pwkit

Release 0.6.99

September 09, 2015

Contents

This documentation has a lot of stubs.

CHAPTER 1

About the Software

`pwkit` is a collection of Peter Williams' miscellaneous Python tools. I'm packaging them so that other people can install them off of PyPI or Conda and run my code without having to go to too much work. That's the hope, at least.

1.1 Installation

The most recent stable version of `pwkit` is available on the Python package index, so you should be able to install this package simply by running `pip install pwkit`. The package is also available in the conda package manager by installing it from binstar.org; the command `conda install -c pkgw pwkit` should suffice.

If you want to download the source code and install `pwkit` manually, the package uses the standard Python setuptools, so running `python setup.py install` will do the trick.

1.2 Authors

`pwkit` is authored by Peter K. G. Williams and collaborators. Despite this package being named after me, contributions are welcome and will be given full credit. I just don't want to have to make up a decent name for this package right now.

Contributions have come from (alphabetically by surname):

- Maïca Clavel
- Elisabeth Newton
- Denis Ryzhkov (I copied method_decorator)

1.3 Copyright and License

CHAPTER 2

Foundations

This documentation has a lot of stubs.

2.1 Core utilities (`pwkit`)

The toplevel *`pwkit`* module includes a few basic abstractions that show up throughout the rest of the codebase. These include:

- *The Holder namespace object*
- *Utilities for exceptions*
- *Abstractions between Python versions 2 and 3*

2.1.1 The `Holder` namespace object

`Holder` is a "namespace object" that primarily exists so that you can fill it with named attributes however you want. This is convenient for, say, implementing functions that return complex data in a way that's amenable to future extension.

class `pwkit.`**`Holder`**`(`*__decorating=None*, ***kwargs*`)`

Create a new *`Holder`*. Any keyword arguments will be assigned as properties on the object itself, for instance, `o = Holder (foo=1)` yields an object such that `o.foo` is 1.

The *__decorating* keyword is used to implement the *`Holder`* decorator functionality, described below.

While the *`Holder`* is primarily meant for bare-bones namespace management, it does provide several convenience functions: *`Holder.get()`*, *`Holder.set()`*, *`Holder.set_one()`*, *`Holder.has()`*, *`Holder.copy()`*, *`Holder.to_dict()`*, and *`Holder.to_pretty()`*.

`Holder.`**`__unicode__`**`()`

Placeholder.

`Holder.`**`__str__`**`()`

Placeholder.

`Holder.`**`__repr__`**`()`

Placeholder.

`Holder.`**`__iter__`**`()`

Placeholder.

Holder.**__contains__**(*key*)
Placeholder.

Holder.**get**(*name*, *defval=None*)
Placeholder.

Holder.**set**(***kwargs*)
Placeholder.

Holder.**set_one**(*name*, *value*)
Placeholder.

Holder.**has**(*name*)
Placeholder.

Holder.**copy**()
Placeholder.

Holder.**to_dict**()
Placeholder.

Holder.**to_pretty**(*format='str'*)
Placeholder.

@pwkit.**Holder**
Placeholder decorator documentation.

2.1.2 Utilities for exceptions

PKError (fmt, *args):
Placeholder.

reraise_context (fmt, *args):
Placeholder.

2.1.3 Abstractions between Python versions 2 and 3

pwkit.**text_type**
The builtin class corresponding to text in this Python interpreter: either `unicode` in Python 2, or `str` in Python 3.

pwkit.**binary_type**
The builtin class corresponding to binary data in this Python interpreter: either `str` in Python 2, or `bytes` in Python 3.

pwkit.**unicode_to_str**(*s*)
A function for implementing the `__str__` method of classes, the meaning of which differs between Python versions 2 and 3. In all cases, you should implement `__unicode__` on your classes. Setting the `__str__` property of a class to *`unicode_to_str()`* will cause it to Do The Right Thing™, which means returning the UTF-8 encoded version of its Unicode expression in Python 2, or returning the Unicode expression directly in Python 3:

```
import pwkit

class MyClass (object):
    def __unicode__ (self):
        return u'my value'
```

```
    __str__ = pwkit.unicode_to_str
```

2.2 Convenient file input and output (`pwkit.io`)

The `pwkit` package provides many tools to ease reading and writing data files. The most generic such tools are located in the *`pwkit.io`* module. The most important tool is the *`Path`* class for object-oriented navigation of the filesystem.

class `pwkit.io.`**`Path`**(*path*)

This is an extended version of the `pathlib.Path` class. (`pathlib` is built into Python 3.x and is available as a backport to Python 2.x.) It represents a path on the filesystem.

The key methods on *`Path`* instances are:

- *`absolute()`* — see also *`resolve()`*
- *`as_hdf_store()`*
- *`as_uri()`*
- *`chmod()`*
- *`cwd()`*
- *`ensure_parent()`*
- *`exists()`*
- *`expand()`*
- *`glob()`*
- *`is_absolute()`*
- *`is_block_device()`*
- *`is_char_device()`*
- *`is_dir()`*
- *`is_fifo()`*
- *`is_file()`*
- *`is_socket()`*
- *`is_symlink()`*
- *`iterdir()`* — see also *`scandir()`*
- *`joinpath()`*
- *`make_relative()`*
- *`match()`*
- *`mkdir()`*
- *`open()`* — see also *`try_open()`*
- *`read_lines()`*
- *`read_fits()`* — see also *`read_fits_bintable()`*
- *`read_fits_bintable()`* — see also *`read_fits()`*

- *read_hdf()*
- *read_inifile()*
- *read_numpy_text()*
- *read_pandas()*
- *read_pickle()*
- *read_pickles()*
- *read_tabfile()*
- *relative_to()* — see also *make_relative()*
- *rellink_to()* — see also *symlink_to()*
- *rename()*
- *resolve()* — see also *absolute()*
- *rglob()*
- *rmdir()* — see also *rmtree()*
- *rmtree()* — see also *rmdir()*
- *scandir()* — see also *iterdir()*
- *stat()*
- *symlink_to()* — see also *rellink_to()*
- *touch()*
- *try_open()* — see also *open()*
- *try_unlink()* — see also *unlink()*
- *unlink()* — see also *try_unlink()*
- *with_name()*
- *with_suffix()*
- *write_pickle()*
- *write_pickles()*

Attributes are:

- *anchor*
- *drive*
- *name*
- *parts*
- *parent*
- *parents*
- root
- *stem*
- *suffix*
- *suffixes*

There are also some *free functions* in the `pwkit.io` module, but they are generally being superseded by operations on `Path` objects.

2.2.1 `Path` methods

`Path.`**`absolute`**`()`
Return an absolute version of the path. Unlike `resolve()`, does not normalize the path or resolve symlinks.

`Path.`**`as_hdf_store`**`(`*mode='r'*, ***kwargs*`)`
Return the path as an opened `pandas.HDFStore` object. Note that the `HDFStore` constructor unconditionally prints messages to standard output when opening and closing files, so use of this function will pollute your program's standard output. The *kwargs* are forwarded to the `HDFStore` constructor.

`Path.`**`as_uri`**`()`
Return the path stringified as a *file:///* URI.

`Path.`**`chmod`**`(`*mode*`)`
Change the mode of the named path. Remember to use octal `0o755` notation!

`Path.`**`ensure_parent`**`(`*mode=0o777*, *parents=False*`)`
Ensure that this path's *parent* directory exists. Returns a boolean indicating whether the parent directory already existed. Will attempt to create superior parent directories if *parents* is true. Unlike `Path.mkdir()`, will not raise an exception if parents already exist.

`Path.`**`exists`**`()`
Returns whether the path exists.

`Path.`**`expand`**`(`*user=False*, *vars=False*, *glob=False*, *resolve=False*`)`
Return a new `Path` with various expansions performed. All expansions are disabled by default but can be enabled by passing in true values in the keyword arguments.

user [bool (default False)] Expand `~` and `~user` home-directory constructs. If a username is unmatched or `$HOME` is unset, no change is made. Calls `os.path.expanduser()`.

vars [bool (default False)] Expand `$var` and `${var}` environment variable constructs. Unknown variables are not substituted. Calls `os.path.expandvars()`.

glob [bool (default False)] Evaluate the path as a `glob` expression and use the matched path. If the glob does not match anything, do not change anything. If the glob matches more than one path, raise an `IOError`.

resolve [bool (default False)] Call `resolve()` on the return value before returning it.

`Path.`**`glob`**`(`*pattern*`)`
Assuming that the path is a directory, iterate over its contents and return sub-paths matching the given shell-style glob pattern.

`Path.`**`is_absolute`**`()`
Returns whether the path is absolute.

`Path.`**`is_block_device`**`()`
Returns whether the path resolves to a block device file.

`Path.`**`is_char_device`**`()`
Returns whether the path resolves to a character device file.

`Path.`**`is_dir`**`()`
Returns whether the path resolves to a directory.

`Path.`**`is_fifo`**`()`
Returns whether the path resolves to a Unix FIFO.

Path.**is_file**()
Returns whether the path resolves to a regular file.

Path.**is_socket**()
Returns whether the path resolves to a Unix socket.

Path.**is_symlink**()
Returns whether the path resolves to a symbolic link.

Path.**iterdir**()
Assuming the path is a directory, generate a sequence of sub-paths corresponding to its contents.

Path.**joinpath**(**args*)
Combine this path with several new components. If one of the arguments is absolute, all previous components are discarded.

Path.**make_relative**(*other*)
Return a new path that is the equivalent of this one relative to the path *other*. Unlike *relative_to()*, this will not throw an error if *self* is not a sub-path of *other*; instead, it will use `..` to build a relative path. This can result in invalid relative paths if *other* contains a directory symbolic link.

If *self* is an absolute path, it is returned unmodified.

Path.**match**(*pattern*)
Test whether this path matches the given shell glob pattern.

Path.**mkdir**(*mode=0o777*, *parents=False*)
Create a directory at this path location. Creates parent directories if *parents* is true. Raises `OSError` if the path already exists, even if *parents* is true.

Path.**open**(*mode='r'*, *buffering=-1*, *encoding=None*, *errors=None*, *newline=None*)
Open the file pointed at by the path and return a `file` object. **TODO**: verify whether semantics correspond to `io.open()` or plain builtin *open()*.

Path.**read_lines**(*mode='rt'*, *noexistok=False*, ***kwargs*)
Generate a sequence of lines from the file pointed to by this path, by opening as a regular file and iterating over it. The lines therefore contain their newline characters. If *noexistok*, a nonexistent file will result in an empty sequence rather than an exception. *kwargs* are passed to *Path.open()*.

Path.**read_fits**(***kwargs*)
Open as a FITS file, returning a `astropy.io.fits.HDUList` object. Keyword arguments are passed to `astropy.io.fits.open()`; valid ones likely include:

- `mode = 'readonly'` (or "update", "append", "denywrite", "ostream")
- `memmap = None`
- `save_backup = False`
- `cache = True`
- `uint = False`
- `ignore_missing_end = False`
- `checksum = False`
- `disable_image_compression = False`
- `do_not_scale_image_data = False`
- `ignore_blank = False`
- `scale_back = False`

Path.**read_fits_bintable**(*hdu=1*, *drop_nonscalar_ok=True*, ***kwargs*)
Open as a FITS file, read in a binary table, and return it as a `pandas.DataFrame`, converted with `pkwit.numutil.fits_recarray_to_data_frame()`. The *hdu* argument specifies which HDU to read, with its default 1 indicating the first FITS extension. The *drop_nonscalar_ok* argument specifies if non-scalar table values (which are inexpressible in `pandas.DataFrame`s) should be silently ignored (``True`) or cause a `ValueError` to be raised (`False`). Other **kwargs** are passed to `astropy.io.fits.open()`, (see *Path.read_fits()*) although the open mode is hardcoded to be `"readonly"`.

Path.**read_hdf**(*key*, ***kwargs*)
Open as an HDF5 file using `pandas` and return the item stored under the key *key*. *kwargs* are passed to `pandas.read_hdf()`.

Path.**read_inifile**(*noexistok=False*, *typed=False*)
Open assuming an "ini-file" format and return a generator yielding data records using either *pwkit.inifile.read_stream()* (if *typed* is false) or `pwkit.tinifile.read_stream()` (if it's true). The latter version is designed to work with numerical data using the *pwkit.msmt* subsystem. If *noexistok* is true, a nonexistent file will result in no items being generated rather than an `IOError` being raised.

Path.**read_numpy_text**(***kwargs*)
Read this path into a `numpy.ndarray` as a text file using `numpy.loadtxt()`. In normal conditions the returned array is two-dimensional, with the first axis spanning the rows in the file and the second axis columns (but see the *unpack* keyword). *kwargs* are passed to `numpy.loadtxt()`; they likely are:

dtype [data type] The data type of the resulting array.

comments [str] If specific, a character indicating the start of a comment.

delimiter [str] The string that separates values. If unspecified, any span of whitespace works.

converters [dict] A dictionary mapping zero-based column *number* to a function that will turn the cell text into a number.

skiprows [int (default=0)] Skip this many lines at the top of the file

usecols [sequence] Which columns keep, by number, starting at zero.

unpack [bool (default=False)] If true, the return value is transposed to be of shape `(cols, rows)`.

ndmin [int (default=0)] The returned array will have at least this many dimensions; otherwise mono-dimensional axes will be squeezed.

Path.**read_pandas**(*format='table'*, ***kwargs*)
Read using `pandas`. The function `pandas.read_FORMAT` is called where `FORMAT` is set from the argument *format*. *kwargs* are passed to this function. Supported formats likely include `clipboard`, `csv`, `excel`, `fwf`, `gbq`, `html`, `json`, `msgpack`, `pickle`, `sql`, `sql_query`, `sql_table`, `stata`, `table`. Note that `hdf` is not supported because it requires a non-keyword argument; see *Path.read_hdf()*.

Path.**read_pickle**()
Open the file, unpickle one object from it using `cPickle`, and return it.

Path.**read_pickles**()
Generate a sequence of objects by opening the path and unpickling items until EOF is reached.

Path.**read_tabfile**(*tabwidth=8*, *mode='rt'*, *noexistok=False*, ***kwargs*)
Read this path as a table of typed measurements via *pwkit.tabfile.read()*. Returns a generator for a sequence of *pwkit.Holder* objects, one for each row in the table, with attributes for each of the columns.

tabwidth [int (default=8)] The tab width to assume. Defaults to 8 and should not be changed unless absolutely necessary.

mode [str (default='rt')] The file open mode, passed to `io.open()`.

noexistok [bool (default=False)] If true, a nonexistent file will result in no items being generated, as opposed to an `IOError`.

kwargs [keywords] Additional arguments are passed to `io.open()`.

`Path.`**`relative_to`**`(`*`*other`*`)`
Return this path as made relative to another path identified by *other*. If this is not possible, raise `ValueError`.

`Path.`**`rellink_to`**`(`*`target`*`, `*`force=False`*`)`
Make this path a symlink pointing to the given *target*, generating the proper relative path using *`make_relative()`*. This gives different behavior than *`symlink_to()`*. For instance, `Path('a/b').symlink_to('c')` results in `a/b` pointing to the path `c`, whereas *`rellink_to()`* results in it pointing to `../c`. This can result in broken relative paths if (continuing the example) `a` is a symbolic link to a directory.

If either *target* or *self* is absolute, the symlink will point at the absolute path to *target*. The intention is that if you're trying to link `/foo/bar` to `bee/boo`, it probably makes more sense for the link to point to `/path/to/.../bee/boo` rather than `../../../../bee/boo`.

If *force* is true, *`try_unlink()`* will be called on *self* before the link is made, forcing its re-creation.

`Path.`**`rename`**`(`*`target`*`)`
Rename this path to *target*.

`Path.`**`resolve`**`()`
Make this path absolute, resolving all symlinks and normalizing.

`Path.`**`rglob`**`(`*`pattern`*`)`
Recursively yield all files and directories matching the shell glob pattern *pattern* below this path.

`Path.`**`rmdir`**`()`
Delete this path, if it is an empty directory.

`Path.`**`rmtree`**`()`
Recursively delete this directory and its contents. If any errors are encountered, they will be printed to standard error.

`Path.`**`scandir`**`()`
Iteratively scan this path, assuming it's a directory. This requires and uses the `scandir` module. The generated values are `scandir.DirEntry` objects which have some information pre-filled. These objects have methods `inode()`, `is_dir()`, `is_file()`, `is_symlink()`, and `stat()`. They have attributes `name` (the basename of the entry) and `path` (its full path).

`Path.`**`stat`**`()`
Run `os.stat()` on the path and return the result.

`Path.`**`symlink_to`**`(`*`target`*`, `*`target_is_directory=False`*`)`
Make this path a symlink pointing to the given target.

`Path.`**`touch`**`(`*`mode=0o666`*`, `*`exist_ok=True`*`)`
Create a file at this path with the given mode, if needed.

`Path.`**`try_open`**`(`*`null_if_noexist=False`*`, `*`**kwargs`*`)`
Call *`Path.open()`* on this path (passing *kwargs*) and return the result. If the file doesn't exist, the behavior depends on *null_if_noexist*. If it is false (the default), `None` is returned. Otherwise, `os.devnull` is opened and returned.

`Path.`**`try_unlink`**`()`
Try to unlink this path. If it doesn't exist, no error is returned. Returns a boolean indicating whether the path was really unlinked.

`Path.`**`unlink`**`()`
Unlink this file or symbolic link.

`Path.`**`with_name`**`(`*name*`)`
Return a new path with the file name changed.

`Path.`**`with_suffix`**`(`*suffix*`)`
Return a new path with the file suffix changed, or a new suffix added if there was none before. *suffix* should start with a `"."`.

