pandalone Documentation

Release 0.1.10.dev0

Authors: see AUTHORS.rst

April 09, 2016

Contents

pandalone is a collection of utilities for working with *hierarchical-data* using *relocatable-paths*.

Release 0.1.10.dev0

Date 2016-04-09 15:14:02

Documentation https://pandalone.readthedocs.org/

Source https://github.com/pandalone/pandalone

PyPI repo https://pypi.python.org/pypi/pandalone

Keywords calculation, data, dependencies, engineering, excel, library, numpy, pandas, processing, python, resolution, scientific, simulink, tree, utility

Currently only 2 portions of the envisioned functionality are ready for use:

- `pandalone.xleash`: A mini-language for "throwing the rope" around rectangular areas of Excel-sheets.
- `pandalone.mappings`: Hierarchical string-like objects that may be used for indexing, facilitating renaming keys and column-names at a later stage.

Our goal is to facilitate the composition of *engineering-models* from loosely-coupled *components*. Initially envisioned as an *indirection-framework* around *pandas* coupled with a *dependency-resolver*, every such model should auto-adapt and process only values available, and allow *remapping* of the paths accessing them, to run on renamed/relocated *value-trees* without component-code modifications.

It is an open source library written for *python-3.4* but tested under both *python-2.7* and *python-3.3+*, for *Windows* and *Linux*.

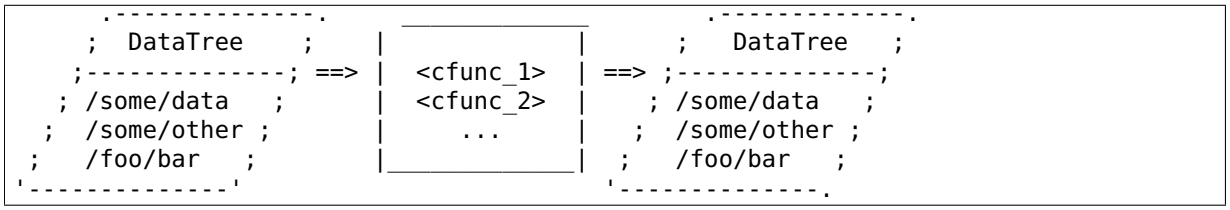
Note: The project, as of May-2015, is considered at an alpha-stage, without any released version in *pypi* yet.

CHAPTER 1

Introduction

1.1 Overview

At the most fundamental level, an "execution" or a "run" of any data-processing can be thought like that:



- The *data-tree* might come from *json*, *hdf5*, *excel-workbooks*, or plain dictionaries and lists. Its values are strings and numbers, *numpy-lists*, *pandas* or *xray-datasets*, etc.
- The *component-functions* must abide to the following simple signature:

```
cfunc_do_something(pandelone, datatree)
```

  and must not return any value, just read and write into the data-tree.

- Here is a simple component-function:

```
def cfunc_standardize(pandelone, datatree):
    pin, pon = pandelone.paths(),
    df = datatree.get(pin.A)
    df[pon.A.B_std] = df[pin.A.B] / df[pin.A.B].std()
```

- Notice the use of the *relocatable-paths* marked specifically as input or output.
- TODO: continue rough example in tutorial...

1.2 Quick-start

Note: The program runs on **Python-2.7+** and **Python-3.3+** (preferred) and requires **numpy/scipy**, **pandas** and **win32** libraries along with their *native backends* to be installed. If you do not have such an environment already installed, please read Install section below for suitable distributions such as *Anaconda* or *WinPython*.

Assuming that you have a working python-environment, open a *command-shell*, (in *Windows* use **`cmd.exe`** BUT ensure **`python.exe`** is in its `PATH`), try the following commands:

Tip: The commands beginning with `$`, below, imply a *Unix* like operating system with a *POSIX* shell (*Linux, OS X*). Although the commands are simple and easy to translate in its *Windows* `cmd.exe` counterpart, it would be worthwile to install Cygwin to get the same environment on *Windows*. If you choose to do that, include also the following packages in the *Cygwin*'s installation wizard:

```
* git, git-completion
* make, zip, unzip, bzip2, dos2unix
* openssh, curl, wget
```

But do not install/rely on cygwin's outdated python environment.

Install

```
$ pip install pandalone                    ## Use --pre if version-string has a build-suffi
```

Or in case you need the very latest from `master` branch :

```
$ pip install git+https://github.com/pandalone/pandalone.git
```

See: Install

Run

```
$ pandalone --version
```

1.3 Discussion

CHAPTER 2

Install

Current version(0.1.10.dev0) runs on **Python-2.7+** and **Python-3.3+** and requires **numpy/scipy**, **pandas** and **win32** libraries along with their *native backends* to be installed.

It has been tested under *Windows* and *Linux* and *Python-3.3+* is the preferred interpreter, i.e, the *Excel* interface and desktop-UI runs only with it.

It is distributed on Wheels.

2.1 Python installation

Warning: On *Windows* it is strongly suggested **NOT to install the standard CPython distribution**, unless:

1. you have *administrative priviledges*,
2. you are an experienced python programmer, so that
3. you know how to hunt dependencies from *PyPi* repository and/or the Unofficial Windows Binaries for Python Extension Packages.

As explained above, this project depends on packages with *native-backends* that require the use of *C* and *Fortran* compilers to build from sources. To avoid this hassle, you should choose one of the user-friendly distributions suggested below.

Below is a matrix of the two suggested self-wrapped python distributions for running this program (we excluded here default *python* included in *linux*). Both distributions:

- are free (as of freedom),
- do not require *admin-rights* for installation in *Windows*, and
- have been tested to run successfully this program (also tested on default *linux* distros).

Distributions	*WinPython*	*Anaconda*
Platform	**Windows**	**Windows**, **Mac OS**, **Linux**
Ease of Installation	Fair (requires fiddling with the `PATH` and the Registry after install)	• *Anaconda:* Easy • *MiniConda:* Moderate
Ease of Use	Easy	Moderate (should use **`conda`** and/or **`pip`** depending on whether a package contains native libraries
# of Packages	Only what's included in the downloaded-archive	Many 3rd-party packages uploaded by users
Notes	After installation, see Frequently Asked Questions (FAQ) for: • Registering WinPython installation • Adding your installation in `PATH`	• Check also the lighter miniconda. • For installing native-dependencies with **`conda`** see files: - `requirements/miniconda.conda` - `.travis.yaml`
	Check also installation instructions from the pandas site.	

2.2 Package installation

Before installing it, make sure that there are no older versions left over on the python installation you are using. To cleanly uninstall it, run this command until you cannot find any project installed:

```
$ pip uninstall pandalone                    ## Use pip3 if both python-2 & 3 are in PATH.
```

You can install the project directly from the *PyPi* repo the "standard" way, by typing the **`pip`** in the console:

```
    $ pip install pandalone
```

- If you want to install a *pre-release* version (the version-string is not plain numbers, but ends with `alpha`, `beta.2` or something else), use additionally `--pre`.

```
$ pip install pandalone
```

- Also you can install the very latest version straight from the sources:

```
    $ pip install git+git://github.com/pandalone/pandalone.git  --pre
```

- If you want to upgrade an existing installation along with all its dependencies, add also `--upgrade` (or `-U` equivalently), but then the build might take some considerable time to finish. Also there is the possibility the upgraded libraries might break existing programs(!) so use it with caution, or from within a *virtualenv* (isolated Python environment).
- To install it for different Python environments, repeat the procedure using the appropriate **`python.exe`** interpreter for each environment.

- **Tip:** To debug installation problems, you can export a non-empty `DISTUTILS_DEBUG` and *distutils* will print detailed information about what it is doing and/or print the whole command line when an external program (like a C compiler) fails.

After installation, it is important that you check which version is visible in your `PATH`:

```
$ pndlcmd --version
0.1.10.dev0
```

To install for different Python versions, repeat the procedure for every required version.

2.3 Older versions

To install an older released version issue the console command:

```
$ pip install pandalone=0.0.1                    ## Use --pre if version-string has a build-suffix
```

or alternatively straight from the sources:

```
    $ pip install git+https://github.com/pandalone/pandalone.git@v0.0.9-alpha.3.1  --pre
```

Of course you can substitute `v0.0.9-alpha.3.1` with any slug from "commits", "branches" or "releases" that you will find on project's github-repo).

Note: If you have another version already installed, you have to use `--ignore-installed` (or `-I`). For using the specific version, check this (untested) stackoverflow question.

You can install each version in a separate *virtualenv* (isolated Python environment) and shy away from all this. Check

2.4 Installing sources

If you download the sources you have more options for installation. There are various methods to get hold of them:

- Download the *source* distribution from *PyPi* repo.
- Download a release-snapshot from github
- Clone the *git-repository* at *github*.

  Assuming you have a working installation of git you can fetch and install the latest version of the project with the following series of commands:

```
    $ git clone "https://github.com/pandalone/pandalone.git" pandalone.git
    $ cd pandalone.git
    $ python setup.py install                                ## Use python3 if both python-2 & 3
```

When working with sources, you need to have installed all libraries that the project depends on:

```
$ pip install -r requirements/execution.pip .
```

The previous command installs a "snapshot" of the project as it is found in the sources. If you wish to link the project's sources with your python environment, install the project in development mode:

```
$ python setup.py develop
```

Note: This last command installs any missing dependencies inside the project-folder.

2.5 Project files and folders

The files and folders of the project are listed below:

```
+--pandalone/       ## (package) Python-code
+--tests/           ## (package) Test-cases
+--doc/             ## Documentation folder
+--setup.py         ## (script) The entry point for setuptools, installing, testing, etc
+--requirements/    ## (txt-files) Various pip and conda dependencies.
+--README.rst
+--CHANGES.rst
+--AUTHORS.rst
+--CONTRIBUTING.rst
+--LICENSE.txt
```

2.6 Discussion

CHAPTER 3

Usage

Currently 2 portions of this library are ready for use: *pandalone.xleash* and *pandalone.mappings*

3.1 Cmd-line usage

> **Warning:** Not implemented in yet.

The command-line usage below requires the Python environment to be installed, and provides for executing an experiment directly from the OS's shell (i.e. **cmd** in windows or **bash** in POSIX), and in a *single* command.

[TBD]

3.2 GUI usage

> **Attention:** Desktop UI requires Python 3!

For a quick-'n-dirty method to explore the structure of the data-tree and run an experiment, just run:

```
$ pandalone gui
```

3.3 Excel usage

> **Attention:** Excel-integration requires Python-3 and *Windows* or *OS X*!

In *Windows* and *OS X* you may utilize the excellent xlwings library to use Excel files for providing input and output to the experiment.

To create the necessary template-files in your current-directory you should enter:

```
$ pandalone excel
```

You could type instead `pandalone excel` *`file_path`* to specify a different destination path.

[TBD]

3.4 Python usage

Example python REPL (Read-Eval-Print Loop) example-commands are given below that setup and run an *experiment*.

First run **python** or **ipython** and try to import the project to check its version:

```
>>> import pandalone

>>> pandalone.__version__           ## Check version once more.
'0.1.10.dev0'

>>> pandalone.__file__              ## To check where it was installed.
/usr/local/lib/site-package/pandalone-...
```

If everything works, create the *data-tree* to hold the input-data (strings and numbers). You assemble data-tree by the use of:

- sequences,
- dictionaries,
- `pandas.DataFrame`,
- `pandas.Series`, and
- URI-references to other data-trees.

[TBD]

3.5 Discussion

CHAPTER 4

Getting Involved

This project is hosted in **github**. To provide feedback about bugs and errors or questions and requests for enhancements, use github's Issue-tracker.

4.1 Sources & Dependencies

To get involved with development, you need a POSIX environment to fully build it (*Linux*, *OSX* or *Cygwin* on *Windows*).

First you need to download the latest sources:

```
$ git clone https://github.com/pandalone/pandalone.git pandalone.git
$ cd pandalone.git
```

Virtualenv

You may choose to work in a *virtualenv* (isolated Python environment), to install dependency libraries isolated from system's ones, and/or without *admin-rights* (this is recommended for *Linux*/*Mac OS*).

> **Attention:** If you decide to reuse stystem-installed packages using `--system-site-packages` with `virtualenv <= 1.11.6` (to avoid, for instance, having to reinstall *numpy* and *pandas* that require native-libraries) you may be bitten by bug #461 which prevents you from upgrading any of the pre-installed packages with **`pip`**.

Liclipse IDE

Within the sources there are two sample files for the comprehensive LiClipse IDE:

- `eclipse.project`
- `eclipse.pydevproject`

Remove the `eclipse` prefix, (but leave the dot()) and import it as "existing project" from Eclipse's `File` menu.

Another issue is caused due to the fact that LiClipse contains its own implementation of *Git*, *EGit*, which badly interacts with unix *symbolic-links*, such as the `docs/docs`, and it detects working-directory changes even after a fresh checkout. To workaround this, Right-click on the above file *Properties* → *Team* → *Advanced* → *Assume Unchanged*

Then you can install all project's dependencies in *'development mode* using the `setup.py` script:

```
$ python setup.py --help                              ## Get help for this script.
Common commands: (see '--help-commands' for more)

  setup.py build      will build the package underneath 'build/'
  setup.py install    will install the package

Global options:
  --verbose (-v)      run verbosely (default)
  --quiet (-q)        run quietly (turns verbosity off)
  --dry-run (-n)      don't actually do anything
...

$ python setup.py develop                              ## Also installs dependencies into project's
$ python setup.py build                                ## Check that the project indeed builds ok.
```

You should now run the test-cases to check that the sources are in good shape:

```
$ python setup.py test
```

Note: The above commands installed the dependencies inside the project folder and for the *virtual-environment*. That is why all build and testing actions have to go through `python setup.py some_cmd`.

If you are dealing with installation problems and/or you want to permantly install dependant packages, you have to *deactivate* the virtual-environment and start installing them into your *base* python environment:

```
$ deactivate
$ python setup.py develop
```

or even try the more *permanent* installation-mode:

```
$ python setup.py install                # May require admin-rights
```

4.2 Design

See architecture live-document.

For submitting code, use `UTF-8` everywhere, unix-eol(`LF`) and set `git --config core.autocrlf = input`.

The typical development procedure is like this:

1. Modify the sources in small, isolated and well-defined changes, i.e. adding a single feature, or fixing a specific bug.
2. Add test-cases “proving” your code.
3. Rerun all test-cases to ensure that you didn’t break anything, and check their *coverage* remain above 80%:

```
        $ python setup.py test_code_cover
```

> **Tip:** You can enter just: `python setup.py test_all` instead of the above cmd-line since it has been *aliased* in the `setup.cfg` file. Check this file for more example commands to use during development.

4. To see the rendered results of the documents, issue the following commands and read the result html at `build/sphinx/html/index.html`:

```
$ python setup.py build_sphinx                  # Builds html docs
$ python setup.py build_sphinx -b doctest       # Checks if python-code embeded in comme
```

5. If there are no problems, commit your changes with a descriptive message.
6. Repeat this cycle for further modifications.
7. If you made a rather important modification, update also documentation (i.e. README.rst) the `CHANGES` and `AUTHORS`.
8. When you are finished, push the changes upstream to *github* and make a *merge_request*. You can check whether your merge-request indeed passed the tests by checking its build-status, on both integration site: Travis: , Appveyor: .

Hint: Skim through these small guides:

- IPython developer's documentantion: The perfect pull request
- Effective pull requests and other good practices for teams using github

---

4.3 Discussion

CHAPTER 5

API reference

These are the modules in maturity order:

xleash	A mini-language for “throwing the rope” around rectangular areas of Excel-sheets.
mappings	Hierarchical string-like objects useful for indexing, that can be rename/relocated at a la
pandata	A *pandas-model* is a tree of strings, numbers, sequences, dicts, pandas instances and re
components	Defines the building-blocks of a “model”:

5.1 Module: pandalone.xleash

A mini-language for “throwing the rope” around rectangular areas of Excel-sheets.

5.1.1 About

Any *decent* dataset is stored in **csv**. Consequently, many datasets are still trapped in excel-sheets.

XLeash defines a url-fragment notation (*xl-ref*) that renders the *capturing* of tables from sheets as practical as reading a **csv**, even when the exact position of those tables are not known beforehand.

An additional goal is to apply the same *lassoing* operation recursively, to build *data-trees*. For that end, the syntax supports *filter* transformations such as:

- setting the dimensionality of the result tables,
- creating higher-level objects from 2D *capture-rect* (dictionaries, *numpy-arrays* & *dataframes*).

It is based on xlrd library but also checked for compatibility with xlwings *COM-client* library. It requires *numpy* and (optionally) *pandas*. It is developed on python-3 but also tested on python-2 for compatibility.

5.1.2 Overview

The *xl-ref* notation extends ordinary *A1* and *RC* excel *coordinates* with conditional *traversing* operations, based on the cell’s empty/full *state*. For instance, to extract a contigious table near the `A1` cell, and make a `pandas.DataFrame` out of it use this:

```
from pandalone import xleash

df = xleash.lasso('path/to/workbook.xlsx#A1(DR):..(DR):LU:["df"]')
```

Xl-ref Syntax

```
[<url>]#[<sheet>!][<1st-edge>][:[<2nd-edge>][:<expansions>]][:<filters>]
```

- See *edge*, *expansion-moves*, *filters* for details.
- Missing *edges* are implicitly replaced by `^^:__` (top-left/bottom-right).
- Spaces are allowed only in *filters*.

Annotated Example

```
target-moves-----
landing-cell--  |

            #C3(UL):..(RD):RULD:["pipe": ["odict", "recursive"]]
             --- --- - ----------------------------
1st-edge-------      |     |                   |
2nd-edge--------------     |                   |
expansions------------------                   |
filters----------------------------------------
```

Which means:

1. *Target* the *1st edge* of the *capture-rect* by starting from `C3` *landing-cell*. If it is a *full-cell*, stop, otherwise start moving above and to the left of `C3` and stop on the first *full-cell*;
2. continue from the last *target* and travel the *exterior* row and column right and down, stopping on their last *full-cell*;
3. *capture* all the cells between the 2 targets.
4. try *expansions* to all directions if any neighbouring *full-cell*;
5. finally *filter* the values of the *capture-rect* to wrap them up in an ordered- dictionary, and dive into its values searching for *xl-ref*, and replace them.

Basic Usage

The simplest way to *lasso* a *xl-ref* is through *`lasso()`*. A common task is capturing all sheet data but without any bordering nulls:

```
>>> from pandalone import xleash

>>> values = xleash.lasso('path/to/workbook.xlsx#:')
```

Assuming that the *full-cell* of the 1st sheet of the workbook on disk are those marked with `'X'`, then the result *capture-rect* of the above call would be a 2D *list-of-lists* with the values contained in `C2:E4`:

```
  A B C D E
1    -----
2    |    X|
3    |X    |
4    |  X  |
5    -----
```

If you do not wish to let the library read your workbooks, you can invoke the function with a pre-loaded sheet. Here we will use the utility *`ArraySheet`*:

```
>>> sheet = xleash.ArraySheet([[None, None,  'A',   None],
...                            [None, 2.2,   'foo', None],
...                            [None, None,   2,    None],
...                            [None, None,   None, 3.14],
... ])
>>> xleash.lasso('#A1(DR):..(DR):RULD', sheet=sheet)
[[None, 'A'],
 [2.2, 'foo'],
 [None, 2]]
```

This *capture-rect* in this case was *B1* and *C3* as can be seen by inspecting the `st` and `nd` fields of the full `Xlref` results returned:

```
>>> xleash.lasso('#A1(DR):..(DR):RULD', sheet=sheet, return_lasso=True)
Lasso(xl_ref='#A1(DR):..(DR):RULD',
      url_file=None,
      sh_name=None,
      st_edge=Edge(land=Cell(row='1', col='A'), mov='DR', mod=None),
      nd_edge=Edge(land=Cell(row='.', col='.'), mov='DR', mod=None),
      exp_moves='RULD',
      call_spec=None,
      sheet=ArraySheet(SheetId(book='wb', ids=['sh', 0]),
                           [[None None 'A' None]
                            [None 2.2 'foo' None]
                            [None None 2 None]
                            [None None None 3.14]]),
      st=Coords(row=0, col=1),
      nd=Coords(row=2, col=2),
      values=[[None, 'A'],
              [2.2, 'foo'],
              [None, 2]],
      base_coords=None,
      ...
```

For controlling explicitly the configuration parameters and the opening of workbooks, use separate instances of *Ranger* and *SheetsFactory*, that are the workhorses of this library:

```
>>> with xleash.SheetsFactory() as sf:
...     sf.add_sheet(sheet, wb_ids='foo_wb', sh_ids='Sheet1')
...     ranger = xleash.Ranger(sf, base_opts={'verbose': True})
...     ranger.do_lasso('foo_wb#Sheet1!__').values
3.14
```

Notice that it returned a scalar value since we specified only the *1st edge* as '__', which points to the bottom row and most-left column of the sheet.

