petl Documentation

Release 0.22

Alistair Miles

February 20, 2014

`petl` is a 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

For examples of `petl` in use, see the case studies below:

- Comparing Tables

CHAPTER 1

Overview

The tables below gives an overview of the main functions in the `petl` module.

See also the alphabetic *genindex* of all functions in the package.

CHAPTER 2

Introduction

2.1 Installation

This module is available from the Python Package Index. On Linux distributions you should be able to do `easy_install petl` or `pip install petl`. On other platforms you can download manually, extract and run `python setup.py install`.

2.2 Dependencies and extensions

This package has been written with no dependencies other than the Python core modules, for ease of installation and maintenance. However, there are many third party packages which could usefuly be used with `petl`, e.g., providing access to data from Excel or other file types. Some extensions with these additional dependencies are provided by the petlx package, a companion package to `petl`.

2.3 Conventions - row containers and row iterators

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

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

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

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

4. each item returned by the iterator is either a list or a tuple
5. the first item returned by the iterator is a *header row* comprising a list or tuple of *fields*
6. each subsequent item returned by the iterator is a *data row* comprising a list or tuple of *data values*
7. a *field* is typically a string (*str* or *unicode*) 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 that supports rich comparison operators

So, for example, the list of lists shown below is a row container:

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

Note that, under this convention, an object returned by the `csv.reader()` function from the standard Python `csv` module is a row iterator and *not* a row container, because it can only be iterated over once, e.g.:

```
>>> from StringIO import StringIO
>>> import csv
>>> csvdata = """foo,bar
... a,1
... b,2
... """
>>> rowiterator = csv.reader(StringIO(csvdata))
>>> for row in rowiterator:
...     print row
...
['foo', 'bar']
['a', '1']
['b', '2']
>>> for row in rowiterator:
...     print row
...
>>> # can only iterate once
```

However, it is straightforward to define functions that support the above convention for row containers and provide access to data from CSV or other types of file or data source, see e.g. the `fromcsv()` function in this package.

The main reason for requiring that row containers support independent row 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 return a header row and/or data rows of varying lengths.

Note also that many features of `petl` depend on sorting which will only work if the data values support rich comparison operators.

2.4 Transformation pipelines

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

E.g., given the following data in a file at 'example1.csv' in the current working directory:

```
foo,bar,baz
a,1,3.4
b,2,7.4
c,6,2.2
d,9,8.1
```

...the following code does not actually read the file, nor does it load any of its contents into memory:

```
>>> from petl import *
>>> table1 = fromcsv('example1.csv')
```

Rather, *table1* is a row container object, which can be iterated over.

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

```
>>> table2 = convert(table1, 'foo', 'upper')
>>> table3 = convert(table2, 'bar', int)
>>> table4 = convert(table3, 'baz', float)
>>> table5 = addfield(table4, 'quux', expr('{bar} * {baz}'))
```

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

So in effect, a 5 step transformation 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 `look()`, or any of the functions which write data to a file or database (e.g., `tocsv()`, `totext()`, `tosqlite3()`, `todb()`), will pull data through the pipeline and cause all of the transformation steps to be executed on the requested rows, e.g.:

```
>>> look(table5)
+-------+-------+-------+--------------------+
| 'foo' | 'bar' | 'baz' | 'quux'             |
+=======+=======+=======+====================+
| 'A'   | 1     | 3.4   | 3.4                |
+-------+-------+-------+--------------------+
| 'B'   | 2     | 7.4   | 14.8               |
+-------+-------+-------+--------------------+
| 'C'   | 6     | 2.2   | 13.200000000000001 |
+-------+-------+-------+--------------------+
| 'D'   | 9     | 8.1   | 72.89999999999999  |
+-------+-------+-------+--------------------+
```

...although note that `look()` will by default only request the first 10 rows, and so at most only 10 rows will be processed. Calling `look()` to inspect the first few rows of a table is often an efficient way to examine the output of a transformation pipeline, without having to execute the transformation over all of the input data.

2.5 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 `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 `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 obviously undesirable to throw away the sorted data and start again from scratch each time. It is better to cache the sorted data, if possible, so it can be re-used.

The `sort()` function and all functions which use `sort()` 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 `cache()` function, which can be used to cache in memory up to a configurable number of rows from a table.

Changed in version 0.16.

Use of the cachetag() method is now deprecated.

CHAPTER 3

Extract - reading tables from files, databases and other sources

The following functions extract a table from a file-like source or database. For everything except `fromdb()` the `source` argument provides information about where to read 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.

`petl.`**`fromcsv`**(*source=None*, *dialect=<class csv.excel at 0x2ad5050>*, ***kwargs*)

Wrapper for the standard `csv.reader()` function. Returns a table providing access to the data in the given delimited file. E.g.:

```
>>> import csv
>>> # set up a CSV file to demonstrate with
... with open('test.csv', 'wb') as f:
...     writer = csv.writer(f)
...     writer.writerow(['foo', 'bar'])
...     writer.writerow(['a', 1])
...     writer.writerow(['b', 2])
...     writer.writerow(['c', 2])
...
>>> # now demonstrate the use of petl.fromcsv
... from petl import fromcsv, look
>>> testcsv = fromcsv('test.csv')
>>> look(testcsv)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '2'   |
+-------+-------+
| 'c'   | '2'   |
+-------+-------+
```

The *filename* 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 `convert()`.

Supports transparent reading from URLs, `.gz` and `.bz2` files.

`petl.`**`fromtsv`**(*source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `fromcsv()` but with different default dialect (tab delimited).

Supports transparent reading from URLs, `.gz` and `.bz2` files.

New in version 0.9.

`petl.`**`fromucsv`**(*source=None*, *dialect=<class csv.excel at 0x2ad5050>*, *encoding='utf-8'*, ***kwargs*)
Returns a table containing unicode data extracted from a delimited file via the given encoding. Like `fromcsv()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

`petl.`**`fromutsv`**(*source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `fromucsv()` but with different default dialect (tab delimited).

New in version 0.19.

