cellpy Documentation

Release 1.0.0post4

Jan Petter Maehlen

Aug 21, 2023

CONTENTS

CHAPTER

ONE

- A LIBRARY FOR ASSISTING IN ANALYSING BATTERIES AND CELLS

This Python Package was developed to help the researchers at IFE, Norway, in their cumbersome task of interpreting and handling data from cycling tests of batteries and cells.

Version: 1.0.0post4

CHAPTER

TWO

INSTALLATION AND DEPENDENCIES

The easiest way to install `cellpy` is to install with conda or pip.

With conda:

```
conda install -c conda-forge cellpy
```

Or if you prefer installing using pip:

```
python -m pip install cellpy
```

Have a look at the documentation for more detailed installation procedures, especially with respect to "difficult" dependencies when installing with pip.

CHAPTER

THREE

LICENCE

`cellpy` is free software made available under the MIT License.

CHAPTER

FOUR

FEATURES

- Load test-data and store in a common format.
- Summarize and compare data.
- Filter out the steps of interest.
- Process and plot the data.
- And more…

CHAPTER

FIVE

CONTENTS

5.1 Main description

5.1.1 Installation

If you are (relatively) new to installing python packages, please jump to the getting started tutorial (*The getting started with cellpy tutorial (opinionated version)*) for an opinionated step-by-step procedure.

Stable release

Conda

Usually, the easiest way to install `cellpy` is by using conda:

```
$ conda install cellpy --channel conda-forge
```

This will also install all of the critical dependencies, as well as `jupyter` that comes in handy when working with `cellpy`.

Pip

If you would like to install only `cellpy`, you should install using pip. A small warning if you are on Windows: `cellpy` uses several packages that are a bit cumbersome to install on windows (e.g. `scipy`, `numpy` and `pytables`).

Install `cellpy` by running this command in your terminal:

```
$ python -m pip install cellpy
```

You can install pre-releases by adding the `--pre` flag.

If you are on Windows and plan to work with Arbin files, we recommend that you try to install pyodbc (Python ODBC bridge). Either by using pip or from conda-forge:

```
$ python -m pip install pyodbc
```

or:

```
$ conda install -c conda-forge pyodbc
```

Some of the utilities in `cellpy` have additional dependencies:

- Using the `ocv_rlx` utilities requires `lmfit` and `matplotlib`.
- For using the `batch` utilities efficiently, you should install `bokeh`, `plotly`, and `matplotlib` for plotting. Also, `holoviews` is a good tool to have.

If this is the first time you install `cellpy`, it is recommended that you run the setup script:

```
$ cellpy setup -i
```

This will install a `.cellpy_prms_USER.conf` file in your home directory (USER = your user name). Feel free to edit this to fit your needs.

If you are OK with letting `cellpy` select your settings, you can omit the *-i* (interactive mode).

Hint: Since `cellpy` uses several packages that are a bit cumbersome to install on windows, you circumvent this by install one of the `anaconda` python packages (python 3.9 or above) before installing `cellpy`. Remark, that if you chose `miniconda`, you need to manually install `scipy`, `numpy` and `pytables` using `conda`:

```
$ conda install scipy numpy pytables
```

Hint: It is recommended to run the `cellpy setup` command also after each time you upgrade `cellpy`. It will keep the settings you already have in your prms-file and, if the newer version has introduced some new parameters, it will add those too.

Hint: You can restore your prms-file by running `cellpy setup -r` if needed (*i.e.* get a copy of the default file copied to your user folder).

Caution: Since Arbin (at least some versions) uses access database files, you will need to install `pyodbc`, a python ODBC bridge that can talk to database files. On windows, at least if you don´t have a newer version of office 365, you most likely need to use Microsoft's dll for handling access database formats, and you might run into 32bit *vs.* 64bit issues. The simplest solution is to have the same "bit" for python and the access dll (or office). More advanced options are explained in more details in the getting-started tutorial. For Posix-type systems, you will need to download and install `mdbtools`. If you are on Windows and you cannot get your `pyodbc` to work, you can try the same there also (search for Windows binaries and set the appropriate settings in your `cellpy` config file).

From sources

The sources for `cellpy` can be downloaded from the Github repo.

You can clone the public repository by:

```
$ git clone git://github.com/jepegit/cellpy
```

Once you have a copy of the source, you can install in development mode using pip:

```
$ pip install -e .
```

(assuming that you are in the project folder, *i.e.* the folder that contains the setup.py file)

Further reading

You can find more information in the Tutorials, particularly in '*The getting started with cellpy tutorial (opinionated version)*'.

5.1.2 Usage

1. Simple usage as a python library

To use `cellpy`, start with importing it:

```
>>> import cellpy
```

Let us define some variables:

```
>>> file_name = r"C:\data\20141030_CELL_6_cc_01.res"
>>> mass = 0.982 # mass of active material in mg
>>> out_folder = r"C:\processed_data"
```

Then load the data into the data-class (this is data obtained using an Arbin battery tester, for the moment we assume that you are using the default settings where the default data-format is the Arbin .res format):

```
>>> c = cellpy.get(file_name, mass=mass)
```

Here we are choosing to go for the default options, and cellpy will load the file (using the file-loader "arbin_res" since the filename extension is .res), create a summary (for each cycle) and generate a step table (parsing the data and finding out what each step in each cycle is).

You can now save your data as a tester agnostic cellpy-file (uses hdf5 file format, and will include your summary and step table):

```
>>> c.save("cellpyfiles/20141030_CELL_6_cc_0.h5")
```

You can also save your data in csv-format easily by:

```
>>> c.to_csv(out_folder)
```

Or maybe you want to take a closer look at the capacities for the different cycles? No problem. Now you are set to extract data for specific cycles and steps:

```
>>> list_of_cycles = c.get_cycle_numbers()
>>> number_of_cycles = len(list_of_cycles)
>>> print(f"you have {number_of_cycles} cycles")
you have 658 cycles
>>> current_voltage_df = c.get_cap(5) # current and voltage for cycle 5 (as pandas.
↪DataFrame)
```

You can also look for open circuit voltage steps:

```
>>> cycle = 44
>>> time_voltage_df1 = c.get_ocv(ocv_type='ocvrlx_up', cycle_number=cycle)
>>> time_voltage_df2 = c.get_ocv(ocv_type='ocvrlx_down', cycle_number=cycle)
```

There are many more methods available, including methods for selecting steps and cycles (`get_current`, `get_voltage`, *etc.*) or tuning the data (*e.g.* `split` and `merge`).

Take a look at the index page (modules), some of the tutorials (tutorials) or notebook examples (Example notebooks).

2. Convenience methods and tools

The `cellpy.get` method interprets the file-type from the file extension and automatically creates the step table as well as the summary table:

```
>>> import cellpy
>>> c = cellpy.get(r"C:\data\20141030_CELL_6_cc_01.res", mass=0.982)
>>> # or load the cellpy-file:
>>> # cellpy.get("cellpyfiles/20141030_CELL_6_cc_0.h5")
```

If you provide the raw-file name and the cellpy-file name as input, `cellpy.get` only loads the raw-file if the cellpy-file is older than the raw-file:

```
>>> c = cellpy.get(raw_file_name, cellpyfile=cellpy_file_name)
```

Also, if your cell test consists of several raw files, you can provide a list of filenames:

```
>>> raw_files = [rawfile_01, rawfile_02]
>>> c.get(raw_files, cellpy_file)
```

`cellpy` will merge the two files for you and shift the running numbers (such as data-point) into one "continuous" file.

`cellpy` contains a logger (the logs are saved in the cellpy logging directory as defined in the config file). You can set the log level (to the screen) by:

```
>>> from cellpy import log
>>> log.setup_logging(default_level="DEBUG")
```

If you would like to use more sophisticated methods (*e.g.* database readers), take a look at the tutorial (if it exists), check the source code, or simply send an e-mail to one of the authors.

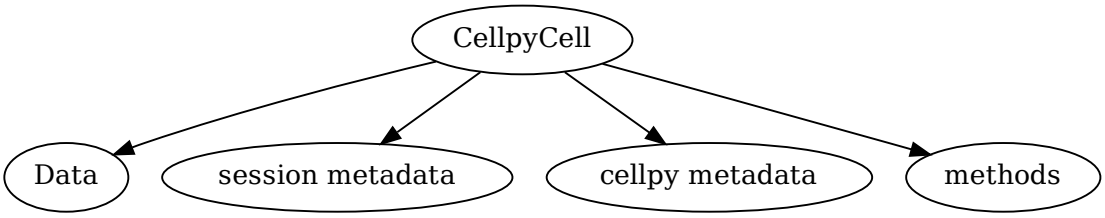
5.1.3 File Formats and Data Structures

> **Warning:** This part of the documentation is currently being updated.

The most important file formats and data structures for cellpy is summarized here. It is also possible to look into the source-code at the repository https://github.com/jepegit/cellpy.

Data Structures

CellpyCell - main structure



This class is the main work-horse for cellpy where all the functions for reading, selecting, and tweaking your data is located. It also contains the header definitions, both for the cellpy hdf5 format, and for the various cell-tester file-formats that can be read. The class can contain several tests and each test is stored in a list.

The class contains several attributes that can be assigned directly:

```
CellpyCell.tester = "arbin_res"  # TODO - update this
CellpyCell.auto_dirs = True
print(CellpyCell.cellpy_datadir)
```

CellpyCell - methods

The `CellpyCell` object contains lots of methods for manipulating, extracting and summarising the data from the run(s). Two methods are typically automatically run when you create your `CellpyCell` object when running `cellpy.get(filename)`:

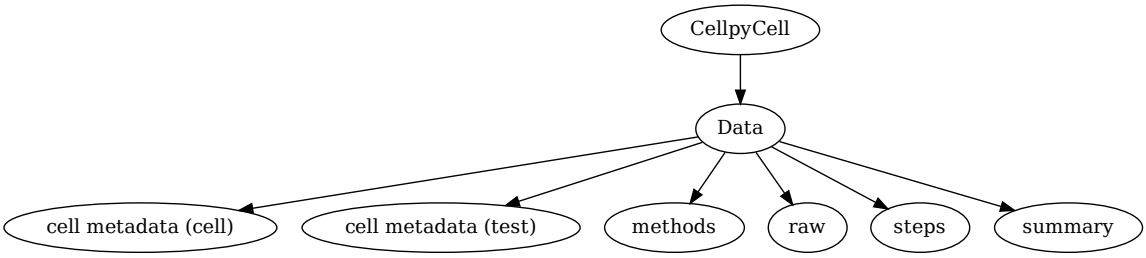
- `make_step_table`: creates a statistical summary of all the steps in the run(s) and categorizes the step type from that. It is also possible to give the step types directly (step_specifications).
- `make_summary`: create a summary based on cycle number.

Other methods worth mentioning are (based on what I typically use):

- `load`: load a cellpy file.
- `load_raw`: load raw data file(s) (merges automatically if several filenames are given as a list).
- `get_cap`: get the capacity-voltage graph from one or more cycles in three different formats as well as optionally interpolated, normalized and/or scaled.
- `get_cycle_numbers`: get the cycle numbers for your run.
- `get_ocv`: get the rest steps after each charge and discharge step.

Take a look at API section (Module index, `cellpy.readers.cellreader.CellpyCell`) for more info.

Data



The data for the experiment are stored in the class attribute `CellpyCell.data` (a `cellpy.cellreader.Data` instance).

The instance contain general information about the run-settings (such as mass etc.). The measurement data, information, and summary is stored in three `pandas.DataFrames`:

- `raw`: raw data from the run.
- `steps`: stats from each step (and step type), created using the `CellpyCell.make_step_table` method.
- `summary`: summary data vs. cycle number (e.g. coulombic coulombic efficiency), created using the `CellpyCell.make_summary` method.

The headers (columns) for the different DataFrames were given earlier in this chapter. As mentioned above, the `Data` object also contains metadata for the run.

metadata

```
cell_no = None
mass = prms.Materials.default_mass  # active material (in mg)
tot_mass = prms.Materials.default_mass  # total material (in mg)
no_cycles = 0.0
charge_steps = None
discharge_steps = None
ir_steps = None
ocv_steps = None
nom_cap = prms.DataSet.nom_cap  # mAh/g (for finding c-rates)
mass_given = False
material = prms.Materials.default_material
merged = False
file_errors = None  # not in use at the moment
loaded_from = None  # loaded from (can be list if merged)
channel_index = None
channel_number = None
creator = None
item_ID = None
schedule_file_name = None
start_datetime = None
test_ID = None
name = None
```

(continues on next page)

(continued from previous page)

```
cycle_mode = prms.Reader.cycle_mode
active_electrode_area = None  # [cm2]
active_electrode_thickness = None  # [micron]
electrolyte_type = None  #
electrolyte_volume = None  # [micro-liter]
active_electrode_type = None
counter_electrode_type = None
reference_electrode_type = None
experiment_type = None
cell_type = None
separator_type = None
active_electrode_current_collector = None
reference_electrode_current_collector = None
comment = None
```

The `Data` object can also take custom metadata if provided as keyword arguments (for developers).

FileID

The `FileID` object contains information about the raw file(s) and is used when comparing the cellpy-file with the raw file(s) (for example to check if it has been updated compared to the cellpy-file). Notice that `FileID` will contain a list of file identification parameters if the run is from several raw files.

Column headings

`cellpy` uses `pandas.DataFrame` objects internally. The column headers of the dataframes are defined in corresponding dataclass objects that can be accessed using both dot-notation and through normal dictionary look-up.

