# Importing and transforming data

Mike Blazanin

## Contents



## Where are we so far?

1. Introduction: `vignette("gc01_gcplyr")`
2. **Importing and transforming data:** `vignette("gc02_import_transform")`
3. Incorporating design information: `vignette("gc03_incorporate_designs")`
4. Pre-processing and plotting your data: `vignette("gc04_preprocess_plot")`
5. Processing your data: `vignette("gc05_process")`
6. Analyzing your data: `vignette("gc06_analyze")`
7. Dealing with noise: `vignette("gc07_noise")`

8. Statistics, merging other data, and other resources: `vignette("gc08_conclusion")`
9. Working with multiple plates: `vignette("gc09_multiple_plates")`

Previously, we gave a quick demonstration of what `gcplyr` can do. Here, we're going into more detail about how to import your data into `R` and and transform it into a better layout.

If you haven't already, load the necessary packages.

```
library(gcplyr)
```

# Data formats and layouts

`gcplyr` was built to easily read all the most-common tabular file formats. While this most explicitly includes formats like csv, xls, and xlsx, in fact `gcplyr` functions for `read`ing or `import`ing can work with *any* file format that `R`'s built-in `read.table` function can handle. If you're working with something besides csv, xls, or xlsx, simply specify the arguments you would need for `read.table` to the relevant `gcplyr` function and it should handle the rest.

Aside from file *formats*, growth curve data and designs can be *organized* in one of three different 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 and wide-shaped data into tidy-shaped data, then running analyses.

So, how can you tell which layout 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. 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 |
| 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. Yes, that's a lot of rows! But tidy-shaped data is the best format for analyses, and is common in a number of `R` packages, including `ggplot`, where it's sometimes called a "long" format.

| 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_*` or `import_*` functions of `gcplyr`.

**If your data is block-shaped:** use `import_blockmeasures` and start in the next section: **Importing block-shaped data**

**If your data is wide-shaped:** use `read_wides` and skip down to the **Importing wide-shaped data** section

**If your data is already tidy-shaped:** use `read_tidys` and skip down to the **Importing tidy-shaped data** section.

# Importing block-shaped data

To import block-shaped data, use the `import_blockmeasures` function. `import_blockmeasures` only requires a list of filenames (or relative file paths) and **will return a wide-shaped data.frame** that you can save in R.

## A basic example

Here's a simple example. First, we need to create a series of example block-shaped `.csv` files. When working with real growth curve data, these files would be output by the plate reader, but we'll generate them with `make_example`. In this case, `make_example` creates the files and returns a vector with the names of the files, which we'll store in `temp_filenames`.

```
temp_filenames <- make_example(vignette = 2, example = 1)
#> Files have been written
```

