

---

# **petl Documentation**

*Release 1.1.1*

**Alistair Miles**

**Apr 06, 2017**



<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Installation	3
1.2	Dependencies and extensions	3
1.3	Design goals	3
1.4	ETL pipelines	4
1.5	Functional and object-oriented programming styles	5
1.6	Interactive use	5
1.7	IPython notebook integration	6
1.8	petl executable	6
1.9	Conventions - table containers and table iterators	6
1.10	Extensions - integrating custom data sources	7
1.11	Caching	8
1.12	Acknowledgments	9
<b>2</b>	<b>Extract/Load - reading/writing tables from files, databases and other sources</b>	<b>11</b>
2.1	Extract (read)	11
2.2	Load (write)	11
2.3	Python objects	12
2.4	Delimited files	12
2.5	Pickle files	14
2.6	Text files	15
2.7	XML files	17
2.8	HTML files	19
2.9	JSON files	19
2.10	Databases	21
2.11	Excel .xls files (xlrd/xlwt)	24
2.12	Excel .xlsx files (openpyxl)	24
2.13	Arrays (NumPy)	25
2.14	DataFrames (pandas)	26
2.15	HDF5 files (PyTables)	27
2.16	Bcolz ctables	29
2.17	Text indexes (Whoosh)	31
2.18	I/O helper classes	34
<b>3</b>	<b>Transform - transforming tables</b>	<b>37</b>
3.1	Basic transformations	37
3.2	Header manipulations	46

3.3	Converting values . . . . .	48
3.4	Selecting rows . . . . .	52
3.5	Regular expressions . . . . .	56
3.6	Unpacking compound values . . . . .	58
3.7	Transforming rows . . . . .	59
3.8	Sorting . . . . .	62
3.9	Joins . . . . .	64
3.10	Set operations . . . . .	71
3.11	Deduplicating rows . . . . .	75
3.12	Reducing rows (aggregation) . . . . .	78
3.13	Reshaping tables . . . . .	83
3.14	Filling missing values . . . . .	88
3.15	Validation . . . . .	91
3.16	Intervals (intervaltree) . . . . .	92
<b>4</b>	<b>Utility functions</b>	<b>99</b>
4.1	Basic utilities . . . . .	99
4.2	Visualising tables . . . . .	102
4.3	Lookup data structures . . . . .	103
4.4	Parsing string/text values . . . . .	107
4.5	Counting . . . . .	108
4.6	Timing . . . . .	113
4.7	Statistics . . . . .	114
4.8	Materialising tables . . . . .	114
4.9	Randomly generated tables . . . . .	115
4.10	Miscellaneous . . . . .	116
<b>5</b>	<b>Changes</b>	<b>119</b>
5.1	Version 1.1.0 . . . . .	119
5.2	Version 1.0 . . . . .	120
<b>6</b>	<b>Contributing</b>	<b>125</b>
6.1	Running the test suite . . . . .	125
6.2	Dependencies . . . . .	125
6.3	Running database tests . . . . .	126
6.4	Running doctests . . . . .	126
6.5	Building the documentation . . . . .	126
6.6	Automatically running all tests . . . . .	126
6.7	Contributing code via GitHub . . . . .	126
<b>7</b>	<b>Related Work</b>	<b>127</b>
<b>8</b>	<b>Indices and tables</b>	<b>133</b>
	<b>Python Module Index</b>	<b>135</b>

*petl* is a general purpose Python package for extracting, transforming and loading tables of data.

- Documentation: <http://petl.readthedocs.org/>
- Source Code: <https://github.com/alimanfoo/petl>
- Download: <http://pypi.python.org/pypi/petl>
- Mailing List: <http://groups.google.com/group/python-etl>

Please feel free to ask questions via the mailing list ([python-etl@googlegroups.com](mailto:python-etl@googlegroups.com)).

To report installation problems, bugs or any other issues please email [python-etl@googlegroups.com](mailto:python-etl@googlegroups.com) or raise an issue on [GitHub](#).

For an example of *petl* in use, see the [case study on comparing tables](#).

For an alphabetic list of all functions in the package, see the [genindex](#).

---

**Note:** Version 1.0 is a new major release of *petl*. See the [Changes](#) section for more information about what has changed in version 1.0.

---



### Installation

This package is available from the [Python Package Index](#). If you have `pip` you should be able to do:

```
$ pip install petl
```

You can also download manually, extract and run `python setup.py install`.

To verify the installation, the test suite can be run with `nose`, e.g.:

```
$ pip install nose
$ nosetests -v petl
```

`petl` is compatible with Python versions 2.6, 2.7 and 3.4, and has been tested under Linux and Windows operating systems.

### Dependencies and extensions

This package is written in pure Python and has no installation requirements other than the Python core modules.

Some of the functions in this package require installation of third party packages. This is indicated in the relevant parts of the documentation.

Some domain-specific and/or experimental extensions to `petl` are available from the `petlx` package.

### Design goals

This package is designed primarily for convenience and ease of use, especially when working interactively with data that are unfamiliar, heterogeneous and/or of mixed quality.

*petl* transformation pipelines make minimal use of system memory and can scale to millions of rows if speed is not a priority. However if you are working with very large datasets and/or performance-critical applications then other packages may be more suitable, e.g., see [pandas](#), [pytables](#), [bcolz](#) and [blaze](#). See also [Related Work](#).

## ETL pipelines

This package makes extensive use of lazy evaluation and iterators. This means, generally, that a pipeline will not actually be executed until data is requested.

E.g., given a file at 'example.csv' in the current working directory:

```
>>> example_data = """foo,bar,baz
... a,1,3.4
... b,2,7.4
... c,6,2.2
... d,9,8.1
... """
>>> with open('example.csv', 'w') as f:
...     f.write(example_data)
... 
```

...the following code **does not** actually read the file or load any of its contents into memory:

```
>>> import petl as etl
>>> table1 = etl.fromcsv('example.csv')
```

Rather, *table1* is a **table container** (see [Conventions - table containers and table iterators](#) below) which can be iterated over, extracting data from the underlying file on demand.

Similarly, if one or more transformation functions are applied, e.g.:

```
>>> table2 = etl.convert(table1, 'foo', 'upper')
>>> table3 = etl.convert(table2, 'bar', int)
>>> table4 = etl.convert(table3, 'baz', float)
>>> table5 = etl.addfield(table4, 'quux', lambda row: row.bar * row.baz)
```

...no actual transformation work will be done until data are requested from *table5* (or any of the other tables returned by the intermediate steps).

So in effect, a 5 step pipeline has been set up, and rows will pass through the pipeline on demand, as they are pulled from the end of the pipeline via iteration.

A call to a function like *petl.util.vis.look()*, or any of the functions which write data to a file or database (e.g., *petl.io.csv.tocsv()*, *petl.io.text.totext()*, *petl.io.sqlite3.tosqlite3()*, *petl.io.db.todb()*), will pull data through the pipeline and cause all of the transformation steps to be executed on the requested rows, e.g.:

```
>>> etl.look(table5)
+-----+-----+-----+-----+
| foo | bar | baz | quux |
+=====+=====+=====+=====+
| 'A' | 1 | 3.4 | 3.4 |
+-----+-----+-----+-----+
| 'B' | 2 | 7.4 | 14.8 |
+-----+-----+-----+-----+
| 'C' | 6 | 2.2 | 13.200000000000001 |
+-----+-----+-----+-----+
```





```
>>> l = [['foo', 'bar'], ['a', 1], ['b', 2], ['c', 2]]
>>> table = etl.wrap(l)
>>> table
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
| 'c' | 2 |
+-----+-----+
```

By default data values are rendered using the built-in `repr()` function. To see the string (`str()`) values instead, `print()` the table, e.g.:

```
>>> print(table)
+-----+-----+
| foo | bar |
+=====+=====+
| a | 1 |
+-----+-----+
| b | 2 |
+-----+-----+
| c | 2 |
+-----+-----+
```

## IPython notebook integration

Table objects also implement `_repr_html_()` and so will be displayed as an HTML table if returned from a cell in an IPython notebook. The functions `petl.util.vis.display()` and `petl.util.vis.displayall()` also provide more control over rendering of tables within an IPython notebook.

For examples of usage see the `repr_html` notebook.

## petl executable

Also included in the `petl` distribution is a script to execute simple transformation pipelines directly from the operating system shell. E.g.:

```
$ petl "dummytable().tocsv()" > example.csv
$ cat example.csv | petl "fromcsv().cut('foo', 'baz').convert('baz', float).selectgt(
→ 'baz', 0.5).head().data().totsv()"
```

The `petl` script is extremely simple, it expects a single positional argument, which is evaluated as Python code but with all of the functions in the `petl` namespace imported.

## Conventions - table containers and table iterators

This package defines the following convention for objects acting as containers of tabular data and supporting row-oriented iteration over the data.

A **table container** (also referred to here as a **table**) is any object which satisfies the following:

1. implements the `__iter__` method
2. `__iter__` returns a **table iterator** (see below)
3. all table iterators returned by `__iter__` are independent, i.e., consuming items from one iterator will not affect any other iterators

A **table iterator** is an iterator which satisfies the following:

4. each item returned by the iterator is a sequence (e.g., tuple or list)
5. the first item returned by the iterator is a **header row** comprising a sequence of **header values**
6. each subsequent item returned by the iterator is a **data row** comprising a sequence of **data values**
7. a **header value** is typically a string (*str*) but may be an object of any type as long as it implements `__str__` and is pickleable
8. a **data value** is any pickleable object

So, for example, a list of lists is a valid table container:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
```

Note that an object returned by the `csv.reader()` function from the standard Python `csv` module is a table iterator and **not** a table container, because it can only be iterated over once. However, it is straightforward to define functions that support the table container convention and provide access to data from CSV or other types of file or data source, see e.g. the `petl.io.csv.fromcsv()` function.

The main reason for requiring that table containers support independent table iterators (point 3) is that data from a table may need to be iterated over several times within the same program or interactive session. E.g., when using `petl` in an interactive session to build up a sequence of data transformation steps, the user might want to examine outputs from several intermediate steps, before all of the steps are defined and the transformation is executed in full.

Note that this convention does not place any restrictions on the lengths of header and data rows. A table may contain a header row and/or data rows of varying lengths.

## Extensions - integrating custom data sources

The `petl.io` module has functions for extracting data from a number of well-known data sources. However, it is also straightforward to write an extension that enables integration with other data sources. For an object to be usable as a `petl` table it has to implement the **table container** convention described above. Below is the source code for an `ArrayView` class which allows integration of `petl` with numpy arrays. This class is included within the `petl.io.numpy` module but also provides an example of how other data sources might be integrated:

```
>>> import petl as etl
>>> class ArrayView(etl.Table):
...     def __init__(self, a):
...         # assume that a is a numpy array
...         self.a = a
...     def __iter__(self):
...         # yield the header row
...         header = tuple(self.a.dtype.names)
...         yield header
...         # yield the data rows
...         for row in self.a:
...             yield tuple(row)
... 
```

Now this class enables the use of numpy arrays with *petl* functions, e.g.:

```
>>> import numpy as np
>>> a = np.array([('apples', 1, 2.5),
...              ('oranges', 3, 4.4),
...              ('pears', 7, 0.1)],
...              dtype='U8, i4, f4')
>>> t1 = ArrayView(a)
>>> t1
```

f0	f1	f2
'apples'	1	2.5
'oranges'	3	4.4000001
'pears'	7	0.1

```
>>> t2 = t1.cut('f0', 'f2').convert('f0', 'upper').addfield('f3', lambda row: row.f2_
↳ * 2)
>>> t2
```

f0	f2	f3
'APPLES'	2.5	5.0
'ORANGES'	4.4000001	8.8000001907348633
'PEARS'	0.1	0.20000000298023224

If you develop an extension for a data source that you think would also be useful for others, please feel free to submit a PR to the [petl GitHub repository](#), or if it is a domain-specific data source, the [petlx GitHub repository](#).

## Caching

This package tries to make efficient use of memory by using iterators and lazy evaluation where possible. However, some transformations cannot be done without building data structures, either in memory or on disk.

An example is the `petl.transform.sorts.sort()` function, which will either sort a table entirely in memory, or will sort the table in memory in chunks, writing chunks to disk and performing a final merge sort on the chunks. Which strategy is used will depend on the arguments passed into the `petl.transform.sorts.sort()` function when it is called.

In either case, the sorting can take some time, and if the sorted data will be used more than once, it is undesirable to start again from scratch each time. It is better to cache the sorted data, if possible, so it can be re-used.

The `petl.transform.sorts.sort()` function, and all functions which use it internally, provide a `cache` keyword argument which can be used to turn on or off the caching of sorted data.

There is also an explicit `petl.util.materialise.cache()` function, which can be used to cache in memory up to a configurable number of rows from any table.

## Acknowledgments

This package is maintained by Alistair Miles <[alimanfoo@googlemail.com](mailto:alimanfoo@googlemail.com)> with funding from the MRC Centre for Genomics and Global Health.

The following people have contributed to the development of this package:

- Alexander Stauber
- Andreas Porevopoulos ([sv1jsb](#))
- Andrew Kim ([andrewakim](#))
- Caleb Lloyd ([caleblloyd](#))
- César Roldán ([ihuro](#))
- Chris Lasher ([gotgenes](#))
- Dustin Engstrom ([engstrom](#))
- Florent Xicluna ([florentx](#))
- Jonathan Camile ([dey tao](#))
- Jonathan Moss ([a-musing-moose](#))
- Kenneth Borthwick
- Krisztián Fekete ([krisztianfekete](#))
- Michael Rea ([rea725](#))
- Olivier Macchioni ([omacchioni](#))
- Olivier Poitrey ([rs](#))
- Paul Jensen ([psnj](#))
- Peder Jakobsen ([pjakobsen](#))
- Phillip Knaus ([phillipknaus](#))
- Richard Pearson ([podpearson](#))
- Robert DeSimone ([icenine457](#))
- Roger Woodley ([rogerkwoodley](#))
- Zach Palchick ([palchicz](#))
- [adamsdarlingtower](#)
- [imazor](#)
- [james-unified](#)
- [Mgutjahr](#)
- [shayh](#)
- [thatneat](#)
- [titusz](#)
- [zigen](#)



---

## Extract/Load - reading/writing tables from files, databases and other sources

---

### Extract (read)

The “from...” functions extract a table from a file-like source or database. For everything except `petl.io.db.fromdb()` the `source` argument provides information about where to extract the underlying data from. If the `source` argument is `None` or a string it is interpreted as follows:

- `None` - read from `stdin`
- string starting with `http://`, `https://` or `ftp://` - read from URL
- string ending with `.gz` or `.bgz` - read from file via gzip decompression
- string ending with `.bz2` - read from file via bz2 decompression
- any other string - read directly from file

Some helper classes are also available for reading from other types of file-like sources, e.g., reading data from a Zip file, a string or a subprocess, see the section on *I/O helper classes* below for more information.

Be aware that loading data from `stdin` breaks the table container convention, because data can usually only be read once. If you are sure that data will only be read once in your script or interactive session then this may not be a problem, however note that some `petl` functions do access the underlying data source more than once and so will not work as expected with data from `stdin`.

### Load (write)

The “to...” functions load data from a table into a file-like source or database. For functions that accept a `source` argument, if the `source` argument is `None` or a string it is interpreted as follows:

- `None` - write to `stdout`
- string ending with `.gz` or `.bgz` - write to file via gzip decompression

- string ending with `.bz2` - write to file via bz2 decompression
- any other string - write directly to file

Some helper classes are also available for writing to other types of file-like sources, e.g., writing to a Zip file or string buffer, see the section on *I/O helper classes* below for more information.

## Python objects

`petl.io.base.fromcolumns` (*cols*, *header=None*, *missing=None*)

View a sequence of columns as a table, e.g.:

```
>>> import petl as etl
>>> cols = [[0, 1, 2], ['a', 'b', 'c']]
>>> tbl = etl.fromcolumns(cols)
>>> tbl
+----+-----+
| f0 | f1 |
+====+=====+
| 0 | 'a' |
+----+-----+
| 1 | 'b' |
+----+-----+
| 2 | 'c' |
+----+-----+
```

If columns are not the same length, values will be padded to the length of the longest column with *missing*, which is `None` by default, e.g.:

```
>>> cols = [[0, 1, 2], ['a', 'b']]
>>> tbl = etl.fromcolumns(cols, missing='NA')
>>> tbl
+----+-----+
| f0 | f1 |
+====+=====+
| 0 | 'a' |
+----+-----+
| 1 | 'b' |
+----+-----+
| 2 | 'NA' |
+----+-----+
```

See also `petl.io.json.fromdicts()`.

New in version 1.1.0.

## Delimited files

`petl.io.csv.fromcsv` (*source=None*, *encoding=None*, *errors='strict'*, *header=None*, *\*\*csvargs*)

Extract a table from a delimited file. E.g.:

```
>>> import petl as etl
>>> import csv
>>> # set up a CSV file to demonstrate with
... table1 = [['foo', 'bar'],
```



```

...         ['a', 1],
...         ['b', 2],
...         ['c', 2]]
>>> with open('example.csv', 'w') as f:
...     writer = csv.writer(f)
...     writer.writerows(table1)
...
>>> # now demonstrate the use of fromcsv()
... table2 = etl.fromcsv('example.csv')
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | '1' |
+-----+-----+
| 'b' | '2' |
+-----+-----+
| 'c' | '2' |
+-----+-----+

```

The *source* argument is the path of the delimited file, all other keyword arguments are passed to `csv.reader()`. So, e.g., to override the delimiter from the default CSV dialect, provide the *delimiter* keyword argument.

Note that all data values are strings, and any intended numeric values will need to be converted, see also `petl.transform.conversions.convert()`.

`petl.io.csv.tocsv(table, source=None, encoding=None, errors='strict', write_header=True, **csvargs)`

Write the table to a CSV file. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> etl.tocsv(table1, 'example.csv')
>>> # look what it did
... print(open('example.csv').read())
foo,bar
a,1
b,2
c,2

```

The *source* argument is the path of the delimited file, and the optional *write\_header* argument specifies whether to include the field names in the delimited file. All other keyword arguments are passed to `csv.writer()`. So, e.g., to override the delimiter from the default CSV dialect, provide the *delimiter* keyword argument.

Note that if a file already exists at the given location, it will be overwritten.

`petl.io.csv.appendcsv(table, source=None, encoding=None, errors='strict', write_header=False, **csvargs)`

Append data rows to an existing CSV file. As `petl.io.csv.tocsv()` but the file is opened in append mode and the table header is not written by default.

Note that no attempt is made to check that the fields or row lengths are consistent with the existing data, the data rows from the table are simply appended to the file.

`petl.io.csv.teecsv(table, source=None, encoding=None, errors='strict', write_header=True, **csvargs)`

Returns a table that writes rows to a CSV file as they are iterated over.

`petl.io.csv.fromtsv` (*source=None, encoding=None, errors='strict', header=None, \*\*csvargs*)  
 Convenience function, as `petl.io.csv.fromcsv()` but with different default dialect (tab delimited).

`petl.io.csv.totsv` (*table, source=None, encoding=None, errors='strict', write\_header=True, \*\*csvargs*)  
 Convenience function, as `petl.io.csv.tocsv()` but with different default dialect (tab delimited).

`petl.io.csv.appendtsv` (*table, source=None, encoding=None, errors='strict', write\_header=False, \*\*csvargs*)  
 Convenience function, as `petl.io.csv.appendcsv()` but with different default dialect (tab delimited).

`petl.io.csv.teetsv` (*table, source=None, encoding=None, errors='strict', write\_header=True, \*\*csvargs*)  
 Convenience function, as `petl.io.csv.teecsv()` but with different default dialect (tab delimited).

## Pickle files

`petl.io.pickle.frompickle` (*source=None*)  
 Extract a table From data pickled in the given file. The rows in the table should have been pickled to the file one at a time. E.g.:

```
>>> import petl as etl
>>> import pickle
>>> # set up a file to demonstrate with
... with open('example.p', 'wb') as f:
...     pickle.dump(['foo', 'bar'], f)
...     pickle.dump(['a', 1], f)
...     pickle.dump(['b', 2], f)
...     pickle.dump(['c', 2.5], f)
...
>>> # now demonstrate the use of frompickle()
... table1 = etl.frompickle('example.p')
>>> table1
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
| 'c' | 2.5 |
+-----+-----+
```

`petl.io.pickle.topickle` (*table, source=None, protocol=-1, write\_header=True*)  
 Write the table to a pickle file. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> etl.topickle(table1, 'example.p')
>>> # look what it did
... table2 = etl.frompickle('example.p')
>>> table2
+-----+-----+
```

```

| foo | bar |
+-----+-----+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
| 'c' | 2 |
+-----+-----+

```

Note that if a file already exists at the given location, it will be overwritten.

The pickle file format preserves type information, i.e., reading and writing is round-trippable for tables with non-string data values.

`petl.io.pickle.appendpickle` (*table*, *source=None*, *protocol=-1*, *write\_header=False*)

Append data to an existing pickle file. I.e., as `petl.io.pickle.topickle()` but the file is opened in append mode.

Note that no attempt is made to check that the fields or row lengths are consistent with the existing data, the data rows from the table are simply appended to the file.

`petl.io.pickle.teepickle` (*table*, *source=None*, *protocol=-1*, *write\_header=True*)

Return a table that writes rows to a pickle file as they are iterated over.

## Text files

`petl.io.text.fromtext` (*source=None*, *encoding=None*, *errors='strict'*, *strip=None*, *header=('lines',)*)

Extract a table from lines in the given text file. E.g.:

```

>>> import petl as etl
>>> # setup example file
... text = 'a,1\nb,2\nc,2\n'
>>> with open('example.txt', 'w') as f:
...     f.write(text)
...
12
>>> table1 = etl.fromtext('example.txt')
>>> table1
+-----+
| lines |
+-----+
| 'a,1' |
+-----+
| 'b,2' |
+-----+
| 'c,2' |
+-----+

>>> # post-process, e.g., with capture()
... table2 = table1.capture('lines', '(.*),(.*)$', ['foo', 'bar'])
>>> table2
+-----+-----+
| foo | bar |
+-----+-----+
| 'a' | '1' |
+-----+-----+

```

```
| 'b' | '2' |
+-----+-----+
| 'c' | '2' |
+-----+-----+
```

Note that the `strip()` function is called on each line, which by default will remove leading and trailing whitespace, including the end-of-line character - use the `strip` keyword argument to specify alternative characters to strip. Set the `strip` argument to `False` to disable this behaviour and leave line endings in place.

`petl.io.text.totext` (*table*, *source=None*, *encoding=None*, *errors='strict'*, *template=None*, *prologue=None*, *epilogue=None*)

Write the table to a text file. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> prologue = '''{| class="wikitable"
... |-
... ! foo
... ! bar
... '''
>>> template = '''|-
... | {foo}
... | {bar}
... '''
>>> epilogue = '|}'
>>> etl.totext(table1, 'example.txt', template=template,
... prologue=prologue, epilogue=epilogue)
>>> # see what we did
... print(open('example.txt').read())
{| class="wikitable"
|-
! foo
! bar
|-
| a
| 1
|-
| b
| 2
|-
| c
| 2
|}
```

The *template* will be used to format each row via `str.format`.

`petl.io.text.appendtext` (*table*, *source=None*, *encoding=None*, *errors='strict'*, *template=None*, *prologue=None*, *epilogue=None*)

Append the table to a text file.

`petl.io.text.teetext` (*table*, *source=None*, *encoding=None*, *errors='strict'*, *template=None*, *prologue=None*, *epilogue=None*)

Return a table that writes rows to a text file as they are iterated over.

## XML files

`petl.io.xml.fromxml` (*source*, \**args*, \*\**kwargs*)

Extract data from an XML file. E.g.:

```
>>> import petl as etl
>>> # setup a file to demonstrate with
... d = '''<table>
...     <tr>
...         <td>foo</td><td>bar</td>
...     </tr>
...     <tr>
...         <td>a</td><td>1</td>
...     </tr>
...     <tr>
...         <td>b</td><td>2</td>
...     </tr>
...     <tr>
...         <td>c</td><td>2</td>
...     </tr>
... </table>'''
>>> with open('example1.xml', 'w') as f:
...     f.write(d)
...
212
>>> table1 = etl.fromxml('example1.xml', 'tr', 'td')
>>> table1
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | '1' |
+-----+-----+
| 'b' | '2' |
+-----+-----+
| 'c' | '2' |
+-----+-----+
```

If the data values are stored in an attribute, provide the attribute name as an extra positional argument:

```
>>> d = '''<table>
...     <tr>
...         <td v='foo'/><td v='bar'/>
...     </tr>
...     <tr>
...         <td v='a'/><td v='1'/>
...     </tr>
...     <tr>
...         <td v='b'/><td v='2'/>
...     </tr>
...     <tr>
...         <td v='c'/><td v='2'/>
...     </tr>
... </table>'''
>>> with open('example2.xml', 'w') as f:
...     f.write(d)
...
220
>>> table2 = etl.fromxml('example2.xml', 'tr', 'td', 'v')
```

```
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | '1' |
+-----+-----+
| 'b' | '2' |
+-----+-----+
| 'c' | '2' |
+-----+-----+
```

Data values can also be extracted by providing a mapping of field names to element paths:

```
>>> d = '''<table>
...     <row>
...         <foo>a</foo><baz><bar v='1'/><bar v='3'/></baz>
...     </row>
...     <row>
...         <foo>b</foo><baz><bar v='2'/></baz>
...     </row>
...     <row>
...         <foo>c</foo><baz><bar v='2'/></baz>
...     </row>
... </table>'''
>>> with open('example3.xml', 'w') as f:
...     f.write(d)
...
223
>>> table3 = etl.fromxml('example3.xml', 'row',
...                      {'foo': 'foo', 'bar': ('baz/bar', 'v')})
>>> table3
+-----+-----+
| bar          | foo |
+=====+=====+
| ('1', '3') | 'a' |
+-----+-----+
| '2'          | 'b' |
+-----+-----+
| '2'          | 'c' |
+-----+-----+
```

If `lxml` is installed, full XPath expressions can be used.

Note that the implementation is currently **not** streaming, i.e., the whole document is loaded into memory.

If multiple elements match a given field, all values are reported as a tuple.

If there is more than one element name used for row values, a tuple or list of paths can be provided, e.g., `fromxml('example.html', './tr', ('th', 'td'))`.

For writing to an XML file, see `petl.io.text.totext()`.

## HTML files

`petl.io.html.tohtml` (*table*, *source=None*, *encoding=None*, *errors='strict'*, *caption=None*, *repr=<type 'unicode'>*, *lineterminator='\n'*, *index\_header=False*, *tr\_style=None*, *td\_styles=None*, *truncate=None*)

Write the table as HTML to a file. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> etl.tohtml(table1, 'example.html', caption='example table')
>>> print(open('example.html').read())
<table class='petl'>
<caption>example table</caption>
<thead>
<tr>
<th>foo</th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td style='text-align: right'>1</td>
</tr>
<tr>
<td>b</td>
<td style='text-align: right'>2</td>
</tr>
<tr>
<td>c</td>
<td style='text-align: right'>2</td>
</tr>
</tbody>
</table>
```

The *caption* keyword argument is used to provide a table caption in the output HTML.

`petl.io.html.teehtml` (*table*, *source=None*, *encoding=None*, *errors='strict'*, *caption=None*, *repr=<type 'unicode'>*, *lineterminator='\n'*, *index\_header=False*, *tr\_style=None*, *td\_styles=None*, *truncate=None*)

Return a table that writes rows to a Unicode HTML file as they are iterated over.

## JSON files

`petl.io.json.fromjson` (*source*, *\*args*, *\*\*kwargs*)

Extract data from a JSON file. The file must contain a JSON array as the top level object, and each member of the array will be treated as a row of data. E.g.:

```
>>> import petl as etl
>>> data = '''
... [{"foo": "a", "bar": 1},
... {"foo": "b", "bar": 2},
... {"foo": "c", "bar": 2}]
```

```

... '''
>>> with open('example.json', 'w') as f:
...     f.write(data)
...
74
>>> table1 = etl.fromjson('example.json', header=['foo', 'bar'])
>>> table1
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
| 'c' | 2 |
+-----+-----+

```

If your JSON file does not fit this structure, you will need to parse it via `json.load()` and select the array to treat as the data, see also `petl.io.json.fromdicts()`.

Changed in version 1.1.0.

If no *header* is specified, fields will be discovered by sampling keys from the first *sample* objects in *source*. The header will be constructed from keys in the order discovered. Note that this ordering may not be stable, and therefore it may be advisable to specify an explicit *header* or to use another function like `petl.transform.headers.sortheader()` on the resulting table to guarantee stability.

`petl.io.json.fromdicts(dict, header=None, sample=1000, missing=None)`

View a sequence of Python dict as a table. E.g.:

```

>>> import petl as etl
>>> dicts = [{"foo": "a", "bar": 1},
...         {"foo": "b", "bar": 2},
...         {"foo": "c", "bar": 2}]
>>> table1 = etl.fromdicts(dicts, header=['foo', 'bar'])
>>> table1
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
| 'c' | 2 |
+-----+-----+

```

If *header* is not specified, *sample* items from *dicts* will be inspected to discovery dictionary keys. Note that the order in which dictionary keys are discovered may not be stable,

See also `petl.io.json.fromjson()`.

Changed in version 1.1.0.

If no *header* is specified, fields will be discovered by sampling keys from the first *sample* dictionaries in *dicts*. The header will be constructed from keys in the order discovered. Note that this ordering may not be stable, and therefore it may be advisable to specify an explicit *header* or to use another function like `petl.transform.headers.sortheader()` on the resulting table to guarantee stability.

`petl.io.json.tojson(table, source=None, prefix=None, suffix=None, *args, **kwargs)`

Write a table in JSON format, with rows output as JSON objects. E.g.:



```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> etl.tojson(table1, 'example.json', sort_keys=True)
>>> # check what it did
... print(open('example.json').read())
[{"bar": 1, "foo": "a"}, {"bar": 2, "foo": "b"}, {"bar": 2, "foo": "c"}]
```

Note that this is currently not streaming, all data is loaded into memory before being written to the file.

`petl.io.json.tojsonarrays` (*table*, *source=None*, *prefix=None*, *suffix=None*, *output\_header=False*, *\*args*, *\*\*kwargs*)

Write a table in JSON format, with rows output as JSON arrays. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 2]]
>>> etl.tojsonarrays(table1, 'example.json')
>>> # check what it did
... print(open('example.json').read())
[["a", 1], ["b", 2], ["c", 2]]
```

Note that this is currently not streaming, all data is loaded into memory before being written to the file.