All the headers are set internally in `cellpy` and you can get them directly by e.g.

```
from cellpy.parameters.internal_settings import headers_normal

cycle_column_header = headers_normal.cycle_index_txt
```

column headings - raw data (or "normal" data)

```
@dataclass
class HeadersNormal(BaseHeaders):
    aci_phase_angle_txt: str = "aci_phase_angle"
    ref_aci_phase_angle_txt: str = "ref_aci_phase_angle"
    ac_impedance_txt: str = "ac_impedance"
    ref_ac_impedance_txt: str = "ref_ac_impedance"
    charge_capacity_txt: str = "charge_capacity"
    charge_energy_txt: str = "charge_energy"
    current_txt: str = "current"
    cycle_index_txt: str = "cycle_index"
    data_point_txt: str = "data_point"
    datetime_txt: str = "date_time"
    discharge_capacity_txt: str = "discharge_capacity"
```

(continues on next page)

(continued from previous page)

```
    discharge_energy_txt: str = "discharge_energy"
    internal_resistance_txt: str = "internal_resistance"
    power_txt: str = "power"
    is_fc_data_txt: str = "is_fc_data"
    step_index_txt: str = "step_index"
    sub_step_index_txt: str = "sub_step_index"
    step_time_txt: str = "step_time"
    sub_step_time_txt: str = "sub_step_time"
    test_id_txt: str = "test_id"
    test_time_txt: str = "test_time"
    voltage_txt: str = "voltage"
    ref_voltage_txt: str = "reference_voltage"
    dv_dt_txt: str = "dv_dt"
    frequency_txt: str = "frequency"
    amplitude_txt: str = "amplitude"
    channel_id_txt: str = "channel_id"
    data_flag_txt: str = "data_flag"
    test_name_txt: str = "test_name"
```

column headings - summary data

```
@dataclass
class HeadersSummary(BaseHeaders):
    """In addition to the headers defined here, the summary might also contain
    specific headers (ending in _gravimetric or _areal).
    """

    postfixes = ["gravimetric", "areal"]

    cycle_index: str = "cycle_index"
    data_point: str = "data_point"
    test_time: str = "test_time"
    datetime: str = "date_time"
    discharge_capacity_raw: str = "discharge_capacity"
    charge_capacity_raw: str = "charge_capacity"
    test_name: str = "test_name"
    data_flag: str = "data_flag"
    channel_id: str = "channel_id"

    coulombic_efficiency: str = "coulombic_efficiency"
    cumulated_coulombic_efficiency: str = "cumulated_coulombic_efficiency"

    discharge_capacity: str = "discharge_capacity"
    charge_capacity: str = "charge_capacity"
    cumulated_charge_capacity: str = "cumulated_charge_capacity"
    cumulated_discharge_capacity: str = "cumulated_discharge_capacity"

    coulombic_difference: str = "coulombic_difference"
    cumulated_coulombic_difference: str = "cumulated_coulombic_difference"
    discharge_capacity_loss: str = "discharge_capacity_loss"
```

(continues on next page)

(continued from previous page)

```
    charge_capacity_loss: str = "charge_capacity_loss"
    cumulated_discharge_capacity_loss: str = "cumulated_discharge_capacity_loss"
    cumulated_charge_capacity_loss: str = "cumulated_charge_capacity_loss"

    normalized_charge_capacity: str = "normalized_charge_capacity"
    normalized_discharge_capacity: str = "normalized_discharge_capacity"

    shifted_charge_capacity: str = "shifted_charge_capacity"
    shifted_discharge_capacity: str = "shifted_discharge_capacity"

    ir_discharge: str = "ir_discharge"
    ir_charge: str = "ir_charge"
    ocv_first_min: str = "ocv_first_min"
    ocv_second_min: str = "ocv_second_min"
    ocv_first_max: str = "ocv_first_max"
    ocv_second_max: str = "ocv_second_max"
    end_voltage_discharge: str = "end_voltage_discharge"
    end_voltage_charge: str = "end_voltage_charge"
    cumulated_ric_disconnect: str = "cumulated_ric_disconnect"
    cumulated_ric_sei: str = "cumulated_ric_sei"
    cumulated_ric: str = "cumulated_ric"
    normalized_cycle_index: str = "normalized_cycle_index"
    low_level: str = "low_level"
    high_level: str = "high_level"

    temperature_last: str = "temperature_last"
    temperature_mean: str = "temperature_mean"

    charge_c_rate: str = "charge_c_rate"
    discharge_c_rate: str = "discharge_c_rate"
    pre_aux: str = "aux_"
```

column headings - step table

```
@dataclass
class HeadersStepTable(BaseHeaders):
    test: str = "test"
    ustep: str = "ustep"
    cycle: str = "cycle"
    step: str = "step"
    test_time: str = "test_time"
    step_time: str = "step_time"
    sub_step: str = "sub_step"
    type: str = "type"
    sub_type: str = "sub_type"
    info: str = "info"
    voltage: str = "voltage"
    current: str = "current"
    charge: str = "charge"
    discharge: str = "discharge"
```

(continues on next page)

(continued from previous page)

```
    point: str = "point"
    internal_resistance: str = "ir"
    internal_resistance_change: str = "ir_pct_change"
    rate_avr: str = "rate_avr"
```

column headings - journal pages

```
@dataclass
class HeadersJournal(BaseHeaders):
    filename: str = "filename"
    mass: str = "mass"
    total_mass: str = "total_mass"
    loading: str = "loading"
    area: str = "area"
    nom_cap: str = "nom_cap"
    experiment: str = "experiment"
    fixed: str = "fixed"
    label: str = "label"
    cell_type: str = "cell_type"
    instrument: str = "instrument"
    raw_file_names: str = "raw_file_names"
    cellpy_file_name: str = "cellpy_file_name"
    group: str = "group"
    sub_group: str = "sub_group"
    comment: str = "comment"
    argument: str = "argument"


CellpyCell.keys_journal_session = ["starred", "bad_cells", "bad_cycles", "notes"]
```

step types

Identifiers for the different steps have pre-defined names given in the class attribute list *list_of_step_types* and is written to the "step" column.

```
list_of_step_types = ['charge', 'discharge',
                      'cv_charge', 'cv_discharge',
                      'charge_cv', 'discharge_cv',
                      'ocvrlx_up', 'ocvrlx_down', 'ir',
                      'rest', 'not_known']
```

Tester dependent attributes

For each type of testers that are supported by `cellpy`, a set of column headings and other different settings/attributes might also exist. These definitions stored in the `cellpy.parameters.internal_settings` module and are also injected into the `CellpyCell` class upon initiation.

5.1.4 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/jepegit/cellpy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

cellpy could always use more documentation, whether as part of the official cellpy docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/jepegit/cellpy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up `cellpy` for local development.

1. Fork the `cellpy` repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/cellpy.git
```

3. Create a local python virtual environment and activate it using python's venv utility:

```
$ python -m venv .venv
$ source .venv/bin/activate  # or .venv\Scripts\activate on Windows
```

   Or use `conda` environments. See the conda documentation for more information. A suitable environment yaml configuration file can be found in the root of the repository (`dev_environment.yml`; to create the environment, run `conda env create -f dev_environment.yml`).

4. Install your local copy into your virtualenv:

```
$ python -m pip install . -e
```

5. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

6. When you're done making changes, check that your changes pass the tests:

```
$ pytest
```

   If there are any libraries missing (it could happen) just pip install them into your virtual environment (or conda install them into your conda environment).

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website (or your IDE if that option exists).

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. The pull request should not include any gluten.

Tips

To self-hypnotize yourself to sleep well at night:

```
$ echo "You feel sleepy"
$ echo "You are a great person"
```

5.1.5 Credits

Development Lead

- Jan Petter Maehlen <jepe@ife.no>
- Muhammad Abdelhamid <muhammad.abdelhamid@ife.no>
- Asbjoern Ulvestad <asbjorn.ulvestad@ife.no>
- Julia Wind <julia.wind@ife.no>

Contributors

- Tor Kristian Vara
- Ulrik Aalborg Eriksen
- Carl Erik Lie Foss <carl.foss@ife.no>
- Amund Midtgard Raniseth <amund.raniseth@gmail.com>
- Michael Chon (@chonmj)
- Kavin Teenakul (@kevinsmia1939)
- Jayce Slesar (@jayceslesar)

5.1.6 History

1.0.0 (2023)

- Unit handling: new unit handling (using pint)
- Unit handling: renaming summary headers
- Unit handling: new cellpy-file-format version
- Unit handling: tool for converting old to new format
- Unit handling: parsing input parameters for units
- Templates: using one repository with sub-folders

- Templates: adding more documentation
- File handling: allow for external raw files (ssh)
- Readers: neware.txt (one version/model)
- Readers: arbin_sql7 (experimental, @jtgibson91)
- Batch plotting: collectors for both data collection, plotting and saving
- OCV-rlx: improvements of the OCV-rlx tools
- Internals: rename main classes (CellpyData -> CellpyCell, Cell -> Data)
- Internals: rename .cell property to .data
- Internals: allow for only one Data-object pr CellpyCell object
- CLI: general improvements and bug fixes
- CLI: move editing of db-file to the edit sub-command

0.4.3 (2023)

- Neware txt loader (supports one specific format only, other formats will have to wait for v.1.0)

0.4.2 (2022)

- Changed definition of Coulombic Difference (negative of previous)
- Updated loaders with hooks and additional base class TxtLoader with configuration mechanism
- Support for Maccor txt files
- Supports only python 3.8 and up
- Optional parameters through batch and pages
- Several bug fixes and minor improvements / adjustments
- Restrict use of instrument label to only one option
- Fix bug in example file (@kevinsmia1939)

0.4.1 (2021)

- Updated documentations
- CLI improvements
- New argument for get_cap: max_cycle
- Reverting from using Documents to user home for location of prm file in windows.
- Easyplot by Amund
- Arbin sql reader by Muhammad

0.4.0 (2020)

- Reading arbin .res files with auxiliary data should now work.
- Many bugs have been removed - many new introduced.
- Now on conda-forge (can be installed using conda).

0.4.0 a2 (2020)

- Reading PEC files now updated and should work

0.4.0 a1 (2020)

- New column names (lowercase and underscore)
- New batch concatenating and plotting routines

0.3.3 (2020)

- Switching from git-flow to github-flow
- New cli options for running batches
- cli option for creating template notebooks
- Using ruamel.yaml instead of pyyaml
- Using python-box > 4
- Several bug-fixes

0.3.2 (2019)

- Starting fixing documentation
- TODO: create conda package
- TODO: extensive tests

0.3.1 (2019)

- Refactoring - renaming from dfsummary to summary
- Refactoring - renaming from step_table to steps
- Refactoring - renaming from dfdata to raw
- Refactoring - renaming cellpy.data to cellpy.get
- Updated save and load cellpy files allowing for new naming
- Implemented cellpy new and cellpy serve cli functionality

0.3.0 (2019)

- New batch-feature
- Improved make-steps and make-summary functionality
- Improved cmd-line interface for setup
- More helper functions and tools
- Experimental support for other instruments
- invoke tasks for developers

0.2.1 (2018)

- Allow for using mdbtools also on win
- Slightly faster find_files using cache and fnmatch
- Bug fix: error in sorting files when using pathlib fixed

0.2.0 (2018-10-17)

- Improved creation of step tables (much faster)
- Default compression on cellpy (hdf5) files
- Bug fixes

0.1.22 (2018-07-17)

- Parameters can be set by dot-notation (python-box).
- The parameter Instruments.cell_configuration is removed.
- Options for getting voltage curves in different formats.
- Fixed python 3.6 issues with Read the Docs.
- Can now also be used on posix (the user must install mdb_tools first).
- Improved logging allowing for custom log-directory.

0.1.21 (2018-06-09)

- No legacy python.

0.1.0 (2016-09-26)

- First release on PyPI.

5.2 Examples and tutorials

5.2.1 Basics usage

The getting started with `cellpy` tutorial (opinionated version)

This tutorial will help you getting started with `cellpy` and tries to give you a step-by-step recipe. It starts with installation, and you should select the installation method that best suits your needs (or your level).

How to install and run `cellpy` - the tea spoon explanation for standard users

If you are used to installing stuff from the command line (or shell), then things might very well run smoothly. If you are not, then you might want to read through the guide for complete beginners first (see below *Setting up cellpy on Windows for complete beginners*).

1. Install a scientific stack of python 3.x

If the words "virtual environment" or "miniconda" do not ring any bells, you should install the Anaconda scientific Python distribution. Go to www.anaconda.com and select the Anaconda distribution (press the `Download Now` button). Use at least python 3.9, and select the 64 bit version (if you fail at installing the 64 bit version, then you can try the weaker 32 bit version). Download it and let it install.

> **Caution:** The bin version matters sometimes, so try to make a mental note of what you selected. E.g., if you plan to use the Microsoft Access odbc driver (see below), and it is 32-bit, you probably should chose to install a 32-bit python version).

Python should now be available on your computer, as well as a huge amount of python packages. And Anaconda is kind enough to also install an alternative command window called "Anaconda Prompt" that has the correct settings ensuring that the conda command works as it should.

2. Create a virtual environment

This step can be omitted (but its not necessarily smart to do so). Create a virtual conda environment called `cellpy` (the name is not important, but it should be a name you are able to remember) by following the steps below:

Open up the "Anaconda Prompt" (or use the command window) and type

```
conda create -n cellpy
```

This creates your virtual environment (here called *cellpy*) in which `cellpy` will be installed and used.

You then have to activate the environment:

```
conda activate cellpy
```

3. Install `cellpy`

In your activated `cellpy` environment in the Anaconda Prompt if you chose to make one, or in the base environment if you chose not to, run:

```
conda install -c conda-forge cellpy
```

Congratulations, you have (hopefully) successfully installed cellpy.

If you run into problems, doublecheck that all your dependencies are installed and check your Microsoft Access odbc drivers.

4. Check your cellpy installation

The easiest way to check if `cellpy` has been installed, is to issue the command for printing the version number to the screen

```
cellpy info --version
```

If the program prints the expected version number, you probably succeeded. If it crashes, then you will have to retrace your steps, redo stuff and hope for the best. If it prints an older (lower) version number than you expect, there is a big chance that you have installed it earlier, and what you would like to do is to do an `upgrade` instead of an `install`

```
python -m pip install --upgrade cellpy
```

If you want to install a pre-release (a version that is so bleeding edge that it ends with a alpha or beta release identification, *e.g.* ends with .b2). Then you will need to add the –pre modifier

```
python -m pip install --pre cellpy
```

To run a more complete check of your installation, there exist a `cellpy` sub-command than can be helpful

```
cellpy info --check
```

5. Set up `cellpy`

After you have installed `cellpy` it is highly recommended that you create an appropriate configuration file and folders for raw data, cellpy-files, logs, databases and output data (and inform `cellpy` about it).

To do this, run the setup command:

```
cellpy setup
```

To run the setup in interactive mode, use -i:

```
cellpy setup -i
```

This creates the cellpy configuration file `.cellpy_prms_USERNAME.conf` in your home directory (USERNAME = your user name) and creates the standard cellpy_data folders (if they do not exist). The `-i` option makes sure that the setup is done interactively: The program will ask you about where specific folders are, *e.g.* where you would like to put your outputs and where your cell data files are located. If the folders do not exist, `cellpy` will try to create them.

If you want to specify a root folder different from the default (your HOME folder), you can use the `-d` option *e.g.* `cellpy setup -i -d /Users/kingkong/cellpydir`

Hint: You can always edit your configurations directly in the cellpy configuration file `.cellpy_prms_USER.conf`. This file should be located inside your home directory, /. in posix and c:usersUSERNAME in not-too-old windows.

6. Create a notebook and run `cellpy`

Inside your Anaconda Prompt window, write:

```
jupyter notebook  # or jupyter lab
```

Your browser should then open and you are ready to write your first cellpy script.

There are many good tutorials on how to work with jupyter. This one by Real Python is good for beginners: Jupyter Notebook: An Introduction

Setting up `cellpy` on Windows for complete beginners

This guide provides step-by-step instructions for installing Cellpy on a Windows system, especially tailored for beginners.

1. Installing Python

- First, download Python from the official website. Choose the latest version for Windows.
- **Run the downloaded installer. On the first screen of the setup, ensure to check the box** saying "Add Python to PATH" before clicking "Install Now".
- After installation, you can verify it by opening the Command Prompt (see below) and typing:

  ```
  python --version
  ```

  This command should return the version of Python that you installed.

2. Opening Command Prompt

- Press the Windows key, usually located at the bottom row of your keyboard, between the Ctrl and Alt keys.
- Type "Command Prompt" into the search bar that appears at the bottom of the screen when you press the Windows key.
- Click on the "Command Prompt" application to open it.

3. Creating a Virtual Environment

A virtual environment is a tool that helps to keep dependencies required by different projects separate by creating isolated Python environments for them. Here's how to create one:

- Open Command Prompt.
- Navigate to the directory where you want to create your virtual environment using the *cd* command. For example:

```
cd C:\Users\YourUsername\Documents
```

- Type the following command and press enter to create a new virtual environment (replace *envname* with the name you want to give to your virtual environment):

```
python -m venv envname
```

- To activate the virtual environment, type the following command and press enter:

```
envname\Scripts\activate
```

  You'll know it worked if you see *(envname)* before the prompt in your Command Prompt window.

4. Installing Jupyter Notebook and matplotlib

Jupyter Notebook is an open-source web application that allows you to create documents containing live code, equations, visualizations, and text. It's very useful, especially for beginners. To install Jupyter Notebook:

- Make sure your virtual environment is activated.
- Type the following command and press enter:

```
python -m pip install jupyter matplotlib
```

5. Installing `cellpy`

Next, you need to install `cellpy`. You can install it via pip (Python's package manager). To install `cellpy`:

- Make sure your virtual environment is activated.
- Type the following command and press enter:

```
python -m pip install cellpy
```

6. Launching Jupyter Notebook

- Make sure your virtual environment is activated.
- Type the following command and press enter:

```
jupyter notebook
```

- This will open a new tab in your web browser with the Jupyter's interface. From there, create a new Python notebook by clicking on "New" > "Python 3".

7. Trying out cellpy

Here's a simple example of how to use Cellpy in a Jupyter notebook:

- In the first cell of the notebook, import Cellpy by typing:

```
import cellpy
```

  Press *Shift* + *Enter* to run the cell.

- In the new cell, load your data file (replace "datafile.res" and "/path/to/your/data" with your actual filename and path):

```
filepath = "/path/to/your/data/datafile.res"

c = cellpy.get(filepath)  # create a new cellpy object
```

  Press *Shift* + *Enter* to run the cell and load the data.

- To see a summary of the loaded data, create a new cell and type:

```
print(c.data.summary.head())
```

  Press *Shift* + *Enter* to run the cell and print the summary.

Congratulations! You've successfully set up Cellpy in a virtual environment on your Windows PC and loaded your first data file. For more information and examples, check out the official Cellpy documentation.

Cellpy includes convenient functions for accessing the data. Here's a basic example of how to plot voltage vs. capacity.

- In a new cell in your Jupyter notebook, first, import matplotlib, which is a Python plotting library:

```
import matplotlib.pyplot as plt
```

  Press *Shift* + *Enter* to run the cell.

- Then, iterate through all cycles numbers, extract the capacity curves and plot:

```
for cycle in c.get_cycle_numbers():
    d = c.get_cap(cycle)
    plt.plot(d["capacity"], d["voltage"])
plt.show()
```

  Press *Shift* + *Enter* to run the cell.

  This will produce a plot for each cycle in the loaded data.

Once you've loaded your data, you can save it to a hdf5 file for later use:

```
c.save("saved_data.h5")
```

This saves the loaded data to a file named 'saved_data.h5'.

Now, lets try to create some dQ/dV plots. dQ/dV is a plot of the change in capacity (Q) with respect to the change in voltage (V). It's often used in battery analysis to observe specific electrochemical reactions. Here's how to create one:

- In a new cell in your Jupyter notebook, first, if you have not imported matplotlib:

```
import matplotlib.pyplot as plt
```

  Press *Shift* + *Enter* to run the cell.

- Then, calculate dQ/dV using Cellpy's ica utility:

```
import cellpy.utils.ica as ica

dqdv = ica.dqdv_frames(c, cycle=[1, 10, 100], voltage_resolution=0.01)
```

Press *Shift* + *Enter* to run the cell.

- Now, you can create a plot of dQ/dV. In a new cell, type:

```
plt.figure(figsize=(10, 8))
plt.plot(dqdv["v"], dqdv["dq"], label="dQ/dV")
plt.xlabel("Voltage (V)")
plt.ylabel("dQ/dV (Ah/V)")
plt.legend()
plt.grid(True)
plt.show()
```

Press *Shift* + *Enter* to run the cell.

In the code above, *plt.figure* is used to create a new figure, *plt.plot* plots the data, *plt.xlabel* and *plt.ylabel* set the labels for the x and y axes, *plt.legend* adds a legend to the plot, *plt.grid* adds a grid to the plot, and *plt.show* displays the plot.

With this, you should be able to see the dQ/dV plot in your notebook.

Remember that the process of creating a dQ/dV plot can be quite memory-intensive, especially for large datasets, so it may take a while for the plot to appear.

For more information and examples, check out the official Cellpy documentation and the matplotlib documentation.

This recipe can only take you a certain distance. If you want to become more efficient with Python and Cellpy, you might want to try to install it using the method described in the chapter "Installing and setting up cellpy" in the official Cellpy documentation.

More about installing and setting up `cellpy`

Fixing dependencies

To make sure your environment contains the correct packages and dependencies required for running cellpy, you can create an environment based on the available `environment.yml` file. Download the environment.yml file and place it in the directory shown in your Anaconda Prompt. If you want to change the name of the environment, you can do this by changing the first line of the file. Then type (in the Anaconda Prompt):

```
conda env create -f environment.yml
```

Then activate your environment:

```
conda activate cellpy
```

`cellpy` relies on a number of other python package and these need to be installed. Most of these packages are included when creating the environment based on the `environment.yml` file as outlined above.

Basic dependencies

In general, you need the typical scientific python pack, including

- `numpy`
- `scipy`
- `pandas`

Additional dependencies are:

- `pytables` is needed for working with the hdf5 files (the cellpy-files):

```
conda install -c conda-forge pytables
```

- `lmfit` is required to use some of the fitting routines in `cellpy`:

```
conda install -c conda-forge lmfit
```

- `holoviz` and `plotly`: plotting library used in several of our example notebooks.
- `jupyter`: used for tutorial notebooks and in general very useful tool for working with and sharing your `cellpy` results.

For more details, I recommend that you look at the documentation of these packages (google it) and install them. You can most likely use the same method as for pytables *etc*.

Additional requirements for .res files

Note! .res files from Arbin testers are actually in a Microsoft Access format.

For Windows users: if you do not have one of the most recent Office versions, you might not be allowed to install a driver of different bit than your office version is using (the installers can be found here). Also remark that the driver needs to be of the same bit as your Python (so, if you are using 32 bit Python, you will need the 32 bit driver).

For POSIX systems: I have not found any suitable drivers. Instead, `cellpy` will try to use `mdbtools`to first export the data to temporary csv-files, and then import from those csv-file (using the `pandas` library). You can install `mdbtools` using your systems preferred package manager (*e.g.* `apt-get install mdbtools`).

The cellpy configuration file

The paths to raw data, the cellpy data base file, file locations etc. are set in the `.cellpy_prms_USER.conf` file that is located in your home directory.

To get the filepath to your config file (and other cellpy info), run:

```
cellpy info -l
```

The config file is written in YAML format and it should be relatively easy to edit it in a text editor.

Within the config file, the paths are the most important parts that need to be set up correctly. This tells `cellpy` where to find (and save) different files, such as the database file and raw data.

Furthermore, the config file contains details about the database-file to be used for cell info and metadata (i.e. type and structure of the database file such as column headers etc.). For more details, see chapter on Configuring cellpy.

The 'database' file

The database file should contain information (cell name, type, mass loading etc.) on your cells, so that cellpy can find and link the test data to the provided metadata.

The database file is also useful when working with the `cellpy` batch routine.

Useful `cellpy` commands

To help installing and controlling your `cellpy` installation, a CLI (command-line-interface) is provided with several commands (including the already mentioned `info` for getting information about your installation, and `setup` for helping you to set up your installation and writing a configuration file).

To get a list of these commands including some basic information, you can issue

```
cellpy --help
```

This will output some (hopefully) helpful text

```
Usage: cellpy [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  edit   Edit your cellpy config file.
  info   This will give you some valuable information about your cellpy.
  new    Set up a batch experiment.
  pull   Download examples or tests from the big internet.
  run    Run a cellpy process.
  serve  Start a Jupyter server
  setup  This will help you to setup cellpy.
```

You can get information about the sub-commands by issuing –-help after them also. For example, issuing

```
cellpy info --help
```

gives

```
Usage: cellpy info [OPTIONS]

Options:
 -v, --version    Print version information.
 -l, --configloc  Print full path to the config file.
 -p, --params     Dump all parameters to screen.
 -c, --check      Do a sanity check to see if things works as they should.
 --help           Show this message and exit.
```

Running your first script

As with most software, you are encouraged to play a little with it. I hope there are some useful stuff in the code repository (for example in the examples folder).

Hint: The `cellpy pull` command can assist in downloading both examples and tests.

Start by trying to import `cellpy` in an interactive Python session. If you have an icon to press to start up the Python in interactive mode, do that (it could also be for example an ipython console or a Jupyter Notebook). You can also start an interactive Python session if you are in your terminal window of command window by just writing `python` and pressing enter. *Hint:* Remember to activate your cellpy (or whatever name you chose) environment.

Once inside Python, try issuing `import cellpy`. Hopefully you should not see any error-messages.

```
Python 3.9.9 | packaged by conda-forge | (main, Dec 20 2021, 02:36:06)
[MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import cellpy
>>>
```

Nothing bad happened this time. If you got an error message, try to interpret it and check if you have skipped any steps in this tutorial. Maybe you are missing the `box` package? If so, go out of the Python interpreter if you started it in your command window, or open another command window and write

```
pip install python-box
```

and try again.

Now let's try to be a bit more ambitious. Start up python again if you are not still running it and try this:

```
>>> from cellpy import prmreader
>>> prmreader.info()
```

The `prmreader.info()` command should print out information about your cellpy settings. For example where you selected to look for your input raw files (`prms.Paths.rawdatadir`).

Try scrolling to find your own `prms.Paths.rawdatadir`. Does it look right? These settings can be changed by either re-running the `cellpy setup -i` command (not in Python, but in the command window / terminal window). You probably need to use the `--reset` flag this time since it is not your first time running it).

Interacting with your data

Read cell data

We assume that we have cycled a cell and that we have two files with results (we had to stop the experiment and re-start for some reason). The files are in the .res format (Arbin).

The easiest way to load data is to use the `cellpy.get` method:

```
import cellpy

electrode_mass = 0.658 # active mass of electrode in mg
file_name = "20170101_ife01_cc_01.res"
cell_data = cellpy.get(file_name, mass=electrode_mass, cycle_mode="anode")
```

Note: Even though the `CellpyCell` object in the example above got the name `cell_data`, it is more common to just simply name it `c` (*i.e.* `c = cellpy.get(...)`). Similarly, the `cellpy` naming convention for `cellpy.utils.Batch` objects is to name them `b` (*i.e.* `b = batch.init(...)` (assuming then that the `batch` module was imported somewhere earlier in the code)).

If you prefer, you can obtain the same by using `cellpy.cellreader.CellpyCell` object directly. However, we recommend using the `cellpy.get` method. But just in case you want to know how to do it...

First, import the cellreader-object from `cellpy`:

```
import os
from cellpy import cellreader
```

Then define some settings and variables and create the CellpyCell-object:

```
raw_data_dir = r"C:\raw_data"
out_data_dir = r"C:\processed_data"
cellpy_data_dir = r"C:\CellpyCell"
cycle_mode = "anode" # default is usually "anode", but...
# These can also be set in the configuration file

electrode_mass = 0.658 # active mass of electrode in mg

# list of files to read (Arbin .res type):
raw_file = ["20170101_ife01_cc_01.res", "20170101_ife01_cc_02.res"]
# the second file is a 'continuation' of the first file...

# list consisting of file names with full path
raw_files = [os.path.join(raw_data_dir, f) for f in raw_file]

# creating the CellpyCell object and set the cycle mode:
cell_data = cellreader.CellpyCell()
cell_data.cycle_mode = cycle_mode
```

Now we will read the files, merge them, and create a summary:

```
# if the list of files are in a list they are automatically merged:
cell_data.from_raw([raw_files])
cell_data.set_mass(electrode_mass)
cell_data.make_summary()
# Note: make_summary will automatically run the
# make_step_table function if it does not exist.
```

Save / export data

When you have loaded your data and created your `CellpyCell` object, it is time to save everything in the cellpy-format:

```
# defining a name for the cellpy_file (hdf5-format)
cellpy_data_dir = r"C:\cellpy_data\cellpy_files"
cellpy_file = os.path.join(cellpy_data_dir, "20170101_ife01_cc2.h5")
cell_data.save(cellpy_file)
```

The cellpy format is much faster to load than the raw-file formats typically encountered. It also includes the summary and step-tables, and it is easy to add more data to the file later on.

To export data to csv format, `CellpyCell` has a method called `to_csv`.