`petl.`**`frompickle`**(*source=None*)
Returns a table providing access to the 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 pickle
>>> # set up a file to demonstrate with
... with open('test.dat', '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 petl.frompickle
... from petl import frompickle, look
>>> testdat = frompickle('test.dat')
>>> look(testdat)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2.5   |
+-------+-------+
```

Supports transparent reading from URLs, `.gz` and `.bz2` files.

`petl.`**`fromsqlite3`**(*source*, *query*, **args*, ***kwargs*)
Provides access to data from an `sqlite3` database file via a given query. E.g.:

```
>>> import sqlite3
>>> from petl import look, fromsqlite3
>>> # set up a database to demonstrate with
```

```
>>> data = [['a', 1],
...         ['b', 2],
...         ['c', 2.0]]
>>> connection = sqlite3.connect('test.db')
>>> c = connection.cursor()
>>> c.execute('create table foobar (foo, bar)')
<sqlite3.Cursor object at 0x2240b90>
>>> for row in data:
...     c.execute('insert into foobar values (?, ?)', row)
...
<sqlite3.Cursor object at 0x2240b90>
<sqlite3.Cursor object at 0x2240b90>
<sqlite3.Cursor object at 0x2240b90>
>>> connection.commit()
>>> c.close()
>>>
>>> # now demonstrate the petl.fromsqlite3 function
... foobar = fromsqlite3('test.db', 'select * from foobar')
>>> look(foobar)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| u'a'  | 1     |
+-------+-------+
| u'b'  | 2     |
+-------+-------+
| u'c'  | 2.0   |
+-------+-------+
```

Changed in version 0.10.2.

Either a database file name or a connection object can be given as the first argument.

petl.**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 sqlite3
>>> from petl import look, fromdb
>>> connection = sqlite3.connect('test.db')
>>> table = fromdb(connection, 'select * from foobar')
>>> look(table)
```

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

```
>>> import psycopg2
>>> from petl import look, fromdb
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> table = fromdb(connection, 'select * from test')
>>> look(table)
```

E.g., using *MySQLdb* (assuming you've installed it first):

```
>>> import MySQLdb
>>> from petl import look, fromdb
>>> connection = MySQLdb.connect(passwd="moonpie", db="thangs")
>>> table = fromdb(connection, 'select * from test')
>>> look(table)
```

Changed in version 0.10.2.

The first argument may also be a function that creates a cursor. E.g.:

```
>>> import psycopg2
>>> from petl import look, fromdb
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> mkcursor = lambda: connection.cursor(cursor_factory=psycopg2.extras.DictCursor)
>>> table = fromdb(mkcursor, 'select * from test')
>>> look(table)
```

N.B., each call to the function should return a new cursor.

Changed in version 0.18.

Added support for server-side cursors.

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 psycopg2
>>> from petl import look, fromdb
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> table = fromdb(lambda: connection.cursor(name='arbitrary'), 'select * from test')
>>> look(table)
```

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

```
>>> import MySQLdb
>>> from petl import look, fromdb
>>> connection = MySQLdb.connect(passwd="moonpie", db="thangs")
>>> table = fromdb(lambda: connection.cursor(MySQLdb.cursors.SSCursor), 'select * from test')
>>> look(table)
```

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.**fromtext**(*source=None, header=['lines'], strip=None*)

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

```
>>> # example data
... with open('test.txt', 'w') as f:
...     f.write('a\t1\n')
...     f.write('b\t2\n')
...     f.write('c\t3\n')
...
>>> from petl import fromtext, look
>>> table1 = fromtext('test.txt')
>>> look(table1)
+--------------+
| 'lines'      |
+==============+
| 'a\t1'       |
+--------------+
| 'b\t2'       |
+--------------+
| 'c\t3'       |
+--------------+
```

The `fromtext()` function provides a starting point for custom handling of text files. E.g., using `capture()`:

```
>>> from petl import capture
>>> table2 = capture(table1, 'lines', '(.*)\\t(.*)$', ['foo', 'bar'])
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '2'   |
+-------+-------+
| 'c'   | '3'   |
+-------+-------+
```

Supports transparent reading from URLs, `.gz` and `.bz2` files.

Changed in version 0.4.

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.

`petl.`**`fromutext`**(*source=None, header=[u'lines'], encoding='utf-8', strip=None*)
Construct a table from lines in the given text file via the given encoding. Like `fromtext()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

`petl.`**`fromxml`**(*source*, **args*, ***kwargs*)
Access data in an XML file. E.g.:

```
>>> from petl import fromxml, look
>>> data = """<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(data)
...     f.close()
...
>>> table1 = fromxml('example1.xml', 'tr', 'td')
>>> look(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, e.g.:

```
>>> data = """<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(data)
...     f.close()
...
>>> table2 = fromxml('example2.xml', 'tr', 'td', 'v')
>>> look(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, e.g.:

```
>>> data = """<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(data)
...     f.close()
...
>>> table3 = fromxml('example3.xml', 'row', {'foo': 'foo', 'bar': ('baz/bar', 'v')})
>>> look(table3)
+-------+------------+
| 'foo' | 'bar'      |
+=======+============+
| 'a'   | ('1', '3') |
+-------+------------+
| 'b'   | '2'        |
+-------+------------+
| 'c'   | '2'        |
```

```
+-------+------------+
```

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

Supports transparent reading from URLs, `.gz` and `.bz2` files.

New in version 0.4.

Changed in version 0.6: If multiple elements match a given field, all values are reported as a tuple.

petl.**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.:

```
>>> from petl import fromjson, look
>>> data = '[{"foo": "a", "bar": 1}, {"foo": "b", "bar": 2}, {"foo": "c", "bar": 2}]'
>>> with open('example1.json', 'w') as f:
...     f.write(data)
...
>>> table1 = fromjson('example1.json')
>>> look(table1)
+--------+--------+
| u'foo' | u'bar' |
+========+========+
| u'a'   | 1      |
+--------+--------+
| u'b'   | 2      |
+--------+--------+
| u'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 `fromdicts()`.

Supports transparent reading from URLs, `.gz` and `.bz2` files.

New in version 0.5.

petl.**fromdicts**(*dicts*, *header=None*)

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

```
>>> from petl import fromdicts, look
>>> dicts = [{"foo": "a", "bar": 1}, {"foo": "b", "bar": 2}, {"foo": "c", "bar": 2}]
>>> table = fromdicts(dicts)
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+
```

See also `fromjson()`.

New in version 0.5.