`Path.`**`write_pickle`**`(`*obj*`)`
Dump *obj* to this path using `cPickle`.

`Path.`**`write_pickles`**`(`*objs*`)`
objs must be iterable. Write each of its values to this path in sequence using `cPickle`.

static `Path.`**`cwd`**`()`
Returns a new path containing the absolute path of the current working directory.

2.2.2 `Path` attributes

`Path.`**`anchor`**
The concatenation of *`Path.drive`* and `Path.root`.

`Path.`**`drive`**
The Windows or network drive of the path. The empty string on POSIX.

`Path.`**`name`**
The final path component.

`Path.`**`parts`**
A tuple of the path components. The path `/a/b` maps to `("/", "a", "b")`.

`Path.`**`parent`**
The path's logical parent; that is, the path with the final component removed. The parent of `foo` is `.`; the parent of `.` is `.`; the parent of `/` is `/`.

`Path.`**`parents`**
An immutable sequence giving the logical ancestors of the path. Given a *`Path`* `p`, `p.parents[0]` is the same as `p.parent`, `p.parents[1]` matches `p.parent.parent`, and so on. This item is of finite size, however, so going too far (e.g. `p.parents[17]`) will yield an `IndexError`.

`Path.`**`stem`**
The final component without its suffix. The stem of `"foo.tar.gz"` is `"foo.tar"`.

`Path.`**`suffix`**
The suffix of the final path component. The suffix of `"foo.tar.gz"` is `".gz"`.

`Path.`**`suffixes`**
A list of all suffixes on the final component. The suffixes of `"foo.tar.gz"` are `[".tar", ".gz"]`.

2.2.3 Other functions in `pwkit.io`

These are generally superseded by operations on *`Path`*.

`pwkit.io.`**`try_open`**`(`**args*, ***kwargs*`)`
Placeholder.

`pwkit.io.`**`words`**`(`*linegen*`)`
Placeholder.

pwkit.io.**pathwords**(*path*, *mode='rt'*, *noexistok=False*, ***kwargs*)
Placeholder.

pwkit.io.**pathlines**(*path*, *mode='rt'*, *noexistok=False*, ***kwargs*)
Placeholder.

pwkit.io.**make_path_func**(**baseparts*)
Placeholder.

pwkit.io.**djoin**(**args*)
Placeholder.

pwkit.io.**rellink**(*source*, *dest*)
Placeholder.

pwkit.io.**ensure_dir**(*path*, *parents=False*)
Placeholder.

pwkit.io.**ensure_symlink**(*src*, *dst*)
Placeholder.

2.3 Numerical utilities (pwkit.numutil)

The numpy and scipy packages provide a whole host of routines, but there are still some that are missing. The *pwkit.numutil* module provides several useful additions:

- *Making functions that auto-broadcast their arguments*
- *Convenience functions for statistics*
- *Convenience functions for pandas.DataFrame objects*
- *Parallelized versions of simple math algorithms*
- *Tophat and Step Functions*

2.3.1 Making functions that auto-broadcast their arguments

@pwkit.numutil.**broadcastize**(*n_arr*, *ret_spec=0*, *force_float=True*)
Wrap a function to automatically broadcast numpy.ndarray arguments.

It's often desirable to write numerical utility functions in a way that's compatible with vectorized processing. It can be tedious to do this, however, since the function arguments need to turned into arrays and checked for compatible shape, and scalar values need to be special cased.

The @broadcastize decorator takes care of these matters. The decorated function can be implemented in vectorized form under the assumption that all array arguments have been broadcast to the same shape. The broadcasting of inputs and (potentially) de-vectorizing of the return values are done automatically. For instance, if you decorate a function foo(x,y) with @numutil.broadcastize(2), you can implement it assuming that both *x* and *y* are numpy.ndarray objects that have at least one dimension and are both of the same shape. If the function is called with only scalar arguments, *x* and *y* will have shape (1,) and the function's return value will be turned back into a scalar before reaching the caller.

The *n_arr* argument specifies the number of array arguments that the function takes. These are required to be at the beginning of its argument list.

The *ret_spec* argument specifies the structure of the function's return value.

- `0` indicates that the value has the same shape as the (broadcasted) vector arguments. If the arguments are all scalar, the return value will be scalar too.
- `1` indicates that the value is an array of higher rank than the input arguments. For instance, if the input has shape `(3,)`, the output might have shape `(4,4,3)`; in general, if the input has shape `s`, the output will have shape `t + s` for some tuple `t`. If the arguments are all scalar, the output will have a shape of just `t`. The `numpy.asarray()` function is called on such arguments, so (for instance) you can return a list of arrays `[a, b]` and it will be converted into a `numpy.ndarray`.
- `None` indicates that the value is completely independ of the inputs. It is returned as-is.
- A tuple `t` indicates that the return value is also a tuple. The elements of the *ret_spec* tuple should contain the values listed above, and each element of the return value will be handled accordingly.

The default *ret_spec* is `0`, i.e. the return value is expected to be an array of the same shape as the argument(s).

If *force_float* is true (the default), the input arrays will be converted to floating-point types if necessary (with `numpy.asfarray()`) before being passed to the function.

Example:

```
@numutil.broadcastize (2, ret_spec=(0, 1, None)):
def myfunction (x, y, extra_arg):
    print ('a random non-vector argument is:', extra_arg)
    z = x + y
    z[np.where (y)] *= 2
    higher_vector = [x, y, z]
    return z, higher_vector, 'hello'
```

2.3.2 Convenience functions for statistics

pwkit.numutil.**rms**(*x*)
Placeholder.

pwkit.numutil.**weighted_mean**(*values*, *uncerts*, ***kwargs*)
Placeholder.

pwkit.numutil.**weighted_mean_df**(*df*, ***kwargs*)
The same as *weighted_mean()*, except the argument is expected to be a two-column `pandas.DataFrame` whose first column gives the data values and second column gives their uncertainties. Returns `(weighted_mean, uncertainty_in_mean)`.

pwkit.numutil.**weighted_variance**(*x*, *weights*)
Placeholder.

2.3.3 Convenience functions for `pandas.DataFrame` objects

pwkit.numutil.**reduce_data_frame**(*df*, *chunk_slicers*, *avg_cols=()*, *uavg_cols=()*, *minmax_cols=()*, *nchunk_colname=u'nchunk'*, *uncert_prefix=u'u'*, *min_points_per_chunk=3*)
Placeholder.

pwkit.numutil.**reduce_data_frame_evenly_with_gaps**(*df*, *valcol*, *target_len*, *maxgap*, ***kwargs*)
Placeholder.

pwkit.numutil.**slice_around_gaps**(*values*, *maxgap*)
Placeholder.

pwkit.numutil.**slice_evenly_with_gaps**(*values*, *target_len*, *maxgap*)
Placeholder.

pwkit.numutil.**dfsmooth**(*window*, *df*, *ucol*, *k=None*)
Placeholder.

pwkit.numutil.**fits_recarray_to_data_frame**(*recarray*, *drop_nonscalar_ok=True*)
Convert a FITS data table, stored as a Numpy record array, into a Pandas DataFrame object. By default, non-scalar columns are discarded, but if *drop_nonscalar_ok* is False then a `ValueError` is raised. Column names are lower-cased. Example:

```
from pwkit import io, numutil
hdu_list = io.Path ('my-table.fits').read_fits ()
# assuming the first FITS extension is a binary table:
df = numutil.fits_recarray_to_data_frame (hdu_list[1].data)
```

FITS data are big-endian, whereas nowadays almost everything is little-endian. This seems to be an issue for Pandas DataFrames, where `df[['col1', 'col2']]` triggers an assertion for me if the underlying data are not native-byte-ordered. This function normalizes the read-in data to native endianness to avoid this.

See also *pwkit.io.Path.read_fits_bintable()*.

pwkit.numutil.**data_frame_to_astropy_table**(*dataframe*)
This is a backport of the Astropy method `astropy.table.table.Table.from_pandas()`. It converts a Pandas `pandas.DataFrame` object to an Astropy `astropy.table.Table`.

pwkit.numutil.**usmooth**(*window*, *uncerts*, **data*, ***kwargs*)
Placeholder.

pwkit.numutil.**page_data_frame**(*df, pager_argv=[u'less'], **kwargs*)
Render a DataFrame as text and send it to a terminal pager program (e.g. *less*), so that one can browse a full table conveniently.

df The DataFrame to view

pager_argv: default `['less']` A list of strings passed to `subprocess.Popen` that launches the pager program

kwargs Additional keywords are passed to `pandas.DataFrame.to_string()`.

Returns `None`. Execution blocks until the pager subprocess exits.

2.3.4 Parallelized versions of simple math algorithms

pwkit.numutil.**parallel_newton**(*func*, *x0*, *fprime=None*, *par_args=()*, *simple_args=()*, *tol=1.48e-8*, *maxiter=50*, *parallel=True*, ***kwargs*)
Placeholder. A parallelized version of `scipy.optimize.newton()`.

pwkit.numutil.**parallel_quad**(*func*, *a*, *b*, *par_args=()*, *simple_args=()*, *parallel=True*, ***kwargs*)
Placeholder. A parallelized version of `scipy.integrate.quad()`.

2.3.5 Tophat and Step Functions

pwkit.numutil.**unit_tophat_ee**(*x*)
Placeholder.

pwkit.numutil.**unit_tophat_ei**(*x*)
Placeholder.

pwkit.numutil.**unit_tophat_ie**(*x*)
Placeholder.

pwkit.numutil.**unit_tophat_ii**(*x*)
Placeholder.

pwkit.numutil.**make_tophat_ee**(*lower*, *upper*)
Placeholder.

pwkit.numutil.**make_tophat_ei**(*lower*, *upper*)
Placeholder.

pwkit.numutil.**make_tophat_ie**(*lower*, *upper*)
Placeholder.

pwkit.numutil.**make_tophat_ii**(*lower*, *upper*)
Placeholder.

pwkit.numutil.**make_step_lcont**(*transition*)
Placeholder.

pwkit.numutil.**make_step_rcont**(*transition*)
Placeholder.

2.4 Framework for easy parallelized processing (pwkit.parallel)

parallel - Tools for parallel processing.

Functions:

make_parallel_helper Return an object that sets up parallel computations.

See the make_parallel_helper() documentation for more details, but in short:

```
from pwkit.parallel import make_parallel_helper

def my_parallelizable_function (arg1, arg1, parallel=True):
    phelp = make_parallel_helper (parallel)
    ...

    with phelp.get_map () as map:
       results1 = map (my_subfunc1, subargs1)
       ...
       results2 = map (my_subfunc2, subargs2)

    ... do stuff with results1 and results2 ...
```

Setting *parallel=True* will use all cores. *parallel=0.5* will use about half your machine. *parallel=False* will use serial processing. The helper must be used as a context manager (the "with" statement) because the parallel computation may involve creating and destroying heavyweight resources (namely, child processes).

Along with standard *map*, *ParallelHelper* instances support a "partially-Pickling" *map*-like function *ppmap* that works around Pickle-related limitations in the *multiprocessing* library. See the docs for *pwkit.parallel.serial_ppmap* for usage information.

pwkit.parallel.**make_parallel_helper**(*parallel_arg*, ***kwargs*)
Return a *ParallelHelper* object that can be used for easy parallelization of computations. *parallel_arg* is an object that lets the caller easily specify the kind of parallelization they are interested in. Allowed values are:

False Serial processing only.

True Parallel processing using all available cores.

1 Equivalent to *False*.

(other positive integer) Parallel processing using the specified number of cores.

x, 0 < x < 1 Parallel processing using about (x*N) cores, where N is the total number of cores in the system. Note that the meanings of *0.99* and *1* as arguments are very different.

(ParallelHelper instance) Returns the instance.

The `**kwargs` are passed on to the appropriate ParallelHelper constructor, if the caller wants to do something tricky.

Expected usage is:

```
from pwkit.parallel import make_parallel_helper

def sub_operation (arg):
    ... do some computation ...
    return result

def my_parallelizable_function (arg1, arg2, parallel=True):
    phelp = make_parallel_helper (parallel)

    with phelp.get_map () as map:
        op_results = map (sub_operation, args)

    ... reduce "op_results" in some way ...
    return final_result
```

This means that *my_parallelizable_function* doesn't have to worry about all of the various fancy things the caller might want to do in terms of special parallel magic.

Note that *sub_operation* above must be defined in a stand-alone fashion because of the way Python's *multiprocessing* module works. This can be worked around somewhat with the special *get_ppmap* variant. This returns a "partially-Pickling" map operation – with a different calling signature – that allows un-Pickle-able values to be used. See the documentation for *pwkit.parallel.serial_ppmap* for usage information.

2.5 Quick enumerations of constant values (`pwkit.simpleenum`)

The *`pwkit.simpleenum`* module contains a single decorator function for creating "enumerations", by which we mean a group of named, un-modifiable values. For example:

```
from pwkit.simpleenum import enumeration

@enumeration
class Constants (object):
  period_days = 2.771
  period_hours = period_days * 24
  n_iters = 300
  # etc

def myfunction ():
  print ('the period is', Constants.period_hours, 'hours')
```

The `class` declaration syntax is handy here because it lets you define new values in relation to old values. In the above example, you cannot change any of the properties of `Constants` once it is constructed.

Important: If you populate an enumeration with a mutable data type, however, we're unable to prevent you from modifying it. For instance, if you do this:

```
@enumeration
class Dangerous (object):
  mutable = [1, 2]
  immutable = (1, 2)
```

You can then do something like write `Dangerous.mutable.append (3)` and modify the value stored in the enumeration. If you're concerned about this, make sure to populate the enumeration with immutable classes such as `tuple`, `frozenset`, `int`, and so on.

---

pwkit.simpleenum.**enumeration**(*cls*)

A very simple decorator for creating enumerations. Unlike Python 3.4 enumerations, this just gives a way to use a class declaration to create an immutable object containing only the values specified in the class.

If the attribute `__pickle_compat__` is set to True in the decorated class, the resulting enumeration value will be callable such that `EnumClass(x) = x`. This is needed to unpickle enumeration values that were previously implemented using `enum.Enum`.

CHAPTER 3

Scientific Algorithms

This documentation has a lot of stubs.

3.1 Basic astronomical calculations (`pwkit.astutil`)

This module collects many utilities for performing basic astronomical calculations, including:

- *Useful Constants*
- *Sexagesimal Notation*
- *Working with Angles*
- *Simple Operations on 2D Gaussians*
- *Basic Astrometry*

3.1.1 Useful Constants

`pwkit.astutil.`**`pi`**
Placeholder.

`pwkit.astutil.`**`twopi`**
Placeholder.

`pwkit.astutil.`**`halfpi`**
Placeholder.

`pwkit.astutil.`**`R2A`**
Placeholder.

`pwkit.astutil.`**`A2R`**
Placeholder.

`pwkit.astutil.`**`R2D`**
Placeholder.

`pwkit.astutil.`**`D2R`**
Placeholder.

`pwkit.astutil.`**`R2H`**
Placeholder.

pwkit.astutil.**H2R**
Placeholder.

pwkit.astutil.**F2S**
Placeholder.

pwkit.astutil.**S2F**
Placeholder.

pwkit.astutil.**J2000**
Placeholder.

3.1.2 Sexagesimal Notation

pwkit.astutil.**fmthours**(*radians*, *norm='wrap'*, *precision=3*, *seps='::'*)
Placeholder.

pwkit.astutil.**fmtdeglon**(*radians*, *norm='wrap'*, *precision=2*, *seps='::'*)
Placeholder.

pwkit.astutil.**fmtdeglat**(*radians*, *norm='raise'*, *precision=2*, *seps='::'*)
Placeholder.

pwkit.astutil.**fmtradec**(*rarad*, *decrad*, *precision=2*, *raseps='::'*, *decseps='::'*, *intersep=' '*)
Placeholder.

pwkit.astutil.**parsehours**(*hrstr*)
Placeholder.

pwkit.astutil.**parsedeglat**(*latstr*)
Placeholder.

pwkit.astutil.**parsedeglon**(*lonstr*)
Placeholder.

3.1.3 Working with Angles

pwkit.astutil.**angcen**(*a*)
Placeholder.

pwkit.astutil.**orientcen**(*a*)
Placeholder.

pwkit.astutil.**sphdist**(*lat1*, *lon1*, *lat2*, *lon2*)
Placeholder.

pwkit.astutil.**sphbear**(*lat1*, *lon1*, *lat2*, *lon2*, *tol=1e-15*)
Placeholder.

pwkit.astutil.**sphofs**(*lat1*, *lon1*, *r*, *pa*, *tol=1e-2*, *rmax=None*)
Placeholder.

pwkit.astutil.**parang**(*hourangle*, *declination*, *latitude*)
Placeholder.

3.1.4 Simple Operations on 2D Gaussians

pwkit.astutil.**gaussian_convolve**(*maj1*, *min1*, *pa1*, *maj2*, *min2*, *pa2*)
Placeholder.

pwkit.astutil.**gaussian_deconvolve**(*smaj*, *smin*, *spa*, *bmaj*, *bmin*, *bpa*)
Placeholder.

3.1.5 Basic Astrometry

pwkit.astutil.**get_2mass_epoch**(*tmra*, *tmdec*, *debug=False*)
Placeholder.

pwkit.astutil.**get_simbad_astrometry_info**(*ident*, *items=...*, *debug=False*)
Placeholder.

class pwkit.astutil.**AstrometryInfo**(*simbadident=None*, ***kwargs*)
Placeholder.

AstrometryInfo.**verify**(*complain=True*)
Placeholder.

AstrometryInfo.**predict**(*mjd*, *complain=True*, *n=20000*)
Placeholder.

AstrometryInfo.**prin_prediction**(*ptup*)
Placeholder.