## Databases

**Note:** The automatic table creation feature of `petl.io.db.todb()` requires `SQLAlchemy` to be installed, e.g.:

```
$ pip install sqlalchemy
```

`petl.io.db.fromdb` (*dbo*, *query*, *\*args*, *\*\*kwargs*)

Provides access to data from any DB-API 2.0 connection via a given query. E.g., using `sqlite3`:

```
>>> import petl as etl
>>> import sqlite3
>>> connection = sqlite3.connect('example.db')
>>> table = etl.fromdb(connection, 'SELECT * FROM example')
```

E.g., using `psycopg2` (assuming you've installed it first):

```
>>> import petl as etl
>>> import psycopg2
>>> connection = psycopg2.connect('dbname=example user=postgres')
>>> table = etl.fromdb(connection, 'SELECT * FROM example')
```

E.g., using `pymysql` (assuming you've installed it first):

```
>>> import petl as etl
>>> import pymysql
```

```
>>> connection = pymysql.connect(password='moonpie', database='thangs')
>>> table = etl.fromdb(connection, 'SELECT * FROM example')
```

The *dbo* argument may also be a function that creates a cursor. N.B., each call to the function should return a new cursor. E.g.:

```
>>> import petl as etl
>>> import psycopg2
>>> connection = psycopg2.connect('dbname=example user=postgres')
>>> mkcursor = lambda: connection.cursor(cursor_factory=psycopg2.extras.
↳DictCursor)
>>> table = etl.fromdb(mkcursor, 'SELECT * FROM example')
```

The parameter *dbo* may also be an SQLAlchemy engine, session or connection object.

The parameter *dbo* may also be a string, in which case it is interpreted as the name of a file containing an `sqlite3` database.

Note that the default behaviour of most database servers and clients is for the entire result set for each query to be sent from the server to the client. If your query returns a large result set this can result in significant memory usage at the client. Some databases support server-side cursors which provide a means for client libraries to fetch result sets incrementally, reducing memory usage at the client.

To use a server-side cursor with a PostgreSQL database, e.g.:

```
>>> import petl as etl
>>> import psycopg2
>>> connection = psycopg2.connect('dbname=example user=postgres')
>>> table = etl.fromdb(lambda: connection.cursor(name='arbitrary'),
...                   'SELECT * FROM example')
```

For more information on server-side cursors see the following links:

- <http://initd.org/psycopg/docs/usage.html#server-side-cursors>
- <http://mysql-python.sourceforge.net/MySQLdb.html#using-and-extending>

`petl.io.db.todb` (*table*, *dbo*, *tablename*, *schema=None*, *commit=True*, *create=False*, *drop=False*, *constraints=True*, *metadata=None*, *dialect=None*, *sample=1000*)

Load data into an existing database table via a DB-API 2.0 connection or cursor. Note that the database table will be truncated, i.e., all existing rows will be deleted prior to inserting the new data. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'],
...         ['a', 1],
...         ['b', 2],
...         ['c', 2]]
>>> # using sqlite3
... import sqlite3
>>> connection = sqlite3.connect('example.db')
>>> # assuming table "foobar" already exists in the database
... etl.todb(table, connection, 'foobar')
>>> # using psycopg2
>>> import psycopg2
>>> connection = psycopg2.connect('dbname=example user=postgres')
>>> # assuming table "foobar" already exists in the database
... etl.todb(table, connection, 'foobar')
>>> # using pymysql
>>> import pymysql
```

```
>>> connection = pymysql.connect(password='moonpie', database='thangs')
>>> # tell MySQL to use standard quote character
... connection.cursor().execute('SET SQL_MODE=ANSI_QUOTES')
>>> # load data, assuming table "foobar" already exists in the database
... etl.todb(table, connection, 'foobar')
```

N.B., for MySQL the statement `SET SQL_MODE=ANSI_QUOTES` is required to ensure MySQL uses SQL-92 standard quote characters.

A cursor can also be provided instead of a connection, e.g.:

```
>>> import psycopg2
>>> connection = psycopg2.connect('dbname=example user=postgres')
>>> cursor = connection.cursor()
>>> etl.todb(table, cursor, 'foobar')
```

The parameter `dbo` may also be an SQLAlchemy engine, session or connection object.

The parameter `dbo` may also be a string, in which case it is interpreted as the name of a file containing an sqlite3 database.

If `create=True` this function will attempt to automatically create a database table before loading the data. This functionality requires [SQLAlchemy](#) to be installed.

#### Keyword arguments:

**table** [table container] Table data to load

**dbo** [database object] DB-API 2.0 connection, callable returning a DB-API 2.0 cursor, or SQLAlchemy connection, engine or session

**tablename** [string] Name of the table in the database

**schema** [string] Name of the database schema to find the table in

**commit** [bool] If True commit the changes

**create** [bool] If True attempt to create the table before loading, inferring types from a sample of the data (requires SQLAlchemy)

**drop** [bool] If True attempt to drop the table before recreating (only relevant if `create=True`)

**constraints** [bool] If True use length and nullable constraints (only relevant if `create=True`)

**metadata** [sqlalchemy.MetaData] Custom table metadata (only relevant if `create=True`)

**dialect** [string] One of {'access', 'sybase', 'sqlite', 'informix', 'firebird', 'mysql', 'oracle', 'maxdb', 'postgresql', 'mssql'} (only relevant if `create=True`)

**sample** [int] Number of rows to sample when inferring types etc. Set to 0 to use the whole table (only relevant if `create=True`)

**Note:** This function is in principle compatible with any DB-API 2.0 compliant database driver. However, at the time of writing some DB-API 2.0 implementations, including `cx_Oracle` and MySQL's Connector/Python, are not compatible with this function, because they only accept a list argument to the `cursor.executemany()` function called internally by *petl*. This can be worked around by proxying the cursor objects, e.g.:

```
>>> import cx_Oracle
>>> connection = cx_Oracle.Connection(...)
>>> class CursorProxy(object):
...     def __init__(self, cursor):
...         self._cursor = cursor
```

```
...     def executemany(self, statement, parameters, **kwargs):
...         # convert parameters to a list
...         parameters = list(parameters)
...         # pass through to proxied cursor
...         return self._cursor.executemany(statement, parameters, **kwargs)
...     def __getattr__(self, item):
...         return getattr(self._cursor, item)
...
>>> def get_cursor():
...     return CursorProxy(connection.cursor())
...
>>> import petl as etl
>>> etl.todb(tbl, get_cursor, ...)
```

Note however that this does imply loading the entire table into memory as a list prior to inserting into the database.

---

`petl.io.db.appenddb` (*table, dbo, tablename, schema=None, commit=True*)

Load data into an existing database table via a DB-API 2.0 connection or cursor. As `petl.io.db.todb()` except that the database table will be appended, i.e., the new data will be inserted into the table, and any existing rows will remain.

## Excel .xls files (xlrd/xlwt)

---

**Note:** The following functions require `xlrd` and `xlwt` to be installed, e.g.:

```
$ pip install xlrd xlwt-future
```

---

`petl.io.xls.fromxls` (*filename, sheet=None, use\_view=True*)

Extract a table from a sheet in an Excel .xls file.

N.B., the sheet name is case sensitive.

`petl.io.xls.toxls` (*tbl, filename, sheet, encoding=None, style\_compression=0, styles=None*)

Write a table to a new Excel .xls file.

## Excel .xlsx files (openpyxl)

---

**Note:** The following functions require `openpyxl` to be installed, e.g.:

```
$ pip install openpyxl
```

---

`petl.io.xlsx.fromxlsx` (*filename, sheet=None, range\_string=None, row\_offset=0, column\_offset=0, \*\*kwargs*)

Extract a table from a sheet in an Excel .xlsx file.

N.B., the sheet name is case sensitive.

The *sheet* argument can be omitted, in which case the first sheet in the workbook is used by default.

The `range_string` argument can be used to provide a range string specifying a range of cells to extract.

The `row_offset` and `column_offset` arguments can be used to specify offsets.

Any other keyword arguments are passed through to `openpyxl.load_workbook()`.

`petl.io.xlsx.toxlsx(tbl, filename, sheet=None, encoding=None)`

Write a table to a new Excel .xlsx file.

## Arrays (NumPy)

**Note:** The following functions require `numpy` to be installed, e.g.:

```
$ pip install numpy
```

`petl.io.numpy.fromarray(a)`

Extract a table from a `numpy` structured array, e.g.:

```
>>> import petl as etl
>>> import numpy as np
>>> a = np.array([('apples', 1, 2.5),
...              ('oranges', 3, 4.4),
...              ('pears', 7, 0.1)],
...              dtype='U8, i4, f4')
>>> table = etl.fromarray(a)
>>> table
+-----+-----+-----+
| f0      | f1 | f2      |
+=====+=====+=====+
| 'apples' | 1  | 2.5     |
+-----+-----+-----+
| 'oranges' | 3  | 4.400001 |
+-----+-----+-----+
| 'pears'   | 7  | 0.1     |
+-----+-----+-----+
```

`petl.io.numpy.toarray(table, dtype=None, count=-1, sample=1000)`

Load data from the given `table` into a `numpy` structured array. E.g.:

```
>>> import petl as etl
>>> table = [('foo', 'bar', 'baz'),
...          ('apples', 1, 2.5),
...          ('oranges', 3, 4.4),
...          ('pears', 7, .1)]
>>> a = etl.toarray(table)
>>> a
array([('apples', 1, 2.5), ('oranges', 3, 4.4), ('pears', 7, 0.1)],
      dtype=(numpy.record, [('foo', '<U7'), ('bar', '<i8'), ('baz', '<f8')]))
>>> # the dtype can be specified as a string
... a = etl.toarray(table, dtype='a4, i2, f4')
>>> a
array([(b'app1', 1, 2.5), (b'oran', 3, 4.400000095367432),
      (b'pear', 7, 0.10000000149011612)],
      dtype=[('foo', 'S4'), ('bar', '<i2'), ('baz', '<f4')])
>>> # the dtype can also be partially specified
```

```
... a = etl.toarray(table, dtype={'foo': 'a4'})
>>> a
array([(b'appl', 1, 2.5), (b'oran', 3, 4.4), (b'pear', 7, 0.1)],
      dtype=[('foo', 'S4'), ('bar', '<i8'), ('baz', '<f8')])
```

If the dtype is not completely specified, *sample* rows will be examined to infer an appropriate dtype.

`petl.io.numpy.torecarray(*args, **kwargs)`

Convenient shorthand for `toarray(*args, **kwargs).view(np.recarray)`.

`petl.io.numpy.valuestoarray(vals, dtype=None, count=-1, sample=1000)`

Load values from a table column into a `numpy` array, e.g.:

```
>>> import petl as etl
>>> table = [('foo', 'bar', 'baz'),
...         ('apples', 1, 2.5),
...         ('oranges', 3, 4.4),
...         ('pears', 7, .1)]
>>> table = etl.wrap(table)
>>> table.values('bar').array()
array([1, 3, 7])
>>> # specify dtype
... table.values('bar').array(dtype='i4')
array([1, 3, 7], dtype=int32)
```

## DataFrames (pandas)

**Note:** The following functions require `pandas` to be installed, e.g.:

```
$ pip install pandas
```

`petl.io.pandas.fromdataframe(df, include_index=False)`

Extract a table from a `pandas` DataFrame. E.g.:

```
>>> import petl as etl
>>> import pandas as pd
>>> records = [('apples', 1, 2.5), ('oranges', 3, 4.4), ('pears', 7, 0.1)]
>>> df = pd.DataFrame.from_records(records, columns=('foo', 'bar', 'baz'))
>>> table = etl.fromdataframe(df)
>>> table
+-----+-----+-----+
| foo      | bar | baz |
+-----+-----+-----+
| 'apples' |  1 | 2.5 |
+-----+-----+-----+
| 'oranges' |  3 | 4.4 |
+-----+-----+-----+
| 'pears'   |  7 | 0.1 |
+-----+-----+-----+
```

`petl.io.pandas.todataframe(table, index=None, exclude=None, columns=None, coerce_float=False, nrows=None)`

Load data from the given *table* into a `pandas` DataFrame. E.g.:

```

>>> import petl as etl
>>> table = [('foo', 'bar', 'baz'),
...         ('apples', 1, 2.5),
...         ('oranges', 3, 4.4),
...         ('pears', 7, .1)]
>>> df = etl.todataframe(table)
>>> df
   foo bar baz
0  apples  1  2.5
1  oranges  3  4.4
2   pears  7  0.1

```

## HDF5 files (PyTables)

**Note:** The following functions require PyTables to be installed, e.g.:

```

$ # install HDF5
$ apt-get install libhdf5-7 libhdf5-dev
$ # install other prerequisites
$ pip install cython
$ pip install numpy
$ pip install numexpr
$ # install PyTables
$ pip install tables

```

`petl.io.pytables.fromhdf5` (*source, where=None, name=None, condition=None, condvars=None, start=None, stop=None, step=None*)

Provides access to an HDF5 table. E.g.:

```

>>> import petl as etl
>>> import tables
>>> # set up a new hdf5 table to demonstrate with
... h5file = tables.openFile('example.h5', mode='w',
...                          title='Example file')
>>> h5file.createGroup('/', 'testgroup', 'Test Group')
/testgroup (Group) 'Test Group'
  children := []
>>> class FooBar(tables.IsDescription):
...     foo = tables.Int32Col(pos=0)
...     bar = tables.StringCol(6, pos=2)
...
>>> h5table = h5file.createTable('/testgroup', 'testtable', FooBar,
...                              'Test Table')
>>> # load some data into the table
... table1 = (('foo', 'bar'),
...           (1, b'asdfgh'),
...           (2, b'qwerty'),
...           (3, b'zxcvbn'))
>>> for row in table1[1:]:
...     for i, f in enumerate(table1[0]):
...         h5table.row[f] = row[i]
...     h5table.row.append()
...

```

```

>>> h5file.flush()
>>> h5file.close()
>>> #
... # now demonstrate use of fromhdf5
... table1 = etl.fromhdf5('example.h5', '/testgroup', 'testtable')
>>> table1
+-----+-----+
| foo | bar      |
+=====+=====+
|  1 | b'asdfgh'|
+-----+-----+
|  2 | b'qwerty'|
+-----+-----+
|  3 | b'zxcvbn'|
+-----+-----+

>>> # alternatively just specify path to table node
... table1 = etl.fromhdf5('example.h5', '/testgroup/testtable')
>>> # ...or use an existing tables.File object
... h5file = tables.openFile('example.h5')
>>> table1 = etl.fromhdf5(h5file, '/testgroup/testtable')
>>> # ...or use an existing tables.Table object
... h5tbl = h5file.getNode('/testgroup/testtable')
>>> table1 = etl.fromhdf5(h5tbl)
>>> # use a condition to filter data
... table2 = etl.fromhdf5(h5tbl, condition='foo < 3')
>>> table2
+-----+-----+
| foo | bar      |
+=====+=====+
|  1 | b'asdfgh'|
+-----+-----+
|  2 | b'qwerty'|
+-----+-----+

>>> h5file.close()

```

`petl.io.pytables.fromhdf5sorted`(*source*, *where=None*, *name=None*, *sortby=None*, *checkCSI=False*, *start=None*, *stop=None*, *step=None*)

Provides access to an HDF5 table, sorted by an indexed column, e.g.:

```

>>> import petl as etl
>>> import tables
>>> # set up a new hdf5 table to demonstrate with
... h5file = tables.openFile('example.h5', mode='w', title='Test file')
>>> h5file.createGroup('/', 'testgroup', 'Test Group')
/testgroup (Group) 'Test Group'
  children := []
>>> class FooBar(tables.IsDescription):
...     foo = tables.Int32Col(pos=0)
...     bar = tables.StringCol(6, pos=2)
...
>>> h5table = h5file.createTable('/testgroup', 'testtable', FooBar, 'Test Table')
>>> # load some data into the table
... table1 = (('foo', 'bar'),
...           (3, b'asdfgh'),
...           (2, b'qwerty'),
...           (1, b'zxcvbn'))

```



```

>>> for row in table1[1:]:
...     for i, f in enumerate(table1[0]):
...         h5table.row[f] = row[i]
...         h5table.row.append()
...
>>> h5table.cols.foo.createCSIndex() # CS index is required
0
>>> h5file.flush()
>>> h5file.close()
>>> #
... # access the data, sorted by the indexed column
... table2 = etl.fromhdf5sorted('example.h5', '/testgroup', 'testtable',
...                             sortby='foo')
>>> table2
+-----+-----+
| foo | bar      |
+=====+=====+
|  1 | b'zxcvbn'|
+-----+-----+
|  2 | b'qwerty'|
+-----+-----+
|  3 | b'asdfgh'|
+-----+-----+

```

`petl.io.pytables.tohdf5`(*table*, *source*, *where=None*, *name=None*, *create=False*, *drop=False*, *description=None*, *title=''*, *filters=None*, *expectedrows=10000*, *chunkshape=None*, *byteorder=None*, *createparents=False*, *sample=1000*)

Write to an HDF5 table. If *create* is *False*, assumes the table already exists, and attempts to truncate it before loading. If *create* is *True*, a new table will be created, and if *drop* is *True*, any existing table will be dropped first. If *description* is *None*, the description will be guessed. E.g.:

```

>>> import petl as etl
>>> table1 = (('foo', 'bar'),
...          (1, b'asdfgh'),
...          (2, b'qwerty'),
...          (3, b'zxcvbn'))
>>> etl.tohdf5(table1, 'example.h5', '/testgroup', 'testtable',
...            drop=True, create=True, createparents=True)
>>> etl.fromhdf5('example.h5', '/testgroup', 'testtable')
+-----+-----+
| foo | bar      |
+=====+=====+
|  1 | b'asdfgh'|
+-----+-----+
|  2 | b'qwerty'|
+-----+-----+
|  3 | b'zxcvbn'|
+-----+-----+

```

`petl.io.pytables.appendhdf5`(*table*, *source*, *where=None*, *name=None*)

As `petl.io.hdf5.tohdf5()` but don't truncate the target table before loading.

## Bcolz ctables

**Note:** The following functions require `bcolz` to be installed, e.g.:

```
$ pip install bcolz
```

`petl.io.bcolz.frombcolz` (*source, expression=None, outcols=None, limit=None, skip=0*)  
 Extract a table from a bcolz ctable, e.g.:

```
>>> import petl as etl
>>> import bcolz
>>> cols = [
...     ['apples', 'oranges', 'pears'],
...     [1, 3, 7],
...     [2.5, 4.4, .1]
... ]
>>> names = ('foo', 'bar', 'baz')
>>> ctbl = bcolz.ctable(cols, names=names)
>>> tbl = etl.frombcolz(ctbl)
>>> tbl
+-----+-----+-----+
| foo      | bar | baz |
+=====+=====+=====+
| 'apples' |  1 | 2.5 |
+-----+-----+-----+
| 'oranges' |  3 | 4.4 |
+-----+-----+-----+
| 'pears'   |  7 | 0.1 |
+-----+-----+-----+
```

If *expression* is provided it will be executed by bcolz and only matching rows returned, e.g.:

```
>>> tbl2 = etl.frombcolz(ctbl, expression='bar > 1')
>>> tbl2
+-----+-----+-----+
| foo      | bar | baz |
+=====+=====+=====+
| 'oranges' |  3 | 4.4 |
+-----+-----+-----+
| 'pears'   |  7 | 0.1 |
+-----+-----+-----+
```

New in version 1.1.0.

`petl.io.bcolz.tobcolz` (*table, dtype=None, sample=1000, \*\*kwargs*)  
 Load data into a bcolz ctable, e.g.:

```
>>> import petl as etl
>>> table = [('foo', 'bar', 'baz'),
...         ('apples', 1, 2.5),
...         ('oranges', 3, 4.4),
...         ('pears', 7, .1)]
>>> ctbl = etl.tobcolz(table)
>>> ctbl
ctable((3,), [('foo', '<U7'), ('bar', '<i8'), ('baz', '<f8')])
  nbytes: 132; cbytes: 1023.98 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
[('apples', 1, 2.5) ('oranges', 3, 4.4) ('pears', 7, 0.1)]
>>> ctbl.names
['foo', 'bar', 'baz']
>>> ctbl['foo']
```

```
carray((3,), <U7)
  nbytes: 84; cbytes: 511.98 KB; ratio: 0.00
  cparams := cparams(clevel=5, shuffle=True, cname='blosclz')
['apples' 'oranges' 'pears']
```

Other keyword arguments are passed through to the ctable constructor.

New in version 1.1.0.

`petl.io.bcolz.appendbcolz` (*table, obj, check\_names=True*)

Append data into a bcolz ctable. The *obj* argument can be either an existing ctable or the name of a directory where an on-disk ctable is stored.

New in version 1.1.0.

## Text indexes (Whoosh)

**Note:** The following functions require [Whoosh](#) to be installed, e.g.:

```
$ pip install whoosh
```

`petl.io.whoosh.fromtextindex` (*index\_or\_dirname, indexname=None, docnum\_field=None*)

Extract all documents from a Whoosh index. E.g.:

```
>>> import petl as etl
>>> import os
>>> # set up an index and load some documents via the Whoosh API
... from whoosh.index import create_in
>>> from whoosh.fields import *
>>> schema = Schema(title=TEXT(stored=True), path=ID(stored=True),
...                 content=TEXT)
>>> dirname = 'example.whoosh'
>>> if not os.path.exists(dirname):
...     os.mkdir(dirname)
...
>>> index = create_in(dirname, schema)
>>> writer = index.writer()
>>> writer.add_document(title=u"First document", path=u"/a",
...                    content=u"This is the first document we've added!")
>>> writer.add_document(title=u"Second document", path=u"/b",
...                    content=u"The second one is even more interesting!")
>>> writer.commit()
>>> # extract documents as a table
... table = etl.fromtextindex(dirname)
>>> table
+-----+-----+
| path | title |
+=====+=====+
| '/a' | 'First document' |
+-----+-----+
| '/b' | 'Second document' |
+-----+-----+
```

Keyword arguments:

**index\_or\_dirname** Either an instance of *whoosh.index.Index* or a string containing the directory path where the index is stored.

**indexname** String containing the name of the index, if multiple indexes are stored in the same directory.

**docnum\_field** If not None, an extra field will be added to the output table containing the internal document number stored in the index. The name of the field will be the value of this argument.

```
petl.io.whoosh.searchtextindex(index_or_dirname, query, limit=10, indexname=None,
                                docnum_field=None, score_field=None, fieldboosts=None,
                                search_kwargs=None)
```

Search a Whoosh index using a query. E.g.:

```
>>> import petl as etl
>>> import os
>>> # set up an index and load some documents via the Whoosh API
... from whoosh.index import create_in
>>> from whoosh.fields import *
>>> schema = Schema(title=TEXT(stored=True), path=ID(stored=True),
...                 content=TEXT)
>>> dirname = 'example.whoosh'
>>> if not os.path.exists(dirname):
...     os.mkdir(dirname)
...
>>> index = create_in('example.whoosh', schema)
>>> writer = index.writer()
>>> writer.add_document(title=u"Oranges", path=u"/a",
...                    content=u"This is the first document we've added!")
>>> writer.add_document(title=u"Apples", path=u"/b",
...                    content=u"The second document is even more "
...                            u"interesting!")
>>> writer.commit()
>>> # demonstrate the use of searchtextindex()
... table1 = etl.searchtextindex('example.whoosh', 'oranges')
>>> table1
+-----+-----+
| path | title |
+-----+-----+
| '/a' | 'Oranges' |
+-----+-----+

>>> table2 = etl.searchtextindex('example.whoosh', 'doc*')
>>> table2
+-----+-----+
| path | title |
+-----+-----+
| '/a' | 'Oranges' |
+-----+-----+
| '/b' | 'Apples' |
+-----+-----+
```

Keyword arguments:

**index\_or\_dirname** Either an instance of *whoosh.index.Index* or a string containing the directory path where the index is to be stored.

**query** Either a string or an instance of *whoosh.query.Query*. If a string, it will be parsed as a multi-field query, i.e., any terms not bound to a specific field will match **any** field.

**limit** Return at most *limit* results.

**indexname** String containing the name of the index, if multiple indexes are stored in the same directory.

**docnum\_field** If not None, an extra field will be added to the output table containing the internal document number stored in the index. The name of the field will be the value of this argument.

**score\_field** If not None, an extra field will be added to the output table containing the score of the result. The name of the field will be the value of this argument.

**fieldboosts** An optional dictionary mapping field names to boosts.

**search\_kwargs** Any extra keyword arguments to be passed through to the Whoosh *search()* method.

```
petl.io.whoosh.searchtextindexpage(index_or_dirname, query, pagenum, pagelen=10, indexname=None, docnum_field=None, score_field=None, fieldboosts=None, search_kwargs=None)
```

Search an index using a query, returning a result page.

Keyword arguments:

**index\_or\_dirname** Either an instance of *whoosh.index.Index* or a string containing the directory path where the index is to be stored.

**query** Either a string or an instance of *whoosh.query.Query*. If a string, it will be parsed as a multi-field query, i.e., any terms not bound to a specific field will match **any** field.

**pagenum** Number of the page to return (e.g., 1 = first page).

**pagelen** Number of results per page.

**indexname** String containing the name of the index, if multiple indexes are stored in the same directory.

**docnum\_field** If not None, an extra field will be added to the output table containing the internal document number stored in the index. The name of the field will be the value of this argument.

**score\_field** If not None, an extra field will be added to the output table containing the score of the result. The name of the field will be the value of this argument.

**fieldboosts** An optional dictionary mapping field names to boosts.

**search\_kwargs** Any extra keyword arguments to be passed through to the Whoosh *search()* method.

```
petl.io.whoosh.totextindex(table, index_or_dirname, schema=None, indexname=None, merge=False, optimize=False)
```

Load all rows from *table* into a Whoosh index. N.B., this will clear any existing data in the index before loading. E.g.:

```
>>> import petl as etl
>>> import datetime
>>> import os
>>> # here is the table we want to load into an index
... table = (('f0', 'f1', 'f2', 'f3', 'f4'),
...          ('AAA', 12, 4.3, True, datetime.datetime.now()),
...          ('BBB', 6, 3.4, False, datetime.datetime(1900, 1, 31)),
...          ('CCC', 42, 7.8, True, datetime.datetime(2100, 12, 25)))
>>> # define a schema for the index
... from whoosh.fields import *
>>> schema = Schema(f0=TEXT(stored=True),
...                 f1=NUMERIC(int, stored=True),
...                 f2=NUMERIC(float, stored=True),
...                 f3=BOOLEAN(stored=True),
...                 f4=DATETIME(stored=True))
>>> # load index
... dirname = 'example.whoosh'
>>> if not os.path.exists(dirname):
```

```
...     os.mkdir(dirname)
...
>>> etl.totextindex(table, dirname, schema=schema)
```

Keyword arguments:

**table** A table container with the data to be loaded.

**index\_or\_dirname** Either an instance of *whoosh.index.Index* or a string containing the directory path where the index is to be stored.

**schema** Index schema to use if creating the index.

**indexname** String containing the name of the index, if multiple indexes are stored in the same directory.

**merge** Merge small segments during commit?

**optimize** Merge all segments together?

`petl.io.whoosh.appendtextindex` (*table*, *index\_or\_dirname*, *indexname=None*, *merge=True*, *optimize=False*)

Load all rows from *table* into a Whoosh index, adding them to any existing data in the index.

Keyword arguments:

**table** A table container with the data to be loaded.

**index\_or\_dirname** Either an instance of *whoosh.index.Index* or a string containing the directory path where the index is to be stored.

**indexname** String containing the name of the index, if multiple indexes are stored in the same directory.

**merge** Merge small segments during commit?

**optimize** Merge all segments together?

## I/O helper classes

The following classes are helpers for extract (`from...()`) and load (`to...()`) functions that use a file-like data source.

An instance of any of the following classes can be used as the `source` argument to data extraction functions like `petl.io.csv.fromcsv()` etc., with the exception of `petl.io.sources.StdoutSource` which is write-only.

An instance of any of the following classes can also be used as the `source` argument to data loading functions like `petl.io.csv.tocsv()` etc., with the exception of `petl.io.sources.StdinSource`, `petl.io.sources.URLSource` and `petl.io.sources.PopenSource` which are read-only.

The behaviour of each source can usually be configured by passing arguments to the constructor, see the source code of the `petl.io.sources` module for full details.

```
class petl.io.sources.FileSource(filename, **kwargs)
```

```
class petl.io.sources.GzipSource(filename, **kwargs)
```

```
class petl.io.sources.BZ2Source(filename, **kwargs)
```

```
class petl.io.sources.ZipSource(filename, membername, pwd=None, **kwargs)
```

```
class petl.io.sources.StdinSource
```

```
class petl.io.sources.StdoutSource
```

**class** petl.io.sources.**URLSource** (\*args, \*\*kwargs)

**class** petl.io.sources.**MemorySource** (s=None)

Memory data source. E.g.:

```
>>> import petl as etl
>>> data = b'foo,bar\na,1\nb,2\nc,2\n'
>>> source = etl.MemorySource(data)
>>> tbl = etl.fromcsv(source)
>>> tbl
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | '1' |
+-----+-----+
| 'b' | '2' |
+-----+-----+
| 'c' | '2' |
+-----+-----+

>>> sink = etl.MemorySource()
>>> tbl.tojson(sink)
>>> sink.getvalue()
b'{"foo": "a", "bar": "1"}, {"foo": "b", "bar": "2"}, {"foo": "c", "bar": "2"}'
```

Also supports appending.

**class** petl.io.sources.**PopenSource** (\*args, \*\*kwargs)





---

## Transform - transforming tables

---

### Basic transformations

`petl.transform.basics.head(table, n=5)`

Select the first  $n$  data rows. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 5],
...          ['d', 7],
...          ['f', 42],
...          ['f', 3],
...          ['h', 90]]
>>> table2 = etl.head(table1, 4)
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' |  1 |
+-----+-----+
| 'b' |  2 |
+-----+-----+
| 'c' |  5 |
+-----+-----+
| 'd' |  7 |
+-----+-----+
```

See also `petl.transform.basics.tail()`, `petl.transform.basics.rowslice()`.