```
# export data to csv
out_data_directory = r"C:\processed_data\csv"
# this exports the summary data to a .csv file:
cell_data.to_csv(out_data_directory, sep=";", cycles=False, raw=False)
# export also the current voltage cycles by setting cycles=True
# export also the raw data by setting raw=True
```

Note: `CellpyCell` objects store the data (including the summary and step-tables) in `pandas DataFrames`. This means that you can easily export the data to other formats, such as Excel, by using the `to_excel` method of the DataFrame object. In addition, `CellpyCell` objects have a method called `to_excel` that exports the data to an Excel file.

More about the `cellpy.get` method

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

The following keyword arguments is current supported by `cellpy.get`:

```
# from the docstring:
Args:
    filename (str, os.PathLike, OtherPath, or list of raw-file names): path to file(s)␣
↪to load
    instrument (str): instrument to use (defaults to the one in your cellpy config file)
    instrument_file (str or path): yaml file for custom file type
    cellpy_file (str, os.PathLike, OtherPath): if both filename (a raw-file) and cellpy_
↪file (a cellpy file)
        is provided, cellpy will try to check if the raw-file is has been updated since␣
↪the
        creation of the cellpy-file and select this instead of the raw file if cellpy␣
↪thinks
        they are similar (use with care!).
    logging_mode (str): "INFO" or "DEBUG"
    cycle_mode (str): the cycle mode (e.g. "anode" or "full_cell")
    mass (float): mass of active material (mg) (defaults to mass given in cellpy-file or␣
↪1.0)
```

(continues on next page)

(continued from previous page)

```
    nominal_capacity (float): nominal capacity for the cell (e.g. used for finding C-
↪rates)
    loading (float): loading in units [mass] / [area]
    area (float): active electrode area (e.g. used for finding the areal capacity)
    estimate_area (bool): calculate area from loading if given (defaults to True)
    auto_pick_cellpy_format (bool): decide if it is a cellpy-file based on suffix.
    auto_summary (bool): (re-) create summary.
    units (dict): update cellpy units (used after the file is loaded, e.g. when creating␣
↪summary).
    step_kwargs (dict): sent to make_steps
    summary_kwargs (dict): sent to make_summary
    selector (dict): passed to load (when loading cellpy-files).
    testing (bool): set to True if testing (will for example prevent making .log files)
    **kwargs: sent to the loader
```

Reading a cellpy file:

```
c = cellpy.get("my_cellpyfile.cellpy")
# or
c = cellpy.get("my_cellpyfile.h5")
```

Reading anode half-cell data from arbin sql:

```
c = cellpy.get("my_cellpyfile", instrument="arbin_sql", cycle_mode="anode")
# Remark! if sql prms are not set in your config-file you have to set them manually (e.g.
↪ setting values in
#    prms.Instruments.Arbin.VAR)
```

Reading data obtained by exporting csv from arbin sql using non-default delimiter sign:

```
c = cellpy.get("my_cellpyfile.csv", instrument="arbin_sql_csv", sep=";")
```

Reading data obtained by exporting a csv file from Maccor using a sub-model (this example uses one of the models already available inside `cellpy`):

```
c = cellpy.get(filename="name.txt", instrument="maccor_txt", model="one", mass=1.0)
```

Reading csv file using the custom loader where the format definitions are given in a user-supplied yaml-file:

```
c = cellpy.get(filename="name.txt", instrument_file="my_custom_file_format.yml")
```

If you specify both the raw file name(s) and the cellpy file name to``cellpy.get`` you can make `cellpy` select whether-or-not to load directly from the raw-file or use the cellpy-file instead. `cellpy` will check if the raw file(s) is/are updated since the last time you saved the cellpy file - if not, then it will load the cellpy file instead (this is usually much faster than loading the raw file(s)). You can also input the masses and enforce that it creates a summary automatically.

```
cell_data.get(
    raw_files=[raw_files],
    cellpy_file=cellpy_file,
    mass=electrode_mass,
    auto_summary=True,
)
```

(continues on next page)

(continued from previous page)

```
if not cell_data.check():
    print("Could not load the data")
```

Working with external files

To work with external files you will need to set some environment variables. This can most easily be done by creating a file called `.env_cellpy` in your user directory (e.g. `C:\Users\jepe`):

```
# content of .env_cellpy
CELLPY_PASSWORD=1234
CELLPY_KEY_FILENAME=C:\\Users\\jepe\\.ssh\\id_key
CELLPY_HOST=myhost.com
CELLPY_USER=jepe
```

You can then load the file using the `cellpy.get` method by providing the full path to the file, including the protocol (e.g. `scp://`) and the user name and host (e.g. `jepe@myhost.com`):

```
# assuming appropriate ``.env_cellpy`` file is present
raw_file = "scp://jepe@myhost.com/path/to/file.txt"
c = cellpy.get(filename=raw_file, instrument="maccor_txt", model="one", mass=1.0)
```

cellpy will automatically download the file to a temporary directory and read it.

Save / export data

Saving data to cellpy format is done by the `CellpyCell.save` method. To export data to csv format, `CellpyCell` has a method called `to_csv`.

```
# export data to csv
out_data_directory = r"C:\processed_data\csv"
# this exports the summary data to a .csv file:
cell_data.to_csv(out_data_directory, sep=";", cycles=False, raw=False)
# export also the current voltage cycles by setting cycles=True
# export also the raw data by setting raw=True
```

Stuff that you might want to do with `cellpy`

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

A more or less random collection of things that you might want to do with `cellpy`. This is not a tutorial, but rather a collection of examples.

Extract current-voltage graphs

If you have loaded your data into a CellpyCell-object, let's now consider how to extract current-voltage graphs from your data. We assume that the name of your CellpyCell-object is `cell_data`:

```
cycle_number = 5
charge_capacity, charge_voltage = cell_data.get_ccap(cycle_number)
discharge_capacity, discharge_voltage = cell_data.get_dcap(cycle_number)
```

You can also get the capacity-voltage curves with both charge and discharge:

```
capacity, charge_voltage = cell_data.get_cap(cycle_number)
# the second capacity (charge (delithiation) for typical anode half-cell experiments)
# will be given "in reverse".
```

The `CellpyCell` object has several get-methods, including getting current, timestamps, etc.

Extract summaries of runs

Summaries of runs includes data pr. cycle for your data set. Examples of summary data is charge- and discharge-values, coulombic efficiencies and internal resistances. These are calculated by the `make_summary` method.

Remark that note all the possible summary statistics are calculated as default. This means that you might have to re-run the `make_summary` method with appropriate parameters as input (e.g. `normalization_cycle`, to give the appropriate cycle numbers to use for finding nominal capacity).

Another method is responsible for investigating the individual steps in the data (`make_step_table`). It is typically run automatically before creating the summaries (since the summary creation depends on the step_table). This table is interesting in itself since it contains delta, minimum, maximum and average values for the measured values pr. step. This is used to find out what type of step it is, *e.g.* a charge-step or maybe a ocv-step. It is possible to provide information to this function if you already knows what kind of step each step is. This saves `cellpy` for a lot of work.

Remark that the default is to calculate values for each unique (step-number - cycle-number) pair. For some experiments, a step can be repeated many times pr. cycle. And if you need for example average values of the voltage for each step (for example if you are doing GITT experiments), you would need to tell `make_step_table` that it should calculate for all the steps (`all_steps=True`).

Create dQ/dV plots

The methods for creating incremental capacity curves is located in the `cellpy.utils.ica` module (utils-ica).

Do some plotting

The plotting methods are located in the `cellpy.utils.plotting` module (utils-plotting).

What else?

There are many things you can do with `cellpy`. The idea is that you should be able to use `cellpy` as a tool to do your own analysis. This means that you need to know a little bit about python and how to use the different modules. It is not difficult, but it requires some playing around and maybe reading some of the source code. Let's keep our fingers crossed and hope that the documentation will be improved in the future.

Why not just try out the highly popular (?) `cellpy.utils.batch` utility. You will need to make (or copy from a friend) the "database" (an excel-file with appropriate headers in the first row) and make sure that all the paths are set up correctly in you *cellpy* configuration file. Then you can process many cells in one go. And compare them.

Or, for example: If you would like to do some interactive plotting of your data, try to install *plotly* and use *Jupyter Lab* to make some fancy plots and dash-boards.

And why not: make a script that goes through all your thousands of measured cells, extracts the life-time (e.g. number of cycles until the capacity has dropped below 80% of the average of the three first cycles), and plot this versus time the cell was put. And maybe color the data-points based on who was doing the experiment?

Configuring cellpy

How the configuration parameters are set and read

When `cellpy` is imported, a default set of parameters is set. Then it tries to read the parameters from your .conf-file (located in your user directory). If successful, the parameters set in your .conf-file will over-ride the default.

The parameters are stored in the module `cellpy.parameters.prms`.

If you would like to change some of the settings during your script (or in your `jupyter notebook`), *e.g.* if you want to use the `cycle_mode` option "cathode" instead of the default "anode", then import the prms class and set new values:

```
from cellpy import parameters.prms

# Changing cycle_mode to cathode
prms.Reader.cycle_mode = 'cathode'

# Changing delimiter to  ',' (used when saving .csv files)
prms.Reader.sep = ','

# Changing the default folder for processed (output) data
prms.Paths.outdatadir = 'experiment01/processed_data'
```

The configuration file

`cellpy` tries to read your .conf-file when imported the first time, and looks in your user directory after files named `.cellpy_prms_SOMENAME.conf`.

If you have run `cellpy setup` in the cmd window or in the shell, the configuration file will be placed in the appropriate place. It will have the name `.cellpy_prms_USERNAME.conf` (where USERNAME is your username).

The configuration file is a YAML-file and it is reasonably easy to read and edit (but remember that YAML is rather strict with regards to spaces and indentations).

As an example, here are the first lines from one of the authors' configuration file:

```
---
Paths:
    outdatadir: C:\scripts\processing_cellpy\out
    rawdatadir: I:\Org\MPT-BAT-LAB\Arbin-data
    cellpydatadir: C:\scripts\processing_cellpy\cellpyfiles
    db_path: C:\scripts\processing_cellpy\db
    filelogdir: C:\scripts\processing_cellpy\logs
    examplesdir: C:\scripts\processing_cellpy\examples
    notebookdir: C:\scripts\processing_cellpy\notebooks
    templatedir: C:\scripting\processing_cellpy\templates
    batchfiledir: C:\scripts\processing_cellpy\batchfiles
    db_filename: 2023_Cell_Analysis_db_001.xlsx
    env_file: .env_cellpy


FileNames:
    file_name_format: YYYYMMDD_[NAME]EEE_CC_TT_RR
```

The first part contains definitions of the different paths, files and file-patterns that `cellpy` will use. This is the place where you most likely will have to do some edits sometime.

The next part contains definitions required when using a database:

```
# settings related to the db used in the batch routine
Db:
    db_type: simple_excel_reader
    db_table_name: db_table
    db_header_row: 0
    db_unit_row: 1
    db_data_start_row: 2
    db_search_start_row: 2
    db_search_end_row: -1

# definitions of headers for the simple_excel_reader
DbCols:
    id:
    - id
    - int
    exists:
    - exists
    - bol
    batch:
    - batch
    - str
    sub_batch_01:
    - b01
    - str
    .
    .
```

This part is rather long (since it needs to define the column names used in the db excel sheet).

The next part contains settings regarding your dataset and the `cellreader`, as well as for the different `instruments`. At the bottom you will find the settings for the `batch` utility.

```
# settings related to your data
DataSet:
    nom_cap: 3579

# settings related to the reader
Reader:
    Reader:
        diagnostics: false
        filestatuschecker: size
        force_step_table_creation: true
        force_all: false
        sep: ;
        cycle_mode: anode
        sorted_data: true
        select_minimal: false
        limit_loaded_cycles:
        ensure_step_table: false
        voltage_interpolation_step: 0.01
        time_interpolation_step: 10.0
        capacity_interpolation_step: 2.0
        use_cellpy_stat_file: false
        auto_dirs: true

# settings related to the instrument loader
# (each instrument can have its own set of settings)
Instruments:
    tester: arbin
    custom_instrument_definitions_file:

Arbin:
    max_res_filesize: 1000000000
    chunk_size:
    max_chunks:
    use_subprocess: false
    detect_subprocess_need: false
    sub_process_path:
    office_version: 64bit
    SQL_server: localhost
    SQL_UID:
    SQL_PWD:
    SQL_Driver: ODBC Driver 17 for SQL Server
    odbc_driver:
Maccor:
    default_model: one

# settings related to running the batch procedure
Batch:
    fig_extension: png
    backend: bokeh
    notebook: true
    dpi: 300
    markersize: 4
    symbol_label: simple
```

(continues on next page)

(continued from previous page)

```
    color_style_label: seaborn-deep
    figure_type: unlimited
    summary_plot_width: 900
    summary_plot_height: 800
    summary_plot_height_fractions:
    - 0.2
    - 0.5
    - 0.3
...
```

As you can see, the author of this particular file most likely works with silicon as anode material for lithium ion batteries (the `nom_cap` is set to 3579 mAh/g, *i.e.* the theoretical gravimetric lithium capacity for silicon at normal temperatures) and is using windows.

By the way, if you are wondering what the '.' means... it means nothing - it was just something I added in this tutorial text to indicate that there is more stuff in the actual file than what is shown here.

Working with the `pandas.DataFrame` objects directly

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

The `CellpyCell` object stores the data in several `pandas.DataFrame` objects. The easies way to get to the DataFrames is by the following procedure:

```
# Assumed name of the CellpyCell object: c

# get the 'test':
data = c.data
# data is now a cellpy Data object (cellpy.readers.cellreader.Data)

# pandas.DataFrame with data vs cycle number (coulombic efficiency, charge-capacity etc.
↪):
summary_data = data.summary
# you could also get the summary data by:
summary_data = c.data.summary

# pandas.DataFrame with the raw data:
raw_data = data.raw

# pandas.DataFrame with statistics on each step and info about step type:
step_info = data.steps
```

You can then manipulate your data with the standard `pandas.DataFrame` methods (and `pandas` methods in general).

Happy pandas-ing!

Data mining / using a database

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

One important motivation for developing the `cellpy` project is to facilitate handling many cell testing experiments within a reasonable time and with a "tunable" degree of automation. It is therefore convenient to be able to couple both the meta-data (look-up) to some kind of data-base, as well as saving the extracted key parameters to either the same or another database (where I recommend the latter). The database(s) will be a valuable asset for further data analyses (either using statistical methods, e.g. Bayesian modelling, or as input to machine learning algorithms, for example deep learning using cnn).

Meta-data database

TODO.

Parameters and feature extraction

TODO.

Bayesian modelling

TODO.

Example: reinforcement deep learning (resnet)

TODO.

The cellpy command

To assist in using `cellpy` more efficiently, a set of routines are available from the command line by issuing the `cellpy` command at the shell (or in the cmd window).