CHAPTER 4

Transform - transforming tables

petl.**rename**(*table*, **args*)

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

```
>>> from petl import look, rename
>>> look(table1)
+-------+-------+
| 'sex' | 'age' |
+=======+=======+
| 'M'   | 12    |
+-------+-------+
| 'F'   | 34    |
+-------+-------+
| '-'   | 56    |
+-------+-------+

>>> # rename a single field
... table2 = rename(table1, 'sex', 'gender')
>>> look(table2)
+----------+-------+
| 'gender' | 'age' |
+==========+=======+
| 'M'      | 12    |
+----------+-------+
| 'F'      | 34    |
+----------+-------+
| '-'      | 56    |
+----------+-------+

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

>>> # the returned table object can also be used to modify the field mapping using the suffix nc
... table4 = rename(table1)
```

```
>>> table4['sex'] = 'gender'
>>> table4['age'] = 'age_years'
>>> look(table4)
+----------+-------------+
| 'gender' | 'age_years' |
+==========+=============+
| 'M'      | 12          |
+----------+-------------+
| 'F'      | 34          |
+----------+-------------+
| '-'      | 56          |
+----------+-------------+
```

Changed in version 0.4.

Function signature changed to support the simple 2 argument form when renaming a single field.

petl.**setheader**(*table*, *fields*)

Override fields in the given table. E.g.:

```
>>> from petl import setheader, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+

>>> table2 = setheader(table1, ['foofoo', 'barbar'])
>>> look(table2)
+----------+----------+
| 'foofoo' | 'barbar' |
+==========+==========+
| 'a'      | 1        |
+----------+----------+
| 'b'      | 2        |
+----------+----------+
```

See also `extendheader()`, `pushheader()`.

petl.**extendheader**(*table*, *fields*)

Extend fields in the given table. E.g.:

```
>>> from petl import extendheader, look
>>> look(table1)
+-------+---+-------+
| 'foo' |   |       |
+=======+===+=======+
| 'a'   | 1 | True  |
+-------+---+-------+
| 'b'   | 2 | False |
+-------+---+-------+

>>> table2 = extendheader(table1, ['bar', 'baz'])
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
```

```
| 'a'   | 1     | True  |
+-------+-------+-------+
| 'b'   | 2     | False |
+-------+-------+-------+
```

See also `setheader()`, `pushheader()`.

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

```
>>> from petl import pushheader, look
>>> look(table1)
+-----+---+
| 'a' | 1 |
+=====+===+
| 'b' | 2 |
+-----+---+

>>> table2 = pushheader(table1, ['foo', 'bar'])
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
```

Useful, e.g., where data are from a CSV file that has not included a header row.

`petl.`**`skip`**(*table*, *n*)
Skip *n* rows (including the header row).

E.g.:

```
>>> from petl import skip, look
>>> look(table1)
+--------+-------+-------+
| '#aaa' | 'bbb' | 'ccc' |
+========+=======+=======+
| '#mmm' |       |       |
+--------+-------+-------+
| 'foo'  | 'bar' |       |
+--------+-------+-------+
| 'a'    | 1     |       |
+--------+-------+-------+
| 'b'    | 2     |       |
+--------+-------+-------+

>>> table2 = skip(table1, 2)
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
```

See also `skipcomments()`.

petl.**skipcomments**(*table*, *prefix*)

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

```
>>> from petl import skipcomments, look
>>> look(table1)
+---------+-------+-------+
| '##aaa' | 'bbb' | 'ccc' |
+=========+=======+=======+
| '##mmm' |       |       |
+---------+-------+-------+
| '#foo'  | 'bar' |       |
+---------+-------+-------+
| '##nnn' | 1     |       |
+---------+-------+-------+
| 'a'     | 1     |       |
+---------+-------+-------+
| 'b'     | 2     |       |
+---------+-------+-------+

>>> table2 = skipcomments(table1, '##')
>>> look(table2)
+--------+-------+
| '#foo' | 'bar' |
+========+=======+
| 'a'    | 1     |
+--------+-------+
| 'b'    | 2     |
+--------+-------+
```

New in version 0.4.

petl.**rowslice**(*table*, **sliceargs*)

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

```
>>> from petl import rowslice, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 5     |
+-------+-------+
| 'd'   | 7     |
+-------+-------+
| 'f'   | 42    |
+-------+-------+

>>> table2 = rowslice(table1, 2)
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
```

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

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

Changed in version 0.3.

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

petl.**head**(*table*, *n=10*)

Choose the first n data rows. E.g.:

```
>>> from petl import head, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 5     |
+-------+-------+
| 'd'   | 7     |
+-------+-------+
| 'f'   | 42    |
+-------+-------+
| 'f'   | 3     |
+-------+-------+
| 'h'   | 90    |
+-------+-------+

>>> table2 = head(table1, 4)
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
```

```
+-------+-------+
| 'c'   | 5     |
+-------+-------+
| 'd'   | 7     |
+-------+-------+
```

Syntactic sugar, equivalent to `rowslice(table, n)`.

`petl.`**`tail`**(*table*, *n=10*)

Choose the last n data rows.

E.g.:

```
>>> from petl import tail, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 5     |
+-------+-------+
| 'd'   | 7     |
+-------+-------+
| 'f'   | 42    |
+-------+-------+
| 'f'   | 3     |
+-------+-------+
| 'h'   | 90    |
+-------+-------+
| 'k'   | 12    |
+-------+-------+
| 'l'   | 77    |
+-------+-------+
| 'q'   | 2     |
+-------+-------+

>>> table2 = tail(table1, 4)
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'h'   | 90    |
+-------+-------+
| 'k'   | 12    |
+-------+-------+
| 'l'   | 77    |
+-------+-------+
| 'q'   | 2     |
+-------+-------+
```

See also `head()`, `rowslice()`.

`petl.`**`cut`**(*table*, **args*, ***kwargs*)

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

```
>>> from petl import look, cut
>>> look(table1)
```

```
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | 2.7   |
+-------+-------+-------+
| 'B'   | 2     | 3.4   |
+-------+-------+-------+
| 'B'   | 3     | 7.8   |
+-------+-------+-------+
| 'D'   | 42    | 9.0   |
+-------+-------+-------+
| 'E'   | 12    |       |
+-------+-------+-------+

>>> table2 = cut(table1, 'foo', 'baz')
>>> look(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 = cut(table1, 0, 2)
>>> look(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 = cut(table1, 'bar', 0)
>>> look(table4)
+-------+-------+
| 'bar' | 'foo' |
+=======+=======+
| 1     | 'A'   |
+-------+-------+
| 2     | 'B'   |
+-------+-------+
| 3     | 'B'   |
```

```
+-------+-------+
| 42    | 'D'   |
+-------+-------+
| 12    | 'E'   |
+-------+-------+

>>> # select a range of fields
... table5 = cut(table1, *range(0, 2))
>>> look(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 `cutout()`.

`petl.`**`cutout`**(*table*, **args*, ***kwargs*)

Remove fields. E.g.:

```
>>> from petl import cutout, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | 2.7   |
+-------+-------+-------+
| 'B'   | 2     | 3.4   |
+-------+-------+-------+
| 'B'   | 3     | 7.8   |
+-------+-------+-------+
| 'D'   | 42    | 9.0   |
+-------+-------+-------+
| 'E'   | 12    |       |
+-------+-------+-------+

>>> table2 = cutout(table1, 'bar')
>>> look(table2)
+-------+-------+
| 'foo' | 'baz' |
+=======+=======+
| 'A'   | 2.7   |
+-------+-------+
| 'B'   | 3.4   |
+-------+-------+
| 'B'   | 7.8   |
+-------+-------+
| 'D'   | 9.0   |
```

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

See also `cut()`.

New in version 0.3.

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

Select rows meeting a condition. E.g.:

```
>>> from petl import select, look
>>> look(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 record
... table2 = select(table1, lambda rec: rec[0] == 'a' and rec[1] > 88.1)
... # table2 = select(table1, lambda rec: rec['foo'] == 'a' and rec['baz'] > 88.1)
... # table2 = select(table1, lambda rec: rec.foo == 'a' and rec.baz > 88.1)
>>> look(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 expr()
... table3 = select(table1, "{foo} == 'a' and {baz} > 88.1")
>>> look(table3)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'a'   | 2     | 88.2  |
+-------+-------+-------+

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

```
+-------+-------+-------+
```

Changed in version 0.4.

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

petl.**selectop**(*table*, *field*, *value*, *op*, *complement=False*)
Select rows where the function *op* applied to the given field and the given value returns true.

Changed in version 0.4.

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

petl.**selecteq**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field equals the given value.

Changed in version 0.4.

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

petl.**selectne**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field does not equal the given value.

Changed in version 0.4.

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

petl.**selectlt**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field is less than the given value.

Changed in version 0.4.

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

petl.**selectle**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field is less than or equal to the given value.

Changed in version 0.4.

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

petl.**selectgt**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field is greater than the given value.

Changed in version 0.4.

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

petl.**selectge**(*table*, *field*, *value*, *complement=False*)
Select rows where the given field is greater than or equal to the given value.

Changed in version 0.4.

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

petl.**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*.

Changed in version 0.4.

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

petl.**selectrangeopenleft**(*table*, *field*, *minv*, *maxv*, *complement=False*)
Select rows where the given field is greater than or equal to *minv* and less than *maxv*.

Changed in version 0.4.

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

petl.**selectrangeopenright**(*table*, *field*, *minv*, *maxv*, *complement=False*)
Select rows where the given field is greater than *minv* and less than or equal to *maxv*.

Changed in version 0.4.

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

petl.**selectrangeclosed**(*table*, *field*, *minv*, *maxv*, *complement=False*)
Select rows where the given field is greater than *minv* and less than *maxv*.

Changed in version 0.4.

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

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

New in version 0.10.

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

Changed in version 0.4.

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

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

Changed in version 0.4.

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

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

Changed in version 0.4.

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

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

Changed in version 0.4.

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

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

Changed in version 0.4.

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

petl.**selectre**(*table*, *field*, *pattern*, *flags=0*, *complement=False*)
Select rows where a regular expression search using the given pattern on the given field returns a match. E.g.:

```
>>> from petl import selectre, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'aa'  | 4     | 9.3   |
+-------+-------+-------+
| 'aaa' | 2     | 88.2  |
+-------+-------+-------+
| 'b'   | 1     | 23.3  |
+-------+-------+-------+
| 'ccc' | 8     | 42.0  |
+-------+-------+-------+
| 'bb'  | 7     | 100.9 |
+-------+-------+-------+
| 'c'   | 2     |       |
+-------+-------+-------+

>>> table2 = selectre(table1, 'foo', '[ab]{2}')
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'aa'  | 4     | 9.3   |
+-------+-------+-------+
| 'aaa' | 2     | 88.2  |
+-------+-------+-------+
| 'bb'  | 7     | 100.9 |
+-------+-------+-------+
```

See also `re.search()`.

Changed in version 0.4.

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

petl.**rowselect**(*table*, *where*, *complement=False*)
Select rows matching a condition. The *where* argument should be a function accepting a hybrid row object (supports accessing values either by position or by field name) as argument and returning True or False.

Deprecated since version 0.10.

Use `select()` instead, it supports the same signature.

petl.**recordselect**(*table*, *where*, *missing=None*, *complement=False*)
Select rows matching a condition. The *where* argument should be a function accepting a record (row as dictionary of values indexed by field name) as argument and returning True or False.

Deprecated since version 0.9.

Use `select()` instead.

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

Changed in version 0.4.

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

`petl.`**`fieldselect`**(*table*, *field*, *where*, *complement=False*)
Select rows matching a condition. The *where* argument should be a function accepting a single data value as argument and returning True or False.

Deprecated since version 0.10.

Use `select()` instead, it supports the same signature.

`petl.`**`facet`**(*table*, *field*)
Return a dictionary mapping field values to tables.

E.g.:

```
>>> from petl import facet, look
>>> look(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 = facet(table1, 'foo')
>>> foo.keys()
['a', 'c', 'b', 'd']
>>> look(foo['a'])
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'a'   | 4     | 9.3   |
+-------+-------+-------+
| 'a'   | 2     | 88.2  |
+-------+-------+-------+

>>> look(foo['c'])
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'c'   | 8     | 42.0  |
+-------+-------+-------+
| 'c'   | 2     |       |
+-------+-------+-------+
```

See also `facetcolumns()`.

petl.**rangefacet**(*table*, *field*, *width*, *minv=None*, *maxv=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Return a dictionary mapping ranges to tables. E.g.:

```
>>> from petl import rangefacet, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 3     |
+-------+-------+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
| 'c'   | 4     |
+-------+-------+
| 'd'   | 3     |
+-------+-------+

>>> rf = rangefacet(table1, 'bar', 2)
>>> rf.keys()
[(1, 3), (3, 5), (5, 7), (7, 9)]
>>> look(rf[(1, 3)])
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+

>>> look(rf[(7, 9)])
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
```

Note that the last bin includes both edges.

petl.**replace**(*table*, *field*, *a*, *b*)

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

New in version 0.5.

petl.**replaceall**(*table*, *a*, *b*)

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

New in version 0.5.

petl.**convert**(*table*, **args*, ***kwargs*)

Transform values under one or more fields via arbitrary functions, method invocations or dictionary translations.