AstrometryInfo.**fill_from_simbad**(*ident*, *debug=False*)
Placeholder.

3.2 File-format-agnostic loading of astronomical images (pwkit.astimage)

pwkit.astimage – generic loading of (radio) astronomical images

Use *open (path, mode)* to open an astronomical image, regardless of its file format.

The emphasis of this module is on getting 90%-good-enough semantics and a really, genuinely, uniform interface. This can be tough to achieve.

class pwkit.astimage.**AstroImage**(*path*, *mode*)
An astronomical image.

path The filesystem path of the image.

mode Its access mode: 'r' for read, 'rw' for read/write.

shape The data shape, like numpy.ndarray.shape.

bmaj If not None, the restoring beam FWHM major axis in radians.

bmin If not None, the restoring beam FWHM minor axis in radians.

bpa If not None, the restoring beam position angle (east from celestial north) in radians.

units Lower-case string describing image units (e.g., jy/beam, jy/pixel). Not standardized between formats.

pclat Latitude (usually dec) of the pointing center in radians.

pclon Longitude (usually RA) of the pointing center in radians.

charfreq Characteristic observing frequency of the image in GHz.

mjd Mean MJD of the observations.

axdescs If not None, list of strings describing the axis types. Not standardized.

size The number of pixels in the image (=shape.prod ()).

Methods:

close Close the image.

read Read all of the data.

write Rewrite all of the data.

toworld Convert pixel coordinates to world coordinates.

topixel Convert world coordinates to pixel coordinates.

simple Convert to a 2D lat/lon image.

subimage Extract a sub-cube of the image.

save_copy Save a copy of the image.

save_as_fits Save a copy of the image in FITS format.

delete Delete the on-disk image.

subimage(*pixofs*, *shape*)
Extract a sub-cube of this image.

Both *pixofs* and *shape* should be integer arrays with as many elements as this image has axes. Thinking of this operation as taking a Python slice of an N-dimensional cube, the i'th axis of the sub-image is slices from *pixofs[i]* to *pixofs[i] + shape[i]*.

class pwkit.astimage.**MIRIADImage**(*path*, *mode*)
A MIRIAD format image. Requires the *mirtask* module from miriad-python.

class pwkit.astimage.**PyrapImage**(*path*, *mode*)
A CASA-format image loaded with the 'pyrap' Python module.

class pwkit.astimage.**FITSImage**(*path*, *mode*)
A FITS format image.

class pwkit.astimage.**SimpleImage**(*parent*)
A 2D, latitude/longitude image, referenced to a parent image.

3.3 The Bayesian Blocks algorithm (pwkit.bblocks)

pwkit.bblocks - Bayesian Blocks analysis, with a few extensions.

Bayesian Blocks analysis for the "time tagged" case described by Scargle+ 2013. Inspired by the bayesian_blocks implementation by Jake Vanderplas in the AstroML package, but that turned out to have some limitations.

We have iterative determination of the best number of blocks (using an ad-hoc routine described in Scargle+ 2013) and bootstrap-based determination of uncertainties on the block heights (ditto).

Functions are:

bin_bblock() Bayesian Blocks analysis with counts and bins.

tt_bblock() BB analysis of time-tagged events.

bs_tt_bblock() Like *tt_bblock()* with bootstrap-based uncertainty assessment. NOTE: the uncertainties are not very reliable!

pwkit.bblocks.**bin_bblock**(*widths*, *counts*, *p0=0.05*)
Fundamental Bayesian Blocks algorithm. Arguments:

widths - Array of consecutive cell widths. counts - Array of numbers of counts in each cell. p0=0.05 - Probability of preferring solutions with additional bins.

Returns a Holder with:

blockstarts - Start times of output blocks. counts - Number of events in each output block. finalp0 - Final value of p0, after iteration to minimize *nblocks*. nblocks - Number of output blocks. ncells - Number of input cells/bins. origp0 - Original value of p0. rates - Event rate associated with each block. widths - Width of each output block.

pwkit.bblocks.**tt_bblock**(*tstarts*, *tstops*, *times*, *p0=0.05*)
Bayesian Blocks for time-tagged events. Arguments:

tstarts - Array of input bin start times. tstops - Array of input bin stop times. times - Array of event arrival times. p0=0.05 - Probability of preferring solutions with additional bins.

Returns a Holder with:

blockstarts - Start times of output blocks. counts - Number of events in each output block. finalp0 - Final value of p0, after iteration to minimize *nblocks*. ledges - Times of left edges of output blocks. midpoints - Times of midpoints of output blocks. nblocks - Number of output blocks. ncells - Number of input cells/bins. origp0 - Original value of p0. rates - Event rate associated with each block. redges - Times of right edges of output blocks. widths - Width of each output block.

Bin start/stop times are best derived from a 1D Voronoi tesselation of the event arrival times, with some kind of global observation start/stop time setting the extreme edges.

pwkit.bblocks.**bs_tt_bblock**(*times*, *tstarts*, *tstops*, *p0=0.05*, *nbootstrap=512*)
Bayesian Blocks for time-tagged events with bootstrapping uncertainty assessment. THE UNCERTAINTIES ARE NOT VERY GOOD! Arguments:

tstarts - Array of input bin start times. tstops - Array of input bin stop times. times - Array of event arrival times. p0=0.05 - Probability of preferring solutions with additional bins. nbootstrap=512 - Number of bootstrap runs to perform.

Returns a Holder with:

blockstarts - Start times of output blocks. bsrates - Mean event rate in each bin from bootstrap analysis. bsrstds - ~Uncertainty: stddev of event rate in each bin from bootstrap analysis. counts - Number of events in each output block. finalp0 - Final value of p0, after iteration to minimize *nblocks*. ledges - Times of left edges of output blocks. midpoints - Times of midpoints of output blocks. nblocks - Number of output blocks. ncells - Number of input cells/bins. origp0 - Original value of p0. rates - Event rate associated with each block. redges - Times of right edges of output blocks. widths - Width of each output block.

3.4 Constants in CGS units (`pwkit.cgs`)

pwkit.cgs - Physical constants in CGS.

Specifically, ESU-CGS im which the electron charge is measured in esu Franklin statcoulomb.

a0 - Bohr radius [cm] alpha - Fine structure constant [ø] arad - Radiation constant [erg/cm³/K] aupercm - AU per cm c - Speed of light [cm/s] cgsperjy - [erg/s/cm²/Hz] per Jy cmperau - cm per AU cmperpc - cm per parsec conjaaev -

eV/Angstrom conjugation factor: AA = conjaaev / eV [Å·eV] e - electron charge [esu] ergperev - erg per eV euler - Euler's constant (2.71828...) [ø] evpererg - eV per erg G - Gravitational constant [cm³/g/s²] h - Planck's constant [erg s] hbar - Reduced Planck's constant [erg·s] jypercgs - Jy per [erg/s/cm²/Hz] k - Boltzmann's constant [erg/K] lsun - Luminosity of the Sun [erg/s] me - Mass of the electron [g] mearth - Mass of the Earth [g] mjup - Mass of Jupiter [g] mp - Mass of the proton [g] msun - Mass of the Sun [g] mu_e - Magnetic moment of the electron [esu·cm²/s] pcpercm - parsec per cm pi - Pi [ø] r_e - Classical radius of the electron [cm] rearth - Radius of the earth [cm] rjup - Radius of Jupiter [cm] rsun - Radius of the Sun [cm] ryd1 - Rydberg energy [erg] sigma - Stefan-Boltzmann constant [erg/s/K] sigma_T - Thomson cross section of the electron [cm²] spersyr - Seconds per sidereal year syrpers - Sidereal years per second tsun - Effective temperature of the Sun [K]

Functions:

blambda - Planck function (Hz, K) -> erg/s/cm²/Hz/sr. bnu - Planck function (cm, K) -> erg/s/cm²/cm/sr. exp - Numpy exp() function. log - Numpy log() function. log10 - Numpy log10() function. sqrt - Numpy sqrt() function.

For reference: the esu has dimensions of g^(1/2) cm^(3/2) s^-1. Electric and magnetic field have g^(1/2) cm^(-1/2) s^-1. [esu * field] = dyne.

3.5 Representations of and computations with ellipses (`pwkit.ellipses`)

pwkit.ellipses - utilities for manipulating 2D Gaussians and ellipses

XXXXXXX XXX this code is in an incomplete state of being vectorized!!! XXXXXXX

Useful for sources and bivariate error distributions. We can express the shape of the function in several ways, which have different strengths and weaknesses:

- "biv", as in Gaussian bivariate: sigma x, sigma y, cov(x,y)
- "ell", as in ellipse: major, minor, PA [*]
- "abc": coefficients such that z = exp (ax² + bxy + cy²)

[*] Any slice through a 2D Gaussian is an ellipse. Ours is defined such it is the same as a Gaussian bivariate when major = minor.

Note that when considering astronomical position angles, conventionally defined as East from North, the Dec/lat axis should be considered the X axis and the RA/long axis should be considered the Y axis.

NOTE: Pineau et al 2011A&A...527A.126P has some relevant equations, including ones for computing the overlap of two error ellipses, which is something I've had trouble figuring out in the past.

`pwkit.ellipses.`**`sigmascale`**(*nsigma*)
: Say we take a Gaussian bivariate and convert the parameters of the distribution to an ellipse (major, minor, PA). By what factor should we scale those axes to make the area of the ellipse correspond to the n-sigma confidence interval?

  Negative or zero values result in NaN.

`pwkit.ellipses.`**`clscale`**(*cl*)
: Say we take a Gaussian bivariate and convert the parameters of the distribution to an ellipse (major, minor, PA). By what factor should we scale those axes to make the area of the ellipse correspond to the confidence interval CL? (I.e. 0 < CL < 1)

`pwkit.ellipses.`**`bivell`**(*sx*, *sy*, *cxy*)
: Given the parameters of a Gaussian bivariate distribution, compute the parameters for the equivalent 2D Gaussian in ellipse form (major, minor, pa).

  Inputs:

- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var
- cxy: covariance (not correlation coefficient) of x and y

Outputs:

- mjr: major axis of equivalent 2D Gaussian (sigma, not FWHM)
- mnr: minor axis
- pa: position angle, rotating from +x to +y

Lots of sanity-checking because it's obnoxiously easy to have numerics that just barely blow up on you.

pwkit.ellipses.**bivnorm**(*sx*, *sy*, *cxy*)

Given the parameters of a Gaussian bivariate distribution, compute the correct normalization for the equivalent 2D Gaussian. It's 1 / (2 pi sqrt (sx**2 sy**2 - cxy**2). This function adds a lot of sanity checking.

Inputs:

- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var
- cxy: covariance (not correlation coefficient) of x and y

Returns: the scalar normalization

pwkit.ellipses.**bivabc**(*sx*, *sy*, *cxy*)

Compute nontrivial parameters for evaluating a bivariate distribution as a 2D Gaussian. Inputs:

- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var
- cxy: covariance (not correlation coefficient) of x and y

Returns: (a, b, c), where z = k exp ($ax^2 + bxy + cy^2$)

The proper value for k can be obtained from bivnorm().

pwkit.ellipses.**databiv**(*xy*, *coordouter=False*, ***kwargs*)

Compute the main parameters of a bivariate distribution from data. The parameters are returned in the same format as used in the rest of this module.

- xy: a 2D data array of shape (2, nsamp) or (nsamp, 2)
- coordouter: if True, the coordinate axis is the outer axis; i.e. the shape is (2, nsamp). Otherwise, the coordinate axis is the inner axis; i.e. shape is (nsamp, 2).

Returns: (sx, sy, cxy)

In both cases, the first slice along the coordinate axis gives the X data (i.e., xy[0] or xy[:,0]) and the second slice gives the Y data (xy[1] or xy[:,1]).

pwkit.ellipses.**bivrandom**(*x0*, *y0*, *sx*, *sy*, *cxy*, *size=None*)

Compute random values distributed according to the specified bivariate distribution.

Inputs:

- x0: the center of the x distribution (i.e. its intended mean)
- y0: the center of the y distribution
- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var

- cxy: covariance (not correlation coefficient) of x and y
- size (optional): the number of values to compute

Returns: array of shape (size, 2); or just (2,), if size was not specified.

The bivariate parameters of the generated data are approximately recoverable by calling 'databiv(retval)'.

pwkit.ellipses.**ellpoint**(*mjr*, *mnr*, *pa*, *th*)

Compute a point on an ellipse parametrically. Inputs:

- mjr: major axis (sigma not FWHM) of the ellipse
- mnr: minor axis (sigma not FWHM) of the ellipse
- pa: position angle (from +x to +y) of the ellipse, radians
- th: the parameter, 0 <= th < 2pi: the eccentric anomaly

Returns: (x, y)

th may be a vector, in which case x and y will be as well.

pwkit.ellipses.**elld2**(*x0*, *y0*, *mjr*, *mnr*, *pa*, *x*, *y*)

Given an 2D Gaussian expressed as an ellipse (major, minor, pa), compute a "squared distance parameter" such that

z = exp (-0.5 * d2)

Inputs:

- x0: position of Gaussian center on x axis
- y0: position of Gaussian center on y axis
- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians
- x: x coordinates of the locations for which to evaluate d2
- y: y coordinates of the locations for which to evaluate d2

Returns: d2, distance parameter defined as above.

x0, y0, mjr, and mnr may be in any units so long as they're consistent. x and y may be arrays (of the same shape), in which case d2 will be an array as well.

pwkit.ellipses.**ellbiv**(*mjr*, *mnr*, *pa*)

Given a 2D Gaussian expressed as an ellipse (major, minor, pa), compute the equivalent parameters for a Gaussian bivariate distribution. We assume that the ellipse is normalized so that the functions evaluate identicall for major = minor.

Inputs:

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

Returns:

- sx: standard deviation (not variance) of x var
- sy: standard deviation (not variance) of y var

- cxy: covariance (not correlation coefficient) of x and y

pwkit.ellipses.**ellabc**(*mjr*, *mnr*, *pa*)
Given a 2D Gaussian expressed as an ellipse (major, minor, pa), compute the nontrivial parameters for its evaluation.

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

Returns: (a, b, c), where z = exp (ax² + bxy + cy²)

pwkit.ellipses.**ellplot**(*mjr*, *mnr*, *pa*)
Utility for debugging.

pwkit.ellipses.**abcell**(*a*, *b*, *c*)
Given the nontrivial parameters for evaluation a 2D Gaussian as a polynomial, compute the equivalent ellipse parameters (major, minor, pa)

Inputs: (a, b, c), where z = exp (ax² + bxy + cy²)

Returns:

- mjr: major axis (sigma not FWHM) of the Gaussian
- mnr: minor axis (sigma not FWHM) of the Gaussian
- pa: position angle (from +x to +y) of the Gaussian, radians

pwkit.ellipses.**abcd2**(*x0*, *y0*, *a*, *b*, *c*, *x*, *y*)
Given an 2D Gaussian expressed as the ABC polynomial coefficients, compute a "squared distance parameter" such that

z = exp (-0.5 * d2)

Inputs:

- x0: position of Gaussian center on x axis
- y0: position of Gaussian center on y axis
- a: such that z = exp (ax² + bxy + cy²)
- b: see above
- c: see above
- x: x coordinates of the locations for which to evaluate d2
- y: y coordinates of the locations for which to evaluate d2

Returns: d2, distance parameter defined as above.

This is pretty trivial.

3.6 Modeling sources in images (`pwkit.immodel`)

pwkit.immodel - Analytical modeling of astronomical images.

This is derived from copl/pylib/bgfit.py and copl/bin/imsrcdebug. I keep on wanting this code so I should put it somewhere more generic. Such as here. Also, given the history, there are a lot more bells and whistles in the code than the currently exposed UI really needs.

3.7 Bayesian confidence intervals for count rates (pwkit.kbn_conf)

pwkit.kbn_conf - calculate Poisson-like confidence intervals assuming a background

This module implements the Bayesian confidence intervals for Poisson processes in a background using the approach described in Kraft, Burrows, & Nousek (1991). That paper provides tables of values; this module can calculate intervals for arbitrary inputs. Requires *scipy*.

This implementation almost directly transcribes the equations. We do, however, work in log-gamma space to try to avoid overflows with large values of N or B.

Functions:

kbn_conf - Compute a single confidence limit. vec_kbn_conf - Vectorized version of *kbn_conf*.

TODO: tests!

pwkit.kbn_conf.**kbn_conf**(*N*, *B*, *CL*)

Given a (integer) number of observed Poisson events *N* and a (real) expected number of background events *B* and a confidence limit *CL* (between 0 and 1), return the confidence interval on the source event rate.

Returns: (Smin, Smax)

This interval is calculated using the Bayesian formalism of Kraft, Burrows, & Nousek (1991), which assumes no uncertainty in *B* and returns the smallest possible interval that satisfies the above properties.

Example: in a certain time interval, 3 events were recorded. Based on external knowledge, it is expected that on average 0.5 background events will be recorded in the same interval. The 95% confidence interval on the source event rate is

```
>>> kbn_conf.kbn_conf (3, 0.5, 0.95)
<<< (0.22156, 7.40188)
```

which agrees with the entry in Table 2 of KBN91.