Alternatively you can call the *make_default_Ranger()* for extending library's defaults.

API

- User-facing higher-level functionality:

lasso(xlref[, sheets_factory, base_opts, ...])	High-level function to *lasso* around spreadsheet's rect-re
Ranger(sheets_factory[, base_opts, ...])	The director-class that performs all stages required for "
Ranger.do_lasso(xlref, **context_kwds)	The director-method that does all the job of hrowing a *la*
Lasso(xl_ref, url_file, sh_name, st_edge, ...)	All the fields used by the algorithm, populated stage-by-s
make_default_Ranger([sheets_factory, ...])	Makes a defaulted *Ranger*.
get_default_opts([overrides])	Default *opts* used by *lasso()* when constructing its inter

- Related to *capturing* algorithm:

resolve_capture_rect(states_matrix, ...[, ...])	Performs *targeting*, *capturing* and *expansions* based
coords2Cell(row, col)	Make *A1* `Cell` from *resolved* coords, with rudimentar
EmptyCaptureException	Thrown when *targeting* fails.

get_default_filters([overrides])	The default available *filters* used by `lasso()` when constructing
xlwings_dims_call_spec()	A list *call-spec* for `_redim_filter()` *filter* that imitates results of

- Related to parsing and basic structure used throughout: .. currentmodule:: pandalone.xleash._parse .. autosummary:

```
    parse_xlref
    parse_expansion_moves
    parse_call_spec
    Cell
    Coords
    Edge
```

- **IO** back-end functionality:

_sheets.SheetsFactory()	A caching-store of `ABCSheet` instances, serving t
_sheets.ABCSheet.read_rect(st, nd)	Fecth the actual values from the backend Excel-
_sheets.ArraySheet(arr[, ids, ids])	A sample `ABCSheet` made out of 2D-list or numpy
_sheets.ABCSheet	A delegating to backend factory and sheet-wrap
_xlrd.XlrdSheet(sheet, book_fname[, epoch1904])	The *xlrd* workbook wrapper required by xleash l
_xlrd.open_sheet(wb_url, sheet_id, opts)	Opens the local or remote `wb_url` *xlrd* workbook

More Syntax Examples

Another typical but more advanced case is when a sheet contains a single table with a "header"-row and a "index"-column. There are (at least) 3 ways to do it, beyond specifying the exact *coordinates*:

```
  A B C D E
1  -------     B2:E4          ## Exact referencing.
2  |  X X X|     ^^.__          ## From top-left full-cell to bottom-right.
3  |X X X X|     A1(DR):__:U1   ## Start from A1 and move down and right
3  |X X X X|                    #    until B3; capture till bottom-left;
4  |X X X X|                    #    expand once upwards (to header row).
   -------     A1(RD):__:L1   ## Start from A1 and move down by row
                                #    until C1; capture till bottom-left;
                                #    expand once left (to index column).
```

Note that if `B1` were full, the results would still be the same, because `?` expands only if any full-cell found in row/column.

In case where the sheet contains more than one *disjoint* tables, the bottom-left cell of the sheet would not coincide with table-end, so the handy last two *xl-ref* above would not work.

For that we may resort to *dependent* referencing for the *2nd edge*, and define its position in relation to the *1st target*:

```
  A B C D E
1  -----    _^:..(LD+):L1     ## Start from top-right(E2) and target left
2  |  X X|                    #    left(D2); from there capture left-down
3  |X X X|                    #    till 1st empty-cell(C4, regardless of
4  |X X X|                    #    col/row order); expand left once.
```

```
   -----    ^_(U):..(UR):U1   ## Start from B5 and target 1st cell up;
5         X                     #    capture from there till D3; expand up.
```

In the presence of *empty-cell* breaking the *exterior* row/column of the *1st landing-cell*, the capturing becomes more intricate:

```
  A B C D E
1  -----      B2:D_
2  |  X X|      A1(RD):..(RD):L1D
3  |X X  |      D_:^^
3  |X    |      A^(DR):D_:U
4  |  X  |X
   -----


  A B C D E
1    ---      ^^(RD):..(RD)
2    |X X|      _^(R):^.(DR)
3   X|X  |
     ---
3   X
4     X   X


  A B C D E
1  ---        B2:C4
2  |  X|X       A1(RD):^_
3  |X X|        C_:^^
3  |X  |        A^(DR):C_:U
4  |  X|  X     ^^(RD):..(D):D
   ---         D2(L+):^_
```

See also:

Example spreadsheet: `xleash.xlsx`

5.1.3 Definitions

lasso, lassoing It may denote 3 things:

- the whole procedure of *parsing* the *xl-ref* syntax, *capturing* values from spreadsheet rect-regions and sending them through any *filters* specified in the xl-ref;
- the *`lasso()`* and/or *`Ranger.do_lasso()`* functions performing the above job;
- the *`Lasso`* storing intermediate and final results of the above algorithm.

xl-ref Any url with its fragment abiding to the syntax defined herein.

- The *fragment* describes how to *capture* rects from excel-sheets, and it is composed of 2 *edge* references followed by *expansions* and *filters*.
- The *file-part* should resolve to an excel-file.

parse, parsing The stage where the input string gets splitted and checked for validity against the *xl-ref* syntax.

edge An *edge* might signify:

- the syntactic construct of the *xl-ref*, composed of a pair of row/column *coordinates*, optionally followed by parenthesized *target-moves*, like `A1(LU)`;
- the bounding cells of the *target-rect*;
- the bounding cells of the *capture-rect*.

In all cases above there are 2 instances; the *1st* and *2nd*.

1st, 2nd It may refer to the *1st*/*2nd*:

- *edge* of some *xl-ref*;
- *landing-cell* of an *edge*;
- *target-cell* of an *edge*;
- *capture-cell* of a *capture-rect*.

The *1st-edge' supports 'absolute' 'coordinates' only, while the *2nd-edge* supports also *dependent* ones from the *1st* *target-cell*.

landing-cell The cell identified by the *coordinates* of the *edge* alone.

target-cell, target-rect The bounding *cell* identified after applying *target-moves* on the *landing-cell*.

target, targeting The process of identifying any *target-cell* bounding the *target-rect*.

- The search for the *target-cell* starts from the *landing-cell*, follows the specified *target-moves*, and ends when a *state-change* is detected on an *exterior* column or row, according to the enacted *termination-rule*.
- Failure to identify any target-cell raises a *`EmptyCaptureException`* which is subsequently translated as *empty* *capture-rect* by *`Ranger`* when *opts* contain `{"no_empty": false}` (default).
- The process is followed by *expansions* to identify the *capture-rect*.

Note that in the case of a *dependent* *2nd* *edge*, the *target-rect* would always be the same, irrespective of whether *target-moves* denoted a row-by-row or column-by-column traversal.

capture, capturing It is the overall procedure of:

1. *targeting* both *edge* refs to come up with the *target-rect*;
2. performing *expansions* to identify the *capture-rect*;
3. extracting the values and feed them to *filters*.

capture-rect, capture-cell The *rectangular-area* of the sheet denoted by the two *capture-cells* identified by *capturing*, that is, after applying *expansions* on *target-rect*.

directions The 4 primitive *directions* that are denoted with one of the letters `LURD`. Thee are used to express both *target-moves* and *expansions*.

coordinate, coordinates Any pair of a cell/column *coordinates* specifying cell positions, (i.e. *landing-cell*, *target-cell*, bounds of the *capture-rect*) written as the first part of the *edge* syntax, or implicitely resolved. They can be expressed in `A1` or `RC` format or as a zero-based `(row, col)` tuple (*num*). Each *coordinate* might be *absolute* or *dependent*, independently.

traversing, traversal-operations Either the *target-moves* or the *expansion-moves* that comprise the *capturing*.

target-moves Specify the cell traversing order while *targeting* using primitive *directions* pairs. The pairs `UD` and `LR` (and their inverse) are invalid. I.e. `DR` means:

> *"Start going right, column-by-column, traversing each column from top to bottom."*

move-modifier One of `+` and `-` chars that might trail the *target-moves* and define which the *termination-rule* to follow if *landing-cell* is *full-cell*, i.e. `A1(RD+)`

expansions, expansion-moves Due to *state-change* on the 'exterior' cells the *capture-rect* might be smaller that a wider contigious but "convex" rectangular area.

The *expansions* attempt to remedy this by providing for expanding on arbitrary *directions* accompanied by a multiplicity for each one. If multiplicity is unspecified, infinite assumed, so it expands until an empty/full row/column is met.

absolute Any cell row/col identified with column-characters, row-numbers, or the following special-characters:

- ^ The top/Left full-cell *coordinate*.
- _ The bottom/right full-cell *coordinate*.

dependent, base-cell A *landing-cell* whose any *coordinate* is identified with a dot(`.`), which resolves to the *base-coordinate* depending on which *edge* it is referring to:

- *1st* edge: The coordinates of the base_cell field of the *Lasso* given to the *`Ranger.do_lasso()`*; must not be `None`.
- *2nd* edge: the *target-cell* coordinates of the *1st* edge.

An *edge* might contain a "mix" of *absolute* and *dependent* coordinates.

state, full-cell, empty-cell A cell is *full* when it is not *empty* / *blank* (in Excel's parlance).

states-matrix A boolean matrix denoting the *state* of the cells, having the same size as a sheet it was derived from.

state-change Whether we are traversing from an *empty-cell* to a *full-cell*, and vice-versa, while *targeting*.

termination-rule The condition to stop *targeting* while traversing from *landing-cell*. The are 2 rules: *search-same* and *search-opposite*.

See also:

Check *Target-termination enactment* for the enactment of the rules.

search-opposite The *target-cell* is the FIRST *full-cell* found while traveling from the *landing-cell* according to the *target-moves*.

search-same The coordinates of the *target-cell* are given by the LAST *full-cell* on the *exterior* column/row according to the *target-moves*; the order of the moves is insignificant in that case.

exterior The *column* and the *row* of the *landing-cell*; the *search-same termination-rule* gets to be triggered by 'full-cells' only on them.

filter, filters The last part of the *xl-ref* specifying predefined functions to apply for transforming the cell-values of *capture-rect*, abiding to the **json** syntax. They may be *bulk* or *element-wise*.

bulk, bulk-filter A *filter* treating *capture-rect* values as a whole , i.e. transposing arrays, is_empty

element-wise, element-wise-filter A *filter* diving into *capture-rect* values, i.e. for python-eval.

call-specifier, call-spec The structure to specify some function call in the *filter* part; it can either be a json *string*, *list* or *object* like that:

- string: `"func_name"`
- list: `["func_name", ["arg1", "arg2"], {"k1": "v1"}]` where the last 2 parts are optional and can be given in any order;
- object: `{"func": "func_name", "args": ["arg1"], "kws": {"k":"v"}}` where the `args` and `kws` are optional.

If the outer-most filter is a dictionary, a `'pop'` kwd is popped-out as the *opts*.

opts Key-value pairs affecting the *lassoing* (i.e. opening xlrd-workbooks). Read the code to be sure what are the available choices :-(They are a combination of options specified in code (i.e. in the *lasso()* and those extracted from *filters* by the 'opts' key, and they are stored in the *Lasso*.

5.1.4 Details

Target-moves

There are 12 *target-moves* named with a *single* or a *pair* of letters denoting the 4 primitive *directions*, LURD:

```
        U
 UL------UR
LU     |||     RU
      |||
 |     |||     |
 -----|-----
L------X------R
 -----|-----
 |     |||     |
      |||
LD     |||     RD
 DL------DR
        D

- The 'X' at the center points the starting cell.
```

So a RD move means *"traverse cells first by rows then by columns"*, or more lengthy description would be:

> *"Start moving *right* till 1st state change, and then move *down* to the next row, and start traversing right again."*

Target-cells

Using these moves we can identify a *target-cell* in relation to the *landing-cell*. For instance, given this xl-sheet below, there are multiple ways to identify (or target) the non-empty values X, below:

```
  A B C D E F
1
2
3     X        ------ C3    A1(RD)   _^(L)      F3(L)
4         X    ------ E4    A4(R)    _4(L)      D1(DR)
5   X          ------ B5    A1(DR)   A_(UR)     _5(L)
6           X  ------ F6    __       _^(D)      A_(R)

- The 'X' signifies non-empty cells.
```

So we can target cells with "absolute coordinates", the usual A1 notation, augmented with the following special characters:

- undesrcore(_) for bottom/right, and
- accent(^) for top/left

columns/rows of the sheet with non-empty values.

When no LURD moves are specified, the target-cell coinceds with the starting one.

See also:

Target-termination enactment section

Capturing

To specify a complete *capture-rect* we need to identify a 2nd cell. The 2nd target-cell may be specified:

- either with *absolute* coordinates, as above, or
- with *dependent* coords, using the dot(`.`) to refer to the 1st cell.

In the above example-sheet, here are some ways to specify refs:

```
  A  B C D E  F
1

2
      -----
   ---   |
3  |  |X|   |
   |-----
4  || | |  X||
   || ------- C3:E4   A1(RD):..(RD)   _^(L):..(DR)   _4(L):A1(RD)
5  ||X  |    |
   |---------- B4:E5   A_(UR):..(RU)   _5(L):1_(UR)    E1(D):A.(DR)
6  |    |     X
   ------------ B3:C6   A1(RD):^_       ^^:C_           C_:^^
```

Warning: Of course, the above rects WILL FAIL since the *target-moves* will stop immediately due to `X` values being surrounded by empty-cells.
But the above diagram was to just convey the general idea. To make it work, all the in-between cells of the peripheral row and columns should have been also non-empty.

Note: The *capturing* moves from *1st target-cell* to *2nd target-cell* are independent from the implied *target-moves* in the case of *dependent* coords.

More specifically, the *capturing* will always fetch the same values regardless of "row-first" or "column-first" order; this is not the case with *targeting* (`LURD`) moves.

For instance, to capture `B4:E5` in the above sheet we may use `_5(L):E.(U)`. In that case the target cells are `B5` and `E4` and the *target-moves* to reach the 2nd one are `UR` which are different from the `U` specified on the 2nd cell.

Target-termination enactment

The guiding principle for when to enact each rule is to always *capture* a matrix of *full-cell*.

- If the *landing-cell* is *empty-cell*, always *search-opposite*, that is, stop on the first *full-cell*.
- When the *landing-cell* is *full-cell*, it depends on the 'move-modifier':
  - If `+` exists, apply *search-same*.
  - If `-` exists, stop on *landing-cell*.
  - If no modifier, behave like `'-'` (`stop on 'landing-cell'`) `except when on a '2nd' edge with both its coordinates 'dependent'` (`..''`), where the *search-same* is applied

So, both *move-modifier* apply only when *landing-cell* is *full-cell* , and `-` actually makes sense only when *2nd* edge is *dependent*.

If the termination conditions is not met, an *EmptyCaptureException* is raised, which is translated as *empty* *capture-rect* by *Ranger* when *opts* contain `{"no_empty": false}` (default).

Expansions

Captured-rects ("values") may be limited due to *empty-cell* in the 1st row/column traversed. To overcome this, the xl-ref may specify *expansions* directions using a 3rd `:`-section like that:

```
_5(L):1_(UR):RDL1U1
```

This particular case means:

> *"Try expanding Right and Down repeatedly and then try once Left and Up."*

Expansion happens on a row-by-row or column-by-column basis, and terminates when a full empty(or non-empty) line is met.

Example-refs are given below for capturing the 2 marked tables:

```
  A  B C D E F  G
1
   -----------
   |---------|
2  ||  1 X X  ||
   ||         ||
3  ||X X   X X||
   ||         ||
4  ||X X X 2 X||
   ||         ||
5  ||X   X X X||
   ----------- A1(RD):..(RD):DRL1
6   |X        |
    ------------ A1(RD):..(RD):L1DR       A_(UR):^^(RD)
7               X

- The 'X' signify non-empty cells.
- The '1' and '2' signify the identified target-cells.
```

pandalone.xleash.**resolve_capture_rect**(*states_matrix*, *up_dn_margins*, *st_edge*, *nd_edge=None*, *exp_moves=None*, *base_coords=None*)

Performs *targeting*, *capturing* and *expansions* based on the *states-matrix*.

To get the margin_coords, use one of:

- *ABCSheet.get_margin_coords()*
- *io._sheets.margin_coords_from_states_matrix()*

Its results can be fed into `read_capture_values()`.

Parameters

- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use *ABCSheet.get_states_matrix()* to derrive it.
- **Coords) up_dn_margins** ((*Coords*,) – the top-left/bottom-right coords with full-cells
- **st_edge** (Edge) – "uncooked" as matched by regex

- **nd_edge** (Edge) – "uncooked" as matched by regex
- **or none exp_moves** (*list*) – Just the parsed string, and not None.
- **base_coords** (Coords) – The base for a *dependent 1st* edge.

Returns a (Coords, Coords) with the 1st and 2nd *capture-cell* ordered from top-left –> bottom-right.

Return type tuple

Raises **EmptyCaptureException** – When *targeting* failed, and no *target* cell identified.

Examples::

```
>>> from pandalone.xleash import Edge, margin_coords_from_states_matrix
```

```
>>> states_matrix = np.array([
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 1, 1, 1],
...     [0, 0, 1, 0, 0, 1],
...     [0, 0, 1, 1, 1, 1]
... ], dtype=bool)
>>> up, dn = margin_coords_from_states_matrix(states_matrix)
```

```
>>> st_edge = Edge(Cell('1', 'A'), 'DR')
>>> nd_edge = Edge(Cell('.', '.'), 'DR')
>>> resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
(Coords(row=3, col=2), Coords(row=4, col=2))
```

Using dependenent coordinates for the 2nd edge:

```
>>> st_edge = Edge(Cell('_', '_'), None)
>>> nd_edge = Edge(Cell('.', '.'), 'UL')
>>> rect = resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
>>> rect
(Coords(row=2, col=2), Coords(row=4, col=5))
```

Using sheet's margins:

```
>>> st_edge = Edge(Cell('^', '_'), None)
>>> nd_edge = Edge(Cell('_', '^'), None)
>>> rect == resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
True
```

Walking backwards:

```
>>> st_edge = Edge(Cell('^', '_'), 'L')          # Landing is full, so 'L' ignored.
>>> nd_edge = Edge(Cell('_', '_'), 'L', '+')    # '+' or would also stop.
>>> rect == resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
True
```

class pandalone.xleash.**ABCSheet**

Bases: object

A delegating to backend factory and sheet-wrapper with utility methods.

Parameters

- **_states_matrix** (*np.ndarray*) – The *states-matrix* cached, so recreate object to refresh it.
- **_margin_coords** (*dict*) – limits used by _resolve_cell(), cached, so recreate object to refresh it.