`petl.transform.basics.tail(table, n=5)`

Select the last  $n$  data rows. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 5],
...          ['d', 7],
...          ['f', 42],
...          ['f', 3],
...          ['h', 90],
...          ['k', 12],
...          ['l', 77],
...          ['q', 2]]
>>> table2 = etl.tail(table1, 4)
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'h' | 90 |
+-----+-----+
| 'k' | 12 |
+-----+-----+
| 'l' | 77 |
+-----+-----+
| 'q' | 2 |
+-----+-----+

```

See also `petl.transform.basics.head()`, `petl.transform.basics.rowslice()`.

`petl.transform.basics.rowslice` (*table*, \**sliceargs*)

Choose a subsequence of data rows. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['c', 5],
...          ['d', 7],
...          ['f', 42]]
>>> table2 = etl.rowslice(table1, 2)
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+

>>> table3 = etl.rowslice(table1, 1, 4)
>>> table3
+-----+-----+
| foo | bar |
+=====+=====+
| 'b' | 2 |
+-----+-----+
| 'c' | 5 |
+-----+-----+
| 'd' | 7 |

```

```

+-----+-----+
>>> table4 = etl.rowslice(table1, 0, 5, 2)
>>> table4
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'c' | 5 |
+-----+-----+
| 'f' | 42 |
+-----+-----+

```

Positional arguments are used to slice the data rows. The *sliceargs* are passed through to `itertools.islice()`.

See also `petl.transform.basics.head()`, `petl.transform.basics.tail()`.

`petl.transform.basics.cut` (*table*, \**args*, \*\**kwargs*)

Choose and/or re-order fields. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2.7],
...          ['B', 2, 3.4],
...          ['B', 3, 7.8],
...          ['D', 42, 9.0],
...          ['E', 12]]
>>> table2 = etl.cut(table1, 'foo', 'baz')
>>> table2
+-----+-----+
| foo | baz |
+=====+=====+
| 'A' | 2.7 |
+-----+-----+
| 'B' | 3.4 |
+-----+-----+
| 'B' | 7.8 |
+-----+-----+
| 'D' | 9.0 |
+-----+-----+
| 'E' | None |
+-----+-----+

>>> # fields can also be specified by index, starting from zero
... table3 = etl.cut(table1, 0, 2)
>>> table3
+-----+-----+
| foo | baz |
+=====+=====+
| 'A' | 2.7 |
+-----+-----+
| 'B' | 3.4 |
+-----+-----+
| 'B' | 7.8 |
+-----+-----+
| 'D' | 9.0 |

```

```

+-----+-----+
| 'E' | None |
+-----+-----+

>>> # field names and indices can be mixed
... table4 = etl.cut(table1, 'bar', 0)
>>> table4
+-----+-----+
| bar | foo |
+=====+=====+
| 1 | 'A' |
+-----+-----+
| 2 | 'B' |
+-----+-----+
| 3 | 'B' |
+-----+-----+
| 42 | 'D' |
+-----+-----+
| 12 | 'E' |
+-----+-----+

>>> # select a range of fields
... table5 = etl.cut(table1, *range(0, 2))
>>> table5
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 1 |
+-----+-----+
| 'B' | 2 |
+-----+-----+
| 'B' | 3 |
+-----+-----+
| 'D' | 42 |
+-----+-----+
| 'E' | 12 |
+-----+-----+

```

Note that any short rows will be padded with *None* values (or whatever is provided via the *missing* keyword argument).

See also `petl.transform.basics.cutout()`.

`petl.transform.basics.cutout` (*table*, \**args*, \*\**kwargs*)

Remove fields. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2.7],
...          ['B', 2, 3.4],
...          ['B', 3, 7.8],
...          ['D', 42, 9.0],
...          ['E', 12]]
>>> table2 = etl.cutout(table1, 'bar')
>>> table2
+-----+-----+
| foo | baz |
+=====+=====+

```

```

| 'A' | 2.7 |
+-----+-----+
| 'B' | 3.4 |
+-----+-----+
| 'B' | 7.8 |
+-----+-----+
| 'D' | 9.0 |
+-----+-----+
| 'E' | None |
+-----+-----+

```

See also `petl.transform.basics.cut()`.

`petl.transform.basics.movefield(table, field, index)`

Move a field to a new position.

`petl.transform.basics.cat(*tables, **kwargs)`

Concatenate tables. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          [1, 'A'],
...          [2, 'B']]
>>> table2 = [['bar', 'baz'],
...          ['C', True],
...          ['D', False]]
>>> table3 = etl.cat(table1, table2)
>>> table3
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'A' | None |
+-----+-----+-----+
| 2 | 'B' | None |
+-----+-----+-----+
| None | 'C' | True |
+-----+-----+-----+
| None | 'D' | False |
+-----+-----+-----+

>>> # can also be used to square up a single table with uneven rows
... table4 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2],
...          ['B', '2', '3.4'],
...          [u'B', u'3', u'7.8', True],
...          ['D', 'xyz', 9.0],
...          ['E', None]]
>>> table5 = etl.cat(table4)
>>> table5
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | 2 |
+-----+-----+-----+
| 'B' | '2' | '3.4' |
+-----+-----+-----+
| 'B' | '3' | '7.8' |
+-----+-----+-----+

```

```

| 'D' | 'xyz' | 9.0 |
+-----+-----+-----+
| 'E' | None | None |
+-----+-----+-----+

>>> # use the header keyword argument to specify a fixed set of fields
... table6 = [['bar', 'foo'],
...           ['A', 1],
...           ['B', 2]]
>>> table7 = etl.cat(table6, header=['A', 'foo', 'B', 'bar', 'C'])
>>> table7
+-----+-----+-----+-----+-----+
| A      | foo | B      | bar | C      |
+-----+-----+-----+-----+-----+
| None   | 1   | None   | 'A' | None   |
+-----+-----+-----+-----+-----+
| None   | 2   | None   | 'B' | None   |
+-----+-----+-----+-----+-----+

>>> # using the header keyword argument with two input tables
... table8 = [['bar', 'foo'],
...           ['A', 1],
...           ['B', 2]]
>>> table9 = [['bar', 'baz'],
...           ['C', True],
...           ['D', False]]
>>> table10 = etl.cat(table8, table9, header=['A', 'foo', 'B', 'bar', 'C'])
>>> table10
+-----+-----+-----+-----+-----+
| A      | foo | B      | bar | C      |
+-----+-----+-----+-----+-----+
| None   | 1   | None   | 'A' | None   |
+-----+-----+-----+-----+-----+
| None   | 2   | None   | 'B' | None   |
+-----+-----+-----+-----+-----+
| None   | None | None   | 'C' | None   |
+-----+-----+-----+-----+-----+
| None   | None | None   | 'D' | None   |
+-----+-----+-----+-----+-----+

```

Note that the tables do not need to share exactly the same fields, any missing fields will be padded with *None* or whatever is provided via the *missing* keyword argument.

Note that this function can be used with a single table argument, in which case it has the effect of ensuring all data rows are the same length as the header row, truncating any long rows and padding any short rows with the value of the *missing* keyword argument.

By default, the fields for the output table will be determined as the union of all fields found in the input tables. Use the *header* keyword argument to override this behaviour and specify a fixed set of fields for the output table.

`petl.transform.basics.stack(*tables, **kwargs)`

Concatenate tables, without trying to match headers. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          [1, 'A'],
...          [2, 'B']]
>>> table2 = [['bar', 'baz'],
...          ['C', True],

```

```

...         ['D', False]]
>>> table3 = etl.stack(table1, table2)
>>> table3
+-----+-----+
| foo | bar |
+=====+=====+
| 1 | 'A' |
+-----+-----+
| 2 | 'B' |
+-----+-----+
| 'C' | True |
+-----+-----+
| 'D' | False |
+-----+-----+

>>> # can also be used to square up a single table with uneven rows
... table4 = [['foo', 'bar', 'baz'],
...           ['A', 1, 2],
...           ['B', '2', '3.4'],
...           [u'B', u'3', u'7.8', True],
...           ['D', 'xyz', 9.0],
...           ['E', None]]
>>> table5 = etl.stack(table4)
>>> table5
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | 2 |
+-----+-----+-----+
| 'B' | '2' | '3.4' |
+-----+-----+-----+
| 'B' | '3' | '7.8' |
+-----+-----+-----+
| 'D' | 'xyz' | 9.0 |
+-----+-----+-----+
| 'E' | None | None |
+-----+-----+-----+

```

Similar to `petl.transform.basics.cat()` except that no attempt is made to align fields from different tables. Data rows are simply emitted in order, trimmed or padded to the length of the header row from the first table.

New in version 1.1.0.

`petl.transform.basics.skipcomments` (*table*, *prefix*)

Skip any row where the first value is a string and starts with *prefix*. E.g.:

```

>>> import petl as etl
>>> table1 = [['##aaa', 'bbb', 'ccc'],
...           ['##mmm'],
...           ['#foo', 'bar'],
...           ['##nnn', 1],
...           ['a', 1],
...           ['b', 2]]
>>> table2 = etl.skipcomments(table1, '##')
>>> table2
+-----+-----+
| #foo | bar |

```

```

+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+

```

Use the *prefix* parameter to determine which string to consider as indicating a comment.

`petl.transform.basics.addfield` (*table*, *field*, *value=None*, *index=None*, *missing=None*)

Add a field with a fixed or calculated value. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['M', 12],
...          ['F', 34],
...          ['- ', 56]]
>>> # using a fixed value
... table2 = etl.addfield(table1, 'baz', 42)
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'M' | 12 | 42 |
+-----+-----+-----+
| 'F' | 34 | 42 |
+-----+-----+-----+
| '- ' | 56 | 42 |
+-----+-----+-----+

>>> # calculating the value
... table2 = etl.addfield(table1, 'baz', lambda rec: rec['bar'] * 2)
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'M' | 12 | 24 |
+-----+-----+-----+
| 'F' | 34 | 68 |
+-----+-----+-----+
| '- ' | 56 | 112 |
+-----+-----+-----+

```

Use the *index* parameter to control the position of the inserted field.

`petl.transform.basics.addcolumn` (*table*, *field*, *col*, *index=None*, *missing=None*)

Add a column of data to the table. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 1],
...          ['B', 2]]
>>> col = [True, False]
>>> table2 = etl.addcolumn(table1, 'baz', col)
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | True |

```



```
+-----+-----+-----+
| 'B' | 2 | False |
+-----+-----+-----+
```

Use the *index* parameter to control the position of the new column.

`petl.transform.basics.addrownumbers` (*table*, *start=1*, *step=1*, *field='row'*)  
Add a field of row numbers. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 9],
...          ['C', 2],
...          ['F', 1]]
>>> table2 = etl.addrownumbers(table1)
>>> table2
+-----+-----+-----+
| row | foo | bar |
+=====+=====+=====+
| 1 | 'A' | 9 |
+-----+-----+-----+
| 2 | 'C' | 2 |
+-----+-----+-----+
| 3 | 'F' | 1 |
+-----+-----+-----+
```

Parameters *start* and *step* control the numbering.

`petl.transform.basics.addfieldusingcontext` (*table*, *field*, *query*)

Like `petl.transform.basics.addfield()` but the *query* function is passed the previous, current and next rows, so values may be calculated based on data in adjacent rows. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 1],
...          ['B', 4],
...          ['C', 5],
...          ['D', 9]]
>>> def upstream(prv, cur, nxt):
...     if prv is None:
...         return None
...     else:
...         return cur.bar - prv.bar
...
>>> def downstream(prv, cur, nxt):
...     if nxt is None:
...         return None
...     else:
...         return nxt.bar - cur.bar
...
>>> table2 = etl.addfieldusingcontext(table1, 'baz', upstream)
>>> table3 = etl.addfieldusingcontext(table2, 'quux', downstream)
>>> table3
+-----+-----+-----+-----+
| foo | bar | baz | quux |
+=====+=====+=====+=====+
| 'A' | 1 | None | 3 |
+-----+-----+-----+-----+
```

```

| 'B' | 4 | 3 | 1 |
+-----+-----+-----+-----+
| 'C' | 5 | 1 | 4 |
+-----+-----+-----+-----+
| 'D' | 9 | 4 | None |
+-----+-----+-----+-----+

```

The *field* parameter is the name of the field to be added. The *query* parameter is a function operating on the current, previous and next rows and returning the value.

`petl.transform.basics.annex(*tables, **kwargs)`

Join two or more tables by row order. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 9],
...          ['C', 2],
...          ['F', 1]]
>>> table2 = [['foo', 'baz'],
...          ['B', 3],
...          ['D', 10]]
>>> table3 = etl.annex(table1, table2)
>>> table3
+-----+-----+-----+-----+
| foo | bar | foo | baz |
+=====+=====+=====+=====+
| 'A' | 9 | 'B' | 3 |
+-----+-----+-----+-----+
| 'C' | 2 | 'D' | 10 |
+-----+-----+-----+-----+
| 'F' | 1 | None | None |
+-----+-----+-----+-----+

```

See also `petl.transform.joins.join()`.

## Header manipulations

`petl.transform.headers.rename(table, *args, **kwargs)`

Replace one or more values in the table's header row. E.g.:

```

>>> import petl as etl
>>> table1 = [['sex', 'age'],
...          ['m', 12],
...          ['f', 34],
...          ['- ', 56]]
>>> # rename a single field
... table2 = etl.rename(table1, 'sex', 'gender')
>>> table2
+-----+-----+
| gender | age |
+=====+=====+
| 'm'    | 12 |
+-----+-----+
| 'f'    | 34 |
+-----+-----+
| '- '   | 56 |

```

```

+-----+-----+
>>> # rename multiple fields by passing dictionary as second argument
... table3 = etl.rename(table1, {'sex': 'gender', 'age': 'age_years'})
>>> table3
+-----+-----+
| gender | age_years |
+=====+=====+
| 'm'    | 12       |
+-----+-----+
| 'f'    | 34       |
+-----+-----+
| '-'    | 56       |
+-----+-----+

```

The field to rename can be specified as an index (i.e., integer representing field position).

If any nonexistent fields are specified, the default behaviour is to raise a *FieldSelectionError*. However, if *strict* keyword argument is *False*, any nonexistent fields specified will be silently ignored.

`petl.transform.headers.setheader(table, header)`

Replace header row in the given table. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2]]
>>> table2 = etl.setheader(table1, ['foofoo', 'barbar'])
>>> table2
+-----+-----+
| foofoo | barbar |
+=====+=====+
| 'a'    | 1      |
+-----+-----+
| 'b'    | 2      |
+-----+-----+

```

See also `petl.transform.headers.extendheader()`, `petl.transform.headers.pushheader()`.

`petl.transform.headers.extendheader(table, fields)`

Extend header row in the given table. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo'],
...          ['a', 1, True],
...          ['b', 2, False]]
>>> table2 = etl.extendheader(table1, ['bar', 'baz'])
>>> table2
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'a' | 1  | True |
+-----+-----+-----+
| 'b' | 2  | False|
+-----+-----+-----+

```

See also `petl.transform.headers.setheader()`, `petl.transform.headers.pushheader()`.

`petl.transform.headers.pushheader` (*table*, *header*, \**args*)  
Push rows down and prepend a header row. E.g.:

```
>>> import petl as etl
>>> table1 = [['a', 1],
...          ['b', 2]]
>>> table2 = etl.pushheader(table1, ['foo', 'bar'])
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
```

The header row can either be a list or positional arguments.

`petl.transform.headers.prefixheader` (*table*, *prefix*)  
Prefix all fields in the table header.

`petl.transform.headers.suffixheader` (*table*, *suffix*)  
Suffix all fields in the table header.

`petl.transform.headers.sortheader` (*table*, *reverse=False*, *missing=None*)  
Re-order columns so the header is sorted.

New in version 1.1.0.

`petl.transform.headers.skip` (*table*, *n*)  
Skip *n* rows, including the header row. E.g.:

```
>>> import petl as etl
>>> table1 = ['#aaa', 'bbb', 'ccc'],
...          ['#mmm'],
...          ['foo', 'bar'],
...          ['a', 1],
...          ['b', 2]]
>>> table2 = etl.skip(table1, 2)
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
| 'b' | 2 |
+-----+-----+
```

See also `petl.transform.basics.skipcomments()`.

## Converting values

`petl.transform.conversions.convert` (*table*, \**args*, \*\**kwargs*)  
Transform values under one or more fields via arbitrary functions, method invocations or dictionary translations.  
E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', '2.4', 12],
...          ['B', '5.7', 34],
...          ['C', '1.2', 56]]
>>> # using a built-in function:
... table2 = etl.convert(table1, 'bar', float)
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 2.4 | 12 |
+-----+-----+-----+
| 'B' | 5.7 | 34 |
+-----+-----+-----+
| 'C' | 1.2 | 56 |
+-----+-----+-----+

>>> # using a lambda function::
... table3 = etl.convert(table1, 'baz', lambda v: v*2)
>>> table3
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | '2.4' | 24 |
+-----+-----+-----+
| 'B' | '5.7' | 68 |
+-----+-----+-----+
| 'C' | '1.2' | 112 |
+-----+-----+-----+

>>> # a method of the data value can also be invoked by passing
... # the method name
... table4 = etl.convert(table1, 'foo', 'lower')
>>> table4
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | '2.4' | 12 |
+-----+-----+-----+
| 'b' | '5.7' | 34 |
+-----+-----+-----+
| 'c' | '1.2' | 56 |
+-----+-----+-----+

>>> # arguments to the method invocation can also be given
... table5 = etl.convert(table1, 'foo', 'replace', 'A', 'AA')
>>> table5
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'AA' | '2.4' | 12 |
+-----+-----+-----+
| 'B' | '5.7' | 34 |
+-----+-----+-----+
| 'C' | '1.2' | 56 |
+-----+-----+-----+

```

```

>>> # values can also be translated via a dictionary
... table7 = etl.convert(table1, 'foo', {'A': 'Z', 'B': 'Y'})
>>> table7
+-----+-----+-----+
| foo | bar  | baz |
+=====+=====+=====+
| 'Z' | '2.4' | 12 |
+-----+-----+-----+
| 'Y' | '5.7' | 34 |
+-----+-----+-----+
| 'C' | '1.2' | 56 |
+-----+-----+-----+

>>> # the same conversion can be applied to multiple fields
... table8 = etl.convert(table1, ('foo', 'bar', 'baz'), str)
>>> table8
+-----+-----+-----+
| foo | bar  | baz |
+=====+=====+=====+
| 'A' | '2.4' | '12' |
+-----+-----+-----+
| 'B' | '5.7' | '34' |
+-----+-----+-----+
| 'C' | '1.2' | '56' |
+-----+-----+-----+

>>> # multiple conversions can be specified at the same time
... table9 = etl.convert(table1, {'foo': 'lower',
...                               'bar': float,
...                               'baz': lambda v: v * 2})
>>> table9
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 2.4 | 24 |
+-----+-----+-----+
| 'b' | 5.7 | 68 |
+-----+-----+-----+
| 'c' | 1.2 | 112 |
+-----+-----+-----+

>>> # ...or alternatively via a list
... table10 = etl.convert(table1, ['lower', float, lambda v: v*2])
>>> table10
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 2.4 | 24 |
+-----+-----+-----+
| 'b' | 5.7 | 68 |
+-----+-----+-----+
| 'c' | 1.2 | 112 |
+-----+-----+-----+

>>> # conversion can be conditional
... table11 = etl.convert(table1, 'baz', lambda v: v * 2,
...                        where=lambda r: r.foo == 'B')
>>> table11

```

```

+-----+-----+-----+
| foo | bar  | baz |
+=====+=====+=====+
| 'A' | '2.4' | 12 |
+-----+-----+-----+
| 'B' | '5.7' | 68 |
+-----+-----+-----+
| 'C' | '1.2' | 56 |
+-----+-----+-----+

>>> # conversion can access other values from the same row
... table12 = etl.convert(table1, 'baz',
...                       lambda v, row: v * float(row.bar),
...                       pass_row=True)
>>> table12
+-----+-----+-----+
| foo | bar  | baz |
+=====+=====+=====+
| 'A' | '2.4' | 28.799999999999997 |
+-----+-----+-----+
| 'B' | '5.7' | 193.8 |
+-----+-----+-----+
| 'C' | '1.2' | 67.2 |
+-----+-----+-----+

```

Note that either field names or indexes can be given.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return `True` if the conversion should be applied on that row, else `False`.

The `pass_row` keyword argument can be given, which if `True` will mean that both the value and the containing row will be passed as arguments to the conversion function (so, i.e., the conversion function should accept two arguments).

`petl.transform.conversions.convertall` (*table*, \*args, \*\*kwargs)

Convenience function to convert all fields in the table using a common function or mapping. See also `convert()`.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return `True` if the conversion should be applied on that row, else `False`.

`petl.transform.conversions.convertnumbers` (*table*, *strict=False*, \*\*kwargs)

Convenience function to convert all field values to numbers where possible. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz', 'quux'],
...          ['1', '3.0', '9+3j', 'aaa'],
...          ['2', '1.3', '7+2j', None]]
>>> table2 = etl.convertnumbers(table1)
>>> table2
+-----+-----+-----+
| foo | bar | baz  | quux |
+=====+=====+=====+
| 1 | 3.0 | (9+3j) | 'aaa' |
+-----+-----+-----+
| 2 | 1.3 | (7+2j) | None |
+-----+-----+-----+

```

`petl.transform.conversions.replace` (*table*, *field*, *a*, *b*, \*\*kwargs)

Convenience function to replace all occurrences of *a* with *b* under the given field. See also `convert()`.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.replaceall(table, a, b, **kwargs)
```

Convenience function to replace all instances of *a* with *b* under all fields. See also `convertall()`.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.format(table, field, fmt, **kwargs)
```

Convenience function to format all values in the given *field* using the *fmt* format string.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.formatall(table, fmt, **kwargs)
```

Convenience function to format all values in all fields using the *fmt* format string.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.interpolate(table, field, fmt, **kwargs)
```

Convenience function to interpolate all values in the given *field* using the *fmt* string.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.interpolateall(table, fmt, **kwargs)
```

Convenience function to interpolate all values in all fields using the *fmt* string.

The `where` keyword argument can be given with a callable or expression which is evaluated on each row and which should return True if the conversion should be applied on that row, else False.

```
petl.transform.conversions.update(table, field, value, **kwargs)
```

Convenience function to convert a field to a fixed value. Accepts the `where` keyword argument. See also `convert()`.

## Selecting rows

```
petl.transform.selects.select(table, *args, **kwargs)
```

Select rows meeting a condition. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 4, 9.3],
...          ['a', 2, 88.2],
...          ['b', 1, 23.3],
...          ['c', 8, 42.0],
...          ['d', 7, 100.9],
...          ['c', 2]]
>>> # the second positional argument can be a function accepting
... # a row
... table2 = etl.select(table1,
...                      lambda rec: rec.foo == 'a' and rec.baz > 88.1)
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
```



```

| 'a' | 2 | 88.2 |
+-----+-----+-----+

>>> # the second positional argument can also be an expression
... # string, which will be converted to a function using petl.expr()
... table3 = etl.select(table1, "{foo} == 'a' and {baz} > 88.1")
>>> table3
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 2 | 88.2 |
+-----+-----+-----+

>>> # the condition can also be applied to a single field
... table4 = etl.select(table1, 'foo', lambda v: v == 'a')
>>> table4
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 4 | 9.3 |
+-----+-----+-----+
| 'a' | 2 | 88.2 |
+-----+-----+-----+

```

The complement of the selection can be returned (i.e., the query can be inverted) by providing `complement=True` as a keyword argument.

- `petl.transform.selects.selectop` (*table, field, value, op, complement=False*)  
Select rows where the function *op* applied to the given field and the given value returns *True*.
- `petl.transform.selects.selecteq` (*table, field, value, complement=False*)  
Select rows where the given field equals the given value.
- `petl.transform.selects.selectne` (*table, field, value, complement=False*)  
Select rows where the given field does not equal the given value.
- `petl.transform.selects.selectlt` (*table, field, value, complement=False*)  
Select rows where the given field is less than the given value.
- `petl.transform.selects.selectle` (*table, field, value, complement=False*)  
Select rows where the given field is less than or equal to the given value.
- `petl.transform.selects.selectgt` (*table, field, value, complement=False*)  
Select rows where the given field is greater than the given value.
- `petl.transform.selects.selectge` (*table, field, value, complement=False*)  
Select rows where the given field is greater than or equal to the given value.
- `petl.transform.selects.selectrangeopen` (*table, field, minv, maxv, complement=False*)  
Select rows where the given field is greater than or equal to *minv* and less than or equal to *maxv*.
- `petl.transform.selects.selectrangeopenleft` (*table, field, minv, maxv, complement=False*)  
Select rows where the given field is greater than or equal to *minv* and less than *maxv*.
- `petl.transform.selects.selectrangeopenright` (*table, field, minv, maxv, complement=False*)  
Select rows where the given field is greater than *minv* and less than or equal to *maxv*.
- `petl.transform.selects.selectrangeclosed` (*table, field, minv, maxv, complement=False*)  
Select rows where the given field is greater than *minv* and less than *maxv*.

`petl.transform.selects.selectcontains` (*table, field, value, complement=False*)  
 Select rows where the given field contains the given value.

`petl.transform.selects.selectin` (*table, field, value, complement=False*)  
 Select rows where the given field is a member of the given value.

`petl.transform.selects.selectnotin` (*table, field, value, complement=False*)  
 Select rows where the given field is not a member of the given value.

`petl.transform.selects.selectis` (*table, field, value, complement=False*)  
 Select rows where the given field *is* the given value.

`petl.transform.selects.selectisnot` (*table, field, value, complement=False*)  
 Select rows where the given field *is not* the given value.

`petl.transform.selects.selectisinstance` (*table, field, value, complement=False*)  
 Select rows where the given field is an instance of the given type.

`petl.transform.selects.selecttrue` (*table, field, complement=False*)  
 Select rows where the given field evaluates *True*.

`petl.transform.selects.selectfalse` (*table, field, complement=False*)  
 Select rows where the given field evaluates *False*.

`petl.transform.selects.selectnone` (*table, field, complement=False*)  
 Select rows where the given field is *None*.

`petl.transform.selects.selectnotnone` (*table, field, complement=False*)  
 Select rows where the given field is not *None*.

`petl.transform.selects.selectusingcontext` (*table, query*)  
 Select rows based on data in the current row and/or previous and next row. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 1],
...          ['B', 4],
...          ['C', 5],
...          ['D', 9]]
>>> def query(prv, cur, nxt):
...     return ((prv is not None and (cur.bar - prv.bar) < 2)
...           or (nxt is not None and (nxt.bar - cur.bar) < 2))
...
>>> table2 = etl.selectusingcontext(table1, query)
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'B' | 4 |
+-----+-----+
| 'C' | 5 |
+-----+-----+
```

The *query* function should accept three rows and return a boolean value.

`petl.transform.selects.rowlenselect` (*table, n, complement=False*)  
 Select rows of length *n*.

`petl.transform.selects.facet` (*table, key*)  
 Return a dictionary mapping field values to tables. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 4, 9.3],
...          ['a', 2, 88.2],
...          ['b', 1, 23.3],
...          ['c', 8, 42.0],
...          ['d', 7, 100.9],
...          ['c', 2]]
>>> foo = etl.facet(table1, 'foo')
>>> sorted(foo.keys())
['a', 'b', 'c', 'd']
>>> foo['a']
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 4 | 9.3 |
+-----+-----+-----+
| 'a' | 2 | 88.2 |
+-----+-----+-----+

>>> foo['c']
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'c' | 8 | 42.0 |
+-----+-----+-----+
| 'c' | 2 |      |
+-----+-----+-----+

```

See also `petl.util.materialise.facetcolumns()`.

`petl.transform.selects.biselect` (*table*, \**args*, \*\**kwargs*)

Return two tables, the first containing selected rows, the second containing remaining rows. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 4, 9.3],
...          ['a', 2, 88.2],
...          ['b', 1, 23.3],
...          ['c', 8, 42.0],
...          ['d', 7, 100.9],
...          ['c', 2]]
>>> table2, table3 = etl.biselect(table1, lambda rec: rec.foo == 'a')
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'a' | 4 | 9.3 |
+-----+-----+-----+
| 'a' | 2 | 88.2 |
+-----+-----+-----+

>>> table3
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'b' | 1 | 23.3 |
+-----+-----+-----+
| 'c' | 8 | 42.0 |
+-----+-----+-----+

```

```
+-----+-----+-----+
| 'd' | 7 | 100.9 |
+-----+-----+-----+
| 'c' | 2 | |
+-----+-----+-----+
```

New in version 1.1.0.

