cellpy Documentation

Release 0.1.13

Jan Petter Maehlen

May 15, 2017

Contents

Contents:

CHAPTER 1

cellpy

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.

- Free software: MIT license
- Documentation: https://cellpy.readthedocs.io.

Features

- Load test-data and store in hdf5 format.
- Filter out the steps of interest.
- Process and plot the data.
- And more...

Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

Stable release

To install `cellpy`, run this command in your terminal:

```
$ pip install cellpy
```

This is the preferred method to install `cellpy`, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

`cellpy` uses setuptools, and the developers of *setuptools* recommend notifying the users the following:

- if you would like to install `cellpy` to somewhere other than the main site-packages directory, then you should first install setuptools using the instructions for Custom Installation Locations, before installing `cellpy`.

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

```
$ cellpy setup
```

This will install a `_cellpy_prms_USER.config` file in your home directory (USER = your user name). Edit this file and save it as `_cellpy_prms_OTHERNAME.conf` to prevent it from being written over in case the setup script is run on a later occasion.

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

Note: At the moment, I have not really figured out how to implement and install something for reading access database files on other operating systems than windows. So, for now, I guess `cellpy` only will work on windows (and automatic building with Travis gets challenging).

From sources

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

You can either clone the public repository:

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

Or download the tarball:

```
$ curl  -OL https://github.com/jepegit/cellpy/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use `cellpy`, start with importing the needed modules:

```
>>> from cellpy import cellreader
```

Lets define some variables:

```
>>> FileName  = r"C:\data\20141030_CELL_6_cc_01.res"
>>> Mass      = 0.982 # mass of active material in mg
>>> OutFolder = 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):

```
>>> d = cellreader.cellpydata()
>>> d.load_raw(FileName) # this tells cellpy to read the arbin data file (.res format)
>>> d.set_mass(Mass)
```

Create a summary (for each cycle) and generate a step table (parsing the data and finding out what each step in each cycle is):

```
>>> d.make_summary()
>>> d.create_step_table()
```

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

```
>>> d.exportcsv(OutFolder)
```

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 = d.get_cycle_numbers()
>>> number_of_cycles = len(list_of_cycles)
>>> print "you have %i cycles" % (number_of_cycles)
you have 658 cycles
>>> current,voltage = d.get_cap(5) # current and voltage for cycle 5
```

You can also look for open circuit voltage steps:

```
>>> cycle = 44
>>> time1, voltage1 = d.get_ocv(ocv_type='ocvrlx_up', cycle_number=cycle)
>>> time2, voltage2 = d.get_ocv(ocv_type='ocvrlx_down', cycle_number=cycle)
```

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.

CHAPTER 4

Tutorials

The cellpy command

At the moment, only a very limited set of things can be achieved by running 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:
  configloc
  setup
  version
```

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 version
[cellpy] version: 0.1.11

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

The most important command is probably the `setup` command (that should be run when you install cellpy for the first time).

Configuring cellpy

How the configuration parameters are set and read

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

The parameters are stored in the module `cellpy.parameters.prms` as in several dictionaries. I know, this is probably not the most convenient method, but it is very easy (at least I hope so) to change these into class-type stuff in a later release of `cellpy` (using for example `type(x, y, z)` *etc*. or `setattr` *etc*).

If you during your script (or in your `jupyter notebook`) would like to change some of the settings (*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'
```

In some of the modules or classes, selected parameters are already 'transformed' to class attributes, and those can be assigned intuitively:

```
from cellpy import dbreader as dr

print(dr.db_sheet_cols.batch)
# prints the column number for the column containing the "batch" label

dr.db_sheet_cols.batch = 3
# sets the column number for the column containing the "batch" label to 3

print(dr.db_sheet_cols.batch)
# prints '3', the new column number for the column containing the "batch" label
```

A more thorough description of this will come in later releases (0.2.0 and up).

The configuration file

`cellpy` tries to read your .conf-file when imported the first time, and looks in your user directory (*e.g.* C:\Users\USERNAME on not-too-old versions of windows) after files named `_cellpy_prms_SOMENAME.conf`. If you have run `cellpy -setup` in the cmd window or in the shell, a file named `_cellpy_prms_USERNAME.conf` (where USERNAME is your username) should exist in your home directory. This 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:

```
---
DataSet:
  nom_cap: 3579
Db:
  db_type: simple_excel_reader
```

```
FileNames: {}
Instruments:
  cell_configuration: anode
  tester: arbin
Paths:
  cellpydatadir:  C:\ExperimentalData\BatteryTestData\Arbin\HDF5
  db_filename: 2017_Cell_Analysis_db_001.xlsx
  db_path: C:\Users\jepe\Documents\Databases\Experiments\arbin
  filelogdir: C:\Scripting\Processing\Celldata\outdata
  outdatadir: C:\Scripting\Processing\Celldata\outdata
  rawdatadir: I:\Org\ensys\EnergyStorageMaterials\Data-backup\Arbin
Reader:
  auto_dirs: true
  cellpy_datadir: null
  chunk_size: null
  cycle_mode: anode
  daniel_number: 5
  .
  .
```

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, he or she is using windows.

Looking further down in the file, you come to some sections related to the 'excel database reader':

```
excel_db_cols:
  A1: 28
  A2: 29
  A3: 30
  A4: 31
  .
  .
```

Here you can set custom column numbers for where the database reader should look for stuff. For example, if you have your entry specifying active material (mass) in column 100, then edit your configuration file entry `active_material`:

```
excel_db_cols:
  .
  .
  active_material: 35
  .
  .
```

To:

```
excel_db_cols:
  .
  .
  active_material: 100
  .
  .
```

A more in-depth description of this will come in later releases (0.2.0 and up). 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 are more stuff in the actual file than what is shown here.

Basic interaction 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).

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

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

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

```
raw_data_dir = r"C:\raw_data"
out_data_dir = r"C:\processed_data"
cellpy_data_dir = r"C:\cellpydata"
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 cellpydata object and sets the cycle mode:
cell_data = cellreader.cellpydata()
cell_data.set_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.load_raw([raw_files])
cell_data.set_mass(electrode_mass)
cell_data.make_summary()
# Note: make_summary will automatically run the
# create_step_table function if it does not exist.
```

And save it:

```
# defining a name for the cellpy_file (hdf5-format)
cellpy_file = os.path.join(cellpy_data_dir, "20170101_ife01_cc2.h5")
cell_data.save_test(cellpy_file)
```

For convenience, `cellpy` also has a method that simplifies this process a little bit. Using the `loadcell` method, you can specify both the raw file name(s) and the cellpy file name, and `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.loadcell(raw_files=[raw_files], cellpy_file=cellpy_file,
                       mass=[electrode_mass], summary_on_raw=True,
                       force_raw=False)
```

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

Extract current-voltage graphs

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

Create dQ/dV plots

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

Save / export data

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

```
# export data to csv
out_data_directory = r"C:\processed_data\csv"
# this exports the summary data to a .csv file:
cell_data.exportcsv(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
```

Using some of the cellpy special utilities

Fitting ocv-rlx data

TODO.

Fitting ica data

TODO.

Data mining / using a database

TODO.

Using the batch utilities

TODO.

