0       Strain 46    No Phage
#> 33     0   A5            0        Strain 5    No Phage
#> 34     0   B5            0       Strain 11    No Phage
#> 35     0   C5            0       Strain 17    No Phage
#> 36     0   D5            0       Strain 23    No Phage
#> 37     0   E5            0       Strain 29    No Phage
#> 38     0   F5            0       Strain 35    No Phage
#> 39     0   G5            0       Strain 41    No Phage
#> 40     0   H5            0       Strain 47    No Phage
#> 41     0   A6            0        Strain 6    No Phage
#> 42     0   B6            0       Strain 12    No Phage
#> 43     0   C6            0       Strain 18    No Phage
#> 44     0   D6            0       Strain 24    No Phage
#> 45     0   E6            0       Strain 30    No Phage
#> 46     0   F6            0       Strain 36    No Phage
#> 47     0   G6            0       Strain 42    No Phage
#> 48     0   H6            0       Strain 48    No Phage
#> 49     0   A7            0        Strain 1 Phage Added
#> 50     0   B7            0        Strain 7 Phage Added
#> 51     0   C7            0       Strain 13 Phage Added
#> 52     0   D7            0       Strain 19 Phage Added
#> 53     0   E7            0       Strain 25 Phage Added
#> 54     0   F7            0       Strain 31 Phage Added
#> 55     0   G7            0       Strain 37 Phage Added
#> 56     0   H7            0       Strain 43 Phage Added
#> 57     0   A8            0        Strain 2 Phage Added
#> 58     0   B8            0        Strain 8 Phage Added
#> 59     0   C8            0       Strain 14 Phage Added
#> 60     0   D8            0       Strain 20 Phage Added
#> 61     0   E8            0       Strain 26 Phage Added
#> 62     0   F8            0       Strain 32 Phage Added
#> 63     0   G8            0       Strain 38 Phage Added
#> 64     0   H8            0       Strain 44 Phage Added
#> 65     0   A9            0        Strain 3 Phage Added
#> 66     0   B9            0        Strain 9 Phage Added
#> 67     0   C9            0       Strain 15 Phage Added
#> 68     0   D9            0       Strain 21 Phage Added
#> 69     0   E9            0       Strain 27 Phage Added
#> 70     0   F9            0       Strain 33 Phage Added
#> 71     0   G9            0       Strain 39 Phage Added
#> 72     0   H9            0       Strain 45 Phage Added
#> 73     0  A10            0        Strain 4 Phage Added
#> 74     0  B10            0       Strain 10 Phage Added
#> 75     0  C10            0       Strain 16 Phage Added
#> 76     0  D10            0       Strain 22 Phage Added
#> 77     0  E10            0       Strain 28 Phage Added
#> 78     0  F10            0       Strain 34 Phage Added
#> 79     0  G10            0       Strain 40 Phage Added
#> 80     0  H10            0       Strain 46 Phage Added
#> 81     0  A11            0        Strain 5 Phage Added
#> 82     0  B11            0       Strain 11 Phage Added
#> 83     0  C11            0       Strain 17 Phage Added
```

```
#> 84     0  D11            0       Strain 23 Phage Added
#> 85     0  E11            0       Strain 29 Phage Added
#> 86     0  F11            0       Strain 35 Phage Added
#> 87     0  G11            0       Strain 41 Phage Added
#> 88     0  H11            0       Strain 47 Phage Added
#> 89     0  A12            0        Strain 6 Phage Added
#> 90     0  B12            0       Strain 12 Phage Added
#> 91     0  C12            0       Strain 18 Phage Added
#> 92     0  D12            0       Strain 24 Phage Added
#> 93     0  E12            0       Strain 30 Phage Added
#> 94     0  F12            0       Strain 36 Phage Added
#> 95     0  G12            0       Strain 42 Phage Added
#> 96     0  H12            0       Strain 48 Phage Added
#> 97   900   A1            0        Strain 1    No Phage
#> 98   900   B1            0        Strain 7    No Phage
#> 99   900   C1            0       Strain 13    No Phage
#> 100  900   D1            0       Strain 19    No Phage
#> 101  900   E1            0       Strain 25    No Phage
#> 102  900   F1            0       Strain 31    No Phage
#> 103  900   G1            0       Strain 37    No Phage
#> 104  900   H1            0       Strain 43    No Phage
#> 105  900   A2            0        Strain 2    No Phage
#> 106  900   B2            0        Strain 8    No Phage
#> 107  900   C2            0       Strain 14    No Phage
#> 108  900   D2            0       Strain 20    No Phage
#> 109  900   E2            0       Strain 26    No Phage
#> 110  900   F2            0       Strain 32    No Phage
#> 111  900   G2            0       Strain 38    No Phage
#> 112  900   H2            0       Strain 44    No Phage
#> 113  900   A3            0        Strain 3    No Phage
#> 114  900   B3            0        Strain 9    No Phage
#> 115  900   C3            0       Strain 15    No Phage
#> 116  900   D3            0       Strain 21    No Phage
#> 117  900   E3            0       Strain 27    No Phage
#> 118  900   F3            0       Strain 33    No Phage
#> 119  900   G3            0       Strain 39    No Phage
#> 120  900   H3            0       Strain 45    No Phage
#> 121  900   A4            0        Strain 4    No Phage
#> 122  900   B4            0       Strain 10    No Phage
#> 123  900   C4            0       Strain 16    No Phage
#> 124  900   D4            0       Strain 22    No Phage
#> 125  900   E4            0       Strain 28    No Phage
#> 126  900   F4            0       Strain 34    No Phage
#> 127  900   G4            0       Strain 40    No Phage
#> 128  900   H4            0       Strain 46    No Phage
#> 129  900   A5            0        Strain 5    No Phage
#> 130  900   B5            0       Strain 11    No Phage
#> 131  900   C5            0       Strain 17    No Phage
#> 132  900   D5            0       Strain 23    No Phage
#> 133  900   E5            0       Strain 29    No Phage
#> 134  900   F5            0       Strain 35    No Phage
#> 135  900   G5            0       Strain 41    No Phage
#> 136  900   H5            0       Strain 47    No Phage
```

