traytable

Release 0.2.0

Dennis Brookner

May 25, 2022

CONTENTS

A python package for tabulating crystallization results across many trays

`traytable` provides methods for

- storing all information about a crystallization screen in a dictionary of dictionaries
- extracting and tabulating all data about "hits" into a `pandas` dataframe.

Install via pip:

```
pip install traytable
```

CHAPTER

ONE

TRAYTABLE QUICKSTART

A python package for tabulating crystallization results across many trays

`traytable` provides methods for

- storing all information about a crystallization screen in dictionaries
- extracting and tabulating all data about "hits" into a `pandas` dataframe.

The goal of `traytable` is for all crystallization data to be inputted once and only once, and then conveniently looked up and reused whenever needed.

You can find a jupyter notebook with a brief demonstration of package functionality *here*.

1.1 Installation

```
pip install traytable
```

1.2 Usage

A super brief example:

```
import traytable as tt

myscreen = tt.screen(row='protein', col='PEG', maxwell='H6') # Each row is a different␣
↪[protein], and each column is a different %PEG
tray1 = tt.tray(myscreen, rows=[1,8], cols=[10,20]) # Rows vary from 1 to 8, columns␣
↪vary from 10 to 20
results = tt.well(tray1, 'A3', 'good') # there is a good crystal in well A3 of tray 1
results = tt.well(tray1, ['E4', 'E5'], 'needle', old_df=results) # there are needle-y␣
↪crystals in wells E4 and E5 of tray 1
```

The return `results` from `tt.well()` is a `pandas` data frame where every crystal you've logged gets its own row, and every parameter you've indicated gets its own column. This makes it easy to keep track of the best conditions for your crystals across many trays with slightly different conditions. Note that upon logging your "hits", there's no need to input [protein] or %PEG; that information is already encoded by the tray and well you specified!

1.2.1 Required arguments

`tt.screen()` requires:

- `row`: a string indicating the parameter that is encoded by each row in a tray
- `col`: a string indicating the parameter that is encoded by each column in a tray
- `maxwell`: a string indicating the name of the well in the bottom right corner of each tray. Any size tray is supported; however, currently, rows must be named with letters, and columns must be named with numbers

`tt.tray()` requires:

- `screen`: The screen, as created by `tt.screen()`, that this tray should inherit parameters from. You can't create a tray without a screen.
- `rows`: Specify the values to assign to each row with either a single number (to assign to all rows), a list of two numbers (to evenly space among the rows) or a list of numbers explicitly specifying a value for each row. With 8 rows, you might say `rows=5`, `rows=[1,8]` or `rows=[1,2,3,4,6,8,10,12]`.
- `cols`: Specify values for columns, with the same format as for `rows`.

`tt.well()` requires:

- `tray`: The tray
- `well`: The well; must be a string of format '[letter][number]', and must fall into the range specified by the screen's `maxwell`
- `quality`: Any type, though I recommend either a short categorical string (e.g. 'good', 'bad', 'needles', or 'multilattice') or a numerical score, in order to best utilize the tools of `pandas` to manipulate and summarize your results.
- `old_df`: Not strictly required, but to append previous results, pass previous returns from `tt.well()` to the next call as `old_df =`

1.2.2 Optional arguments

All three of these methods (`tt.screen()`, `tt.tray()`, and `tt.well()`) will accept any additional named arguments, and include them as columns in the final data frame. As you would expect, arguments passed to `tt.screen()` will apply to all wells in all trays in the screen, and arguments passed to `tt.tray()` will apply to all wells in that tray. For example:

```
detailedscreen = tt.screen(row='protein', col='PEG', maxwell='H6', construct='HEWL',␣
↪buffer='imidazole', bufferconc=20, salt='MnCl2', saltconc=125)
tray1 = tt.tray(detailedscreen, rows=[1,8], cols=[10,20], date='2021-01-01', setby='robot
↪', weathernotes='very humid day')
results = tt.well(tray1, 'A1', 'good', appxnum=3, notes='rod-shaped')
```