E.g.:

```
>>> from petl import convert, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | '2.4' | 12    |
+-------+-------+-------+
| 'B'   | '5.7' | 34    |
+-------+-------+-------+
| 'C'   | '1.2' | 56    |
+-------+-------+-------+

>>> # using the built-in float function:
... table2 = convert(table1, 'bar', float)
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 2.4   | 12    |
+-------+-------+-------+
| 'B'   | 5.7   | 34    |
+-------+-------+-------+
| 'C'   | 1.2   | 56    |
+-------+-------+-------+

>>> # using a lambda function::
... table3 = convert(table1, 'baz', lambda v: v*2)
>>> look(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 = convert(table1, 'foo', 'lower')
>>> look(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 = convert(table1, 'foo', 'replace', 'A', 'AA')
>>> look(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 = convert(table1, 'foo', {'A': 'Z', 'B': 'Y'})
>>> look(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 = convert(table1, ('foo', 'bar', 'baz'), unicode)
>>> look(table8)
+-------+--------+-------+
| 'foo' | 'bar'  | 'baz' |
+=======+========+=======+
| u'A'  | u'2.4' | u'12' |
+-------+--------+-------+
| u'B'  | u'5.7' | u'34' |
+-------+--------+-------+
| u'C'  | u'1.2' | u'56' |
+-------+--------+-------+

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

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

```
>>> # ...or alternatively via suffix notation on the returned table object
... table11 = convert(table1)
>>> table11['foo'] = 'lower'
>>> table11['bar'] = float
>>> table11['baz'] = lambda v: v*2
>>> look(table11)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'a'   | 2.4   | 24    |
+-------+-------+-------+
| 'b'   | 5.7   | 68    |
+-------+-------+-------+
| 'c'   | 1.2   | 112   |
+-------+-------+-------+

>>> # conversion can be conditional
... table12 = convert(table1, 'baz', lambda v: v*2, where=lambda r: r.foo == 'B')
>>> look(table12)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | '2.4' |    12 |
+-------+-------+-------+
| 'B'   | '5.7' |    68 |
+-------+-------+-------+
| 'C'   | '1.2' |    56 |
+-------+-------+-------+
```

Note that either field names or indexes can be given.

Changed in version 0.11.

Now supports multiple field conversions.

Changed in version 0.22.

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.**convertall**(*table*, **args*, ***kwargs*)

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

New in version 0.4.

petl.**fieldconvert**(*table*, *converters=None*, *failonerror=False*, *errorvalue=None*, ***kwargs*)

Transform values in one or more fields via functions or method invocations.

Deprecated since version 0.11.

Use `convert()` instead.

petl.**convertnumbers**(*table*, ***kwargs*)

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

```
>>> from petl import convertnumbers, look
>>> look(table1)
+-------+-------+--------+--------+
| 'foo' | 'bar' | 'baz'  | 'quux' |
+=======+=======+========+========+
```

```
| '1'   | '3.0' | '9+3j' | 'aaa'  |
+-------+-------+--------+--------+
| '2'   | '1.3' | '7+2j' | None   |
+-------+-------+--------+--------+

>>> table2 = convertnumbers(table1)
>>> look(table2)
+-------+-------+--------+--------+
| 'foo' | 'bar' | 'baz'  | 'quux' |
+=======+=======+========+========+
| 1     | 3.0   | (9+3j) | 'aaa'  |
+-------+-------+--------+--------+
| 2     | 1.3   | (7+2j) | None   |
+-------+-------+--------+--------+
```

New in version 0.4.

petl.**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.:

```
>>> from petl import search, look
>>> look(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 = search(table1, '.g.')
>>> look(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 = search(table1, 'foo', '.g.')
>>> look(table3)
+----------+-------+-------------------------+
| 'foo'    | 'bar' | 'baz'                   |
+==========+=======+=========================+
| 'orange' | 12    | 'oranges are nice fruit' |
+----------+-------+-------------------------+
| 'mango'  | 42    | 'I like them'           |
+----------+-------+-------------------------+
```

New in version 0.10.

petl.**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()`.

New in version 0.5.

Changed in version 0.10.

Renamed 'resub' to 'sub'.

petl.**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.:

```
>>> from petl import split, look
>>> look(table1)
+------+------------+---------+
| 'id' | 'variable' | 'value' |
+======+============+=========+
| '1'  | 'parad1'   | '12'    |
+------+------------+---------+
| '2'  | 'parad2'   | '15'    |
+------+------------+---------+
| '3'  | 'tempd1'   | '18'    |
+------+------------+---------+
| '4'  | 'tempd2'   | '19'    |
+------+------------+---------+

>>> table2 = split(table1, 'variable', 'd', ['variable', 'day'])
>>> look(table2)
+------+---------+------------+-------+
| 'id' | 'value' | 'variable' | 'day' |
+======+=========+============+=======+
| '1'  | '12'    | 'para'     | '1'   |
+------+---------+------------+-------+
| '2'  | '15'    | 'para'     | '2'   |
+------+---------+------------+-------+
| '3'  | '18'    | 'temp'     | '1'   |
+------+---------+------------+-------+
| '4'  | '19'    | 'temp'     | '2'   |
+------+---------+------------+-------+
```

See also `re.split()`.

petl.**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.:

```
>>> from petl import capture, look
>>> look(table1)
+------+------------+---------+
| 'id' | 'variable' | 'value' |
+======+============+=========+
| '1'  | 'A1'       | '12'    |
+------+------------+---------+
| '2'  | 'A2'       | '15'    |
+------+------------+---------+
| '3'  | 'B1'       | '18'    |
+------+------------+---------+
| '4'  | 'C12'      | '19'    |
```

```
+------+-----------+---------+

>>> table2 = capture(table1, 'variable', '(\w)(\d+)', ['treat', 'time'])
>>> look(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 = capture(table1, 'variable', '(\w)(\d+)', ['treat', 'time'], include_original=True)
>>> look(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.

See also `split()`, `re.search()`.

..versionchanged:: 0.18

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.

