wellmap

Release 3.4.0

unknown

Jul 26, 2022

GETTING STARTED

Many medium-throughput experiments produce data in 24-, 96-, or 384-well plate format. However, it can be a challenge to keep track of which wells (e.g. A1, B2, etc.) correspond to which experimental conditions (e.g. genotype, drug concentration, replicate number, etc.) for large numbers of experiments. It can also be a challenge to write analysis scripts flexible enough to handle the different plate layouts that will inevitably come up as more and more experiments are run.

The *wellmap* package solves these challenges by introducing a TOML-based file format that succinctly describes the organization of wells on plates. The file format is designed to be human-readable and -writable, so it can serve as a standalone digital record. The file format can also be easily parsed in python and R to help write analysis scripts that will work regardless of how you (or your collaborators) organize wells on your plates.

CHAPTER

ONE

CITATION

Kundert, K. Wellmap: a file format for microplate layouts. *BMC Res Notes* **14**, 164 (2021). https://doi.org/10.1186/s13104-021-05573-0

1.1 Basic usage with python

The following steps show how to get started with *wellmap* in python:

1. Install *wellmap* from PyPI. Note that python3.6 is required:

```
$ pip install wellmap
```

2. Write a *TOML file* describing the layout of an experiment. For example, the following layout might be used for a standard curve:

Listing 1: `std_curve.toml`

```
# The [row] and [col] sections specify which conditions are being tested in
# which wells.  The fields within these sections (e.g. `dilution`, `replicate`)
# can be anything.  If your plates aren't organized by row and column, there
# are other ways to define the plate layout; see the "File format" section for
# more details.

[col]
1.dilution = 1e5
2.dilution = 1e4
3.dilution = 1e3
4.dilution = 1e2
5.dilution = 1e1
6.dilution = 1e0

[row]
A.replicate = 1
B.replicate = 2
C.replicate = 3
```

3. Confirm that the layout is correct by using the *wellmap* command-line program to produce a visualization of the layout. This is an important step, because it's much easier to spot mistakes in the visualization than in the layout file itself.

```
$ wellmap std_curve.toml
```

This map shows that:

- Each row is a different replicate.
- Each column is a different dilution.

It is also possible to create maps like this directly from python, which may be useful in interactive sessions such as Jupyter notebooks:

```
>>> import wellmap
>>> wellmap.show("std_curve.toml")
<Figure size 321.203x255 with 4 Axes>
```

4. Load the data from the experiment in question into a tidy data frame. Tidy data are easier to work with in general, and are required by *wellmap* in particular. If you aren't familiar with the concept of tidy data, this article is a good introduction. The basic idea is to ensure that:

   - Each variable is represented by a single column.
   - Each observation is represented by a single row.

   If possible, it's best to export data from the instrument that collected it directly to a tidy format. When this isn't possible, though, you'll need to tidy the data yourself. For example, consider the following data (which corresponds to the layout from above). This is qPCR data, where a higher C_q value indicates that less material is present. The data are shaped like the plate itself, e.g. a row in the data for every row on the plate, and a column in the data for every column on the plate. It's not uncommon for microplate instruments to export data in this format.

Table 1: `std_curve.csv`

Cq	1	2	3	4	5	6
A	24.180858612060	24.740119934081	27.183801651000	27.742996215820	31.294982910156	29.967061996459961
B	24.157117843627	23.779703140258	27.917179489135	27.268831253052	30.862966537475	30.602731132507324
C	24.238229751586	24.787008285522	27.614759826660	26.779314041137	29.929668426514	29.675703945159912

Below is the code to load this data into a tidy `pandas.DataFrame` with the following columns:

- *row*: A letter identifying a row on the microplate, e.g. A-H
- *col*: A number identifying a column on the microplate, e.g. 1-12
- *Cq*: The C_q value measured for the identified well.

```
>>> import pandas as pd
>>> def load_cq(path):
...     return (pd
...             .read_csv(path)
...             .rename(columns={'Cq': 'row'})
...             .melt(
...                     id_vars=['row'],
...                     var_name='col',
...                     value_name='Cq',
...             )
...     )
>>> data = load_cq('std_curve.csv')
>>> data
   row col         Cq
```

(continues on next page)

(continued from previous page)

```
0    A   1  24.180859
1    B   1  24.157118
2    C   1  24.238230
3    A   2  20.740120
4    B   2  20.779703
5    C   2  20.787008
6    A   3  17.183802
7    B   3  17.171795
8    C   3  17.147598
9    A   4  13.774300
10   B   4  13.768831
11   C   4  13.779314
12   A   5  10.294983
13   B   5  10.362967
14   C   5  10.292967
15   A   6   6.967062
16   B   6   6.870273
17   C   6   6.735704
```

5. Use *wellmap.load()* to associate the labels specified in the TOML file (e.g. the dilutions and replicates) with the experimental data (e.g. the C_q values). This process has three steps:

   - Load a data frame containing the data (see above).
   - Load another data frame containing the labels.
   - Merge the two data frames.

   For the sake of clarity and completeness, we will first show how to perform these steps *manually*. Practically, though, it's easier to let *wellmap* perform them *automatically*.

Manual merge

Use the *wellmap.load()* function to create a pandas.DataFrame containing the information from the TOML file. This data frame will have columns for each label we specified: *replicate*, *dilution*. It will also have six columns identifying the wells in different ways: *well*, *well0*, *row*, *col*, *row_i*, *col_j*. These columns are redundant, but this redundancy makes it easier to merge the labels with the data. For example, if the wells are named "A1,A2,…" in the data, the *well* column can be used for the merge. If the wells are named "A01,A02,…", the *well0* column can be used instead. If the wells are named in some non-standard way, the *row_i* and *col_j* columns can be used to calculate an appropriate merge column.

```
>>> import wellmap
>>> labels = wellmap.load('std_curve.toml')
>>> labels
   well well0 row col  row_i  col_j  replicate  dilution
0    A1   A01   A   1      0      0          1  100000.0
1    A2   A02   A   2      0      1          1   10000.0
2    A3   A03   A   3      0      2          1    1000.0
3    A4   A04   A   4      0      3          1     100.0
4    A5   A05   A   5      0      4          1      10.0
5    A6   A06   A   6      0      5          1       1.0
6    B1   B01   B   1      1      0          2  100000.0
7    B2   B02   B   2      1      1          2   10000.0
```

(continues on next page)

(continued from previous page)

```
8    B3   B03   B   3      1      2           2    1000.0
9    B4   B04   B   4      1      3           2     100.0
10   B5   B05   B   5      1      4           2      10.0
11   B6   B06   B   6      1      5           2       1.0
12   C1   C01   C   1      2      0           3  100000.0
13   C2   C02   C   2      2      1           3   10000.0
14   C3   C03   C   3      2      2           3    1000.0
15   C4   C04   C   4      2      3           3     100.0
16   C5   C05   C   5      2      4           3      10.0
17   C6   C06   C   6      2      5           3       1.0
```

Use the `pandas.merge()` function to associate the labels with the data. In this case, both data frames have columns named *row* and *col*, so `pandas` will automatically use those for the merge. It is also easy to merge using columns with different names; see the documentation on `pandas.merge()` for more information.

```
>>> import pandas as pd
>>> df = pd.merge(labels, data)
>>> df
   well well0 row col  row_i  col_j  replicate  dilution         Cq
0    A1   A01   A   1      0      0          1  100000.0  24.180859
1    A2   A02   A   2      0      1          1   10000.0  20.740120
2    A3   A03   A   3      0      2          1    1000.0  17.183802
3    A4   A04   A   4      0      3          1     100.0  13.774300
4    A5   A05   A   5      0      4          1      10.0  10.294983
5    A6   A06   A   6      0      5          1       1.0   6.967062
6    B1   B01   B   1      1      0          2  100000.0  24.157118
7    B2   B02   B   2      1      1          2   10000.0  20.779703
8    B3   B03   B   3      1      2          2    1000.0  17.171795
9    B4   B04   B   4      1      3          2     100.0  13.768831
10   B5   B05   B   5      1      4          2      10.0  10.362967
11   B6   B06   B   6      1      5          2       1.0   6.870273
12   C1   C01   C   1      2      0          3  100000.0  24.238230
13   C2   C02   C   2      2      1          3   10000.0  20.787008
14   C3   C03   C   3      2      2          3    1000.0  17.147598
15   C4   C04   C   4      2      3          3     100.0  13.779314
16   C5   C05   C   5      2      4          3      10.0  10.292967
17   C6   C06   C   6      2      5          3       1.0   6.735704
```

Automatic merge

While it's good to understand how the labels are merged with the data, it's better to let *`wellmap`* perform the merge for you. Not only is this more succinct, it also handles some tricky corner cases behind the scenes, e.g. layouts with multiple data files.