## Regular expressions

`petl.transform.regex.search` (*table*, \*args, \*\*kwargs)

Perform a regular expression search, returning rows that match a given pattern, either anywhere in the row or within a specific field. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['orange', 12, 'oranges are nice fruit'],
...          ['mango', 42, 'I like them'],
...          ['banana', 74, 'lovely too'],
...          ['cucumber', 41, 'better than mango']]
>>> # search any field
... table2 = etl.search(table1, '.g.')
>>> table2
+-----+-----+-----+
| foo      | bar | baz |
+-----+-----+-----+
| 'orange' | 12 | 'oranges are nice fruit' |
+-----+-----+-----+
| 'mango'  | 42 | 'I like them' |
+-----+-----+-----+
| 'cucumber' | 41 | 'better than mango' |
+-----+-----+-----+

>>> # search a specific field
... table3 = etl.search(table1, 'foo', '.g.')
>>> table3
+-----+-----+-----+
| foo      | bar | baz |
+-----+-----+-----+
| 'orange' | 12 | 'oranges are nice fruit' |
+-----+-----+-----+
| 'mango'  | 42 | 'I like them' |
+-----+-----+-----+
```

The complement can be found via `petl.transform.regex.searchcomplement()`.

`petl.transform.regex.searchcomplement` (*table*, \*args, \*\*kwargs)

Perform a regular expression search, returning rows that **do not** match a given pattern, either anywhere in the row or within a specific field. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['orange', 12, 'oranges are nice fruit'],
...          ['mango', 42, 'I like them'],
...          ['banana', 74, 'lovely too'],
...          ['cucumber', 41, 'better than mango']]
```

```

>>> # search any field
... table2 = etl.searchcomplement(table1, '.g.')
>>> table2
+-----+-----+-----+-----+
| foo      | bar | baz          |
+=====+=====+=====+=====+
| 'banana' | 74 | 'lovely too' |
+-----+-----+-----+-----+

>>> # search a specific field
... table3 = etl.searchcomplement(table1, 'foo', '.g.')
>>> table3
+-----+-----+-----+-----+
| foo      | bar | baz          |
+=====+=====+=====+=====+
| 'banana' | 74 | 'lovely too' |
+-----+-----+-----+-----+
| 'cucumber' | 41 | 'better than mango' |
+-----+-----+-----+-----+

```

This returns the complement of `petl.transform.regex.search()`.

`petl.transform.regex.sub` (*table, field, pattern, repl, count=0, flags=0*)

Convenience function to convert values under the given field using a regular expression substitution. See also `re.sub()`.

`petl.transform.regex.split` (*table, field, pattern, newfields=None, include\_original=False, maxsplit=0, flags=0*)

Add one or more new fields with values generated by splitting an existing value around occurrences of a regular expression. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'variable', 'value'],
...          ['1', 'parad1', '12'],
...          ['2', 'parad2', '15'],
...          ['3', 'tempd1', '18'],
...          ['4', 'tempd2', '19']]
>>> table2 = etl.split(table1, 'variable', 'd', ['variable', 'day'])
>>> table2
+-----+-----+-----+-----+
| id | value | variable | day |
+=====+=====+=====+=====+
| '1' | '12' | 'para'   | '1' |
+-----+-----+-----+-----+
| '2' | '15' | 'para'   | '2' |
+-----+-----+-----+-----+
| '3' | '18' | 'temp'   | '1' |
+-----+-----+-----+-----+
| '4' | '19' | 'temp'   | '2' |
+-----+-----+-----+-----+

```

By default the field on which the split is performed is omitted. It can be included using the `include_original` argument.

`petl.transform.regex.capture` (*table, field, pattern, newfields=None, include\_original=False, flags=0, fill=None*)

Add one or more new fields with values captured from an existing field searched via a regular expression. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'variable', 'value'],
...          ['1', 'A1', '12'],
...          ['2', 'A2', '15'],
...          ['3', 'B1', '18'],
...          ['4', 'C12', '19']]
>>> table2 = etl.capture(table1, 'variable', '(\w)(\d+)',
...                      ['treat', 'time'])
>>> table2
+-----+-----+-----+-----+
| id | value | treat | time |
+=====+=====+=====+=====+
| '1' | '12' | 'A' | '1' |
+-----+-----+-----+-----+
| '2' | '15' | 'A' | '2' |
+-----+-----+-----+-----+
| '3' | '18' | 'B' | '1' |
+-----+-----+-----+-----+
| '4' | '19' | 'C' | '12' |
+-----+-----+-----+-----+

>>> # using the include_original argument
... table3 = etl.capture(table1, 'variable', '(\w)(\d+)',
...                      ['treat', 'time'],
...                      include_original=True)
>>> table3
+-----+-----+-----+-----+
| id | variable | value | treat | time |
+=====+=====+=====+=====+
| '1' | 'A1' | '12' | 'A' | '1' |
+-----+-----+-----+-----+
| '2' | 'A2' | '15' | 'A' | '2' |
+-----+-----+-----+-----+
| '3' | 'B1' | '18' | 'B' | '1' |
+-----+-----+-----+-----+
| '4' | 'C12' | '19' | 'C' | '12' |
+-----+-----+-----+-----+

```

By default the field on which the capture is performed is omitted. It can be included using the *include\_original* argument.

The *fill* parameter can be used to provide a list or tuple of values to use if the regular expression does not match. The *fill* parameter should contain as many values as there are capturing groups in the regular expression. If *fill* is *None* (default) then a `petl.transform.TransformError` will be raised on the first non-matching value.

## Unpacking compound values

`petl.transform.unpacks.unpack`(*table*, *field*, *newfields=None*, *include\_original=False*, *missing=None*)

Unpack data values that are lists or tuples. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          [1, ['a', 'b']],
...          [2, ['c', 'd']],

```

```

...         [3, ['e', 'f']]]
>>> table2 = etl.unpack(table1, 'bar', ['baz', 'quux'])
>>> table2
+-----+-----+-----+
| foo | baz | quux |
+=====+=====+=====+
|  1 | 'a' | 'b'  |
+-----+-----+-----+
|  2 | 'c' | 'd'  |
+-----+-----+-----+
|  3 | 'e' | 'f'  |
+-----+-----+-----+

```

This function will attempt to unpack exactly the number of values as given by the number of new fields specified. If there are more values than new fields, remaining values will not be unpacked. If there are less values than new fields, *missing* values will be added.

See also `petl.transform.unpacks.unpackdict()`.

`petl.transform.unpacks.unpackdict` (*table*, *field*, *keys=None*, *includeoriginal=False*, *sample-size=1000*, *missing=None*)

Unpack dictionary values into separate fields. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          [1, {'baz': 'a', 'quux': 'b'}],
...          [2, {'baz': 'c', 'quux': 'd'}],
...          [3, {'baz': 'e', 'quux': 'f'}]]
>>> table2 = etl.unpackdict(table1, 'bar')
>>> table2
+-----+-----+-----+
| foo | baz | quux |
+=====+=====+=====+
|  1 | 'a' | 'b'  |
+-----+-----+-----+
|  2 | 'c' | 'd'  |
+-----+-----+-----+
|  3 | 'e' | 'f'  |
+-----+-----+-----+

```

See also `petl.transform.unpacks.unpack()`.

## Transforming rows

`petl.transform.maps.fieldmap` (*table*, *mappings=None*, *failonerror=False*, *errorvalue=None*)

Transform a table, mapping fields arbitrarily between input and output. E.g.:

```

>>> import petl as etl
>>> from collections import OrderedDict
>>> table1 = [['id', 'sex', 'age', 'height', 'weight'],
...          [1, 'male', 16, 1.45, 62.0],
...          [2, 'female', 19, 1.34, 55.4],
...          [3, 'female', 17, 1.78, 74.4],
...          [4, 'male', 21, 1.33, 45.2],
...          [5, '-', 25, 1.65, 51.9]]
>>> mappings = OrderedDict()

```

```

>>> # rename a field
... mappings['subject_id'] = 'id'
>>> # translate a field
... mappings['gender'] = 'sex', {'male': 'M', 'female': 'F'}
>>> # apply a calculation to a field
... mappings['age_months'] = 'age', lambda v: v * 12
>>> # apply a calculation to a combination of fields
... mappings['bmi'] = lambda rec: rec['weight'] / rec['height']**2
>>> # transform and inspect the output
... table2 = etl.fieldmap(table1, mappings)
>>> table2
+-----+-----+-----+-----+
| subject_id | gender | age_months | bmi |
+-----+-----+-----+-----+
|          1 | 'M'   |         192 | 29.48870392390012 |
+-----+-----+-----+-----+
|          2 | 'F'   |         228 | 30.8531967030519 |
+-----+-----+-----+-----+
|          3 | 'F'   |         204 | 23.481883600555488 |
+-----+-----+-----+-----+
|          4 | 'M'   |         252 | 25.55260331279326 |
+-----+-----+-----+-----+
|          5 | '-'   |         300 | 19.0633608815427 |
+-----+-----+-----+-----+

```

Note also that the mapping value can be an expression string, which will be converted to a lambda function via `petl.util.base.expr()`.

`petl.transform.maps.rowmap` (*table*, *rowmapper*, *header*, *failonerror=False*)  
 Transform rows via an arbitrary function. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'sex', 'age', 'height', 'weight'],
...           [1, 'male', 16, 1.45, 62.0],
...           [2, 'female', 19, 1.34, 55.4],
...           [3, 'female', 17, 1.78, 74.4],
...           [4, 'male', 21, 1.33, 45.2],
...           [5, '-', 25, 1.65, 51.9]]
>>> def rowmapper(row):
...     transmf = {'male': 'M', 'female': 'F'}
...     return [row[0],
...             transmf[row['sex']] if row['sex'] in transmf else None,
...             row.age * 12,
...             row.height / row.weight ** 2]
>>> table2 = etl.rowmap(table1, rowmapper,
...                     header=['subject_id', 'gender', 'age_months',
...                              'bmi'])
>>> table2
+-----+-----+-----+-----+
| subject_id | gender | age_months | bmi |
+-----+-----+-----+-----+
|          1 | 'M'   |         192 | 0.0003772112382934443 |
+-----+-----+-----+-----+
|          2 | 'F'   |         228 | 0.0004366015456998006 |
+-----+-----+-----+-----+
|          3 | 'F'   |         204 | 0.0003215689675106949 |
+-----+-----+-----+-----+

```



4	'M'	252	0.0006509906805544679
5	None	300	0.0006125608384287258

The `rowmapper` function should accept a single row and return a single row (list or tuple).

`petl.transform.maps.rowmapmany` (*table*, *rowgenerator*, *header*, *failonerror=False*)  
Map each input row to any number of output rows via an arbitrary function. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'sex', 'age', 'height', 'weight'],
...          [1, 'male', 16, 1.45, 62.0],
...          [2, 'female', 19, 1.34, 55.4],
...          [3, '-', 17, 1.78, 74.4],
...          [4, 'male', 21, 1.33]]
>>> def rowgenerator(row):
...     transmf = {'male': 'M', 'female': 'F'}
...     yield [row[0], 'gender',
...            transmf[row['sex']] if row['sex'] in transmf else None]
...     yield [row[0], 'age_months', row.age * 12]
...     yield [row[0], 'bmi', row.height / row.weight ** 2]
...
>>> table2 = etl.rowmapmany(table1, rowgenerator,
...                          header=['subject_id', 'variable', 'value'])
>>> table2.lookall()
+-----+-----+-----+
| subject_id | variable      | value                |
+-----+-----+-----+
| 1          | 'gender'      | 'M'                  |
+-----+-----+-----+
| 1          | 'age_months'  | 192                  |
+-----+-----+-----+
| 1          | 'bmi'         | 0.0003772112382934443 |
+-----+-----+-----+
| 2          | 'gender'      | 'F'                  |
+-----+-----+-----+
| 2          | 'age_months'  | 228                  |
+-----+-----+-----+
| 2          | 'bmi'         | 0.0004366015456998006 |
+-----+-----+-----+
| 3          | 'gender'      | None                 |
+-----+-----+-----+
| 3          | 'age_months'  | 204                  |
+-----+-----+-----+
| 3          | 'bmi'         | 0.0003215689675106949 |
+-----+-----+-----+
| 4          | 'gender'      | 'M'                  |
+-----+-----+-----+
| 4          | 'age_months'  | 252                  |
+-----+-----+-----+
```

The `rowgenerator` function should accept a single row and yield zero or more rows (lists or tuples).

See also the `petl.transform.reshape.melt()` function.

`petl.transform.maps.rowgroupmap` (*table*, *key*, *mapper*, *header=None*, *presorted=False*, *buffer-size=None*, *tempdir=None*, *cache=True*)

Group rows under the given key then apply *mapper* to yield zero or more output rows for each input group of

rows.

## Sorting

`petl.transform.sorts.sort` (*table*, *key=None*, *reverse=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Sort the table. Field names or indices (from zero) can be used to specify the key. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['C', 2],
...          ['A', 9],
...          ['A', 6],
...          ['F', 1],
...          ['D', 10]]
>>> table2 = etl.sort(table1, 'foo')
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 9 |
+-----+-----+
| 'A' | 6 |
+-----+-----+
| 'C' | 2 |
+-----+-----+
| 'D' | 10 |
+-----+-----+
| 'F' | 1 |
+-----+-----+

>>> # sorting by compound key is supported
... table3 = etl.sort(table1, key=['foo', 'bar'])
>>> table3
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 6 |
+-----+-----+
| 'A' | 9 |
+-----+-----+
| 'C' | 2 |
+-----+-----+
| 'D' | 10 |
+-----+-----+
| 'F' | 1 |
+-----+-----+

>>> # if no key is specified, the default is a lexical sort
... table4 = etl.sort(table1)
>>> table4
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 6 |
+-----+-----+
| 'A' | 9 |
```

```
+-----+-----+
| 'C' | 2 |
+-----+-----+
| 'D' | 10 |
+-----+-----+
| 'F' | 1 |
+-----+-----+
```

The *buffersize* argument should be an *int* or *None*.

If the number of rows in the table is less than *buffersize*, the table will be sorted in memory. Otherwise, the table is sorted in chunks of no more than *buffersize* rows, each chunk is written to a temporary file, and then a merge sort is performed on the temporary files.

If *buffersize* is *None*, the value of *petl.config.sort\_buffersize* will be used. By default this is set to 100000 rows, but can be changed, e.g.:

```
>>> import petl.config
>>> petl.config.sort_buffersize = 500000
```

If *petl.config.sort\_buffersize* is set to *None*, this forces all sorting to be done entirely in memory.

By default the results of the sort will be cached, and so a second pass over the sorted table will yield rows from the cache and will not repeat the sort operation. To turn off caching, set the *cache* argument to *False*.

`petl.transform.sorts.mergesort(*tables, **kwargs)`

Combine multiple input tables into one sorted output table. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['A', 9],
...          ['C', 2],
...          ['D', 10],
...          ['A', 6],
...          ['F', 1]]
>>> table2 = [['foo', 'bar'],
...          ['B', 3],
...          ['D', 10],
...          ['A', 10],
...          ['F', 4]]
>>> table3 = etl.mergesort(table1, table2, key='foo')
>>> table3.lookall()
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 9 |
+-----+-----+
| 'A' | 6 |
+-----+-----+
| 'A' | 10 |
+-----+-----+
| 'B' | 3 |
+-----+-----+
| 'C' | 2 |
+-----+-----+
| 'D' | 10 |
+-----+-----+
| 'D' | 10 |
+-----+-----+
```

```
| 'F' | 1 |
+-----+-----+
| 'F' | 4 |
+-----+-----+
```

If the input tables are already sorted by the given key, give `presorted=True` as a keyword argument.

This function is equivalent to concatenating the input tables using `cat()` then sorting, however this function will typically be more efficient, especially if the input tables are presorted.

Keyword arguments:

**key** [string or tuple of strings, optional] Field name or tuple of fields to sort by (defaults to *None* lexical sort)

**reverse** [bool, optional] *True* if sort in reverse (descending) order (defaults to *False*)

**presorted** [bool, optional] *True* if inputs are already sorted by the given key (defaults to *False*)

**missing** [object] Value to fill with when input tables have different fields (defaults to *None*)

**header** [sequence of strings, optional] Specify a fixed header for the output table

**bufferize** [int, optional] Limit the number of rows in memory per input table when inputs are not presorted

`petl.transform.sorts.issorted(table, key=None, reverse=False, strict=False)`

Return *True* if the table is ordered (i.e., sorted) by the given key. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 1, True],
...          ['b', 3, True],
...          ['b', 2]]
>>> etl.issorted(table1, key='foo')
True
>>> etl.issorted(table1, key='bar')
False
>>> etl.issorted(table1, key='foo', strict=True)
False
>>> etl.issorted(table1, key='foo', reverse=True)
False
```

## Joins

`petl.transform.joins.join(left, right, key=None, lkey=None, rkey=None, presorted=False, bufferize=None, tempdir=None, cache=True, lprefix=None, rprefix=None)`

Perform an equi-join on the given tables. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red'],
...          [3, 'purple']]
>>> table2 = [['id', 'shape'],
...          [1, 'circle'],
...          [3, 'square'],
...          [4, 'ellipse']]
>>> table3 = etl.join(table1, table2, key='id')
>>> table3
```

```

+----+-----+-----+
| id | colour  | shape  |
+====+=====+=====+
| 1 | 'blue'  | 'circle' |
+----+-----+-----+
| 3 | 'purple' | 'square' |
+----+-----+-----+

>>> # if no key is given, a natural join is tried
... table4 = etl.join(table1, table2)
>>> table4
+----+-----+-----+
| id | colour  | shape  |
+====+=====+=====+
| 1 | 'blue'  | 'circle' |
+----+-----+-----+
| 3 | 'purple' | 'square' |
+----+-----+-----+

>>> # note behaviour if the key is not unique in either or both tables
... table5 = [['id', 'colour'],
...           [1, 'blue'],
...           [1, 'red'],
...           [2, 'purple']]
>>> table6 = [['id', 'shape'],
...           [1, 'circle'],
...           [1, 'square'],
...           [2, 'ellipse']]
>>> table7 = etl.join(table5, table6, key='id')
>>> table7
+----+-----+-----+
| id | colour  | shape  |
+====+=====+=====+
| 1 | 'blue'  | 'circle' |
+----+-----+-----+
| 1 | 'blue'  | 'square' |
+----+-----+-----+
| 1 | 'red'   | 'circle' |
+----+-----+-----+
| 1 | 'red'   | 'square' |
+----+-----+-----+
| 2 | 'purple' | 'ellipse' |
+----+-----+-----+

>>> # compound keys are supported
... table8 = [['id', 'time', 'height'],
...           [1, 1, 12.3],
...           [1, 2, 34.5],
...           [2, 1, 56.7]]
>>> table9 = [['id', 'time', 'weight'],
...           [1, 2, 4.5],
...           [2, 1, 6.7],
...           [2, 2, 8.9]]
>>> table10 = etl.join(table8, table9, key=['id', 'time'])
>>> table10
+----+-----+-----+-----+
| id | time | height | weight |
+====+=====+=====+=====+

```

1	2	34.5	4.5
+-----+			
2	1	56.7	6.7
+-----+			

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.joins.leftjoin(left, right, key=None, lkey=None, rkey=None, missing=None, presorted=False, bufferize=None, tempdir=None, cache=True, lprefix=None, rprefix=None)`

Perform a left outer join on the given tables. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red'],
...          [3, 'purple']]
>>> table2 = [['id', 'shape'],
...          [1, 'circle'],
...          [3, 'square'],
...          [4, 'ellipse']]
>>> table3 = etl.leftjoin(table1, table2, key='id')
>>> table3
+-----+-----+-----+
| id | colour | shape |
+====+=====+=====+
| 1 | 'blue' | 'circle' |
+-----+-----+-----+
| 2 | 'red'  | None    |
+-----+-----+-----+
| 3 | 'purple' | 'square' |
+-----+-----+-----+
```

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.joins.lookupjoin(left, right, key=None, lkey=None, rkey=None, missing=None, presorted=False, bufferize=None, tempdir=None, cache=True, lprefix=None, rprefix=None)`

Perform a left join, but where the key is not unique in the right-hand table, arbitrarily choose the first row and ignore others. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'color', 'cost'],
...          [1, 'blue', 12],
...          [2, 'red', 8],
...          [3, 'purple', 4]]
>>> table2 = [['id', 'shape', 'size'],
...          [1, 'circle', 'big'],
...          [1, 'circle', 'small'],
...          [2, 'square', 'tiny'],
...          [2, 'square', 'big']]
```

```

...         [3, 'ellipse', 'small'],
...         [3, 'ellipse', 'tiny']]
>>> table3 = etl.lookupjoin(table1, table2, key='id')
>>> table3
+-----+-----+-----+-----+-----+
| id | color  | cost | shape  | size  |
+====+=====+=====+=====+=====+
| 1 | 'blue' | 12  | 'circle' | 'big' |
+-----+-----+-----+-----+-----+
| 2 | 'red'  | 8   | 'square' | 'tiny' |
+-----+-----+-----+-----+-----+
| 3 | 'purple' | 4  | 'ellipse' | 'small' |
+-----+-----+-----+-----+-----+

```

See also `petl.transform.joins.leftjoin()`.

`petl.transform.joins.rightjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *missing=None*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*, *lprefix=None*, *rprefix=None*)

Perform a right outer join on the given tables. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red'],
...          [3, 'purple']]
>>> table2 = [['id', 'shape'],
...          [1, 'circle'],
...          [3, 'square'],
...          [4, 'ellipse']]
>>> table3 = etl.rightjoin(table1, table2, key='id')
>>> table3
+-----+-----+-----+
| id | colour | shape  |
+====+=====+=====+
| 1 | 'blue' | 'circle' |
+-----+-----+-----+
| 3 | 'purple' | 'square' |
+-----+-----+-----+
| 4 | None   | 'ellipse' |
+-----+-----+-----+

```

If *presorted* is `True`, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.joins.outerjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *missing=None*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*, *lprefix=None*, *rprefix=None*)

Perform a full outer join on the given tables. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red'],
...          [3, 'purple']]
>>> table2 = [['id', 'shape'],

```

```

...         [1, 'circle'],
...         [3, 'square'],
...         [4, 'ellipse']]
>>> table3 = etl.outerjoin(table1, table2, key='id')
>>> table3
+----+-----+-----+
| id | colour | shape |
+====+=====+=====+
| 1 | 'blue' | 'circle' |
+----+-----+-----+
| 2 | 'red'  | None    |
+----+-----+-----+
| 3 | 'purple' | 'square' |
+----+-----+-----+
| 4 | None   | 'ellipse' |
+----+-----+-----+

```

If *presorted* is *True*, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.joins.crossjoin(*tables, **kwargs)`

Form the cartesian product of the given tables. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red']]
>>> table2 = [['id', 'shape'],
...          [1, 'circle'],
...          [3, 'square']]
>>> table3 = etl.crossjoin(table1, table2)
>>> table3
+----+-----+----+-----+
| id | colour | id | shape  |
+====+=====+====+=====+
| 1 | 'blue' | 1 | 'circle' |
+----+-----+----+-----+
| 1 | 'blue' | 3 | 'square' |
+----+-----+----+-----+
| 2 | 'red'  | 1 | 'circle' |
+----+-----+----+-----+
| 2 | 'red'  | 3 | 'square' |
+----+-----+----+-----+

```

If *prefix* is *True* then field names in the output table header will be prefixed by the index of the input table.

`petl.transform.joins.antijoin(left, right, key=None, lkey=None, rkey=None, presorted=False, bufferize=None, tempdir=None, cache=True)`

Return rows from the *left* table where the key value does not occur in the *right* table. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [0, 'black'],
...          [1, 'blue'],
...          [2, 'red'],
...          [4, 'yellow'],

```



```

...         [5, 'white']]
>>> table2 = [['id', 'shape'],
...          [1, 'circle'],
...          [3, 'square']]
>>> table3 = etl.antijoin(table1, table2, key='id')
>>> table3
+-----+-----+
| id | colour |
+=====+=====+
| 0 | 'black' |
+-----+-----+
| 2 | 'red'   |
+-----+-----+
| 4 | 'yellow'|
+-----+-----+
| 5 | 'white' |
+-----+-----+

```

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

```
petl.transform.joins.unjoin(table, value, key=None, autoincrement=(1, 1), presorted=False,
                           buffersize=None, tempdir=None, cache=True)
```

Split a table into two tables by reversing an inner join. E.g.:

```

>>> import petl as etl
>>> # join key is present in the table
... table1 = (('foo', 'bar', 'baz'),
...          ('A', 1, 'apple'),
...          ('B', 1, 'apple'),
...          ('C', 2, 'orange'))
>>> table2, table3 = etl.unjoin(table1, 'baz', key='bar')
>>> table2
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 1 |
+-----+-----+
| 'B' | 1 |
+-----+-----+
| 'C' | 2 |
+-----+-----+

>>> table3
+-----+-----+
| bar | baz |
+=====+=====+
| 1 | 'apple' |
+-----+-----+
| 2 | 'orange' |
+-----+-----+

>>> # an integer join key can also be reconstructed
... table4 = (('foo', 'bar'),
...          ('A', 'apple'),
...          ('B', 'apple'),

```

```

...         ('C', 'orange'))
>>> table5, table6 = etl.unjoin(table4, 'bar')
>>> table5
+-----+-----+
| foo | bar_id |
+=====+=====+
| 'A' |      1 |
+-----+-----+
| 'B' |      1 |
+-----+-----+
| 'C' |      2 |
+-----+-----+

>>> table6
+-----+-----+
| id | bar      |
+=====+=====+
|  1 | 'apple'  |
+-----+-----+
|  2 | 'orange' |
+-----+-----+

```

The *autoincrement* parameter controls how an integer join key is reconstructed, and should be a tuple of (*start*, *step*).

`petl.transform.hashjoins.hashjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *cache=True*, *lprefix=None*, *rprefix=None*)

Alternative implementation of `petl.transform.joins.join()`, where the join is executed by constructing an in-memory lookup for the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

By default data from right hand table is cached to improve performance (only available when *key* is given).

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.hashjoins.hashleftjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *missing=None*, *cache=True*, *lprefix=None*, *rprefix=None*)

Alternative implementation of `petl.transform.joins.leftjoin()`, where the join is executed by constructing an in-memory lookup for the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

By default data from right hand table is cached to improve performance (only available when *key* is given).

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.hashjoins.hashlookupjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *missing=None*, *lprefix=None*, *rprefix=None*)

Alternative implementation of `petl.transform.joins.lookupjoin()`, where the join is executed by constructing an in-memory lookup for the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.hashjoins.hashrightjoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*, *missing=None*, *cache=True*, *lprefix=None*, *rprefix=None*)

Alternative implementation of `petl.transform.joins.rightjoin()`, where the join is executed by constructing an in-memory lookup for the left hand table, then iterating over rows from the right hand table.

May be faster and/or more resource efficient where the left table is small and the right table is large.

By default data from right hand table is cached to improve performance (only available when *key* is given).

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

`petl.transform.hashjoins.hashantijoin` (*left*, *right*, *key=None*, *lkey=None*, *rkey=None*)

Alternative implementation of `petl.transform.joins.antijoin()`, where the join is executed by constructing an in-memory set for all keys found in the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

Left and right tables with different key fields can be handled via the *lkey* and *rkey* arguments.

## Set operations

`petl.transform.setops.complement` (*a*, *b*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*, *strict=False*)

Return rows in *a* that are not in *b*. E.g.:

```
>>> import petl as etl
>>> a = [['foo', 'bar', 'baz'],
...      ['A', 1, True],
...      ['C', 7, False],
...      ['B', 2, False],
...      ['C', 9, True]]
>>> b = [['x', 'y', 'z'],
...      ['B', 2, False],
...      ['A', 9, False],
...      ['B', 3, True],
...      ['C', 9, True]]
>>> aminusb = etl.complement(a, b)
>>> aminusb
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | True |
+-----+-----+-----+
| 'C' | 7 | False |
+-----+-----+-----+

>>> bminusa = etl.complement(b, a)
>>> bminusa
+-----+-----+-----+
| x | y | z |
+=====+=====+=====+
| 'A' | 9 | False |
+-----+-----+-----+
| 'B' | 3 | True |
+-----+-----+-----+
```

Note that the field names of each table are ignored - rows are simply compared following a lexical sort. See also the `petl.transform.setops.recordcomplement()` function.

If *presorted* is *True*, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

Note that the default behaviour is not strictly set-like, because duplicate rows are counted separately, e.g.:

```
>>> a = [['foo', 'bar'],
...      ['A', 1],
...      ['B', 2],
...      ['B', 2],
...      ['C', 7]]
>>> b = [['foo', 'bar'],
...      ['B', 2]]
>>> aminusb = etl.complement(a, b)
>>> aminusb
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 1 |
+-----+-----+
| 'B' | 2 |
+-----+-----+
| 'C' | 7 |
+-----+-----+
```

This behaviour can be changed with the *strict* keyword argument, e.g.:

```
>>> aminusb = etl.complement(a, b, strict=True)
>>> aminusb
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' | 1 |
+-----+-----+
| 'C' | 7 |
+-----+-----+
```

Changed in version 1.1.0.

If *strict* is *True* then strict set-like behaviour is used, i.e., only rows in *a* not found in *b* are returned.

`petl.transform.setops.diff(a, b, presorted=False, buffersize=None, tempdir=None, cache=True, strict=False)`

Find the difference between rows in two tables. Returns a pair of tables. E.g.:

```
>>> import petl as etl
>>> a = [['foo', 'bar', 'baz'],
...      ['A', 1, True],
...      ['C', 7, False],
...      ['B', 2, False],
...      ['C', 9, True]]
>>> b = [['x', 'y', 'z'],
...      ['B', 2, False],
...      ['A', 9, False],
...      ['B', 3, True],
...      ['C', 9, True]]
>>> added, subtracted = etl.diff(a, b)
>>> # rows in b not in a
... added
+-----+-----+
| x | y | z |
+=====+=====+
| 'A' | 9 | False |
```

```

+-----+-----+-----+
| 'B' | 3 | True |
+-----+-----+-----+

>>> # rows in a not in b
... subtracted
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | True |
+-----+-----+-----+
| 'C' | 7 | False |
+-----+-----+-----+

```

Convenient shorthand for `(complement(b, a), complement(a, b))`. See also `petl.transform.setops.complement()`.

If `presorted` is `True`, it is assumed that the data are already sorted by the given key, and the `bufferize`, `tempdir` and `cache` arguments are ignored. Otherwise, the data are sorted, see also the discussion of the `bufferize`, `tempdir` and `cache` arguments under the `petl.transform.sorts.sort()` function.

Changed in version 1.1.0.

If `strict` is `True` then strict set-like behaviour is used.