`petl.`**`unpack`**(*table*, *field*, *newfields=None*, *maxunpack=None*, *include_original=False*)

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

```
>>> from petl import unpack, look
>>> look(table1)
+-------+------------+
| 'foo' | 'bar'      |
+=======+============+
| 1     | ['a', 'b'] |
+-------+------------+
| 2     | ['c', 'd'] |
+-------+------------+
| 3     | ['e', 'f'] |
+-------+------------+

>>> table2 = unpack(table1, 'bar', ['baz', 'quux'])
>>> look(table2)
```

```
+-------+-------+--------+
| 'foo' | 'baz' | 'quux' |
+=======+=======+========+
| 1     | 'a'   | 'b'    |
+-------+-------+--------+
| 2     | 'c'   | 'd'    |
+-------+-------+--------+
| 3     | 'e'   | 'f'    |
+-------+-------+--------+
```

See also `unpackdict()`.

petl.**unpackdict**(*table*, *field*, *keys=None*, *includeoriginal=False*, *samplesize=1000*, *missing=None*)

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

```
>>> from petl import unpackdict, look
>>> look(table1)
+-------+--------------------------+
| 'foo' | 'bar'                    |
+=======+==========================+
| 1     | {'quux': 'b', 'baz': 'a'} |
+-------+--------------------------+
| 2     | {'quux': 'd', 'baz': 'c'} |
+-------+--------------------------+
| 3     | {'quux': 'f', 'baz': 'e'} |
+-------+--------------------------+

>>> table2 = unpackdict(table1, 'bar')
>>> look(table2)
+-------+-------+--------+
| 'foo' | 'baz' | 'quux' |
+=======+=======+========+
| 1     | 'a'   | 'b'    |
+-------+-------+--------+
| 2     | 'c'   | 'd'    |
+-------+-------+--------+
| 3     | 'e'   | 'f'    |
+-------+-------+--------+
```

New in version 0.10.

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

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

```
>>> from petl import fieldmap, look
>>> look(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     |
+------+----------+-------+----------+----------+
```

```
>>> from collections import OrderedDict
>>> 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 = fieldmap(table1, mappings)
>>> look(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   |
+--------------+----------+--------------+--------------------+

>>> # field mappings can also be added and/or updated after the table is created
... # via the suffix notation
... table3 = fieldmap(table1)
>>> table3['subject_id'] = 'id'
>>> table3['gender'] = 'sex', {'male': 'M', 'female': 'F'}
>>> table3['age_months'] = 'age', lambda v: v * 12
>>> # use an expression string this time
... table3['bmi'] = '{weight} / {height}**2'
>>> look(table3)
+--------------+----------+--------------+--------------------+
| '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 `expr()`.

petl.**rowmap**(*table*, *rowmapper*, *fields*, *failonerror=False*, *missing=None*)

Transform rows via an arbitrary function. E.g.:

```
>>> from petl import rowmap, look
>>> look(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[1]] if row[1] in transmf else row[1],
...             row[2] * 12,
...             row[4] / row[3] ** 2]
...
>>> table2 = rowmap(table1, rowmapper, fields=['subject_id', 'gender', 'age_months', 'bmi'])
>>> look(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   |
+--------------+----------+--------------+--------------------+
```

The *rowmapper* function should return a single row (list or tuple).

Changed in version 0.9.

Hybrid row objects supporting data value access by either position or by field name are now passed to the *rowmapper* function.

petl.**recordmap**(*table*, *recmapper*, *fields*, *failonerror=False*)
Transform records via an arbitrary function.

Deprecated since version 0.9.

Use `rowmap()` insteand.

petl.**rowmapmany**(*table*, *rowgenerator*, *fields*, *failonerror=False*, *missing=None*)
Map each input row to any number of output rows via an arbitrary function. E.g.:

```
>>> from petl import rowmapmany, look
>>> look(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[1]] if row[1] in transmf else row[1]]
...     yield [row[0], 'age_months', row[2] * 12]
...     yield [row[0], 'bmi', row[4] / row[3] ** 2]
...
>>> table2 = rowmapmany(table1, rowgenerator, fields=['subject_id', 'variable', 'value'])
>>> look(table2)
+--------------+-------------+--------------------+
| 'subject_id' | 'variable'  | 'value'            |
+==============+=============+====================+
| 1            | 'gender'    | 'M'                |
+--------------+-------------+--------------------+
| 1            | 'age_months' | 192               |
+--------------+-------------+--------------------+
| 1            | 'bmi'       | 29.48870392390012  |
+--------------+-------------+--------------------+
| 2            | 'gender'    | 'F'                |
+--------------+-------------+--------------------+
| 2            | 'age_months' | 228               |
+--------------+-------------+--------------------+
| 2            | 'bmi'       | 30.8531967030519   |
+--------------+-------------+--------------------+
| 3            | 'gender'    | '-'                |
+--------------+-------------+--------------------+
| 3            | 'age_months' | 204               |
+--------------+-------------+--------------------+
| 3            | 'bmi'       | 23.481883600555488 |
+--------------+-------------+--------------------+
| 4            | 'gender'    | 'M'                |
+--------------+-------------+--------------------+
```

The *rowgenerator* function should yield zero or more rows (lists or tuples).

See also the `melt()` function.

Changed in version 0.9.

Hybrid row objects supporting data value access by either position or by field name are now passed to the *rowgenerator* function.

`petl.`**`recordmapmany`**(*table*, *rowgenerator*, *fields*, *failonerror=False*)

Map each input row (as a record) to any number of output rows via an arbitrary function.

Deprecated since version 0.9.

Use `rowmapmany()` instead.

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

Concatenate data from two or more tables. E.g.:

```
>>> from petl import look, cat
>>> look(table1)
+-------+-------+
```

```
| 'foo' | 'bar' |
+=======+=======+
| 1     | 'A'   |
+-------+-------+
| 2     | 'B'   |
+-------+-------+

>>> look(table2)
+-------+-------+
| 'bar' | 'baz' |
+=======+=======+
| 'C'   | True  |
+-------+-------+
| 'D'   | False |
+-------+-------+

>>> table3 = cat(table1, table2)
>>> look(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
... look(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  |        |      |
+-------+-------+--------+------+