To load *and* merge the data using *`wellmap.load()`*, you need to provide the following arguments:

- **data_loader**: A function that accepts a path to a file and returns a `pandas.DataFrame` containing the data from that file. Note that the function we wrote in the previous section fulfills these requirements. If the raw data are tidy to begin with, it is often possible to directly use `pandas.read_csv()` or similar for this argument.
- **merge_cols**: An indication of which columns to merge. In the snippet below, `True` means to use any columns that are shared between the two data frames (e.g. that have the same name). You can also use a

dictionary to be more explicit about which columns to merge on.

Here we also provide the **path_guess** argument, which specifies that the experimental data can be found in a CSV file with the same base name as the layout. It also would've been possible to specify the path to the CSV directly from the TOML file (see *meta.path*), in which case this argument would've been unnecessary.

```
>>> df = wellmap.load(
...         'std_curve.toml',
...         data_loader=load_cq,
...         merge_cols=True,
...         path_guess='{0.stem}.csv',
... )
>>> df
   well well0 row  ... replicate  dilution         Cq
0    A1   A01   A  ...         1  100000.0  24.180859
1    A2   A02   A  ...         1   10000.0  20.740120
2    A3   A03   A  ...         1    1000.0  17.183802
3    A4   A04   A  ...         1     100.0  13.774300
4    A5   A05   A  ...         1      10.0  10.294983
5    A6   A06   A  ...         1       1.0   6.967062
6    B1   B01   B  ...         2  100000.0  24.157118
7    B2   B02   B  ...         2   10000.0  20.779703
8    B3   B03   B  ...         2    1000.0  17.171795
9    B4   B04   B  ...         2     100.0  13.768831
10   B5   B05   B  ...         2      10.0  10.362967
11   B6   B06   B  ...         2       1.0   6.870273
12   C1   C01   C  ...         3  100000.0  24.238230
13   C2   C02   C  ...         3   10000.0  20.787008
14   C3   C03   C  ...         3    1000.0  17.147598
15   C4   C04   C  ...         3     100.0  13.779314
16   C5   C05   C  ...         3      10.0  10.292967
17   C6   C06   C  ...         3       1.0   6.735704

[18 rows x 10 columns]
```

6. Analyze the data given the connection between the labels and the data. This step doesn't involve *wellmap*, but is included here for completeness. The example below makes a linear regression of the data in log-space:

Listing 2: `std_curve.py`

```
#!/usr/bin/env python3

import wellmap
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import linregress

def load_cq(path):
    return (pd
            .read_csv(path)
            .rename(columns={'Cq': 'row'})
            .melt(
                    id_vars=['row'],
```

(continues on next page)

(continued from previous page)

```
                    var_name='col',
                    value_name='Cq',
                )
    )

df = wellmap.load(
      'std_curve.toml',
      data_loader=load_cq,
      merge_cols=True,
      path_guess='{0.stem}.csv',
)

x = df['dilution']
y = df['Cq']
m, b, r, p, err = linregress(np.log10(x), y)

x_fit = np.logspace(0, 5)
y_fit = np.polyval((m, b), np.log10(x_fit))

r2 = r**2
eff = 100 * (10**(1/m) - 1)
label = 'R$^2$={:.5f}\neff={:.2f}%'.format(r2, eff)

plt.plot(x_fit, y_fit, '--', label=label)
plt.plot(x, y, '+')
plt.legend(loc='best')
plt.xscale('log')
plt.xlabel('dilution')
plt.ylabel('Cq')
plt.show()
```

Fig. 1: R^2 is a measure of how well the line fits the data. In this case, the fit is very good. Note that there are three data points for each dilution, but they are hard to tell apart because they are almost perfectly superimposed. Efficiency is a measure of how well the qPCR reaction worked, or more specifically, how close the amount of DNA came to doubling (as would be expected) on each cycle. 100% indicates perfect doubling; 94% is a little on the low side.

1.2 Basic usage with R

The following steps show how to get started with *wellmapr* in R:

1. Install *wellmapr* from GitHub. It's good to be aware that *wellmapr* is written in python and made available to R using the reticulate package. This detail shouldn't affect you in normal usage, but may be relevant if the installation doesn't go smoothly:

```
> devtools::install_github("kalekundert/wellmap", subdir="wellmapr")
```

2. Write a *TOML file* describing the layout of an experiment. For example, the following layout might be used for a standard curve:

Listing 3: std_curve.toml

```
# The [row] and [col] sections specify which conditions are being tested in
# which wells.  The fields within these sections (e.g. `dilution`, `replicate`)
# can be anything.  If your plates aren't organized by row and column, there
# are other ways to define the plate layout; see the "File format" section for
# more details.

[col]
1.dilution = 1e5
2.dilution = 1e4
3.dilution = 1e3
4.dilution = 1e2
5.dilution = 1e1
6.dilution = 1e0

[row]
A.replicate = 1
B.replicate = 2
C.replicate = 3
```

3. Confirm that the layout is correct by using *wellmapr::show()* to produce a visualization of the layout. This is an important step, because it's much easier to spot mistakes in the visualization than in the layout file itself.

```
> wellmapr::show("std_curve.toml")
```

This map shows that:

- Each row is a different replicate.
- Each column is a different dilution.

It's also possible to create maps like this from the command line, which may be more convenient in some cases. The best way to do this is to use reticulate::py_config() to find the path to the python installation used by reticulate, then to invoke the *wellmap* command associated with that installation. The alias is optional, but could be saved in your shell configuration to make the command easier to remember:

```
$ Rscript -e 'reticulate::py_config()'
python:         /home/kale/.local/share/r-miniconda/envs/r-reticulate/bin/python
libpython:      /home/kale/.local/share/r-miniconda/envs/r-reticulate/lib/
↪libpython3.6m.so
pythonhome:     /home/kale/.local/share/r-miniconda/envs/r-reticulate:/home/kale/.
↪local/share/r-miniconda/envs/r-reticulate
version:        3.6.10 | packaged by conda-forge | (default, Apr 24 2020, 16:44:11)␣
↪ [GCC 7.3.0]
numpy:          /home/kale/.local/share/r-miniconda/envs/r-reticulate/lib/python3.6/
↪site-packages/numpy
numpy_version:  1.18.5
$ alias wellmap=/home/kale/.local/share/r-miniconda/envs/r-reticulate/bin/wellmap
$ wellmap std_curve.toml
```

4. Load the data from the experiment in question into a tidy data frame. Tidy data are easier to work with in general, and are required by *wellmapr* in particular. If you aren't familiar with the concept of tidy data, this article is a good introduction. The basic idea is to ensure that:

- Each variable is represented by a single column.
- Each observation is represented by a single row.