If you've saved all the files to a single folder, you can easily get a vector with all their names using `list.files`. If your folder contains other files, you can specify a regular expression `pattern` to limit it to just those you want to import:

```
# Here we print all the files we're going to read
list.files(pattern = "Plate1.*csv")
#>  [1] "Plate1-0_00_00.csv"  "Plate1-0_15_00.csv"  "Plate1-0_30_00.csv"  "Plate1-0_45_00.csv"
#>  [5] "Plate1-1_00_00.csv"  "Plate1-1_15_00.csv"  "Plate1-1_30_00.csv"  "Plate1-1_45_00.csv"
#>  [9] "Plate1-10_00_00.csv" "Plate1-10_15_00.csv" "Plate1-10_30_00.csv" "Plate1-10_45_00.csv"
#> [13] "Plate1-11_00_00.csv" "Plate1-11_15_00.csv" "Plate1-11_30_00.csv" "Plate1-11_45_00.csv"
#> [17] "Plate1-12_00_00.csv" "Plate1-12_15_00.csv" "Plate1-12_30_00.csv" "Plate1-12_45_00.csv"
#> [21] "Plate1-13_00_00.csv" "Plate1-13_15_00.csv" "Plate1-13_30_00.csv" "Plate1-13_45_00.csv"
#> [25] "Plate1-14_00_00.csv" "Plate1-14_15_00.csv" "Plate1-14_30_00.csv" "Plate1-14_45_00.csv"
#> [29] "Plate1-15_00_00.csv" "Plate1-15_15_00.csv" "Plate1-15_30_00.csv" "Plate1-15_45_00.csv"
#> [33] "Plate1-16_00_00.csv" "Plate1-16_15_00.csv" "Plate1-16_30_00.csv" "Plate1-16_45_00.csv"
#> [37] "Plate1-17_00_00.csv" "Plate1-17_15_00.csv" "Plate1-17_30_00.csv" "Plate1-17_45_00.csv"
#> [41] "Plate1-18_00_00.csv" "Plate1-18_15_00.csv" "Plate1-18_30_00.csv" "Plate1-18_45_00.csv"
#> [45] "Plate1-19_00_00.csv" "Plate1-19_15_00.csv" "Plate1-19_30_00.csv" "Plate1-19_45_00.csv"
#> [49] "Plate1-2_00_00.csv"  "Plate1-2_15_00.csv"  "Plate1-2_30_00.csv"  "Plate1-2_45_00.csv"
#> [53] "Plate1-20_00_00.csv" "Plate1-20_15_00.csv" "Plate1-20_30_00.csv" "Plate1-20_45_00.csv"
#> [57] "Plate1-21_00_00.csv" "Plate1-21_15_00.csv" "Plate1-21_30_00.csv" "Plate1-21_45_00.csv"
#> [61] "Plate1-22_00_00.csv" "Plate1-22_15_00.csv" "Plate1-22_30_00.csv" "Plate1-22_45_00.csv"
#> [65] "Plate1-23_00_00.csv" "Plate1-23_15_00.csv" "Plate1-23_30_00.csv" "Plate1-23_45_00.csv"
#> [69] "Plate1-24_00_00.csv" "Plate1-3_00_00.csv"  "Plate1-3_15_00.csv"  "Plate1-3_30_00.csv"
#> [73] "Plate1-3_45_00.csv"  "Plate1-4_00_00.csv"  "Plate1-4_15_00.csv"  "Plate1-4_30_00.csv"
#> [77] "Plate1-4_45_00.csv"  "Plate1-5_00_00.csv"  "Plate1-5_15_00.csv"  "Plate1-5_30_00.csv"
#> [81] "Plate1-5_45_00.csv"  "Plate1-6_00_00.csv"  "Plate1-6_15_00.csv"  "Plate1-6_30_00.csv"
#> [85] "Plate1-6_45_00.csv"  "Plate1-7_00_00.csv"  "Plate1-7_15_00.csv"  "Plate1-7_30_00.csv"
#> [89] "Plate1-7_45_00.csv"  "Plate1-8_00_00.csv"  "Plate1-8_15_00.csv"  "Plate1-8_30_00.csv"
#> [93] "Plate1-8_45_00.csv"  "Plate1-9_00_00.csv"  "Plate1-9_15_00.csv"  "Plate1-9_30_00.csv"
#> [97] "Plate1-9_45_00.csv"

# Here we save them to the temp_filenames variable
temp_filenames <- list.files(pattern = "Plate1.*csv")
```

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, colClasses = "character"))
#>
#>    Time     0
#>
#>       1     2     3     4     5     6     7     8     9    10    11    12
#> A 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
```