Reference info: 1991ApJ...374..344K, doi:10.1086/170124

3.8 Nonlinear least-squares minimization with Levenberg-Marquardt (pwkit.lmmin)

pwkit.lmmin - Pythonic, Numpy-based Levenberg-Marquardt least-squares minimizer

Basic usage:

```
from pwkit.lmmin import Problem, ResidualProblem

def yfunc (params, vals):
    vals[:] = {stuff with params}
def jfunc (params, jac):
    jac[i,j] = {deriv of val[j] w.r.t. params[i]}
    # i.e. jac[i] = {deriv of val wrt params[i]}

p = Problem (npar, nout, yfunc, jfunc=None)
solution = p.solve (guess)

p2 = Problem ()
p2.set_npar (npar) # enables configuration of parameter meta-info
p2.set_func (nout, yfunc, jfunc)
```

Main Solution properties:

prob - The Problem. status - Set of strings; presence of 'ftol', 'gtol', or 'xtol' suggests success. params - Final parameter values. perror - 1σ uncertainties on params. covar - Covariance matrix of parameters. fnorm - Final norm of function output. fvec - Final vector of function outputs. fjac - Final Jacobian matrix of d(fvec)/d(params).

Automatic least-squares model-fitting (subtracts "observed" Y values and multiplies by inverse errors):

def yrfunc (params, modelyvalues): modelyvalues[:] = {stuff with params}

def yjfunc (params, modelyjac): jac[i,j] = {deriv of modelyvalue[j] w.r.t. params[i]}

p.set_residual_func (yobs, errinv, yrfunc, jrfunc, reckless=False) p = ResidualProblem (npar, yobs, errinv, yrfunc, jrfunc=None, reckless=False)

Parameter meta-information:

p.p_value (paramindex, value, fixed=False) p.p_limit (paramindex, lower=-inf, upper=+inf) p.p_step (paramindex, stepsize, maxstep=info, isrel=False) p.p_side (paramindex, sidedness) # one of 'auto', 'pos', 'neg', 'two' p.p_tie (paramindex, tiefunc) # pval = tiefunc (params)

solve() status codes:

Solution.status is a set of strings. The presence of a string in the set means that the specified condition was active when the iteration terminated. Multiple conditions may contribute to ending the iteration. The algorithm likely did not converge correctly if none of 'ftol', 'xtol', or 'gtol' are in status upon termination.

'ftol' (MINPACK/MPFIT equiv: 1, 3) "Termination occurs when both the actual and predicted relative reductions in the sum of squares are at most FTOL. Therefore, FTOL measures the relative error desired in the sum of squares."

'xtol' (MINPACK/MPFIT equiv: 2, 3) "Termination occurs when the relative error between two consecutive iterates is at most XTOL. Therefore, XTOL measures the relative error desired in the approximate solution."

'gtol' (MINPACK/MPFIT equiv: 4) "Termination occurs when the cosine of the angle between fvec and any column of the jacobian is at most GTOL in absolute value. Therefore, GTOL measures the orthogonality desired between the function vector and the columns of the jacobian."

'maxiter' (MINPACK/MPFIT equiv: 5) Number of iterations exceeds maxiter.

'feps' (MINPACK/MPFIT equiv: 6) "ftol is too small. no further reduction in the sum of squares is possible."

'xeps' (MINPACK/MPFIT equiv: 7) "xtol is too small. no further improvement in the approximate solution x is possible."

'geps' (MINPACK/MPFIT equiv: 8) "gtol is too small. fvec is orthogonal to the columns of the jacobian to machine precision."

(This docstring contains only usage information. For important information regarding provenance, license, and academic references, see comments in the module source code.)

class pwkit.lmmin.**Problem**(*npar=None*, *nout=None*, *yfunc=None*, *jfunc=None*, *solclass=<class 'pwkit.lmmin.Solution'>*)

A Levenberg-Marquardt problem to be solved. Attributes:

damp Tanh damping factor of extreme function values.

debug_calls If true, information about function calls is printed.

debug_jac If true, information about jacobian calls is printed.

diag Scale factors for parameter derivatives, used to condition the problem.

epsilon The floating-point epsilon value, used to determine step sizes in automatic Jacobian computation.

factor The step bound is *factor* times the initial value times *diag*.

ftol The relative error desired in the sum of squares.

gtol The orthogonality desired between the function vector and the columns of the Jacobian.

maxiter The maximum number of iterations allowed.

normfunc A function to compute the norm of a vector.

solclass A factory for Solution instances.

xtol The relative error desired in the approximate solution.

Methods:

copy Duplicate this *Problem*.

get_ndof Get the number of degrees of freedom in the problem.

get_nfree Get the number of free parameters (fixed/tied/etc pars are not free).

p_value Set the initial or fixed value of a parameter.

p_limit Set limits on parameter values.

p_step Set the stepsize for a parameter.

p_side Set the sidedness with which auto-derivatives are computed for a par.

p_tie Set a parameter to be a function of other parameters.

set_func Set the function to be optimized.

set_npar Set the number of parameters; allows p_* to be called.

set_residual_func Set the function to a standard model-fitting style.

solve Run the algorithm.

solve_scipy Run the algorithm using the Scipy implementation (for testing).

p_side(*idx*, *sidedness*)
Acceptable values for *sidedness* are “auto”, “pos”, “neg”, and “two”.

class pwkit.lmmin.**Solution**(*prob*)
A parameter solution from the Levenberg-Marquard algorithm. Attributes:

ndof - The number of degrees of freedom in the problem. prob - The *Problem*. status - A set of strings indicating which stop condition(s) arose. niter - The number of iterations needed to obtain the solution. perror - The 1σ errors on the final parameters. params - The final best-fit parameters. covar - The covariance of the function parameters. fnorm - The final function norm. fvec - The final function outputs. fjac - The final Jacobian. nfev - The number of function evaluations needed to obtain the solution. njev - The number of Jacobian evaluations needed to obtain the solution.

The presence of ‘ftol’, ‘gtol’, or ‘xtol’ in *status* suggests success.

3.9 Fitting generic models with least-squares minimization (pwkit.lsqmdl)

pwkit.lsqmdl - model data with least-squares fitting

Classes:

Model - Modeling with any function using Levenberg-Marquardt. Parameter - Information about a specific model parameter. PolynomialModel - Modeling with polynomials. ScaleModel - Modeling with a single scale factor. ComposedModel - Modeling with combinations of pluggable components.

ModelComponent - Base class for ComposedModel components. AddConstantComponent - Adds a single value to all data points. AddValuesComponent - Adds a parameter for every data point. AddPolynomialComponent - Adds a polynomial. SeriesComponent - Apply a set of subcomponents in series. MatMultComponent - Combine subcomponents in a matrix multiplication. ScaleComponent - Multiplies the data by a single value.

Usage:

```
m = Model (func, data, [invsigma], [args]).solve (guess).print_soln ()
    # func takes (p1, p2, p3[, *args]) and returns model data
m = PolynomialModel (maxexponent, x, data, [invsigma]).solve ().plot ()
m = ScaleModel (x, data, [invsigma]).solve ().show_cov ()
    # data = m*x
```

The invsigma are *inverse sigmas*, NOT inverse *variances* (the usual statistical weights). Since most applications deal in sigmas, take care to write:

```
m = Model (func, data, 1./uncerts) # right!
```

not:

```
m = Model (func, data, uncerts) # WRONG
```

If you have zero uncertainty on a measurement, too bad.

class pwkit.lsqmdl.**PolynomialModel**(*maxexponent*, *x*, *data*, *invsigma=None*)

Least-squares polynomial fit.

Because this is a very specialized kind of problem, we don't need an initial guess to solve, and we can use fast built-in numerical routines.

The output parameters are named "a0", "a1", ... and are stored in that order in PolynomialModel.params[]. We have `y = sum(x**i * a[i])`, so "a2" = "params[2]" is the quadratic term, etc.

This model does *not* give uncertainties on the derived coefficients. The as_nonlinear() method can be use to get a *Model* instance with uncertainties.

Methods:

as_nonlinear - Return a (lmmin-based) *Model* equivalent to self.

as_nonlinear(*params=None*)

Return a *Model* equivalent to this object. The nonlinear solver is less efficient, but lets you freeze parameters, compute uncertainties, etc.

If the *params* argument is provided, solve() will be called on the returned object with those parameters. If it is *None* and this object has parameters in *self.params*, those will be use. Otherwise, solve() will not be called on the returned object.

class pwkit.lsqmdl.**ScaleModel**(*x*, *data*, *invsigma=None*)

Solve *data* = *m* * *x* for *m*.

3.10 Math with uncertain and censored measurements (pwkit.msmt)

pwkit.msmt - Working with uncertain measurements.

Classes:

Uval - An empirical uncertain value represented by numerical samples. LimitError - Raised on illegal operations on upper/lower limits. Lval - Container for either precise values or upper/lower limits. Textual - A measurement recorded in textual form.

Generic unary functions on measurements:

absolute - abs(x) arccos - As named. arcsin - As named. arctan - As named. cos - As named. errinfo - Get (limtype, repval, plus_1_sigma, minus_1_sigma) expm1 - exp(x) - 1 exp - As named. fmtinfo - Get (typetag, text, is_imprecise) for textual round-tripping. isfinite - True if the value is well-defined and finite. liminfo - Get (limtype, repval) limtype - -1 if the datum is an upper limit; 1 if lower; 0 otherwise. log10 - As named. log1p - log (1+x) log2 - As named. log - As named. negative - -x reciprocal - 1/x repval - Get a "representative" value if x (in case it is uncertain). sin - As named. sqrt - As named. square - x**2 tan - As named. unwrap - Get a version of x on which algebra can be performed.

Generic binary mathematical-ish functions:

add - x + y divide - x / y; floor-integer division should be respected but usually isn't. multiply - x * y power - x ** y subtract - x - y true_divide - x / y, never with floor-integer division typealign - Return (x*, y*) cast to same algebra-friendly type: float, Uval, or Lval.

Miscellaneous functions:

is_measurement - Check whether an object is numerical find_gamma_params - Compute reasonable Γ distribution parameters given mode/stddev. pk_scoreatpercentile - Simplified version of scipy.stats.scoreatpercentile. sample_double_norm - Sample from a quasi-normal distribution with asymmetric variances. sample_gamma - Sample from a Γ distribution with α/β parametrization.

Variables:

lval_unary_math - Dict of unary math functions operating on Lvals. parsers - Dict of type tag to parsing functions. scalar_unary_math - Dict of unary math functions operating on scalars. textual_unary_math - Dict of unary math functions operating on Textuals. UQUANT_UNCERT - Scale of uncertainty assumed for in cases where it's unquantified. uval_default_repval_method - Default method for computing Uval representative values. uval_dtype - The Numpy dtype of Uval data (often ignored!) uval_nsamples - Number of samples used when constructing Uvals uval_unary_math - Dict of unary math functions operating on Uvals.

class `pwkit.msmt.`**`Lval`**(*kind*, *value*)

A container for either precise values or upper/lower limits. Constructed as `Lval (kind, value)`, where *kind* is `"exact"`, `"uncertain"`, `"toinf"`, `"tozero"`, `"pastzero"`, or `"undef"`. Most easily constructed via `Textual.parse()`. Can also be constructed with `Lval.from_other()`.

Supported operations are `unicode() str() repr() -(neg) abs() + - * / ** += -= *= /= **=`.

class `pwkit.msmt.`**`Textual`**(*tkind*, *dkind*, *data*)

A measurement recorded in textual form.

Textual.from_exact (text, tkind='none') - *text* is passed to float() Textual.parse (text, tkind='none') - *text* as described below.

Transformation kinds are 'none', 'log10', or 'positive'. Expressions for values take the form '1.234', '<2', '>3', '~7', '6to8', '7pm0.1', or '12p1m0.3'.

Methods:

unparse() - Return parsed text (but not tkind!) unwrap() - Express as float/Uval/Lval as appropriate. repval(limitsok=False) - Get single scalar "representative" value. limtype() - -1 if upper limit; +1 if lower; 0 otherwise.

Supported operations: unicode() str() repr() [latexification] -(neg) abs() + - * / **

limtype()
Return -1 if this value is an upper limit, 1 if it is a lower limit, 0 otherwise.

repval(*limitsok=False*)
Get a best-effort representative value as a float. This can be DANGEROUS because it discards limit information, which is rarely wise.

class pwkit.msmt.**Uval**(*data*)
An empirical uncertain value, represented by samples.

Constructors are:

- `Uval.from_other()`
- `Uval.from_fixed()`
- `Uval.from_norm()`
- `Uval.from_unif()`
- `Uval.from_double_norm()`
- `Uval.from_gamma()`
- *Uval.from_pcount()*

Key methods are:

- *repvals()*
- *text_pieces()*
- `format()`
- `debug_distribution()`

Supported operations are: `unicode() str() repr() [latexification] + -(sub) * // / % ** += -= *= //= %= /= **= -(neg) ~ abs()`

static from_pcount(*nevents*)
We assume a Poisson process. nevents is the number of events in some interval. The distribution of values is the distribution of the Poisson rate parameter given this observed number of events, where the "rate" is in units of events per interval of the same duration. The max-likelihood value is nevents, but the mean value is nevents + 1. The gamma distribution is obtained by assuming an improper, uniform prior for the rate between 0 and infinity.

repvals(*method*)
Compute representative statistical values for this Uval. *method* may be either 'pct' or 'gauss'.

Returns (best, plus_one_sigma, minus_one_sigma), where *best* is the "best" value in some sense, and the others correspond to values at the ~84 and 16 percentile limits, respectively. Because of the sampled nature of the Uval system, there is no single method to compute these numbers.

The "pct" method returns the 50th, 15.866th, and 84.134th percentile values.

The "gauss" method computes the mean μ and standard deviation σ of the samples and returns [μ, μ+σ, μ-σ].

text_pieces(*method*, *uplaces=2*)
Return (main, dhigh, dlow, sharedexponent), all as strings. The delta terms do not have sign indicators. Any item except the first may be None.

method is passed to Uval.repvals() to compute representative statistical limits.

pwkit.msmt.**errinfo**(*msmt*)
Return (limtype, repval, errval1, errval2). Like m_liminfo, but also provides error bar information for values that have it.

pwkit.msmt.**fmtinfo**(*value*)
Returns (typetag, text, is_imprecise). Unlike other functions that operate on measurements, this also operates on bools, ints, and strings.

pwkit.msmt.**liminfo**(*msmt*)
Return (limtype, repval). *limtype* is -1 for upper limits, 1 for lower limits, and 0 otherwise; repval is a best-effort representative scalar value for this measurement.

pwkit.msmt.**limtype**(*msmt*)
Return -1 if this value is some kind of upper limit, 1 if this value is some kind of lower limit, 0 otherwise.

pwkit.msmt.**repval**(*msmt*, *limitsok=False*)
Get a best-effort representative value as a float. This is DANGEROUS because it discards limit information, which is rarely wise. m_liminfo() or m_unwrap() are recommended instead.

pwkit.msmt.**unwrap**(*msmt*)
Convert the value into the most basic representation that we can do math on: float if possible, then Uval, then Lval.

pwkit.msmt.**find_gamma_params**(*mode*, *std*)
Given a modal value and a standard deviation, compute corresponding parameters for the gamma distribution.

Intended to be used to replace normal distributions when the value must be positive and the uncertainty is comparable to the best value. Conversion equations determined from the relations given in the sample_gamma() docs.

pwkit.msmt.**sample_double_norm**(*mean*, *std_upper*, *std_lower*, *size*)
Note that this function requires Scipy.

pwkit.msmt.**sample_gamma**(*alpha*, *beta*, *size*)
This is mostly about recording the conversion between Numpy/Scipy conventions and Wikipedia conventions. Some equations:

mean = alpha / beta variance = alpha / beta**2 mode = (alpha - 1) / beta [if alpha > 1; otherwise undefined] skewness = 2 / sqrt (alpha)

pwkit.msmt.**UQUANT_UNCERT = 0.2**
Some values are known to be uncertain, but their uncertainties have not been quantified. This is lame but it happens. In this case, assume a 20% uncertainty.

We could infer uncertainties from the number of written digits: i.e., assuming "1.2" is uncertain by 0.05 or so, while "1.2000" is uncertain by 0.00005 or so. But there are many cases in astronomy where people just list values that are 20% uncertain and give them to multiple decimal places. I'd rather be conservative with these values than overly optimistic.

Code to do the appropriate parsing is in the Python uncertainties package, in its __init__.py:parse_error_in_parentheses().

pwkit.msmt.**uval_dtype**
alias of `float64`

3.11 Period-finding with Phase Dispersion Minimization (`pwkit.pdm`)

pwkit.pdm - period-finding with phase dispersion minimization

As defined in Stellingwerf (1978ApJ...224..953S). See the update in Schwarzenberg-Czerny (1997ApJ...489..941S), however, which corrects the significance test formally; Linnell Nemec & Nemec (1985AJ.....90.2317L) provide a Monte Carlo approach. Also, Stellingwerf has developed "PDM2" which attempts to improve a few aspects; see

- Stellingwerf's page
- The Wikipedia article

class pwkit.pdm.**PDMResult**(*thetas*, *imin*, *pmin*, *mc_tmins*, *mc_pvalue*, *mc_pmins*, *mc_puncert*)

imin
Alias for field number 1

mc_pmins
Alias for field number 5

mc_puncert
Alias for field number 6

mc_pvalue
Alias for field number 4

mc_tmins
Alias for field number 3

pmin
Alias for field number 2

thetas
Alias for field number 0

pwkit.pdm.**pdm**(*t*, *x*, *u*, *periods*, *nbin*, *nshift=8*, *nsmc=256*, *numc=256*, *weights=False*, *parallel=True*)
Perform phase dispersion minimization.

t [1D array] time coordinate

x [1D array, same size as *t*] observed value

u [1D array, same size as *t*] uncertainty on observed value; same units as *x*

periods [1D array] set of candidate periods to sample; same units as *t*

nbin [int] number of phase bins to construct

nshift [int=8] number of shifted binnings to sample to combact statistical flukes

nsmc [int=256] number of Monte Carlo shufflings to compute, to evaluate the significance of the minimal theta value.

numc [int=256] number of Monte Carlo added-noise datasets to compute, to evaluate the uncertainty in the location of the minimal theta value.

weights [bool=False] if True, 'u' is actually weights, not uncertainties. Usually weights = u**-2.

parallel [default True] Controls parallelization of the algorithm. Default uses all available cores. See *pwkit.parallel.make_parallel_helper*.