Resource management is outside of the scope of this class, and must happen in the backend workbook/sheet instance.

xlrd examples:

```
>>> import xlrd
>>> with xlrd.open_workbook(self.tmp) as wb:
...     sheet = xleash.xlrdSheet(wb.sheet_by_name('Sheet1'))
...     ## Do whatever
```

win32 examples:

```
>>> with dsgdsdsfsd as wb:
...     sheet = xleash.win32Sheet(wb.sheet['Sheet1'])
TODO: Win32 Sheet example
```

_close()
Override it to release resources for this sheet.

_close_all()
Override it to release resources this and all sibling sheets.

_read_margin_coords()
Override if possible to read (any of the) limits directly from the sheet.

Returns the 2 coords of the top-left & bottom-right full cells; anyone coords can be None. By default returns `(None, None)`.

Return type (Coords, Coords)

Raise EmptyCaptureException if sheet empty

_read_states_matrix()
Read the *states-matrix* of the wrapped sheet.

Returns A 2D-array with `False` wherever cell are blank or empty.

Return type ndarray

get_margin_coords()
Extract (and cache) margins either internally or from *margin_coords_from_states_matrix()*.

Returns the resolved top-left and bottom-right *xleash.Coords*

Return type tuple

Raise EmptyCaptureException if sheet empty

get_sheet_ids()

Returns a 2-tuple of its wb-name and a sheet-ids of this sheet i.e. name & indx

Return type SheetId or None

get_states_matrix()
Read and cache the *states-matrix* of the wrapped sheet.

Returns A 2D-array with `False` wherever cell are blank or empty.

Return type ndarray

Raise EmptyCaptureException if sheet empty

open_sibling_sheet(*sheet_id*, *opts=None*)
Return a sibling sheet by the given index or name

read_rect(*st*, *nd*)
Fecth the actual values from the backend Excel-sheet.

Parameters

- **st** (Coords) – the top-left edge, inclusive
- **None nd** (*Coords,*) – the bottom-right edge, inclusive(!); when None, must return a scalar value.

Returns

Depends on whether both coords are given:

- If both given, 2D list-lists with the values of the rect, which might be empty if beyond limits.
- If only 1st given, the scalar value, and if beyond margins, raise error!

Return type list

Raise EmptyCaptureException (optionally) if sheet empty

class pandalone.xleash.**ArraySheet**(*arr*, *ids=SheetId(book='wb', ids=['sh', 0])*)
Bases: *pandalone.xleash.io._sheets.ABCSheet*

A sample *ABCSheet* made out of 2D-list or numpy-arrays, for facilitating tests.

pandalone.xleash.**coords2Cell**(*row*, *col*)
Make *A1* *Cell* from *resolved* coords, with rudimentary error-checking.

Examples:

```
>>> coords2Cell(row=0, col=0)
Cell(row='1', col='A')
>>> coords2Cell(row=0, col=26)
Cell(row='1', col='AA')

>>> coords2Cell(row=10, col='.')
Cell(row='11', col='.')

>>> coords2Cell(row=-3, col=-2)
Traceback (most recent call last):
AssertionError: negative row!
```

exception pandalone.xleash.**EmptyCaptureException**
Bases: Exception

Thrown when *targeting* fails.

pandalone.xleash.**margin_coords_from_states_matrix**(*states_matrix*)
Returns top-left/bottom-down margins of full cells from a *state* matrix.

May be used by *ABCSheet.get_margin_coords()* if a backend does not report the sheet-margins internally.

Parameters **states_matrix** (*np.ndarray*) – A 2D-array with False wherever cell are blank or empty. Use *ABCSheet.get_states_matrix()* to derrive it.

Returns the 2 coords of the top-left & bottom-right full cells

Return type (Coords, Coords)

Examples::

```
>>> from io._sheets import margin_coords_from_states_matrix
```

```
>>> states_matrix = np.asarray([
...     [0, 0, 0],
...     [0, 1, 0],
...     [0, 1, 1],
...     [0, 0, 1],
... ])
>>> margins = margin_coords_from_states_matrix(states_matrix)
>>> margins
(Coords(row=1, col=1), Coords(row=3, col=2))
```

Note that the botom-left cell is not the same as `states_matrix` matrix size:

```
>>> states_matrix = np.asarray([
...     [0, 0, 0, 0],
...     [0, 1, 0, 0],
...     [0, 1, 1, 0],
...     [0, 0, 1, 0],
...     [0, 0, 0, 0],
... ])
>>> margin_coords_from_states_matrix(states_matrix) == margins
True
```

pandalone.xleash.**lasso**(*xlref*, *sheets_factory=None*, *base_opts=None*, *available_filters=None*, *return_lasso=False*, ***context_kwds*)

High-level function to *lasso* around spreadsheet's rect-regions according to *xl-ref* strings by using internally a *Ranger* .

Parameters

- **xlref** (*str*) – a string with the *xl-ref* format:

```
<url_file>#<sheet>!<1st_edge>:<2nd_edge>:<expand><js_filt>
```

  i.e.:

```
file:///path/to/file.xls#sheet_name!UPT8(LU-):_.(D+):LDL1{"dims":1}
```

- **sheets_factory** – Factory of sheets from where to parse rect-values; if unspecified, the new *SheetsFactory* created is closed afterwards. Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.
- **or None available_filters** (*dict*) – Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.
- **return_lasso** (*bool*) – If `True`, values are contained in the returned Lasso instance, along with all other artifacts of the *lassoing* procedure.

  For more debugging help, create a `Range` yourself and inspect the `Ranger.intermediate_lasso`.
- **context_kwds** (`Lasso`) – Default *Lasso* fields in case parsed ones are `None` (i.e. you can specify the sheet like that).

Variables or None base_opts (*dict*) – Opts affecting the lassoing procedure that are deep-copied and used as the base-opts for every *Ranger.do_lasso()*, whether invoked directly or recursively by `recursive_filter()`. Read the code to be sure what are the available choices. Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.

Returns Either the captured & filtered values or the final *Lasso*, depending on the `return_lassos` arg.

Example:

```
sheet = _
```

class pandalone.xleash.**Ranger**(*sheets_factory*, *base_opts=None*, *available_filters=None*)

Bases: `object`

The director-class that performs all stages required for "throwing the lasso" around rect-values.

Use it when you need to have total control of the procedure and configuration parameters, since no defaults are assumed.

The *do_lasso()* does the job.

Variables

- **sheets_factory** (SheetsFactory) – Factory of sheets from where to parse rect-values; does not close it in the end. Maybe None, but *do_lasso()* will scream unless invoked with a `context_lasso` arg containing a concrete *ABCSheet*.
- **base_opts** (*dict*) – The *opts* that are deep-copied and used as the defaults for every *do_lasso()*, whether invoked directly or recursively by `recursive_filter()`. If unspecified, no opts are used, but this attr is set to an empty dict. See *get_default_opts()*.
- **or None available_filters** (*dict*) – No filters exist if unspecified. See *get_default_filters()*.
- **intermediate_lasso** (Lasso) – A `('stage', Lasso)` pair with the last *Lasso* instance produced during the last execution of the *do_lasso()*. Used for inspecting/debuging.

_make_init_Lasso(***context_kwds*)
Creates the lasso to be used for each new *do_lasso()* invocation.

_parse_and_merge_with_context(*xlref*, *init_lasso*)
Merges xl-ref parsed-parsed_fields with `init_lasso`, reporting any errors.

Parameters **init_lasso** (Lasso) – Default values to be overridden by non-nulls. Note that `init_lasso.opts` must be a `ChainMap`, as returned by $_{m}ake_{i}nit_{L}asso()$.

Returns a Lasso with any non None parsed-fields updated

_relasso(*lasso*, *stage*, ***kwds*)
Replace lasso-values and updated `intermediate_lasso`.

_resolve_capture_rect(*lasso*, *sheet*)
Also handles *EmptyCaptureException* in case `opts['no_empty'] != False`.

do_lasso(*xlref*, ***context_kwds*)
The director-method that does all the job of hrowing a *lasso* around spreadsheet's rect-regions according to *xl-ref*.

Parameters

- **xlref** (*str*) – a string with the *xl-ref* format:

```
<url_file>#<sheet>!<1st_edge>:<2nd_edge>:<expand><js_filt>
```

i.e.:

```
                file:///path/to/file.xls#sheet_name!UPT8(LU-):_.(D+):LDL1{"dims":1}
```

- **context_kwds** (Lasso) – Default *Lasso* fields in case parsed ones are None

Returns The final *Lasso* with captured & filtered values.

Return type *Lasso*

make_call(*lasso*, *func_name*, *args*, *kwds*)
Executes a *call-spec* respecting any `lax` argument popped from `kwds`.

Parameters **lax** (*bool*) – After overlaying it on *opts*, it governs whether to raise on errors. Defaults to False (scream!).

class pandalone.xleash.**SheetsFactory**
Bases: object

A caching-store of *ABCSheet* instances, serving them based on (workbook, sheet) IDs, optionally creating them from backends.

Variables **_cached_sheets** (*dict*) – A cache of all _Spreadsheets accessed so far, keyed by multiple keys generated by *_derive_sheet_keys()*.

- To avoid opening non-trivial workbooks, use the *add_sheet()* to pre-populate this cache with them.
- The last sheet added becomes the *current-sheet*, and will be served when *xl-ref* does not specify any workbook and sheet.

Tip: For the simplest API usage, try this:

```
>>> sf = SheetsFactory()
>>> sf.add_sheet(some_sheet)
>>> lasso('A1:C3(U)', sf)
```

- The *current-sheet* is served only when wokbook-id is None, that is, the id-pair `('foo.xlsx', None)` does not hit it, so those ids are send to the cache as they are.
- To add another backend, modify the opening-sheets logic (ie clipboard), override *_open_sheet()*.
- It is a resource-manager for contained sheets, wo it can be used wth a `with` statement.

_derive_sheet_keys(*sheet*, *wb_ids=None*, *sh_ids=None*)
Retuns the product of user-specified and sheet-internal keys.

Parameters

- **wb_ids** – a single or a sequence of extra workbook-ids (ie: file, url)
- **sh_ids** – a single or sequence of extra sheet-ids (ie: name, index, None)

_open_sheet(*wb_id*, *sheet_id*, *opts*)
OVERRIDE THIS to change backend.

add_sheet(*sheet*, *wb_ids=None*, *sh_ids=None*)
Updates cache.

Parameters

- **wb_ids** – a single or sequence of extra workbook-ids (ie: file, url)

- **sh_ids** – a single or sequence of extra sheet-ids (ie: name, index, None)

close()
Closes all contained sheets and empties cache.

fetch_sheet(*wb_id*, *sheet_id*, *opts={}*, *base_sheet=None*)

Parameters **base_sheet** (ABCSheet) – The sheet used when unspecified wb_id.

pandalone.xleash.**make_default_Ranger**(*sheets_factory=None*, *base_opts=None*, *available_filters=None*)
Makes a defaulted *Ranger*.

Parameters

- **sheets_factory** – Factory of sheets from where to parse rect-values; if unspecified, a new *SheetsFactory* is created. Remember to invoke its *SheetsFactory.close()* to clear resources from any opened sheets.
- **or None base_opts** (*dict*) – Default opts to affect the lassoing, to be merged with defaults; uses *get_default_opts()*.

  Read the code to be sure what are the available choices :-(.
- **or None available_filters** (*dict*) – The available *filters* to specify a *xl-ref*. Uses *get_default_filters()* if unspecified.

For instance, to make you own sheets-factory and override options, yoummay do this:

```
>>> from pandalone import xleash

>>> with xleash.SheetsFactory() as sf:
...     xleash.make_default_Ranger(sf, base_opts={'lax': True})
<pandalone.xleash._lasso.Ranger object at
...
```

class pandalone.xleash.**XLocation**(*sheet*, *st*, *nd*, *base_coords*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *sheet*, *st*, *nd*, *base_coords*)
Create new instance of XLocation(sheet, st, nd, base_coords)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new XLocation object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new XLocation object replacing specified fields with new values

base_coords
Alias for field number 3

nd
Alias for field number 2

sheet
Alias for field number 0

st
Alias for field number 1

pandalone.xleash.**get_default_opts**(*overrides=None*)
Default *opts* used by *lasso()* when constructing its internal *Ranger*.

Parameters or None overrides (*dict*) – Any items to update the default ones.

pandalone.xleash.**get_default_filters**(*overrides=None*)
The default available *filters* used by *lasso()* when constructing its internal *Ranger*.

param dict or None overrides Any items to update the default ones.

return a dict-of-dicts with 2 items:

- *func*: a function with args: `(Ranger, Lasso, *args, **kwds)`
- *desc*: help-text replaced by `func.__doc__` if missing.

rtype dict

class pandalone.xleash.**Lasso**(*xl_ref*, *url_file*, *sh_name*, *st_edge*, *nd_edge*, *exp_moves*, *call_spec*, *sheet*, *st*, *nd*, *values*, *base_coords*, *opts*)
Bases: `tuple`

All the fields used by the algorithm, populated stage-by-stage by *Ranger*.

Parameters

- **xl_ref** (*str*) – The full url, populated on parsing.
- **sh_name** (*str*) – Parsed sheet name (or index, but still as string), populated on parsing.
- **st_edge** (Edge) – The 1st edge, populated on parsing.
- **nd_edge** (Edge) – The 2nd edge, populated on parsing.
- **st** (Coords) – The top-left targeted coords of the *capture-rect*, populated on *capturing*.‘
- **nd** (Coords) – The bottom-right targeted coords of the *capture-rect*, populated on *capturing*
- **sheet** (ABCSheet) – The fetched from factory or ranger’s current sheet, populated after *capturing* before reading.
- **values** – The excel’s table-values captured by the *lasso*, populated after reading updated while applying *filters*.
- **call_spec** – The *call-spec* derrived from the parsed filters, to be fed into *Ranger.make_call()*.
- **base_coords** (Coords) – On recursive calls it becomes the *base-cell* for the *1st edge*.
- **or ChainMap opts** (*dict*) –
  - Before `parsing`, they are just any ‘opts’ dict found in the *filters*.
  - After *parsing, a 2-map ChainMap with :attr:‘Ranger.base_opts‘ and options extracted from *filters* on top.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *xl_ref=None*, *url_file=None*, *sh_name=None*, *st_edge=None*, *nd_edge=None*, *exp_moves=None*, *call_spec=None*, *sheet=None*, *st=None*, *nd=None*, *values=None*, *base_coords=None*, *opts=None*)
Create new instance of Lasso(xl_ref, url_file, sh_name, st_edge, nd_edge, exp_moves, call_spec, sheet, st, nd, values, base_coords, opts)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new Lasso object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new Lasso object replacing specified fields with new values

base_coords
Alias for field number 11

call_spec
Alias for field number 6

exp_moves
Alias for field number 5

nd
Alias for field number 9

nd_edge
Alias for field number 4

opts
Alias for field number 12

sh_name
Alias for field number 2

sheet
Alias for field number 7

st
Alias for field number 8

st_edge
Alias for field number 3

url_file
Alias for field number 1

values
Alias for field number 10

xl_ref
Alias for field number 0

pandalone.xleash.**xlwings_dims_call_spec**()
A list *call-spec* for `_redim_filter()` *filter* that imitates results of *xlwings* library.

class pandalone.xleash.**Cell**
Bases: *pandalone.xleash._parse.Cell*

A pair of 1-based strings, denoting the “A1” coordinates of a cell.

The “num” coords (numeric, 0-based) are specified using numpy-arrays (*Coords*).

class pandalone.xleash.**Coords**(*row*, *col*)
Bases: tuple

A pair of 0-based integers denoting the “num” coordinates of a cell.

The “A1” coords (1-based coordinates) are specified using *Cell*.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *row*, *col*)
Create new instance of Coords(row, col)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new Coords object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new Coords object replacing specified fields with new values

col
Alias for field number 1

row
Alias for field number 0

class pandalone.xleash.**Edge**
Bases: *pandalone.xleash._parse.Edge*

All the infos required to *target* a cell.

An *Edge* contains *A1* *Cell* as land.

Parameters

- **land** (Cell) – the *landing-cell*
- **mov** (*str*) – use None for missing moves.
- **mod** (*str*) – one of (+, - or None)

class pandalone.xleash.**CallSpec**(*func*, *args*, *kwds*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *func*, *args=[]*, *kwds={}*)
Create new instance of CallSpec(func, args, kwds)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new CallSpec object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new CallSpec object replacing specified fields with new values

args
Alias for field number 1

func
Alias for field number 0

kwds
Alias for field number 2

pandalone.xleash.**parse_xlref**(*xlref*)
Like `_parse_xlref()` but tries also if `xlreaf` is encased by delimiter chars `/\"$%&`.

See also:

_encase_regex

5.1.5 Submodule: pandalone.xleash._parse

The syntax-parsing part *xleash*.

Prefer accessing the public members from the parent module.

class pandalone.xleash._parse.**CallSpec**(*func*, *args*, *kwds*)
Bases: `tuple`

The *call-specifier* for holding the parsed json-filters.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *func*, *args=[]*, *kwds={}*)
Create new instance of CallSpec(func, args, kwds)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new CallSpec object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new CallSpec object replacing specified fields with new values

args
Alias for field number 1

func
Alias for field number 0

kwds
Alias for field number 2

class pandalone.xleash._parse.**Cell**
Bases: *pandalone.xleash._parse.Cell*

A pair of 1-based strings, denoting the "A1" coordinates of a cell.

The "num" coords (numeric, 0-based) are specified using numpy-arrays (*Coords*).

class pandalone.xleash._parse.**Edge**

Bases: *pandalone.xleash._parse.Edge*

All the infos required to *target* a cell.

An *Edge* contains *A1* *Cell* as `land`.

Parameters

- **land** (*Cell*) – the *landing-cell*
- **mov** (*str*) – use None for missing moves.
- **mod** (*str*) – one of (`+`, `-` or None)

pandalone.xleash._parse.**Edge_new**(*row*, *col*, *mov=None*, *mod=None*, *default=None*)

Make a new *Edge* from any non-values supplied, as is capitalized, or nothing.

Parameters

- **None col** (*str,*) – ie `A`
- **None row** (*str,*) – ie `1`
- **None mov** (*str,*) – ie `RU`
- **None mod** (*str,*) – ie `+`

Returns a *Edge* if any non-None

Return type Edge, None

Examples:

```
>>> Edge_new('1', 'a', 'Rul', '-')
Edge(land=Cell(row='1', col='A'), mov='RUL', mod='-')
>>> print(Edge_new('5', '5'))
R5C5
```

No error checking performed:

```
>>> Edge_new('Any', 'foo', 'BaR', '+_&%')
Edge(land=Cell(row='ANY', col='FOO'), mov='BAR', mod='+_&%')

>>> print(Edge_new(None, None, None, None))
None
```

except were coincidental:

```
>>> Edge_new(row=0, col=123, mov='BAR', mod=None)
Traceback (most recent call last):
AttributeError: 'int' object has no attribute 'upper'

>>> Edge_new(row=0, col='A', mov=123, mod=None)
Traceback (most recent call last):
AttributeError: 'int' object has no attribute 'upper'
```

pandalone.xleash._parse.**_parse_xlref**(*xlref*)

Parse a *xl-ref* into a dict.

Parameters **xlref** (*str*) – A url-string abiding to the *xl-ref* syntax.

Returns A dict with all fields, with None with those missing.