```
$ cellpy
Usage: cellpy [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  edit   Edit your cellpy config or database files.
  info   This will give you some valuable information about your cellpy.
  new    Set up a batch experiment (might need git installed).
  pull   Download examples or tests from the big internet (needs git).
  run    Run a cellpy process (e.g.
  serve  Start a Jupyter server.
  setup  This will help you to set up cellpy.
```

The cli is still under development (cli stands for command-line-interface, by the way). Both the `cellpy new` and the `cellpy serve` command worked the last time I tried them. But it might not work on your computer. If you run into problems, let us know.

Information

A couple of commands are implemented to get some information about your `cellpy` environment (currently getting your `cellpy` version and the location of your configuration file):

```
$ cellpy info --version
[cellpy] version: 0.4.1

$ cellpy info --configloc
[cellpy] -> C:\Users\jepe\.cellpy_prms_jepe.conf
```

Setting up `cellpy` from the cli

To get the most out of `cellpy` it is to best to set it up properly. To help with this, you can use the `setup` command. If you include the `--interactive` switch, you will be prompted for your preferred location for the different folders / directories `cellpy` uses (it still will work without them, though).

```
$ cellpy setup --interactive
```

The command will create a starting `cellpy` configuration file (*,cellpy_prms_USERNAME.conf*) or update it if it exists, and create the following directory structure:

```
batchfiles/
cellpyfiles/
db/
examples/
instruments/
logs/
notebooks/
out/
raw/
templates/
```

Note: It is recommended to rerun setup each time you update `cellpy`.

Note: You can get help for each sub-command by turning on the `--help` switch. For example, for `setup`:

```
$ cellpy setup --help
```

You will then get some more detailed information on the different switches you have at your disposal:

```
Usage: cellpy setup [OPTIONS]

  This will help you to setup cellpy.

```

(continues on next page)

(continued from previous page)

```
Options:
  -i, --interactive       Allows you to specify div. folders and setting.
  -nr, --not-relative     If root-dir is given, put it directly in the root
                          (/) folder i.e. do not put it in your home directory.
                          Defaults to False. Remark that if you specifically
                          write a path name instead of selecting the suggested
                          default, the path you write will be used as is.
  -dr, --dry-run          Run setup in dry mode (only print - do not execute).
                          This is typically used when developing and testing
                          cellpy. Defaults to False.
  -r, --reset             Do not suggest path defaults based on your current
                          configuration-file
  -d, --root-dir PATH     Use custom root dir. If not given, your home
                          directory will be used as the top level where
                          cellpy-folders will be put. The folder path must
                          follow directly after this option (if used).
                          Example: $ cellpy setup -d 'MyDir'
  -n, --folder-name PATH
  -t, --testuser TEXT     Fake name for fake user (for testing)
  --help                  Show this message and exit.
```

The cellpy templating system

If you are performing the same type of data processing for many cells, and possibly many times, it is beneficial to start out with a template.

Currently, `cellpy` provides a template system defaulting to a set of `Jupyter notebooks` and a folder structure where the code is based on the `batch` utility (`cellpy.utils.batch`).

The templates are pulled from the *cellpy_templates* repository. It uses `cookiecutter` under the hood (and therefore needs *git* installed).

This repository contains several template sets. The default is named *standard*, but you can set another default in your configuration file.

You can also make your own templates and store them locally on your computer (in the *templates* directory). The template should be in a zip file and start with "cellpy_template" and end with ".zip".

```
$ cellpy new --help


Usage: cellpy new [OPTIONS]

  Set up a batch experiment (might need git installed).

Options:
  -t, --template TEXT        Provide template name.
  -d, --directory TEXT       Create in custom directory.
  -p, --project TEXT         Provide project name (i.e. sub-directory name).
  -e, --experiment TEXT      Provide experiment name (i.e. lookup-value).
  -u, --local-user-template  Use local template from the templates directory.
  -s, --serve                Run Jupyter.
```

(continues on next page)

(continued from previous page)

```
  -r, --run                   Use PaperMill to run the notebook(s) from the
                              template (will only work properly if the
                              notebooks can be sorted in correct run-order by
                              'sorted'.
  -j, --lab                   Use Jupyter Lab instead of Notebook when serving.
  -l, --list                  List available templates and exit.
  --help                      Show this message and exit.
```

Automatically running batches

The run command is used for running the appropriate editor for your database, and for running (processing) files in batches.

```
$ cellpy run --help

Usage: cellpy run [OPTIONS] [NAME]

  Run a cellpy process (batch-job, edit db, ...).
  You can use this to launch specific applications.

  Examples:

      edit your cellpy database

         cellpy run db

      run a batch job described in a journal file

         cellpy run -j my_experiment.json

Options:
  -j, --journal         Run a batch job defined in the given journal-file
  -k, --key             Run a batch job defined by batch-name
  -f, --folder          Run all batch jobs iteratively in a given folder
  -p, --cellpy-project  Use PaperMill to run the notebook(s) within the given
                        project folder (will only work properly if the
                        notebooks can be sorted in correct run-order by
                        'sorted'). Warning! since we are using `click` - the
                        NAME will be 'converted' when it is loaded (same as
                        print(name) does) - so you can't use backslash ('\')
                        as normal in windows (use either '/' or '\\' instead).
  -d, --debug           Run in debug mode.
  -s, --silent          Run in silent mode.
  --raw                 Force loading raw-file(s).
  --cellpyfile          Force cellpy-file(s).
  --minimal             Minimal processing.
  --nom-cap FLOAT       nominal capacity (used in calculating rates etc)
  --batch_col TEXT      batch column (if selecting running from db)
  --project TEXT        name of the project (if selecting running from db)
  -l, --list            List batch-files.
  --help                Show this message and exit.
```

5.2.2 Using some of the cellpy special utilities

Open Circuit Relaxation

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

Plotting selected Open Circuit Relaxation points

```
# assuming b is a cellpy.utils.batch.Batch object

from cellpy.utils.batch_tools.batch_analyzers import OCVRelaxationAnalyzer
# help(OCVRelaxationAnalyzer)

print(" analyzing ocv relaxation data ".center(80, "-"))
analyzer = OCVRelaxationAnalyzer()
analyzer.assign(b.experiment)
analyzer.direction = "down"
analyzer.do()
dfs = analyzer.last
df_file_one, _df_file_two = dfs

# keeping only the columns with voltages
ycols = [col for col in df_file_one.columns if col.find("point")>=0]

# removing the first ocv rlx (relaxation before starting cycling)
df = df_file_one.iloc[1:, :]
# tidy format
df = df.melt(
    id_vars = "cycle", var_name="point", value_vars=ycols,
    value_name="voltage"
)

# using holoviews for plotting
curve = hv.Curve(
    df, kdims=["cycle", "point"],
    vdims="voltage"
).groupby("point").overlay().opts(xlim=(1,10), width=800)
```

Open Circuit Relaxation modeling

TODO.

Extracting ica data

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

Example: get dq/dv data for selected cycles

This example shows how to get dq/dv data for a selected cycle. The data is extracted from the cellpy object using the .get_cap method. The method `dqdv_cycle` and the `dqdv` methods are *cellpy-agnostic* and should work also for other data sources (e.g. numpy arrays).

```
import matplotlib.pyplot as plt
from cellpy.utils import ica

v4, dqdv4 = ica.dqdv_cycle(
    data.get_cap(
        4,
        categorical_column=True,
        method = "forth-and-forth")
)

v10, dqdv10 = ica.dqdv_cycle(
    data.get_cap(
        10,
        categorical_column=True,
        method = "forth-and-forth")
)

plt.plot(v4,dqdv4, label="cycle 4")
plt.plot(v10, dqdv10, label="cycle 10")
plt.legend()
```

Example: get dq/dv data for selected cycles

This example shows how to get dq/dv data directly from the cellpy object. This methodology is more convenient if you want to get the data easily from the cellpy object in a pandas dataframe format.

```
# assuming that b is a cellpy.utils.batch.Batch object

c = b.experiment.data["20160805_test001_45_cc"]  # get the cellpy object
tidy_ica = ica.dqdv_frames(c)  # create the dqdv data
cycles = list(range(1,3)) + [10, 15]
tidy_ica = tidy_ica.loc[tidy_ica.cycle.isin(cycles), :]  # select cycles
```

Fitting ica data

TODO.

Using the batch utilities

The steps given in this tutorial describes how to use the new version of the batch utility. The part presented here is chosen such that it resembles how the old utility worked. However, under the hood, the new batch utility is very different from the old. A more detailed guide will come soon.

So, with that being said, here is the promised description.

Starting (setting things up)

A database

Currently, the only supported "database" is Excel (yes, I am not kidding). So, there is definitely room for improvements if you would like to contribute to the code-base.

The Excel work-book must contain a page called `db_table`. And the top row of this page should consist of the correct headers as defined in your cellpy config file. You then have to give an identification name for the cells you would like to investigate in one of the columns labelled as batch columns (typically "b01", "b02", ..., "b07"). You can find an example of such an Excel work-book in the test-data.

A tool for running the job

Jupyter Notebooks is the recommended "tool" for running the cellpy batch feature. The first step is to import the `cellpy.utils.batch.Batch` class from `cellpy`. The `Batch` class is a utility class for pipe-lining batch processing of cell cycle data.

```
from cellpy.utils import batch, plotutils
from cellpy import prms
from cellpy import prmreader
```

The next step is to initialize it:

```
project = "experiment_set_01"
name = "new_exiting_chemistry"  # the name you set in the database
batch_col = "b01"
b = batch.init(name, project, batch_col=batch_col)
```

and set some parameters that *Batch* needs:

```
# setting additional parameters if the defaults are not to your liking:
b.experiment.export_raw = True
b.experiment.export_cycles = True
b.experiment.export_ica = True
b.experiment.all_in_memory = True  # store all data in memory, defaults to False
b.save_cellpy_file = True

b.force_raw_file = False
b.force_cellpy_file = True
```

Extracting meta-data

The next step is to extract and collect the information needed from your data-base into a DataFrame, and create an appropriate folder structure (*outdir/project_name/batch_name/raw_data*)

```
# load info from your db and write the journal pages
b.create_journal()
b.create_folder_structure()
```

Processing data

To run the processing, you should then use the convenience function `update`. This function loads all your data-files and saves csv-files of the results.

```
b.update()
```

The next step is to create some summary csv-files (*e.g.* containing charge capacities *vs.* cycle number for all your data-files) and plot the results.

```
b.make_summaries()
b.plot_summaries()
```

Now it is time to relax and maybe drink a cup of coffee.

Further investigations and analyses

There are several paths to go from here. I recommend looking at the raw data for the different cells briefly to check if everything looks sensible. You can get the names of the different datasets (cells) by issuing:

```
b.experiment.cell_names
```

You can get the CellpyCell-object for a given cell by writing:

```
cell = b.experiment.data[name_of_cell]
plotutils.raw_plot(my_cell)
```

If you want to investigate further, you can either use one of the available analysis-engines (they work on batch objects processing all the cells at once) or you can continue on a single cell basis (latter is currently recommended).

Another tip is to make new Notebooks for each type of "investigation" you would like to perform. You can load the info-df-file you created in the initial steps, or you could load the individual cellpy-files (if you did not turn off automatic saving to cellpy-format).

You should be able to find examples of processing either by downloading the examples or by looking in the repo.

Collectors

Todo.

Have a look at the data

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

Here are some examples how to get a peak at the data. If we need an interactive plot of the raw-data, we can use the `plotutils.raw_plot` function. If we would like to see some statistics for some of the cycles, the `plotutils.cycle_info_plot` is your friend. Let´s start by importing cellpy and the `plotutils` utility:

```
import cellpy
from cellpy.utils import plotutils
```

Let´s load some data first:

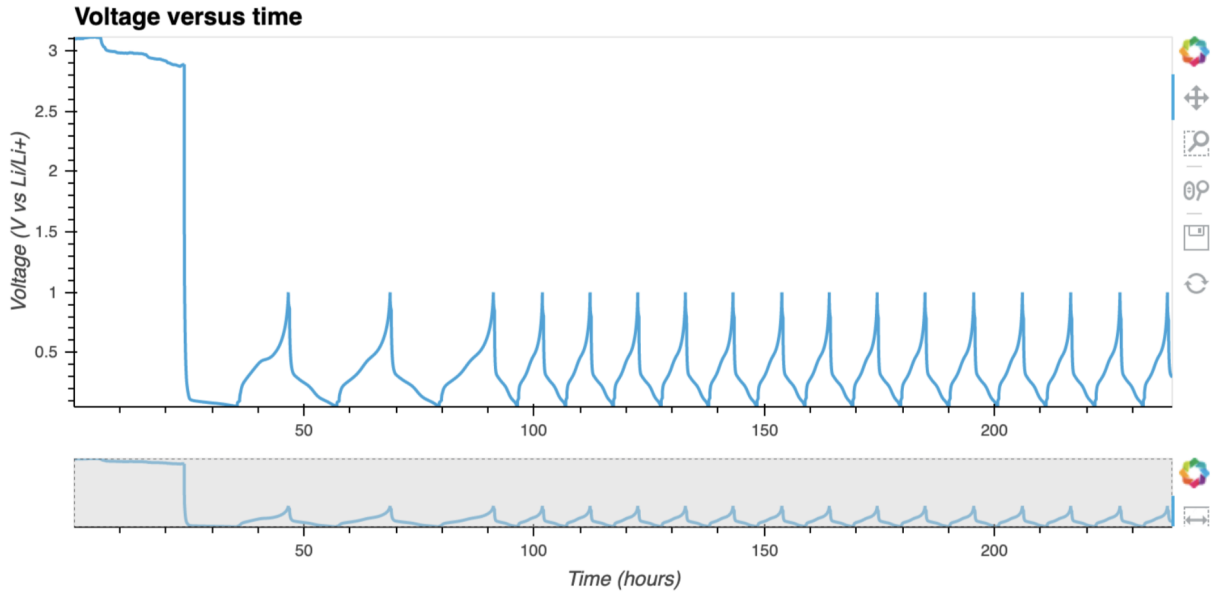
```
cell = cellpy.get("../testdata/hdf5/20160805_test001_45_cc.h5", mass=0.8)
```

Here we used the convenience method `cellpy.get` to load some example data. If everything went well, you will see an output approximately like this:

```
(cellpy) - Making CellpyCell class and setting prms
(cellpy) - Loading cellpy-file: ../testdata/hdf5/20160805_test001_45_cc.h5
(cellpy) - Setting mass: 0.8
(cellpy) - Creating step table
(cellpy) - Creating summary data
(cellpy) - assuming cycling in anode half-cell (discharge before charge) mode
(cellpy) - Created CellpyCell object
```

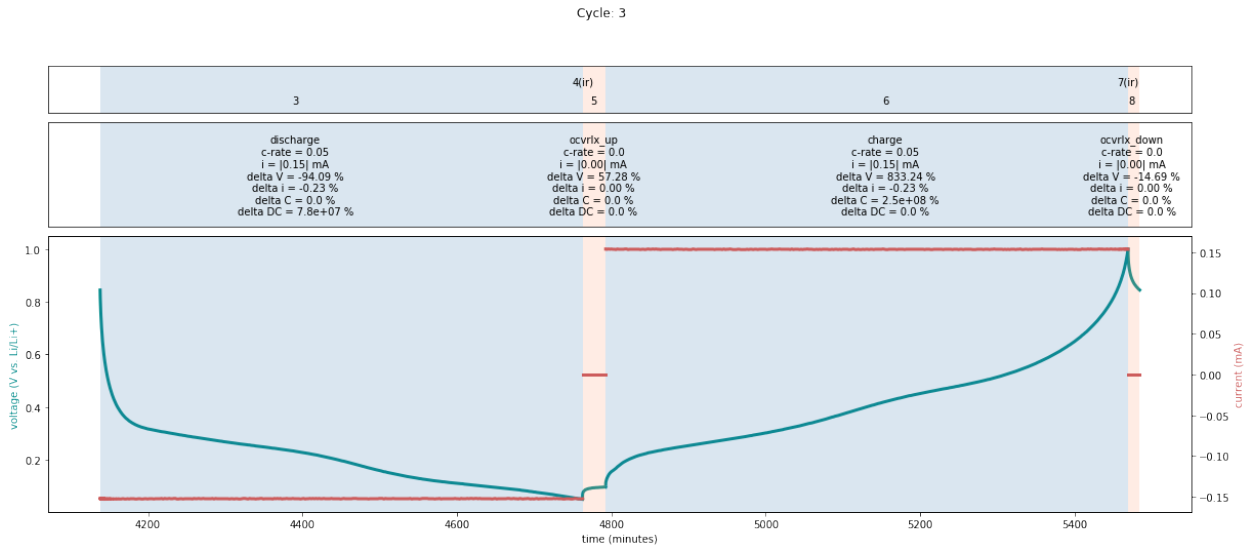
If you have `holoviews` installed, you can conjure an interactive figure:

```
plotutils.raw_plot(cell)
```



Sometimes it is necessary to have a look at some statistics for each cycle and step. This can be done using the `cycle_info_plot` method:

```
fig = plotutils.cycle_info_plot(
    cell,
    cycle=3,
    use_bokeh=False,
)
```



Note: If you chose to work within a Jupyter Notebook, you are advised to try some of the web-based plotting tools. For example, you might consider installing holoviz suite.

Easyplot

Todo.

Templates

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

Some info can be found in one of the example notebooks (*Example notebooks*). It would be more logical if that information was here instead. Sorry for that.

Custom loaders

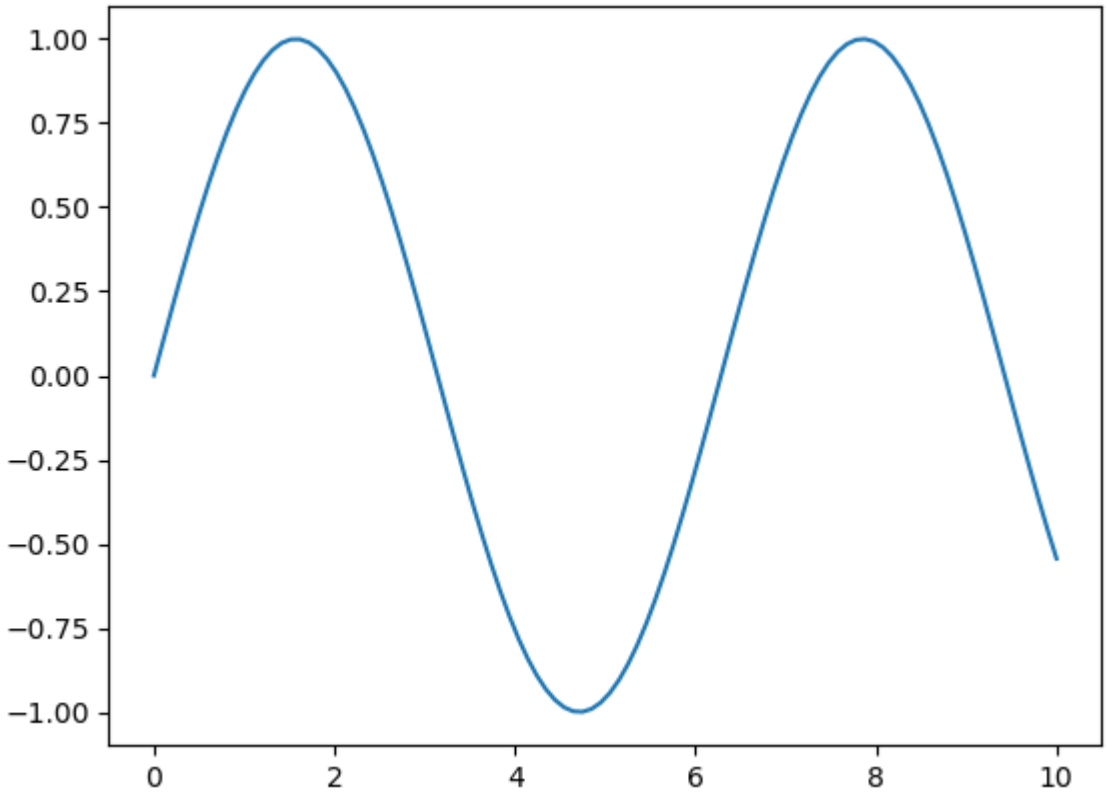
Todo.

5.2.3 Example notebooks

How to make a very simple plot

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.plot(x, y);
```



[ ]:

How to use `get_cap` to select and plot voltage curves

[1]:
```
import matplotlib.pyplot as plt
import numpy as np

import cellpy
from cellpy.utils import example_data
```

[2]:
```
c = example_data.cellpy_file()
```

```
(cellpy) - cycle mode not found
```

[3]:
```
c.get_cycle_numbers()
```

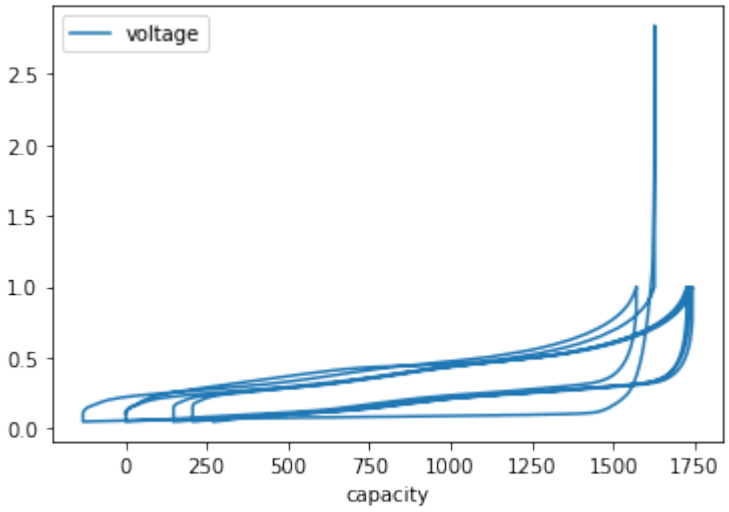
[3]:
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36], dtype=int64)
```

[4]:
```
curves1 = c.get_cap(cycle=[1, 2, 3, 4, 5])
```

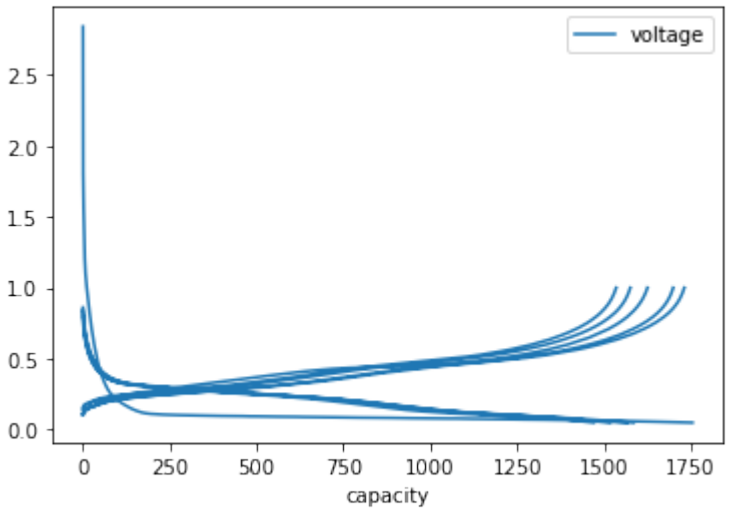
[5]:
```
curves1.head()
```

[5]:
```
      voltage  capacity
1068  0.110245  0.000003
1069  0.115479  0.278791
1070  0.120714  1.425499
1071  0.125948  3.227624
1072  0.131183  5.280277
```

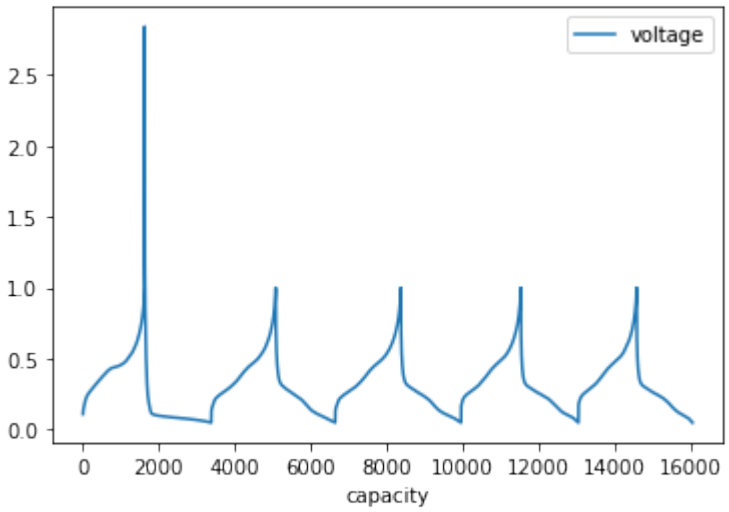
[6]:
```
curves1.plot(x="capacity", y="voltage");
```



[7]:
```
curves2 = c.get_cap(cycle=[1, 2, 3, 4, 5], method="forth-and-forth")
curves2.plot(x="capacity", y="voltage");
```



[8]:
```
curves3 = c.get_cap(cycle=[1, 2, 3, 4, 5], method="forth")
curves3.plot(x="capacity", y="voltage");
```



[9]:
```
import hvplot.pandas
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[10]:
```
curves4 = c.get_cap(
    cycle=[1, 2, 3, 10, 20], method="back-and-forth", label_cycle_number=True
)
curves4.head()
```

[10]:
```
      cycle   voltage  capacity
1068      1  0.110245  0.000003
1069      1  0.115479  0.278791
1070      1  0.120714  1.425499
1071      1  0.125948  3.227624
1072      1  0.131183  5.280277
```

[11]:
```
curves4.hvplot(x="capacity", y="voltage", by="cycle", width=500)
```

[11]:
```
:NdOverlay   [cycle]
   :Curve   [capacity]   (voltage)
```

[ ]:

The cellpy templating system

Often the steps you have to go through when analysing your cells are a bit repetetive. To help facilitate a smoother workflow, cellpy provides a templating system using cookiecutter in the background. The publicly available templates are located in a repository on github (https://github.com/jepegit/cellpy_cookies).

How to start a session and set-up a project using a template

In a command window / shell in the python or conda environment where cellpy is installed, write:

```
> cellpy new
```

cellpy will then list all the directories ("projects") in your project directory (typically called "out") and you must choose one of them (write one of the numbers, or go for the default):

```
Template: standard
Select project folder:
1 - [create new dir]
2 - Internal
3 - SuperBatteries
4 - AmazingCell
5 - MoZEES
6 - SIMBA
7 - LongLife
8 - MoreIsLess
9 - MoZEES
Choose from 1, 2, 3, 4, 5, 6, 7, 8, 9 [1]:
```

If you press enter without writing a number, it will select the default, whatever is written in the square brackets. The projects listed will be different for you than it is here. If this is the first time using cellpy, most likely only the option 1 exists.