Working with the pandas.DataFrame objects directly

The `cellpydata` 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 cellpydata object: cellpy_data

# get the 'test':
cellpy_test = cell_data.get_test()
# cellpy_test is now a cellpy dataset object (cellpy.readers.cellreader.dataset)

# pandas.DataFrame with data vs cycle number (e.g. coulombic efficiency):
summary = cellpy_test.dfsummary

# pandas.DataFrame with the raw data:
rawdata = cellpy_test.dfdata

# pandas.DataFrame with statistics on each step and info about step type:
step_table = cellpy_test.step_table

# run_summary = cellpy_test.run_summary
# This is not implemented yet (overall information like cycle life-time)
```

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

Note: At the moment, **cellpydata** objects can store several sets of test-data (several 'tests'). They are stored in a list. It is not recommended to utilise this *'possible to store multiple tests'* feature as it might be removed very soon (have not decided upon that yet).

Happy pandas-ing!

CHAPTER 5

File Formats and Data Structures

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

cellpydata - 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:

```
cellpydata.tester = "arbin"
cellpydata.auto_dir = True
print cellpydata.cellpy_datadir
```

The data for the run(s) are stored in the class attribute *cellpydata.tests* (this will most likely change in future versions). This attribute is just a list of runs (each run is a *cellpy.cellreader.dataset* instance). This implies that you can store many runs in one *cellpydata* instance. Sometimes this can be necessary, but it is recommended to only store one run in one instance. Most of the functions (the class methods) automatically selects the 0-th item in *cellpydata.tests* if the test_number is not explicitly given.

You may already have figured it out: in cellpy, data for a given cell is usually named a run. And each run is a *cellpy.cellreader.dataset* instance.

Here is a list of other important class attributes in *cellpydata*:

column headings - normal data

```
cellpydata.headers_normal['aci_phase_angle_txt'] = 'ACI_Phase_Angle'
cellpydata.headers_normal['ac_impedance_txt'] = 'AC_Impedance'
cellpydata.headers_normal['charge_capacity_txt'] = 'Charge_Capacity'
cellpydata.headers_normal['charge_energy_txt'] = 'Charge_Energy'
cellpydata.headers_normal['current_txt'] = 'Current'
cellpydata.headers_normal['cycle_index_txt'] = 'Cycle_Index'
cellpydata.headers_normal['data_point_txt'] = 'Data_Point'
cellpydata.headers_normal['datetime_txt'] = 'DateTime'
cellpydata.headers_normal['discharge_capacity_txt'] = 'Discharge_Capacity'
cellpydata.headers_normal['discharge_energy_txt'] = 'Discharge_Energy'
cellpydata.headers_normal['internal_resistance_txt'] = 'Internal_Resistance'
cellpydata.headers_normal['is_fc_data_txt'] = 'Is_FC_Data'
cellpydata.headers_normal['step_index_txt'] = 'Step_Index'
cellpydata.headers_normal['step_time_txt'] = 'Step_Time'
cellpydata.headers_normal['test_id_txt'] = 'Test_ID'
cellpydata.headers_normal['test_time_txt'] = 'Test_Time'
cellpydata.headers_normal['voltage_txt'] = 'Voltage'
cellpydata.headers_normal['dv_dt_txt'] = 'dV/dt'
```

column headings - summary data