If possible, it's best to export data from the instrument that collected it directly to a tidy format. When this isn't possible, though, you'll need to tidy the data yourself. For example, consider the following data (which corresponds to the layout from above). This is qPCR data, where a higher C_q value indicates that less material is present. The data are shaped like the plate itself, e.g. a row in the data for every row on the plate, and a column in the data for every column on the plate. It's not uncommon for microplate instruments to export data in this format.

Table 2: `std_curve.csv`

Cq	1	2	3	4	5	6
A	24.180858612060547	20.740119934082	17.2	13.8	10.3	6.967061996459961
B	24.157117843627	20.779703140258	17.2	13.8	10.4	6.87027311325073
C	24.238229751586	20.787008285522	17.2	13.8	10.3	6.73570394515991

Below is the code to load this data into a tidy tibble with the following columns:

- *row*: A letter identifying a row on the microplate, e.g. A-H
- *col*: A number identifying a column on the microplate, e.g. 1-12
- *Cq*: The C_q value measured for the identified well.

```
> library(tidyverse)
>
> load_cq <- function(path) {
+   read_csv(path) %>%
+   rename(row = Cq) %>%
+   pivot_longer(
+       !row,
+       names_to = "col",
+       values_to = "Cq",
+   )
+ }
> data <- load_cq("std_curve.csv")
> data
# A tibble: 18 x 3
   row   col      Cq
   <chr> <chr> <dbl>
 1 A     1      24.2
 2 A     2      20.7
 3 A     3      17.2
 4 A     4      13.8
 5 A     5      10.3
 6 A     6       6.97
 7 B     1      24.2
 8 B     2      20.8
 9 B     3      17.2
10 B     4      13.8
11 B     5      10.4
12 B     6       6.87
13 C     1      24.2
14 C     2      20.8
```

(continues on next page)

(continued from previous page)

```
15 C     3     17.1
16 C     4     13.8
17 C     5     10.3
18 C     6      6.74
```

5. Use *wellmapr::load()* to associate the labels specified in the TOML file (e.g. the dilutions and replicates) with the experimental data (e.g. the C_q values). This process has three steps:

   - Load a data frame containing the data (see above).
   - Load another data frame containing the labels.
   - Merge the two data frames.

   For the sake of clarity and completeness, we will first show how to perform these steps *manually*. Practically, though, it's easier to let *wellmapr* perform them *automatically*.

Manual merge

Use the *wellmapr::load()* function to create a tibble containing the information from the TOML file. This data frame will have columns for each label we specified: *replicate*, *dilution*. It will also have six columns identifying the wells in different ways: *well*, *well0*, *row*, *col*, *row_i*, *col_j*. These columns are redundant, but this redundancy makes it easier to merge the labels with the data. For example, if the wells are named "A1,A2,..." in the data, the *well* column can be used for the merge. If the wells are named "A01,A02,...", the *well0* column can be used instead. If the wells are named in some non-standard way, the *row_i* and *col_j* columns can be used to calculate an appropriate merge column.

```
> layout <- wellmapr::load("std_curve.toml")
> layout
   well well0 row col row_i col_j replicate dilution
1    A1   A01   A   1     0     0         1    1e+05
2    A2   A02   A   2     0     1         1    1e+04
3    A3   A03   A   3     0     2         1    1e+03
4    A4   A04   A   4     0     3         1    1e+02
5    A5   A05   A   5     0     4         1    1e+01
6    A6   A06   A   6     0     5         1    1e+00
7    B1   B01   B   1     1     0         2    1e+05
8    B2   B02   B   2     1     1         2    1e+04
9    B3   B03   B   3     1     2         2    1e+03
10   B4   B04   B   4     1     3         2    1e+02
11   B5   B05   B   5     1     4         2    1e+01
12   B6   B06   B   6     1     5         2    1e+00
13   C1   C01   C   1     2     0         3    1e+05
14   C2   C02   C   2     2     1         3    1e+04
15   C3   C03   C   3     2     2         3    1e+03
16   C4   C04   C   4     2     3         3    1e+02
17   C5   C05   C   5     2     4         3    1e+01
18   C6   C06   C   6     2     5         3    1e+00
```

Use the dplyr::inner_join() function to associate the labels with the data. In this case, both data frames have columns named *row* and *col*, so those columns are automatically used for the merge (as indicated). It is also easy to merge using columns with different names; see the documentation on dplyr::inner_join() for more information.

```
> inner_join(layout, data)
Joining, by = c("row", "col")
   well well0 row col row_i col_j replicate dilution        Cq
1    A1   A01   A   1     0     0         1    1e+05 24.180859
2    A2   A02   A   2     0     1         1    1e+04 20.740120
3    A3   A03   A   3     0     2         1    1e+03 17.183802
4    A4   A04   A   4     0     3         1    1e+02 13.774300
5    A5   A05   A   5     0     4         1    1e+01 10.294983
6    A6   A06   A   6     0     5         1    1e+00  6.967062
7    B1   B01   B   1     1     0         2    1e+05 24.157118
8    B2   B02   B   2     1     1         2    1e+04 20.779703
9    B3   B03   B   3     1     2         2    1e+03 17.171795
10   B4   B04   B   4     1     3         2    1e+02 13.768831
11   B5   B05   B   5     1     4         2    1e+01 10.362967
12   B6   B06   B   6     1     5         2    1e+00  6.870273
13   C1   C01   C   1     2     0         3    1e+05 24.238230
14   C2   C02   C   2     2     1         3    1e+04 20.787008
15   C3   C03   C   3     2     2         3    1e+03 17.147598
16   C4   C04   C   4     2     3         3    1e+02 13.779314
17   C5   C05   C   5     2     4         3    1e+01 10.292967
18   C6   C06   C   6     2     5         3    1e+00  6.735704
```

Automatic merge

While it's good to understand how the labels are merged with the data, it's better to let *wellmapr* perform the merge for you. Not only is this more succinct, it also handles some tricky corner cases behind the scenes, e.g. layouts with multiple data files.

To load *and* merge the data using *wellmapr::load()*, you need to provide the following arguments:

- **data_loader**: A function that accepts a path to a file and returns a tibble containing the data from that file. Note that the function we wrote in the previous section fulfills these requirements. If the raw data are tidy to begin with, it is often possible to directly use readr::read_csv() or similar for this argument.
- **merge_cols**: An indication of which columns to merge. In the snippet below, TRUE means to use any columns that are shared between the two data frames (e.g. that have the same name). You can also use a dictionary to be more explicit about which columns to merge on.

Here we also provide the **path_guess** argument, which specifies that the experimental data can be found in a CSV file with the same base name as the layout. Note that this argument uses the syntax for string formatting in python, as described in the *API documentation*. It also would've been possible to specify the path to the CSV directly from the TOML file (see *meta.path*), in which case this argument would've been unnecessary.

```
> wellmapr::load(
+     "std_curve.toml",
+     data_loader = load_cq,
+     merge_cols = TRUE,
+     path_guess = "{0.stem}.csv",
+ )
   well well0 row col row_i col_j                          path replicate dilution␣
↪       Cq
0    A1   A01   A   1     0     0 <environment: 0x56501964bc60>         1    1e+05␣
↪24.180859
1    A2   A02   A   2     0     1 <environment: 0x565019653a68>         1    1e+04␣
↪20.740120
```

(continues on next page)