>>> look(cat(table4))
+-------+-------+--------+
| 'foo' | 'bar' | 'baz'  |
+=======+=======+========+
| 'A'   | 1     | 2      |
+-------+-------+--------+
| 'B'   | '2'   | '3.4'  |
+-------+-------+--------+
| u'B'  | u'3'  | u'7.8' |
+-------+-------+--------+
| 'D'   | 'xyz' | 9.0    |
+-------+-------+--------+
```

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

>>> # use the header keyword argument to specify a fixed set of fields
... look(table5)
+--------+--------+
| 'bar'  | 'foo'  |
+========+========+
| 'A'    | 1      |
+--------+--------+
| 'B'    | 2      |
+--------+--------+

>>> table6 = cat(table5, header=['A', 'foo', 'B', 'bar', 'C'])
>>> look(table6)
+------+-------+------+-------+------+
| 'A'  | 'foo' | 'B'  | 'bar' | 'C'  |
+======+=======+======+=======+======+
| None | 1     | None | 'A'   | None |
+------+-------+------+-------+------+
| None | 2     | None | 'B'   | None |
+------+-------+------+-------+------+

>>> # using the header keyword argument with two input tables
... look(table7)
+--------+--------+
| 'bar'  | 'foo'  |
+========+========+
| 'A'    | 1      |
+--------+--------+
| 'B'    | 2      |
+--------+--------+

>>> look(table8)
+--------+--------+
| 'bar'  | 'baz'  |
+========+========+
| 'C'    | True   |
+--------+--------+
| 'D'    | False  |
+--------+--------+

>>> table9 = cat(table7, table8, header=['A', 'foo', 'B', 'bar', 'C'])
>>> look(table9)
+------+-------+------+-------+------+
| '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.

Changed in version 0.5.

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.**duplicates**(*table*, *key=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import duplicates, look
>>> look(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 = duplicates(table1, 'foo')
>>> look(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 = duplicates(table1, key=['foo', 'bar'])
>>> look(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 *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 `sort()` function.

See also `unique()` and `distinct()`.

petl.**unique**(*table*, *key=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import unique, look
>>> look(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 = unique(table1, 'foo')
>>> look(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 *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 `sort()` function.

New in version 0.10.

See also `duplicates()` and `distinct()`.

petl.**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.:

```
>>> from petl import conflicts, look
>>> look(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 = conflicts(table1, 'foo')
>>> look(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.

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 `sort()` function.

Changed in version 0.8.

Added the *include* and *exclude* keyword arguments. The *exclude* keyword argument replaces the *ignore* keyword argument in previous versions.

`petl.`**`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.:

```
>>> from petl import sort, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'C'   | 2     |
+-------+-------+
| 'A'   | 9     |
+-------+-------+
| 'A'   | 6     |
+-------+-------+
| 'F'   | 1     |
+-------+-------+
| 'D'   | 10    |
+-------+-------+

>>> table2 = sort(table1, 'foo')
>>> look(table2)
```

```
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 9     |
+-------+-------+
| 'A'   | 6     |
+-------+-------+
| 'C'   | 2     |
+-------+-------+
| 'D'   | 10    |
+-------+-------+
| 'F'   | 1     |
+-------+-------+

>>> # sorting by compound key is supported
... table3 = sort(table1, key=['foo', 'bar'])
>>> look(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 = sort(table1)
>>> look(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.transform.defaultbuffersize* will be used. By default this is set to 100000 rows, but can be changed, e.g.:

```
>>> import petl.transform
>>> petl.transform.defaultbuffersize = 500000
```

If *petl.transform.defaultbuffersize* is set to *None*, this forces all sorting to be done entirely in memory.

Changed in version 0.16.

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.**join**(*left*, *right*, *key=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import join, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 3    | 'purple' |
+------+----------+

>>> look(table2)
+------+-----------+
| 'id' | 'shape'   |
+======+===========+
| 1    | 'circle'  |
+------+-----------+
| 3    | 'square'  |
+------+-----------+
| 4    | 'ellipse' |
+------+-----------+

>>> table3 = join(table1, table2, key='id')
>>> look(table3)
+------+----------+----------+
| 'id' | 'colour' | 'shape'  |
+======+==========+==========+
| 1    | 'blue'   | 'circle' |
+------+----------+----------+
| 3    | 'purple' | 'square' |
+------+----------+----------+

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

>>> # note behaviour if the key is not unique in either or both tables
... look(table5)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
```

```
+------+----------+
| 1    | 'red'    |
+------+----------+
| 2    | 'purple' |
+------+----------+

>>> look(table6)
+------+-----------+
| 'id' | 'shape'   |
+======+===========+
| 1    | 'circle'  |
+------+-----------+
| 1    | 'square'  |
+------+-----------+
| 2    | 'ellipse' |
+------+-----------+

>>> table7 = join(table5, table6, key='id')
>>> look(table7)
+------+----------+-----------+
| 'id' | 'colour' | 'shape'   |
+======+==========+===========+
| 1    | 'blue'   | 'circle'  |
+------+----------+-----------+
| 1    | 'blue'   | 'square'  |
+------+----------+-----------+
| 1    | 'red'    | 'circle'  |
+------+----------+-----------+
| 1    | 'red'    | 'square'  |
+------+----------+-----------+
| 2    | 'purple' | 'ellipse' |
+------+----------+-----------+

>>> # compound keys are supported
... look(table8)
+------+--------+----------+
| 'id' | 'time' | 'height' |
+======+========+==========+
| 1    | 1      | 12.3     |
+------+--------+----------+
| 1    | 2      | 34.5     |
+------+--------+----------+
| 2    | 1      | 56.7     |
+------+--------+----------+

>>> look(table9)
+------+--------+----------+
| 'id' | 'time' | 'weight' |
+======+========+==========+
| 1    | 2      | 4.5      |
+------+--------+----------+
| 2    | 1      | 6.7      |
+------+--------+----------+
| 2    | 2      | 8.9      |
+------+--------+----------+

>>> table10 = join(table8, table9, key=['id', 'time'])
>>> look(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 *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 `sort()` function.

petl.**leftjoin**(*left*, *right*, *key=None*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import leftjoin, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 3    | 'purple' |
+------+----------+

>>> look(table2)
+------+-----------+
| 'id' | 'shape'   |
+======+===========+
| 1    | 'circle'  |
+------+-----------+
| 3    | 'square'  |
+------+-----------+
| 4    | 'ellipse' |
+------+-----------+

>>> table3 = leftjoin(table1, table2, key='id')
>>> look(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 *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 `sort()` function.

petl.**lookupjoin**(*left*, *right*, *key=None*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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.:

```
>>> from petl import lookupjoin, look
>>> look(table1)
+------+----------+--------+
| 'id' | 'color'  | 'cost' |
+======+==========+========+
| 1    | 'blue'   | 12     |
+------+----------+--------+
| 2    | 'red'    | 8      |
+------+----------+--------+
| 3    | 'purple' | 4      |
+------+----------+--------+

>>> look(table2)
+------+-----------+---------+
| 'id' | 'shape'   | 'size'  |
+======+===========+=========+
| 1    | 'circle'  | 'big'   |
+------+-----------+---------+
| 1    | 'circle'  | 'small' |
+------+-----------+---------+
| 2    | 'square'  | 'tiny'  |
+------+-----------+---------+
| 2    | 'square'  | 'big'   |
+------+-----------+---------+
| 3    | 'ellipse' | 'small' |
+------+-----------+---------+
| 3    | 'ellipse' | 'tiny'  |
+------+-----------+---------+

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

See also `leftjoin()`.