```
cellpydata.headers_summary["discharge_capacity"] = "Discharge_Capacity(mAh/g)"
cellpydata.headers_summary["charge_capacity"] = "Charge_Capacity(mAh/g)"
cellpydata.headers_summary["cumulated_charge_capacity"] = "Cumulated_Charge_
↪Capacity(mAh/g)"
cellpydata.headers_summary["cumulated_discharge_capacity"] = "Cumulated_Discharge_
↪Capacity(mAh/g)"
cellpydata.headers_summary["coulombic_efficiency"] = "Coulombic_Efficiency(percentage)
↪"
cellpydata.headers_summary["cumulated_coulombic_efficiency"] = "Cumulated_Coulombic_
↪Efficiency(percentage)"
cellpydata.headers_summary["coulombic_difference"] = "Coulombic_Difference(mAh/g)"
cellpydata.headers_summary["cumulated_coulombic_difference"] = "Cumulated_Coulombic_
↪Difference(mAh/g)"
cellpydata.headers_summary["discharge_capacity_loss"] = "Discharge_Capacity_Loss(mAh/
↪g)"
cellpydata.headers_summary["charge_capacity_loss"] = "Charge_Capacity_Loss(mAh/g)"
cellpydata.headers_summary["cumulated_discharge_capacity_loss"] = "Cumulated_
↪Discharge_Capacity_Loss(mAh/g)"
cellpydata.headers_summary["cumulated_charge_capacity_loss"] = "Cumulated_Charge_
↪Capacity_Loss(mAh/g)"
cellpydata.headers_summary["ir_discharge"] = "IR_Discharge(Ohms)"
cellpydata.headers_summary["ir_charge"] = "IR_Charge(Ohms)"
cellpydata.headers_summary["ocv_first_min"] = "OCV_First_Min(V)"
cellpydata.headers_summary["ocv_second_min"] = "OCV_Second_Min(V)"
cellpydata.headers_summary["ocv_first_max"] = "OCV_First_Max(V)"
cellpydata.headers_summary["ocv_second_max"] = "OCV_Second_Max(V)"
cellpydata.headers_summary["date_time_txt"] = "Date_Time_Txt(str)"
cellpydata.headers_summary["end_voltage_discharge"] = "End_Voltage_Discharge(V)"
cellpydata.headers_summary["end_voltage_charge"] = "End_Voltage_Charge(V)"
cellpydata.headers_summary["cumulated_ric_disconnect"] = "RIC_Disconnect(none)"
cellpydata.headers_summary["cumulated_ric_sei"] = "RIC_SEI(none)"
cellpydata.headers_summary["cumulated_ric"] = "RIC(none)"
cellpydata.headers_summary["low_level"] = "Low_Level(percentage)"  # Sum of␣
↪irreversible capacity
```

```
cellpydata.headers_summary["high_level"] = "High_Level(percentage)"  # SEI loss
cellpydata.headers_summary["shifted_charge_capacity"] = "Charge_Endpoint_Slippage(mAh/
↪g)"
cellpydata.headers_summary["shifted_discharge_capacity"] = "Discharge_Endpoint_
↪Slippage(mAh/g)"
cellpydata.headers_summary["temperature_last"] = "Last_Temperature(C)"
cellpydata.headers_summary["temperature_mean"] = "Average_Temperature(C)"
cellpydata.headers_summary["pre_aux"] = "Aux_"
```

column headings - step table

```
cellpydata.headers_step_table["test"] = "test"
cellpydata.headers_step_table["cycle"] = "cycle"
cellpydata.headers_step_table["step"] = "step"
cellpydata.headers_step_table["sub_step"] = "sub_step"
cellpydata.headers_step_table["type"] = "type"
cellpydata.headers_step_table["sub_type"] = "sub_type"
cellpydata.headers_step_table["info"] = "info"
cellpydata.headers_step_table["pre_current"] = "I_"
cellpydata.headers_step_table["pre_voltage"] = "V_"
cellpydata.headers_step_table["pre_charge"] = "Charge_"
cellpydata.headers_step_table["pre_discharge"] = "Discharge_"
cellpydata.headers_step_table["pre_point"] = "datapoint_"
cellpydata.headers_step_table["pre_time"] = "time_"
cellpydata.headers_step_table["post_mean"] = "avr"
cellpydata.headers_step_table["post_std"] = "std"
cellpydata.headers_step_table["post_max"] = "max"
cellpydata.headers_step_table["post_min"] = "min"
cellpydata.headers_step_table["post_start"] = "start"
cellpydata.headers_step_table["post_end"] = "end"
cellpydata.headers_step_table["post_delta"] = "delta"
cellpydata.headers_step_table["post_rate"] = "rate"
cellpydata.headers_step_table["internal_resistance"] = "IR"
cellpydata.headers_step_table["internal_resistance_change"] = "IR_pct_change"
```

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']
```

For each type of testers that are supported by cellpy, a set of column headings and other different settings/attributes must be provided. These definitions are now put inside the cellpydata class, but will be moved out later.