(continued from previous page)

```
2    A3   A03   A   3     0     2 <environment: 0x56501965d790>          1    1e+03␣
↪17.183802
3    A4   A04   A   4     0     3 <environment: 0x565019665598>          1    1e+02␣
↪13.774300
4    A5   A05   A   5     0     4 <environment: 0x56501966f2c0>          1    1e+01␣
↪10.294983
5    A6   A06   A   6     0     5 <environment: 0x565019673298>          1    1e+00 ␣
↪6.967062
6    B1   B01   B   1     1     0 <environment: 0x56501967b0a0>          2    1e+05␣
↪24.157118
7    B2   B02   B   2     1     1 <environment: 0x565019684dc8>          2    1e+04␣
↪20.779703
8    B3   B03   B   3     1     2 <environment: 0x56501968cbd0>          2    1e+03␣
↪17.171795
9    B4   B04   B   4     1     3 <environment: 0x5650196968f8>          2    1e+02␣
↪13.768831
10   B5   B05   B   5     1     4 <environment: 0x56501969e700>          2    1e+01␣
↪10.362967
11   B6   B06   B   6     1     5 <environment: 0x5650196a8428>          2    1e+00 ␣
↪6.870273
12   C1   C01   C   1     2     0 <environment: 0x5650196b0230>          3    1e+05␣
↪24.238230
13   C2   C02   C   2     2     1 <environment: 0x5650196b9f58>          3    1e+04␣
↪20.787008
14   C3   C03   C   3     2     2 <environment: 0x5650196c3c80>          3    1e+03␣
↪17.147598
15   C4   C04   C   4     2     3 <environment: 0x5650196cba88>          3    1e+02␣
↪13.779314
16   C5   C05   C   5     2     4 <environment: 0x5650196d57b0>          3    1e+01␣
↪10.292967
17   C6   C06   C   6     2     5 <environment: 0x5650196dd5b8>          3    1e+00 ␣
↪6.735704
```

6. Analyze the data given the connection between the labels and the data. This step doesn't involve *wellmap*, but is included here for completeness. The example below makes a linear regression of the data in log-space:

Listing 4: `std_curve.R`

```r
library(tidyverse)

load_cq <- function(path) {
  read_csv(path) %>%
  rename(row = Cq) %>%
  pivot_longer(
      !row,
      names_to = "col",
      values_to = "Cq",
  )
}

df <- wellmapr::load(
    "std_curve.toml",
```

(continues on next page)

(continued from previous page)

```
    data_loader = load_cq,
    merge_cols = TRUE,
    path_guess = "{0.stem}.csv",
)

ggplot(df, aes(x = dilution, y = Cq)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_x_log10()
```

1.3 Example layouts

Below are examples of plate layouts used in actual experiments.

1.3.1 -galactosidase assay

The following layout was used to measure the expression of -galactosidase in different conditions. Particularly noteworthy are the *fit_start_min* and *fit_stop_min* parameters. In this assay, the concentration of the enzyme is deduced from a linear fit of absorbance over time (measured using a plate reader). However, the reaction becomes non-linear as the substrate is exhausted, which happens at different times for different conditions (i.e. depending on how much enzyme is expressed). The *fit_start_min* and *fit_stop_min* parameters specify which data points are in the linear regime. The default is to use the data points between 5–30 min, but several wells use different cutoffs to better fit the data. This is an good example of how the fine-grained control provided by *wellmap* can be used to facilitate analysis.

Listing 5: `beta_gal_assay.toml`

```
[expt]
spacer = 'lz'
ligand = 'theophylline'
fit_start_min = 5
fit_stop_min = 30

[row.A]
growth_time_h = 6
[row.B]
growth_time_h = 8
[row.C]
growth_time_h = 10
[row.D]
growth_time_h = 16

[col.3]
sgrna = 'on'
ligand_mM = 0
[col.4]
sgrna = 'on'
ligand_mM = 30
[col.5]
```

(continues on next page)

(continued from previous page)

```
sgrna = 'off'
ligand_mM = 0
[col.6]
sgrna = 'off'
ligand_mM = 30

[well.B5]
fit_start_min = 0
fit_stop_min = 15
[well.C5]
fit_start_min = 5
fit_stop_min = 15
[well.D5]
fit_start_min = 0
fit_stop_min = 15
[well.D6]
fit_start_min = 0
fit_stop_min = 15
```

1.3.2 Bradford assay

The following layout was used to measure the concentration of purified protein mutants using a Bradford assay. There are a few things worth noting for this example:

- The same standard curve can be used for many experiments, so it makes sense to keep those concentrations in a separate file, to be included as necessary. Specifying these concentrations in a single place reduces redundancy and decreases the chance of making mistakes.
- The wells in the standard curve layout are specified using *[block]* instead of *[row]* and *[col]*. This makes it safe to include the standard curve in other layouts, because the blocks won't grow as more wells are added to the layout.
- The `[bradford]` block provides information on how to parse and interpret the data, e.g. what format the data is in and what wavelengths were measured. This information can be accessed in analysis scripts via the **extras** argument to *load()*:

```
>>> import wellmap
>>> df, ex = wellmap.load('bradford_assay.toml', extras=True)
>>> ex
{'bradford': {'format': 'biotek', 'absorbance': '595/450'}}
```

Listing 6: `bradford_standards.toml`

```
[block.9x3.A1]
standard = true

# Pierce BCA Protein Assay Kit
# Catalog: Thermo #23225
# Manual: tinyurl.com/y8uj7dzy
[block.1x3]
A1.ug_mL = 2000
```

(continues on next page)

(continued from previous page)

```
A2.ug_mL = 1500
A3.ug_mL = 1000
A4.ug_mL = 750
A5.ug_mL = 500
A6.ug_mL = 250
A7.ug_mL = 125
A8.ug_mL = 25
A9.ug_mL = 0
```

Listing 7: bradford_assay.toml

```
[meta]
include = 'bradford_standards.toml'

[bradford]
format = 'biotek'
absorbance = '595/450'

[block.3x2]
D1.sample  = 'Y37A'
D4.sample  = 'D42A'
D7.sample  = 'T44A'
D10.sample = 'Y45A'
F1.sample  = 'Y37E'
F4.sample  = 'T44P'
F7.sample  = 'Y45R'

[row]
'D,F'.dilution = 1
'E,G'.dilution = 5
```

1.3.3 qPCR timecourse

The following layout was used in a qPCR experiment to measure the change in GFP expression (compared to the 16S reference gene) over time in different ligand conditions. Note that the TOML file has very little redundancy, even though the layout isn't particularly regular.

Listing 8: qpcr_timecourse.toml

```
[expt]
sgrna = 'ligRNA-'

[block.4x3.A1]
time = 00:00:00
[block.8x3.A9]
time = 00:02:00
[block.8x3.D1]
time = 00:04:20
[block.8x3.D9]
time = 00:07:00
```

(continues on next page)

(continued from previous page)

```
[block.8x3.G1]
time = 00:10:00
[block.8x3.G9]
time = 00:13:20
[block.8x3.J1]
time = 00:17:00
[block.8x3.J9]
time = 00:21:00
[block.8x3.M1]
time = 00:25:20
[block.8x3.M9]
time = 00:30:00

[col.'1,3,...,17']
primers = 'gfp'
[col.'2,4,...,18']
primers = '16s'

[block.2x3.A1]
ligand = 'apo'
[block.2x3.A3]
ligand = 'holo'
[block.2x12.D1]
ligand = 'apo→apo'
[block.2x15.A9]
ligand = 'apo→apo'
[block.2x12.D3]
ligand = 'apo→holo'
[block.2x15.A11]
ligand = 'apo→holo'
[block.2x12.D5]
ligand = 'holo→apo'
[block.2x15.A13]
ligand = 'holo→apo'
[block.2x12.D7]
ligand = 'holo→holo'
[block.2x15.A15]
ligand = 'holo→holo'

# Controls:
[block.2x3.A5]
control = 'no GFP'
[block.2x3.A7]
control = 'no RT'
[block.2x3.A17]
control = 'no cDNA'
```

1.4 Related software

There are a handful of other packages that may be helpful when working with microplate experiments. Most of these packages parse plate layouts from spreadsheet files. In contrast, wellmap parses layout information from text files using a file format designed specifically for encoding plate layouts. As a result, these files are:

- Less redundant.
- Easier to read.
- Easier to write.