1.2.3 Other things of note

The `clonetray()` method

To save some typing, you can create trays with `tt.clonetray()`. Usage is `newtray = tt.clonetray(oldtray, **kwargs)`. Any additional arguments passed to `clonetray()` will supercede the associated parameter from the parent tray. For example:

```
# assume screen already exists
tray1 = tt.tray(screen, rows=[1,8], cols=[5,10], date='2021-01-02'
tray2 = tt.clonetray(tray1, date='2021-01-03')
```

The `read_rockmaker()` method

Note: More extensive documentation of this feature to come.
It is possible to score crystals on RockMaker/RockImager via the online GUI. As a preliminary means of interfacing between `traytable` and RockMaker, I have added the `tt.read_rockmaker()` function, which accepts a "Score Report" `.csv` file and reads it into a `traytable` format.

Special treatment of dates

A crystal will frequently have two dates associated with it - when the tray was set, and when the crystal is being logged. Two things of note happen to address this:

- Arguments named `'date'` passed to `tt.tray()` and `tt.well()` automatically become columns named `'date_set'` and `'date_logged'`, respectively.
- If both `'date'`s are present and in ISO format (`YYYY-MM-DD`), they are subtracted (via the `datetime` module) to compute a new column `days_elapsed`. This is an especially important datapoint in crystallization, so it makes sense to give it special treatment. This also avoids the redundant input of date set, date logged, and days elapsed, when the latter is of course determined by the two former.

Using `pandas` methods

As mentioned above, `tt.well()` returns a `pandas` dataframe. This means that you can use `pandas` methods and features as desired. One frequent usage might be printing out only select columns with bracket notation, or accessing a certain column with dot notation, e.g.

```
concise_results = results[['protein', 'PEG', 'quality']]
```

or

```
import numpy as np
number_of_crystals = np.sum(results.appxnum())
```

You can also use the built-in plotting backend of `pandas`, which can be nifty to visualize what conditions are working best.

```
results.plot.scatter('protein', 'PEG')
```

A slightly fancier plot:

```
import numpy as np

results['proteinplot'] = results.protein +  np.random.normal(scale=0.15,␣
↪size=len(results))
results['PEGplot'] = results.PEG +  np.random.normal(scale=0.15, size=len(results))

colordict= {'good':'green',
            'needles':'red'}

results.plot.scatter('proteinplot', 'PEGplot', alpha=0.5, c=results.quality.
↪map(colordict))
```

Mismatching columns

Results from subsequent calls to `tt.well()` are appended via an "`outer_join`", meaning that columns present in one dataframe but not the other will give `NaN` values where appropriate, but no errors. This gives flexibility to vary the kinds of details you include across different trays and wells, while still keeping the "core" data common to all crystals in one place.

CHAPTER

TWO

TRAYTABLE METHODS

Note that the submodules `traytable.screens`, `traytable.wells`, `traytable.csv` exist purely for bookkeeping, and all methods below are available from the top-level `import traytable`.

2.1 Making screens and trays

`traytable.screens.`**`clonetray`**(*oldtray*, ***kwargs*)

Copy a tray, overriding parameters as desired

Parameters

- **`oldtray`** (*`dict`*) – Tray to be copied
- **`**kwargs`** (*`any type`*) – Accepts all arguments accepted by tray(), including rows and cols. Any parameters extant in oldtray not specified here will be copied into newtray

Returns **newtray** – Dictionary to be passed to well() for logging hits from this tray

Return type dict

`traytable.screens.`**`screen`**(*row*, *col*, *maxwell*, ***kwargs*)

Create a screen with global parameters

Parameters

- **`row`** (*`string`*) – Parameter encoded by the row letter
- **`col`** (*`string`*) – Parameter encoded by the column number
- **`maxwell`** (*`string`*) – Name of the well in the bottom-right corner of each tray, e.g. 'H6'
- **`**kwargs`** (*`any type`*) – Any named arguments become global parameters to be applied to all wells in all trays in the screen

Returns **screen** – A dictionary containing the screen

Return type dict

`traytable.screens.`**`tray`**(*screen*, *rows*, *cols*, ***kwargs*)

Create a tray, based on a screen and row/column specifications

Parameters

- **`screen`** (*`dict`*) – Screen from which the tray inherits global parameters
- **`rows`** (*`list or float`*) – Value(s) to be used as row specifications. Must be a single number, a list of two numbers, or a list of length matchng the number of rows.