Returns named tuple of:

thetas [1D array] values of theta statistic, same size as *periods*

imin index of smallest (best) value in *thetas*

pmin the *period* value with the smallest (best) *theta*

mc_tmins 1D array of size *nsmc* with Monte Carlo samplings of minimal theta values for shufflings of the data; assesses significance of the peak

mc_pvalue probability (between 0 and 1) of obtaining the best theta value in a randomly-shuffled dataset

mc_pmins 1D array of size *numc* with Monte Carlo samplings of best period values for noise-added data; assesses uncertainty of *pmin*

mc_puncert standard deviation of *mc_pmins*; approximate uncertainty on *pmin*.

We don't do anything clever, so runtime scales at least as `t.size * periods.size * nbin * nshift * (nsmc + numc + 1)`.

3.12 Loading the outputs of PHOENIX atmospheric models (`pwkit.phoenix`)

pwkit.phoenix - Working with Phoenix atmospheric models.

Functions:

- load_spectrum - Load a model spectrum into a Pandas DataFrame.

Requires Pandas.

Individual data files for the BT-Settl models are about 120 MB, and there are a million variations, so we do not consider bundling them with pwkit. Therefore, we can safely expect that the model will be accessible as a path on the filesystem.

Current BT-Settl models may be downloaded from a SPECTRA directory within the BT-Settl download site (see the README). E.g.:

```
http://phoenix.ens-lyon.fr/Grids/BT-Settl/CIFIST2011bc/SPECTRA/
```

File names are generally:

```
lte{Teff/100}-{Logg}{[M/H]}a[alpha/H].GRIDNAME.spec.7.[gz|bz2|xz]
```

The first three columns are wavelength in Å, log10(F_λ), and log10(B_λ), where the latter is the blackbody flux for the given Teff. The fluxes can nominally be converted into absolute units with an offset of 8 in log space, but I doubt that can be trusted much. Subsequent columns are related to various spectral lines. See http://phoenix.ens-lyon.fr/Grids/FORMAT .

The files do not come sorted!

`pwkit.phoenix.`**`load_spectrum`**(*path*, *smoothing=181*)

Load a Phoenix model atmosphere spectrum.

path [string] The file path to load.

smoothing [integer] Smoothing to apply. If None, do not smooth. If an integer, smooth with a Hamming window. Otherwise, the variable is assumed to be a different smoothing window, and the data will be convolved with it.

Returns a Pandas DataFrame containing the columns:

wlen Sample wavelength in Angstrom.

flam Flux density in erg/cm²/s/Å. See *pwkit.synphot* for related tools.

Loading takes about 5 seconds on my current laptop. Un-smoothed spectra have about 630,000 samples.

3.13 Flux density models of radio calibrators (pwkit.radio_cal_models)

pwkit.radio_cal_models - models of radio calibrator flux densities.

From the command line:

```
python -m pwkit.radio_cal_models [-f] <source> <freq[mhz]>
python -m pwkit.radio_cal_models [-f] CasA     <freq[mhz]> <year>
```

Print the flux density of the specified calibrator at the specified frequency, in Janskys.

Arguments:

<source> the source name (e.g., 3c348)

<freq> the observing frequency in MHz (e.g., 1420)

<year> is the decimal year of the observation (e.g., 2007.8). Only needed if <source> is CasA.

-f activates "flux" mode, where a three-item string is printed that can be passed to MIRIAD tasks that accept a model flux and spectral index argument.

pwkit.radio_cal_models.**cas_a**(*freq_mhz*, *year*)
Return the flux of Cas A given a frequency and the year of observation. Based on the formula given in Baars et al., 1977.

Parameters:

freq - Observation frequency in MHz. year - Year of observation. May be floating-point.

Returns: s, flux in Jy.

pwkit.radio_cal_models.**init_cas_a**(*year*)
Insert an entry for Cas A into the table of models. Need to specify the year of the observations to account for the time variation of Cas A's emission.

3.14 Synthetic photometry (pwkit.synphot)

pwkit.synphot - Synthetic photometry and database of instrumental bandpasses.

The basic structure is that we have a registry of bandpass info. You can use it to create Bandpass objects that can perform various calculations, especially the computation of synthetic photometry given a spectral model. Some key attributes of each bandpass are pre-computed so that certain operations can be done without needing to load the actual bandpass profile (though so far none of these profiles are very large at all).

Classes:

AlreadyDefinedError - Raised when re-registering bandpass info. Bandpass - Performs standard computations given a bandpass profile. NotDefinedError - Raised when needed bandpass info is unavailable. Registry - A registry of known bandpass profiles.

Functions:

get_std_registry - Retrieve a Registry pre-filled with builtin telescope info. (unlisted) - Various internal utilities may be useful for reference.

Variables:

builtin_registrars - Hashtable of functions to register the builtin telescopes.

3.14.1 Example

from pwkit import synphot as ps, cgs as pc, msmt as pm reg = ps.get_std_registry () print (reg.telescopes ()) # list known telescopes print (reg.bands ('2MASS')) # list known 2MASS bands bp = reg.get ('2MASS', 'Ks') mag = 12.83 mjy = pm.repval (bp.mag_to_fnu (mag) * pc.jypercgs * 1e3) print ('%.2f mag is %.2f mjy in 2MASS/Ks' % (mag, mjy))

3.14.2 Conventions

It is very important to maintain consistent conventions throughout.

Wavelengths are measured in angstroms. Flux densities are either per-wavelength (f_λ, "flam") or per-frequency (f_ν, "fnu"). These are measured in units of erg/s/cm²/Å and erg/s/cm²/Hz, respectively. Janskys can be converted to f_ν by multiplying by cgs.cgsperjy. f_ν's and f_λ's can be interconverted for a given filter if you know its "pivot wavelength". Some of the routines below show how to calculate this and do the conversion. "AB magnitudes" can be directly converted to Janskys and, thus, f_ν's.

Filter bandpasses can be expressed in two conventions: either "equal-energy" (EE) or "quantum-efficiency" (QE). The former gives the response per unit energy across the band, while the latter gives the response per photon. The EE convention can be integrated directly against a model spectrum, so we store all bandpasses internally in this convention. CCDs are photon-counting devices and so their response curves are generally expressed in the QE convention. Interconversion is easy: EE = QE * λ.

We don't expect any particular normalization of bandpass response curves.

The "width" of a bandpass is not a well-defined quantity, but is often needed for display purposes or approximate calculations. We use the locations of the half-maximum points (in the EE convention) to define the band edges.

This module requires Scipy and Pandas. It doesn't reeeeallllly need Pandas but it's convenient.

3.14.3 References

Casagrande & VandenBerg (2014; arxiv:1407.6095) has a lot of good stuff; see also references therein.

References for specific bandpasses are given in their implementation docstrings.

class `pwkit.synphot.`**`Bandpass`**

Computations regarding a particular filter bandpass.

Functions:

calc_halfmax_points - Calculate the wavelengths of the filter half-maximum values. calc_pivot_wavelength - Calculate the filter's pivot wavelength. halfmax_points - Get the filter half-maximum points (calculated if not cached). jy_to_flam - Convert Jy in this filter to a f_λ. mag_to_flam - Convert a magnitude in this filter to a f_λ. mag_to_fnu - Convert a magnitude in this filter to a f_ν. pivot_wavelength - Get the filter's pivot wavelength (calculated if not cached). synphot - Compute synthetic photometry given a model spectrum.

Attributes:

band - The name of this bandpass' associated band. native_flux_kind - Which kind of flux this bandpass is calibrated to: 'flam', 'fnu', or 'none'. registry - This object's parent Registry instance. telescope - The name of this bandpass' associated telescope.

The underlying bandpass shape is assumed to be sampled at discrete points. It is stored in _data and loaded on-demand. The object is a Pandas DataFrame containing at least the columns 'wlen' and 'resp'. The former holds the wavelengths of the sample points, in Ångström and in ascending order. The latter gives the response curve in the EE convention. No particular normalization is assumed. Other columns may be present but are not used generically.

calc_pivot_wavelength()
Compute and return the bandpass' pivot wavelength.

This value is computed directly from the bandpass data, not looked up in the Registry. Most of the values in the Registry were in fact derived from this function originally.

halfmax_points()
Get the bandpass' half-maximum wavelengths. These can be used to compute a representative bandwidth, or for display purposes.

Unlike calc_halfmax_points(), this function will use a cached value if available.

jy_to_flam(*jy*)
Convert a f_ν flux density measured in Janskys to a f_λ flux density.

This conversion is bandpass-dependent because it depends on the pivot wavelength of the bandpass used to measure the flux density.

mag_to_flam(*mag*)
Convert a magnitude in this band to a f_λ flux density.

It is assumed that the magnitude has been computed in the appropriate photometric system. The definition of "appropriate" will vary from case to case.

mag_to_fnu(*mag*)
Convert a magnitude in this band to a f_ν flux density.

It is assumed that the magnitude has been computed in the appropriate photometric system. The definition of "appropriate" will vary from case to case.

pivot_wavelength()
Get the bandpass' pivot wavelength.

Unlike calc_pivot_wavelength(), this function will use a cached value if available.

synphot(*wlen*, *flam*)
wlen and *flam* give a tabulated model spectrum in wavelength and f_λ units. We interpolate linearly over both the model and the bandpass since they're both discretely sampled.

Note that quadratic interpolation is both much slower and can blow up fatally in some cases. The latter issue might have to do with really large X values that aren't zero-centered, maybe?

I used to use the quadrature integrator, but Romberg doesn't issue complaints the way quadrature did. I should probably acquire some idea about what's going on under the hood.

class pwkit.synphot.**Registry**
A registry of known bandpass properties.

Methods:

bands - Return a list of bands associated with a telescope. get - Get a Bandpass object for a known telescope and filter. register_bpass - Register a Bandpass class. register_halfmaxes - Register precomputed half-max points. register_pivot_wavelength - Register precomputed pivot wavelengths. telescopes - Return a list of telescopes known to this registry.

bands(*telescope*)
Return a list of bands associated with the specified telescope.

telescopes()
Return a list of telescopes known to this registry.

pwkit.synphot.**get_std_registry**()
Get a Registry object pre-filled with information for standard telescopes.

3.15 Scaling relations for physical properties of ultra-cool dwarfs (pwkit.ucd_physics)

pwkit.ucd_physics - Physical calculations for (ultra)cool dwarfs.

These functions generally implement various nontrivial physical relations published in the literature. See docstrings for references.

Functions:

bcj_from_spt J-band bolometric correction from SpT.

bck_from_spt K-band bolometric correction from SpT.

load_bcah98_mass_radius Load Baraffe+ 1998 mass/radius data.

mass_from_j Mass from absolute J magnitude.

mk_radius_from_mass_bcah98 Radius from mass, using BCAH98 models.

tauc_from_mass Convective turnover time from mass.

pwkit.ucd_physics.**bcj_from_spt**(*spt*)
Calculate a bolometric correction constant for a J band magnitude based on a spectral type, using the fit of Wilking+ (1999AJ....117..469W).

spt - Numerical spectral type. M0=0, M9=9, L0=10, ...

Returns: the correction *bcj* such that *m_bol* = *j_abs* + *bcj*, or NaN if *spt* is out of range.

Valid values of *spt* are between 0 and 10.

pwkit.ucd_physics.**bck_from_spt**(*spt*)
Calculate a bolometric correction constant for a J band magnitude based on a spectral type, using the fits of Wilking+ (1999AJ....117..469W), Dahn+ (2002AJ....124.1170D), and Nakajima+ (2004ApJ...607..499N).

spt - Numerical spectral type. M0=0, M9=9, L0=10, ...

Returns: the correction *bck* such that *m_bol* = *k_abs* + *bck*, or NaN if *spt* is out of range.

Valid values of *spt* are between 2 and 30.

pwkit.ucd_physics.**load_bcah98_mass_radius**(*tablelines*, *metallicity=0*, *heliumfrac=0.275*, *age_gyr=5.0*, *age_tol=0.05*)
Load mass and radius from the main data table for the famous models of Baraffe+ (1998A&A...337..403B).

tablelines An iterable yielding lines from the table data file. I've named the file '1998A&A...337..403B_tbl1-3.dat' in some repositories (it's about 150K, not too bad).

metallicity The metallicity of the model to select.

heliumfrac The helium fraction of the model to select.

age_gyr The age of the model to select, in Gyr.

age_tol The tolerance on the matched age, in Gyr.

Returns: (mass, radius), where both are Numpy arrays.

The ages in the data table vary slightly at fixed metallicity and helium fraction. Therefore, there needs to be a tolerance parameter for matching the age.

pwkit.ucd_physics.**mass_from_j**(*j_abs*)
Estimate mass in cgs from absolute J magnitude, using the relationship of Delfosse+ (2000A&A...364..217D).

j_abs - The absolute J magnitude.

Returns: the estimated mass in grams.

If j_abs > 11, a fixed result of 0.1 Msun is returned. Values of j_abs < 5.5 are illegal and get NaN. There is a discontinuity in the relation at j_abs = 11, which yields 0.0824 Msun.

pwkit.ucd_physics.**mk_radius_from_mass_bcah98**(**args*, ***kwargs*)

Create a function that maps (sub)stellar mass to radius, based on the famous models of Baraffe+ (1998A&A...337..403B).

tablelines An iterable yielding lines from the table data file. I've named the file '1998A&A...337..403B_tbl1-3.dat' in some repositories (it's about 150K, not too bad).

metallicity The metallicity of the model to select.

heliumfrac The helium fraction of the model to select.

age_gyr The age of the model to select, in Gyr.

age_tol The tolerance on the matched age, in Gyr.

Returns: a function mtor(mass_g), return a radius in cm as a function of a mass in grams. The mass must be between 0.05 and 0.7 Msun.

The ages in the data table vary slightly at fixed metallicity and helium fraction. Therefore, there needs to be a tolerance parameter for matching the age.

This function requires Scipy.

pwkit.ucd_physics.**tauc_from_mass**(*mass_g*)

Estimate the convective turnover time from mass, using the method described in Cook+ (2014ApJ...785...10C).

mass_g - UCD mass in grams.

Returns: the convective turnover timescale in seconds.

Masses larger than 1.3 Msun are out of range and yield NaN. If the mass is <0.1 Msun, the turnover time is fixed at 70 days.

The Cook method was inspired by the description in McLean+ (2012ApJ...746...23M). It is a hybrid of the method described in Reiners & Basri (2010ApJ...710..924R) and the data shown in Kiraga & Stepien (2007AcA....57..149K). However, this version imposes the 70-day cutoff in terms of mass, not spectral type, so that it is entirely defined in terms of a single quantity.

There are discontinuities between the different break points! Any future use should tweak the coefficients to make everything smooth.

CHAPTER 4

Command-line tools

This documentation has a lot of stubs.

4.1 Quick astronomical calculations (`astrotool`)

pwkit.cli.astrotool - the 'astrotool' program.

4.2 Quick operations on astronomical images (`pwkit.cli.imtool`)

pwkit.cli.imtool - the 'imtool' program.

4.3 Single-command compilation of LaTeX documents (`latexdriver`)

pwkit.cli.latexdriver - the 'latexdriver' program.

This used to be a nice little shell script, but for portability it's better to do this in Python.

4.4 Wrap the output of a sub-program with extra information (`wrapout`)

pwkit.cli.wrapout - the 'wrapout' program.

CHAPTER 5

Data Visualization

This documentation has a lot of stubs.

5.1 Mapping arbitrary data to color scales (`pwkit.colormaps`)

pwkit.colormaps – tools to conver arrays of real-valued data to other formats (usually, RGB24) for visualization.

TODO: "heated body" map.

The main interface is the *factory_map* dictionary from colormap names to factory functions. *base_factory_names* lists the names of a set of color maps. Additional ones are available with the suffixes "_reverse" and "_sqrt" that apply the relevant transforms.

The factory functions return another function, the "mapper". Each mapper takes a single argument, an array of values between 0 and 1, and returns the mapped colors. If the input array has shape S, the returned value has a shape (S + (3,)), with mapped[...,0] being the R values, between 0 and 1, etc.

Example:

> data = np.array ([<things between 0 and 1>]) mapper = factory_map['cubehelix_blue']() rgb = mapper (data) green_values = rgb[:,1] last_rgb = rgb[-1]

The basic colormap names are:

moreland_bluered Divergent colormap from intense blue (at 0) to intense red (at 1), passing through white

cubehelix_dagreen From black to white through rainbow colors

cubehelix_blue From black to white, with blue hues

pkgw From black to red, through purplish

black_to_white, black_to_red, black_to_green, black_to_blue From black to the named colors.

white_to_black, white_to_red, white_to_green, white_to_blue From white to the named colors.

The mappers can also take keyword arguments, including at least "transform", which specifies simple transforms that can be applied to the colormaps. These are (in terms of symbolic constants and literal string values):

> 'none' - No transform (the default) 'reverse' - x -> 1 - x (reverses the colormap) 'sqrt' - x -> sqrt (x)

For each transform other than "none", *factory_map* contains an entry with an underscore and the transform name applied (e.g., "pkgw_reverse") that has that transform applied.