Wellmap also includes a tool for visualizing plate layouts, which makes it easy to see if there's a mistake in your layouts. None of the alternatives provide a comparable tool.

1.4.1 plater

An R library that parses plate layouts from a spreadsheet files into tidy data frames. The documentation is excellent and the library is easy to use. Multiple plates are supported, and in some cases the data and the layout can be put in the same file. The biggest drawback (other than using spreadsheets to store layout information and not providing a way to visualize layouts) is that it cannot be used with python.

1.4.2 plate_map_to_list

A command-line tool that converts spreadsheet files containing plate layouts into tidy CSV or TSV files. By virtue of being a command-line program, this can be used no matter what language your analysis scripts are written in. However, the command-line approach depends on generated intermediate files, which may clutter up your directories. More importantly, it's possible for the generated files to get out of sync with the original layouts, which could cause confusion. You also have to merge the layout with the experimental data yourself, although this is generally a simple operation.

1.4.3 Bioplate

A python library that can parse plate layouts from spreadsheet files. However, no easy way is provided to merge this layout information with experimental data.

1.4.4 Plateo

A python library focused on simulating robotic pipetting protocols. It can parse plate layouts from spreadsheet files, but does not provide an easy way to merge this information with experimental data.

1.4.5 cellHTS

An R library focused on analyzing data from high-throughput RNAi experiments. The pipeline involves a bespoke file format for describing plate layouts, but it is not suitable for general use.

1.4.6 platetools

An R library that seems related to microplate layouts. I can't figure out exactly what it does, though; the documentation is inscrutable.

1.5 Getting help

If you find a bug or need help getting *wellmap* to work, please open a new issue on Github. Pull requests are also welcome!

1.6 File format

The basic organization of a *wellmap* file is as follows: first you specify a group of wells, then you specify the experimental parameters associated with those wells. For example, the following snippet specifies that well A1 has a concentration of 100:

```
[well.A1]
conc = 100
```

The file format is based on TOML, so refer to the TOML documentation for a complete description of the basic syntax. Typically, square brackets (i.e. tables) are used to identify groups of wells and key/value pairs are used to set the experimental parameters for those wells. Note however that all of the following are equivalent:

```
[well.A1]
conc = 100

[well]
A1.conc = 100

well.A1.conc = 100
```

Most of this document focuses on describing the various ways to succinctly specify different groups of wells, e.g. *[row.A]*, *[col.1]*, *[block.WxH.A1]*, etc. There is no need to specify the size of the plate. The data frame returned by *load()* will contain a row for each well implied by the layout file.

Experimental parameters can be specified by setting any key associated with a well group (e.g. `conc` in the above examples) to a scalar value (e.g. string, integer, float, boolean, date, time, etc.). There are no restrictions on what these parameters can be named, although complex names (e.g. with spaces or punctuation) may need to be quoted. The data frame returned by *load()* will contain a column named for each parameter associated with any well in the layout. Not every well needs to have a value for every parameter; missing values will be represented in the data frame by `nan`.

1.6.1 [meta]

Miscellaneous fields that affect how *wellmap* parses the file. This is the only section that does not describe the organization of any wells.

Note: All paths specified in this section can either be absolute (if they begin with a '/') or relative (if they don't). Relative paths are considered relative to the directory containing the TOML file itself, regardless of what the current working directory is.

---

meta.path

The path to the file containing the actual data for this layout. The **path_guess** argument of the *load()* function can be used to provide a default path when this option is not specified. If the layout includes multiple plates (i.e. if it has one or more *[plate.NAME]* sections), use *meta.paths* and not *meta.path*.

meta.paths

The paths to the files containing the actual data for each plate described in the layout. You can specify these paths either as a format string or a mapping:

- Format string: The "{}" will be replaced with the name of the plate (e.g. "NAME" for `[plate.NAME]`):

```
[meta]
paths = 'path/to/file_{}.dat'
```

- Mapping: Plate names (e.g. "NAME" for `[plate.NAME]`) are mapped to paths. This is more verbose, but more flexible than the format string approach:

```
[meta.paths]
a = 'path/to/file_a.dat'
b = 'path/to/file_b.dat'
```

If the layout doesn't explicitly define any plates (i.e. if it has no *[plate.NAME]* sections), use *meta.path* and not *meta.paths*.

meta.include

The paths to one or more files that should effectively be copied-and-pasted into this layout. This is useful for sharing common features between similar layouts, e.g. reusing a standard curve layout between multiple experiments, or even reusing entire layouts for replicates with different data paths. This setting can either be a string, a dictionary, or a list:

- String: The path to a single layout file to include.
- Dictionary: The path to a single layout file in include, with additional metadata. The dictionary can have the following keys:
  - *path* (string, required): The path to include.
  - *shift* (string, optional): Reposition all the wells in the included layout. This setting has the following syntax: `<well> to <well>`. For example, `A1 to B2` would shift all wells down and to the right by one. Some caveats: the included file cannot use the *[irow.A]* or *[icol.1]* well groups (this restriction may be possible to remove, let me know if it causes you problems), wells cannot be shifted to negative row or column indices, and the shift will not apply to any files that are concatenated to the included file via *meta.concat*.
- List: The paths to multiple layout files to include. Each item in the list can either be a string or a dictionary; both will be interpreted as described above. If multiple files define the same well groups, the later files will take precedence over the earlier ones.

Examples:

The first layout describes a generic 10-fold serial dilution. The second layout expands on the first by specifying which sample is in each row. Note that the first layout could not be used on its own because it doesn't specify any rows:

Listing 9: `serial_dilution.toml`

```
[col]
1.conc = 1e4
2.conc = 1e3
3.conc = 1e2
4.conc = 1e1
5.conc = 1e0
6.conc = 0
```

Listing 10: `meta_include.toml`

```
[meta]
include = 'serial_dilution.toml'

[row.'A,B']
sample = ''

[row.'C,D']
sample = ''
```

The following layouts demonstrate the *shift* option. Note that both layouts specify the same 2x2 block, but the block from the included file is moved down and to the right in the final layout:

Listing 11: `shift_parent.toml`

```
[block.2x2.A1]
x = 2
```

Listing 12: meta_include_shift.toml

```toml
[meta.include]
path = 'shift_parent.toml'
shift = 'A1 to C3'

[block.2x2.A1]
x = 1
```

meta.concat

The paths of one or more TOML files that should be loaded independently of this file and concatenated to the resulting data frame. This is useful for combining multiple independent experiments (e.g. replicates performed on different days) into a single layout for analysis. Unlike *meta.include*, the referenced paths have no effect on how this file is parsed, and are not themselves affected by anything in this file.

The paths can be specified either as a string, a list, or a dictionary. Use a string to load a single path and a list to load multiple paths. Use a dictionary to load multiple paths and to assign a unique plate name (its key in the dictionary) to each one. Assigning plate names in this manner is useful when concatenating multiple single-plate layouts (as in the example below), because it keeps the wells from different plates easy to distinguish. Note that the plate names specified via dictionary keys will override any plate names specified in the layouts themselves.

Example:

The first two layouts describe the same experiment with different samples. The third layout combines the first two for easier analysis.

Listing 13: expt_1.toml

```toml
[block.4x4.A1]
sample = ''
```

Listing 14: expt_2.toml

```toml
[block.4x4.A1]
sample = ''
```

Listing 15: `concat.toml`

```toml
[meta.concat]
X = 'expt_1.toml'
Y = 'expt_2.toml'
```

meta.alert

A message that should be printed to the terminal every time this file is loaded. For example, if something went wrong during the experiment that would affect how the data is interpreted, put that here to be reminded of that every time you look at the data.