- **cols** (*list or float*) – Value(s) to be used as column specifications. Must be a single number, a list of two numbers, or a list of length matching the number of columns.
- ****kwargs** (*any type*) – Set any named parameters to apply them to all wells in the tray

Returns **tray** – Dictionary to be passed to well() for logging hits from this tray

Return type dict

2.2 Logging crystals

traytable.wells.**well**(*tray*, *well*, *quality*, *old_df=None*, ***kwargs*)

Add one or more rows to a dataframe of crystal hits

Parameters

- **tray** (*dict*) – Tray, as created by traytable.tray()
- **well** (*string or list of strings*) – Well name(s), in format '[letter][number]'
- **quality** (*string*) – Short categorical description, e.g. "good" or "needles"
- **old_df** (*pandas.core.frame.DataFrame, optional*) – Working dataframe to append to. If None, creates a new dataframe.
- ****kwargs** (*any type*) – Any additional named arguments will become columns in the dataframe

Raises

- **TypeError** – Improper type for well name
- **ValueError** – Row or column specified by 'well' is out of the range specified by tray['maxwell']

Returns **df** – Dataframe containing the new reults, optionally concatenated with old_df

Return type pandas.core.frame.DataFrame

The methods `setrows()` and `setcols()` are exported by the package, but not documented here because their use is not recommended, and they may be deprecated in a future version.

CHAPTER

THREE

EXAMPLES

The following jupyter notebook(s) contain example usage of `traytable`

[1]:
```
import traytable as tt
import matplotlib.pyplot as plt
```

Download this notebook and try it out yourself here

3.1 Making a screen

First, initialize the screen with `screen()`. This function requires that you specify

- the parameter that varies by row
- the parameter that varies by column
- the plate shape, in the form of a "max well", e.g. the well in the bottom right corner of the plate.

Note that `row` refers to the parameter encoded by the row name; this is the parameter that is the same within a row, rather than the parameter that varies across the row. Likewise for columns.

Finally, whatever additional named arguments you pass to `screen()` become "screen static" global parameters that apply to all wells in all trays in the screen. Perhaps you include the protein construct, a nickname for the screen, or the type of plate you're using.

[2]:
```
myscreen = tt.screen(row = 'protein', col = 'PEG', maxwell = 'H6',
                     construct = 'HEWL', buffer = 'imidazole 20mM')
```

Now let's make a tray. Like with `screen()`, `tray()` will parse any additional named arguments as "tray static" parameters that apply to all wells in the tray. A common example might be the date the tray was set, or a buffer or additive that is the same across the plate.

Most importantly, `tray()` accepts arguments `rows` and `cols` to specify the values of the parameters varying across the plate. These can be set in three ways:

- with a list of two numbers, e.g. `row = [4, 18]` which would evenly space values across the rows (with number of rows determined via the `maxwell` parameter for the screen
- with a list of numbers equal in length to the number of rows/columns, which get mapped to rows/columns explicitly
- with a single number, which will be used for all rows/columns

[3]:
```
tray1 = tt.tray(myscreen, date = '2021-01-01', pH = 5.8,
                rows = [4,18],
                cols = [20,25])
```

The `clonetray()` method clones a tray with useage `newtray = clonetray(screen, oldtray, **kwargs)` where you can override specific parameters of the tray being cloned. When trays are similar (or identical) this saves some typing.

[5]:
```
tray2 = tt.clonetray(tray1, date = '2021-01-03',
                     rows = [4, 5, 6, 7, 8, 10, 12, 14])
```

In this case, using `clonetray()` instead of `tray()` saves you from having to re-specify the pH and the column values, which haven't changed from the previous tray.