The initial inspiration was an implementation of the ideas in "Diverging Color Maps for Scientific Visualization (Expanded)", Kenneth Moreland,

http://www.cs.unm.edu/~kmorel/documents/ColorMaps/index.html

I've realized that I'm not too fond of the white mid-values in these color maps in many cases. So I also added an implementation of the "cube helix" color map, described by D. A. Green in

"A colour scheme for the display of astronomical intensity images" http://adsabs.harvard.edu/abs/2011BASI...39..289G (D. A. Green, 2011 Bull. Ast. Soc. of India, 39 289)

I made up the pkgw map myself (who'd have guessed?).

5.2 Tracing contours (`pwkit.contours`)

pwkit.contours - Tracing contours in functions and data.

Uses my own homebrew algorithm. So far, it's only tested on extremely well-behaved functions, so probably doesn't cope well with poorly-behaved ones.

`pwkit.contours.`**`analytic_2d`**(*f*, *df*, *x0*, *y0*, *maxiters=5000*, *defeta=0.05*, *netastep=12*, *vtol1=0.001*, *vtol2=1e-08*, *maxnewt=20*, *dorder=7*, *goright=False*)

Sample a contour in a 2D analytic function. Arguments:

f A function, mapping (x, y) -> z.

df The partial derivative: df (x, y) -> [dz/dx, dz/dy]. If None, the derivative of f is approximated numerically with scipy.derivative.

x0 Initial x value. Should be of "typical" size for the problem; avoid 0.

y0 Initial y value. Should be of "typical" size for the problem; avoid 0.

Optional arguments:

maxiters Maximum number of points to create. Default 5000.

defeta Initially offset by distances of defeta*[df/dx, df/dy] Default 0.05.

netastep Number of steps between defeta and the machine resolution in which we test eta values for goodness. (OMG FIXME doc). Default 12.

vtol1 Tolerance for constancy in the value of the function in the initial offset step. The value is only allowed to vary by `f(x0,y0) * vtol1`. Default 1e-3.

vtol2 Tolerance for constancy in the value of the function in the along the contour. The value is only allowed to vary by `f(x0,y0) * vtol2`. Default 1e-8.

maxnewt Maximum number of Newton's method steps to take when attempting to hone in on the desired function value. Default 20.

dorder Number of function evaluations to perform when evaluating the derivative of f numerically. Must be an odd integer greater than 1. Default 7.

goright If True, trace the contour rightward (as looking uphill), rather than leftward (the default).

5.3 Utilities for data visualization (`pwkit.data_gui_helpers`)

pwkit.data_gui_helpers - helpers for GUIs looking at data arrays

Classes:

Clipper - Map data into [0,1] ColorMapper - Map data onto RGB colors using *pwkit.colormaps* Stretcher - Map data within [0,1] using a stretch like sqrt, etc.

Functions:

data_to_argb32 - Turn arbitrary data values into ARGB32 colors. data_to_imagesurface - Turn arbitrary data values into a Cairo ImageSurface.

pwkit.data_gui_helpers.**data_to_argb32**(*data*, *cmin=None*, *cmax=None*, *stretch=u'linear'*, *cmap=u'black_to_blue'*)

Turn arbitrary data values into ARGB32 colors.

There are three steps to this process: clipping the data values to a maximum and minimum; stretching the spacing between those values; and converting their amplitudes into colors with some kind of color map.

***data* - Input data; can (and should) be a MaskedArray if some values are** invalid.

***cmin* - The data clip minimum; all values <= cmin are treated** identically. If None (the default), *data.min ()* is used.

***cmax* - The data clip maximum; all values >= cmax are treated** identically. If None (the default), *data.max ()* is used.

***stretch* - The stretch function name; 'linear', 'sqrt', or 'square'; see** the Stretcher class.

***cmap* - The color map name; defaults to 'black_to_blue'. See the** *pwkit.colormaps* module for more choices.

Returns a Numpy array of the same shape as *data* with dtype *np.uint32*, which represents the ARGB32 colorized version of the data. If your colormap is restricted to a single R or G or B channel, you can make color images by bitwise-or'ing together different such arrays.

class pwkit.data_gui_helpers.**Stretcher**(*mode*)

Assumes that its inputs are in [0, 1]. Maps its outputs to the same range.

5.4 Easy visualization of matrices with GTK+ version 2 (pwkit.ndshow_gtk2)

pwkit.ndshow_gtk2 - visualize data arrays with the Gtk+2 toolkit.

Functions:

view - Show a GUI visualizing a 2D array. cycle - Show a GUI cycling through planes of a 3D array.

Classes:

Viewport - A GtkDrawingArea that renders a portion of an array. Viewer - A GUI window for visualizing a 2D array. Cycler - A GUI window for cycling through planes of a 3D array.

UI features of the viewport:

- click-drag to pan
- scrollwheel to zoom in/out (Ctrl to do so more aggressively)
- (Shift to change color scale adjustment sensitivity)
- double-click to recenter
- shift-click-drag to adjust color scale (prototype)

Added by the toplevel window viewer:

- Ctrl-A to autoscale data to fit window
- Ctrl-E to center the data in the window

- Ctrl-F to fullscreen the window
- Escape to un-fullscreen it
- Ctrl-W to close the window
- Ctrl-1 to set scale to unity
- Ctrl-S to save the data to "data.png" under the current rendering options (but not zoomed to the current view of the data).

Added by cycler:

- Ctrl-K to move to next plane
- Ctrl-J to move to previous plane
- Ctrl-C to toggle automatic cycling

5.5 Easy visualization of matrices with GTK+ version 3 (`pwkit.ndshow_gtk3`)

pwkit.ndshow_gtk3 - visualize data arrays with the Gtk+3 toolkit.

Functions:

view - Show a GUI visualizing a 2D array. cycle - Show a GUI cycling through planes of a 3D array.

Classes:

Viewport - A GtkDrawingArea that renders a portion of an array. Viewer - A GUI window for visualizing a 2D array. Cycler - A GUI window for cycling through planes of a 3D array.

UI features of the viewport:

- click-drag to pan
- scrollwheel to zoom in/out (Ctrl to do so more aggressively) (Shift to change color scale adjustment sensitivity)
- double-click to recenter
- shift-click-drag to adjust color scale (prototype)

Added by the toplevel window viewer:

- Ctrl-A to autoscale data to fit window
- Ctrl-E to center the data in the window
- Ctrl-F to fullscreen the window
- Escape to un-fullscreen it
- Ctrl-W to close the window
- Ctrl-1 to set scale to unity
- Ctrl-S to save the data to "data.png" under the current rendering options (but not zoomed to the current view of the data).

Added by cycler:

- Ctrl-K to move to next plane
- Ctrl-J to move to previous plane

- Ctrl-C to toggle automatic cycling

CHAPTER 6

Data input and output

This documentation has a lot of stubs.

6.1 Streaming output from other programs (`pwkit.slurp`)

The *`pwkit.slurp`* module makes it convenient to read output generated by other programs. This is accomplished with a context-manager class known as *`Slurper`*, which is built on top of the standard `subprocess.Popen` module.

The chief advantage of *`Slurper`* above `subprocess.Popen` is that it provides convenient, *streaming* access to subprogram output, maintaining the distinction between "stdout" (standard output, written to file descriptor #1) and "stderr" (standard error, written to file descriptor #2). It can also forward signals to the child program.

Standard usage might look like:

```
from pwkit import slurp
argv = ['cat', '/etc/passwd']

with slurp.Slurper (argv, linebreak=True) as slurper:
    for etype, data in slurper:
        if etype == 'stdout':
            print ('got line:', data)

print ('exit code was:', slurper.proc.returncode)
```

`Slurper` is a context manager to ensure that the child process is always cleaned up. Within the context manager body, you should iterate over the *`Slurper`* instance to get a series of "event" 2-tuples consisting of a Unicode string giving the event type, and the event data. Most, but not all, events have to do with receiving data over the stdout or stderr pipes. The events are:

Event type	Event data	Description
`"stdout"`	The output	Data were received from the subprogram's standard output.
`"stderr"`	The output	Data were received from the subprogram's standard error.
`"forwarded-signal"`	The signal number	This process received a signal and forwarded it to the child.
`"timeout"`	`None`	No data were received from the child within a fixed timeout.

The data provided on the `"stdout"` and `"stderr"` events follow the usual Python patterns for EOF. Namely, when either of those pipes is closed by the subprocess, a final event is sent in which the data payload has zero length. (It may be either a bytes object or a Unicode string depending on whether decoding is enabled; see below.)

Warning: It is important to realize that programs that use the standard C I/O routines, such as Python programs, buffer their output by default. The *pwkit.slurp* module may appear to be having problems while really the child program is batching up its output and writing it all at once. This can be surprising because the default behavior is line-buffered when `stdout` is connected to a TTY (as when you run programs in your terminal), but buffered in large blocks when connected to a pipe (as when using this module). On systems built on `glibc`, you can control this by using the `stdbuf` program to launch your subprogram with different buffering options. To run the command `foo bar` with both stdout and stderr buffered at the line level, run `stdbuf -oL -eL foo bar`. To disable buffering on both streams, run `stdbuf -o0 -e0 foo bar`.

class pwkit.slurp.**Slurper**(*argv=None*, *env=None*, *cwd=None*, *propagate_signals=True*, *timeout=10*, *linebreak=False*, *encoding=None*, *stdin=slurp.Redirection.DevNull*, *stdout=slurp.Redirection.Pipe*, *stderr=slurp.Redirection.Pipe*, *executable=None*)

Construct a context manager used to read output from a subprogram. *argv* is used to launch the subprogram using `subprocess.Popen` with the *shell* keyword set to False. *env*, *cwd*, *executable*, *stdin*, *stdout*, and *stderr* are forwarded to the `subprocess.Popen` constructor as well.

Regarding the redirection parameters *stdin*, *stdout*, and *stderr*, the constants in the *Redirection* object gives more user-friendly names to the analogues provided by the `subprocess` module, with the addition of a `Redirection.DevNull` option emulating behavior added in Python 3. Otherwise these values are passed to `subprocess.Popen` verbatim, so you can use anyting that `subprocess.Popen` would accept. Keep in mind that you can only fetch the subprogram's output if one or both of the output paramers are set to `Redirection.Pipe`!

If *propagate_signals* is true, signals received by the parent process will be forwarded to the child process. This can be valuable to obtain correct behavior on SIGINT, for instance. Forwarded signals are SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGUSR1, and SIGUSR2. This is done by overwriting the calling process' Python signal handlers; the original handlers are restored upon exit from the with-statement block.

If *linebreak* is true, output from the child process will be gathered into whole lines (split by `"\n"`) before being sent to the caller. *The newline characters will be discarded*, making it impossible to tell whether the final line of output ended with a newline or not.

If *encoding* is not `None`, a decoder will be created with `codecs.getincrementaldecoder()` and the subprocess output will be converted from bytes to Unicode before being returned to the calling process.

timeout sets the timeout for the internal `select.select()` call used to check for output from the subprogram. It is measured in seconds.

Slurper instances have attributes *argv*, *env*, *cwd*, *executable*, *propagate_signals*, :*timeout*, *linebreak*, attr:*encoding*, *stdin*, :*stdout*, and *stderr* recording the construction parameters.

pwkit.slurp.**Redirection**

An enum-like object defining ways to redirect the I/O streams of the subprogram. These values are identical to those used in `subprocess` but with nicer names.

Constant	Meaning
`Redirection.Pipe`	Pipe output to the calling program.
`Redirection.Stdout`	Only valid for `stderr`; merge it with `stdout`
`Redirection.DevNull`	Direct input from `/dev/null`, or output thereto.

The whole *raison d'être* of *pwkit.slurp* is to make it easy to communicate output between programs, so you probably will probably want to use `Redirection.Pipe` for `stdout` and `stderr` most of the time.

6.1.1 `Slurper` reference

`Slurper.`**`proc`**
The `subprocess.Popen` instance of the child program. After the program has exited, you can access its exit code as `Slurper.proc.returncode`.

`Slurper.`**`argv`**
The `argv` of the program to be launched.

`Slurper.`**`env`**
Environment dictionary for the program to be launched.

`Slurper.`**`cwd`**
The working directory for the program to be launched.

`Slurper.`**`executable`**
The name of the executable to launch (`argv[0]` is allowed to differ from this).

`Slurper.`**`propagate_signals`**
Whether to forward the subprogram any signals that are received by the calling process.

`Slurper.`**`timeout`**
The timeout (in seconds) for waiting for output from the child program. If nothing is received, a `"timeout"` event is generated.

`Slurper.`**`linebreak`**
Whether to gather the subprogram output into textual lines.

`Slurper.`**`encoding`**
The encoding to be used to decode the subprogram output from bytes to Unicode, or `None` if no such decoding is to be done.

`Slurper.`**`stdin`**
How to redirect the standard input of the subprogram, if at all.

`Slurper.`**`stdout`**
How to redirect the standard output of the subprogram, if at all. If not `Pipe`, no `"stdout"` events will be received.

`Slurper.`**`stderr`**
How to redirect the standard error of the subprogram, if at all. If not `Pipe`, no `"stderr"` events will be received. If `Stdout`, events that would have had a type of `"stderr"` will have a type of `"stdout"` instead.

6.2 A simple "ini" file format (`pwkit.inifile`)

A simple parser for ini-style files that's better than Python's ConfigParser/configparser.

Functions:

read Generate a stream of *pwkit.Holder* instances from an ini-format file.

mutate Rewrite an ini file chunk by chunk.

write Write a stream of *pwkit.Holder* instances to an ini-format file.

mutate_stream Lower-level version; only operates on streams, not path names.

read_stream Lower-level version; only operates on streams, not path names.

write_stream Lower-level version; only operates on streams, not path names.

mutate_in_place Rewrite an ini file specififed by its path name, in place.

pwkit.inifile.**mutate_stream**(*instream*, *outstream*)
Python 3 compat note: we're assuming *stream* gives bytes not unicode.

pwkit.inifile.**read_stream**(*stream*)
Python 3 compat note: we're assuming *stream* gives bytes not unicode.

pwkit.inifile.**write_stream**(*stream*, *holders*, *defaultsection=None*)
Very simple writing in ini format. The simple stringification of each value in each Holder is printed, and no escaping is performed. (This is most relevant for multiline values or ones containing pound signs.) *None* values are skipped.

Arguments:

stream A text stream to write to.

holders An iterable of objects to write. Their fields will be written as sections.

defaultsection=None Section name to use if a holder doesn't contain a *section* field.

pwkit.inifile.**write**(*stream_or_path*, *holders*, ***kwargs*)
Very simple writing in ini format. The simple stringification of each value in each Holder is printed, and no escaping is performed. (This is most relevant for multiline values or ones containing pound signs.) *None* values are skipped.

Arguments:

stream A text stream to write to.

holders An iterable of objects to write. Their fields will be written as sections.

defaultsection=None Section name to use if a holder doesn't contain a *section* field.

6.3 Outputting data in LaTeX format (pwkit.latex)

pwkit.latex - various helpers for the LaTeX typesetting system.

6.3.1 Classes

Referencer Accumulate a numbered list of bibtex references, then output them.

TableBuilder Create awesome deluxetables programmatically.

6.3.2 Functions

latexify_l3col Format value in LaTeX, suitable for tables of limit values.

latexify_n2col Format a number in LaTeX in 2-column decimal-aligned formed.

latexify_u3col Format value in LaTeX, suitable for tables of uncertain values.

latexify Format a value in LaTeX appropriately.

6.3.3 Helpers for TableBuilder

AlignedNumberFormatter Format numbers, aligning them at the decimal point.

BasicFormatter Base class for formatters.

BoolFormatter Format a boolean; default is True -> bullet, False -> nothing.

LimitFormatter Format measurements for a table of limits.

MaybeNumberFormatter Format numbers with a fixed number of decimal places, or objects with __pk_latex__().

UncertFormatter Format measurements for a table of detailed uncertainties.

WideHeader Helper for multi-column headers.

XXX: Barely tested!

class pwkit.latex.**AlignedNumberFormatter**(*nplaces=1*)
Format numbers. Allows the number of decimal places to be specified, and aligns the numbers at the decimal point.

class pwkit.latex.**BasicFormatter**
Base class for formatting table cells in a TableBuilder.

Generally a formatter will also provide methods for turning input data into fancified LaTeX output that can be used by the column's "data function".

colinfo(*builder*)
Return (nlcol, colspec, headprefix), where:

nlcol - The number of LaTeX columns encompassed by this logical column.

colspec - Its LaTeX column specification (None to force user to specify).

headprefix - Prefix applied before heading items in {} (e.g., "colhead").

class pwkit.latex.**BoolFormatter**
Format booleans. Attributes *truetext* and *falsetext* set what shows up for true and false values, respectively.

class pwkit.latex.**LimitFormatter**
Format measurements (cf pwkit.msmt) with nice-looking limit information. Specific uncertainty information is discarded. The default formats do not involve fancy subscripts or superscripts, so row struts are not needed ... by default.

class pwkit.latex.**MaybeNumberFormatter**(*nplaces=1*, *align='c'*)
Format Python objects. If it's a number, format it as such, without any fancy column alignment, but with a specifiable number of decimal places. Otherwise, call latexify() on it.

class pwkit.latex.**Referencer**
Accumulate a numbered list of bibtex references. Methods:

refkey (bibkey) Return a string that should be used to give a numbered reference to the given bibtex key. "thiswork" is handled specially.

dump () Return a string with citet{} commands identifing all of the numbered references.

Attributes:

thisworktext text referring to "this work"; defaults to that.

thisworkmarker special symbol used to denote "this work"; defaults to star.

Bibtex keys beginning with asterisks have the rest of their value used for the citation text, rather than "citet{<key>}".

class pwkit.latex.**TableBuilder**(*label*)
Build and then emit a nice deluxetable.