`petl.transform.setops.recordcomplement(a, b, bufferize=None, tempdir=None, cache=True, strict=False)`

Find records in `a` that are not in `b`. E.g.:

```

>>> import petl as etl
>>> a = [['foo', 'bar', 'baz'],
...      ['A', 1, True],
...      ['C', 7, False],
...      ['B', 2, False],
...      ['C', 9, True]]
>>> b = [['bar', 'foo', 'baz'],
...      [2, 'B', False],
...      [9, 'A', False],
...      [3, 'B', True],
...      [9, 'C', True]]
>>> aminusb = etl.recordcomplement(a, b)
>>> aminusb
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | True |
+-----+-----+-----+
| 'C' | 7 | False |
+-----+-----+-----+

>>> bminusa = etl.recordcomplement(b, a)
>>> bminusa
+-----+-----+-----+
| bar | foo | baz |
+=====+=====+=====+
| 3 | 'B' | True |
+-----+-----+-----+
| 9 | 'A' | False |
+-----+-----+-----+

```

Note that both tables must have the same set of fields, but that the order of the fields does not matter. See also the `petl.transform.setops.complement()` function.

See also the discussion of the `bufferize`, `tempdir` and `cache` arguments under the `petl.transform.sorts.sort()` function.

`petl.transform.setops.recorddiff(a, b, bufferize=None, tempdir=None, cache=True, strict=False)`

Find the difference between records in two tables. E.g.:

```
>>> import petl as etl
>>> a = [['foo', 'bar', 'baz'],
...      ['A', 1, True],
...      ['C', 7, False],
...      ['B', 2, False],
...      ['C', 9, True]]
>>> b = [['bar', 'foo', 'baz'],
...      [2, 'B', False],
...      [9, 'A', False],
...      [3, 'B', True],
...      [9, 'C', True]]
>>> added, subtracted = etl.recorddiff(a, b)
>>> added
+-----+-----+-----+
| bar | foo | baz  |
+=====+=====+=====+
|  3 | 'B' | True  |
+-----+-----+-----+
|  9 | 'A' | False |
+-----+-----+-----+

>>> subtracted
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'A' |  1 | True  |
+-----+-----+-----+
| 'C' |  7 | False |
+-----+-----+-----+
```

Convenient shorthand for `(recordcomplement(b, a), recordcomplement(a, b))`. See also `petl.transform.setops.recordcomplement()`.

See also the discussion of the `bufferize`, `tempdir` and `cache` arguments under the `petl.transform.sorts.sort()` function.

Changed in version 1.1.0.

If `strict` is `True` then strict set-like behaviour is used.

`petl.transform.setops.intersection(a, b, presorted=False, bufferize=None, tempdir=None, cache=True)`

Return rows in `a` that are also in `b`. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...           ['A', 1, True],
...           ['C', 7, False],
...           ['B', 2, False],
...           ['C', 9, True]]
>>> table2 = [['x', 'y', 'z'],
```

```

...         ['B', 2, False],
...         ['A', 9, False],
...         ['B', 3, True],
...         ['C', 9, True]]
>>> table3 = etl.intersection(table1, table2)
>>> table3
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'B' |  2 | False |
+-----+-----+-----+
| 'C' |  9 | True  |
+-----+-----+-----+

```

If *presorted* is *True*, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

`petl.transform.setops.hashcomplement(a, b, strict=False)`

Alternative implementation of `petl.transform.setops.complement()`, where the complement is executed by constructing an in-memory set for all rows found in the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

Changed in version 1.1.0.

If *strict* is *True* then strict set-like behaviour is used, i.e., only rows in *a* not found in *b* are returned.

`petl.transform.setops.hashintersection(a, b)`

Alternative implementation of `petl.transform.setops.intersection()`, where the intersection is executed by constructing an in-memory set for all rows found in the right hand table, then iterating over rows from the left hand table.

May be faster and/or more resource efficient where the right table is small and the left table is large.

## Deduplicating rows

`petl.transform.dedup.duplicates(table, key=None, presorted=False, bufferize=None, tempdir=None, cache=True)`

Select rows with duplicate values under a given key (or duplicate rows where no key is given). E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2.0],
...          ['B', 2, 3.4],
...          ['D', 6, 9.3],
...          ['B', 3, 7.8],
...          ['B', 2, 12.3],
...          ['E', None, 1.3],
...          ['D', 4, 14.5]]
>>> table2 = etl.duplicates(table1, 'foo')
>>> table2
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'B' |  2 | 3.4  |

```

```

+-----+-----+-----+
| 'B' | 3 | 7.8 |
+-----+-----+-----+
| 'B' | 2 | 12.3 |
+-----+-----+-----+
| 'D' | 6 | 9.3 |
+-----+-----+-----+
| 'D' | 4 | 14.5 |
+-----+-----+-----+

>>> # compound keys are supported
... table3 = etl.duplicates(table1, key=['foo', 'bar'])
>>> table3
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'B' | 2 | 3.4 |
+-----+-----+-----+
| 'B' | 2 | 12.3 |
+-----+-----+-----+

```

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

See also `petl.transform.dedup.unique()` and `petl.transform.dedup.distinct()`.

`petl.transform.dedup.unique` (*table*, *key=None*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*)

Select rows with unique values under a given key (or unique rows if no key is given). E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2],
...          ['B', '2', '3.4'],
...          ['D', 'xyz', 9.0],
...          ['B', u'3', u'7.8'],
...          ['B', '2', 42],
...          ['E', None, None],
...          ['D', 4, 12.3],
...          ['F', 7, 2.3]]
>>> table2 = etl.unique(table1, 'foo')
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | 2 |
+-----+-----+-----+
| 'E' | None | None |
+-----+-----+-----+
| 'F' | 7 | 2.3 |
+-----+-----+-----+

```

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

See also `petl.transform.dedup.duplicates()` and `petl.transform.dedup.distinct()`.



`petl.transform.dedup.conflicts` (*table*, *key*, *missing=None*, *include=None*, *exclude=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Select rows with the same key value but differing in some other field. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2.7],
...          ['B', 2, None],
...          ['D', 3, 9.4],
...          ['B', None, 7.8],
...          ['E', None],
...          ['D', 3, 12.3],
...          ['A', 2, None]]
>>> table2 = etl.conflicts(table1, 'foo')
>>> table2
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 'A' | 1 | 2.7 |
+-----+-----+-----+
| 'A' | 2 | None |
+-----+-----+-----+
| 'D' | 3 | 9.4 |
+-----+-----+-----+
| 'D' | 3 | 12.3 |
+-----+-----+-----+
```

Missing values are not considered conflicts. By default, *None* is treated as the missing value, this can be changed via the *missing* keyword argument.

One or more fields can be ignored when determining conflicts by providing the *exclude* keyword argument. Alternatively, fields to use when determining conflicts can be specified explicitly with the *include* keyword argument. This provides a simple mechanism for analysing the source of conflicting rows from multiple tables, e.g.:

```
>>> table1 = [['foo', 'bar'], [1, 'a'], [2, 'b']]
>>> table2 = [['foo', 'bar'], [1, 'a'], [2, 'c']]
>>> table3 = etl.cat(etl.addfield(table1, 'source', 1),
...                 etl.addfield(table2, 'source', 2))
>>> table4 = etl.conflicts(table3, key='foo', exclude='source')
>>> table4
+-----+-----+-----+
| foo | bar | source |
+=====+=====+=====+
| 2 | 'b' | 1 |
+-----+-----+-----+
| 2 | 'c' | 2 |
+-----+-----+-----+
```

If *presorted* is *True*, it is assumed that the data are already sorted by the given key, and the *buffersize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *buffersize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

`petl.transform.dedup.distinct` (*table*, *key=None*, *count=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Return only distinct rows in the table.

If the *count* argument is not *None*, it will be used as the name for an additional field, and the values of the field will be the number of duplicate rows.

If the *key* keyword argument is passed, the comparison is done on the given key instead of the full row.

See also `petl.transform.dedup.duplicates()`, `petl.transform.dedup.unique()`, `petl.transform.reductions.groupselectfirst()`, `petl.transform.reductions.groupselectlast()`.

`petl.transform.dedup.isunique(table, field)`

Return True if there are no duplicate values for the given field(s), otherwise False. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b'],
...          ['b', 2],
...          ['c', 3, True]]
>>> etl.isunique(table1, 'foo')
False
>>> etl.isunique(table1, 'bar')
True
```

The *field* argument can be a single field name or index (starting from zero) or a tuple of field names and/or indexes.

## Reducing rows (aggregation)

`petl.transform.reductions.aggregate(table, key, aggregation=None, value=None, pre-sorted=False, buffersize=None, tempdir=None, cache=True)`

Group rows under the given key then apply aggregation functions. E.g.:

```
>>> import petl as etl
>>>
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 3, True],
...          ['a', 7, False],
...          ['b', 2, True],
...          ['b', 2, False],
...          ['b', 9, False],
...          ['c', 4, True]]
>>> # aggregate whole rows
... table2 = etl.aggregate(table1, 'foo', len)
>>> table2
+-----+-----+
| foo | value |
+=====+=====+
| 'a' |      2 |
+-----+-----+
| 'b' |      3 |
+-----+-----+
| 'c' |      1 |
+-----+-----+

>>> # aggregate single field
... table3 = etl.aggregate(table1, 'foo', sum, 'bar')
>>> table3
+-----+-----+
| foo | value |
```

```

+====+=====+
| 'a' |    10 |
+----+-----+
| 'b' |    13 |
+----+-----+
| 'c' |     4 |
+----+-----+

>>> # alternative signature using keyword args
... table4 = etl.aggregate(table1, key=('foo', 'bar'),
...                        aggregation=list, value=('bar', 'baz'))
>>> table4
+----+-----+
| foo | bar | value |
+====+=====+
| 'a' |  3 | [(3, True)] |
+----+-----+
| 'a' |  7 | [(7, False)] |
+----+-----+
| 'b' |  2 | [(2, True), (2, False)] |
+----+-----+
| 'b' |  9 | [(9, False)] |
+----+-----+
| 'c' |  4 | [(4, True)] |
+----+-----+

>>> # aggregate multiple fields
... from collections import OrderedDict
>>> import petl as etl
>>>
>>> aggregation = OrderedDict()
>>> aggregation['count'] = len
>>> aggregation['minbar'] = 'bar', min
>>> aggregation['maxbar'] = 'bar', max
>>> aggregation['sumbar'] = 'bar', sum
>>> # default aggregation function is list
... aggregation['listbar'] = 'bar'
>>> aggregation['listbarbaz'] = ('bar', 'baz'), list
>>> aggregation['bars'] = 'bar', etl.strjoin(', ')
>>> table5 = etl.aggregate(table1, 'foo', aggregation)
>>> table5
+----+-----+-----+-----+-----+-----+-----+-----+
↪-----+-----+
| foo | count | minbar | maxbar | sumbar | listbar | listbarbaz |
↪          | bars   |
+====+=====+=====+=====+=====+=====+=====+=====+
| 'a' |    2 |    3 |    7 |    10 | [3, 7] | [(3, True), (7, False)] |
↪          | '3, 7' |
+----+-----+-----+-----+-----+-----+-----+-----+
↪-----+-----+
| 'b' |    3 |    2 |    9 |    13 | [2, 2, 9] | [(2, True), (2, False), (9,
↪ False)] | '2, 2, 9' |
+----+-----+-----+-----+-----+-----+-----+-----+
↪-----+-----+
| 'c' |    1 |    4 |    4 |    4 | [4] | [(4, True)] |
↪          | '4' |
+----+-----+-----+-----+-----+-----+-----+
↪-----+-----+

```

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

`petl.transform.reductions.rowreduce`(*table*, *key*, *reducer*, *header=None*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*)

Group rows under the given key then apply *reducer* to produce a single output row for each input group of rows. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 3],
...          ['a', 7],
...          ['b', 2],
...          ['b', 1],
...          ['b', 9],
...          ['c', 4]]
>>> def sumbar(key, rows):
...     return [key, sum(row[1] for row in rows)]
...
>>> table2 = etl.rowreduce(table1, key='foo', reducer=sumbar,
...                        header=['foo', 'barsum'])
>>> table2
+-----+-----+
| foo | barsum |
+=====+=====+
| 'a' |      10 |
+-----+-----+
| 'b' |      12 |
+-----+-----+
| 'c' |       4 |
+-----+-----+
```

N.B., this is not strictly a “reduce” in the sense of the standard Python `reduce()` function, i.e., the *reducer* function is *not* applied recursively to values within a group, rather it is applied once to each row group as a whole.

See also `petl.transform.reductions.aggregate()` and `petl.transform.reductions.fold()`.

`petl.transform.reductions.mergeduplicates`(*table*, *key*, *missing=None*, *presorted=False*, *bufferize=None*, *tempdir=None*, *cache=True*)

Merge duplicate rows under the given key. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, 2.7],
...          ['B', 2, None],
...          ['D', 3, 9.4],
...          ['B', None, 7.8],
...          ['E', None, 42.],
...          ['D', 3, 12.3],
...          ['A', 2, None]]
>>> table2 = etl.mergeduplicates(table1, 'foo')
>>> table2
+-----+-----+-----+
| foo | bar          | baz          |
+=====+=====+=====+
| 'A' | Conflict({1, 2}) |          2.7 |
```

'B'	2	7.8
'D'	3	Conflict({9.4, 12.3})
'E'	None	42.0

Missing values are overridden by non-missing values. Conflicting values are reported as an instance of the Conflict class (sub-class of frozenset).

If *presorted* is True, it is assumed that the data are already sorted by the given key, and the *bufferize*, *tempdir* and *cache* arguments are ignored. Otherwise, the data are sorted, see also the discussion of the *bufferize*, *tempdir* and *cache* arguments under the `petl.transform.sorts.sort()` function.

See also `petl.transform.dedup.conflicts()`.

`petl.transform.reductions.merge(*tables, **kwargs)`

Convenience function to combine multiple tables (via `petl.transform.sorts.mergesort()`) then combine duplicate rows by merging under the given key (via `petl.transform.reductions.mergeduplicates()`). E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          [1, 'A', True],
...          [2, 'B', None],
...          [4, 'C', True]]
>>> table2 = [['bar', 'baz', 'quux'],
...          ['A', True, 42.0],
...          ['B', False, 79.3],
...          ['C', False, 12.4]]
>>> table3 = etl.merge(table1, table2, key='bar')
>>> table3
+-----+-----+-----+-----+
| bar | foo | baz | quux |
+-----+-----+-----+-----+
| 'A' | 1 | True | 42.0 |
+-----+-----+-----+-----+
| 'B' | 2 | False | 79.3 |
+-----+-----+-----+-----+
| 'C' | 4 | Conflict({False, True}) | 12.4 |
+-----+-----+-----+-----+
```

Keyword arguments are the same as for `petl.transform.sorts.mergesort()`, except *key* is required.

`petl.transform.reductions.fold(table, key, f, value=None, presorted=False, bufferize=None, tempdir=None, cache=True)`

Reduce rows recursively via the Python standard `reduce()` function. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'count'],
...          [1, 3],
...          [1, 5],
...          [2, 4],
...          [2, 8]]
>>> import operator
>>> table2 = etl.fold(table1, 'id', operator.add, 'count',
...                  presorted=True)
>>> table2
```

```
+-----+-----+
| key | value |
+=====+=====+
|  1  |    8  |
+-----+-----+
|  2  |   12  |
+-----+-----+
```

See also `petl.transform.reductions.aggregate()`, `petl.transform.reductions.rowreduce()`.

`petl.transform.reductions.groupcountdistinctvalues` (*table, key, value*)

Group by the *key* field then count the number of distinct values in the *value* field.

`petl.transform.reductions.groupselectfirst` (*table, key, presorted=False, buffersize=None, tempdir=None, cache=True*)

Group by the *key* field then return the first row within each group. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, True],
...          ['C', 7, False],
...          ['B', 2, False],
...          ['C', 9, True]]
>>> table2 = etl.groupselectfirst(table1, key='foo')
>>> table2
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'A' |  1 | True |
+-----+-----+-----+
| 'B' |  2 | False|
+-----+-----+-----+
| 'C' |  7 | False|
+-----+-----+-----+
```

See also `petl.transform.reductions.groupselectlast()`, `petl.transform.dedup.distinct()`.

`petl.transform.reductions.groupselectlast` (*table, key, presorted=False, buffersize=None, tempdir=None, cache=True*)

Group by the *key* field then return the last row within each group. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, True],
...          ['C', 7, False],
...          ['B', 2, False],
...          ['C', 9, True]]
>>> table2 = etl.groupselectlast(table1, key='foo')
>>> table2
+-----+-----+-----+
| foo | bar | baz  |
+=====+=====+=====+
| 'A' |  1 | True |
+-----+-----+-----+
| 'B' |  2 | False|
+-----+-----+-----+
```

```
| 'C' | 9 | True |
+-----+-----+
```

See also `petl.transform.reductions.groupselectfirst()`, `petl.transform.dedup.distinct()`.

New in version 1.1.0.

`petl.transform.reductions.groupselectmin(table, key, value, presorted=False, buffer-size=None, tempdir=None, cache=True)`

Group by the *key* field then return the row with the minimum of the *value* field within each group. N.B., will only return one row for each group, even if multiple rows have the same (minimum) value.

`petl.transform.reductions.groupselectmax(table, key, value, presorted=False, buffer-size=None, tempdir=None, cache=True)`

Group by the *key* field then return the row with the maximum of the *value* field within each group. N.B., will only return one row for each group, even if multiple rows have the same (maximum) value.

## Reshaping tables

`petl.transform.reshape.melt(table, key=None, variables=None, variablefield='variable', valuefield='value')`

Reshape a table, melting fields into data. E.g.:

```
>>> import petl as etl
>>> table1 = [['id', 'gender', 'age'],
...          [1, 'F', 12],
...          [2, 'M', 17],
...          [3, 'M', 16]]
>>> table2 = etl.melt(table1, 'id')
>>> table2.lookall()
+-----+-----+-----+
| id | variable | value |
+-----+-----+-----+
| 1 | 'gender' | 'F'   |
+-----+-----+-----+
| 1 | 'age'    | 12   |
+-----+-----+-----+
| 2 | 'gender' | 'M'   |
+-----+-----+-----+
| 2 | 'age'    | 17   |
+-----+-----+-----+
| 3 | 'gender' | 'M'   |
+-----+-----+-----+
| 3 | 'age'    | 16   |
+-----+-----+-----+

>>> # compound keys are supported
... table3 = [['id', 'time', 'height', 'weight'],
...          [1, 11, 66.4, 12.2],
...          [2, 16, 53.2, 17.3],
...          [3, 12, 34.5, 9.4]]
>>> table4 = etl.melt(table3, key=['id', 'time'])
>>> table4.lookall()
+-----+-----+-----+
| id | time | variable | value |
+-----+-----+-----+
+-----+-----+-----+
```

```

| 1 | 11 | 'height' | 66.4 |
+----+-----+-----+-----+
| 1 | 11 | 'weight' | 12.2 |
+----+-----+-----+-----+
| 2 | 16 | 'height' | 53.2 |
+----+-----+-----+-----+
| 2 | 16 | 'weight' | 17.3 |
+----+-----+-----+-----+
| 3 | 12 | 'height' | 34.5 |
+----+-----+-----+-----+
| 3 | 12 | 'weight' | 9.4 |
+----+-----+-----+-----+

>>> # a subset of variable fields can be selected
... table5 = etl.melt(table3, key=['id', 'time'],
...                 variables=['height'])
>>> table5.lookall()
+----+-----+-----+-----+
| id | time | variable | value |
+====+=====+=====+=====+
| 1 | 11 | 'height' | 66.4 |
+----+-----+-----+-----+
| 2 | 16 | 'height' | 53.2 |
+----+-----+-----+-----+
| 3 | 12 | 'height' | 34.5 |
+----+-----+-----+-----+

```

See also `petl.transform.reshape.recast()`.

`petl.transform.reshape.recast` (*table*, *key=None*, *variablefield='variable'*, *valuefield='value'*, *samplesize=1000*, *reducers=None*, *missing=None*)

Recast molten data. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'variable', 'value'],
...          [3, 'age', 16],
...          [1, 'gender', 'F'],
...          [2, 'gender', 'M'],
...          [2, 'age', 17],
...          [1, 'age', 12],
...          [3, 'gender', 'M']]
>>> table2 = etl.recast(table1)
>>> table2
+----+-----+-----+
| id | age | gender |
+====+=====+=====+
| 1 | 12 | 'F'    |
+----+-----+-----+
| 2 | 17 | 'M'    |
+----+-----+-----+
| 3 | 16 | 'M'    |
+----+-----+-----+

>>> # specifying variable and value fields
... table3 = [['id', 'vars', 'vals'],
...          [3, 'age', 16],
...          [1, 'gender', 'F'],
...          [2, 'gender', 'M'],

```



```

...         [2, 'age', 17],
...         [1, 'age', 12],
...         [3, 'gender', 'M']]
>>> table4 = etl.recast(table3, variablefield='vars', valuefield='vals')
>>> table4
+----+-----+-----+
| id | age | gender |
+====+=====+=====+
| 1 | 12 | 'F'    |
+----+-----+-----+
| 2 | 17 | 'M'    |
+----+-----+-----+
| 3 | 16 | 'M'    |
+----+-----+-----+

>>> # if there are multiple values for each key/variable pair, and no
... # reducers function is provided, then all values will be listed
... table6 = [['id', 'time', 'variable', 'value'],
...          [1, 11, 'weight', 66.4],
...          [1, 14, 'weight', 55.2],
...          [2, 12, 'weight', 53.2],
...          [2, 16, 'weight', 43.3],
...          [3, 12, 'weight', 34.5],
...          [3, 17, 'weight', 49.4]]
>>> table7 = etl.recast(table6, key='id')
>>> table7
+----+-----+-----+
| id | weight          |
+====+=====+=====+
| 1 | [66.4, 55.2] |
+----+-----+-----+
| 2 | [53.2, 43.3] |
+----+-----+-----+
| 3 | [34.5, 49.4] |
+----+-----+-----+

>>> # multiple values can be reduced via an aggregation function
... def mean(values):
...     return float(sum(values)) / len(values)
...
>>> table8 = etl.recast(table6, key='id', reducers={'weight': mean})
>>> table8
+----+-----+-----+
| id | weight          |
+====+=====+=====+
| 1 | 60.800000000000004 |
+----+-----+-----+
| 2 | 48.25 |
+----+-----+-----+
| 3 | 41.95 |
+----+-----+-----+

>>> # missing values are padded with whatever is provided via the
... # missing keyword argument (None by default)
... table9 = [['id', 'variable', 'value'],
...          [1, 'gender', 'F'],
...          [2, 'age', 17],
...          [1, 'age', 12],

```

```

...         [3, 'gender', 'M']]
>>> table10 = etl.recast(table9, key='id')
>>> table10
+-----+-----+-----+
| id | age | gender |
+=====+=====+=====+
| 1 | 12 | 'F' |
+-----+-----+-----+
| 2 | 17 | None |
+-----+-----+-----+
| 3 | None | 'M' |
+-----+-----+-----+

```

Note that the table is scanned once to discover variables, then a second time to reshape the data and recast variables as fields. How many rows are scanned in the first pass is determined by the *samplesize* argument.

See also `petl.transform.reshape.melt()`.

`petl.transform.reshape.transpose(table)`

Transpose rows into columns. E.g.:

```

>>> import petl as etl
>>> table1 = [['id', 'colour'],
...          [1, 'blue'],
...          [2, 'red'],
...          [3, 'purple'],
...          [5, 'yellow'],
...          [7, 'orange']]
>>> table2 = etl.transpose(table1)
>>> table2
+-----+-----+-----+-----+-----+-----+
| id      | 1      | 2      | 3      | 5      | 7      |
+=====+=====+=====+=====+=====+=====+
| 'colour' | 'blue' | 'red'  | 'purple' | 'yellow' | 'orange' |
+-----+-----+-----+-----+-----+-----+

```

See also `petl.transform.reshape.recast()`.

`petl.transform.reshape.pivot(table, f1, f2, f3, aggfun, missing=None, presorted=False, buffer-size=None, tempdir=None, cache=True)`

Construct a pivot table. E.g.:

```

>>> import petl as etl
>>> table1 = [['region', 'gender', 'style', 'units'],
...          ['east', 'boy', 'tee', 12],
...          ['east', 'boy', 'golf', 14],
...          ['east', 'boy', 'fancy', 7],
...          ['east', 'girl', 'tee', 3],
...          ['east', 'girl', 'golf', 8],
...          ['east', 'girl', 'fancy', 18],
...          ['west', 'boy', 'tee', 12],
...          ['west', 'boy', 'golf', 15],
...          ['west', 'boy', 'fancy', 8],
...          ['west', 'girl', 'tee', 6],
...          ['west', 'girl', 'golf', 16],
...          ['west', 'girl', 'fancy', 1]]
>>> table2 = etl.pivot(table1, 'region', 'gender', 'units', sum)
>>> table2
+-----+-----+-----+

```

```

| region | boy | girl |
+-----+-----+-----+
| 'east' | 33 | 29 |
+-----+-----+-----+
| 'west' | 35 | 23 |
+-----+-----+-----+

>>> table3 = etl.pivot(table1, 'region', 'style', 'units', sum)
>>> table3
+-----+-----+-----+-----+
| region | fancy | golf | tee |
+-----+-----+-----+-----+
| 'east' | 25 | 22 | 15 |
+-----+-----+-----+-----+
| 'west' | 9 | 31 | 18 |
+-----+-----+-----+-----+

>>> table4 = etl.pivot(table1, 'gender', 'style', 'units', sum)
>>> table4
+-----+-----+-----+-----+
| gender | fancy | golf | tee |
+-----+-----+-----+-----+
| 'boy' | 15 | 29 | 24 |
+-----+-----+-----+-----+
| 'girl' | 19 | 24 | 9 |
+-----+-----+-----+-----+

```

See also `petl.transform.reshape.recast()`.

`petl.transform.reshape.flatten(table)`

Convert a table to a sequence of values in row-major order. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['A', 1, True],
...          ['C', 7, False],
...          ['B', 2, False],
...          ['C', 9, True]]
>>> list(etl.flatten(table1))
['A', 1, True, 'C', 7, False, 'B', 2, False, 'C', 9, True]

```

See also `petl.transform.reshape.unflatten()`.

`petl.transform.reshape.unflatten(*args, **kwargs)`

Convert a sequence of values in row-major order into a table. E.g.:

```

>>> import petl as etl
>>> a = ['A', 1, True, 'C', 7, False, 'B', 2, False, 'C', 9]
>>> table1 = etl.unflatten(a, 3)
>>> table1
+-----+-----+-----+
| f0 | f1 | f2 |
+-----+-----+-----+
| 'A' | 1 | True |
+-----+-----+-----+
| 'C' | 7 | False |
+-----+-----+-----+
| 'B' | 2 | False |

```

```

+-----+-----+-----+
| 'C' | 9 | None |
+-----+-----+-----+

>>> # a table and field name can also be provided as arguments
... table2 = [['lines'],
...           ['A'],
...           [1],
...           [True],
...           ['C'],
...           [7],
...           [False],
...           ['B'],
...           [2],
...           [False],
...           ['C'],
...           [9]]
>>> table3 = etl.unflatten(table2, 'lines', 3)
>>> table3
+-----+-----+-----+
| f0 | f1 | f2 |
+=====+=====+=====+
| 'A' | 1 | True |
+-----+-----+-----+
| 'C' | 7 | False |
+-----+-----+-----+
| 'B' | 2 | False |
+-----+-----+-----+
| 'C' | 9 | None |
+-----+-----+-----+

```

See also `petl.transform.reshape.flatten()`.

## Filling missing values

`petl.transform.fills.filldown` (*table*, *\*fields*, *\*\*kwargs*)

Replace missing values with non-missing values from the row above. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...           [1, 'a', None],
...           [1, None, .23],
...           [1, 'b', None],
...           [2, None, None],
...           [2, None, .56],
...           [2, 'c', None],
...           [None, 'c', .72]]
>>> table2 = etl.filldown(table1)
>>> table2.lookall()
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'a' | None |
+-----+-----+-----+
| 1 | 'a' | 0.23 |
+-----+-----+-----+

```

```

| 1 | 'b' | 0.23 |
+-----+-----+-----+
| 2 | 'b' | 0.23 |
+-----+-----+-----+
| 2 | 'b' | 0.56 |
+-----+-----+-----+
| 2 | 'c' | 0.56 |
+-----+-----+-----+
| 2 | 'c' | 0.72 |
+-----+-----+-----+

>>> table3 = etl.filldown(table1, 'bar')
>>> table3.lookall()
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'a' | None |
+-----+-----+-----+
| 1 | 'a' | 0.23 |
+-----+-----+-----+
| 1 | 'b' | None |
+-----+-----+-----+
| 2 | 'b' | None |
+-----+-----+-----+
| 2 | 'b' | 0.56 |
+-----+-----+-----+
| 2 | 'c' | None |
+-----+-----+-----+
| None | 'c' | 0.72 |
+-----+-----+-----+

>>> table4 = etl.filldown(table1, 'bar', 'baz')
>>> table4.lookall()
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'a' | None |
+-----+-----+-----+
| 1 | 'a' | 0.23 |
+-----+-----+-----+
| 1 | 'b' | 0.23 |
+-----+-----+-----+
| 2 | 'b' | 0.23 |
+-----+-----+-----+
| 2 | 'b' | 0.56 |
+-----+-----+-----+
| 2 | 'c' | 0.56 |
+-----+-----+-----+
| None | 'c' | 0.72 |
+-----+-----+-----+

```

Use the *missing* keyword argument to control which value is treated as missing (*None* by default).