3.2 Logging hits!

Our two trays have some crystals! We can log wells with good (or bad!) crystals via the `well()` function. `well()` requires the tray, well, and a short string to describe crystal quality; any other named parameters (perhaps a more verbose description, or a number of crystals) are accepted and get their own column in the resulting dataframe.

For all but the first call to `well()`, don't forget `old_df=df` to concatenate the new results with the old results.

[7]:
```
df = tt.well(tray1, 'A6', 'good', quantity = 3)
df = tt.well(tray1, 'B6', 'good', quantity = 2, note = "chunkier than usual", old_df=df)
df = tt.well(tray1, 'C6', 'needles', old_df=df)
```

[8]:
```
df
```

[8]:
```
   protein   PEG  quality construct          buffer        date   pH   tray  \
0      4.0  25.0     good      HEWL  imidazole 20mM  2021-01-01  5.8  tray1
1      6.0  25.0     good      HEWL  imidazole 20mM  2021-01-01  5.8  tray1
2      8.0  25.0  needles      HEWL  imidazole 20mM  2021-01-01  5.8  tray1

  well  quantity                 note
0   A6       3.0                  NaN
1   B6       2.0  chunkier than usual
2   C6       NaN                  NaN
```

The `well()` function uses the tray and well to look up all the data you've logged in your screens.

If you have many wells, all of the same quality, you can log them all at once:

[9]:
```
df = tt.well(tray2, ['B3', 'C3', 'D3', 'E3'], 'needles', old_df=df)
df = tt.well(tray2, ['A5', 'A6', 'B5'], 'good', old_df=df, note='borderline')
df
```

[9]:
```
  protein   PEG  quality construct          buffer        date   pH   tray  \
0     4.0  25.0     good      HEWL  imidazole 20mM  2021-01-01  5.8  tray1
1     6.0  25.0     good      HEWL  imidazole 20mM  2021-01-01  5.8  tray1
2     8.0  25.0  needles      HEWL  imidazole 20mM  2021-01-01  5.8  tray1
3       5  22.0  needles      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
4       6  22.0  needles      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
5       7  22.0  needles      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
```

(continues on next page)

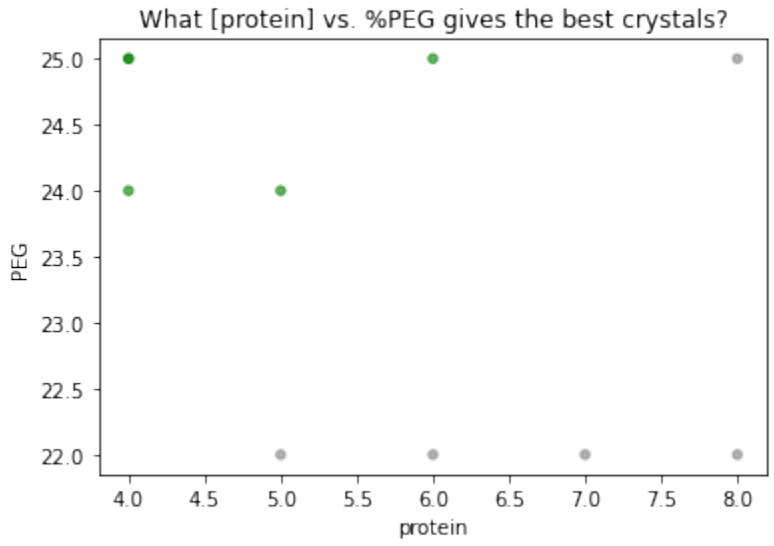
(continued from previous page)

```
6       8  22.0  needles      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
7       4  24.0     good      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
8       4  25.0     good      HEWL  imidazole 20mM  2021-01-03  5.8  tray2
9       5  24.0     good      HEWL  imidazole 20mM  2021-01-03  5.8  tray2

  well  quantity                 note
0   A6       3.0                  NaN
1   B6       2.0  chunkier than usual
2   C6       NaN                  NaN
3   B3       NaN                  NaN
4   C3       NaN                  NaN
5   D3       NaN                  NaN
6   E3       NaN                  NaN
7   A5       NaN           borderline
8   A6       NaN           borderline
9   B5       NaN           borderline
```

Finally, let's visualize which conditions are giving good crystals vs. needles.