For example, if we would like to look at some cells belonging the the MoreIsLess project, we write 8 and press Enter. Next cellpy needs you to confirm that the project name is correct (shown in square brackets) by pressing Enter again. If you want to change the name, you can also give it here.

```
Choose from 1, 2, 3, 4, 5, 6, 7, 8, 9 [1]: 8
project_name [MoreIsLess]:
```

However, in this example we will chose to create a new directory ("project") by selecting the default (by just pressing enter).

```
Choose from 1, 2, 3, 4, 5, 6, 7, 8, 9 [1]:
New name [cellpy_project]:
```

Write my_project and press Enter. Confirm the project name by pressing Enter once more:

```
New name [cellpy_project]: my_project
created my_project
project_name [my_project]:
```

Next we need to provide a session id (for example the batch name used in your db). Here we chose to call it experiment_001:

```
session_id: experiment_001
```

`cellpy` then asks you to confirm the version of `cellpy` you are using (not all templates works with all versions), author name (your user name most likely), and date. Accept all by pressing Enter for each item.

```
cellpy_version [1.0.0a]:
author_name [jepe]:
date [2023-01-18]:
```

Next `cellpy` will suggest a name for the directory / folder that the template will create, as well as the "core" name of the notebook(s) (or script). Once again, confirm the by pressing Enter:

```
experiment_folder_name [2023_01_18_experiment_001]:
notebook_name [experiment_001]:
```

Under the hood, `cellpy` uses `cookiecutter` to download and create the structure from the template:

```
   Cellpy Cookie says: 'using cookie from the cellpy_cookies repository'
   Cellpy Cookie says: 'setting up project in the following directory: C:\cellpy_data\
↪notebooks\my_project\2023_01_18_experiment_001'
created:
2023_01_18_experiment_001/
    00_experiment_001_notes_home.ipynb
    01_experiment_001_loader.ipynb
    02_experiment_001_life.ipynb
    03_experiment_001_cycles.ipynb
    04_experiment_001_ica.ipynb
    05_experiment_001_plots.ipynb
    data/
        external/
        interim/
        processed/
        raw/
    out/
        note.md
```

Congratulations - you have succesfully created a directory structure and several `jupyter` notebooks you can use for processing your cell data!

Run the notebooks

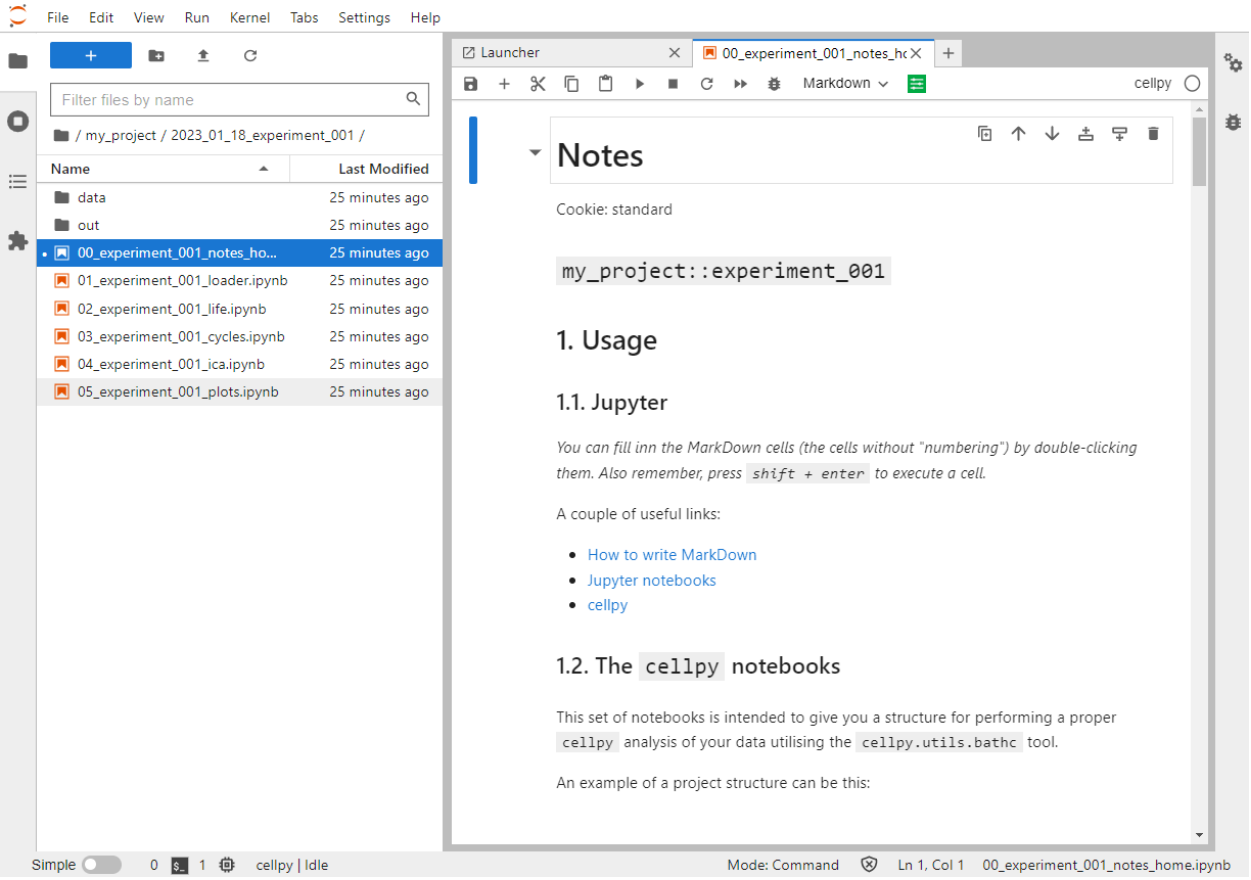
Start `jupyter` (either jupyter notebook or jupyterlab). If you want, there exists a `cellpy` command for this as well. The benefit is that it starts up the `jupyter` server in the correct directory. Obviously, this will not work if you dont have `jupyter` properly installed.

In a command window / shell in the python or conda environment where `cellpy` is installed, write:

```
> cellpy serve -l
```

The `-l` option means that you want to run `jupyterlab`. You can also tell `cellpy` to automatically start `jupyter` after it has finished populating stuff from the template by issuing the `-s` option when you run the `new` sub-command.

Hopefully you will see something like this:



More in-depth on the templating system

Get some help:

```
> cellpy new --help



Usage: cellpy new [OPTIONS]

  Set up a batch experiment (might need git installed).

Options:
  -t, --template TEXT         Provide template name.
  -d, --directory TEXT        Create in custom directory.
  -p, --project TEXT          Provide project name (i.e. sub-directory name).
  -e, --experiment TEXT       Provide experiment name (i.e. lookup-value).
  -u, --local-user-template   Use local template from the templates directory.
  -s, --serve                 Run Jupyter.
  -r, --run                   Use PaperMill to run the notebook(s) from the
                              template (will only work properly if the
                              notebooks can be sorted in correct run-order by
                              'sorted'.
  -j, --lab                   Use Jupyter Lab instead of Notebook when serving.
```

(continues on next page)

(continued from previous page)

```
  -l, --list                List available templates and exit.
  --help                    Show this message and exit.
```

Assume you have the following inside your templates folder (cellpy_data:nbsphinx-math:*templates*):

```
> ls
cellpy_cookie_advanced.zip  cellpy_cookie_easy.zip  cellpy_cookie_pytorch.zip  just_a_
↪regular_zipped_folder.zip
```

You can list the available templates using the `--list` option:

```
> cellpy new --list

[cellpy] batch templates
[cellpy] - default: standard
[cellpy] - registered templates (on github):
                standard          ('https://github.com/jepegit/cellpy_cookies.git',
↪'standard')
                ife               ('https://github.com/jepegit/cellpy_cookies.git', 'ife
↪')
[cellpy] - local templates (C:\cellpy_data\templates):
                advanced          ('C:\\cellpy_data\\templates\\cellpy_cookie_advanced.
↪zip', None)
                easy              ('C:\\cellpy_data\\templates\\cellpy_cookie_easy.zip',
↪ None)
                pytorch           ('C:\\cellpy_data\\templates\\cellpy_cookie_pytorch.
↪zip', None)
```

Your local templates needs to adhere to the following structure:

- it must be in a zipped file (.zip)
- the zip-file needs to start with "cellpy_cookie" and contain one top level folder with the same name
- the top folder can contain several templates
- each "sub-template" must contain a cookiecutter configuration file (you can copy-paste from the github repository to get a starting point for editing)
- you will also need the "hooks" folder with the "post_gen_project.py" and the "pre_gen_project.py" files (copy-paste them from the github repository and modify if needed)
- `cookiecutter` variables are enclosed in double curly brackets

Here is an example (content of the zip file "cellpy_cookie_standard.zip"):

```
└───cellpy_cookie_standard
    └───standard
        │   cookiecutter.json
        │
        ├───hooks
        │       post_gen_project.py
        │       pre_gen_project.py
        │
        └───{{cookiecutter.experiment_folder_name}}
            │   00_{{cookiecutter.notebook_name}}_notes.ipynb
```

(continues on next page)

(continued from previous page)

```
    01_{{cookiecutter.notebook_name}}_loader.ipynb
    02_{{cookiecutter.notebook_name}}_life.ipynb
    03_{{cookiecutter.notebook_name}}_cycles.ipynb
    04_{{cookiecutter.notebook_name}}_ica.ipynb
    05_{{cookiecutter.notebook_name}}_plots.ipynb

├──data
    ├──external

    ├──interim

    ├──processed

    └──raw

└──out
        note.md
```

5.2.4 Examples

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

Look in the examples folder in the GitHub repository for more examples. You can also download examples using the `cellpy pull --examples` command.

5.2.5 Some tips and tricks

Overriding settings

If you would like to override some of the standard settings, then the route of less friction is to import the prms class and set new values directly.

Below is a simple example where we set an option so that we can load arbin-files (.res) using mdbtools (i.e. spawning a subprocess that converts the .res-file to .csv-files, then loading and deleting the .csv-files):

```
from cellpy import prms

# use mdbtools even though you are on windows
prms.Instruments.Arbin.use_subprocess = True
```

Another typical value you might want to change (but not permanently by setting its value in the config file) is the cell-type ("anode" or something else) and informing about the "quality" of your raw-data.

```
from cellpy import prms

# I have set "anode" as default in my config-file, but this time
# I am going to look at a cathode
prms.Reader.cycle_mode = "cathode"
```

(continues on next page)

(continued from previous page)

```
# I don't trust that my data are sorted
prms.Reader.sorted_data = False

# I am using a "strange" version of an cell-tester / software setup
# that don't make any stat file (last "reading" for each cycle).
prms.Reader.use_cellpy_stat_file = False
```

Another typical setting you might want to change during a session (and doing it without having to edit your config file) is the input and output directories.

```
from cellpy import prms

# use an temporary folder for playing around and testing stuff
prms.Paths.rawdatadir = r"C:\tmp"
prms.Paths.cellpydatadir = r"C:\tmp"
prms.Paths.filelogdir = r"C:\tmp"
```

More tips and tricks will come soon!

5.2.6 Loaders

Arbin

TODO.

Maccor

TODO.

PEC

TODO.

Neware

TODO.

Biologics

TODO.

Custom

TODO.

5.3 Developers guide

5.3.1 Getting ready to develop

Get the code

Details of how to get the code can be found in the `contributing` chapter. If you are reading this, you have probably already done this step.

You also need to work a little bit within the terminal. If you are not familiar with this, you should try to get some basic knowledge on that before you continue.

And, you will need to have a IDE (integrated development environment) installed. Good alternatives are PyCharm (https://www.jetbrains.com/pycharm/) and VSCode (https://code.visualstudio.com/).

Install python 3.9 or higher

There are many ways to install python. We recommend using the Miniconda distribution. You can download it from here: https://docs.conda.io/en/latest/miniconda.html

> From the conda website: *"Miniconda is a free minimal installer for conda. It is a small, bootstrap version of Anaconda that includes only conda, Python, the packages they depend on, and a small number of other useful packages, including pip, zlib and a few others."*

The main benefit of using conda is that it makes it easy to install and manage packages and environments. It also makes it easy to install packages that are not available on PyPI (the Python Package Index), including for example `pandoc` which is needed to build the documentation.

Create a development environment

Once you have installed conda, you should create a new environment for developing `cellpy`. Take a look at the documentation for conda to learn more about environments and how to use them if you are not familiar with this concept.

The easiest way to create a working environment is to use the `dev_environment.yml` file in the root of the `cellpy` repository. This will generate an environment called `cellpy_dev` with all the packages needed to develop `cellpy`. You can create the environment by running the following command in a terminal:

```
conda env create -f dev_environment.yml
```

You can then activate the environment by running:

```
conda activate cellpy_dev
```

Next, you need to install `cellpy` in development mode. This is done by running the following from the root folder of `cellpy`:

```
python -m pip install -e . # the dot is important!
```

Installing `cellpy` in development mode means that you can edit the code and the changes will be reflected when you import `cellpy` in python. This is very useful when developing code, since you can test your changes without having to reinstall the package every time you make a change.

As a final check, you should check if the tests don't fail (still being in the root directory of `cellpy`):

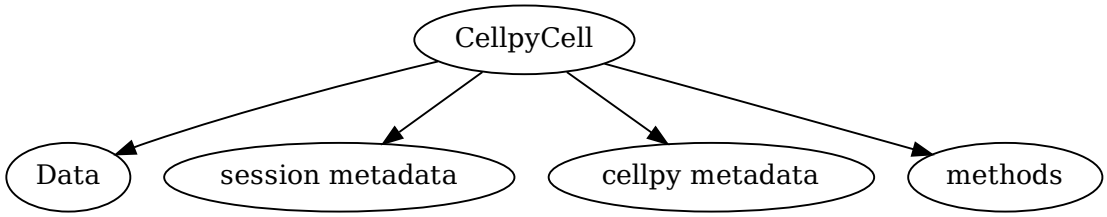
```
pytest .
```

Fingers crossed! If the tests pass, you are ready to start developing!

5.3.2 Main `cellpy` objects

Note: This chapter would benefit from some more love and care. Any help on that would be highly appreciated.

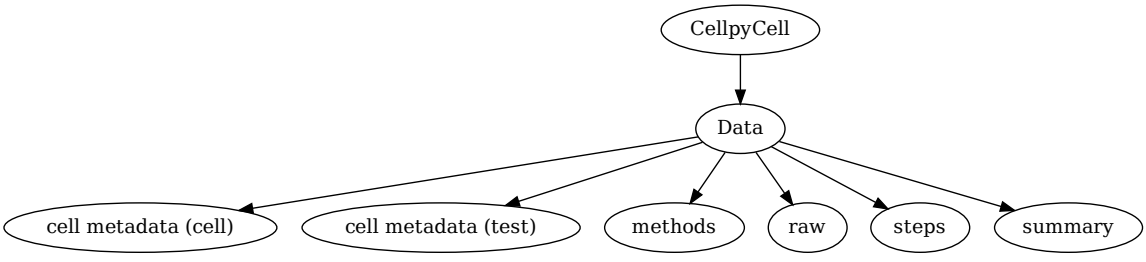
The `CellpyCell` object



The `CellpyCell` object contains the main methods as well as the actual data:

```
cellpy_instance = CellpyCell(...)
```

Data is stored within an instance of the `Data` class.