```
#> B 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> C 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> D 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> E 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> F 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> G 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> H 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
```

This file corresponds to all the reads for a single plate taken at the very first timepoint. We can see that the second row of the file contains some metadata about the timepoint when this plate read read was taken. Then, the data itself starts with column headers on row 4 and rownames in column 1.

If we want to read these files into `R`, we simply provide `import_blockmeasures` with the vector of file names, and save the result to some `R` object (here, `imported_blockdata`). `import_blockmeasures` assumes your data starts on the first row and column, and ends on the last row and column, unless you specify otherwise.

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

head(imported_blockdata, c(6, 8))
#>       block_name    A1    A2    A3    A4    A5    A6    A7
#> 1 Plate1-0_00_00 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 2 Plate1-0_15_00 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 3 Plate1-0_30_00 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 4 Plate1-0_45_00 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 5 Plate1-1_00_00 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 6 Plate1-1_15_00 0.002 0.003 0.002 0.003 0.003 0.002 0.002
```

Here we can see that `import_blockmeasures` has created a wide-shaped R object containing the data from all of our reads. It has also added the file names under the `block_name` column, so that we can easily track which row came from which file.

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns are coded by letter. `import_blockmeasures` allows you to specify the column by letter too.

```
# We can specify rows or columns by Excel-style letters too
imported_blockdata <- import_blockmeasures(
  files = temp_filenames,
  startrow = 4, 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, with block-shaped data the timepoint is nearly always specified somewhere in the input file. `import_blockmeasures` 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_filenames[1], header = FALSE, colClasses = "character"))
#>
#>    Time     0
#>
```

```
#>       1     2     3     4     5     6     7     8     9    10    11    12
#> A 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> B 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> C 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> D 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> E 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> F 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> G 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> H 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
```

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 `import_blockmeasures` command:

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

head(imported_blockdata, c(6, 8))
#>       block_name time    A1    A2    A3    A4    A5    A6
#> 1 Plate1-0_00_00    0 0.002 0.002 0.002 0.002 0.002 0.002
#> 2 Plate1-0_15_00  900 0.002 0.002 0.002 0.002 0.002 0.002
#> 3 Plate1-0_30_00 1800 0.002 0.002 0.002 0.002 0.002 0.002
#> 4 Plate1-0_45_00 2700 0.002 0.002 0.002 0.002 0.002 0.002
#> 5 Plate1-1_00_00 3600 0.002 0.002 0.002 0.002 0.002 0.002
#> 6 Plate1-1_15_00 4500 0.002 0.003 0.002 0.003 0.003 0.002
```

You can see that the metadata you specified has been added as a column in our output `data.frame`. When specifying metadata, 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.

You can also specify the location of metadata with Excel-style lettering.

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

## Reading multiple blocks from a single file

`import_blockmeasures` can also import multiple blocks from a single file, which some plate readers may output. In this case, you simply have to specify a vector of rows and columns that define the location of each block within the file.

First, let's create an example file (normally this file would be created by the plate reader).

```
make_example(vignette = 2, example = 2)
#> Files have been written
#> [1] "./blocks_single.csv"
```

Let's take a look at what the file looks like:

```
print_df(head(read.csv("blocks_single.csv", header = FALSE,
                       colClasses = "character"),
              c(20, 8)))
#> block_name Plate1-0_00_00
#>       time              0
#>                         1     2     3     4     5     6     7
#>          A          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          B          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          C          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          D          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          E          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          F          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          G          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          H          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>
#> block_name Plate1-0_15_00
#>       time            900
#>                         1     2     3     4     5     6     7
#>          A          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          B          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          C          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          D          0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>          E          0.002 0.002 0.002 0.002 0.002 0.002 0.002
```

We can see that the first block has some metadata above it, then the block of data itself. After that there's an empty row before the next block starts. In fact, if we look at the whole file, we'll notice that all the blocks go from column 1 ("A" in Excel) to column 13 ("M" in Excel), they start on rows 3, 15, 27, 39, etc, and end on rows 11, 23, 35, 47, etc. When we look in the file, we can also see that the very last block starts on row 1155 and ends on row 1163. Let's read this information in using `import_blockmeasures`:

```
imported_blockdata <- import_blockmeasures(
  "blocks_single.csv",
  startrow = seq(from = 3, to = 1155, by = 12),
  endrow = seq(from = 11, to = 1163, by = 12),
  startcol = 1, endcol = 13)
```

Here we've used the built-in `R` function `seq` to generate the full vector of `startrow`s and `endrow`s. If we take a look, we can see that it's been read successfully:

```
head(imported_blockdata, c(6, 8))
#>      block_name    A1    A2    A3    A4    A5    A6    A7
#> 1 blocks_single 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 2 blocks_single 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 3 blocks_single 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 4 blocks_single 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 5 blocks_single 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 6 blocks_single 0.002 0.003 0.002 0.003 0.003 0.002 0.002
```

Now let's add some metadata. Because we're reading from a single file, we need to specify the metadata slightly differently. Instead of the metadata being a single vector `c(row,column)` with the location, it's going to be a list of two vectors, one with the row numbers, and one with the column numbers.

Going back to the file, we can see that the time of the block is saved in the second column, in rows 2, 14, 26, 38, ... through 1154.

```
imported_blockdata <- import_blockmeasures(
  "blocks_single.csv",
  startrow = seq(from = 3, to = 1155, by = 12),
  endrow = seq(from = 11, to = 1163, by = 12),
  startcol = 1, endcol = 13,
  metadata = list("time" = list(seq(from = 2, to = 1154, by = 12), 2)))
```

And now if we take a look at the resulting object, we can see that the time metadata has been incorporated.

```
head(imported_blockdata, c(6, 8))
#>      block_name time    A1    A2    A3    A4    A5    A6
#> 1 blocks_single    0 0.002 0.002 0.002 0.002 0.002 0.002
#> 2 blocks_single  900 0.002 0.002 0.002 0.002 0.002 0.002
#> 3 blocks_single 1800 0.002 0.002 0.002 0.002 0.002 0.002
#> 4 blocks_single 2700 0.002 0.002 0.002 0.002 0.002 0.002
#> 5 blocks_single 3600 0.002 0.002 0.002 0.002 0.002 0.002
#> 6 blocks_single 4500 0.002 0.003 0.002 0.003 0.003 0.002
```

### What to do next

Now that you've imported your block-shaped data, you'll need to transform it for later analyses. Jump directly to the **Transforming data** section.

## 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) that you can save in R.

### A basic 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 curve data, these files would be output by the plate reader. All you need to do is know the file name(s) to put in you `R` code. In this example, the file name is `widedata.csv`.

```
make_example(vignette = 2, example = 3)
#> Files have been written
#> [1] "./widedata.csv"
```

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

```
# Let's take a peek at what this file looks like
print_df(head(read.csv("widedata.csv", header = FALSE,
                       colClasses = "character"),
              c(10, 10)))
```

```
#> Experiment name Experiment_1
#>      Start date   2023-11-03
#>
#>
#>            Time           A1    B1    C1    D1    E1    F1    G1    H1    A2
#>               0        0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>             900        0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>            1800        0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>            2700        0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#>            3600        0.002 0.002 0.002 0.003 0.003 0.002 0.002 0.003 0.002
```

This file contains all the reads for a single plate taken across all timepoints. We can see that the first two rows contain some metadata saved by the plate reader, like the name of the experiment and the date of the experiment. Then, we can see that the data starts on the 5th row with a header. The first column contains the timepoint information, and each subsequent column corresponds to a well in the plate.

If we want to read this file into `R`, we simply provide `read_wides` with the file name, and save the result to some `R` object (here, `imported_widedata`). `read_wides` assumes your data starts on the first row and column, and ends on the last row and column, unless you specify otherwise.