Return type dict

Examples:

```
>>> res = parse_xlref('workbook.xlsx#Sheet1!A1(DR+):Z20(UL):L1U2R1D1:'
...                                '{"opts":{}, "func": "foo"}')
>>> sorted(res.items())
 [('call_spec', CallSpec(func='foo', args=[], kwds={})),
 ('exp_moves', 'L1U2R1D1'),
 ('nd_edge', Edge(land=Cell(row='20', col='Z'), mov='UL', mod=None)),
 ('opts', {}),
 ('sh_name', 'Sheet1'),
 ('st_edge', Edge(land=Cell(row='1', col='A'), mov='DR', mod='+')), ('url_file', 'workbook.x
```

Shortcut for all sheet from top-left to bottom-right full-cells:

```
>>> res=parse_xlref('#:')
>>> sorted(res.items())
[('call_spec', None),
 ('exp_moves', None),
 ('nd_edge', Edge(land=Cell(row='_', col='_'), mov=None, mod=None)),
 ('opts', None),
 ('sh_name', None),
 ('st_edge', Edge(land=Cell(row='^', col='^'), mov=None, mod=None)),
 ('url_file', None),
 ('xl_ref', '#:')]
```

Errors:

```
>>> parse_xlref('A1(DR)Z20(UL)')
Traceback (most recent call last):
SyntaxError: No fragment-part (starting with '#'): A1(DR)Z20(UL)

>>> parse_xlref('#A1(DR)Z20(UL)')          ## Missing ':'.
Traceback (most recent call last):
SyntaxError: Not an xl-ref syntax: A1(DR)Z20(UL)
```

But as soon as syntax is matched, subsequent errors raised are `ValueErrors`:

```
>>> parse_xlref("#A1:B1:{'Bad_JSON_str'}")
Traceback (most recent call last):
ValueError: Filters are not valid JSON:
Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
  JSON:
{'Bad_JSON_str'}
```

pandalone.xleash._parse.**_parse_xlref_fragment**(*xlref_fragment*)

Parses a *xl-ref* fragment.

Parameters **xlref_fragment** (*str*) – the url-fragment part of the *xl-ref* string, including the '#' char.

Returns

dictionary containing the following parameters:

- sheet: (str, int, None) i.e. `sheet_name`
- st_edge: (Edge, None) the 1st-ref, with raw cell i.e. `Edge(land=Cell(row='8', col='UPT'), mov='LU', mod='-')`
- nd_edge: (Edge, None) the 2nd-ref, with raw cell i.e. `Edge(land=Cell(row='_', col='.'), mov='D', mod='+')`
- exp_moves: (sequence, None), as i.e. `LDL1` parsed by `parse_expansion_moves()`
- js_filt: dict i.e. `{"dims: 1}`

Return type dict

Examples:

```
>>> res = _parse_xlref_fragment('Sheet1!A1(DR+):Z20(UL):L1U2R1D1:'
...                             '{"opts":{}, "func": "foo"}')
>>> sorted(res.items())
[('call_spec', CallSpec(func='foo', args=[], kwds={})),
 ('exp_moves', 'L1U2R1D1'),
 ('nd_edge', Edge(land=Cell(row='20', col='Z'), mov='UL', mod=None)),
 ('opts', {}),
 ('sh_name', 'Sheet1'),
 ('st_edge', Edge(land=Cell(row='1', col='A'), mov='DR', mod='+'))]
```

Shortcut for all sheet from top-left to bottom-right full-cells:

```
>>> res=_parse_xlref_fragment(':')
>>> sorted(res.items())
[('call_spec', None),
 ('exp_moves', None),
 ('nd_edge', Edge(land=Cell(row='_', col='_'), mov=None, mod=None)),
 ('opts', None),
 ('sh_name', None),
 ('st_edge', Edge(land=Cell(row='^', col='^'), mov=None, mod=None))]
```

Errors:

```
>>> _parse_xlref_fragment('A1(DR)Z20(UL)')
Traceback (most recent call last):
SyntaxError: Not an xl-ref syntax: A1(DR)Z20(UL)
```

pandalone.xleash._parse.**_repeat_moves**(*moves*, *times=None*)

Returns an iterator that repeats `moves` x `times`, or infinite if unspecified.

Used when parsing primitive *directions*.

Parameters

- **moves** (*str*) – the moves to repeat ie `RU1D?`
- **times** (*str*) – N of repetitions. If `None` it means infinite repetitions.

Returns An iterator of the moves

Return type

iterator

Examples:

```
>>> list(_repeat_moves('LUR', '3'))
['LUR', 'LUR', 'LUR']
>>> list(_repeat_moves('ABC', '0'))
[]
>>> _repeat_moves('ABC')  ## infinite repetitions
repeat('ABC')
```

pandalone.xleash._parse.**parse_call_spec**(*call_spec_values*)

Parse *call-specifier* from json-filters.

Parameters **call_spec_values** – This is a *non-null* structure specifying some function call in the `filter` part, which it can be either:

- string: `"func_name"`

- list: `["func_name", ["arg1", "arg2"], {"k1": "v1"}]` where the last 2 parts are optional and can be given in any order;
- object: `{"func": "func_name", "args": ["arg1"], "kwds": {"k":"v"}}` where the `args` and `kwds` are optional.

Returns the 3-tuple `func, args=(), kwds={}` with the defaults as shown when missing.

`pandalone.xleash._parse.`**`parse_expansion_moves`**(*exp_moves*)

Parse rect-expansion into a list of dir-letters iterables.

Parameters **exp_moves** – A string with a sequence of primitive moves: es. L1U1R1D1

Returns A list of primitive-dir chains.

Return type list

Examples:

```
>>> res = parse_expansion_moves('lu1urd?')
>>> res
[repeat('L'), repeat('U', 1), repeat('UR'), repeat('D', 1)]

# infinite generator
>>> [next(res[0]) for i in range(10)]
['L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L', 'L']

>>> list(res[1])
['U']

>>> parse_expansion_moves('1LURD')
Traceback (most recent call last):
ValueError: Invalid rect-expansion(1LURD) due to:
        'NoneType' object has no attribute 'groupdict'
```

`pandalone.xleash._parse.`**`parse_xlref`**(*xlref*)

Like `_parse_xlref()` but tries also if `xlreaf` is encased by delimiter chars `/\"$%&`.

See also:

_encase_regex

5.1.6 Submodule: pandalone.xleash.io

Backends for opening sheets from various sources.

5.1.7 Submodule: pandalone.xleash.io._sheets

The algorithmic part of *capturing*.

Prefer accessing the public members from the parent module.

class `pandalone.xleash.io._sheets.`**`ABCSheet`**

Bases: `object`

A delegating to backend factory and sheet-wrapper with utility methods.

Parameters

- **`_states_matrix`** (*np.ndarray*) – The *states-matrix* cached, so recreate object to refresh it.
- **`_margin_coords`** (*dict*) – limits used by `_resolve_cell()`, cached, so recreate object to refresh it.

Resource management is outside of the scope of this class, and must happen in the backend workbook/sheet instance.

xlrd examples:

```
>>> import xlrd
>>> with xlrd.open_workbook(self.tmp) as wb:
...     sheet = xleash.xlrdSheet(wb.sheet_by_name('Sheet1'))
...     ## Do whatever
```

win32 examples:

```
>>> with dsgdsdsfsd as wb:
...     sheet = xleash.win32Sheet(wb.sheet['Sheet1'])
TODO: Win32 Sheet example
```

_close()
Override it to release resources for this sheet.

_close_all()
Override it to release resources this and all sibling sheets.

_read_margin_coords()
Override if possible to read (any of the) limits directly from the sheet.

Returns the 2 coords of the top-left & bottom-right full cells; anyone coords can be None. By default returns `(None, None)`.

Return type (Coords, Coords)

Raise EmptyCaptureException if sheet empty

_read_states_matrix()
Read the *states-matrix* of the wrapped sheet.

Returns A 2D-array with `False` wherever cell are blank or empty.

Return type ndarray

get_margin_coords()
Extract (and cache) margins either internally or from *margin_coords_from_states_matrix()*.

Returns the resolved top-left and bottom-right *xleash.Coords*

Return type tuple

Raise EmptyCaptureException if sheet empty

get_sheet_ids()

Returns a 2-tuple of its wb-name and a sheet-ids of this sheet i.e. name & indx

Return type SheetId or None

get_states_matrix()
Read and cache the *states-matrix* of the wrapped sheet.

Returns A 2D-array with `False` wherever cell are blank or empty.

Return type ndarray

Raise EmptyCaptureException if sheet empty

open_sibling_sheet(*sheet_id*, *opts=None*)
Return a sibling sheet by the given index or name

read_rect(*st*, *nd*)
Fecth the actual values from the backend Excel-sheet.

Parameters

- **st** (Coords) – the top-left edge, inclusive
- **None nd** (*Coords,*) – the bottom-right edge, inclusive(!); when None, must return a scalar value.

Returns

Depends on whether both coords are given:

- If both given, 2D list-lists with the values of the rect, which might be empty if beyond limits.
- If only 1st given, the scalar value, and if beyond margins, raise error!

Return type list

Raise EmptyCaptureException (optionally) if sheet empty

class pandalone.xleash.io._sheets.**ArraySheet**(*arr*, *ids=SheetId(book='wb', ids=['sh', 0])*)

Bases: *pandalone.xleash.io._sheets.ABCSheet*

A sample *ABCSheet* made out of 2D-list or numpy-arrays, for facilitating tests.

class pandalone.xleash.io._sheets.**SheetId**(*book*, *ids*)

Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *book*, *ids*)
Create new instance of SheetId(book, ids)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new SheetId object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new SheetId object replacing specified fields with new values

book
Alias for field number 0

ids
Alias for field number 1

class pandalone.xleash.io._sheets.**SheetsFactory**

Bases: object

A caching-store of *ABCSheet* instances, serving them based on (workbook, sheet) IDs, optionally creating them from backends.

Variables **_cached_sheets** (*dict*) – A cache of all _Spreadsheets accessed so far, keyed by multiple keys generated by *_derive_sheet_keys()*.

•To avoid opening non-trivial workbooks, use the *add_sheet()* to pre-populate this cache with them.

- The last sheet added becomes the *current-sheet*, and will be served when *xl-ref* does not specify any workbook and sheet.

Tip: For the simplest API usage, try this:

```
>>> sf = SheetsFactory()
>>> sf.add_sheet(some_sheet)
>>> lasso('A1:C3(U)', sf)
```

- The *current-sheet* is served only when wokbook-id is `None`, that is, the id-pair `('foo.xlsx', None)` does not hit it, so those ids are send to the cache as they are.
- To add another backend, modify the opening-sheets logic (ie clipboard), override *`_open_sheet()`*.
- It is a resource-manager for contained sheets, wo it can be used wth a `with` statement.

_derive_sheet_keys(*sheet*, *wb_ids=None*, *sh_ids=None*)
Retuns the product of user-specified and sheet-internal keys.

Parameters

- **wb_ids** – a single or a sequence of extra workbook-ids (ie: file, url)
- **sh_ids** – a single or sequence of extra sheet-ids (ie: name, index, None)

_open_sheet(*wb_id*, *sheet_id*, *opts*)
OVERRIDE THIS to change backend.

add_sheet(*sheet*, *wb_ids=None*, *sh_ids=None*)
Updates cache.

Parameters

- **wb_ids** – a single or sequence of extra workbook-ids (ie: file, url)
- **sh_ids** – a single or sequence of extra sheet-ids (ie: name, index, None)

close()
Closes all contained sheets and empties cache.

fetch_sheet(*wb_id*, *sheet_id*, *opts={}*, *base_sheet=None*)

Parameters **base_sheet** (`ABCSheet`) – The sheet used when unspecified `wb_id`.

pandalone.xleash.io._sheets.**margin_coords_from_states_matrix**(*states_matrix*)
Returns top-left/bottom-down margins of full cells from a *state* matrix.

May be used by *`ABCSheet.get_margin_coords()`* if a backend does not report the sheet-margins internally.

Parameters **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use *`ABCSheet.get_states_matrix()`* to derrive it.

Returns the 2 coords of the top-left & bottom-right full cells

Return type (Coords, Coords)

Examples::

```
>>> from io._sheets import margin_coords_from_states_matrix
```

```
>>> states_matrix = np.asarray([
...     [0, 0, 0],
...     [0, 1, 0],
...     [0, 1, 1],
...     [0, 0, 1],
... ])
>>> margins = margin_coords_from_states_matrix(states_matrix)
>>> margins
(Coords(row=1, col=1), Coords(row=3, col=2))
```

Note that the botom-left cell is not the same as `states_matrix` matrix size:

```
>>> states_matrix = np.asarray([
...     [0, 0, 0, 0],
...     [0, 1, 0, 0],
...     [0, 1, 1, 0],
...     [0, 0, 1, 0],
...     [0, 0, 0, 0],
... ])
>>> margin_coords_from_states_matrix(states_matrix) == margins
True
```

5.1.8 Submodule: pandalone.xleash.io._xlrd

Implements the *xlrd* backend of *xleash* that reads in-file Excel-spreadsheets.

class pandalone.xleash.io._xlrd.**XlrdSheet**(*sheet*, *book_fname*, *epoch1904=False*)

Bases: *pandalone.xleash.io._sheets.ABCSheet*

The *xlrd* workbook wrapper required by xleash library.

_close()
Override it to release resources for this sheet.

_close_all()
Override it to release resources this and all sibling sheets.

_read_states_matrix()
See super-method.

open_sibling_sheet(*sheet_id*, *opts=None*)
Gets by-index only if `sheet_id` is `int`, otherwise tries both by name and index.

read_rect(*st*, *nd*)
See super-method.

pandalone.xleash.io._xlrd.**_open_sheet_by_name_or_index**(*xlrd_book*, *wb_id*, *sheet_id*, *opts=None*)

Parameters

- **or str or None sheet_id** (*int*) – If `None`, opens 1st sheet.
- **opts** (*dict*) – does nothing with them

pandalone.xleash.io._xlrd.**_parse_cell**(*xcell*, *epoch1904=False*)
Parse a xl-xcell.

Parameters

- **xcell** (*xlrd.sheet.Cell*) – an excel xcell

- **epoch1904** (*bool*) – Which date system was in force when this file was last saved. False => 1900 system (the Excel for Windows default). True => 1904 system (the Excel for Macintosh default).

Returns formatted xcell value

Return type int, float, datetime.datetime, bool, None, str, datetime.time, float('nan')

Examples:

```
>>> import xlrd
>>> from xlrd.sheet import Cell
>>> _parse_cell(Cell(xlrd.XL_CELL_NUMBER, 1.2))
1.2

>>> _parse_cell(Cell(xlrd.XL_CELL_DATE, 1.2))
datetime.datetime(1900, 1, 1, 4, 48)

>>> _parse_cell(Cell(xlrd.XL_CELL_TEXT, 'hi'))
'hi'
```

pandalone.xleash.io._xlrd.**open_sheet**(*wb_url*, *sheet_id*, *opts*)
Opens the local or remote `wb_url` *xlrd* workbook wrapped as `XlrdSheet`.

5.1.9 Submodule: pandalone.xleash._capture

The algorithmic part of *capturing*.

Prefer accessing the public members from the parent module.

pandalone.xleash._capture.**CHECK_CELLTYPE = False**
When `True`, most coord-functions accept any 2-tuples.

exception pandalone.xleash._capture.**EmptyCaptureException**
Bases: `Exception`

Thrown when *targeting* fails.

pandalone.xleash._capture.**_col2num**(*coord*)
Resolves special coords or converts Excel A1 columns to a zero-based, reporting invalids.

Parameters **coord** (*str*) – excel-column coordinate or one of ^_.

Returns excel column number, >= 0

Return type int

Examples:

```
>>> col = _col2num('D')
>>> col
3
>>> _col2num('d') == col
True
>>> _col2num('AaZ')
727
>>> _col2num('10')
9
>>> _col2num(9)
8
```

Negatives (from left-end) are preserved:

```
>>> _col2num('AaZ')
727
```

Fails ugly:

```
>>> _col2num('%$')
Traceback (most recent call last):
ValueError: substring not found

>>> _col2num([])
Traceback (most recent call last):
TypeError: int() argument must be a string, a bytes-like object or
           a number, not 'list'
```

pandalone.xleash._capture.**_expand_rect**(*states_matrix*, *r1*, *r2*, *exp_moves*)

Applies the *expansion-moves* based on the `states_matrix`.

Parameters

- **state** –
- **r1** (`Coords`) – any vertice of the rect to expand
- **r2** (`Coords`) – any vertice of the rect to expand
- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use `ABCSheet.get_states_matrix()` to derrive it.
- **exp_moves** – Just the parsed string, and not `None`.

Returns a sorted rect top-left/bottom-right

Examples:

```
>>> states_matrix = np.array([
...     #0  1  2  3  4  5
...     [0, 0, 0, 0, 0, 0], #0
...     [0, 0, 1, 1, 1, 0], #1
...     [0, 1, 0, 0, 1, 0], #2
...     [0, 1, 1, 1, 1, 0], #3
...     [0, 0, 0, 0, 0, 1], #4
... ], dtype=bool)

>>> r1, r2 = (Coords(2, 1), Coords(2, 1))
>>> _expand_rect(states_matrix, r1, r2, 'U')
(Coords(row=2, col=1), Coords(row=2, col=1))

>>> r1, r2 = (Coords(3, 1), Coords(2, 1))
>>> _expand_rect(states_matrix, r1, r2, 'R')
(Coords(row=2, col=1), Coords(row=3, col=4))

>>> r1, r2 = (Coords(2, 1), Coords(6, 1))
>>> _expand_rect(states_matrix, r1, r2, 'r')
(Coords(row=2, col=1), Coords(row=6, col=5))

>>> r1, r2 = (Coords(2, 3), Coords(2, 3))
>>> _expand_rect(states_matrix, r1, r2, 'LURD')
(Coords(row=1, col=1), Coords(row=3, col=4))
```

pandalone.xleash._capture.**_extract_states_vector**(*states_matrix*, *dn_coords*, *land*, *mov*)

Extract a slice from the states-matrix by starting from `land` and following `mov`.

pandalone.xleash._capture.**_resolve_cell**(*cell*, *up_coords*, *dn_coords*, *base_coords=None*)

Translates any special coords to absolute ones.

To get the margin_coords, use one of:

- `ABCSheet.get_margin_coords()`
- *io._sheets.margin_coords_from_states_matrix()*

Parameters

- **cell** (`Cell`) – The "A1" cell to translate its coords.
- **up_coords** (`Coords`) – the top-left resolved coords with full-cells
- **dn_coords** (`Coords`) – the bottom-right resolved coords with full-cells
- **base_coords** (`Coords`) – A resolved cell to base dependent coords (`.`).

Returns the resolved cell-coords

Return type *Coords*

Examples:

```
>>> up = Coords(1, 2)
>>> dn = Coords(10, 6)
>>> base = Coords(40, 50)

>>> _resolve_cell(Cell(col='B', row='5'), up, dn)
Coords(row=4, col=1)

>>> _resolve_cell(Cell('^', '^'), up, dn)
Coords(row=1, col=2)

>>> _resolve_cell(Cell('_', '_'), up, dn)
Coords(row=10, col=6)

>>> base == _resolve_cell(Cell('.', '.'), up, dn, base)
True

>>> _resolve_cell(Cell('-1', '-2'), up, dn)
Coords(row=10, col=5)

>>> _resolve_cell(Cell('A', 'B'), up, dn)
Traceback (most recent call last):
ValueError: invalid cell(Cell(row='A', col='B')) due to:
        invalid row('A') due to: invalid literal for int() with base 10: 'A'
```

But notice when base-cell missing:

```
>>> _resolve_cell(Cell('1', '.'), up, dn)
Traceback (most recent call last):
ValueError: invalid cell(Cell(row='1', col='.')) due to:
Cannot resolve relative-col without base-coord!
```

pandalone.xleash._capture.**_resolve_coord**(*cname*, *cfunc*, *coord*, *up_coord*, *dn_coord*, *base_coords=None*)

Translates special coords or converts Excel string 1-based rows/cols to zero-based, reporting invalids.

Parameters

- **cname** (*str*) – the coord-name, one of 'row', 'column'
- **cfunc** (*function*) – the function to convert coord `str --> int`

- **str coord** (*int,*) – the "A1" coord to translate
- **up_coord** (*int*) – the resolved *top* or *left* margin zero-based coordinate
- **dn_coord** (*int*) – the resolved *bottom* or *right* margin zero-based coordinate
- **None base_coords** (*int,*) – the resolved basis for dependent coord, if any

Returns the resolved coord or None if it were not a special coord.