Supported testers are:

- arbin

Testers that is planned supported:

- biologic

- pec
- maccor

Tester dependent attributes

arbin

Three tables are read from the .res file:

- normal table: contains measurement data.
- global table: contains overall parametres for the test.
- stats table: contains statistics (for each cycle).

table names

```
tablename_normal = "Channel_Normal_Table"
tablename_global = "Global_Table"
tablename_statistic = "Channel_Statistic_Table"
```

column headings - global table

```
applications_path_txt = 'Applications_Path'
channel_index_txt = 'Channel_Index'
channel_nuer_txt = 'Channel_Number'
channel_type_txt = 'Channel_Type'
comments_txt = 'Comments'
creator_txt = 'Creator'
daq_index_txt = 'DAQ_Index'
item_id_txt = 'Item_ID'
log_aux_data_flag_txt = 'Log_Aux_Data_Flag'
log_chanstat_data_flag_txt = 'Log_ChanStat_Data_Flag'
log_event_data_flag_txt = 'Log_Event_Data_Flag'
log_smart_battery_data_flag_txt = 'Log_Smart_Battery_Data_Flag'
mapped_aux_conc_cnumber_txt = 'Mapped_Aux_Conc_CNumber'
mapped_aux_di_cnumber_txt = 'Mapped_Aux_DI_CNumber'
mapped_aux_do_cnumber_txt = 'Mapped_Aux_DO_CNumber'
mapped_aux_flow_rate_cnumber_txt = 'Mapped_Aux_Flow_Rate_CNumber'
mapped_aux_ph_number_txt = 'Mapped_Aux_PH_Number'
mapped_aux_pressure_number_txt = 'Mapped_Aux_Pressure_Number'
mapped_aux_temperature_number_txt = 'Mapped_Aux_Temperature_Number'
mapped_aux_voltage_number_txt = 'Mapped_Aux_Voltage_Number'
schedule_file_name_txt = 'Schedule_File_Name'
start_datetime_txt = 'Start_DateTime'
test_id_txt = 'Test_ID'
test_name_txt = 'Test_Name'
```

column headings - normal table

```
aci_phase_angle_txt = 'ACI_Phase_Angle'
ac_impedance_txt = 'AC_Impedance'
charge_capacity_txt = 'Charge_Capacity'
charge_energy_txt = 'Charge_Energy'
current_txt = 'Current'
cycle_index_txt = 'Cycle_Index'
data_point_txt = 'Data_Point'
datetime_txt = 'DateTime'
discharge_capacity_txt = 'Discharge_Capacity'
discharge_energy_txt = 'Discharge_Energy'
internal_resistance_txt = 'Internal_Resistance'
is_fc_data_txt = 'Is_FC_Data'
step_index_txt = 'Step_Index'
step_time_txt = 'Step_Time'
test_id_txt = 'Test_ID'
test_time_txt = 'Test_Time'
voltage_txt = 'Voltage'
dv_dt_txt = 'dV/dt'
```

cellpydata - methods

Todo

dataset

Each run is a *cellpy.cellreader.dataset* 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:

- normal data
- step table
- summary data

Todo.

fileID

Todo

CHAPTER 6

Examples

Look in the Examples folder on the GitHub repository (https://github.com/jepegit/cellpy).

CHAPTER 7

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. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv cellpy
$ cd cellpy/
$ python setup.py develop
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 cellpy tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

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.

(Note: currently, tox is not set up properly, the plan is to have it up-and-running before version 0.2.0)

Pull Request Guidelines

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

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7. Check https://travis-ci.org/jepegit/cellpy/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ py.test tests.test_cellpy
$ py.test tests.test_readers
```

CHAPTER 8

Credits

Development Lead

- Jan Petter Maehlen <jepe@ife.no>
- Asbjoern Ulvestad <asbjorn.ulbestad@ife.no>
- Tor Kristian Vara <tor.vara@nmbu.no>

Contributors

None yet. Why not be the first?

CHAPTER 9

History

0.1.0 (2016-09-26)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search