The `Data` instance can be reached using the `data` property:

```
cell_data = cellpy_instance.data
```

The `Data` object

The `Data` object contains the data and the meta-data for the cell characterisation experiment(s).

The cellpy file format

As default, cellpy stores files in the hdf5 format.

5.3.3 Structure of the `cellpy` package

Folder structure

The `cellpy` repository is structured as follows:

```
cellpy               # the main folder for the cellpy repository
├──  .github         # github specific files (e.g. github actions)
├──  bin              # binary files (mdbtools for win)
├──  cellpy           # the main folder for the cellpy package
├──  docs             # the main folder for the cellpy documentation
├──  examples
├──  test_journal     # data etc for the tests
├──  testdata         # data etc for the tests
├──  tests            # here are the tests
├──  .coverage
├──  .env_example
├──  .gitattributes
├──  .gitignore
├──  .readthedocs.yaml
├──  bumpver.toml
├──  appveyor.yml
├──  AUTHORS.rst        <-- picked up by sphinx (in docs)
├──  README.rst         <-- picked up by sphinx (in docs)
├──  CODE_OF_CONDUCT.md
├──  CONTRIBUTING.rst   <-- picked up by sphinx (in docs)
├──  HISTORY.rst        <-- picked up by sphinx (in docs)
├──  LICENSE            <-- picked up by sphinx (in docs)
├──  MANIFEST.in
├──  notes.md           <-- log of notes
├──  pyproject.toml
├──  dev_environment.yml
├──  environment.yml
├──  requirements_dev.txt
├──  requirements.txt
├──  noxfile.py
├──  setup.py
└──  tasks.py           <-- invoke tasks
```

The `cellpy` source code is structured as follows:

```
cellpy\cellpy
├──  internals
│   ├──  __init__.py
```

(continues on next page)

(continued from previous page)

```
│   └── core.py
├── parameters
│   ├── legacy
│   │   ├── __init__.py
│   │   └── update_headers.py
│   ├── .cellpy_prms_default.conf
│   ├── __init__.py
│   ├── internal_settings.py
│   ├── prmreader.py
│   └── prms.py
├── readers
│   ├── instruments
│   │   ├── .benchmarks
│   │   ├── configurations
│   │   │   ├── __init__.py
│   │   │   ├── maccor_txt_four.py
│   │   │   ├── maccor_txt_one.py
│   │   │   ├── maccor_txt_three.py
│   │   │   ├── maccor_txt_two.py
│   │   │   ├── maccor_txt_zero.py
│   │   │   └── neware_txt_zero.py
│   │   ├── loader_specific_modules
│   │   │   ├── __init__.py
│   │   │   └── biologic_file_format.py
│   │   ├── processors
│   │   │   ├── __init__.py
│   │   │   ├── post_processors.py
│   │   │   └── pre_processors.py
│   │   ├── __init__.py
│   │   ├── arbin_res.py
│   │   ├── arbin_sql.py
│   │   ├── arbin_sql_7.py
│   │   ├── arbin_sql_csv.py
│   │   ├── arbin_sql_h5.py
│   │   ├── arbin_sql_xlsx.py
│   │   ├── base.py
│   │   ├── biologics_mpr.py
│   │   ├── custom.py
│   │   ├── ext_nda_reader.py
│   │   ├── local_instrument.py
│   │   ├── maccor_txt.py
│   │   ├── neware_txt.py
│   │   ├── pec_csv.py
│   │   └── SQL Table IDs.txt
│   ├── __init__.py
│   ├── cellreader.py
│   ├── core.py
│   ├── dbreader.py
│   ├── filefinder.py
│   └── sql_dbreader.py
└── utils
    ├── batch_tools
```

(continues on next page)

(continued from previous page)

```
    │   ├── __init__.py
    │   ├── batch_analyzers.py
    │   ├── batch_core.py
    │   ├── batch_experiments.py
    │   ├── batch_exporters.py
    │   ├── batch_helpers.py
    │   ├── batch_journals.py
    │   ├── batch_plotters.py
    │   ├── batch_reporters.py
    │   ├── dumpers.py
    │   ├── engines.py
    │   └── sqlite_from_excel_db.py
    ├── data
    │   ├── raw
    │   │   └── 20160805_test001_45_cc_01.res
    │   └── 20160805_test001_45_cc.h5
    ├── __init__.py
    ├── batch.py
    ├── collectors.py
    ├── collectors_old.py
    ├── diagnostics.py
    ├── easyplot.py
    ├── example_data.py
    ├── helpers.py
    ├── ica.py
    ├── live.py
    ├── ocv_rlx.py
    ├── plotutils.py
    └── processor.py
...
```

Handling of parameters

TODO: explain how parameters are handled

`.cellpy_prms_{user}.conf`

`.env_cellpy` and environment variables.

`cellpy.prms`

`cellpy.parameters.internal_settings`

Logging

cellpy uses the standard python logging module.

Utilities

5.3.4 About loaders

Instrument loaders are used by CellpyCell to load data from a file into a Data object. The instrument loaders are all located in the cellpy.readers.instruments package (*i.e.* in the folder *cellpy/readers/instruments*). They are Python modules (.py files) and are automatically registered when CellpyCell is initialized:

CellpyCell.register_instrument_readers -> core.generate_default_factory -> core.find_all_instruments

(Note that there are some naming conventions for the instrument loaders, see find_all_instruments in *cellpy/core.py*)

The specific instrument loader is selected in CellpyCell through CellpyCell._set_instrument:

CellpyCell.from_raw -> CellpyCell.set_instrument -> CellpyCell._set_instrument

The instrument class is instantiated and bound to the CellpyCell instance (CellpyCell.loader_class). The actual method (loader_executor) that will be executed when loading the data (through CellpyCell.from_raw) is bound to the attribute CellpyCell.loader_method.

When CellpyCell.from_raw is called, the AtomicLoad.loader_executor is in charge of executing the instrument loaders DataLoader.loader method. One implication of this, since the DataLoader is a subclass of AtomicLoad, is that it is very "dangerous" to override loader_executor in the DataLoader.

Structure of instrument loaders

Each reader is a subclass of DataLoader (in base.py). It must implement at least the following methods: get_raw_units, get_raw_limits, and loader.

The instrument loaders must all have a class named DataLoader which is a subclasses of BaseLoader (in *cellpy/readers/instruments/base.py*) or one of its available subclasses (see below).

The following subclasses of BaseLoader are available (v.1.0): AutoLoader and TxtLoader.

Subclassing BaseLoader

The BaseLoader class is a subclass of AtomicLoad (in *cellpy/readers/instruments/base.py*) using abc.ABCMeta as metaclass.

For databases and similar, it is recommended to subclass BaseLoader directly. This allows you to set the class attribute _is_db to True. This will make the loader_executor method in AtomicLoad to not copy the "file" to a temporary location before issuing the loader method.

Subclassing `AutoLoader`

The `AutoLoader` class is a subclass of `BaseLoader`. This class can be sub-classed if you want to make a data-reader for different type of "easily parsed" files (for example csv-files).

The `AutoLoader` class implements loading a configuration from a configuration module of file (see below), and performs pre- and post-processing of the data/raw-file (the processors are turned on or off in the configuration). Subclasses of the `AutoLoader` class must implement the following methods: `parse_loader_parameters`, `parse_formatter_parameters`, and `query_file`.

The `query_file` method must return a `pandas.DataFrame` and accept a filename as argument, e.g.:

```
def query_file(self, name):
    return pd.read_csv(name)
```

You can´t provide additional arguments to the `query_file` method, but instead promote them to instance variables using the `parse_formatter_parameter` method:

```
def parse_loader_parameters(self, **kwargs):
    self.warn_bad_lines = kwargs.get("warn_bad_lines", None)
```

and then use the instance variables in the `query_file` method:

```
def query_file(self, name):
    return pd.read_csv(name, warn_bad_lines=self.warn_bad_lines)
```

Subclassing `TxtLoader`

The `TxtLoader` class is a subclass of `AutoLoader`. The subclass of a `TxtLoader` gets its information by loading model specifications from its respective module(`cellpy.readers.instruments.configurations.<module>`) or configuration file (yaml).

The `TxtLoader` class uses `pandas.read_csv` as its query method, and reads configurations from modules in `cellpy.readers.instruments.configuration` (or config file). It also implements the `model` keyword.

Examples of modules where the loader is subclassing `TxtLoader` are `maccor_txt`, `neware_txt` and `local_instrument`.

The `local_instrument` loader

This module is used for loading data using the corresponding Local yaml file with definitions on how the data should be loaded. This loader is based on the `TxtLoader` and can only be used to load csv-type files.

The `custom` loader

This module is used for loading data using the *instrument="custom"* method. If no *instrument_file* is given (either directly or through the use of the :: separator), the default instrument file (yaml) will be used. This loader is based on the `AutoLoader`. It is more flexible than the loader in `local_instrument` and can for example chose between several file querying methods (csv, xls, xlsx), but it is also more complicated to use.

5.3.5 Writing documentation

All contributions on documenting `cellpy` is highly welcomed.

Types of documentation

Code can be documented by several means. All are valuable. For example:

- adding examples to the *examples* folder
- improving the documentation in the *docs* folder
- improving the doc-strings in the code
- adding descriptive tests to the code
- providing example data (e.g. raw-files in different formats)
- tutorials on YouTube
- blogs etc.
- apps

Add examples to the examples folder

Todo

Working on the main documentation

The docs are hosted on Read the Docs

- Stable: https://cellpy.readthedocs.io/en/stable/
- Latest: https://cellpy.readthedocs.io/en/latest/
- Admin: https://readthedocs.org/projects/cellpy/

Sphinx is used to render the documentation.

Link to help: https://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html

Notebooks can also be used.

Sphinx tooling

List of extensions used

- sphinx.ext.autodoc
- sphinx.ext.viewcode
- sphinx.ext.napoleon
- sphinx.ext.intersphinx
- nbsphinx
- sphinx.ext.graphviz

Current HTML theme:

- sphinx_rtd_theme

Available "variables":

```
|ProjectVersion| -> writes "Version: <version number>"
```

Dependencies (python packages):

- pandoc
- sphinx-rtd-theme
- nbsphinx

Dependencies (non-python):

- pandoc
- graphviz

Creating the API documentation:

```
# in the repository root folder
sphinx-apidoc -o docs/source cellpy/
```

Conventions

Order of headers:

```
========
Header 1
========

Header 2
========

Header 3
--------

Header 4
........

Header 5
~~~~~~~~
```

Note that many of the documents (.rst files) are linked through an *index.rst* file. This file most likely contains the Header 1, so the actual document you are working on needs to start with Header 2.

Doc-strings

Todo

Tests

Todo

5.3.6 Create package for PyPI

Build package

```
$ python -m build
```

Upload package

The `cellpy` package on PyPI requires 2-factor identification. For this to work, you need to create a token. See below for instructions on how to do this.

```
$ python -m twine upload dist/* -u __token__ -p pypi-<token>"
```

Note: The following instructions are copied from the PyPI website (https://pypi.org/help/#apitoken). Please see the PyPI website for the latest instructions.

Get the API token:

- Verify your email address (check your account settings).
- In your account settings, go to the API tokens section and select "Add API token"

To use an API token:

- Set your username to "__token__"
- Set your password to the token value, including the pypi- prefix

---

5.3.7 Create conda package

This is a short description on how to update the conda-forge recipe for `cellpy`:

- **Fork**
  If not already done, make a fork of https://github.com/conda-forge/cellpy-feedstock
- **Clone**
  If not already done, clone the repo (your-name/cellpy-feedstock)

  ```
  >>> git clone https://github.com/your-name/cellpy-feedstok.git
  >>> git remote add upstream https://github.com/conda-forge/cellpy-feedstock
  ```

- **Get recent changes**

```
>>> git fetch upstream
>>> git rebase upstream/main
```

This can also be done (I think) via the web interface by navigating to your fork and clicking the button "Sync fork".

- **Make a new branch in your local clone**

```
>>> git checkout -b update_x_x_x
```

- **Edit**
  - hash (*pypi - release history - Download files*)
  - version (use normalized format *e.g.* `0.5.2a3` not `0.5.2.a3`!)
  - build number (should be 0 for new versions)
- **Add and commit**
  e.g. "updated feedstock to version 1.0.1"
- **Push**

```
>>> git push origin <branch-name>
```

- **Re-render if needed (different requirements, platforms, issues)**

```
>>> conda install -c conda-forge conda-smithy
>>> conda smithy rerender -c auto
```

This can also be done (I think) via adding a special comment during the pull request/merging on github.

- Create a pull request via the web interface by navigating to https://github.com/your-name/cellpy-feedstok.git with your web browser and clicking the button create pull request.
- Wait until the automatic checks have complete (takes several minutes)
- Merge pull request (big green button)
- Drink a cup of coffee or walk the dog
- **Check if the new version is there:**

```
>>> conda search -f cellpy
```

- Now you can delete the branch (if you want)

5.3.8 Various

Using `pytest` fixtures

Retrieve constants during tests

There is a fixture in `conftest.py` aptly named `parameters` making all the variables defined in `fdv.py` accessible for the tests. So, please, add additional parameter / constant by editing the `fdv.py` file.

Other

You can check the `conftest.py` file to see what other fixtures are available.

Example

```
from cellpy import prms

# using the ``parameters`` and the ``cellpy_data_instance`` fixtures.

def test_set_instrument_selecting_default(cellpy_data_instance, parameters):
    prms.Instruments.custom_instrument_definitions_file = parameters.custom_instrument_
↪definitions_file
    cellpy_data_instance.set_instrument(instrument="custom")
```

Adding another config parameter

1. Edit `prms.py`
2. Check / update the `internal_settings.py` file as well to ensure that copying / splitting `cellpy` objects behaves properly.
3. Check / update the `.cellpy_prms_default.conf` file

The relevant files are located in the `parameters` folder:

```
cellpy/
    parameters/
        .cellpy_prms_default.conf
        prms.py
        internal_settings.py
```

Installing *pyodbc* on Mac (no *conda*)

If you do not want to use *conda*, you might miss a couple of libraries.

The easiest fix is to install *uniuxodbc* using *brew* as explained in Stack Overflow #54302793.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search