Row examples:

```
>>> cname = 'row'

>>> r0 = _resolve_coord(cname, _row2num, '1', 1, 10)
>>> r0
0
>>> r0 == _resolve_coord(cname, _row2num, 1, 1, 10)
True
>>> _resolve_coord(cname, _row2num, '^', 1, 10)
1
>>> _resolve_coord(cname, _row2num, '_', 1, 10)
10
>>> _resolve_coord(cname, _row2num, '.', 1, 10, 13)
13
>>> _resolve_coord(cname, _row2num, '-3', 0, 10)
8
```

But notice when base-cell missing:

```
>>> _resolve_coord(cname, _row2num, '.', 0, 10, base_coords=None)
Traceback (most recent call last):
ValueError: Cannot resolve relative-row without base-coord!
```

Other ROW error-checks:

```
>>> _resolve_coord(cname, _row2num, '0', 0, 10)
Traceback (most recent call last):
ValueError: invalid row('0') due to: Uncooked-coord cannot be zero!

>>> _resolve_coord(cname, _row2num, 'a', 0, 10)
Traceback (most recent call last):
ValueError: invalid row('a') due to: invalid literal for int() with base 10: 'a'

>>> _resolve_coord(cname, _row2num, None, 0, 10)
Traceback (most recent call last):
ValueError: invalid row(None) due to:
        int() argument must be a string,
        a bytes-like object or a number, not 'NoneType'
```

Column examples:

```
>>> cname = 'column'

>>> _resolve_coord(cname, _col2num, 'A', 1, 10)
0
>>> _resolve_coord(cname, _col2num, 'DADA', 1, 10)
71084
>>> _resolve_coord(cname, _col2num, '.', 1, 10, 13)
13
>>> _resolve_coord(cname, _col2num, '-4', 0, 10)
7
```

And COLUMN error-checks:

```
>>> _resolve_coord(cname, _col2num, None, 0, 10)
Traceback (most recent call last):
ValueError: invalid column(None) due to: int() argument must be a string,
            a bytes-like object or a number, not 'NoneType'

>>> _resolve_coord(cname, _col2num, 0, 0, 10)
Traceback (most recent call last):
ValueError: invalid column(0) due to: Uncooked-coord cannot be zero!
```

pandalone.xleash._capture.**_row2num**(*coord*)

Resolves special coords or converts Excel 1-based rows to zero-based, reporting invalids.

Parameters **int coord** (*str,*) – excel-row coordinate or one of ^_.

Returns excel row number, >= 0

Return type int

Examples:

```
>>> row = _row2num('1')
>>> row
0
>>> row == _row2num(1)
True
```

Negatives (from bottom) are preserved:

```
>>> _row2num('-1')
-1
```

Fails ugly:

```
>>> _row2num('.')
Traceback (most recent call last):
ValueError: invalid literal for int() with base 10: '.'
```

pandalone.xleash._capture.**_sort_rect**(*r1*, *r2*)

Sorts rect-vertices in a 2D-array (with vertices in rows).

Example:

```
>>> _sort_rect((5, 3), (4, 6))
array([[4, 3],
       [5, 6]])
```

pandalone.xleash._capture.**_target_opposite**(*states_matrix*, *dn_coords*, *land*, *moves*, *edge_name=''*)

Follow moves from `land` and stop on the 1st full-cell.

Parameters

- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use `ABCSheet.get_states_matrix()` to derrive it.
- **dn_coords** (`Coords`) – the bottom-right for the top-left of full-cells
- **land** (`Coords`) – the landing-cell
- **moves** (*str*) – MUST not be empty

Returns the identified target-cell's coordinates

Return type *Coords*

Examples:

```
>>> states_matrix = np.array([
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 1, 1, 1],
...     [0, 0, 1, 0, 0, 1],
...     [0, 0, 1, 1, 1, 1]
... ])
>>> args = (states_matrix, Coords(4, 5))

>>> _target_opposite(*(args + (Coords(0, 0), 'DR')))
Coords(row=3, col=2)

>>> _target_opposite(*(args + (Coords(0, 0), 'RD')))
Coords(row=2, col=3)
```

It fails if a non-empty target-cell cannot be found, or it ends-up beyond bounds:

```
>>> _target_opposite(*(args + (Coords(0, 0), 'D')))
Traceback (most recent call last):
pandalone.xleash._capture.EmptyCaptureException: No opposite-target found
                while moving(D) from landing-Coords(row=0, col=0)!

>>> _target_opposite(*(args + (Coords(0, 0), 'UR')))
Traceback (most recent call last):
pandalone.xleash._capture.EmptyCaptureException: No opposite-target found
                while moving(UR) from landing-Coords(row=0, col=0)!
```

But notice that the landing-cell maybe outside of bounds:

```
>>> _target_opposite(*(args + (Coords(3, 10), 'L')))
Coords(row=3, col=5)
```

pandalone.xleash._capture.**_target_same**(*states_matrix*, *dn_coords*, *land*, *moves*, *edge_name=''*)

Scan term:`exterior` row and column on specified `moves` and stop on the last full-cell.

Parameters

- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use `ABCSheet.get_states_matrix()` to derrive it.
- **dn_coords** (`Coords`) – the bottom-right for the top-left of full-cells
- **land** (`Coords`) – the landing-cell which MUST be within bounds
- **moves** – which MUST not be empty

Returns the identified target-cell's coordinates

Return type *Coords*

Examples:

```
>>> states_matrix = np.array([
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 1, 1, 1],
...     [0, 0, 1, 0, 0, 1],
...     [0, 0, 1, 1, 1, 1]
... ])
>>> args = (states_matrix, Coords(4, 5))

>>> _target_same(*(args + (Coords(4, 5), 'U')))
```

```
    Coords(row=2, col=5)

    >>> _target_same(*(args + (Coords(4, 5), 'L')))
    Coords(row=4, col=2)

    >>> _target_same(*(args + (Coords(4, 5), 'UL', )))
    Coords(row=2, col=2)
```

It fails if landing is empty or beyond bounds:

```
    >>> _target_same(*(args + (Coords(2, 2), 'DR')))
    Traceback (most recent call last):
    pandalone.xleash._capture.EmptyCaptureException: No same-target found
                    while moving(DR) from landing-Coords(row=2, col=2)!

    >>> _target_same(*(args + (Coords(10, 3), 'U')))
    Traceback (most recent call last):
    pandalone.xleash._capture.EmptyCaptureException: No same-target found
                    while moving(U) from landing-Coords(row=10, col=3)!
```

pandalone.xleash._capture.**_target_same_vector**(*states_matrix*, *dn_coords*, *land*, *mov*)

Parameters

- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use `ABCSheet.get_states_matrix()` to derrive it.
- **dn_coords** (`Coords`) – the bottom-right for the top-left of full-cells
- **land** (`Coords`) – The landing-cell, which MUST be full!

pandalone.xleash._capture.**coords2Cell**(*row*, *col*)

Make *A1* `Cell` from *resolved* coords, with rudimentary error-checking.

Examples:

```
    >>> coords2Cell(row=0, col=0)
    Cell(row='1', col='A')
    >>> coords2Cell(row=0, col=26)
    Cell(row='1', col='AA')

    >>> coords2Cell(row=10, col='.')
    Cell(row='11', col='.')

    >>> coords2Cell(row=-3, col=-2)
    Traceback (most recent call last):
    AssertionError: negative row!
```

pandalone.xleash._capture.**resolve_capture_rect**(*states_matrix*, *up_dn_margins*, *st_edge*, *nd_edge=None*, *exp_moves=None*, *base_coords=None*)

Performs *targeting*, *capturing* and *expansions* based on the *states-matrix*.

To get the margin_coords, use one of:

- `ABCSheet.get_margin_coords()`
- *io._sheets.margin_coords_from_states_matrix()*

Its results can be fed into `read_capture_values()`.

Parameters

- **states_matrix** (*np.ndarray*) – A 2D-array with `False` wherever cell are blank or empty. Use `ABCSheet.get_states_matrix()` to derrive it.
- **Coords) up_dn_margins** (*(Coords,*) – the top-left/bottom-right coords with full-cells
- **st_edge** (Edge) – "uncooked" as matched by regex
- **nd_edge** (Edge) – "uncooked" as matched by regex
- **or none exp_moves** (*list*) – Just the parsed string, and not `None`.
- **base_coords** (Coords) – The base for a *dependent 1st* edge.

Returns a `(Coords, Coords)` with the 1st and 2nd *capture-cell* ordered from top-left –> bottom-right.

Return type tuple

Raises **EmptyCaptureException** – When *targeting* failed, and no *target* cell identified.

Examples::

```
>>> from pandalone.xleash import Edge, margin_coords_from_states_matrix
```

```
>>> states_matrix = np.array([
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 0, 0, 0],
...     [0, 0, 0, 1, 1, 1],
...     [0, 0, 1, 0, 0, 1],
...     [0, 0, 1, 1, 1, 1]
... ], dtype=bool)
>>> up, dn = margin_coords_from_states_matrix(states_matrix)
```

```
>>> st_edge = Edge(Cell('1', 'A'), 'DR')
>>> nd_edge = Edge(Cell('.', '.'), 'DR')
>>> resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
(Coords(row=3, col=2), Coords(row=4, col=2))
```

Using dependenent coordinates for the 2nd edge:

```
>>> st_edge = Edge(Cell('_', '_'), None)
>>> nd_edge = Edge(Cell('.', '.'), 'UL')
>>> rect = resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
>>> rect
(Coords(row=2, col=2), Coords(row=4, col=5))
```

Using sheet's margins:

```
>>> st_edge = Edge(Cell('^', '_'), None)
>>> nd_edge = Edge(Cell('_', '^'), None)
>>> rect == resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
True
```

Walking backwards:

```
>>> st_edge = Edge(Cell('^', '_'), 'L')          # Landing is full, so 'L' ignored.
>>> nd_edge = Edge(Cell('_', '_'), 'L', '+')    # '+' or would also stop.
>>> rect == resolve_capture_rect(states_matrix, (up, dn), st_edge, nd_edge)
True
```

5.1.10 Submodule: pandalone.xleash._filter

The high-level functionality, the filtering and recursive *lassoing*.

Prefer accessing the public members from the parent module.

class pandalone.xleash._filter.**XLocation**(*sheet*, *st*, *nd*, *base_coords*)
Bases: tuple

Fields denoting the position of a sheet/cell while running a *element-wise-filter*.

Practically func:run_filter_elementwise() preserves these fields if the processed ones were ‘None.

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

__getstate__()
Exclude the OrderedDict from pickling

static __new__(*_cls*, *sheet*, *st*, *nd*, *base_coords*)
Create new instance of XLocation(sheet, st, nd, base_coords)

__repr__()
Return a nicely formatted representation string

_asdict()
Return a new OrderedDict which maps field names to their values.

classmethod _make(*iterable*, *new=<built-in method __new__ of type object at 0x971940>*, *len=<built-in function len>*)
Make a new XLocation object from a sequence or iterable

_replace(*_self*, ***kwds*)
Return a new XLocation object replacing specified fields with new values

base_coords
Alias for field number 3

nd
Alias for field number 2

sheet
Alias for field number 0

st
Alias for field number 1

pandalone.xleash._filter.**_classify_rect_shape**(*st*, *nd*)
Identifies rect from its edge-coordinates (row, col, 2d-table)..

Parameters

- **st** (Coords) – the top-left edge of capture-rect, inclusive
- **or None nd** (Coords) – the bottom-right edge of capture-rect, inclusive

Returns

in int based on the input like that:

- 0: only st given
- 1: st and nd point the same cell
- 2: row
- 3: col
- 4: 2d-table

Examples:

```
>>> _classify_rect_shape((1,1), None)
0
>>> _classify_rect_shape((2,2), (2,2))
1
>>> _classify_rect_shape((2,2), (2,20))
2
>>> _classify_rect_shape((2,2), (20,2))
3
>>> _classify_rect_shape((2,2), (20,20))
4
```

pandalone.xleash._filter.**_downdim**(*values*, *new_ndim*)

Squeeze it, and then flatten it, before inflating it.

Parameters

- **values** – The scalar ot 2D-results of `Sheet.read_rect()`
- **new_dim** (*int*) – The new dimension the result should have

pandalone.xleash._filter.**_redim**(*values*, *new_ndim*)

Reshapes the *capture-rect* values of `read_capture_rect()`.

Parameters

- **values** (*(nested) list, **) – The scalar ot 2D-results of `Sheet.read_rect()`
- **new_ndim** –

Returns reshaped values

Return type list of lists, list, *

Examples:

```
>>> _redim([1, 2], 2)
[[1, 2]]

>>> _redim([[1, 2]], 1)
[1, 2]

>>> _redim([], 2)
[[]]

>>> _redim([[3.14]], 0)
3.14

>>> _redim([[11, 22]], 0)
[11, 22]

>>> arr = [[[11], [22]]]
>>> arr == _redim(arr, None)
True

>>> _redim([[11, 22]], 0)
[11, 22]
```

pandalone.xleash._filter.**_updim**(*values*, *new_ndim*)

Append trivial dimensions to the left.

Parameters

- **values** – The scalar ot 2D-results of `Sheet.read_rect()`
- **new_dim** (*int*) – The new dimension the result should have

pandalone.xleash._filter.**get_default_filters**(*overrides=None*)
The default available *filters* used by *lasso()* when constructing its internal *Ranger*.

param dict or None overrides Any items to update the default ones.

return a dict-of-dicts with 2 items:

- *func*: a function with args: `(Ranger, Lasso, *args, **kwds)`
- *desc*: help-text replaced by `func.__doc__` if missing.

rtype dict

pandalone.xleash._filter.**pipe_filter**(*ranger*, *lasso*, **filters*, ***kwds*)
A *bulk-filter* that applies all call-specifiers one after another on the *capture-rect* values.

Parameters **filters** (*list*) – the json-parsed *call-spec*

pandalone.xleash._filter.**py_filter**(*ranger*, *lasso*, *expr*)
A *bulk-filter* that passes values through a python-expression using `asteval` library.

The `expr` may access read-write all `locals()` of this method (`ranger`, *lasso*), the `numpy` funcs, and the *pandalone.xleash* module under the `xleash` variable.

The `expr` may return either:

- the processed values, or
- an instance of the *Lasso*, in which case only its `opt` field is checked and replaced with original if missing. So better use `namedtuple._replace()` on the current *lasso* which exists in the expr's namespace.

Parameters **expr** (*str*) – The python-expression, which may comprise of multiple statements.

pandalone.xleash._filter.**pyeval_filter**(*ranger*, *lasso*, **filters*, ***kwds*)
A *element-wise-filter* that uses `asteval` to evaluate string values as python expressions.

The `expr` fecthed from `term:'capturing` may access read-write all `locals()` of this method (ie: `ranger`, *lasso*), the `numpy` funcs, and the *pandalone.xleash* module under the `xleash` variable.

The `expr` may return either:

- the processed values, or
- an instance of the *Lasso*, in which case only its `opt` field is checked and replaced with original if missing. So better use `namedtuple._replace()` on the current *lasso* which exists in the expr's namespace.

Parameters

- **eval_all** (*bool*) – If `True` raise on 1st error and stop diving cells. Defaults to `False`.
- **filters** (*list*) – Any *filters* to apply after invoking the `element_func`.
- **or str include** (*list*) – Items to include when diving into "indexed" values. See `run_filter_elementwise()`.
- **or str exclude** (*list*) – Items to exclude when diving into "indexed" values. See `run_filter_elementwise()`.
- **or None depth** (*int*) – How deep to dive into nested structures, "indexed" or lists. If `< 0`, no limit. If 0, stops completely. See `run_filter_elementwise()`.

Note that in python-3 the signature woudl be:

```
def pyeval_filter(ranger, lasso, element_func, filters,
                  include=None, exclude=None, depth=-1):
```

Example:

```
>>> expr = '''
... res = array([[0.5, 0.3, 0.1, 0.1]])
... res * res.T
... '''
>>> lasso = Lasso(values=expr, opts={})
>>> ranger = Ranger(None)
>>> eval_filter(ranger, lasso).values
array([[ 0.25,  0.15,  0.05,  0.05],
       [ 0.15,  0.09,  0.03,  0.03],
       [ 0.05,  0.03,  0.01,  0.01],
       [ 0.05,  0.03,  0.01,  0.01]])
```

pandalone.xleash._filter.**recursive_filter**(*ranger*, *lasso*, **filters*, ***kwds*)

A *element-wise-filter* that expand recursively any *xl-ref* strings elements in *capture-rect* values.

Parameters

- **filters** (*list*) – Any *filters* to apply after invoking the `element_func`.
- **or str include** (*list*) – Items to include when diving into "indexed" values. See `run_filter_elementwise()`.
- **or str exclude** (*list*) – Items to exclude when diving into "indexed" values. See `run_filter_elementwise()`.
- **or None depth** (*int*) – How deep to dive into nested structures, "indexed" or lists. If `< 0`, no limit. If 0, stops completely. See `run_filter_elementwise()`.

Note that in python-3 the signature would be:

```
def recursive_filter(ranger, lasso, element_func, filters,
                     include=None, exclude=None, depth=-1):
```

pandalone.xleash._filter.**redim_filter**(*ranger*, *lasso*, *scalar=None*, *cell=None*, *row=None*, *col=None*, *table=None*)

A *bulk-filter* that reshapes sand/or transpose captured values, depending on rect's shape.

Each dimension might be a single int or None, or a pair [dim, transpose].

pandalone.xleash._filter.**run_filter_elementwise**(*ranger*, *lasso*, *element_func*, *filters*, *include=None*, *exclude=None*, *depth=-1*, **args*, ***kwds*)

Runner of all *element-wise filters*.

It applies the `element_func` on elements extracted from `lasso.values` by treating the later first as "indexed" objects (Mappings, Series and Dataframes.), and if that fails, as nested lists.

- The `include`/`exclude` filter args work only for "indexed" objects with `items()` or `iteritems()` and indexing methods.
  - If no filter arg specified, expands for all keys.
  - If only `include` specified, rejects all keys not explicitly contained in this filter arg.

- If only `exclude` specified, expands all keys not explicitly contained in this filter arg.
- When both `include`/`exclude` exist, only those explicitly included are accepted, unless also excluded.

- Lower the `logging` level to see other than syntax-errors on recursion reported on `log`.
- Only those in *`XLocation`* are passed recursively.

Parameters

- **`element_func`** (*`list`*) – A function implementing the element-wise *filter* and returning a 2-tuple `(is_proccessed, new_val_or_lasso)`, like that:

```
def element_func(ranger, lasso, context, elval)
    proced = False
    try:
        elval = int(elval)
        proced = True
    except ValueError:
        pass
    return proced, elval
```

  Its `kwds` may contain the `include`, `exclude` and `depth` args. Any exception raised from `element_func` will cancel the diving.
- **`filters`** (*`list`*) – Any *filters* to apply after invoking the `element_func`.
- **`or str include`** (*`list`*) – Items to include when diving into "indexed" values. See description above.
- **`or str exclude`** (*`list`*) – Items to exclude when diving into "indexed" values. See description above.
- **`or None depth`** (*`int`*) – How deep to dive into nested structures, "indexed" or lists. If `< 0`, no limit. If 0, stops completely.

Params args To be relayed to 'element_func'.

Params kwds To be relayed to 'element_func'.

`pandalone.xleash._filter.`**`xlwings_dims_call_spec`**`()`
A list *call-spec* for `_redim_filter()` *filter* that imitates results of *xlwings* library.

5.1.11 Submodule: `pandalone.xleash._lasso`

The high-level functionality, the filtering and recursive *lassoing*.