1.6.2 [expt]

Specify parameters that apply to every well in the layout, e.g. parameters that aren't being varied. These parameters are important to record for two reasons that may not be immediately obvious. First, they contribute to the complete annotation of the experiment, which will make the experiment easier for others (including yourself, after a few months) to understand. Second, they make it easier to write reusable analysis scripts, because the scripts can rely on every layout specifying every relevant parameter, not only those parameters that are being varied.

Avoid using this section for metadata such as your name, the date, the name of the experiment, etc. While this kind of metadata does apply to every well, it doesn't affect how the data will be analyzed. Including it here needlessly bloats the data frame returned by *`load()`*. It's better to put this information in top-level key/value pairs (e.g. outside of any well group). Analysis scripts can still access this information using the **extras** argument to the *`load()`* function, but it will not clutter the data frame used for analysis.

Note that the *`wellmap`* command by default only displays experimental parameters that have at least two different values across the whole layout, which normally excludes *[expt]* parameters. To see such a parameter anyways, provide its name as one of the <attr> arguments.

Example:

This layout demonstrates the difference between *[expt]* parameters and metadata. All of the wells on this plate have the same sample, but the sample is relevant to the analysis and might vary in other layouts analyzed by the same script. In contrast, the name and date are just (useful) metadata.

Listing 16: `expt.toml`

```toml
name = "Kale Kundert"
date = 2020-05-26

[expt]
sample = ''

# Without this, the plate wouldn't have any wells.
[block.4x4.A1]
```

1.6.3 [plate.NAME]

Specify parameters that differ between plates. Each plate must have a unique name, which will be included in the data frame returned by *load()*. The names can be any valid TOML key. In other words, almost any name is allowed, but complex names (e.g. with spaces or punctuation) may need to be quoted. Note that these names are also used in *meta.paths* to associate data with each plate.

Any parameters specified outside of a plate will apply to all plates. Any key/value pairs specified at the top-level of a plate will apply to the whole plate. Any well groups specified within a plate (e.g. `[plate.NAME.row.A]`) will only apply to that plate, and will take precedence over values specified in the same well groups (e.g. `[row.A]`) outside the plate. Refer to the *Precedence rules* for more information.

Example:

The following layout shows how to define parameters that apply to:

- All plates (conc).
- One specific plate (sample=).
- Part of one specific plate (sample=,).

Listing 17: `plate.toml`

```toml
[plate.X]
sample = ''

[plate.Y.block.2x4.A1]
sample = ''

[plate.Y.block.2x4.A3]
sample = ''

[col.'1,3']
conc = 0

[col.'2,4']
conc = 100

# Without this, plate X wouldn't have any rows.
[row.'A,B,C,D']
```

1.6.4 [row.A]

Specify parameters for all the wells in the given row (e.g. "A"). Rows must be specified as letters, either upper- or lower-case. If necessary, rows beyond "Z" can be specified with multiple letters (e.g. "AA", "AB", etc.). You can use the *pattern syntax* to specify multiple rows at once, e.g. `[row.'A,C,E']` or `[row.'A,C,...,G']`.

Examples:

The following layout specifies a different sample for each row:

Listing 18: `row.toml`

```
[row]
A.sample = ''
B.sample = ''
C.sample = ''
D.sample = ''

# Indicate how many columns there are.
[col.'1,2,3,4']
```

The following layout uses the *pattern syntax* to specify the same sample in multiple rows:

Listing 19: `row_pattern.toml`

```
[row.'A,C']
sample = ''

[row.'B,D']
sample = ''

# Indicate how many columns there are.
[col.'1,2,3,4']
```

1.6.5 [col.1]

Specify parameters for all the wells in the given column (e.g. "1"). Columns must be specified using integer numbers, starting from 1. You can use the *pattern syntax* to specify multiple columns at once, e.g. `[col.'1,3,5']` or `[col.'1,3,...,7']`.

Examples:

The following layout specifies a different sample for each column:

Listing 20: `col.toml`

```
[col]
1.sample = ''
2.sample = ''
3.sample = ''
4.sample = ''

# Indicate how many rows there are.
[row.'A,B,C,D']
```

The following layout uses the *pattern syntax* to specify the same sample in multiple columns:

Listing 21: `col_pattern.toml`

```
[col.'1,3']
sample = ''

[col.'2,4']
sample = ''

# Indicate how many rows there are.
[row.'A,B,C,D']
```

1.6.6 [irow.A]

Similar to *[row.A]*, but "interleaved" with the row above or below it. This layout is sometimes used for experiments that may be sensitive to neighbor effects or slight gradients across the plate.

Example:

The following layout interleaves samples between rows. Note that on the even columns, `[irow.A]` alternates "down" while `[irow.B]` alternates "up". In this fashion, A interleaves with B, C interleaves with D, etc.

Listing 22: `irow.toml`

```
[irow]
A.sample = ''
B.sample = ''
C.sample = ''
D.sample = ''

# Indicate how many columns there are.
[col.'1,2,...,4']
```

1.6.7 [icol.1]

Similar to *[col.1]*, but "interleaved" with the column to the left or right of it. This layout is sometimes used for experiments that may be sensitive to neighbor effects or slight gradients across the plate.

Example:

The following layout interleaves samples between columns. Note that on the rows columns (i.e. B/D/H/F), `[icol.1]` alternates "right" while `[icol.2]` alternates "left". In this fashion, 1 interleaves with 2, 3 interleaves with 4, etc.

Listing 23: `icol.toml`

```
[icol]
1.sample = ''
2.sample = ''
3.sample = ''
4.sample = ''

# Indicate how many rows there are.
[row.'A,B,...,D']
```

1.6.8 [block.WxH.A1]

Specify parameters for a block of wells W columns wide, H rows tall, and with the given well (e.g. "A1") in the top-left corner. You can use the *pattern syntax* to specify multiple blocks at once, e.g. `[block.2x2.'A1,A5']` or `[block.2x2.'A1,E5,...,E9']`.

Examples:

The following layout defines blocks of various sizes, each representing a different sample:

Listing 24: `block.toml`

```
[block.2x2]
A1.sample = ''
A3.sample = ''

[block.4x1]
C1.sample = ''
D1.sample = ''
```

The following layout uses the *pattern syntax* to specify the same sample in multiple blocks:

Listing 25: `block_pattern.toml`

```
[block.2x2.'A1,C3']
sample = ''

[block.2x2.'A3,C1']
sample = ''
```

1.6.9 [well.A1]

Specify parameters for the given well (e.g. "A1"). You can use the *pattern syntax* specify multiple wells at once, e.g. `[well.'A1,A3']` or `[well.'A1,B3,...,C11']`.

Examples:

The following layout specifies samples for two individual wells:

Listing 26: `well.toml`

```toml
[well.A1]
sample = ''

[well.D4]
sample = ''
```

The following layout uses the *pattern syntax* to specify the same sample for multiple wells:

Listing 27: `well_pattern.toml`

```toml
[well.'A1,D4,...,D4']
sample = ''
```

1.6.10 Pattern syntax

You can specify multiple indices for any row, column, block, or well. This can often help reduce redundancy, which in turn helps reduce the chance of mistakes. The following table shows some examples of this syntax:

Syntax	Meaning
`[row.A-D]`	A, B, C, D
`[row.'A,C']`	A, C
`[row.'A-C,F-H']`	A, B, C, F, G, H
`[row.'A,C,...,G']`	A, C, E, G
`[col.1-4]`	1, 2, 3, 4
`[col.'1,3']`	1, 3
`[col.'1-3,7-9']`	1, 2, 3, 7, 8, 9
`[col.'1,3,...,7']`	1, 3, 5, 7
`[well.A1-B2]`	A1, A2, B1, B2
`[well.'A1,A3']`	A1, A3
`[well.'A1-B2,A5-B6']`	A1, A2, B1, B2, A5, A6, B5, B6
`[well.'A1,C3,...,E5']`	A1, A3, A5, C1, C3, C5, E1, E3, E5

There are three forms of this syntax. The first uses a hyphen to specify a range of positions for a single row, column, block, or well. The second uses commas to specify multiple arbitrary positions for the same. These two forms can be used together, if desired. Note that the comma syntax needs to be quoted, because TOML doesn't allow unquoted keys to contain commas.