`petl.transform.fills.fillright` (*table*, *missing=None*)

Replace missing values with preceding non-missing values. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],

```

```

...         [1, 'a', None],
...         [1, None, .23],
...         [1, 'b', None],
...         [2, None, None],
...         [2, None, .56],
...         [2, 'c', None],
...         [None, 'c', .72]]
>>> table2 = etl.fillright(table1)
>>> table2.lookall()
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'a' | 'a' |
+-----+-----+-----+
| 1 | 1 | 0.23 |
+-----+-----+-----+
| 1 | 'b' | 'b' |
+-----+-----+-----+
| 2 | 2 | 2 |
+-----+-----+-----+
| 2 | 2 | 0.56 |
+-----+-----+-----+
| 2 | 'c' | 'c' |
+-----+-----+-----+
| None | 'c' | 0.72 |
+-----+-----+-----+

```

Use the *missing* keyword argument to control which value is treated as missing (*None* by default).

`petl.transform.fills.filleft` (*table*, *missing=None*)

Replace missing values with following non-missing values. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          [1, 'a', None],
...          [1, None, .23],
...          [1, 'b', None],
...          [2, None, None],
...          [2, None, .56],
...          [2, 'c', None],
...          [None, 'c', .72]]
>>> table2 = etl.filleft(table1)
>>> table2.lookall()
+-----+-----+-----+
| foo | bar | baz |
+=====+=====+=====+
| 1 | 'a' | None |
+-----+-----+-----+
| 1 | 0.23 | 0.23 |
+-----+-----+-----+
| 1 | 'b' | None |
+-----+-----+-----+
| 2 | None | None |
+-----+-----+-----+
| 2 | 0.56 | 0.56 |
+-----+-----+-----+
| 2 | 'c' | None |
+-----+-----+-----+

```

```
| 'c' | 'c' | 0.72 |
+-----+-----+-----+
```

Use the *missing* keyword argument to control which value is treated as missing (*None* by default).

## Validation

`petl.transform.validation.validate` (*table*, *constraints=None*, *header=None*)

Validate a *table* against a set of *constraints* and/or an expected *header*, e.g.:

```
>>> import petl as etl
>>> # define some validation constraints
... header = ('foo', 'bar', 'baz')
>>> constraints = [
...     dict(name='foo_int', field='foo', test=int),
...     dict(name='bar_date', field='bar', test=etl.dateparser('%Y-%m-%d')),
...     dict(name='baz_enum', field='baz', assertion=lambda v: v in ['Y', 'N']),
...     dict(name='not_none', assertion=lambda row: None not in row)
...     dict(name='qux_int', field='qux', test=int, optional=True),
... ]
>>> # now validate a table
... table = (('foo', 'bar', 'bazzz'),
...          (1, '2000-01-01', 'Y'),
...          ('x', '2010-10-10', 'N'),
...          (2, '2000/01/01', 'Y'),
...          (3, '2015-12-12', 'x'),
...          (4, None, 'N'),
...          ('y', '1999-99-99', 'z'),
...          (6, '2000-01-01'),
...          (7, '2001-02-02', 'N', True))
>>> problems = etl.validate(table, constraints=constraints, header=header)
>>> problems.lookall()
+-----+-----+-----+-----+-----+
| name          | row | field | value      | error              |
+=====+=====+=====+=====+=====+
| '__header__' | 0  | None  | None       | 'AssertionError'  |
+-----+-----+-----+-----+-----+
| 'foo_int'     | 2  | 'foo' | 'x'        | 'ValueError'      |
+-----+-----+-----+-----+-----+
| 'bar_date'    | 3  | 'bar' | '2000/01/01' | 'ValueError'      |
+-----+-----+-----+-----+-----+
| 'baz_enum'    | 4  | 'baz' | 'x'        | 'AssertionError'  |
+-----+-----+-----+-----+-----+
| 'bar_date'    | 5  | 'bar' | None       | 'AttributeError'  |
+-----+-----+-----+-----+-----+
| 'not_none'    | 5  | None  | None       | 'AssertionError'  |
+-----+-----+-----+-----+-----+
| 'foo_int'     | 6  | 'foo' | 'y'        | 'ValueError'      |
+-----+-----+-----+-----+-----+
| 'bar_date'    | 6  | 'bar' | '1999-99-99' | 'ValueError'      |
+-----+-----+-----+-----+-----+
| 'baz_enum'    | 6  | 'baz' | 'z'        | 'AssertionError'  |
+-----+-----+-----+-----+-----+
| '__len__'     | 7  | None  |            | 2 | 'AssertionError'  |
+-----+-----+-----+-----+-----+
| 'baz_enum'    | 7  | 'baz' | None       | 'AssertionError'  |
```

```
+-----+-----+-----+-----+-----+
| '__len__' | 8 | None | 4 | 'AssertionError' |
+-----+-----+-----+-----+-----+
```

Returns a table of validation problems.

## Intervals (intervaltree)

**Note:** The following functions require the package `intervaltree` to be installed, e.g.:

```
$ pip install intervaltree
```

`petl.transform.intervals.intervaljoin` (*left*, *right*, *lstart*='start', *lstop*='stop', *rstart*='start', *rstop*='stop', *lkey*=None, *rkey*=None, *include\_stop*=False, *lprefix*=None, *rprefix*=None)

Join two tables by overlapping intervals. E.g.:

```
>>> import petl as etl
>>> left = [['begin', 'end', 'quux'],
...        [1, 2, 'a'],
...        [2, 4, 'b'],
...        [2, 5, 'c'],
...        [9, 14, 'd'],
...        [1, 1, 'e'],
...        [10, 10, 'f']]
>>> right = [['start', 'stop', 'value'],
...          [1, 4, 'foo'],
...          [3, 7, 'bar'],
...          [4, 9, 'baz']]
>>> table1 = etl.intervaljoin(left, right,
...                           lstart='begin', lstop='end',
...                           rstart='start', rstop='stop')
>>> table1.lookall()
+-----+-----+-----+-----+-----+
| begin | end | quux | start | stop | value |
+-----+-----+-----+-----+-----+
| 1     | 2  | 'a'  | 1     | 4   | 'foo' |
+-----+-----+-----+-----+-----+
| 2     | 4  | 'b'  | 1     | 4   | 'foo' |
+-----+-----+-----+-----+-----+
| 2     | 4  | 'b'  | 3     | 7   | 'bar' |
+-----+-----+-----+-----+-----+
| 2     | 5  | 'c'  | 1     | 4   | 'foo' |
+-----+-----+-----+-----+-----+
| 2     | 5  | 'c'  | 3     | 7   | 'bar' |
+-----+-----+-----+-----+-----+
| 2     | 5  | 'c'  | 4     | 9   | 'baz' |
+-----+-----+-----+-----+-----+

>>> # include stop coordinate in intervals
... table2 = etl.intervaljoin(left, right,
...                           lstart='begin', lstop='end',
...                           rstart='start', rstop='stop',
...                           include_stop=True)
```



```
>>> table2.lookall()
+-----+-----+-----+-----+
| begin | end | quux | start | stop | value |
+=====+=====+=====+=====+
| 1 | 2 | 'a' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+
| 2 | 4 | 'b' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+
| 2 | 4 | 'b' | 3 | 7 | 'bar' |
+-----+-----+-----+-----+
| 2 | 4 | 'b' | 4 | 9 | 'baz' |
+-----+-----+-----+-----+
| 2 | 5 | 'c' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+
| 2 | 5 | 'c' | 3 | 7 | 'bar' |
+-----+-----+-----+-----+
| 2 | 5 | 'c' | 4 | 9 | 'baz' |
+-----+-----+-----+-----+
| 9 | 14 | 'd' | 4 | 9 | 'baz' |
+-----+-----+-----+-----+
| 1 | 1 | 'e' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+
```

Note start coordinates are included and stop coordinates are excluded from the interval. Use the *include\_stop* keyword argument to include the upper bound of the interval when finding overlaps.

An additional key comparison can be made, e.g.:

```
>>> import petl as etl
>>> left = (('fruit', 'begin', 'end'),
...        ('apple', 1, 2),
...        ('apple', 2, 4),
...        ('apple', 2, 5),
...        ('orange', 2, 5),
...        ('orange', 9, 14),
...        ('orange', 19, 140),
...        ('apple', 1, 1))
>>> right = (('type', 'start', 'stop', 'value'),
...          ('apple', 1, 4, 'foo'),
...          ('apple', 3, 7, 'bar'),
...          ('orange', 4, 9, 'baz'))
>>> table3 = etl.intervaljoin(left, right,
...                            lstart='begin', lstop='end', lkey='fruit',
...                            rstart='start', rstop='stop', rkey='type')
>>> table3.lookall()
+-----+-----+-----+-----+-----+-----+
| fruit  | begin | end | type   | start | stop | value |
+=====+=====+=====+=====+=====+=====+
| 'apple' | 1 | 2 | 'apple' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+-----+
| 'apple' | 2 | 4 | 'apple' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+-----+
| 'apple' | 2 | 4 | 'apple' | 3 | 7 | 'bar' |
+-----+-----+-----+-----+-----+
| 'apple' | 2 | 5 | 'apple' | 1 | 4 | 'foo' |
+-----+-----+-----+-----+-----+
| 'apple' | 2 | 5 | 'apple' | 3 | 7 | 'bar' |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+
```

```
| 'orange' |      2 |      5 | 'orange' |      4 |      9 | 'baz' |
+-----+-----+-----+-----+-----+-----+
```

`petl.transform.intervals.intervalleftjoin(left, right, lstart='start', lstop='stop', rstart='start', rstop='stop', lkey=None, rkey=None, include_stop=False, missing=None, lprefix=None, rprefix=None)`

Like `petl.transform.intervals.intervaljoin()` but rows from the left table without a match in the right table are also included. E.g.:

```
>>> import petl as etl
>>> left = [['begin', 'end', 'quux'],
...        [1, 2, 'a'],
...        [2, 4, 'b'],
...        [2, 5, 'c'],
...        [9, 14, 'd'],
...        [1, 1, 'e'],
...        [10, 10, 'f']]
>>> right = [['start', 'stop', 'value'],
...          [1, 4, 'foo'],
...          [3, 7, 'bar'],
...          [4, 9, 'baz']]
>>> table1 = etl.intervalleftjoin(left, right,
...                               lstart='begin', lstop='end',
...                               rstart='start', rstop='stop')
>>> table1.lookall()
+-----+-----+-----+-----+-----+-----+
| begin | end | quux | start | stop | value |
+-----+-----+-----+-----+-----+-----+
|      1 |  2 | 'a'  |      1 |   4 | 'foo' |
+-----+-----+-----+-----+-----+-----+
|      2 |  4 | 'b'  |      1 |   4 | 'foo' |
+-----+-----+-----+-----+-----+-----+
|      2 |  4 | 'b'  |      3 |   7 | 'bar' |
+-----+-----+-----+-----+-----+-----+
|      2 |  5 | 'c'  |      1 |   4 | 'foo' |
+-----+-----+-----+-----+-----+-----+
|      2 |  5 | 'c'  |      3 |   7 | 'bar' |
+-----+-----+-----+-----+-----+-----+
|      2 |  5 | 'c'  |      4 |   9 | 'baz' |
+-----+-----+-----+-----+-----+-----+
|      9 | 14 | 'd'  | None  | None | None  |
+-----+-----+-----+-----+-----+-----+
|      1 |  1 | 'e'  | None  | None | None  |
+-----+-----+-----+-----+-----+-----+
|     10 | 10 | 'f'  | None  | None | None  |
+-----+-----+-----+-----+-----+-----+
```

Note start coordinates are included and stop coordinates are excluded from the interval. Use the `include_stop` keyword argument to include the upper bound of the interval when finding overlaps.

`petl.transform.intervals.intervaljoinvalues(left, right, value, lstart='start', lstop='stop', rstart='start', rstop='stop', lkey=None, rkey=None, include_stop=False)`

Convenience function to join the left table with values from a specific field in the right hand table.

Note start coordinates are included and stop coordinates are excluded from the interval. Use the `include_stop` keyword argument to include the upper bound of the interval when finding overlaps.

```
petl.transform.intervals.intervalantijoin(left, right, lstart='start', lstop='stop',
                                             rstart='start', rstop='stop', lkey=None,
                                             rkey=None, include_stop=False, miss-
                                             ing=None)
```

Return rows from the *left* table with no overlapping rows from the *right* table.

Note start coordinates are included and stop coordinates are excluded from the interval. Use the *include\_stop* keyword argument to include the upper bound of the interval when finding overlaps.

```
petl.transform.intervals.intervallookup(table, start='start', stop='stop', value=None, in-
                                          clude_stop=False)
```

Construct an interval lookup for the given table. E.g.:

```
>>> import petl as etl
>>> table = [['start', 'stop', 'value'],
...         [1, 4, 'foo'],
...         [3, 7, 'bar'],
...         [4, 9, 'baz']]
>>> lkp = etl.intervallookup(table, 'start', 'stop')
>>> lkp.search(0, 1)
[]
>>> lkp.search(1, 2)
[(1, 4, 'foo')]
>>> lkp.search(2, 4)
[(1, 4, 'foo'), (3, 7, 'bar')]
>>> lkp.search(2, 5)
[(1, 4, 'foo'), (3, 7, 'bar'), (4, 9, 'baz')]
>>> lkp.search(9, 14)
[]
>>> lkp.search(19, 140)
[]
>>> lkp.search(0)
[]
>>> lkp.search(1)
[(1, 4, 'foo')]
>>> lkp.search(2)
[(1, 4, 'foo')]
>>> lkp.search(4)
[(3, 7, 'bar'), (4, 9, 'baz')]
>>> lkp.search(5)
[(3, 7, 'bar'), (4, 9, 'baz')]
```

Note start coordinates are included and stop coordinates are excluded from the interval. Use the *include\_stop* keyword argument to include the upper bound of the interval when finding overlaps.

Some examples using the *include\_stop* and *value* keyword arguments:

```
>>> import petl as etl
>>> table = [['start', 'stop', 'value'],
...         [1, 4, 'foo'],
...         [3, 7, 'bar'],
...         [4, 9, 'baz']]
>>> lkp = etl.intervallookup(table, 'start', 'stop', include_stop=True,
...                           value='value')
>>> lkp.search(0, 1)
['foo']
>>> lkp.search(1, 2)
['foo']
>>> lkp.search(2, 4)
```

```

['foo', 'bar', 'baz']
>>> lkp.search(2, 5)
['foo', 'bar', 'baz']
>>> lkp.search(9, 14)
['baz']
>>> lkp.search(19, 140)
[]
>>> lkp.search(0)
[]
>>> lkp.search(1)
['foo']
>>> lkp.search(2)
['foo']
>>> lkp.search(4)
['foo', 'bar', 'baz']
>>> lkp.search(5)
['bar', 'baz']

```

`petl.transform.intervals.intervallookupone` (*table*, *start*='start', *stop*='stop', *value*=None, *include\_stop*=False, *strict*=True)

Construct an interval lookup for the given table, returning at most one result for each query. E.g.:

```

>>> import petl as etl
>>> table = [['start', 'stop', 'value'],
...         [1, 4, 'foo'],
...         [3, 7, 'bar'],
...         [4, 9, 'baz']]
>>> lkp = etl.intervallookupone(table, 'start', 'stop', strict=False)
>>> lkp.search(0, 1)
>>> lkp.search(1, 2)
(1, 4, 'foo')
>>> lkp.search(2, 4)
(1, 4, 'foo')
>>> lkp.search(2, 5)
(1, 4, 'foo')
>>> lkp.search(9, 14)
>>> lkp.search(19, 140)
>>> lkp.search(0)
>>> lkp.search(1)
(1, 4, 'foo')
>>> lkp.search(2)
(1, 4, 'foo')
>>> lkp.search(4)
(3, 7, 'bar')
>>> lkp.search(5)
(3, 7, 'bar')

```

If *strict*=True, queries returning more than one result will raise a *DuplicateKeyError*. If *strict*=False and there is more than one result, the first result is returned.

Note start coordinates are included and stop coordinates are excluded from the interval. Use the *include\_stop* keyword argument to include the upper bound of the interval when finding overlaps.

`petl.transform.intervals.intervalrecordlookup` (*table*, *start*='start', *stop*='stop', *include\_stop*=False)

As `petl.transform.intervals.intervallookup()` but return records instead of tuples.

`petl.transform.intervals.intervalrecordlookupone` (*table*, *start*='start', *stop*='stop', *include\_stop*=False, *strict*=True)

As `petl.transform.intervals.intervallookupone()` but return records instead of tuples.

`petl.transform.intervals.facetintervallookup` (*table*, *key*, *start*='start', *stop*='stop', *value*=None, *include\_stop*=False)

Construct a faceted interval lookup for the given table. E.g.:

```
>>> import petl as etl
>>> table = (('type', 'start', 'stop', 'value'),
...         ('apple', 1, 4, 'foo'),
...         ('apple', 3, 7, 'bar'),
...         ('orange', 4, 9, 'baz'))
>>> lkp = etl.facetintervallookup(table, key='type', start='start', stop='stop')
>>> lkp['apple'].search(1, 2)
[('apple', 1, 4, 'foo')]
>>> lkp['apple'].search(2, 4)
[('apple', 1, 4, 'foo'), ('apple', 3, 7, 'bar')]
>>> lkp['apple'].search(2, 5)
[('apple', 1, 4, 'foo'), ('apple', 3, 7, 'bar')]
>>> lkp['orange'].search(2, 5)
[('orange', 4, 9, 'baz')]
>>> lkp['orange'].search(9, 14)
[]
>>> lkp['orange'].search(19, 140)
[]
>>> lkp['apple'].search(1)
[('apple', 1, 4, 'foo')]
>>> lkp['apple'].search(2)
[('apple', 1, 4, 'foo')]
>>> lkp['apple'].search(4)
[('apple', 3, 7, 'bar')]
>>> lkp['apple'].search(5)
[('apple', 3, 7, 'bar')]
>>> lkp['orange'].search(5)
[('orange', 4, 9, 'baz')]
```

`petl.transform.intervals.facetintervallookupone` (*table*, *key*, *start*='start', *stop*='stop', *value*=None, *include\_stop*=False, *strict*=True)

Construct a faceted interval lookup for the given table, returning at most one result for each query.

If `strict=True`, queries returning more than one result will raise a `DuplicateKeyError`. If `strict=False` and there is more than one result, the first result is returned.

`petl.transform.intervals.facetintervalrecordlookup` (*table*, *key*, *start*='start', *stop*='stop', *include\_stop*=False)

As `petl.transform.intervals.facetintervallookup()` but return records.

`petl.transform.intervals.facetintervalrecordlookupone` (*table*, *key*, *start*, *stop*, *include\_stop*=False, *strict*=True)

As `petl.transform.intervals.facetintervallookupone()` but return records.

`petl.transform.intervals.intervalsubtract` (*left*, *right*, *lstart*='start', *lstop*='stop', *rstart*='start', *rstop*='stop', *lkey*=None, *rkey*=None, *include\_stop*=False)

Subtract intervals in the right hand table from intervals in the left hand table.

`petl.transform.intervals.collapsedintervals` (*table*, *start*='start', *stop*='stop', *key*=None)

Utility function to collapse intervals in a table.

If no facet *key* is given, returns an iterator over (*start*, *stop*) tuples.

If facet *key* is given, returns an iterator over (*key*, *start*, *stop*) tuples.

## Basic utilities

`petl.util.base.header` (*table*)

Return the header row for the given table. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> etl.header(table)
('foo', 'bar')
```

Note that the header row will always be returned as a tuple, regardless of what the underlying data are.

`petl.util.base.fieldnames` (*table*)

Return the string values of the header row. If the header row contains only strings, then this function is equivalent to `header()`, i.e.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> etl.fieldnames(table)
('foo', 'bar')
>>> etl.header(table)
('foo', 'bar')
```

`petl.util.base.data` (*table*, \**sliceargs*)

Return a container supporting iteration over data rows in a given table (i.e., without the header). E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> d = etl.data(table)
>>> list(d)
[['a', 1], ['b', 2]]
```

Positional arguments can be used to slice the data rows. The `sliceargs` are passed to `itertools.islice()`.

`petl.util.base.values` (*table*, \**field*, \*\**kwargs*)

Return a container supporting iteration over values in a given field or fields. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', True],
...          ['b'],
...          ['b', True],
...          ['c', False]]
>>> foo = etl.values(table1, 'foo')
>>> foo
foo: 'a', 'b', 'b', 'c'
>>> list(foo)
['a', 'b', 'b', 'c']
>>> bar = etl.values(table1, 'bar')
>>> bar
bar: True, None, True, False
>>> list(bar)
[True, None, True, False]
>>> # values from multiple fields
... table2 = [['foo', 'bar', 'baz'],
...          [1, 'a', True],
...          [2, 'bb', True],
...          [3, 'd', False]]
>>> foobaz = etl.values(table2, 'foo', 'baz')
>>> foobaz
('foo', 'baz'): (1, True), (2, True), (3, False)
>>> list(foobaz)
[(1, True), (2, True), (3, False)]
```

The field argument can be a single field name or index (starting from zero) or a tuple of field names and/or indexes. Multiple fields can also be provided as positional arguments.

If rows are uneven, the value of the keyword argument *missing* is returned.

`petl.util.base.dicts` (*table*, \**sliceargs*, \*\**kwargs*)

Return a container supporting iteration over rows as dicts. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> d = etl.dicts(table)
>>> d
{'foo': 'a', 'bar': 1}
{'foo': 'b', 'bar': 2}
>>> list(d)
[{'foo': 'a', 'bar': 1}, {'foo': 'b', 'bar': 2}]
```

Short rows are padded with the value of the *missing* keyword argument.

`petl.util.base.namedtuples` (*table*, \**sliceargs*, \*\**kwargs*)

View the table as a container of named tuples. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> d = etl.namedtuples(table)
>>> d
row(foo='a', bar=1)
row(foo='b', bar=2)
>>> list(d)
```



```
[row(foo='a', bar=1), row(foo='b', bar=2)]
```

Short rows are padded with the value of the *missing* keyword argument.

The *name* keyword argument can be given to override the name of the named tuple class (defaults to 'row').

`petl.util.base.records` (*table*, *\*sliceargs*, *\*\*kwargs*)

Return a container supporting iteration over rows as records, where a record is a hybrid object supporting all possible ways of accessing values. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> d = etl.records(table)
>>> d
('a', 1)
('b', 2)
>>> list(d)
[('a', 1), ('b', 2)]
>>> [r[0] for r in d]
['a', 'b']
>>> [r['foo'] for r in d]
['a', 'b']
>>> [r.foo for r in d]
['a', 'b']
```

Short rows are padded with the value of the *missing* keyword argument.

`petl.util.base.expr` (*s*)

Construct a function operating on a table record.

The expression string is converted into a lambda function by prepending the string with 'lambda rec:', then replacing anything enclosed in curly braces (e.g., "{foo}") with a lookup on the record (e.g., "rec['foo']"), then finally calling `eval()`.

So, e.g., the expression string "{foo} \* {bar}" is converted to the function `lambda rec: rec['foo'] * rec['bar']`

`petl.util.base.rowgroupby` (*table*, *key*, *value=None*)

Convenient adapter for `itertools.groupby()`. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...           ['a', 1, True],
...           ['b', 3, True],
...           ['b', 2]]
>>> # group entire rows
... for key, group in etl.rowgroupby(table1, 'foo'):
...     print(key, list(group))
...
a [('a', 1, True)]
b [('b', 3, True), ('b', 2)]
>>> # group specific values
... for key, group in etl.rowgroupby(table1, 'foo', 'bar'):
...     print(key, list(group))
...
a [1]
b [3, 2]
```

N.B., assumes the input table is already sorted by the given key.

`petl.util.base.empty()`

Return an empty table. Can be useful when building up a table from a set of columns, e.g.:

```
>>> import petl as etl
>>> table = (
...     etl
...     .empty()
...     .addcolumn('foo', ['A', 'B'])
...     .addcolumn('bar', [1, 2])
... )
>>> table
+-----+-----+
| foo | bar |
+=====+=====+
| 'A' |  1 |
+-----+-----+
| 'B' |  2 |
+-----+-----+
```

## Visualising tables

`petl.util.vis.look(table, limit=0, vrepr=None, index_header=None, style=None, truncate=None, width=None)`

Format a portion of the table as text for inspection in an interactive session. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...           ['a', 1],
...           ['b', 2]]
>>> etl.look(table1)
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' |  1 |
+-----+-----+
| 'b' |  2 |
+-----+-----+

>>> # alternative formatting styles
... etl.look(table1, style='simple')
===  ===
foo  bar
===  ===
'a'  1
'b'  2
===  ===

>>> etl.look(table1, style='minimal')
foo  bar
'a'  1
'b'  2

>>> # any irregularities in the length of header and/or data
... # rows will appear as blank cells
... table2 = [['foo', 'bar'],
...           ['a'],
```

```

...         ['b', 2, True]]
>>> etl.look(table2)
+-----+-----+-----+
| foo | bar |      |
+=====+=====+=====+
| 'a' |     |      |
+-----+-----+-----+
| 'b' |  2 | True |
+-----+-----+-----+

```

Three alternative presentation styles are available: 'grid', 'simple' and 'minimal', where 'grid' is the default. A different style can be specified using the *style* keyword argument. The default style can also be changed by setting `petl.config.look_style`.

`petl.util.vis.lookall` (*table*, *\*\*kwargs*)

Format the entire table as text for inspection in an interactive session.

N.B., this will load the entire table into memory.

See also `petl.util.vis.look()` and `petl.util.vis.see()`.

`petl.util.vis.see` (*table*, *limit=0*, *vrepr=None*, *index\_header=None*)

Format a portion of a table as text in a column-oriented layout for inspection in an interactive session. E.g.:

```

>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> etl.see(table)
foo: 'a', 'b'
bar: 1, 2

```

Useful for tables with a larger number of fields.

`petl.util.vis.display` (*table*, *limit=0*, *vrepr=None*, *index\_header=None*, *caption=None*, *tr\_style=None*, *td\_styles=None*, *encoding=None*, *truncate=None*, *epilogue=None*)

Display a table inline within an IPython notebook.

`petl.util.vis.displayall` (*table*, *\*\*kwargs*)

Display **all rows** from a table inline within an IPython notebook (use with caution, big tables will kill your browser).

## Lookup data structures

`petl.util.lookups.lookup` (*table*, *key*, *value=None*, *dictionary=None*)

Load a dictionary with data from the given table. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['b', 3]]
>>> lkp = etl.lookup(table1, 'foo', 'bar')
>>> lkp['a']
[1]
>>> lkp['b']
[2, 3]
>>> # if no value argument is given, defaults to the whole

```

```

... # row (as a tuple)
... lkp = etl.lookup(table1, 'foo')
>>> lkp['a']
[('a', 1)]
>>> lkp['b']
[('b', 2), ('b', 3)]
>>> # compound keys are supported
... table2 = [['foo', 'bar', 'baz'],
...           ['a', 1, True],
...           ['b', 2, False],
...           ['b', 3, True],
...           ['b', 3, False]]
>>> lkp = etl.lookup(table2, ('foo', 'bar'), 'baz')
>>> lkp[('a', 1)]
[True]
>>> lkp[('b', 2)]
[False]
>>> lkp[('b', 3)]
[True, False]
>>> # data can be loaded into an existing dictionary-like
... # object, including persistent dictionaries created via the
... # shelve module
... import shelve
>>> lkp = shelve.open('example.dat', flag='n')
>>> lkp = etl.lookup(table1, 'foo', 'bar', lkp)
>>> lkp.close()
>>> lkp = shelve.open('example.dat', flag='r')
>>> lkp['a']
[1]
>>> lkp['b']
[2, 3]

```

`petl.util.lookups.lookupone` (*table, key, value=None, dictionary=None, strict=False*)

Load a dictionary with data from the given table, assuming there is at most one value for each key. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...           ['a', 1],
...           ['b', 2],
...           ['b', 3]]
>>> # if the specified key is not unique and strict=False (default),
... # the first value wins
... lkp = etl.lookupone(table1, 'foo', 'bar')
>>> lkp['a']
1
>>> lkp['b']
2
>>> # if the specified key is not unique and strict=True, will raise
... # DuplicateKeyError
... try:
...     lkp = etl.lookupone(table1, 'foo', strict=True)
... except etl.errors.DuplicateKeyError as e:
...     print(e)
...
duplicate key: 'b'
>>> # compound keys are supported
... table2 = [['foo', 'bar', 'baz'],
...           ['a', 1, True],

```

```

...         ['b', 2, False],
...         ['b', 3, True],
...         ['b', 3, False]]
>>> lkp = etl.lookupone(table2, ('foo', 'bar'), 'baz')
>>> lkp[('a', 1)]
True
>>> lkp[('b', 2)]
False
>>> lkp[('b', 3)]
True
>>> # data can be loaded into an existing dictionary-like
... # object, including persistent dictionaries created via the
... # shelve module
... import shelve
>>> lkp = shelve.open('example.dat', flag='n')
>>> lkp = etl.lookupone(table1, 'foo', 'bar', lkp)
>>> lkp.close()
>>> lkp = shelve.open('example.dat', flag='r')
>>> lkp['a']
1
>>> lkp['b']
2

```

`petl.util.lookups.dictlookup` (*table, key, dictionary=None*)

Load a dictionary with data from the given table, mapping to dicts. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['b', 3]]
>>> lkp = etl.dictlookup(table1, 'foo')
>>> lkp['a']
[{'foo': 'a', 'bar': 1}]
>>> lkp['b']
[{'foo': 'b', 'bar': 2}, {'foo': 'b', 'bar': 3}]
>>> # compound keys are supported
... table2 = [['foo', 'bar', 'baz'],
...          ['a', 1, True],
...          ['b', 2, False],
...          ['b', 3, True],
...          ['b', 3, False]]
>>> lkp = etl.dictlookup(table2, ('foo', 'bar'))
>>> lkp[('a', 1)]
[{'foo': 'a', 'baz': True, 'bar': 1}]
>>> lkp[('b', 2)]
[{'foo': 'b', 'baz': False, 'bar': 2}]
>>> lkp[('b', 3)]
[{'foo': 'b', 'baz': True, 'bar': 3}, {'foo': 'b', 'baz': False, 'bar': 3}]
>>> # data can be loaded into an existing dictionary-like
... # object, including persistent dictionaries created via the
... # shelve module
... import shelve
>>> lkp = shelve.open('example.dat', flag='n')
>>> lkp = etl.dictlookup(table1, 'foo', lkp)
>>> lkp.close()
>>> lkp = shelve.open('example.dat', flag='r')
>>> lkp['a']

```

```

[{'foo': 'a', 'bar': 1}]
>>> lkp['b']
[{'foo': 'b', 'bar': 2}, {'foo': 'b', 'bar': 3}]

```

`petl.util.lookups.dictlookupone` (*table*, *key*, *dictionary=None*, *strict=False*)

Load a dictionary with data from the given table, mapping to dicts, assuming there is at most one row for each key. E.g.:

```

>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 2],
...          ['b', 3]]
>>> # if the specified key is not unique and strict=False (default),
... # the first value wins
... lkp = etl.dictlookupone(table1, 'foo')
>>> lkp['a']
{'foo': 'a', 'bar': 1}
>>> lkp['b']
{'foo': 'b', 'bar': 2}
>>> # if the specified key is not unique and strict=True, will raise
... # DuplicateKeyError
... try:
...     lkp = etl.dictlookupone(table1, 'foo', strict=True)
... except etl.errors.DuplicateKeyError as e:
...     print(e)
...
duplicate key: 'b'
>>> # compound keys are supported
... table2 = [['foo', 'bar', 'baz'],
...          ['a', 1, True],
...          ['b', 2, False],
...          ['b', 3, True],
...          ['b', 3, False]]
>>> lkp = etl.dictlookupone(table2, ('foo', 'bar'))
>>> lkp[('a', 1)]
{'foo': 'a', 'baz': True, 'bar': 1}
>>> lkp[('b', 2)]
{'foo': 'b', 'baz': False, 'bar': 2}
>>> lkp[('b', 3)]
{'foo': 'b', 'baz': True, 'bar': 3}
>>> # data can be loaded into an existing dictionary-like
... # object, including persistent dictionaries created via the
... # shelve module
... import shelve
>>> lkp = shelve.open('example.dat', flag='n')
>>> lkp = etl.dictlookupone(table1, 'foo', lkp)
>>> lkp.close()
>>> lkp = shelve.open('example.dat', flag='r')
>>> lkp['a']
{'foo': 'a', 'bar': 1}
>>> lkp['b']
{'foo': 'b', 'bar': 2}

```

`petl.util.lookups.recordlookup` (*table*, *key*, *dictionary=None*)

Load a dictionary with data from the given table, mapping to record objects.