```
#> 137  900   A6            0        Strain 6    No Phage
#> 138  900   B6            0       Strain 12    No Phage
#> 139  900   C6            0       Strain 18    No Phage
#> 140  900   D6            0       Strain 24    No Phage
#> 141  900   E6            0       Strain 30    No Phage
#> 142  900   F6            0       Strain 36    No Phage
#> 143  900   G6            0       Strain 42    No Phage
#> 144  900   H6            0       Strain 48    No Phage
#> 145  900   A7            0        Strain 1 Phage Added
#> 146  900   B7            0        Strain 7 Phage Added
#> 147  900   C7            0       Strain 13 Phage Added
#> 148  900   D7            0       Strain 19 Phage Added
#> 149  900   E7            0       Strain 25 Phage Added
#> 150  900   F7            0       Strain 31 Phage Added
#> 151  900   G7            0       Strain 37 Phage Added
#> 152  900   H7            0       Strain 43 Phage Added
#> 153  900   A8            0        Strain 2 Phage Added
#> 154  900   B8            0        Strain 8 Phage Added
#> 155  900   C8            0       Strain 14 Phage Added
#> 156  900   D8            0       Strain 20 Phage Added
#> 157  900   E8            0       Strain 26 Phage Added
#> 158  900   F8            0       Strain 32 Phage Added
#> 159  900   G8            0       Strain 38 Phage Added
#> 160  900   H8            0       Strain 44 Phage Added
#> 161  900   A9            0        Strain 3 Phage Added
#> 162  900   B9            0        Strain 9 Phage Added
#> 163  900   C9            0       Strain 15 Phage Added
#> 164  900   D9            0       Strain 21 Phage Added
#> 165  900   E9            0       Strain 27 Phage Added
#> 166  900   F9            0       Strain 33 Phage Added
#> 167  900   G9            0       Strain 39 Phage Added
#> 168  900   H9            0       Strain 45 Phage Added
#> 169  900  A10            0        Strain 4 Phage Added
#> 170  900  B10            0       Strain 10 Phage Added
#> 171  900  C10            0       Strain 16 Phage Added
#> 172  900  D10            0       Strain 22 Phage Added
#> 173  900  E10            0       Strain 28 Phage Added
#> 174  900  F10            0       Strain 34 Phage Added
#> 175  900  G10            0       Strain 40 Phage Added
#> 176  900  H10            0       Strain 46 Phage Added
#> 177  900  A11            0        Strain 5 Phage Added
#> 178  900  B11            0       Strain 11 Phage Added
#> 179  900  C11            0       Strain 17 Phage Added
#> 180  900  D11            0       Strain 23 Phage Added
#> 181  900  E11            0       Strain 29 Phage Added
#> 182  900  F11            0       Strain 35 Phage Added
#> 183  900  G11            0       Strain 41 Phage Added
#> 184  900  H11            0       Strain 47 Phage Added
#> 185  900  A12            0        Strain 6 Phage Added
#> 186  900  B12            0       Strain 12 Phage Added
#> 187  900  C12            0       Strain 18 Phage Added
#> 188  900  D12            0       Strain 24 Phage Added
#> 189  900  E12            0       Strain 30 Phage Added
```

```
#> 190  900  F12            0       Strain 36 Phage Added
#> 191  900  G12            0       Strain 42 Phage Added
#> 192  900  H12            0       Strain 48 Phage Added
#> 193 1800   A1            0        Strain 1    No Phage
#> 194 1800   B1            0        Strain 7    No Phage
#> 195 1800   C1            0       Strain 13    No Phage
#> 196 1800   D1            0       Strain 19    No Phage
#> 197 1800   E1            0       Strain 25    No Phage
#> 198 1800   F1            0       Strain 31    No Phage
#> 199 1800   G1            0       Strain 37    No Phage
#> 200 1800   H1            0       Strain 43    No Phage
#>  [ reached 'max' / getOption("max.print") -- omitted 9112 rows ]
```

Pre-processing data

[further documentation to-be-written]

Analyzing data

[further documentation to-be-written]

Handling multiple plates simultaneously

[further documentation to-be-written]