The third form uses ellipses to specify simple patterns. This requires exactly 4 comma-separated elements in exactly the following order: the first, second, and fourth must be valid indices, and the third must be an ellipsis ("…"). The

first and fourth indices define the start and end of the pattern (inclusive). The offset between the first and second indices defines the step size. It must be possible to get from the start to the end in steps of the given size.

Note that for wells and blocks, the hyphen ranges and ellipsis patterns can propagate across both rows and columns. In the case of ellipsis patterns, the second index specifies the step size in both dimensions. Consider the `A1,C3,...,E5` example from above: C3 is two rows and two columns away from A1, so this pattern specifies every odd well between A1 and E5.

1.6.11 Precedence rules

It is possible to specify multiple values for a single experimental parameter in a single well. The following layout, where *[expt]* and *[well.A1]* both specify different samples for the same well, shows a typical way for this to happen:

```
[expt]
sample = ''

[well.A1]
sample = ''
```

In these situations, which value is used depends on which well group has higher "precedence". Below is a list of each well group, in order from highest to lowest precedence. In general, well groups that are more "specific" have higher precedence:

- *[well]*
- *[block]*
  - If two blocks have different areas, the smaller one has higher precedence.
  - If two blocks have the same area, the one that appears later in the layout has higher precedence.
- *[row]*
- *[col]*
- *[irow]*
- *[icol]*
- *[expt]*

[plate] groups do not have their own precedence. Instead, well groups used within *[plate]* groups have precedence a half-step higher than the same group used outside a plate. In other words, *[plate.NAME.row.A]* has higher precedence than *[row]*, but lower precedence than *[block]*.

The following layout is contrived, but visually demonstrates most of the precedence rules:

Listing 28: `precedence.toml`

```
[plate.X]

[plate.Y]
precedence = 'plate'

[plate.Z.row.A]
precedence = 'plate.row'

[well.A1]
precedence = 'well'
```

(continues on next page)

(continued from previous page)

```
[block.2x2.A1]
precedence = 'block.2x2'

[block.3x3.A1]
precedence = 'block.3x3'

[row.A]
precedence = 'row'

[col.1]
precedence = 'col'

[expt]
precedence = 'expt'

# Specify how many wells to show.
[block.5x5.A1]
```

Note that the order in which the well groups appear in the layout usually doesn't matter. It only matters if there are two well groups with equal precedence, in which case the one that appears later will be given higher precedence. This situation only really comes up when using patterns. For example, note how earlier values are overridden by later values in the following layout:

Listing 29: `order.toml`

```
[well.A1]
sample = ''

[well.'A1,A2']
sample = ''

[well.A2]
sample = ''
```

1.7 Python API

wellmap.load(toml_path, *[, data_loader, ...])	Load a microplate layout from a TOML file.
wellmap.show(toml_path[, attrs, color])	Visualize the given microplate layout.

1.7.1 wellmap.load

wellmap.**load**(*toml_path*, *, *data_loader=None*, *merge_cols=None*, *path_guess=None*, *path_required=False*, *extras=False*, *report_dependencies=False*, *on_alert=None*)

Load a microplate layout from a TOML file.

Parse the given TOML file and return a `pandas.DataFrame` with a row for each well and a column for each experimental condition specified in that file. If the **data_loader** and **merge_cols** arguments are provided (which is the most typical use-case), that data frame will also contain columns for any data associated with each well.

Parameters

- **toml_path** (*str*, *pathlib.Path*) – The path to a file describing the layout of one or more plates. See the *File format* page for details about this file.
- **data_loader** (*callable*) – Indicates that *load()* should attempt to load the actual data associated with the plate layout, in addition to loading the layout itself. The argument should be a function that takes a `pathlib.Path` to a data file, parses it, and returns a `pandas.DataFrame` containing the parsed data. The function may also take an argument named "extras", in which case the **extras** return value (described below) will be provided. Note that specifying a data loader implies that **path_required** is True.
- **merge_cols** (*bool*, *dict*) – Indicates whether or not—and if so, how—*load()* should merge the data frames representing the plate layout and the actual data (provided by **data_loader**). The argument can either be a boolean or a dictionary:

  If *False* (or falsey, e.g. `None`, `{}`, etc.), the data frames will be returned separately and not be merged. This is the default behavior.

  If *True*, the data frames will be merged using any columns that share the same name. For example, the layout will always have a column named *well*, so if the actual data also has a column named *well*, the merge would happen on those columns.

  If a dictionary, the data frames will be merged using the columns identified in each key-value pair of the dictionary. The keys should be column names from the data frame representing the plate layout (described below; see the **layout** return value), and the values should be column names from the data frame returned by **data_loader**. Below are some examples of this argument:

  – `{'well0': 'Well'}`: Indicates that the "Well" column in the data contains zero-padded well names, like "A01", "A02", etc.

  – `{'row_i': 'Row', 'col_j': 'Col'}`: Indicates that the 'Row' and 'Col' columns in the data contain 0-indexed coordinates (e.g. 0, 1, 2, ...) identifying each row and column, respectively.

  Some details and caveats:

  – In order to successfully merge two columns, the values in those columns must correspond exactly. For example, a column that contains unpadded well names like "A1" cannot be merged with a column that contains padded well names like "A01". This is why the **layout** data frame contains so many redundant columns: to increase the chance that one will correspond exactly with a column provided by the data. In some cases, though, it may be necessary for the **data_loader** function to construct an appropriate merge column.

  – The data frame returned by **data_loader()** must be "tidy". Briefly, a data frame is tidy if each of its columns represents a single variable (e.g. time, fluorescence) and each of its rows represents a single observation.

  - The *path* column of the layout is automatically included in the merge and never has to be specified (although it is not an error to do so). This is makes sense because *load()* itself knows what path each data frame was loaded from.

- **path_guess** (*str*) – Where to look for a data file if none is specified in the given TOML file. In other words, this is the default value for *meta.path*. This path is interpreted relative to the TOML file itself (unless it's an absolute path) and is formatted with a pathlib.Path representing said TOML file. In code, that would be: `path_guess.format(Path(toml_path))`. A typical value would be something like `'{0.stem}.csv'`.
- **path_required** (*bool*) – Indicates whether or not the given TOML file must reference one or more data files. A ValueError will be raised if this condition is not met. Data files found via **path_guess** are acceptable for this purpose.
- **extras** (*bool*) – If true, return a dictionary containing any key/value pairs present in the TOML file but not part of the layout. Typically, this would be used to get information pertaining to the whole analysis and not any wells in particular (e.g. instruments used, preferred algorithms, plotting parameters, etc.).
- **report_dependencies** (*bool*) – If true, return a set of all the TOML files that were read in the process of loading the layout from the given **toml_path**. See the description of **dependencies** below for more details. You can use this information in analysis scripts (e.g. in conjunction with os.path.getmtime()) to avoid repeating expensive analyses if the underlying layout hasn't changed.
- **on_alert** (*callable*) – A callback to invoke if the given TOML file contains a warning for the user. The default behavior is to print the warning to the terminal via stderr. If a callback is provided, it must take two arguments: a pathlib.Path to the TOML file containing the alert, and the message itself. Note that this could be called more than once, e.g. if there are included or concatenated files.