`petl.util.lookups.recordlookupone` (*table*, *key*, *dictionary=None*, *strict=False*)

Load a dictionary with data from the given table, mapping to record objects, assuming there is at most one row for each key.

## Parsing string/text values

`petl.util.parsers.dateparser` (*fmt*, *strict=True*)

Return a function to parse strings as `datetime.date` objects using a given format. E.g.:

```
>>> from petl import dateparser
>>> isodate = dateparser('%Y-%m-%d')
>>> isodate('2002-12-25')
datetime.date(2002, 12, 25)
>>> try:
...     isodate('2002-02-30')
... except ValueError as e:
...     print(e)
...
day is out of range for month
```

If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised.

`petl.util.parsers.timeparser` (*fmt*, *strict=True*)

Return a function to parse strings as `datetime.time` objects using a given format. E.g.:

```
>>> from petl import timeparser
>>> isotime = timeparser('%H:%M:%S')
>>> isotime('00:00:00')
datetime.time(0, 0)
>>> isotime('13:00:00')
datetime.time(13, 0)
>>> try:
...     isotime('12:00:99')
... except ValueError as e:
...     print(e)
...
unconverted data remains: 9
>>> try:
...     isotime('25:00:00')
... except ValueError as e:
...     print(e)
...
time data '25:00:00' does not match format '%H:%M:%S'
```

If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised.

`petl.util.parsers.datetimestrparser` (*fmt*, *strict=True*)

Return a function to parse strings as `datetime.datetime` objects using a given format. E.g.:

```
>>> from petl import datetimestrparser
>>> isodatetimestr = datetimestrparser('%Y-%m-%dT%H:%M:%S')
>>> isodatetimestr('2002-12-25T00:00:00')
datetime.datetime(2002, 12, 25, 0, 0)
>>> try:
...     isodatetimestr('2002-12-25T00:00:99')
... except ValueError as e:
...     print(e)
```

```
...     print(e)
...
unconverted data remains: 9
```

If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised.

`petl.util.parsers.boolparser` (*true\_strings*=(*'true'*, *'t'*, *'yes'*, *'y'*, *'1'*), *false\_strings*=(*'false'*, *'f'*, *'no'*, *'n'*, *'0'*), *case\_sensitive=False*, *strict=True*)

Return a function to parse strings as `bool` objects using a given set of string representations for *True* and *False*. E.g.:

```
>>> from petl import boolparser
>>> mybool = boolparser(true_strings=['yes', 'y'], false_strings=['no', 'n'])
>>> mybool('y')
True
>>> mybool('yes')
True
>>> mybool('Y')
True
>>> mybool('No')
False
>>> try:
...     mybool('foo')
... except ValueError as e:
...     print(e)
...
value is not one of recognised boolean strings: 'foo'
>>> try:
...     mybool('True')
... except ValueError as e:
...     print(e)
...
value is not one of recognised boolean strings: 'true'
```

If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised.

`petl.util.parsers.numparser` (*strict=False*)

Return a function that will attempt to parse the value as a number, trying `int()`, `long()`, `float()` and `complex()` in that order. If all fail, return the value as-is, unless `strict=True`, in which case raise the underlying exception.

## Counting

`petl.util.counting.nrows` (*table*)

Count the number of data rows in a table. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> etl.nrows(table)
2
```

`petl.util.counting.valuecount` (*table*, *field*, *value*, *missing=None*)

Count the number of occurrences of *value* under the given field. Returns the absolute count and relative frequency as a pair. E.g.:



```
>>> import petl as etl
>>> table = [['foo', 'bar'],
...         ['a', 1],
...         ['b', 2],
...         ['b', 7]]
>>> etl.valuecount(table, 'foo', 'b')
(2, 0.6666666666666666)
```

The *field* argument can be a single field name or index (starting from zero) or a tuple of field names and/or indexes.

`petl.util.counting.valuecounter` (*table*, \**field*, \*\**kwargs*)

Find distinct values for the given field and count the number of occurrences. Returns a dict mapping values to counts. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'],
...         ['a', True],
...         ['b'],
...         ['b', True],
...         ['c', False]]
>>> etl.valuecounter(table, 'foo')
Counter({'b': 2, 'c': 1, 'a': 1})
```

The *field* argument can be a single field name or index (starting from zero) or a tuple of field names and/or indexes.

`petl.util.counting.valuecounts` (*table*, \**field*, \*\**kwargs*)

Find distinct values for the given field and count the number and relative frequency of occurrences. Returns a table mapping values to counts, with most common values first. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['a', True, 0.12],
...         ['a', True, 0.17],
...         ['b', False, 0.34],
...         ['b', False, 0.44],
...         ['b']]
>>> etl.valuecounts(table, 'foo')
+-----+-----+-----+
| foo | count | frequency |
+=====+=====+=====+
| 'b' | 3 | 0.6 |
+-----+-----+-----+
| 'a' | 2 | 0.4 |
+-----+-----+-----+

>>> etl.valuecounts(table, 'foo', 'bar')
+-----+-----+-----+-----+
| foo | bar | count | frequency |
+=====+=====+=====+=====+
| 'b' | False | 2 | 0.4 |
+-----+-----+-----+-----+
| 'a' | True | 2 | 0.4 |
+-----+-----+-----+-----+
| 'b' | None | 1 | 0.2 |
+-----+-----+-----+-----+
```

If rows are short, the value of the keyword argument *missing* is counted.

Multiple fields can be given as positional arguments. If multiple fields are given, these are treated as a compound key.

`petl.util.counting.stringpatterncounter` (*table*, *field*)

Profile string patterns in the given field, returning a dict mapping patterns to counts.

`petl.util.counting.stringpatterns` (*table*, *field*)

Profile string patterns in the given field, returning a table of patterns, counts and frequencies. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar'],
...         ['Mr. Foo', '123-1254'],
...         ['Mrs. Bar', '234-1123'],
...         ['Mr. Spo', '123-1254'],
...         [u'Mr. Baz', u'321 1434'],
...         [u'Mrs. Baz', u'321 1434'],
...         ['Mr. Quux', '123-1254-XX']]
>>> etl.stringpatterns(table, 'foo')
+-----+-----+
| pattern | count | frequency |
+-----+-----+
| 'Aa. Aaa' | 3 | 0.5 |
+-----+-----+
| 'Aaa. Aaa' | 2 | 0.3333333333333333 |
+-----+-----+
| 'Aa. Aaaa' | 1 | 0.16666666666666666 |
+-----+-----+

>>> etl.stringpatterns(table, 'bar')
+-----+-----+
| pattern | count | frequency |
+-----+-----+
| '999-9999' | 3 | 0.5 |
+-----+-----+
| '999 9999' | 2 | 0.3333333333333333 |
+-----+-----+
| '999-9999-AA' | 1 | 0.16666666666666666 |
+-----+-----+
```

`petl.util.counting.rowlengths` (*table*)

Report on row lengths found in the table. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['A', 1, 2],
...         ['B', '2', '3.4'],
...         [u'B', u'3', u'7.8', True],
...         ['D', 'xyz', 9.0],
...         ['E', None],
...         ['F', 9]]
>>> etl.rowlengths(table)
+-----+-----+
| length | count |
+-----+-----+
| 3 | 3 |
+-----+-----+
| 2 | 2 |
```

```
+-----+-----+
|         4 |         1 |
+-----+-----+
```

Useful for finding potential problems in data files.

`petl.util.counting.typecounter` (*table, field*)

Count the number of values found for each Python type.

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['A', 1, 2],
...         ['B', u'2', '3.4'],
...         [u'B', u'3', u'7.8', True],
...         ['D', u'xyz', 9.0],
...         ['E', 42]]
>>> etl.typecounter(table, 'foo')
Counter({'str': 5})
>>> etl.typecounter(table, 'bar')
Counter({'str': 3, 'int': 2})
>>> etl.typecounter(table, 'baz')
Counter({'str': 2, 'int': 1, 'NoneType': 1, 'float': 1})
```

The *field* argument can be a field name or index (starting from zero).

`petl.util.counting.typecounts` (*table, field*)

Count the number of values found for each Python type and return a table mapping class names to counts and frequencies. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         [b'A', 1, 2],
...         [b'B', '2', b'3.4'],
...         ['B', '3', '7.8', True],
...         ['D', u'xyz', 9.0],
...         ['E', 42]]
>>> etl.typecounts(table, 'foo')
+-----+-----+-----+
| type   | count | frequency |
+-----+-----+-----+
| 'str'  |     3 |         0.6 |
+-----+-----+-----+
| 'bytes'|     2 |         0.4 |
+-----+-----+-----+

>>> etl.typecounts(table, 'bar')
+-----+-----+-----+
| type   | count | frequency |
+-----+-----+-----+
| 'str'  |     3 |         0.6 |
+-----+-----+-----+
| 'int'  |     2 |         0.4 |
+-----+-----+-----+

>>> etl.typecounts(table, 'baz')
+-----+-----+-----+
| type   | count | frequency |
+-----+-----+-----+
```

'int'   1   0.2
+-----+-----+-----+
'NoneType'   1   0.2
+-----+-----+-----+
'bytes'   1   0.2
+-----+-----+-----+
'float'   1   0.2
+-----+-----+-----+
'str'   1   0.2
+-----+-----+-----+

The *field* argument can be a field name or index (starting from zero).

`petl.util.counting.parsecounter`(*table*, *field*, *parsers*=(('int', <type 'int'>), ('float', <type 'float'>)))

Count the number of *str* or *unicode* values under the given fields that can be parsed as ints, floats or via custom parser functions. Return a pair of *Counter* objects, the first mapping parser names to the number of strings successfully parsed, the second mapping parser names to the number of errors. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['A', 'aaa', 2],
...         ['B', u'2', '3.4'],
...         [u'B', u'3', u'7.8', True],
...         ['D', '3.7', 9.0],
...         ['E', 42]]
>>> counter, errors = etl.parsecounter(table, 'bar')
>>> counter
Counter({'float': 3, 'int': 2})
>>> errors
Counter({'int': 2, 'float': 1})
```

The *field* argument can be a field name or index (starting from zero).

`petl.util.counting.parsecounts`(*table*, *field*, *parsers*=(('int', <type 'int'>), ('float', <type 'float'>)))

Count the number of *str* or *unicode* values that can be parsed as ints, floats or via custom parser functions. Return a table mapping parser names to the number of values successfully parsed and the number of errors. E.g.:

```
>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['A', 'aaa', 2],
...         ['B', u'2', '3.4'],
...         [u'B', u'3', u'7.8', True],
...         ['D', '3.7', 9.0],
...         ['E', 42]]
>>> etl.parsecounts(table, 'bar')
+-----+-----+-----+
| type   | count | errors |
+-----+-----+-----+
| 'float' | 3 | 1 |
+-----+-----+-----+
| 'int'   | 2 | 2 |
+-----+-----+-----+
```

The *field* argument can be a field name or index (starting from zero).

## Timing

`petl.util.timing.progress` (*table*, *batchsize=1000*, *prefix=''*, *out=<open file '<stderr>'*, *mode* *'w'>*)

Report progress on rows passing through. E.g.:

```
>>> import petl as etl
>>> table = etl.dummytable(100000)
>>> table.progress(10000).tocsv('example.csv')
10000 rows in 0.13s (78363 row/s); batch in 0.13s (78363 row/s)
20000 rows in 0.22s (91679 row/s); batch in 0.09s (110448 row/s)
30000 rows in 0.31s (96573 row/s); batch in 0.09s (108114 row/s)
40000 rows in 0.40s (99535 row/s); batch in 0.09s (109625 row/s)
50000 rows in 0.49s (101396 row/s); batch in 0.09s (109591 row/s)
60000 rows in 0.59s (102245 row/s); batch in 0.09s (106709 row/s)
70000 rows in 0.68s (103221 row/s); batch in 0.09s (109498 row/s)
80000 rows in 0.77s (103810 row/s); batch in 0.09s (108126 row/s)
90000 rows in 0.90s (99465 row/s); batch in 0.13s (74516 row/s)
100000 rows in 1.02s (98409 row/s); batch in 0.11s (89821 row/s)
100000 rows in 1.02s (98402 row/s); batches in 0.10 +/- 0.02s [0.09-0.13] (100481_
↪+/- 13340 rows/s [74516-110448])
```

See also `petl.util.timing.clock()`.

`petl.util.timing.clock` (*table*)

Time how long is spent retrieving rows from the wrapped container. Enables diagnosis of which steps in a pipeline are taking the most time. E.g.:

```
>>> import petl as etl
>>> t1 = etl.dummytable(100000)
>>> c1 = etl.clock(t1)
>>> t2 = etl.convert(c1, 'foo', lambda v: v**2)
>>> c2 = etl.clock(t2)
>>> p = etl.progress(c2, 10000)
>>> etl.tocsv(p, 'example.csv')
10000 rows in 0.23s (44036 row/s); batch in 0.23s (44036 row/s)
20000 rows in 0.38s (52167 row/s); batch in 0.16s (63979 row/s)
30000 rows in 0.54s (55749 row/s); batch in 0.15s (64624 row/s)
40000 rows in 0.69s (57765 row/s); batch in 0.15s (64793 row/s)
50000 rows in 0.85s (59031 row/s); batch in 0.15s (64707 row/s)
60000 rows in 1.00s (59927 row/s); batch in 0.15s (64847 row/s)
70000 rows in 1.16s (60483 row/s); batch in 0.16s (64051 row/s)
80000 rows in 1.31s (61008 row/s); batch in 0.15s (64953 row/s)
90000 rows in 1.47s (61356 row/s); batch in 0.16s (64285 row/s)
100000 rows in 1.62s (61703 row/s); batch in 0.15s (65012 row/s)
100000 rows in 1.62s (61700 row/s); batches in 0.16 +/- 0.02s [0.15-0.23] (62528_
↪+/- 6173 rows/s [44036-65012])
>>> # time consumed retrieving rows from t1
... c1.time
0.72430899999999492
>>> # time consumed retrieving rows from t2
... c2.time
1.17042099999999766
>>> # actual time consumed by the convert step
... c2.time - c1.time
0.4461120000000274
```

See also `petl.util.timing.progress()`.



```
{'foo': ['a'], 'baz': [True], 'bar': [1]}
>>> fc['b']
{'foo': ['b', 'b'], 'baz': [True, None], 'bar': [2, 3]}
```

`petl.util.materialise.listoflists` (*tbl*)

`petl.util.materialise.listoftuples` (*tbl*)

`petl.util.materialise.tupleoflists` (*tbl*)

`petl.util.materialise.tupleoftuples` (*tbl*)

`petl.util.materialise.cache` (*table*, *n=None*)

Wrap the table with a cache that caches up to *n* rows as they are initially requested via iteration (cache all rows by default).

## Randomly generated tables

`petl.util.random.randomtable` (*numflds=5*, *numrows=100*, *wait=0*, *seed=None*)

Construct a table with random numerical data. Use *numflds* and *numrows* to specify the number of fields and rows respectively. Set *wait* to a float greater than zero to simulate a delay on each row generation (number of seconds per row). E.g.:

```
>>> import petl as etl
>>> table = etl.randomtable(3, 100, seed=42)
>>> table
+-----+-----+-----+
| f0          | f1          | f2          |
+=====+=====+=====+
| 0.6394267984578837 | 0.025010755222666936 | 0.27502931836911926 |
+-----+-----+-----+
| 0.22321073814882275 | 0.7364712141640124 | 0.6766994874229113 |
+-----+-----+-----+
| 0.8921795677048454 | 0.08693883262941615 | 0.4219218196852704 |
+-----+-----+-----+
| 0.029797219438070344 | 0.21863797480360336 | 0.5053552881033624 |
+-----+-----+-----+
| 0.026535969683863625 | 0.1988376506866485 | 0.6498844377795232 |
+-----+-----+-----+
...

```

Note that the data are generated on the fly and are not stored in memory, so this function can be used to simulate very large tables.

`petl.util.random.dummytable` (*numrows=100*, *fields=(('foo', <functools.partial object>), ('bar', <functools.partial object>), ('baz', <built-in method random of Random object at 0x2a8bb60>))*, *wait=0*, *seed=None*)

Construct a table with dummy data. Use *numrows* to specify the number of rows. Set *wait* to a float greater than zero to simulate a delay on each row generation (number of seconds per row). E.g.:

```
>>> import petl as etl
>>> table1 = etl.dummytable(100, seed=42)
>>> table1
+-----+-----+-----+
| foo | bar          | baz          |
+=====+=====+=====+
| 81 | 'apples' | 0.025010755222666936 |
```

```

+-----+-----+-----+
| 35 | 'pears' | 0.22321073814882275 |
+-----+-----+-----+
| 94 | 'apples' | 0.6766994874229113 |
+-----+-----+-----+
| 69 | 'apples' | 0.5904925124490397 |
+-----+-----+-----+
| 4 | 'apples' | 0.09369523986159245 |
+-----+-----+-----+
...

>>> # customise fields
... import random
>>> from functools import partial
>>> fields = [('foo', random.random),
...          ('bar', partial(random.randint, 0, 500)),
...          ('baz', partial(random.choice,
...                           ['chocolate', 'strawberry', 'vanilla']))]
>>> table2 = etl.dummytable(100, fields=fields, seed=42)
>>> table2
+-----+-----+-----+
| foo          | bar | baz          |
+=====+=====+=====+
| 0.6394267984578837 | 12 | 'vanilla' |
+-----+-----+-----+
| 0.27502931836911926 | 114 | 'chocolate' |
+-----+-----+-----+
| 0.7364712141640124 | 346 | 'vanilla' |
+-----+-----+-----+
| 0.8921795677048454 | 44 | 'vanilla' |
+-----+-----+-----+
| 0.4219218196852704 | 15 | 'chocolate' |
+-----+-----+-----+
...

```

Data generation functions can be specified via the *fields* keyword argument.

Note that the data are generated on the fly and are not stored in memory, so this function can be used to simulate very large tables.

## Miscellaneous

`petl.util.misc.typeset` (*table*, *field*)

Return a set containing all Python types found for values in the given field. E.g.:

```

>>> import petl as etl
>>> table = [['foo', 'bar', 'baz'],
...         ['A', 1, '2'],
...         ['B', u'2', '3.4'],
...         [u'B', u'3', '7.8', True],
...         ['D', u'xyz', 9.0],
...         ['E', 42]]
>>> sorted(etl.typeset(table, 'foo'))
['str']
>>> sorted(etl.typeset(table, 'bar'))
['int', 'str']

```



```
>>> sorted(etl.typeset(table, 'baz'))
[NoneType, 'float', 'str']
```

The *field* argument can be a field name or index (starting from zero).

`petl.util.misc.diffheaders(t1, t2)`

Return the difference between the headers of the two tables as a pair of sets. E.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar', 'baz'],
...          ['a', 1, .3]]
>>> table2 = [['baz', 'bar', 'quux'],
...          ['a', 1, .3]]
>>> add, sub = etl.diffheaders(table1, table2)
>>> add
{'quux'}
>>> sub
{'foo'}
```

`petl.util.misc.diffvalues(t1, t2, f)`

Return the difference between the values under the given field in the two tables, e.g.:

```
>>> import petl as etl
>>> table1 = [['foo', 'bar'],
...          ['a', 1],
...          ['b', 3]]
>>> table2 = [['bar', 'foo'],
...          [1, 'a'],
...          [3, 'c']]
>>> add, sub = etl.diffvalues(table1, table2, 'foo')
>>> add
{'c'}
>>> sub
{'b'}
```

`petl.util.misc.strjoin(s)`

Return a function to join sequences using *s* as the separator. Intended for use with `petl.transform.conversions.convert()`.

`petl.util.misc.nthword(n, sep=None)`

Construct a function to return the *n*th word in a string. E.g.:

```
>>> import petl as etl
>>> s = 'foo bar'
>>> f = etl.nthword(0)
>>> f(s)
'foo'
>>> g = etl.nthword(1)
>>> g(s)
'bar'
```

Intended for use with `petl.transform.conversions.convert()`.

`petl.util.misc.coalesce(*fields, **kwargs)`

Return a function which accepts a row and returns the first non-missing value from the specified fields. Intended for use with `petl.transform.basics.addfield()`.



## Version 1.1.0

- Fixed `petl.transform.reshape.melt()` to work with non-string key argument (#209).
- Added example to docstring of `petl.transform.dedup.conflicts()` to illustrate how to analyse the source of conflicts when rows are merged from multiple tables (#256).
- Added functions for working with bcolz ctables, see `petl.io.bcolz` (#310).
- Added `petl.io.base.fromcolumns()` (#316).
- Added `petl.transform.reductions.groupselectlast()`. (#319).
- Added example in docstring for `petl.io.sources.MemorySource` (#323).
- Added function `petl.transform.basics.stack()` as a simpler alternative to `petl.transform.basics.cat()`. Also behaviour of `petl.transform.basics.cat()` has changed for tables where the header row contains duplicate fields. This was part of addressing a bug in `petl.transform.basics.addfield()` for tables where the header contains duplicate fields (#327).
- Change in behaviour of `petl.io.json.fromdicts()` to preserve ordering of keys if ordered dicts are used. Also added `petl.transform.headers.sortheader()` to deal with unordered cases (#332).
- Added keyword `strict` to functions in the `petl.transform.setops` module to enable users to enforce strict set-like behaviour if desired (#333).
- Added `epilogue` argument to `petl.util.vis.display()` to enable further customisation of content of table display in Jupyter notebooks (#337).
- Added `petl.transform.selects.biselect()` as a convenience for obtaining two tables, one with rows matching a condition, the other with rows not matching the condition (#339).
- Changed `petl.io.json.fromdicts()` to avoid making two passes through the data (#341).
- Changed `petl.transform.basics.addfieldusingcontext()` to enable running calculations (#343).
- Fix behaviour of join functions when tables have no non-key fields (#345).

- Fix incorrect default value for ‘errors’ argument when using codec module (#347).
- Added some documentation on how to write extension classes, see *Introduction* (#349).
- Fix issue with unicode field names (#350).

## Version 1.0

Version 1.0 is a new major release of *petl*. The main purpose of version 1.0 is to introduce support for Python 3.4, in addition to the existing support for Python 2.6 and 2.7. Much of the functionality available in *petl* versions 0.x has remained unchanged in version 1.0, and most existing code that uses *petl* should work unchanged with version 1.0 or with minor changes. However there have been a number of API changes, and some functionality has been migrated from the *petlx* package, described below.

If you have any questions about migrating to version 1.0 or find any problems or issues please email [python-etl@googlegroups.com](mailto:python-etl@googlegroups.com).

### Text file encoding

Version 1.0 unifies the API for working with ASCII and non-ASCII encoded text files, including CSV and HTML.

The following functions now accept an ‘encoding’ argument, which defaults to the value of `locale.getpreferredencoding()` (usually ‘utf-8’): *fromcsv*, *tocsv*, *appendcsv*, *teecsv*, *fromtsv*, *totsv*, *appendtsv*, *teetsv*, *fromtext*, *totext*, *appendtext*, *tohtml*, *teehtml*.

The following functions have been removed as they are now redundant: *fromucsv*, *touchsv*, *appenducsv*, *teeucsv*, *fromtsv*, *toutsv*, *appendtsv*, *teeusv*, *fromtext*, *touttext*, *appendtext*, *tohtml*, *teehtml*.

To migrate code, in most cases it should be possible to simply replace ‘fromucsv’ with ‘fromcsv’, etc.

### *petl.fluent* and *petl.interactive*

The functionality previously available through the *petl.fluent* and *petl.interactive* modules is now available through the root *petl* module.

This means two things.

First, it is now possible to use either functional or fluent (i.e., object-oriented) styles of programming with the root *petl* module, as described in introductory section on *Functional and object-oriented programming styles*.

Second, the default representation of table objects uses the *petl.util.vis.look()* function, so you can simply return a table from the prompt to inspect it, as described in the introductory section on *Interactive use*.

The *petl.fluent* and *petl.interactive* modules have been removed as they are now redundant.

To migrate code, it should be possible to simply replace “import *petl.fluent* as *etl*” or “import *petl.interactive* as *etl*” with “import *petl* as *etl*”.

Note that the automatic caching behaviour of the *petl.interactive* module has **not** been retained. If you want to enable caching behaviour for a particular table, make an explicit call to the *petl.util.materialise.cache()* function. See also *Caching*.

## IPython notebook integration

In version 1.0 *petl* table container objects implement `_repr_html_()` so can be returned from a cell in an IPython notebook and will automatically format as an HTML table.

Also, the `petl.util.vis.display()` and `petl.util.vis.displayall()` functions have been migrated across from the *petlx.ipython* package. If you are working within the IPython notebook these functions give greater control over how tables are rendered. For some examples, see:

[http://nbviewer.ipython.org/github/alimanfoo/petl/blob/v1.0/repr\\_html.ipynb](http://nbviewer.ipython.org/github/alimanfoo/petl/blob/v1.0/repr_html.ipynb)

## Database extract/load functions

The `petl.io.db.todb()` function now supports automatic table creation, inferring a schema from data in the table to be loaded. This functionality has been migrated across from the *petlx* package, and requires *SQLAlchemy* to be installed.

The functions `fromsqlite3`, `tosqlite3` and `appendsqlite3` have been removed as they duplicate functionality available from the existing functions `petl.io.db.fromdb()`, `petl.io.db.todb()` and `petl.io.db.appendddb()`. These existing functions have been modified so that if a string is provided as the `dbo` argument it is interpreted as the name of an `sqlite3` file. It should be possible to migrate code by simply replacing ‘fromsqlite3’ with ‘fromdb’, etc.

## Other functions removed or renamed

The following functions have been removed because they are overly complicated and/or hardly ever used. If you use any of these functions and would like to see them re-instated then please email [python-etl@googlegroups.com](mailto:python-etl@googlegroups.com): `rangefacet`, `rangerowreduce`, `rangeaggregate`, `rangecounts`, `multirangeaggregate`, `lenstats`.

The following functions were marked as deprecated in *petl* 0.x and have been removed in version 1.0: `dataslice` (use `data` instead), `fieldconvert` (use `convert` instead), `fieldselect` (use `select` instead), `parsenumber` (use `numparser` instead), `recordmap` (use `rowmap` instead), `recordmapmany` (use `rowmapmany` instead), `recordreduce` (use `rowreduce` instead), `recordselect` (use `rowselect` instead), `valueset` (use `table.values('foo').set()` instead).

The following functions are no longer available in the root *petl* namespace, but are still available from a subpackage if you really need them: `iterdata` (use `data` instead), `iterdicts` (use `dicts` instead), `iternamedtuples` (use `namedtuples` instead), `iterrecords` (use `records` instead), `itervalues` (use `values` instead).

The following functions have been renamed: `isordered` (renamed to `issorted`), `StringSource` (renamed to `MemorySource`).

The function `selectre` has been removed as it duplicates functionality, use `search` instead.

## Sorting and comparison

A major difference between Python 2 and Python 3 involves comparison and sorting of objects of different types. Python 3 is a lot stricter about what you can compare with what, e.g., `None < 1 < 'foo'` works in Python 2.x but raises an exception in Python 3. The strict comparison behaviour of Python 3 is generally a problem for typical usages of *petl*, where data can be highly heterogeneous and a column in a table may have a mixture of values of many different types, including `None` for missing.

To maintain the usability of *petl* in this type of scenario, and to ensure that the behaviour of *petl* is as consistent as possible across different Python versions, the `petl.transform.sorts.sort()` function and anything that depends on it (as well as any other functions making use of rich comparisons) emulate the relaxed comparison behaviour

that is available under Python 2.x. In fact *petl* goes further than this, allowing comparison of a wider range of types than is possible under Python 2.x (e.g., `datetime` with `None`).

As the underlying code to achieve this has been completely reworked, there may be inconsistencies or unexpected behaviour, so it's worth testing carefully the results of any code previously run using *petl* 0.x, especially if you are also migrating from Python 2 to Python 3.

The different comparison behaviour under different Python versions may also give unexpected results when selecting rows of a table. E.g., the following will work under Python 2.x but raise an exception under Python 3.4:

```
>>> import petl as etl
>>> table = [['foo', 'bar'],
...         ['a', 1],
...         ['b', None]]
>>> # raises exception under Python 3
... etl.select(table, 'bar', lambda v: v > 0)
```

To get the more relaxed behaviour under Python 3.4, use the `petl.transform.selects.selectgt` function, or wrap values with `petl.comparison.Comparable`, e.g.:

```
>>> # works under Python 3
... etl.selectgt(table, 'bar', 0)
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+

>>> # or ...
... etl.select(table, 'bar', lambda v: v > etl.Comparable(0))
+-----+-----+
| foo | bar |
+=====+=====+
| 'a' | 1 |
+-----+-----+
```

## New extract/load modules

Several new extract/load modules have been added, migrating functionality previously available from the *petlx* package:

- *Excel .xls files (xlrd/xlwt)*
- *Excel .xlsx files (openpyxl)*
- *Arrays (NumPy)*
- *DataFrames (pandas)*
- *HDF5 files (PyTables)*
- *Text indexes (Whoosh)*

These modules all have dependencies on third party packages, but these have been kept as optional dependencies so are not required for installing *petl*.

## New validate function

A new `petl.transform.validation.validate()` function has been added to provide a convenient interface when validating a table against a set of constraints.

## New intervals module

A new module has been added providing transformation functions based on intervals, migrating functionality previously available from the `petlx` package:

- *Intervals* (`intervaltree`)

This module requires the `intervaltree` module.

## New configuration module

All configuration variables have been brought together into a new `petl.config` module. See the source code for the variables available, they should be self-explanatory.

## `petl.push` moved to `petlx`

The `petl.push` module remains in an experimental state and has been moved to the `petlx` extensions project.

## Argument names and other minor changes

Argument names for a small number of functions have been changed to create consistency across the API.

There are some other minor changes as well. If you are migrating from `petl` version 0.x the best thing is to run your code and inspect any errors. Email [python-etl@googlegroups.com](mailto:python-etl@googlegroups.com) if you have any questions.

## Source code reorganisation

The source code has been substantially reorganised. This should not affect users of the `petl` package however as all functions in the public API are available through the root `petl` namespace.





Contributions to *petl* are welcome in any form, please feel free to email the [python-etl@googlegroups.com](mailto:python-etl@googlegroups.com) mailing list if you have some code or ideas you'd like to discuss.

Please note that the *petl* package is intended as a stable, general purpose library for ETL work. If you would like to extend *petl* with functionality that is domain-specific, or if you have an experimental or tentative feature that is not yet ready for inclusion in the core *petl* package but you would like to distribute it, please contribute to the *petlx* project instead, or distribute your code as a separate package.

If you are thinking of developing or modifying the *petl* code base in any way, here is some information on how to set up your development environment to run tests etc.

## Running the test suite

The main *petl* test suite can be run with *nose*. E.g., assuming you have the source code repository cloned to the current working directory, you can run the test suite with:

```
$ pip install nose
$ nosetests -v
```

Currently *petl* supports Python 2.6, Python 2.7 and Python 3.4, so the tests should pass under all three Python versions.

## Dependencies

To keep installation as simple as possible on different platforms, *petl* has no installation dependencies. Most functionality also depends only on the Python core libraries.

Some *petl* functions depend on third party packages, however these should be kept as optional requirements. Any tests for modules requiring third party packages should be written so that they are skipped if the packages are not available. See the existing tests for examples of how to do this.

## Running database tests

There are some additional tests within the test suite that require database servers to be setup correctly on the local host. To run these additional tests, make sure you have both MySQL and PostgreSQL servers running locally, and have created a user “petl” with password “test” and all permissions granted on a database called “petl”. Install dependencies:

```
$ pip install pymysql psycopg2 sqlalchemy
```

If these dependencies are not installed, or if a local database server is not found, these tests are skipped.

## Running doctests

Doctests in docstrings should (almost) all be runnable, and should pass if run with Python 3.4. Doctests can be run with `nose`. See the `tox.ini` file for example doctest commands.

## Building the documentation

Documentation is built with `sphinx`. To build:

```
$ pip install sphinx
$ cd docs
$ make html
```

Built docs can then be found in the `docs/_build/html/` directory.

## Automatically running all tests

All of the above tests can be run automatically using `tox`. You will need binaries for Python 2.6, Python 2.7 and Python 3.4 available on your system.

To run all tests **without** installing any of the optional dependencies, do:

```
$ tox -e py26,py27,py34,doctests
```

To run the entire test suite, including installation of **all** optional dependencies, do:

```
$ tox
```

The first time you run this it will take some while all the optional dependencies are installed in each environment.

## Contributing code via GitHub

The best way to contribute code is via a GitHub pull request.

Please include unit tests with any code contributed.

If you are able, please run `tox` and ensure that all the above tests pass before making a pull request.

Thanks!

### **continuum.io**

- <http://continuum.io>

In development, a major revision of NumPy to better support a range of data integration and processing use cases.

### **pandas (Python package)**

- <http://pandas.sourceforge.net/>
- <http://pypi.python.org/pypi/pandas>
- <http://github.com/wesm/pandas>

A Python library for analysis of relational/tabular data, built on NumPy, and inspired by R's dataframe concept. Functionality includes support for missing data, inserting and deleting columns, group by/aggregation, merging, joining, reshaping, pivoting.

### **tabular (Python package)**

- <http://pypi.python.org/pypi/tabular>
- <http://packages.python.org/tabular/html/>

A Python package for working with tabular data. The *tabarray* class supports both row-oriented and column-oriented access to data, including selection and filtering of rows/columns, matrix math (tabular extends NumPy), sort, aggregate, join, transpose, comparisons.

Does require a uniform datatype for each column. All data is handled in memory.

### **datarray (Python package)**

- <http://pypi.python.org/pypi/datarray>
- <http://github.com/fperez/datarray>
- <http://fperez.github.com/datarray-doc>

Datarray provides a subclass of Numpy ndarrays that support individual dimensions (axes) being labeled with meaningful descriptions labeled 'ticks' along each axis indexing and slicing by named axis indexing on any axis with the

tick labels instead of only integers reduction operations (like `.sum`, `.mean`, etc) support named axis arguments instead of only integer indices.

**pydataframe (Python package)**

- <http://code.google.com/p/pydataframe/>

An implementation of an almost R like DataFrame object.

**larry (Python package)**

- <http://pypi.python.org/pypi/la>

The main class of the `la` package is a labeled array, `larry`. A `larry` consists of data and labels. The data is stored as a NumPy array and the labels as a list of lists (one list per dimension). `larry` has built-in methods such as `ranking`, `merge`, `shuffle`, `move_sum`, `zscore`, `demean`, `lag` as well as typical Numpy methods like `sum`, `max`, `std`, `sign`, `clip`. NaNs are treated as missing data.

**picalo (Python package)**

- <http://www.picalo.org/>
- <http://pypi.python.org/pypi/picalo/>
- <http://www.picalo.org/download/api/>

A GUI application and Python library primarily aimed at data analysis for auditors & fraud examiners, but has a number of general purpose data mining and transformation capabilities like `filter`, `join`, `transpose`, `crostable/pivot`.

Does not rely on streaming/iterative processing of data, and has a persistence capability based on `zodb` for handling larger datasets.

**csvkit (Python package)**

- <http://pypi.python.org/pypi/picalo/>
- <http://csvkit.rtfid.org/>

A set of command-line utilities for transforming tabular data from CSV (delimited) files. Includes `csvclean`, `csvcut`, `csvjoin`, `csvsort`, `csvstack`, `csvstat`, `csvgrep`, `csvlook`.

**csvutils (Python package)**

- <http://pypi.python.org/pypi/csvutils>

**python-pipeline (Python package)**

- <http://code.google.com/p/python-pipeline/>

**Google Refine**

- <http://code.google.com/p/google-refine/>

A web application for exploring, filtering, cleaning and transforming a table of data. Some excellent functionality for finding and fixing problems in data. Does have the capability to join two tables, but generally it's one table at a time. Some question marks over ability to handle larger datasets.

Has an extension capability, two third party extensions known at the time of writing, including a [stats extension](#).

**Data Wrangler**

- <http://vis.stanford.edu/wrangler/>
- <http://vis.stanford.edu/papers/wrangler>
- <http://pypi.python.org/pypi/DataWrangler>

A web application for exploring, transforming and cleaning tabular data, in a similar vein to Google Refine but with a strong focus on usability, and more capabilities for transforming tables, including folding/unfolding (similar to R reshape's melt/cast) and cross-tabulation.

Currently a client-side only web application, not available for download. There is also a Python library providing data transformation functions as found in the GUI. The research paper has a good discussion of data transformation and quality issues, esp. w.r.t. tool usability.

#### **Pentaho Data Integration (a.k.a. Kettle)**

- <http://kettle.pentaho.com/>
- <http://wiki.pentaho.com/display/EAI/Getting+Started>
- <http://wiki.pentaho.com/display/EAI/Pentaho+Data+Integration+Steps>

#### **SnapLogic**

- <http://www.snaplogic.com>
- <https://www.snaplogic.org/Documentation/3.2/ComponentRef/index.html>

A data integration platform, where ETL components are web resources with a RESTful interface. Standard components for transforms like filter, join and sort.

#### **Talend**

- <http://www.talend.com>

#### **Jaspersoft ETL**

- <http://www.jaspersoft.com/jasperetl>

#### **CloverETL**

- <http://www.cloveretl.com/>

#### **Apatar**

- <http://apatar.com/>

#### **Jitterbit**

- <http://www.jitterbit.com/>

#### **Scriptella**

- <http://scriptella.javaforge.com/>

#### **Kapow Katalyst**

- <http://kapowsoftware.com/products/kapow-katalyst-platform/index.php>
- <http://kapowsoftware.com/products/kapow-katalyst-platform/extraction-browser.php>
- <http://kapowsoftware.com/products/kapow-katalyst-platform/transformation-normalization.php>

#### **Flat File Checker (FlaFi)**

- <http://www.flat-file.net/>

#### **Orange**

- <http://orange.biolab.si/>

#### **North Concepts Data Pipeline**

- <http://northconcepts.com/data-pipeline/>

#### **SAS Clinical Data Integration**

- <http://www.sas.com/industry/pharma/cdi/index.html>

### **R Reshape Package**

- <http://had.co.nz/reshape/>

### **TableFu**

- <http://propublica.github.com/table-fu/>

### **python-tablefu**

- <https://github.com/eyeseast/python-tablefu>

### **pygrametl (Python package)**

- <http://www.pygrametl.org/>
- <http://people.cs.aau.dk/~chr/pygrametl/pygrametl.html>
- <http://dbtr.cs.aau.dk/DBPublications/DBTR-25.pdf>

### **etlpy (Python package)**

- <http://sourceforge.net/projects/etlpy/>
- <http://etlpy.svn.sourceforge.net/viewvc/etlpy/source/samples/>

Looks abandoned since 2009, but there is some code.

### **OpenETL**

- <https://launchpad.net/openetl>
- <http://bazaar.launchpad.net/~openerp-commiter/openetl/OpenETL/files/head:/lib/openetl/component/transform/>

### **Data River**

- <http://www.datariver.it/>

### **Ruffus**

- <http://www.ruffus.org.uk/>

### **PyF**

- <http://pyfproject.org/>

### **PyDTA**

- <http://presbrey.mit.edu/PyDTA>

### **Google Fusion Tables**

- <http://www.google.com/fusiontables/Home/>

### **pivottable (Python package)**

- <http://pypi.python.org/pypi/pivottable/0.8>

### **PrettyTable (Python package)**

- <http://pypi.python.org/pypi/PrettyTable>

### **PyTables (Python package)**

- <http://www.pytables.org/>

### **plyr**

- <http://plyr.had.co.nz/>

### PowerShell

- <http://technet.microsoft.com/en-us/library/ee176874.aspx> - Import-Csv
- <http://technet.microsoft.com/en-us/library/ee176955.aspx> - Select-Object
- <http://technet.microsoft.com/en-us/library/ee176968.aspx> - Sort-Object
- <http://technet.microsoft.com/en-us/library/ee176864.aspx> - Group-Object

### SwiftRiver

- <http://ushahidi.com/products/swiftriver-platform>

### Data Science Toolkit

- <http://www.datasciencetoolkit.org/about>

### IncPy

- <http://www.stanford.edu/~pgbovine/incpy.html>

Doesn't have any ETL functionality, but possibly (enormously) relevant to exploratory development of a transformation pipeline, because you could avoid having to rerun the whole pipeline every time you add a new step.

### Articles, Blogs, Other

- <http://metadeveloper.blogspot.com/2008/02/iron-python-dsl-for-etl.html>
- [http://www.cs.uoi.gr/~pvassil/publications/2009\\_IJDWM/IJDWM\\_2009.pdf](http://www.cs.uoi.gr/~pvassil/publications/2009_IJDWM/IJDWM_2009.pdf)
- <http://web.tagus.ist.utl.pt/~helena.galhardas/ajax.html>
- <http://stackoverflow.com/questions/1321396/what-are-the-required-functionnalities-of-etl-frameworks>
- <http://stackoverflow.com/questions/3762199/etl-using-python>
- <http://www.jonathanlevin.co.uk/2008/03/open-source-etl-tools-vs-commerical-etl.html>
- <http://www.quora.com/ETL/Why-should-I-use-an-existing-ETL-vs-writing-my-own-in-Python-for-my-data-warehouse-needs>
- <http://synful.us/archives/41/the-poor-mans-etl-python>
- [http://www.gossamer-threads.com/lists/python/python/418041?do=post\\_view\\_threaded#418041](http://www.gossamer-threads.com/lists/python/python/418041?do=post_view_threaded#418041)
- <http://code.activestate.com/lists/python-list/592134/>
- <http://fuzzytolerance.info/code/open-source-etl-tools/>
- <http://www.protocolostomy.com/2009/12/28/codekata-4-data-munging/>
- <http://www.hanselman.com/blog/ParsingCSVsAndPoorMansWebLogAnalysisWithPowerShell.aspx> - nice example of a data transformation problem, done in PowerShell
- <http://www.datascience.co.nz/blog/2011/04/01/the-science-of-data-munging/>
- <http://wesmckinney.com/blog/?p=8> - on grouping with pandas
- <http://stackoverflow.com/questions/4341756/data-recognition-parsing-filtering-and-transformation-gui>

### On memoization...

- <http://wiki.python.org/moin/PythonDecoratorLibrary#Memoize>
- <http://code.activestate.com/recipes/577219-minimalistic-memoization/>
- <http://ubuntuforums.org/showthread.php?t=850487>





## CHAPTER 8

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



**p**

- petl, 1
- petl.io, 9
  - petl.io.bcolz, 29
  - petl.io.csv, 12
  - petl.io.db, 21
  - petl.io.html, 18
  - petl.io.json, 19
  - petl.io.numpy, 25
  - petl.io.pandas, 26
  - petl.io.pickle, 14
  - petl.io.pytables, 27
  - petl.io.sources, 34
  - petl.io.text, 15
  - petl.io.whoosh, 31
  - petl.io.xls, 24
  - petl.io.xlsx, 24
  - petl.io.xml, 16
- petl.transform, 35
  - petl.transform.basics, 37
  - petl.transform.conversions, 48
  - petl.transform.dedup, 75
  - petl.transform.fills, 88
  - petl.transform.headers, 46
  - petl.transform.intervals, 92
  - petl.transform.joins, 64
  - petl.transform.maps, 59
  - petl.transform.reductions, 78
  - petl.transform.regex, 56
  - petl.transform.reshape, 83
  - petl.transform.selects, 52
  - petl.transform.setops, 71
  - petl.transform.sorts, 62
  - petl.transform.unpacks, 58
  - petl.transform.validation, 91
- petl.util, 98



**A**

addcolumn() (in module petl.transform.basics), 44  
 addfield() (in module petl.transform.basics), 44  
 addfieldusingcontext() (in module petl.transform.basics), 45  
 addrownnumbers() (in module petl.transform.basics), 45  
 aggregate() (in module petl.transform.reductions), 78  
 annex() (in module petl.transform.basics), 46  
 antijoin() (in module petl.transform.joins), 68  
 appendbcolz() (in module petl.io.bcolz), 31  
 appendcsv() (in module petl.io.csv), 13  
 appenddb() (in module petl.io.db), 24  
 appendhdf5() (in module petl.io.pytables), 29  
 appendpickle() (in module petl.io.pickle), 15  
 appendtext() (in module petl.io.text), 16  
 appendtextindex() (in module petl.io.whoosh), 34  
 appendtsv() (in module petl.io.csv), 14

**B**

biselect() (in module petl.transform.selects), 55  
 boolparser() (in module petl.util.parsers), 108  
 BZ2Source (class in petl.io.sources), 34

**C**

cache() (in module petl.util.materialise), 115  
 capture() (in module petl.transform.regex), 57  
 cat() (in module petl.transform.basics), 41  
 clock() (in module petl.util.timing), 113  
 coalesce() (in module petl.util.misc), 117  
 collapsedintervals() (in module petl.transform.intervals), 97  
 columns() (in module petl.util.materialise), 114  
 complement() (in module petl.transform.setops), 71  
 conflicts() (in module petl.transform.dedup), 76  
 convert() (in module petl.transform.conversions), 48  
 convertall() (in module petl.transform.conversions), 51  
 convertnumbers() (in module petl.transform.conversions), 51  
 crossjoin() (in module petl.transform.joins), 68

cut() (in module petl.transform.basics), 39  
 cutout() (in module petl.transform.basics), 40

**D**

data() (in module petl.util.base), 99  
 dateparser() (in module petl.util.parsers), 107  
 datetimestr() (in module petl.util.parsers), 107  
 dictlookup() (in module petl.util.lookups), 105  
 dictlookupone() (in module petl.util.lookups), 106  
 dicts() (in module petl.util.base), 100  
 diff() (in module petl.transform.setops), 72  
 diffheaders() (in module petl.util.misc), 117  
 diffvalues() (in module petl.util.misc), 117  
 display() (in module petl.util.vis), 103  
 displayall() (in module petl.util.vis), 103  
 distinct() (in module petl.transform.dedup), 77  
 dummytable() (in module petl.util.random), 115  
 duplicates() (in module petl.transform.dedup), 75

**E**

empty() (in module petl.util.base), 101  
 expr() (in module petl.util.base), 101  
 extendheader() (in module petl.transform.headers), 47

**F**

facet() (in module petl.transform.selects), 54  
 facetcolumns() (in module petl.util.materialise), 114  
 facetintervallookup() (in module petl.transform.intervals), 97  
 facetintervallookupone() (in module petl.transform.intervals), 97  
 facetintervalrecordlookup() (in module petl.transform.intervals), 97  
 facetintervalrecordlookupone() (in module petl.transform.intervals), 97  
 fieldmap() (in module petl.transform.maps), 59  
 fieldnames() (in module petl.util.base), 99  
 FileSource (class in petl.io.sources), 34  
 filldown() (in module petl.transform.fills), 88

fillleft() (in module petl.transform.fills), 90  
 fillright() (in module petl.transform.fills), 89  
 flatten() (in module petl.transform.reshape), 87  
 fold() (in module petl.transform.reductions), 81  
 format() (in module petl.transform.conversions), 52  
 formatall() (in module petl.transform.conversions), 52  
 fromarray() (in module petl.io.numpy), 25  
 frombcolz() (in module petl.io.bcolz), 30  
 fromcolumns() (in module petl.io.base), 12  
 fromcsv() (in module petl.io.csv), 12  
 fromdataframe() (in module petl.io.pandas), 26  
 fromdb() (in module petl.io.db), 21  
 fromdicts() (in module petl.io.json), 20  
 fromhdf5() (in module petl.io.pytables), 27  
 fromhdf5sorted() (in module petl.io.pytables), 28  
 fromjson() (in module petl.io.json), 19  
 frompickle() (in module petl.io.pickle), 14  
 fromtext() (in module petl.io.text), 15  
 fromtextindex() (in module petl.io.whoosh), 31  
 fromtsv() (in module petl.io.csv), 14  
 fromxls() (in module petl.io.xls), 24  
 fromxlsx() (in module petl.io.xlsx), 24  
 fromxml() (in module petl.io.xml), 17

## G

groupcountdistinctvalues() (in module petl.transform.reductions), 82  
 groupselectfirst() (in module petl.transform.reductions), 82  
 groupselectlast() (in module petl.transform.reductions), 82  
 groupselectmax() (in module petl.transform.reductions), 83  
 groupselectmin() (in module petl.transform.reductions), 83  
 GzipSource (class in petl.io.sources), 34

## H

hashantijoin() (in module petl.transform.hashjoins), 71  
 hashcomplement() (in module petl.transform.setops), 75  
 hashintersection() (in module petl.transform.setops), 75  
 hashjoin() (in module petl.transform.hashjoins), 70  
 hashleftjoin() (in module petl.transform.hashjoins), 70  
 hashlookupjoin() (in module petl.transform.hashjoins), 70  
 hashrightjoin() (in module petl.transform.hashjoins), 70  
 head() (in module petl.transform.basics), 37  
 header() (in module petl.util.base), 99

## I

interpolate() (in module petl.transform.conversions), 52  
 interpolateall() (in module petl.transform.conversions), 52  
 intersection() (in module petl.transform.setops), 74  
 intervalantijoin() (in module petl.transform.intervals), 94

intervaljoin() (in module petl.transform.intervals), 92  
 intervaljoinvalues() (in module petl.transform.intervals), 94  
 intervalleftjoin() (in module petl.transform.intervals), 94  
 intervallookup() (in module petl.transform.intervals), 95  
 intervallookupone() (in module petl.transform.intervals), 96  
 intervalrecordlookup() (in module petl.transform.intervals), 96  
 intervalrecordlookupone() (in module petl.transform.intervals), 96  
 intervalsubtract() (in module petl.transform.intervals), 97  
 issorted() (in module petl.transform.sorts), 64  
 isunique() (in module petl.transform.dedup), 78

## J

join() (in module petl.transform.joins), 64

## L

leftjoin() (in module petl.transform.joins), 66  
 limits() (in module petl.util.statistics), 114  
 listofflists() (in module petl.util.materialise), 115  
 listoftuples() (in module petl.util.materialise), 115  
 look() (in module petl.util.vis), 102  
 lookall() (in module petl.util.vis), 103  
 lookup() (in module petl.util.lookups), 103  
 lookupjoin() (in module petl.transform.joins), 66  
 lookupone() (in module petl.util.lookups), 104

## M

melt() (in module petl.transform.reshape), 83  
 MemorySource (class in petl.io.sources), 35  
 merge() (in module petl.transform.reductions), 81  
 mergeduplicates() (in module petl.transform.reductions), 80  
 mergesort() (in module petl.transform.sorts), 63  
 movefield() (in module petl.transform.basics), 41

## N

namedtuples() (in module petl.util.base), 100  
 nrows() (in module petl.util.counting), 108  
 nthword() (in module petl.util.misc), 117  
 numparser() (in module petl.util.parsers), 108

## O

outerjoin() (in module petl.transform.joins), 67

## P

parsecounter() (in module petl.util.counting), 112  
 parsecounts() (in module petl.util.counting), 112  
 petl (module), 1  
 petl.io (module), 9  
 petl.io.bcolz (module), 29

petl.io.csv (module), 12  
 petl.io.db (module), 21  
 petl.io.html (module), 18  
 petl.io.json (module), 19  
 petl.io.numpy (module), 25  
 petl.io.pandas (module), 26  
 petl.io.pickle (module), 14  
 petl.io.pytables (module), 27  
 petl.io.sources (module), 34  
 petl.io.text (module), 15  
 petl.io.whoosh (module), 31  
 petl.io.xls (module), 24  
 petl.io.xlsx (module), 24  
 petl.io.xml (module), 16  
 petl.transform (module), 35  
 petl.transform.basics (module), 37  
 petl.transform.conversions (module), 48  
 petl.transform.dedup (module), 75  
 petl.transform.fills (module), 88  
 petl.transform.headers (module), 46  
 petl.transform.intervals (module), 92  
 petl.transform.joins (module), 64  
 petl.transform.maps (module), 59  
 petl.transform.reductions (module), 78  
 petl.transform.regex (module), 56  
 petl.transform.reshape (module), 83  
 petl.transform.selects (module), 52  
 petl.transform.setops (module), 71  
 petl.transform.sorts (module), 62  
 petl.transform.unpacks (module), 58  
 petl.transform.validation (module), 91  
 petl.util (module), 98  
 pivot() (in module petl.transform.reshape), 86  
 PopenSource (class in petl.io.sources), 35  
 prefixheader() (in module petl.transform.headers), 48  
 progress() (in module petl.util.timing), 113  
 pushheader() (in module petl.transform.headers), 48

## R

randomtable() (in module petl.util.random), 115  
 recast() (in module petl.transform.reshape), 84  
 recordcomplement() (in module petl.transform.setops),  
     73  
 recorddiff() (in module petl.transform.setops), 74  
 recordlookup() (in module petl.util.lookups), 106  
 recordlookupone() (in module petl.util.lookups), 106  
 records() (in module petl.util.base), 101  
 rename() (in module petl.transform.headers), 46  
 replace() (in module petl.transform.conversions), 51  
 replaceall() (in module petl.transform.conversions), 52  
 rightjoin() (in module petl.transform.joins), 67  
 rowgroupby() (in module petl.util.base), 101  
 rowgroupmap() (in module petl.transform.maps), 61  
 rowlengths() (in module petl.util.counting), 110

rowlenselect() (in module petl.transform.selects), 54  
 rowmap() (in module petl.transform.maps), 60  
 rowmapmany() (in module petl.transform.maps), 61  
 rowreduce() (in module petl.transform.reductions), 80  
 rowslice() (in module petl.transform.basics), 38

## S

search() (in module petl.transform.regex), 56  
 searchcomplement() (in module petl.transform.regex), 56  
 searchtextindex() (in module petl.io.whoosh), 32  
 searchtextindexpage() (in module petl.io.whoosh), 33  
 see() (in module petl.util.vis), 103  
 select() (in module petl.transform.selects), 52  
 selectcontains() (in module petl.transform.selects), 53  
 selecteq() (in module petl.transform.selects), 53  
 selectfalse() (in module petl.transform.selects), 54  
 selectge() (in module petl.transform.selects), 53  
 selectgt() (in module petl.transform.selects), 53  
 selectin() (in module petl.transform.selects), 54  
 selectis() (in module petl.transform.selects), 54  
 selectisinstance() (in module petl.transform.selects), 54  
 selectisnot() (in module petl.transform.selects), 54  
 selectle() (in module petl.transform.selects), 53  
 selectlt() (in module petl.transform.selects), 53  
 selectne() (in module petl.transform.selects), 53  
 selectnone() (in module petl.transform.selects), 54  
 selectnotin() (in module petl.transform.selects), 54  
 selectnotnone() (in module petl.transform.selects), 54  
 selectop() (in module petl.transform.selects), 53  
 selectrangeclosed() (in module petl.transform.selects), 53  
 selectrangeopen() (in module petl.transform.selects), 53  
 selectrangeopenleft() (in module petl.transform.selects),  
     53  
 selectrangeopenright() (in module petl.transform.selects),  
     53  
 selecttrue() (in module petl.transform.selects), 54  
 selectusingcontext() (in module petl.transform.selects),  
     54  
 setheader() (in module petl.transform.headers), 47  
 skip() (in module petl.transform.headers), 48  
 skipcomments() (in module petl.transform.basics), 43  
 sort() (in module petl.transform.sorts), 62  
 sorthead() (in module petl.transform.headers), 48  
 split() (in module petl.transform.regex), 57  
 stack() (in module petl.transform.basics), 42  
 stats() (in module petl.util.statistics), 114  
 StdinSource (class in petl.io.sources), 34  
 StdoutSource (class in petl.io.sources), 34  
 stringpatterncounter() (in module petl.util.counting), 110  
 stringpatterns() (in module petl.util.counting), 110  
 strjoin() (in module petl.util.misc), 117  
 sub() (in module petl.transform.regex), 57  
 suffixheader() (in module petl.transform.headers), 48

## T

tail() (in module petl.transform.basics), 37  
teecsv() (in module petl.io.csv), 13  
teehtml() (in module petl.io.html), 19  
teepickle() (in module petl.io.pickle), 15  
teetext() (in module petl.io.text), 16  
teetsv() (in module petl.io.csv), 14  
timeparser() (in module petl.util.parsers), 107  
toarray() (in module petl.io.numpy), 25  
tobcolz() (in module petl.io.bcolz), 30  
tocsv() (in module petl.io.csv), 13  
todataframe() (in module petl.io.pandas), 26  
todb() (in module petl.io.db), 22  
tohdf5() (in module petl.io.pytables), 29  
tohtml() (in module petl.io.html), 19  
tojson() (in module petl.io.json), 20  
tojsonarrays() (in module petl.io.json), 21  
topickle() (in module petl.io.pickle), 14  
torecarray() (in module petl.io.numpy), 26  
totext() (in module petl.io.text), 16  
totextindex() (in module petl.io.whoosh), 33  
totsv() (in module petl.io.csv), 14  
toxls() (in module petl.io.xls), 24  
toxlsx() (in module petl.io.xlsx), 25  
transpose() (in module petl.transform.reshape), 86  
tupleoflists() (in module petl.util.materialise), 115  
tupleoftuples() (in module petl.util.materialise), 115  
typecounter() (in module petl.util.counting), 111  
typecounts() (in module petl.util.counting), 111  
typeset() (in module petl.util.misc), 116

## U

unflatten() (in module petl.transform.reshape), 87  
unique() (in module petl.transform.dedup), 76  
unjoin() (in module petl.transform.joins), 69  
unpack() (in module petl.transform.unpacks), 58  
unpackdict() (in module petl.transform.unpacks), 59  
update() (in module petl.transform.conversions), 52  
URLSource (class in petl.io.sources), 34

## V

validate() (in module petl.transform.validation), 91  
valuecount() (in module petl.util.counting), 108  
valuecounter() (in module petl.util.counting), 109  
valuecounts() (in module petl.util.counting), 109  
values() (in module petl.util.base), 99  
valuestoarray() (in module petl.io.numpy), 26

## Z

ZipSource (class in petl.io.sources), 34