Prefer accessing the public members from the parent module.

class `pandalone.xleash._lasso.`**`Ranger`**(*sheets_factory*, *base_opts=None*, *available_filters=None*)

Bases: `object`

The director-class that performs all stages required for "throwing the lasso" around rect-values.

Use it when you need to have total control of the procedure and configuration parameters, since no defaults are assumed.

The *`do_lasso()`* does the job.

Variables

- **sheets_factory** (SheetsFactory) – Factory of sheets from where to parse rect-values; does not close it in the end. Maybe None, but *do_lasso()* will scream unless invoked with a context_lasso arg containing a concrete *ABCSheet*.
- **base_opts** (*dict*) – The *opts* that are deep-copied and used as the defaults for every *do_lasso()*, whether invoked directly or recursively by recursive_filter(). If unspecified, no opts are used, but this attr is set to an empty dict. See *get_default_opts()*.
- **or None available_filters** (*dict*) – No filters exist if unspecified. See *get_default_filters()*.
- **intermediate_lasso** (Lasso) – A ('stage', Lasso) pair with the last *Lasso* instance produced during the last execution of the *do_lasso()*. Used for inspecting/debuging.

_make_init_Lasso(**context_kwds*)
Creates the lasso to be used for each new *do_lasso()* invocation.

_parse_and_merge_with_context(*xlref*, *init_lasso*)
Merges xl-ref parsed-parsed_fields with init_lasso, reporting any errors.

Parameters **init_lasso** (Lasso) – Default values to be overridden by non-nulls. Note that init_lasso.opts must be a ChainMap, as returned by $_{m}ake_{i}nit_{L}asso()$.

Returns a Lasso with any non None parsed-fields updated

_relasso(*lasso*, *stage*, ***kwds*)
Replace lasso-values and updated intermediate_lasso.

_resolve_capture_rect(*lasso*, *sheet*)
Also handles *EmptyCaptureException* in case opts['no_empty'] != False.

do_lasso(*xlref*, ***context_kwds*)
The director-method that does all the job of hrowing a *lasso* around spreadsheet's rect-regions according to *xl-ref*.

Parameters

- **xlref** (*str*) – a string with the *xl-ref* format:

```
<url_file>#<sheet>!<1st_edge>:<2nd_edge>:<expand><js_filt>
```

i.e.:

```
file:///path/to/file.xls#sheet_name!UPT8(LU-):_.(D+):LDL1{"dims":1}
```

- **context_kwds** (Lasso) – Default *Lasso* fields in case parsed ones are None

Returns The final *Lasso* with captured & filtered values.

Return type *Lasso*

make_call(*lasso*, *func_name*, *args*, *kwds*)
Executes a *call-spec* respecting any lax argument popped from kwds.

Parameters **lax** (*bool*) – After overlaying it on *opts*, it governs whether to raise on errors. Defaults to False (scream!).

pandalone.xleash._lasso.**get_default_opts**(*overrides=None*)
Default *opts* used by *lasso()* when constructing its internal *Ranger*.

Parameters **or None overrides** (*dict*) – Any items to update the default ones.

pandalone.xleash._lasso.**lasso**(*xlref*, *sheets_factory=None*, *base_opts=None*, *available_filters=None*, *return_lasso=False*, ***context_kwds*)

High-level function to *lasso* around spreadsheet's rect-regions according to *xl-ref* strings by using internally a *Ranger* .

Parameters

- **xlref** (*str*) – a string with the *xl-ref* format:

```
<url_file>#<sheet>!<1st_edge>:<2nd_edge>:<expand><js_filt>
```

  i.e.:

```
file:///path/to/file.xls#sheet_name!UPT8(LU-):_.(D+):LDL1{"dims":1}
```

- **sheets_factory** – Factory of sheets from where to parse rect-values; if unspecified, the new *SheetsFactory* created is closed afterwards. Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.
- **or None available_filters** (*dict*) – Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.
- **return_lasso** (*bool*) – If `True`, values are contained in the returned Lasso instance, along with all other artifacts of the *lassoing* procedure.

  For more debugging help, create a `Range` yourself and inspect the `Ranger.intermediate_lasso`.
- **context_kwds** (`Lasso`) – Default *Lasso* fields in case parsed ones are `None` (i.e. you can specify the sheet like that).

Variables or None base_opts (*dict*) – Opts affecting the lassoing procedure that are deep-copied and used as the base-opts for every *Ranger.do_lasso()*, whether invoked directly or recursively by `recursive_filter()`. Read the code to be sure what are the available choices. Delegated to *make_default_Ranger()*, so items override default ones; use a new *Ranger* if that is not desired.

Returns Either the captured & filtered values or the final *Lasso*, depending on the `return_lassos` arg.

Example:

```
sheet = _
```

pandalone.xleash._lasso.**make_default_Ranger**(*sheets_factory=None*, *base_opts=None*, *available_filters=None*)

Makes a defaulted *Ranger*.

Parameters

- **sheets_factory** – Factory of sheets from where to parse rect-values; if unspecified, a new *SheetsFactory* is created. Remember to invoke its *SheetsFactory.close()* to clear resources from any opened sheets.
- **or None base_opts** (*dict*) – Default opts to affect the lassoing, to be merged with defaults; uses *get_default_opts()*.

  Read the code to be sure what are the available choices :-(.

- **or None available_filters** (*dict*) – The available *filters* to specify a *xl-ref*. Uses *get_default_filters()* if unspecified.

For instance, to make you own sheets-factory and override options, yoummay do this:

```
>>> from pandalone import xleash

>>> with xleash.SheetsFactory() as sf:
...     xleash.make_default_Ranger(sf, base_opts={'lax': True})
<pandalone.xleash._lasso.Ranger object at
...
```

5.2 Module: pandalone.mappings

Hierarchical string-like objects useful for indexing, that can be rename/relocated at a later stage.

Pstep	Automagically-constructed *relocatable* paths for accessing data-
pmods_from_tuples(pmods_tuples)	Turns a list of 2-tuples into a *pmods* hierarchy.
Pmod([_alias, _steps, _regxs])	A path-step mapping forming the pmods-hierarchy.

Example:

```
>>> from pandalone.mappings import pmods_from_tuples

>>> pmods = pmods_from_tuples([
...     ('',          'deeper/ROOT'),
...     ('/abc',      'ABC'),
...     ('/abc/foo', 'BAR'),
... ])
>>> p = pmods.step()
>>> p.abc.foo
BAR
>>> p._paths()
['deeper/ROOT/ABC/BAR']
```

- TODO: Implements "anywhere" pmods(//).

class pandalone.mappings.**Pmod**(*_alias=None*, *_steps={}*, *_regxs={}*)

Bases: object

A path-step mapping forming the pmods-hierarchy.

- The *pmods* denotes the hierarchy of all *mappings*, that either *rename* or *relocate* path-steps.
- A single *mapping* transforms an "origin" path to a "destination" one (also called as "from" and "to" paths).
- A mapping always transforms the *final* path-step, like that:

```
FROM_PATH       TO_PATH       RESULT_PATH
---------       -------       -----------
/rename/path    foo       --> /rename/foo        ## renaming
/relocate/path  foo/bar   --> /relocate/foo/bar  ## relocation
''              a/b/c     --> /a/b/c             ## Relocate all paths.
/               a/b/c     --> /a/b/c             ## Relocates 1st "empty-str" step.
```

- The *pmod* is the mapping of that single path-step.
- It is possible to match fully on path-steps using regular-expressions, and then to use any captured-groups from the *final* step into the mapped value:

```
    (/all(.*)/path, foo)   + all_1/path --> /all_1/foo
                           + all_XYZ    --> /all_XYZ        ## no change
    (/all(.*)/path, foo\1) + all_1/path --> /all_1/foo_1
```

If more than one regex match, they are merged in the order declared (the latest one overrides a previous one).

- Any exact child-name matches are applied and merged after regexs.
- Use *pmods_from_tuples()* to construct the pmods-hierarchy.
- The pmods are used internally by class:*Pstep* to correspond the component-paths of their input & output onto the actual value-tree paths.

Variables

- **_alias** (*str*) – (optional) the mapped-name of the pstep for this pmod
- **_steps** (*dict*) – {original_name –> pmod}
- **_regxs** (*OrderedDict*) – {regex_on_originals –> pmod}

Example:

Note: Do not manually construct instances from this class! To construct a hierarchy use the *pmods_from_tuples()*.

You can either use it for massively map paths, either for *renaming* them:

```
>>> pmods = pmods_from_tuples([
...         ('/a',           'A'),
...         ('/~b.*',        r'BB\g<0>'),  ## Previous match.
...         ('/~b.*/~c.(.*)', r'W\1ER'),    ## Capturing-group(1)
... ])
>>> pmods.map_paths(['/a', '/a/foo'])     ## 1st rule
['/A', '/A/foo']

>>> pmods.map_path('/big/stuff')          ## 2nd rule
'/BBbig/stuff'

>>> pmods.map_path('/born/child')         ## 2nd & 3rd rule
'/BBborn/WildER'
```

or to *relocate* them:

```
>>> pmods = pmods_from_tuples([
...         ('/a',           'A/AA'),
...         ('/~b.*/~c(.*)',  r'../C/\1'),
...         ('/~b.*/~.*/~r.*', r'/\g<0>'),
... ])
>>> pmods.map_paths(['/a/foo', '/big/child', '/begin/from/root'])
['/A/AA/foo', '/big/C/hild', '/root']
```

Here is how you relocate “root” (notice that the ’’ path is the root):

```
>>> pmods = pmods_from_tuples([('', '/NEW/ROOT')])
>>> pmods.map_paths(['/a/foo', ''])
['/NEW/ROOT/a/foo', '/NEW/ROOT']
```

__init__(*_alias=None*, *_steps={}*, *_regxs={}*)
Args passed only for testing, remember _regxs to be (k,v) tuple-list!

Note: Volatile arg-defaults (empty dicts) are knowingly used , to preserve memory; should never append in them!

_append_into_regxs(*key*)
Inserts a child-mappings into `_steps` dict.

Parameters **key** (*str*) – the regex-pattern to add

_append_into_steps(*key*)
Inserts a child-mappings into `_steps` dict.

Parameters **key** (*str*) – the step-name to add

_match_regxs(*cstep*)
Return (pmod, regex.match) for those child-pmods matching `cstep`.

_merge(*other*)
Clone and override all its props with props from other-pmod, recursively.

Although it does not modify this, the `other` or their children pmods, it may "share" (crosslink) them, so pmods MUST NOT be modified later.

Parameters **other** (Pmod) – contains the dicts with the overrides

Returns the cloned merged pmod

Return type *Pmod*

Examples:

Look how `_steps` are merged:

```
>>> pm1 = Pmod(_alias='pm1', _steps={
...     'a':Pmod(_alias='A'), 'c':Pmod(_alias='C')})
>>> pm2 = Pmod(_alias='pm2', _steps={
...     'b':Pmod(_alias='B'), 'a':Pmod(_alias='AA')})
>>> pm = pm1._merge(pm2)
>>> sorted(pm._steps.keys())
['a', 'b', 'c']
```

And here it is `_regxs` merging, which preserves order:

```
>>> pm1 = Pmod(_alias='pm1',
...            _regxs=[('d', Pmod(_alias='D')),
...                    ('a', Pmod(_alias='A')),
...                    ('c', Pmod(_alias='C'))])
>>> pm2 = Pmod(_alias='pm2',
...            _regxs=[('b', Pmod(_alias='BB')),
...                    ('a', Pmod(_alias='AA'))])

>>> pm1._merge(pm2)
pmod('pm2', OrderedDict([(re.compile('d'), pmod('D')),
           (re.compile('c'), pmod('C')),
           (re.compile('b'), pmod('BB')),
           (re.compile('a'), pmod('AA'))]))

>>> pm2._merge(pm1)
pmod('pm1', OrderedDict([(re.compile('b'), pmod('BB')),
            (re.compile('d'), pmod('D')),
            (re.compile('a'), pmod('A')),
            (re.compile('c'), pmod('C'))]))
```

_override_regxs(*other*)
Override this pmod's `_regxs` dict with other's, recursively.

- It may "share" (crosslink) the dict and/or its child-pmods between the two pmod args (`self` and `other`).
- No dict is modified (apart from self, which must have been cloned previously by *Pmod._merge()*), to avoid side-effects in case they were "shared".
- It preserves dict-ordering so that `other` order takes precedence (its elements are the last ones).

Parameters

- **self** (Pmod) – contains the dict that would be overridden
- **other** (Pmod) – contains the dict with the overrides

_override_steps(*other*)

Override this pmod's '_steps' dict with other's, recursively.

Same as *_override_regxs()* but without caring for order.

alias(*cstep*)

Like *descend()* but without merging child-pmods.

Returns the expanded alias from child/regexs or None

descend(*cstep*)

Return child-pmod with merged any exact child with all matched regexps, along with its alias regex-expaned.

Parameters **cstep** (*str*) – the child path-step cstep of the pmod to return

Returns the merged-child pmod, along with the alias; both might be None, if nothing matched, or no alias.

Return type tuple(Pmod, str)

Example:

```
>>> pm = Pmod(
...     _steps={'a': Pmod(_alias='A')},
...     _regxs=[('a\w*', Pmod(_alias='AWord')),
...             ('a(\d*)', Pmod(_alias=r'A_\1')),
...     ])
>>> pm.descend('a')
(pmod('A'), 'A')

>>> pm.descend('abc')
(pmod('AWord'), 'AWord')

>>> pm.descend('a12')
(pmod('A_\\1'), 'A_12')

>>> pm.descend('BAD')
(None, None)
```

Notice how children of regexps are merged together:

```
>>> pm = Pmod(
...     _steps={'a':
...        Pmod(_alias='A', _steps={1: 11})},
...     _regxs=[
...        (r'a\w*', Pmod(_alias='AWord',
...                       _steps={2: Pmod(_alias=22)})),
...        (r'a\d*', Pmod(_alias='ADigit',
...                       _steps={3: Pmod(_alias=33)})),
...     ])
>>> sorted(pm.descend('a')[0]._steps)    ## All children and regexps match.
```

```
        [1, 2, 3]

        >>> pm.descend('aa')[0]._steps           ## Only 'a\w*' matches.
        {2: pmod(22)}

        >>> sorted(pm.descend('a1')[0]._steps )  ## Both regexps matches.
        [2, 3]
```

So it is possible to say:

```
        >>> pm.descend('a1')[0].alias(2)
        22
        >>> pm.descend('a1')[0].alias(3)
        33
        >>> pm.descend('a1')[0].descend('BAD')
        (None, None)
        >>> pm.descend('a$')
        (None, None)
```

but it is better to use *map_path()* for this.

map_path(*path*)

Maps a '/rooted/path' using all aliases while descending its child pmods.

It uses any aliases on all child pmods if found.

Parameters **path** (*str*) – a rooted path to transform

Returns the rooted mapped path or '/' if path was '/'

Return type str or None

Examples:

```
        >>> pmods = pmods_from_tuples([
        ...         ('/a',              'A/AA'),
        ...         ('/~a(\\w*)',       r'BB\1'),
        ...         ('/~a\\w*/~d.*',     r'D \g<0>'),
        ...         ('/~a(\\d+)',       r'C/\1'),
        ...         ('/~a(\\d+)/~(c.*)', r'CC-/\1'), # The 1st group is ignored!
        ...         ('/~a\\d+/~e.*',     r'/newroot/\g<0>'), # Rooted mapping.
        ... ])

        >>> pmods.map_path('/a')
        '/A/AA'

        >>> pmods.map_path('/a_hi')
        '/BB_hi'

        >>> pmods.map_path('/a12')
        '/C/12'

        >>> pmods.map_path('/a12/etc')
        '/newroot/etc'
```

Notice how children from *all* matching prior-steps are merged:

```
        >>> pmods.map_path('/a12/dow')
        '/C/12/D dow'
        >>> pmods.map_path('/a12/cow')
        '/C/12/CC-/cow'
```

To map *root* use '' which matches before the 1st slash('/'):

```
        >>> pmods = pmods_from_tuples([('', 'New/Root'),])  ## Relative
        >>> pmods
```

```
        pmod({'': pmod('New/Root')})

        >>> pmods.map_path('/for/plant')
        'New/Root/for/plant'

        >>> pmods_from_tuples([('', '/New/Root'),]).map_path('/for/plant')
        '/New/Root/for/plant'
```

Note: Using slash('/') for "from" path will NOT map *root*:

```
        >>> pmods = pmods_from_tuples([('/', 'New/Root'),])
        >>> pmods
        pmod({'': pmod({'': pmod('New/Root')})})

        >>> pmods.map_path('/for/plant')
        '/for/plant'

        >>> pmods.map_path('//for/plant')
        '/New/Root/for/plant'

        '/root'
```

---

but '' always remains unchanged (whole document):

```
        >>> pmods.map_path('')
        ''
```

step(*pname=''*, *alias=None*)
Create a new *Pstep* having as mappings this pmod.

If no pname specified, creates a *root* pstep.

Delegates to *Pstep.__new__()*.

class pandalone.mappings.**Pstep**
Bases: str

Automagically-constructed *relocatable* paths for accessing data-tree.

The "magic" autocreates psteps as they referenced, making writing code that access data-tree paths, natural, while at the same time the "model" of those tree-data gets discovered.

Each pstep keeps internally the *name* of a data-tree step, which, when created through recursive referencing, concedes with parent's branch leading to this step. That name can be modified with *Pmod* so the same data-accessing code can refer to differently-named values int the data-tree.

Variables

- **_csteps** (*dict*) – the child-psteps by their name (default None)
- **_pmod** (*dict*) – path-modifications used to construct this and relayed to children (default None)
- **_locked** (*int*) – one of - Pstep.CAN_RELOCATE (default), - Pstep.CAN_RENAME, - Pstep.LOCKED (neither from the above).
- **_tags** (*set*) – A set of strings (default ())
- **_schema** (*dict*) – json-schema data.

See *__new__()* for interal constructor.

Usage:

- Use a `Pmod.pstep()` to construct a *root* pstep from mappings. Specify a string argument to construct a relative pstep-hierarchy.
- Just referencing (non_private) attributes, creates them.
- Private attributes and functions (starting with _) exist for specific operations (ie for specifying json-schema, or for collection all paths).
- Assignments are only allowed for string-values, or to private attributes:

```
>>> p = Pstep()
>>> p.assignments = 12
Traceback (most recent call last):
AssertionError: Cannot assign '12' to '/assignments!

>>> p._but_hidden = 'Ok'
```

- Use *_paths()* to get all defined paths so far.
- Construction:

```
>>> Pstep()

>>> Pstep('a')
a
```

Notice that pstesps are surrounded with the back-tick char(‘‘‘).

- Paths are created implicitely as they are referenced:

```
>>> m = {'a': 1, 'abc': 2, 'cc': 33}
>>> p = Pstep('a')
>>> assert m[p] == 1
>>> assert m[p.abc] == 2
>>> assert m[p.a321.cc] == 33

>>> sorted(p._paths())
['a/a321/cc', 'a/abc']
```

- Any “path-mappings” or “pmods” maybe specified during construction:

```
>>> from pandalone.mappings import pmods_from_tuples

>>> pmods = pmods_from_tuples([
...     ('',         'deeper/ROOT'),
...     ('/abc',     'ABC'),
...     ('/abc/foo', 'BAR'),
... ])
>>> p = pmods.step()
>>> p.abc.foo
BAR
>>> p._paths()
['deeper/ROOT/ABC/BAR']
```

- but exceptions are thrown if mapping any step marked as “locked”:

```
>>> p.abc.foo._locked  ## 3: CAN_RELOCATE
3
```

```
>>> p.abc.foo._lock    ## Screams, because foo is already mapped.
Traceback (most recent call last):
ValueError: Cannot rename/relocate 'foo'-->'BAR' due to LOCKED!
```

Warning: Creating an empty(`''`) step in some paths will "root" the path:

```
>>> p = Pstep()
>>> _ = p.a1.b
>>> _ = p.A2
>>> p._paths()
['/A2', '/a1/b']

>>> _ = p.a1.a2.c
>>> _ = p.a1.a2 = ''
>>> p._paths()
['/A2', '/a1/b', '/c']
```

static __new__(*pname=''*, *_proto_or_pmod=None*, *alias=None*)

Constructs a string with str-content which may comes from the mappings.