```
imported_widedata <- read_wides(files = "widedata.csv", startrow = 5)
```

The resulting `data.frame` looks like this:

```
head(imported_widedata, c(6, 10))
#>         file Time    A1    B1    C1    D1    E1    F1    G1    H1
#> 6  widedata    0 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 7  widedata  900 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 8  widedata 1800 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 9  widedata 2700 0.002 0.002 0.002 0.002 0.002 0.002 0.002 0.002
#> 10 widedata 3600 0.002 0.002 0.002 0.003 0.003 0.002 0.002 0.003
#> 11 widedata 4500 0.002 0.003 0.002 0.003 0.003 0.002 0.003 0.003
```

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.

If you're looking at your data in Excel or a similar spreadsheet program, you'll notice that the columns are coded by letter. `read_wides` allows you to specify the column by letter too.

```
imported_widedata <- read_wides(files = "widedata.csv",
                                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. `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 = "widedata.csv",
                                startrow = 5,
                                metadata = list("experiment_name" = c(1, 2),
                                                "start_date" = c(2, 2)))
head(imported_widedata, c(6, 8))
#>        file experiment_name start_date Time    A1    B1    C1    D1
#> 6  widedata    Experiment_1 2023-11-03    0 0.002 0.002 0.002 0.002
#> 7  widedata    Experiment_1 2023-11-03  900 0.002 0.002 0.002 0.002
#> 8  widedata    Experiment_1 2023-11-03 1800 0.002 0.002 0.002 0.002
#> 9  widedata    Experiment_1 2023-11-03 2700 0.002 0.002 0.002 0.002
#> 10 widedata    Experiment_1 2023-11-03 3600 0.002 0.002 0.002 0.003
#> 11 widedata    Experiment_1 2023-11-03 4500 0.002 0.003 0.002 0.003
```

You can also specify the location of metadata with Excel-style lettering.

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

### What to do next

Now that you've imported your wide-shaped data, you'll need to transform it for later analyses. Continue on to the **Transforming data** section.

## Importing tidy-shaped data

To import tidy-shaped data, you could use the built-in `R` functions like `read.table`. However, if you need a few more options, you can use the `gcplyr` function `read_tidys`. Unlike the built-in option, `read_tidys` can import multiple tidy-shaped files at once, can add the filename as a column in the resulting data.frame, and can handle files where the tidy-shaped information doesn't start on the first row and column.

`read_tidys` only requires a filename (or vector of filenames, or relative file paths) and will return a `data.frame` (or list of data.frames) that you can save in R.

If you've read in your tidy-shaped data, you won't need to transform it, so you can skip down to the **What's next?** section.

## Transforming data

Now that you've gotten your data into `R`, we need to transform it before we can do analyses. To reiterate, this is necessary because most plate readers that generate growth curve 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 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 `import_blockmeasures` or `read_wides`.

Then, you have to specify one of:

- the columns your data (the spectrophotometric measures) are in via `data_cols`
- 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_blockdata,
  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"))

print(head(imported_blocks_now_tidy), row.names = FALSE)
#>     block_name time Well Measurements
#>  blocks_single    0   A1        0.002
#>  blocks_single    0   A2        0.002
#>  blocks_single    0   A3        0.002
#>  blocks_single    0   A4        0.002
#>  blocks_single    0   A5        0.002
#>  blocks_single    0   A6        0.002
```

## What's next?

Now that you've imported and transformed your data to be tidy-shaped, you likely want to incorporate some design information on what went into each well (and plate). Alternatively, if you'd like to skip that step for now, you can go directly to pre-processing and plotting your data

1. Introduction: `vignette("gc01_gcplyr")`
2. Importing and transforming data: `vignette("gc02_import_transform")`
3. **Incorporating design information: `vignette("gc03_incorporate_designs")`**
4. **Pre-processing and plotting your data: `vignette("gc04_preprocess_plot")`**
5. Processing your data: `vignette("gc05_process")`
6. Analyzing your data: `vignette("gc06_analyze")`
7. Dealing with noise: `vignette("gc07_noise")`
8. Statistics, merging other data, and other resources: `vignette("gc08_conclusion")`
9. Working with multiple plates: `vignette("gc09_multiple_plates")`