[10]:
```
colordict= {'good':'green',
            'needles':'gray'}
df.plot.scatter('protein', 'PEG', alpha=0.6, c=df.quality.map(colordict))
plt.title('What [protein] vs. %PEG gives the best crystals?')
plt.show()
```



Looks like we should optimize with high PEG, low protein conditions. With `traytable`, no matter how many trays you've set with slightly varied screens, you can always consolidate your results in a single table or plot.

3.2.1 Other things of note

- You may have noticed that optional parameters present in some calls to `well()`, but not others, are harmlessly treated as `NaN` where missing.
- The `setrows()` and `setcols()` methods are called behind the scenes by `tray()` and `clonetray()` via the `rows` and `cols` keyword arguments, respectively, but are also available as stand-alone functions with usage `tray = setrows(tray, rows)` and likewise for columns.

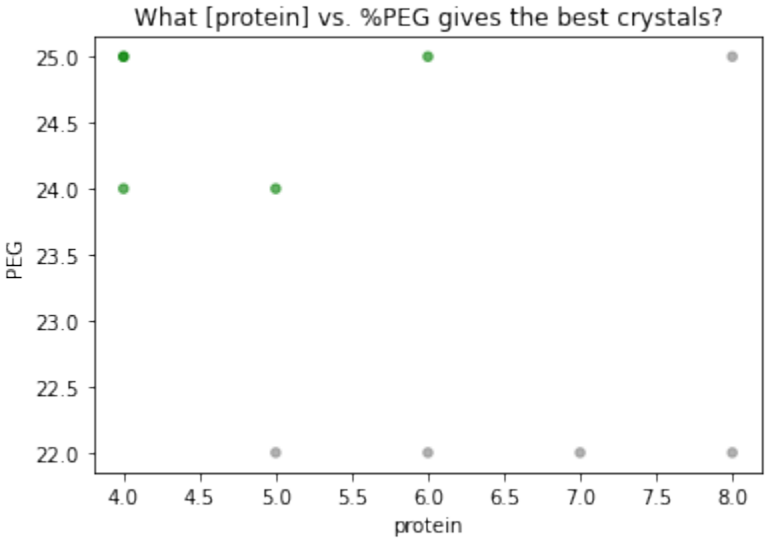
3.3 Just a code chunk

[11]:
```
import traytable as tt
import matplotlib.pyplot as plt

# make trays
myscreen = tt.screen(row = 'protein', col = 'PEG', maxwell = 'H6',
                     construct = 'HEWL', buffer = 'imidazole 20mM')
tray1 = tt.tray(myscreen, date = '2021-01-01', pH = 5.8,
                rows = [4,18],
                cols = [20,25])
tray2 = tt.clonetray(tray1, date = '2021-01-03',
                     rows = [4, 5, 6, 7, 8, 10, 12, 14])

# log results
df = tt.well(tray1, 'A6', 'good', quantity = 3)
df = tt.well(tray1, 'B6', 'good', quantity = 2, note = "chunkier than usual", old_df=df)
df = tt.well(tray1, 'C6', 'needles', old_df=df)
df = tt.well(tray2, ['B3', 'C3', 'D3', 'E3'], 'needles', old_df=df)
df = tt.well(tray2, ['A5', 'A6', 'B5'], 'good', old_df=df, note='borderline')

# plot results
colordict= {'good':'green',
            'needles':'gray'}
df.plot.scatter('protein', 'PEG', alpha=0.6, c=df.quality.map(colordict))
plt.title('What [protein] vs. %PEG gives the best crystals?')
plt.show()
```


What [protein] vs. %PEG gives the best crystals?
PEG
protein
25.0
24.5
24.0
23.5
23.0
22.5
22.0
4.0
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0

PYTHON MODULE INDEX

t

INDEX