These are the valid argument combinations:

```
pname='attr_name',
pname='attr_name', _alias='Mass [kg]'

pname='attr_name', _proto_or_pmod=Pmod

pname='attr_name', _proto_or_pmod=Pstep
pname='attr_name', _proto_or_pmod=Pstep, _alias='Mass [kg]'
```

Parameters

- **pname** (*str*) – this pstep's name which must coincede with the name of the parent-pstep's attribute holding this pstep. It is stored at `_orig` and if no `alias` and unmapped by pmod, this becomes the `alias`.
- **or Pstep _proto_or_pmod** (Pmod) – It can be either:
  - the mappings for this pstep,
  - another pstep to clone attributes from (used when replacing an existing child-pstep), or
  - None.

  The mappings will apply only if *Pmod.descend()* match `pname` and will derrive the alias.
- **alias** (*str*) – Will become the super-str object when no mappaings specified (`_proto_or_pmod` is a dict from some prototype pstep) It gets jsonpointer-escaped if it exists (see `pandata.escape_jsonpointer_part()`)

_derrive_map_tuples()

Recursively extract `(cmap --> alias)` pairs from the pstep-hierarchy.

Parameters

- **pairs** (*list*) – Where to append subtree-paths built.
- **prefix_steps** (*tuple*) – branch currently visiting

Return type [(str, str)]

_fix

Sets `locked` = `CAN_RENAME`. :return: self :raise: ValueError if step has been relocated pstep

_iter_hierarchy(*prefix_steps=()*)
Breadth-first traversing of pstep-hierarchy.

Parameters **prefix_steps** (*tuple*) – Builds here branch currently visiting.

Returns yields the visited pstep along with its path (including it)

Return type (Pstep, [Pstep])

_lock
Set `locked = LOCKED`. :return: self, for chained use :raise: ValueError if step has been renamed/relocated pstep

_locked
Gets *_locked* internal flag or scream on set, when step already renamed/relocated

Prefer using one of *_fix* or *_lock* instead.

Parameters **locked** – One of `CAN_RELOCATE`, `CAN_RENAME`, `LOCKED`.

Raise ValueError when stricter lock-value on a renamed/relocated pstep

_paths(*with_orig=False*, *tag=None*)
Return all children-paths (str-list) constructed so far, in a list.

Parameters

- **with_orig** (*bool*) – wheter to include also orig-path, for debug.
- **tag** (*str*) – If not 'None', fetches all paths with `tag` in their last step.

Return type [str]

Examples:

```
>>> p = Pstep()
>>> _ = p.a1._tag('inp').b._tag('inp').c
>>> _ = p.a2.b2

>>> p._paths()
['/a1/b/c', '/a2/b2']

>>> p._paths(tag='inp')
['/a1', '/a1/b']
```

For debugging set `with_orig` to `True`:

```
>>> pmods = pmods_from_tuples([
...     ('',          'ROOT'),
...     ('/a',     'A/AA'),
... ])
>>> p = pmods.step()
>>> _ = p.a.b
>>> p._paths(with_orig=True)
 ['(-->ROOT)/(a-->A/AA)/b']
```

_schema
Updates json-schema-v4 on this pstep (see `JSchema`).

_schema_exists()
Always use this to avoid needless schema-instantiations.

_tag(*tag*)
Add a "tag" for this pstep.

Returns self, for chained use

_tag_remove(*tag*)
Delete a "tag" from this pstep.

Returns self, for chained use

pandalone.mappings.**_append_step**(*steps*, *step*)
Joins `step` at the right of `steps`, respecting '/', '..', '.', ''.

Parameters

- **steps** (*tuple*) – where to append into ("absolute" when 1st-element is '')
- **step** (*str*) – what to append (may be: `'foo'`, `'.'`, `'..'`, `''`)

Return type tuple

Note: The empty-string('') is the "root" for both `steps` and `step`. An empty-tuple `steps` is considered "relative", equivalent to dot().

Example:

```
>>> _append_step((), 'a')
('a',)

>>> _append_step(('a', 'b'), '..')
('a',)

>>> _append_step(('a', 'b'), '.')
('a', 'b')
```

Not that an "absolute" path has the 1st-step empty(`''`), (so the previous paths above were all "relative"):

```
>>> _append_step(('a', 'b'), '')
('',)

>>> _append_step(('',), '')
('',)

>>> _append_step((), '')
('',)
```

Dot-dots preserve "relative" and "absolute" paths, respectively, and hence do not coalesce when at the left:

```
>>> _append_step(('',), '..')
('',)

>>> _append_step(('',), '.')
('',)

>>> _append_step(('a',), '..')
()

>>> _append_step((), '..')
('..',)

>>> _append_step(('..',), '..')
('..', '..')

>>> _append_step((), '.')
()
```

Single-dots('.') just dissappear:

```
>>> _append_step(('.',), '.')
()

>>> _append_step(('.',), '..')
('..',)
```

pandalone.mappings.**_clone_attrs**(*obj*)
Clone deeply any collection attributes of the passed-in object.

pandalone.mappings.**_join_paths**(**steps*)
Joins all path-steps in a single string, respecting `'/'`, `'..'`, `'.'`, `''`.

Parameters **steps** (*str*) – single json-steps, from left to right

Return type str

Note: If you use `iter_jsonpointer_parts_relaxed()` to generate path-steps, the "root" is signified by the empty(`''`) step; not the slash(`/`)!

Hence a lone slash(`/`) gets splitted to an empty step after "root" like that: `('', '')`, which generates just "root"(`''`).

Therefore a "folder" (i.e. `some/folder/`) when splitted equals `('some', 'folder', '')`, which results again in the "root"(`''`)!

Examples:

```
>>> _join_paths('r', 'a', 'b')
'r/a/b'

>>> _join_paths('', 'a', 'b', '..', 'bb', 'cc')
'/a/bb/cc'

>>> _join_paths('a', 'b', '.', 'c')
'a/b/c'
```

An empty-step "roots" the remaining path-steps:

```
>>> _join_paths('a', 'b', '', 'r', 'aa', 'bb')
'/r/aa/bb'
```

All `steps` have to be already "splitted":

```
>>> _join_paths('a', 'b', '../bb')
'a/b/../bb'
```

Dot-doting preserves "relative" and "absolute" paths, respectively:

```
>>> _join_paths('..')
'..'

>>> _join_paths('a', '..')
'.'

>>> _join_paths('a', '..', '..', '..')
'../..'

>>> _join_paths('', 'a', '..', '..')
''
```

Some more special cases:

```
    >>> _join_paths('..', 'a')
    '../a'

    >>> _join_paths('', '.', '..', '..')
    ''

    >>> _join_paths('.', '..')
    '..'

    >>> _join_paths('..', '.', '..')
    '../..'
```

See also:

_append_step

pandalone.mappings.**pmods_from_tuples**(*pmods_tuples*)

Turns a list of 2-tuples into a *pmods* hierarchy.

- Each tuple defines the renaming-or-relocation of the *final* part of some component path onto another one into value-trees, such as:

```
(/rename/path, foo)          --> rename/foo
(relocate/path, foo/bar)    --> relocate/foo/bar
```

- The "from" path may be: - relative, - absolute(starting with `/`), or - "anywhere"(starting with `//`).
- In case a "step" in the "from" path starts with tilda(), it is assumed to be a regular-expression, and it is removed from it. The "to" path can make use of any "from" capture-groups:

```
('/~all(.*)/path', 'foo')
('~some[\d+]/path', 'foo{')
('//~all(.*)/path', 'foo')
```

Parameters **str) pmods_tuples** (*list(tuple(str,)*) –

Returns a root pmod

Return type *Pmod*

Example:

```
    >>> pmods_from_tuples([
    ...     ('/a', 'A1/A2'),
    ...     ('/a/b', 'B'),
    ... ])
    pmod({'': pmod({'a': pmod('A1/A2', {'b': pmod('B')})})})

    >>> pmods_from_tuples([
    ...     ('/~a*', 'A1/A2'),
    ...     ('/a/~b[123]', 'B'),
    ... ])
    pmod({'': pmod({'a':
            pmod(OrderedDict([(re.compile('b[123]'), pmod('B'))]))},
                 OrderedDict([(re.compile('a*'), pmod('A1/A2'))]))})
```

This is how you map *root*:

```
    >>> pmods = pmods_from_tuples([
    ...     ('', 'relative/Root'),        ## Make all paths relatives.
    ...     ('/a/b', '/Rooted/B'),        ## But map b would be "rooted".
    ... ])
```

```
>>> pmods
pmod({'':
        pmod('relative/Root',
                {'a': pmod({'b':
                        pmod('/Rooted/B')})})})

>>> pmods.map_path('/a/c')
'relative/Root/a/c'

>>> pmods.map_path('/a/b')
'/Rooted/B'
```

But note that '/' maps the 1st "empty-str" step after root:

```
>>> pmods_from_tuples([
...     ('/', 'New/Root'),
... ])
pmod({'': pmod({'': pmod('New/Root')})})
```

TODO: Implement "anywhere" matches.

pandalone.mappings.**pstep_from_df**(*columns_df*, *name_col='names'*)

Creates a *Pstep* instances from a dataframe.

Parameters **columns_df** (*pd.DataFrame*) – pstep's mapped-names in `name_col` column, indexed by paths, and any additional pstep-attributes in the rest columns.

example:

```
========  =========  ==================
paths     names      renames
========  =========  ==================
/A        foo        ['FOO', 'LL']
/B        bar        []
========  =========  ==================
```

5.3 Module: pandalone.components

Defines the building-blocks of a "model":

components and assemblies: See *Component*, *FuncComponent* and *Assembly*.

paths and path-mappings (pmods): See `Pmod`, `pmods_from_tuples()` and `Pstep`.

5.3.1 TODO

1. Assembly use ComponentLoader collecting components with:
   - `gatattr()` and
   - `filter_predicate` default to `attr.__name__.startswith('cfunc_')`.
   - enforce a `disable` flag on them.
2. Component/assembly should have a stackable or common cwd?
3. Components should be easy to run without "framework". - `_build()` –> `run()` - pmods on init OR `run()`? - As ContextManager?
4. Imply a default Assembly.

class pandalone.components.**Assembly**(*components*, *name=None*)
Bases: *pandalone.components.Component*

Example:

```
>>> def cfunc_f1(comp, value_tree):
...     comp.pinp().A
...     comp.pout().B
>>> def cfunc_f2(comp, value_tree):
...     comp.pinp().B
...     comp.pout().C
>>> ass = Assembly(FuncComponent(cfunc) for cfunc in [cfunc_f1, cfunc_f2])
>>> ass._build()
>>> assert list(ass._iter_validations()) == []
>>> ass._inp
['f1/A', 'f2/B']
>>> ass._out
['f1/B', 'f2/C']
```

```
>>> from pandalone.mappings import pmods_from_tuples
```

```
>>> pmod = pmods_from_tuples([
...     ('~.*',  '/root'),
... ])
>>> ass._build(pmod)
>>> sorted(ass._inp + ass._out)
['/root/A', '/root/B', '/root/B', '/root/C']
```

class pandalone.components.**Component**(*name*)
Bases: object

Encapsulates a function and its its inputs/outputs dependencies.

It should be callable, and when executed it may read/modify the data-tree given as its 1st input.

An opportunity to fix the internal-state (i.e. inputs/output/name) is when the *_build()* is invoked.

Variables

- **_name** (*list*) – identifier
- **_inp** (*list*) – list/of/paths required on the data-tree (must not overlap with out)
- **_out** (*list*) – list/of/paths modified on the data-tree (must not overlap with inp)

Mostly defined through *cfuncs*, which provide for defining a component with a single function with a special signature, see *FuncComponent*.

__metaclass__
alias of ABCMeta

_build(*pmod=None*)
Invoked once before run-time and should apply pmaps when given.

_iter_validations()
Yields a msg for each failed validation rule.

Invoke it after *_build()* component.

class pandalone.components.**FuncComponent**(*cfunc*, *name=None*)
Bases: *pandalone.components.Component*

Converts a “cfunc” into a component.

A cfunc is a function that modifies the values-tree with this signature:

```
cfunc_XXXX(comp, vtree)
```

where:

comp: the *FuncComponent* associated with the cfunc

vtree: the part of the data-tree involving the values to be modified by the cfunc

It works also as a utility to developers of a cfuncs, since it is passed as their 1st arg.

The cfuncs may use *pinp()* and *pout()* when accessing its input and output data-tree values respectively. Note that accessing any of those attributes from outside of cfunc, would result in an error.

If a cfunc access additional values with "fixed' paths, then it has to manually add those paths into the `_inp` and `_out` lists.

Example:

This would be a fully "relocatable" cfunc:

```
>>> def cfunc_calc_foobar_rate(comp, value_tree):
...     pi = comp.pinp()
...     po = comp.pout()
...
...     df = value_tree.get(pi)
...
...     df[po.Acc] = df[pi.V] / df[pi.T]
```

To get the unmodified component-paths, use:

```
>>> comp = FuncComponent(cfunc_calc_foobar_rate)
>>> comp._build()
>>> assert list(comp._iter_validations()) == []
>>> sorted(comp._inp + comp._out)
['calc_foobar_rate/Acc', 'calc_foobar_rate/T', 'calc_foobar_rate/V']
```

To get the path-modified component-paths, use:

```
>>> from pandalone.mappings import pmods_from_tuples

>>> pmods = pmods_from_tuples([
...     ('~.*', '/A/B'),
... ])
>>> comp._build(pmods)

>>> sorted(comp.pinp()._paths())
['/A/B/T', '/A/B/V']

>>> comp.pout()._paths()
['/A/B/Acc']

>>> sorted(comp._inp + comp._out)
['/A/B/Acc', '/A/B/T', '/A/B/V']

>>> comp._build(pmods)
>>> sorted(comp._inp + comp._out)
['/A/B/Acc', '/A/B/T', '/A/B/V']
```

_build(*pmod=None*)
Extracts inputs/outputs from cfunc.

pinp(*path=None*)
The suggested `Pstep` for cfunc to use to access inputs.

pout(*path=None*)
The suggested `Pstep` for cfunc to use to access outputs.

5.4 Module: pandalone.pandata

A *pandas-model* is a tree of strings, numbers, sequences, dicts, pandas instances and resolvable URI-references, implemented by *Pandel*.

class pandalone.pandata.**JSONCodec**
Bases: object

Json coders/decoders capable for (almost) all python objects, by pickling them.

Example:

```
>>> import json
>>> obj_list = [
...    3.14,
...    {
...         'aa': pd.DataFrame([]),
...         2: np.array([]),
...         33: {'foo': 'bar'},
...     },
...     pd.DataFrame(np.random.randn(10, 2)),
...     ('b', pd.Series({})),
... ]
>>> for o in obj_list + [obj_list]:
...     s = json.dumps(o, cls=JSONCodec.Encoder)
...     oo = json.loads(s, cls=JSONCodec.Decoder)
...     assert trees_equal(o, oo)
...
```

See also:

For pickle-limitations: https://docs.python.org/3.4/library/pickle.html#pickle-picklable

class pandalone.pandata.**JSchema**
Bases: object

Facilitates the construction of json-schema-v4 nodes on `PStep` code.

It does just rudimentary args-name check. Further validations should apply using a proper json-schema validator.

Parameters

- **type** – if omitted, derived as 'object' if it has children
- **kws** – for all the rest see http://json-schema.org/latest/json-schema-validation.html

class pandalone.pandata.**ModelOperations**
Bases: *pandalone.pandata.ModelOperations*

Customization functions for traversing, I/O, and converting self-or-descendant branch (sub)model values.

static __new__(*inp=None*, *out=None*, *conv=None*)

Parameters

- **inp** (*list*) – the `args-list` to *Pandel._read_branch()*
- **out** – The args to *Pandel._write_branch()*, that may be specified either as:

  - an `args-list`, that will apply for all model data-types (lists, dicts & pandas),
  - a map of `type -> args-list`, where the `None` key is the *catch-all* case,
  - a function returning the `args-list` for some branch-value, with signature: `def get_write_branch_args(branch)`.

- **conv** – The conversion-functions (*convertors*) for the various model's data-types. The convertors have signature `def convert(branch)`, and they may be specified either as:
  - a map of `(from_type, to_type) -> conversion_func()`, where the `None` key is the *catch-all* case,
  - a "master-switch" function returning the appropriate convertor depending on the requested conversion. The master-function's signature is `def get_convertor(from_branch, to_branch)`.

  The minimum convertors demanded by *Pandel* are (at least, check the code for more):
  - DataFrame <-> dict
  - Series <-> dict
  - ndarray <-> list

class pandalone.pandata.**Pandel**(*curate_funcs=()*)

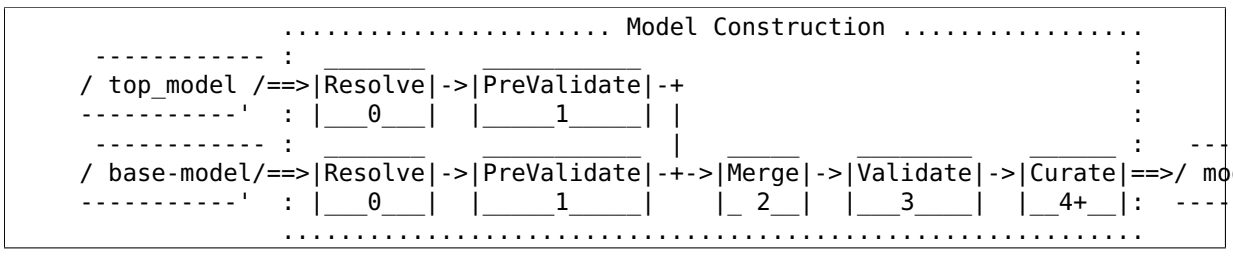
Bases: object

Builds, validates and stores a *pandas-model*, a mergeable stack of JSON-schema abiding trees of strings and numbers, assembled with

- sequences,
- dictionaries,
- pandas.DataFrame,
- pandas.Series, and
- URI-references to other model-trees.

Overview

The **making of a model** involves, among others, schema-validating, reading *subtree-branches* from URIs, cloning, converting and merging multiple *sub-models* in a single *unified-model* tree, without side-effecting given input. All these happen in 4+1 steps:



All steps are executed "lazily" using generators (with yield). Before proceeding to the next step, the previous one must have completed successfully. That way, any ad-hoc code in building-step-5(*curation*), for instance, will not suffer a horrible death due to badly-formed data.

[TODO] The **storing of a model** simply involves distributing model parts into different files and/or formats, again without side-effecting the unified-model. **Building model**

Here is a detailed description of each building-step:

1. *_resolve()* and substitute any json-references present in the submodels with content-fragments fetched from the referred URIs. The submodels are **cloned** first, to avoid side-effecting them.

   Although by default a combination of *JSON* and *CSV* files is expected, this can be customized, either by the content in the json-ref, within the model (see below), or as *explained* below.

   The **extended json-refs syntax** supported provides for passing arguments into *_read_branch()* and *_write_branch()* methods. The syntax is easier to explain by showing what the default *_global_cntxt* corresponds to, for a `DataFrame`:

```
{
  "$ref": "http://example.com/example.json#/foo/bar",
  "$inp": ["AUTO"],
  "$out": ["CSV", "encoding=UTF-8"]
}
```

   And here what is required to read and (later) store into a HDF5 local file with a predefined name:

```
{
  "$ref": "file://./filename.hdf5",
  "$inp": ["AUTO"],
  "$out": ["HDF5"]
}
```

   Warning: Step NOT IMPLEMENTED YET!