Methods:

addcol (headings, datafunc, formatter=None, colspec=None, numbering='(%d)') Define a logical column.

addnote (key, text) Define a table note that can appear in cells.

addhcline (headerrowix, logcolidx, latexdeltastart, latexdeltaend) Add a horizontal line between columns.

notemark (key) Return a tablenotemark{} command for the specified note key.

emit (stream, items) Write the table, with one row for each thing in *items*, to the stream.

If an item has an attribute *tb_row_preamble*, that text is written verbatim before that corresponding row is output.

Attributes:

environment The name of the latex environment to use, default "deluxetable". You may want to specify "deluxetable*", or "mydeluxetable" if using a hacked package.

label The latex reference label of the table. Mandatory.

note A note at the table footer ("tablecomments{}" in LaTeX).

preamble Commands for table preamble. See below.

refs Contents of the table References section.

title Table title. Default "Untitled table".

widthspec Passed to tablewidth{}; default "0em" = auto-widen.

numbercols If True, number each column. This can be disabled on a col-by-col basis by calling *addcol* with *numbering* set to False.

Legal preamble commands are:

```
\rotate
\tablenum{<manual table identifier>}
\tabletypesize{<font size command>}
```

The commands tablecaption, tablecolumns, tablehead, and tablewidth are handled specially.

If tablewidth{} is not provided, the table is set at full width, not its natural width, which is a lame default. The default *widthspec* lets us auto-widen while providing a clear avenue to customizing the width.

addcol (*headings*, *datafunc*, *formatter=None*, *colspec=None*, *numbering=u'(%d)'*)
Define a logical column. Arguments:

headings A string, or list of strings and WideHeaders. The headings are stacked vertically in the table header section.

datafunc Return LaTeX for this cell. Call spec should be (item, [formatter, [tablebuilder]]).

formatter The formatter to use; defaults to a new BasicFormatter.

colspec The LaTeX column specification letters to use; defaults to 'c's.

numbering If non-False, a format for writing this column's number; if False, no number is written.

addhcline (*headerrowidx*, *logcolidx*, *latexdeltastart*, *latexdeltaend*)
Adds a horizontal line below a limited range of columns in the header section. Arguments:

headerrowidx - The 0-based row number *below* which the line will be drawn; i.e. 0 means that the line will be drawn below the first row of header cells.

logcolidx - The 0-based 'logical' column number relative to which the line will be placed; i.e. 1 means that the line placement will be relative to the second column defined in an addcol() call.

latexdeltastart - The relative position at which to start drawing the line relative to that logical column, in LaTeX columns; typically going to be zero.

latexdeltaend - The relative position at which to finish drawing the line, in the standard Python non-inclusive sense. I.e., if you want to underline two LaTeX columns, latexdeltaend = latexdeltastart + 2.

class pwkit.latex.**UncertFormatter**
Format measurements (cf. pwkit.msmt) with detailed uncertainty information, possibly including asymmetric uncertainties. Because of the latter possibility, table rows have to be made extra-high to maintain evenness.

class pwkit.latex.**WideHeader**(*nlogcols*, *content*, *align='c'*)
Information needed for constructing wide table headers.

nlogcols - Number of logical columns consumed by this header. content - The LaTeX to insert for this header's content. align - The alignment of this header; default 'c'.

Rendered as multicolumn{nlatex}{align}{content}, where *nlatex* is the number of LaTeX columns spanned by this header – which may be larger than *nlogcols* if certain logical columns span multiple LaTeX columns.

pwkit.latex.**latexify_l3col**(*obj*, ***kwargs*)
Convert an object to special LaTeX for limit tables.

This conversion is meant for limit values in a table. The return value should span three columns. The first column is the limit indicator: <, >, ~, etc. The second column is the whole part of the value, up until just before the decimal point. The third column is the decimal point and the fractional part of the value, if present. If the item being formatted does not fit this schema, it can be wrapped in something like 'multicolumn{3}{c}{...}'.

pwkit.latex.**latexify_n2col**(*x*, *nplaces=None*, ***kwargs*)
Render a number into LaTeX in a 2-column format, where the columns split immediately to the left of the decimal point. This gives nice alignment of numbers in a table.

pwkit.latex.**latexify_u3col**(*obj*, ***kwargs*)
Convert an object to special LaTeX for uncertainty tables.

This conversion is meant for uncertain values in a table. The return value should span three columns. The first column ends just before the decimal point in the main number value, if it has one. It has no separation from the second column. The second column goes from the decimal point until just before the "plus-or-minus" indicator. The third column goes from the "plus-or-minus" until the end. If the item being formatted does not fit this schema, it can be wrapped in something like 'multicolumn{3}{c}{...}'.

pwkit.latex.**latexify**(*obj*, ***kwargs*)
Render an object in LaTeX appropriately.

6.4 Reading and writing data tables with types and uncertainties (pwkit.tabfile)

pwkit.tabfile - I/O with typed tables of uncertain measurements.

Functions:

read - Read a typed table file. vizread - Read a headerless table file, with columns specified separately write - Write a typed table file.

The table format is line-oriented text. Hashes denote comments. Initial lines of the form "colname = value" set a column name that gets the same value for every item in the table. The header line is prefixed with an @ sign. Subsequent lines are data rows.

pwkit.tabfile.**read**(*path*, *tabwidth=8*, ***kwargs*)
Read a typed tabular text file into a stream of Holders.

Arguments:

path The path of the file to read.

tabwidth=8 The tab width to assume. Please don't monkey with it.

mode='rt' The file open mode (passed to io.open()).

noexistok=False If True and the file is missing, treat it as empty.

****kwargs** Passed to io.open ().

Returns a generator for a stream of *pwkit.Holder's, each of which will contain ints, strings, or some kind of measurement (cf 'pwkit.msmt*).

pwkit.tabfile.**vizread**(*descpath*, *descsection*, *tabpath*, *tabwidth=8*, ***kwargs*)
Read a headerless tabular text file into a stream of Holders.

Arguments:

descpath The path of the table description ini file.

descsection The section in the description file to use.

tabpath The path to the actual table data.

tabwidth=8 The tab width to assume. Please don't monkey with it.

mode='rt' The table file open mode (passed to io.open()).

noexistok=False If True and the file is missing, treat it as empty.

****kwargs** Passed to io.open ().

Returns a generator of a stream of *pwkit.Holder's, each of which will contain ints, strings, or some kind of measurement (cf 'pwkit.msmt*). In this version, the table file does not contain a header, as seen in Vizier data files. The corresponding section in the description ini file has keys of the form "colname = <start> <end> [type]", where <start> and <end> are the **1-based** character numbers defining the column, and [type] is an optional specified of the measurement type of the column (one of the usual b, i, f, u, Lu, Pu).

pwkit.tabfile.**write**(*stream*, *items*, *fieldnames*, *tabwidth=8*)
Write a typed tabular text file to the specified stream.

Arguments:

stream The destination stream.

items An iterable of items to write. Two passes have to be made over the items (to discover the needed column widths), so this will be saved into a list.

fieldnames Either a list of field name strings, or a single string. If the latter, it will be split into a list with .split().

tabwidth=8 The tab width to use. Please don't monkey with it.

Returns nothing.

6.5 An "ini" file format with typed, uncertain data (pwkit.tinifile)

pwkit.tinifile - Dealing with typed ini-format files full of measurements.

Functions:

read Generate *pwkit.Holder* instances of measurements from an ini-format file.

write Write *pwkit.Holder* instances of measurements to an ini-format file.

read_stream Lower-level version; only operates on streams, not path names.

write_stream Lower-level version; only operates on streams, not path names.

pwkit.tinifile.**write_stream**(*stream*, *holders*, *defaultsection=None*, *extrapos=()*, *sha1sum=False*, ***kwargs*)

extrapos is basically a hack for multi-step processing. We have some flux measurements that are computed from luminosities and distances. The flux value is therefore an unwrapped Uval, which doesn't retain memory of any positivity constraint it may have had. Therefore, if we write out such a value using this routine, we may get something like *fx:u = 1pm1*, and the next time it's read in we'll get negative fluxes. Fields listed in *extrapos* will have a "P" constraint added if they are imprecise and their typetag is just "f" or "u".

6.6 Converting Unicode to LaTeX notation (pwkit.unicode_to_latex)

unicode_to_latex - what it says

Provides unicode_to_latex(u) and unicode_to_latex_string(u).

unicode_to_latex returns ASCII bytes that can be fed to LaTeX to reproduce the Unicode string 'u' as closely as possible.

unicode_to_latex_string returns a Unicode string rather than bytes. That is:

```
unicode_to_latex(u) = unicode_to_latex_string(u).encode('ascii').
```

CHAPTER 7

External Software Environments

This documentation has a lot of stubs.

7.1 CASA (`pwkit.environments.casa`)

casa - running software in the CASA environment

To use, export an environment variable $PWKIT_CASA pointing to the CASA installation root. The files $PWKIT_CASA/asdm2MS and $PWKIT_CASA/casapy should exist.

XXX untested with 32-bit, probably won't work. XXX test only on Linux, probably needs work for Macs.

7.1.1 CASA installation notes

Download tarball as linked from http://casa.nrao.edu/casa_obtaining.shtml . Tarball unpacks to some versioned subdirectory. The names and version codes are highly variable and annoying.

7.2 Compact-source photometry with discrete Fourier transformations (`pwkit.environments.casa.dftphotom`)

pwkit.environments.casa.dftphotom - point-source photometry from visibilities

CASA doesn't yet have a task to do this.

7.3 Structured scripting within `casapy` (`pwkit.environments.casa.scripti`

pwkit.environments.casa.scripting - scripted invocation of casapy.

The "casapy" program is extremely resistant to encapsulated scripting – it pops up GUI windows and child processes, leaves log files around, provides a non-vanilla Python environment, and so on. However, sometimes scripting CASA is what we need to do. This tool enables that.

We provide a single-purpose CLI tool for this functionality, so that you can write standalone scripts with a hashbang line of "#! /usr/bin/env pkcasascript" – hashbang lines support only one extra command-line argument, so if we're using "env" we can't take a multitool approach.

class pwkit.environments.casa.scripting.**CasapyScript**(*script*, *raise_on_error=True*, ***kwargs*)

Context manager for launching a script in the casapy environment. This involves creating a temporary wrapper and then using the CasaEnvironment to run it in a temporary directory.

When this context manager is entered, the script is launched and the calling process waits until it finishes. This object is returned. The *with* statement body is then executed so that information can be extracted from the results of the casapy invocation. When the context manager is exited, the casapy files are (usually) cleaned up.

Attributes:

args the arguments to passed to the script.

env the CasaEnvironment used to launch the casapy process.

exitcode the exit code of the casapy process. 0 is success. 127 indicates an intentional error exit by the script; additional diagnostics don't need printing and the work directory doesn't need preservation. Negative values indicate death from a signal.

proc the *subprocess.Popen* instance of casapy; inside the context manager body it's already exited.

rmtree boolean; whether to delete the working tree upon context manager exit.

script the path to the script to be invoked.

workdir the working directory in which casapy was started.

wrapped the path to the wrapper script run inside casapy.

There is a very large overhead to running casapy scripts. The outer Python code sleeps for at least 5 seconds to allow various cleanups to happen.

7.4 Merging spectral windows in visibility data (pwkit.environments.casa.spwglue)

pwkit.environments.casa.spwglue - merge spectral windows in a MeasurementSet

I find that merging windows in this way offers a lot of advantages. This procesing step is very slow, however.

class pwkit.environments.casa.spwglue.**Progress**

This could be split out; it's useful.

7.5 Quick access to basic CASA tasks (pwkit.environments.casa.tasks)

pwkit.environments.casa.tasks - library of clones of CASA tasks

The way that the casapy code is written it's basically impossible to import its tasks into a straight-Python environment (trust me, I've tried), so we're more-or-less duplicating lots of CASA code. I try to provide saner semantics, APIs, etc.

The goal is to make task-like functionality as a real Python library with no side effects, so that we can actually script data processing. While we're at it, we make them available on the command line.

pwkit.environments.casa.tasks.**listobs**(*vis*)

Generates a set of lines of output. Errors are only detected by looking at the output.

7.6 Utilities for Python invocation of CASA tools (`pwkit.environments.casa.util`)

pwkit.environments.casa.util - core utilities for the CASA Python libraries

Variables are:

INVERSE_C_SM Inverse of C in s/m (useful for wavelength to time conversion)

INVERSE_C_NSM Inverse of C in ns/m (ditto).

pol_names Dict mapping CASA polarization codes to their string names.

pol_to_miriad Dict mapping CASA polarization codes to their MIRIAD equivalents.

msselect_keys A set of the keys supported by the CASA ms-select subsystem.

tools An object for constructing CASA tools: `ia = tools.image ()`.

Functions are:

datadir Return the CASA data directory.

logger Create a CASA logger that prints to stderr without leaving a casapy.log file around.

forkandlog Run a function in a subprocess, returning the text it outputs via the CASA logging subsystem.

sanitize_unicode Encode Unicode strings as bytes for interfacing with casac functions.

`pwkit.environments.casa.util.`**`sanitize_unicode`**(*item*)

The Python bindings to CASA tasks expect to receive all string values as binary data (Python 2.X "str" or 3.X "bytes") and not Unicode (Python 2.X "unicode" or 3.X "str"). To prep for Python 3 (not that CASA will ever be compatible with it ...) I true to use the unicode_literals everywhere, and other Python modules are getting better about using Unicode consistently, so this causes problems. This helper converts Unicode into UTF-8 encoded bytes, handling the common data structures that are passed to CASA functions.

I usually import this as just 'b' and write tool.method (b(arg)), in analogy with the b" byte string syntax.

7.7 HEASoft (`pwkit.environments.heasoft`)

heasoft - running software in the HEAsoft/CALDB environment

To use, export an environment variable $PWKIT_HEASOFT pointing to the HEAsoft platform-specific directory, usually known as $HEADAS. E.g., $PWKIT_HEASOFT/bin, $PWKIT_HEASOFT/BUILD_DIR, and $PWKIT_HEASOFT/headas-init.sh should exist. CALDB also needs to be set up as described below.

"pfiles" are set up to land in ~/.local/share/hea-pfiles/.

7.7.1 HEAsoft installation notes

(All examples assume version 6.16 for convenience, substitute as needed of course.)

Installation from source strongly recommended. Download from something like http://heasarc.gsfc.nasa.gov/FTP/software/lheasoft/release/heasoft-6.16src.tar.gz - the website lets you customize the tarball, but it's probably easiest just to do the full install every time. Tarball unpacks into *heasoft-6.16/...* so you can safely curlltar in ~/sw/.

$ cd heasoft-6.16/BUILD_DIR $ ./configure –prefix=/a/heasoft/6.16 $ make # note: not parallel-friendly $ make install

The CALDB setup is so lightweight that it's not worth separating it out:

$ cd /a/heasoft/6.16 $ wget http://heasarc.gsfc.nasa.gov/FTP/caldb/software/tools/caldb.config $ wget http://heasarc.gsfc.nasa.gov/FTP/caldb/software/tools/alias_config.fits

7.8 SAS (`pwkit.environments.sas`)

sas - running software in the SAS environment

To use, export an environment variable $PWKIT_SAS pointing to the SAS installation root. The files $PWKIT_SAS/RELEASE and $PWKIT_SAS/setsas.sh should exist. The "current calibration files" (CCF) should be accessible as $PWKIT_SAS/ccf/; a symlink may make sense if multiple SAS versions are going to be used.

SAS is unusual because you need to set up some magic environment variables specific to the dataset that you're working with. There is also default preparation to be run on each dataset before anything useful can be done.

7.8.1 Unpacking data sets

Data sets are downloaded as tar.gz files. Those unpack to a few files in '.' including a .TAR file, which should be unpacked too. That unpacks to a bunch of data files in '.' as well.

7.8.2 SAS installation notes

Download tarball from, e.g.,

ftp://legacy.gsfc.nasa.gov/xmm/software/sas/14.0.0/64/Linux/Fedora20/

Tarball unpacks installation script and data into '.', and the installation script sets up a SAS install in a versioned subdirectory of '.', so curl|tar should be run from something like /a/sas:

```
$ ./install.sh
```

The CCF are like CALDB and need to be rsynced – see the update-ccf subcommand.

7.8.3 ODF data format notes

ODF files all have names in the format RRRR_NNNNNNNNNN_IIUEEECCMMM.ZZZ where:

RRRR revolution (orbit) number

NNNNNNNNNN obs ID

II The instrument:

- **OM** optical monitor
- **R1** RGS (reflection grating spectrometer) unit 1
- **R2** RGS 2
- **M1** EPIC (imaging camera) MOS 1 detector
- **M2** EPIC (imaging camera) MOS 2 detector
- **PN** EPIC (imaging camera) PN detector
- **RM** EPIC radiation monitor
- **SC** spacecraft

U Scheduling status of exposure:

- **S** scheduled
- **U** unscheduled
- **X** N/A

EEE exposure number

CC CCD/OM-window ID

MMM data type of file (many; not listed here)

ZZZ file extension

See the `make-*-aliases` commands for tools that generate symlinks with saner names.

7.9 SAS (`pwkit.environments.sas.data`)

pwkit.environments.sas.data - loading up SAS data sets

CHAPTER 8

Tools for writing command-line programs

This documentation has a lot of stubs.

8.1 Utilities for command-line programs (`pwkit.cli`)

pwkit.cli - miscellaneous utilities for command-line programs.

Functions:

backtrace_on_usr1 - Make it so that a Python backtrace is printed on SIGUSR1. check_usage - Print usage and exit if –help is in argv. die - Print an error and exit. pop_option - Check for a single command-line option. propagate_sigint - Ensure that calling shells know when we die from SIGINT. show_usage - Print a usage message. unicode_stdio - Ensure that sys.std{in,out,err} accept unicode strings. warn - Print a warning. wrong_usage - Print an error about wrong usage and the usage help.