Returns

If neither **data_loader** nor **merge_cols** were provided:

- **layout** (pandas.DataFrame) – Information about the plate layout parsed from the given TOML file. The data frame will have a row for each well and a column for each experimental condition. In addition, there will be several columns identifying each well:
  - *plate*: The name of the plate for this well. This column will not be present if there are no `[plate]` blocks in the TOML file.
  - *path*: The path to the data file associated with the plate for this well. This column will not be present if no data files were referenced by the TOML file.
  - *well*: The name of the well, e.g. "A1".
  - *well0*: The zero-padded name of the well, e.g. "A01".
  - *row*: The name of the row for this well, e.g. "A".
  - *col*: The name of the column for this well, e.g. "1".
  - *row_i*: The row-index of this well, counting from 0.
  - *col_j*: The column-index of this well, counting from 0.

If **data_loader** was provided but **merge_cols** was not:

- **layout** (pandas.DataFrame) – See above.

- **data** (pandas.DataFrame) – The concatenated result of calling **data_loader()** on every path specified in the given TOML file. See pandas.concat() for more information on how the data from different paths are concatenated.

If **data_loader** and **merge_cols** were both provided:

- **merged** (pandas.DataFrame) – The result of merging the **layout** and **data** data frames along the columns specified by **merge_cols**. See pandas.merge() for more details on the merge itself. The resulting data frame will have one or more rows for each well (more are possible if there are multiple data points per well, e.g. a time course), a column for each experimental condition described in the TOML file, and a column for each kind of data loaded from the data files.

If **extras** was provided:

- **extras** – A dictionary containing any key/value pairs present in the TOML file but not part of the layout. For example, consider the following TOML file:

```
a = 1
b = 2
well.A1.c = 3
```

  If we were to load this file with `extras=True`, this return value would be `{'a': 1, 'b': 2}`.

If **report_dependencies** was provided:

- **dependencies** – A set containing absolute paths to every layout file that was referenced by **toml_path**. This includes **toml_path** itself, and the paths to any *included* or *concatenated* layout files. It does not include paths to *data files*, as these are included already in the *path* column of the **layout** or **merged** data frames.

1.7.2 wellmap.show

wellmap.**show**(*toml_path*, *attrs=None*, *color='rainbow'*)

Visualize the given microplate layout.

It's wise to visualize TOML layouts before doing any analysis, to ensure that all of the wells are correctly annotated. The *wellmap* command-line program is a useful tool for doing this, but sometimes it's more convenient to make visualizations directly from python (e.g. when working in a jupyter notebook). That's what this function is for.

Parameters

- **toml_path** (*str*, *pathlib.Path*) – The path to a file describing the layout of one or more plates. See the *File format* page for details about this file.
- **attrs** (*str*, *list*) – One or more attributes from the above TOML file to visualize. For example, if the TOML file contains something equivalent to `well.A1.conc = 1`, then "conc" would be a valid attribute. If no attributes are specified, the default is to display any attributes that have at least two different values.
- **color** (*str*) – The name of the color scheme to use. Each different value for each different attribute will be assigned a color from this scheme. Any name understood by either colorcet or matplotlib can be used.

Return type matplotlib.figure.Figure

1.8 R API

API documentation for R is available using the `help()` system:

```
> help(load, wellmapr)
> help(show, wellmapr)
```

1.9 Command-line usage

The *wellmap* package comes with a command-line tool (also called *wellmap*) that displays a visual representation of the plate layout described by a TOML file. This is meant to help catch mistakes, which can be easy to make in complex layouts.

For more information on this command and its options, run:

```
$ wellmap -h
Visualize the plate layout described by a wellmap TOML file.

Usage:
    wellmap <toml> [<attr>...] [-o <path>] [-p] [-c <color>] [-f]

Arguments:
    <toml>
        TOML file describing the plate layout to display.  For a complete
        description of the file format, refer to:

        https://wellmap.readthedocs.io/en/latest/file_format.html

    <attr>
        The name(s) of one or more attributes from the above TOML file to
        project onto the plate.  For example, if the TOML file contains
        something equivalent to `well.A1.conc = 1`, then "conc" would be a
        valid attribute.

        If no attributes are specified, the default is to display any
        attributes that have at least two different values.  For complex
        layouts, this may result in a figure too big to fit on the screen.
        The best solution for this is just to specify a smaller number of
        attributes to focus on.

Options:
    -o --output PATH
        Output an image of the layout to the given path.  The file type is
        inferred from the file extension.  If the path contains a dollar sign
        (e.g. '$.svg'), the dollar sign will be replaced with the base name of
        the <toml> path.

    -p --print
        Print a paper copy of the layout, e.g. to reference when setting up an
        experiment.  The default printer for the system will be used.  To see
        the current default printer, run: `lpstat -d`.  To change the default
```

(continues on next page)

(continued from previous page)

```
        printer, run: `lpoptions -d <printer name>`.  When printing, the
        default color scheme is changed to 'dimgray'.  This can still be
        overridden using the '--color' flag.

    -c --color NAME
        Use the given color scheme to illustrate which wells have which
        properties.  The given NAME must be one of the color scheme names
        understood by either `matplotlib` or `colorcet`.  See the links below
        for the full list of supported colors, but some common choices are
        given below.  The default is 'rainbow':

        rainbow:  blue, green, yellow, orange, red
        viridis:  purple, green, yellow
        plasma:   purple, red, yellow
        coolwarm: blue, red
        tab10:    blue, orange, green, red, purple, ...
        dimgray:  gray, black

        Matplotlib colors:
        https://matplotlib.org/examples/color/colormaps_reference.html

        Colorcet colors:
        http://colorcet.pyviz.org/

    -f --foreground
        Don't attempt to return the terminal to the user while the GUI runs.
        This is meant to be used on systems where the program crashes if run in
        the background.
```

1.10 Versions

Wellmap uses semantic versioning. Briefly, this means that minor version upgrades (e.g. 1.1 to 1.2) will never break any existing code, while major version upgrades (e.g. 1.1 to 2.0) might.

1.10.1 v3.4.0 (2022-05-08)

Feature

- Add hyphen range syntax (`eaf2c73`)

1.10.2 v3.3.1 (2022-03-26)

Fix

- Don't drop nans too aggressively (`a001c8f`)

1.10.3 v3.3.0 (2022-01-31)

Feature

- Allow `show(attrs=...)` to be a string (`0572a61`)

1.10.4 v3.2.1 (2021-11-10)

Fix

- Correct color concave well groups (`04015bf`)

1.10.5 v3.2.0 (2021-11-09)

Feature

- Pick colors based on well coordinates (`90a25a1`)

1.10.6 v3.1.1 (2021-10-11)

Fix

- Better error checking (`eea82a3`)

1.10.7 v3.1.0 (2021-10-07)

Feature

- Allow included layouts to be shifted (`2ad8b59`)

Documentation

- Fix typo (`5139943`)

1.10.8 v3.0.1 (2021-10-01)

Fix

- Allow multiple patterns to define the same well (`d0d852c`)

Documentation

- Add link the semantic versioning website (`eb2f5f2`)
- Include the change log in the online docs (`727f03f`)
- Revise manuscript after peer review (`0823802`)
- Briefly describe each alternative sfotware (`336b47a`)

1.10.9 v3.0.0 (2021-04-11)

Feature

- Simplify the *extras* argument (`7558f7a`)

Breaking

- Scipts using the *extras* argument will need to be corrected. (`7558f7a`)

Documentation

- Fix the Bradford assay example (`6e06004`)

1.10.10 v2.1.0 (2021-01-13)

Feature

- Teach wellmap how to print layouts (`2e1cfe4`)

Documentation

- Add an "R API" section (`4bea19a`)
- Consolidate the table in the pattern section (`4588a86`)
- Reformat manuscript for BMC Res Notes (`b689d26`)
- Tweak wording (`f7ece3a`)
- Consistently use lower-case for "python" (`b850377`)
- Translate the "Basic usage" tutorial for R (`fce3931`)
- Tweak manuscript (`7469ae7`)

PYTHON MODULE INDEX

INDEX