2. Loosely *_prevalidate()* each sub-model separately with *json-schema*, where any pandas-instances (DataFrames and Series) are left as is. It is the duty of the developer to ensure that the prevalidation-schema is *loose enough* that it allows for various submodel-forms, prior to merging, to pass.

3. Recursively **clone** and *_merge()* sub-models in a single unified-model tree. Branches from sub-models higher in the stack override the respective ones from the sub-models below, recursively. Different object types need to be **converted** appropriately (ie. merging a `dict` with a `DataFrame` results into a `DataFrame`, so the dictionary has to convert to dataframe).

   The required **conversions** into pandas classes can be customized as *explained* below. Series and DataFrames cannot merge together, and Sequences do not merge with any other object-type (themselfs included), they just "overwrite".

   The default convertor-functions defined both for submodels and models are listed in the following table:

From:	To:	Method:
dict	DataFrame	`pd.DataFrame` (the constructor)
DataFrame	dict	`lambda df: df.to_dict('list')`
dict	Series	`pd.Series` (the constructor)
Series	dict	`lambda sr: sr.to_dict()`

4. Strictly json-*_validate()* the unified-model (ie enforcing `required` schema-rules).

   The required **conversions** from pandas classes can be customized as *explained* below.

   The default convertor-functions are the same as above.

5. (Optionally) Apply the *_curate()* functions on the the model to enforce dependencies and/or any ad-hoc generation-rules among the data. You can think of bash-like expansion patterns, like `${/some/path:=$HOME}` or expressions like `%len(../other/path)`.

Storing model

When storing model-parts, if unspecified, the filenames to write into will be deduced from the jsonpointer-path of the `$out`'s parent, by substituting "strange" chars with undescores(_).

> **Warning:** Functionality NOT IMPLEMENTED YET!

Customization

Some operations within steps (namely *conversion* and *IO*) can be customized by the following means (from lower to higher precedance):

1. The global-default *ModelOperations* instance on the *_global_cntxt*, applied on both submodels and unified-model.

   For example to channel the whole reading/writing of models through HDF5 data-format, it would suffice to modify the *_global_cntxt* like that:

```
pm = FooPandelModel()                          ## some concrete model-maker
io_args = ["HDF5"]
pm.mod_global_operations(inp=io_args, out=io_args)
```

2. [TODO] Extra-properties on the json-schema applied on both submodels and unified-model for the specific path defined. The supported properties are the non-functional properties of *ModelOperations*.

4. Specific-properties regarding *IO* operations within each submodel - see the *resolve* building-step, above.

3. Context-maps of `json_paths -> ModelOperations` instances, installed by *add_submodel()* and *unified_contexts* on the model-maker. They apply to self-or-descedant subtree of each model.

   The `json_path` is a strings obeying a simplified *json-pointer* syntax (no char-normalizations yet), ie `/some/foo/1/pointer`. An empty-string(`''`) matches all model.

   When multiple convertors match for a model-value, the selected convertor to be used is the most specific one (the one with longest prefix). For instance, on the model:

```
[ { "foo": { "bar": 0 } } ]
```

   all of the following would match the `0` value:

   - the global-default *_global_cntxt*,
   - `/`, and
   - `/0/foo`

   but only the last's context-props will be applied.

Atributes

`model`
The model-tree that will receive the merged submodels after *build()* has been invoked. Depending on the submodels, the top-value can be any of the supported model data-types.

_submodel_tuples
The stack of (`submodel`, `path_ops`) tuples. The list's 1st element is the *base-model*, the last one, the *top-model*. Use the *add_submodel()* to build this list.

_global_cntxt
A *ModelOperations* instance acting as the global-default context for the unified-model and all submodels. Use *mod_global_operations()* to modify it.

_curate_funcs
The sequence of *curate* functions to be executed as the final step by *_curate()*. They are "normal" functions (not generators) with signature:

```
def curate_func(model_maker):
    pass      ## ie: modify model_maker.model.
```

Better specify this list of functions on construction time.

_errored
An internal boolean flag that becomes `True` if any build-step has failed, to halt proceeding to the next one. It is `None` if build has not started yet.

Examples

The basic usage requires to subclass your own model-maker, just so that a *json-schema* is provided for both validation-steps, 2 & 4:

```
>>> from collections import OrderedDict as od                    ## Json is better wi
```

```
>>> class MyModel(Pandel):
...     def _get_json_schema(self, is_prevalidation):
...         return {                                              ## Define the json-s
...             '$schema': 'http://json-schema.org/draft-04/schema#',
...             'required': [] if is_prevalidation else ['a', 'b'],     ## Prevalidation is
...             'properties': {
...                 'a': {'type': 'string'},
...                 'b': {'type': 'number'},
...                 'c': {'type': 'number'},
...             }
...         }
```

Then you can instanciate it and add your submodels:

```
>>> mm = MyModel()
>>> mm.add_submodel(od(a='foo', b=1))                            ## submodel-1 (base)
>>> mm.add_submodel(pd.Series(od(a='bar', c=2)))                 ## submodel-2 (top-m
```

You then have to build the final unified-model (any validation errors would be reported at this point):

```
>>> mdl = mm.build()
```

Note that you can also access the unified-model in the *model* attribute. You can now interogate it:

```
>>> mdl['a'] == 'bar'                          ## Value overridden by top-model
True
>>> mdl['b'] == 1                              ## Value left intact from base-model
True
>>> mdl['c'] == 2                              ## New value from top-model
True
```

Lets try to build with invalid submodels:

```
>>> mm = MyModel()
>>> mm.add_submodel({'a': 1})              ## According to the schema, this should have been
>>> mm.add_submodel({'b': 'string'})       ## and this one, a number.
```

```
>>> sorted(mm.build_iter(), key=lambda ex: ex.message)    ## Fetch a list with all validation
[<ValidationError: "'string' is not of type 'number'">,
 <ValidationError: "1 is not of type 'string'">,
 <ValidationError: 'Gave-up building model after step 1.prevalidate (out of 4).'>]
```

```
>>> mdl = mm.model
>>> mdl is None                                    ## No model constructed, failed before me
True
```

And lets try to build with valid submodels but invalid merged-one:

```
>>> mm = MyModel()
>>> mm.add_submodel({'a': 'a str'})
>>> mm.add_submodel({'c': 1})
```

```
>>> sorted(mm.build_iter(), key=lambda ex: ex.message)
[<ValidationError: "'b' is a required property">,
 <ValidationError: 'Gave-up building model after step 3.validate (out of 4).'>]
```

__init__(*curate_funcs=()*)

Parameters **curate_funcs** (*sequence*) – See *_curate_funcs*.

__metaclass__
alias of ABCMeta

_clone_and_merge_submodels(*a*, *b*, *path=''*)
‘ Recursively merge b into a, cloning both.

_curate()
Step-4: Invokes any curate-functions found in *_curate_funcs*.

_get_json_schema(*is_prevalidation*)

Returns a json schema, more loose when `prevalidation` for each case

Return type dictionary

_merge()
Step-2

_prevalidate()
Step-1

_read_branch()
Reads model-branches during *resolve* step.

_resolve()
Step-1

_select_context(*path*, *branch*)
Finds which context to use while visiting model-nodes, by enforcing the precedance-rules described in the *Customizations*.

Parameters

- **path** (*str*) – the branch’s jsonpointer-path
- **branch** (*str*) – the actual branch’s node

Returns the selected *ModelOperations*

_validate()
Step-3

_write_branch()
Writes model-branches during *distribute* step.

add_submodel(*model*, *path_ops=None*)
Pushes on top a submodel, along with its context-map.

Parameters

- **model** – the model-tree (sequence, mapping, pandas-types)
- **path_ops** (*dict*) – A map of `json_paths` –> *ModelOperations* instances acting on the unified-model. The `path_ops` may often be empty.

Examples

To change the default DataFrame –> dictionary convertor for a submodel, use the following:

```
>>> mdl = {'foo': 'bar'}
>>> submdl = ModelOperations(mdl, conv={(pd.DataFrame, dict): lambda df: df.to_dict('rec
```

build()
Attempts to build the model by exhausting *build_iter()*, or raises its 1st error.

Use this method when you do not want to waste time getting the full list of errors.

build_iter()
Iteratively build model, yielding any problems as `ValidationError` instances.

For debugging, the unified model at *model* my contain intermediate results at any time, even if construction has failed. Check the *_errored* flag if neccessary.

mod_global_operations(*operations=None*, ***cntxt_kwargs*)
Since it is the fall-back operation for *conversions* and *IO* operation, it must exist and have all its props well-defined for the class to work correctly.

Parameters

- **operations** (ModelOperations) – Replaces values of the installed context with non-empty values from this one.
- **cntxt_kwargs** – Replaces the keyworded-values on the existing `operations`. See *ModelOperations* for supported keywords.

unified_contexts
A map of `json_paths` –> *ModelOperations* instances acting on the unified-model.

pandalone.pandata.**_U**
alias of `United`

pandalone.pandata.**iter_jsonpointer_parts**(*jsonpath*)
Generates the `jsonpath` parts according to jsonpointer spec.

Parameters **jsonpath** (*str*) – a jsonpath to resolve within document

Returns The parts of the path as generator), without converting any step to int, and None if None.

Author Julian Berman, ankostis

Examples:

```
>>> list(iter_jsonpointer_parts('/a/b'))
['a', 'b']

>>> list(iter_jsonpointer_parts('/a//b'))
['a', '', 'b']
```

```
>>> list(iter_jsonpointer_parts('/'))
['']

>>> list(iter_jsonpointer_parts(''))
[]
```

But paths are strings begining (NOT_MPL: but not ending) with slash('/'):

```
>>> list(iter_jsonpointer_parts(None))
Traceback (most recent call last):
AttributeError: 'NoneType' object has no attribute 'split'

>>> list(iter_jsonpointer_parts('a'))
Traceback (most recent call last):
jsonschema.exceptions.RefResolutionError: Jsonpointer-path(a) must start with '/'!

#>>> list(iter_jsonpointer_parts('/a/'))
#Traceback (most recent call last):
#jsonschema.exceptions.RefResolutionError: Jsonpointer-path(a) must NOT ends with '/'!
```

pandalone.pandata.**iter_jsonpointer_parts_relaxed**(*jsonpointer*)

Like *iter_jsonpointer_parts()* but accepting also non-absolute paths.

The 1st step of absolute-paths is always ''.

Examples:

```
>>> list(iter_jsonpointer_parts_relaxed('a'))
['a']
>>> list(iter_jsonpointer_parts_relaxed('a/'))
['a', '']
>>> list(iter_jsonpointer_parts_relaxed('a/b'))
['a', 'b']

>>> list(iter_jsonpointer_parts_relaxed('/a'))
['', 'a']
>>> list(iter_jsonpointer_parts_relaxed('/a/'))
['', 'a', '']

>>> list(iter_jsonpointer_parts_relaxed('/'))
['', '']

>>> list(iter_jsonpointer_parts_relaxed(''))
['']
```

pandalone.pandata.**parse_value_with_units**(*arg*)

Parses *name-units* pairs (i.e. used as a table-column header).

Returns a United(name, units) named-tuple, or None if bad syntax; note that `name=''` but `units=None` when missing.

Examples:

```
>>> parse_value_with_units('value [units]')
United(name='value', units='units')

>>> parse_value_with_units('foo   bar  <bar/krow>')
United(name='foo   bar', units='bar/krow')

>>> parse_value_with_units('no units')
United(name='no units', units=None)

>>> parse_value_with_units('')
United(name='', units=None)
```

But notice:

```
>>> assert parse_value_with_units('ok but [bad units') is None

>>> parse_value_with_units('<only units>')
United(name='', units='only units')

>>> parse_value_with_units(None)
Traceback (most recent call last):
TypeError: expected string or buffer
```

pandalone.pandata.**resolve_jsonpointer**(*doc*, *jsonpointer*, *default=<object object>*)

Resolve a `jsonpointer` within the referenced `doc`.

Parameters

- **doc** – the referrant document
- **jsonpointer** (*str*) – a jsonpointer to resolve within document

Returns the resolved doc-item or raises `RefResolutionError`

Raises RefResolutionError (if cannot resolve jsonpointer path)

Examples:

```
>>> dt = {
...     'pi':3.14,
...     'foo':'bar',
...     'df': pd.DataFrame(np.ones((3,2)), columns=list('VN')),
...     'sub': {
...         'sr': pd.Series({'abc':'def'}),
...     }
... }
>>> resolve_jsonpointer(dt, '/pi', default=_scream)
3.14
```

```
>>> resolve_jsonpointer(dt, '/pi/BAD')
Traceback (most recent call last):
jsonschema.exceptions.RefResolutionError: Unresolvable JSON pointer('/pi/BAD')@(BAD)
```

```
>>> resolve_jsonpointer(dt, '/pi/BAD', 'Hi!')
'Hi!'
```

Author Julian Berman, ankostis

pandalone.pandata.**set_jsonpointer**(*doc*, *jsonpointer*, *value*, *object_factory=<class 'collections.OrderedDict'>*)

Resolve a `jsonpointer` within the referenced `doc`.

Parameters

- **doc** – the referrant document
- **jsonpointer** (*str*) – a jsonpointer to the node to modify

Raises RefResolutionError (if jsonpointer empty, missing, invalid-contet)

CHAPTER 6

Changes

Contents

6.1 Known deficiencies

6.1.1 TODOs

- **XLeash**
  - **Core:**
    - **Syntax:**
      - [ ] Notation for specifying the “last-sheet”.
      - **[ ] Extend RC-coords: ^-1, _[-6], .-4**
        - [ ] Cell becomes 4-tuple.
      - [ ] Expand meander `@?`
    - **filters:**
      - [ ] Slices and Index args on ‘numpy’ and ‘df’ filters.
    - [ ] Xlrd-read with slices.
  - **Struct:**
    - [ ] Plugins for backends (& syntax?)
  - **TCs**
    - [ ] More TCs.
  - **Backends:**
    - [ ] xlwings
    - [ ] Clipboard
    - [ ] openpyxl
    - [ ] google-drive sheets

  - **[ ] Split own project**
    - [ ] README
  - [ ] Check TODOs in code

6.1.2 Rejected TODOs:

- **xleash**
  - Support cubic areas; pandas create dict-of-dfs from multiple sheets.
  - Use *ast* library for filters; cannot extract safely opts.
  - Build Lasso-structs trees on `recursive` filter for debugging; carefully crafted exception-messages is enough.

6.2 Changelog

- **v0.1.9 (Dec-2015):**
  - pstep: Add `pstep_from_df()` utility.
- **v0.1.8 (Sep-2015):**
  - deps: Do not require flake8.
- **v0.1.7 (Sep-2015):**
  - deps: Do not enforce pandas/numpy version.
- **v0.1.6 (Sep-2015):**
  - xleash: Minor forgotten regression from previous fix context-sheet.
  - pstep: Make steps work as pandas-indexes.
- v0.1.5 (Sep-2015): properly fix context-sheet on `Ranger` anf `SheetsFactory`.
- **v0.1.4 (Sep-2015): xleash fixes**
  - xleash: Temporarily-Hacked for not closing sibling-sheets.
  - xleash: handle gracefully targeting-failures as *empty* captures.
- **v0.1.3 (Sep-2015):**
  - xleash: perl-quoting xlrefs to avoid being treated as python-comments.
- **v0.1.1 (Sep-2015): 1st working release**
  - **xleash:**
    - FIX missing `xleash` package from wheel.
    - Renamed package `xlasso->` `xleash` and factor-out `_filters` module.
    - Added `py-eval` filter.
    - Accept xl-refs quoted by any char.
- **v0.1.0 (Sep-2015): XLasso BROKEN!**
  - Release in *pypi* broken, missing xlasso.
  - The `mappings` and `xlasso` packages are considered ready to be used.
- v0.0.11 (XX-May-2015)

- v0.0.1.dev1 (01-March-2015)

CHAPTER 7

Indices

7.1 Glossary

data-tree The *container* of data consumed and produced by a :term‘model‘, which may contain also the model. Its values are accessed using *path* s. It is implemented by *pandalone.pandata.Pandel* as a mergeable stack of *JSON-schema* abiding trees of strings and numbers, formed with:

- sequences,
- dictionaries,
- pandas instances, and
- URI-references.

value-tree That part of the *data-tree* that relates only to the I/O data processed.

model A collection of *component* s and accompanying *mappings*.

component Encapsulates a data-transformation function, using *path* to refer to its inputs/outputs within the *value-tree*.

path A /file/like string functioning as the *id* of data-values in the *data-tree*. It is composed of *step*, and it follows the syntax of the *JSON-pointer*.

step, pstep, path-step The parts between between two conjecutive slashes(/) within a *path*. The Pstep facilitates their manipulation.

pmod, pmods, pmods-hierarchy, mapping, mappings Specifies a transformation of an “origin” path to a “destination” one (also called as “from” and “to” paths). The mapping always transforms the *final* path-step, and it can either *rename* or *relocate* that step, like that:

```
    ORIGIN          DESTINATION   RESULT_PATH
    ------          -----------   -----------
    /rename/path    foo       --> /rename/foo        ## renaming
    /relocate/path  foo/bar   --> /relocate/foo/bar  ## relocation
    /root           a/b/c     --> /a/b/c             ## Relocates all /root sub-paths.
```

The hierarchy is formed by Pmod instances, which are build when parsing the *mappings* list, above.

JSON-schema The JSON schema is an IETF draft that provides a *contract* for what JSON-data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data. You can learn more about it from this excellent guide, and experiment with this on-line validator.

JSON-pointer JSON Pointer(**RFC 6901**) defines a string syntax for identifying a specific value within a JavaScript Object Notation (JSON) document. It aims to serve the same purpose as *XPath* from the XML world, but it is much simpler.

7.1.1 Index

CHAPTER 8

Glossary

data-tree The *container* of data consumed and produced by a :term‘model‘, which may contain also the model. Its values are accessed using *path* s. It is implemented by *`pandalone.pandata.Pandel`* as a mergeable stack of *JSON-schema* abiding trees of strings and numbers, formed with:

- sequences,
- dictionaries,
- `pandas` instances, and
- URI-references.

value-tree That part of the *data-tree* that relates only to the I/O data processed.

model A collection of *component* s and accompanying *mappings*.

component Encapsulates a data-transformation function, using *path* to refer to its inputs/outputs within the *value-tree*.

path A `/file/like` string functioning as the *id* of data-values in the *data-tree*. It is composed of *step*, and it follows the syntax of the *JSON-pointer*.

step, pstep, path-step The parts between between two conjecutive slashes(`/`) within a *path*. The `Pstep` facilitates their manipulation.

pmod, pmods, pmods-hierarchy, mapping, mappings Specifies a transformation of an “origin” path to a “destination” one (also called as “from” and “to” paths). The mapping always transforms the *final* path-step, and it can either *rename* or *relocate* that step, like that:

```
    ORIGIN          DESTINATION   RESULT_PATH
    ------          -----------   -----------
    /rename/path    foo       --> /rename/foo        ## renaming
    /relocate/path  foo/bar   --> /relocate/foo/bar  ## relocation
    /root           a/b/c     --> /a/b/c             ## Relocates all /root sub-paths.
```

The hierarchy is formed by `Pmod` instances, which are build when parsing the *mappings* list, above.

JSON-schema The JSON schema is an IETF draft that provides a *contract* for what JSON-data is required for a given application and how to interact with it. JSON Schema is intended to define validation, documentation, hyperlink navigation, and interaction control of JSON data. You can learn more about it from this excellent guide, and experiment with this on-line validator.

JSON-pointer JSON Pointer(**RFC 6901**) defines a string syntax for identifying a specific value within a JavaScript Object Notation (JSON) document. It aims to serve the same purpose as *XPath* from the XML world, but it is much simpler.

p

Symbols

A

B

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

T

U

V

X