Context managers:

print_tracebacks - Catch exceptions and print tracebacks without reraising them.

Submodules:

multitool - Framework for command-line programs with sub-commands.

`pwkit.cli.`**`check_usage`**(*docstring*, *argv=None*, *usageifnoargs=False*)

Check if the program has been run with a –help argument; if so, print usage information and exit.

Parameters

- **`docstring`** (*str*) – the program help text
- **`argv`** – the program arguments; taken as `sys.argv` if given as `None` (the default). (Note that this implies `argv[0]` should be the program name and not the first option.)
- **`usageifnoargs`** (*bool*) – if `True`, usage information will be printed and the program will exit if no command-line arguments are passed. If "long", print long usasge. Default is `False`.

This function is intended for small programs launched from the command line. The intention is for the program help information to be written in its docstring, and then for the preamble to contain something like:

```
    """myprogram - this is all the usage help you get"""
    import sys
    ... # other setup
    check_usage (__doc__)
    ... # go on with business
```

If it is determined that usage information should be shown, *show_usage()* is called and the program exits.

See also *wrong_usage()*.

pwkit.cli.**die**(*fmt*, **args*)
Raise a SystemExit exception with a formatted error message.

Parameters

- **fmt** (*str*) – a format string
- **args** – arguments to the format string

If *args* is empty, a SystemExit exception is raised with the argument 'error: ' + str (fmt). Otherwise, the string component is fmt % args. If uncaught, the interpreter exits with an error code and prints the exception argument.

Example:

```
if ndim != 3:
   die ('require exactly 3 dimensions, not %d', ndim)
```

pwkit.cli.**pop_option**(*ident*, *argv=None*)
A lame routine for grabbing command-line arguments. Returns a boolean indicating whether the option was present. If it was, it's removed from the argument string. Because of the lame behavior, options can't be combined, and non-boolean options aren't supported. Operates on sys.argv by default.

Note that this will proceed merrily if argv[0] matches your option.

class pwkit.cli.**print_tracebacks**(*types=(<type 'exceptions.Exception'>,)*, *header=None*, *file=None*)
Context manager that catches exceptions and prints their tracebacks without reraising them. Intended for robust programs that want to continue execution even if something bad happens; this provides the infrastructure to swallow exceptions while still preserving exception information for later debugging.

You can specify which exception classes to catch with the *types* keyword argument to the constructor. The *header* keyword will be printed if specified; this could be used to add contextual information. The *file* keyword specifies the destination for the printed output; default is sys.stderr.

Instances preserve the exception information in the fields 'etype', 'evalue', and 'etb' if your program in fact wants to do something with the information. One basic use would be checking whether an exception did, in fact, occur.

pwkit.cli.**show_usage**(*docstring*, *short*, *stream*, *exitcode*)
Print program usage information and exit.

Parameters **docstring** (*str*) – the program help text

This function just prints *docstring* and exits. In most cases, the function *check_usage()* should be used: it automatically checks sys.argv for a sole "-h" or "–help" argument and invokes this function.

This function is provided in case there are instances where the user should get a friendly usage message that *check_usage()* doesn't catch. It can be contrasted with *wrong_usage()*, which prints a terser usage message and exits with an error code.

pwkit.cli.**unicode_stdio**()
Make sure that the standard I/O streams accept Unicode.

The standard I/O streams accept bytes, not Unicode characters. This means that in principle every Unicode string that we want to output should be encoded to utf-8 before print()ing. But Python 2.X has a hack where, if the output is a terminal, it will automatically encode your strings, using UTF-8 in most cases.

BUT this hack doesn't kick in if you pipe your program's output to another program. So it's easy to write a tool that works fine in most cases but then blows up when you log its output to a file.

The proper solution is just to do the encoding right. This function sets things up to do this in the most sensible way I can devise. This approach sets up compatibility with Python 3, which has the stdio streams be in text mode rather than bytes mode to begin with.

Basically, every command-line Python program should call this right at startup. I'm tempted to just invoke this code whenever this module is imported since I foresee many accidentally omissions of the call.

pwkit.cli.**wrong_usage**(*docstring*, **rest*)

Print a message indicating invalid command-line arguments and exit with an error code.

Parameters

- **docstring** (*str*) – the program help text
- **rest** – an optional specific error message

This function is intended for small programs launched from the command line. The intention is for the program help information to be written in its docstring, and then for argument checking to look something like this:

```
    """mytask <input> <output>

    Do something to the input to create the output.
    """
    ...
    import sys
    ... # other setup
    check_usage (__doc__)
    ... # more setup
    if len (sys.argv) != 3:
       wrong_usage (__doc__, "expect exactly 2 arguments, not %d",
                    len (sys.argv))
```

When called, an error message is printed along with the *first stanza* of *docstring*. The program then exits with an error code and a suggestion to run the program with a –help argument to see more detailed usage information. The "first stanza" of *docstring* is defined as everything up until the first blank line, ignoring any leading blank lines.

The optional message in *rest* is treated as follows. If *rest* is empty, the error message "invalid command-line arguments" is printed. If it is a single item, the stringification of that item is printed. If it is more than one item, the first item is treated as a format string, and it is percent-formatted with the remaining values. See the above example.

See also *check_usage()* and *show_usage()*.

8.2 Parsing keyword-style program arguments (pwkit.kwargv)

The *pwkit.kwargv* module provides a framework for parsing keyword-style arguments to command-line programs. It's designed so that you can easily make a routine with complex, structured configuration parameters that can also be driven from the command line.

Keywords are defined by declaring a subclass of the *ParseKeywords* class with fields corresponding to the support keywords:

```
from pwkit.kwargv import ParseKeywords, Custom

class MyConfig (ParseKeywords):
    foo = 1
    bar = str
    multi = [int]
```

```
    extra = Custom (float, required=True)

    @Custom (str)
    def declination (value):
        from pwkit.astutil import parsedeglat
        return parsedeglat (value)
```

Instantiating the subclass fills in all defaults. Calling the *ParseKeywords.parse()* method parses a list of strings (defaulting to `sys.argv[1:]`) and updates the instance's properties. This framework is designed so that you can provide complex configuration to an algorithm either programmatically, or on the command line. A typical use would be:

```
from pwkit.kwargv import ParseKeywords, Custom

class MyConfig (ParseKeywords):
    niter = 1
    input = str
    scales = [int]
    # ...

def my_complex_algorithm (cfg):
   from pwkit.io import Path
   data = Path (cfg.input).read_fits ()

   for i in xrange (cfg.niter):
       # ....

def call_algorithm_in_code ():
    cfg = MyConfig ()
    cfg.input = 'testfile.fits'
    # ...
    my_complex_algorithm (cfg)

if __name__ == '__main__':
    cfg = MyConfig ().parse ()
    my_complex_algorithm (cfg)
```

You could then execute the module as a program and specify arguments in the form `./program niter=5 input=otherfile.fits`.

8.2.1 Keyword Specification Format

Arguments are specified in the following ways:

- `foo = 1` defines a keyword with a default value, type inferred as `int`. Likewise for `str`, `bool`, `float`.
- `bar = str` defines an string keyword with default value of None. Likewise for `int`, `bool`, `float`.
- `multi = [int]` parses as a list of integers of any length, defaulting to the empty list `[]` (I call these "flexible" lists.). List items are separated by commas on the command line.
- `other = [3.0, int]` parses as a 2-element list, defaulting to `[3.0, None]`. If one value is given, the first array item is parsed, and the second is left as its default. (I call these "fixed" lists.)
- `extra = Custom(float, required=True)` parses like `float` and then customizes keyword properties. Supported properties are the attributes of the *KeywordInfo* class.
- Use `Custom` as a decorator (`@Custom`) on a function `foo` defines a keyword `foo` that's parsed according to the `Custom` specification, then has its value fixed up by calling the `foo()` function after the basic parsing.

That is, the final value is `foo (intermediate_value)`. A common pattern is to use a fixup function for a fixed list where the first few values are mandatory (see *KeywordInfo.minvals* below) but later values can be guessed or defaulted.

See the *KeywordInfo* documentation for specification of additional keyword properties that may be specified. The `Custom` name is simply an alias for *KeywordInfo*.

exception `pwkit.kwargv.`**KwargvError**(*fmt*, **args*)
Raised when invalid arguments have been provided.

exception `pwkit.kwargv.`**ParseError**(*fmt*, **args*)
Raised when the structure of the arguments appears legitimate, but a particular value cannot be parsed into its expected type.

class `pwkit.kwargv.`**KeywordInfo**
Properties that a keyword argument may have.

default = None
The default value for the keyword if it's left unspecified.

fixupfunc = None
If not `None`, the final value of the keyword is set to the return value of `fixupfunc(intermediate_value)`.

maxvals = None
The maximum number of values allowed. This only applies for flexible lists; fixed lists have predetermined sizes.

minvals = 0
The minimum number of values allowed in a flexible list, *if the keyword is specified at all*. If you want `minvals = 1`, use `required = True`.

parser = None
A callable used to convert the argument text to a Python value. This attribute is assigned automatically upon setup.

printexc = False
Print the exception as normal if there's an exception when parsing the keyword value. Otherwise there's just a message along the lines of "cannot parse value <val> for keyword <kw>".

repeatable = False
If true, the keyword value(s) will always be contained in a list. If they keyword is specified multiple times (i.e. `./program kw=1 kw=2`), the list will have multiple items (`cfg.kw = [1, 2]`). If the keyword is list-valued, using this will result in a list of lists.

required = False
Whether an error should be raised if the keyword is not seen while parsing.

scale = None
If not `None`, multiply numeric values by this number after parsing.

sep = u','
The textual separator between items for list-valued keywords.

uiname = None
The name of the keyword as parsed from the command-line. For instance, `some_value = Custom (int, uiname="some-value")` will result in a keyword that the user sets by calling `./program some-value=3`. This provides a mechanism to support keyword names that are not legal Python identifiers.

class pwkit.kwargv.**ParseKeywords**
The template class for defining your keyword arguments. A subclass of *pwkit.Holder*. Declare attributes in a subclass following the scheme described above, then call the *ParseKeywords.parse()* method.

parse(*args=None*)
Parse textual keywords as described by this class's attributes, and update this instance's attributes with the parsed values. *args* is a list of strings; if None, it defaults to sys.argv[1:]. Returns *self* for convenience. Raises *KwargvError* if invalid keywords are encountered.

See also *ParseKeywords.parse_or_die()*.

parse_or_die(*args=None*)
Like *ParseKeywords.parse()*, but calls pkwit.cli.die() if a *KwargvError* is raised, printing the exception text. Returns *self* for convenience.

pwkit.kwargv.**basic**(*args=None*)
Parse the string list *args* as a set of keyword arguments in a very simple-minded way, splitting on equals signs. Returns a *pwkit.Holder* instance with attributes set to strings. The form +foo is mapped to setting foo = True on the *pwkit.Holder* instance. If *args* is None, sys.argv[1:] is used. Raises *KwargvError* on invalid arguments (i.e., ones without an equals sign or a leading plus sign).

8.3 Command-line programs with sub-commands (pwkit.cli.multitool)

pwkit.cli.multitool - Framework for command-line tools with sub-commands

This module provides a framework for quickly creating command-line programs that have multiple independent sub-commands (similar to the way Git's interface works).

Classes:

Command A command supported by the tool.

DelegatingCommand A command that delegates to named sub-commands.

HelpCommand A command that prints the help for other commands.

Multitool The tool itself.

UsageError Raised if illegal command-line arguments are used.

Functions:

invoke_tool Run as a tool and exit.

Standard usage:

```
class MyCommand (multitool.Command):
  name = 'info'
  summary = 'Do something useful.'

  def invoke (self, args, **kwargs):
    print ('hello')

class MyTool (multitool.MultiTool):
  cli_name = 'mytool'
  summary = 'Do several useful things.'

HelpCommand = multitool.HelpCommand # optional
```

```
def commandline ():
  multitool.invoke_tool (globals ())
```

pwkit.cli.multitool.**invoke_tool**(*namespace*, *tool_class=None*)

Invoke a tool and exit.

namespace is a namespace-type dict from which the tool is initialized. It should contain exactly one value that is a *Multitool* subclass, and this subclass will be instantiated and populated (see *Multitool.populate()*) using the other items in the namespace. Instances and subclasses of *Command* will therefore be registered with the *Multitool*. The tool is then invoked.

pwkit.cli.propagate_sigint() and *pwkit.cli.unicode_stdio()* are called at the start of this function. It should therefore be only called immediately upon startup of the Python interpreter.

This function always exits with an exception. The exception will be SystemExit (0) in case of success.

The intended invocation is *invoke_tool (globals ())* in some module that defines a *Multitool* subclass and multiple *Command* subclasses.

If *tool_class* is not None, this is used as the tool class rather than searching *namespace*, potentially avoiding problems with modules containing multiple *Multitool* implementations.

class pwkit.cli.multitool.**Command**

A command in a multifunctional CLI tool.

Attributes:

argspec One-line string summarizing the command-line arguments that should be passed to this command.

help_if_no_args If True, usage help will automatically be displayed if no command-line arguments are given.

more_help Additional help text to be displayed below the summary (optional).

name The command's name, as should be specified at the CLI.

summary A one-line summary of this command's functionality.

Functions:

invoke(self, args, **kwargs) Execute this command.

'name' must be set; other attributes are optional, although at least 'summary' and 'argspec' should be set. 'invoke()' must be implemented.

invoke(*args*, ***kwargs*)

Invoke this command. 'args' is a list of the remaining command-line arguments. 'kwargs' contains at least 'argv0', which is the equivalent of, well, *argv[0]* for this command; 'tool', the originating Multitool instance; and 'parent', the parent DelegatingCommand instance. Other kwargs may be added in an application-specific manner. Basic processing of '–help' will already have been done if invoked through invoke_with_usage().

invoke_with_usage(*args*, ***kwargs*)

Invoke the command with standardized usage-help processing. Same calling convention as *Command.invoke()*.

class pwkit.cli.multitool.**DelegatingCommand**(*populate_from_self=True*)

A command that delegates to sub-commands.

Attributes:

cmd_desc The noun used to desribe the sub-commands.

usage_tmpl A formatting template for long tool usage. The default is almost surely acceptable.

Functions:

register Register a new sub-command.

populate Register many sub-commands automatically.

invoke_command(*cmd*, *args*, ***kwargs*)
This function mainly exists to be overridden by subclasses.

populate(*values*)
Register multiple new commands by investigating the iterable *values*. For each item in *values*, instances of *Command* are registered, and subclasses of *Command* are instantiated (with no arguments passed to the constructor) and registered. Other kinds of values are ignored. Returns 'self'.

register(*cmd*)
Register a new command with the tool. 'cmd' is expected to be an instance of *Command*, although here only the *cmd.name* attribute is investigated. Multiple commands with the same name are not allowed to be registered. Returns 'self'.

class pwkit.cli.multitool.**Multitool**
A command-line tool with multiple sub-commands.

Attributes:

cli_name - The usual name of this tool on the command line. more_help - Additional help text. summary - A one-line summary of this tool's functionality.

Functions:

commandline - Execute a command as if invoked from the command-line. register - Register a new command. populate - Register many commands automatically.

commandline(*argv*)
Run as if invoked from the command line. 'argv' is a Unix-style list of arguments, where the zeroth item is the program name (which is ignored here). Usage help is printed if deemed appropriate (e.g., no arguments are given). This function always terminates with an exception, with the exception being a SystemExit(0) in case of success.

Note that we don't actually use *argv[0]* to set *argv0* because it will generally be the full path to the script name, which is unattractive.

exception pwkit.cli.multitool.**UsageError**(*fmt*, **args*)
Raised if illegal command-line arguments are used in a Multitool program.

CHAPTER 9

Behind-the-scenes infrastructure

This documentation has a lot of stubs.

9.1 Interfacing with other software environments (`pwkit.environments`)

pwkit.environments - working with external software environments

Classes:

Environment - base class for launching programs in an external environment.

Submodules:

heasoft - HEAsoft sas - SAS

Functions:

prepend_environ_path - Prepend into a $PATH in an environment dict. prepend_path - Prepend text into a $PATH-like environment variable. user_data_path - Generate paths for storing miscellaneous user data.

Standard usage is to create an *Environment* instance, then use its *launch(argv, ...)* method to run programs in the specified environment. *launch()* returns a *subprocess.Popen* instance that can be used in the standard ways.

`pwkit.environments.`**`prepend_environ_path`**(*env*, *name*, *text*, *pathsep=':'*)

Prepend *text* into a $PATH-like environment variable. *env* is a dictionary of environment variables and *name* is the variable name. *pathsep* is the character separating path elements, defaulting to *os.pathsep*. The variable will be created if it is not already in *env*. Returns *env*.

Example:

prepend_environ_path (env, b'PATH', b'/mypackage/bin')

`pwkit.environments.`**`prepend_path`**(*orig*, *text*, *pathsep=':'*)

Returns a $PATH-like environment variable with *text* prepended. *orig* is the original variable value, or None. *pathsep* is the character separating path elements, defaulting to *os.pathsep*.

Example:

newpath = cli.prepend_path (oldpath, '/mypackage/bin')

See also *prepend_environ_path*.

9.2 Helper for decorators on class methods (`pwkit.method_decorator`)

Python decorator that knows the class the decorated method is bound to.

Please see full description here: https://github.com/denis-ryzhkov/method_decorator/blob/master/README.md

method_decorator version 0.1.3 Copyright (C) 2013 by Denis Ryzhkov <denisr@denisr.com> MIT License, see http://opensource.org/licenses/MIT

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W