New in version 0.11.

petl.**rightjoin**(*left*, *right*, *key=None*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import rightjoin, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 3    | 'purple' |
```

```
+------+----------+

>>> look(table2)
+------+-----------+
| 'id' | 'shape'   |
+======+===========+
| 1    | 'circle'  |
+------+-----------+
| 3    | 'square'  |
+------+-----------+
| 4    | 'ellipse' |
+------+-----------+

>>> table3 = rightjoin(table1, table2, key='id')
>>> look(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 *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 `sort()` function.

petl.**outerjoin**(*left*, *right*, *key=None*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import outerjoin, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 3    | 'purple' |
+------+----------+

>>> look(table2)
+------+-----------+
| 'id' | 'shape'   |
+======+===========+
| 1    | 'circle'  |
+------+-----------+
| 3    | 'square'  |
+------+-----------+
| 4    | 'ellipse' |
+------+-----------+

>>> table3 = outerjoin(table1, table2, key='id')
>>> look(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 *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 `sort()` function.

petl.**crossjoin**(**tables*)

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

```
>>> from petl import crossjoin, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+

>>> look(table2)
+------+----------+
| 'id' | 'shape'  |
+======+==========+
| 1    | 'circle' |
+------+----------+
| 3    | 'square' |
+------+----------+

>>> table3 = crossjoin(table1, table2)
>>> look(table3)
+------+----------+------+----------+
| 'id' | 'colour' | 'id' | 'shape'  |
+======+==========+======+==========+
| 1    | 'blue'   | 1    | 'circle' |
+------+----------+------+----------+
| 1    | 'blue'   | 3    | 'square' |
+------+----------+------+----------+
| 2    | 'red'    | 1    | 'circle' |
+------+----------+------+----------+
| 2    | 'red'    | 3    | 'square' |
+------+----------+------+----------+
```

See also `join()`, `leftjoin()`, `rightjoint()`, `outerjoin()`.

petl.**antijoin**(*left*, *right*, *key=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import antijoin, look
>>> look(table1)
+------+----------+
```

```
| 'id' | 'colour' |
+======+==========+
| 0    | 'black'  |
+------+----------+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 4    | 'yellow' |
+------+----------+
| 5    | 'white'  |
+------+----------+

>>> look(table2)
+------+----------+
| 'id' | 'shape'  |
+======+==========+
| 1    | 'circle' |
+------+----------+
| 3    | 'square' |
+------+----------+

>>> table3 = antijoin(table1, table2, key='id')
>>> look(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 *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 `sort()` function.

petl.**complement**(*a*, *b*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import complement, look
>>> look(a)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> look(b)
+-----+-----+-------+
```

```
| 'x' | 'y' | 'z'   |
+=====+=====+=======+
| 'B' | 2   | False |
+-----+-----+-------+
| 'A' | 9   | False |
+-----+-----+-------+
| 'B' | 3   | True  |
+-----+-----+-------+
| 'C' | 9   | True  |
+-----+-----+-------+

>>> aminusb = complement(a, b)
>>> look(aminusb)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+

>>> bminusa = complement(b, a)
>>> look(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 `recordcomplement()` function.

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 `sort()` function.

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

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

```
>>> from petl import diff, look
>>> look(a)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> look(b)
+-----+-----+-------+
| 'x' | 'y' | 'z'   |
+=====+=====+=======+
```

```
| 'B' | 2   | False |
+-----+-----+-------+
| 'A' | 9   | False |
+-----+-----+-------+
| 'B' | 3   | True  |
+-----+-----+-------+
| 'C' | 9   | True  |
+-----+-----+-------+

>>> added, subtracted = diff(a, b)
>>> # rows in b not in a
... look(added)
+-----+-----+-------+
| 'x' | 'y' | 'z'   |
+=====+=====+=======+
| 'A' | 9   | False |
+-----+-----+-------+
| 'B' | 3   | True  |
+-----+-----+-------+

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

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

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 `sort()` function.

petl.**recordcomplement**(*a*, *b*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import recordcomplement, look
>>> look(a)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> look(b)
+-------+-------+-------+
| 'bar' | 'foo' | 'baz' |
+=======+=======+=======+
| 2     | 'B'   | False |
+-------+-------+-------+
```

```
| 9     | 'A'   | False |
+-------+-------+-------+
| 3     | 'B'   | True  |
+-------+-------+-------+
| 9     | 'C'   | True  |
+-------+-------+-------+

>>> aminusb = recordcomplement(a, b)
>>> look(aminusb)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+

>>> bminusa = recordcomplement(b, a)
>>> look(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 `complement()` function.

See also the discussion of the *buffersize*, *tempdir* and *cache* arguments under the `sort()` function.

New in version 0.3.

`petl.`**`recorddiff`**(*a*, *b*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import recorddiff, look
>>> look(a)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> look(b)
+-------+-------+-------+
| 'bar' | 'foo' | 'baz' |
+=======+=======+=======+
| 2     | 'B'   | False |
+-------+-------+-------+
| 9     | 'A'   | False |
+-------+-------+-------+
```

```
| 3     | 'B'   | True  |
+-------+-------+-------+
| 9     | 'C'   | True  |
+-------+-------+-------+

>>> added, subtracted = recorddiff(a, b)
>>> look(added)
+-------+-------+-------+
| 'bar' | 'foo' | 'baz' |
+=======+=======+=======+
| 3     | 'B'   | True  |
+-------+-------+-------+
| 9     | 'A'   | False |
+-------+-------+-------+

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

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

See also the discussion of the *buffersize*, *tempdir* and *cache* arguments under the `sort()` function.

New in version 0.3.

petl.**intersection**(*a*, *b*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import intersection, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> look(table2)
+-----+-----+-------+
| 'x' | 'y' | 'z'   |
+=====+=====+=======+
| 'B' | 2   | False |
+-----+-----+-------+
| 'A' | 9   | False |
+-----+-----+-------+
| 'B' | 3   | True  |
+-----+-----+-------+
| 'C' | 9   | True  |
```

```
+-----+-----+-------+

>>> table3 = intersection(table1, table2)
>>> look(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 *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 `sort()` function.

petl.**aggregate**(*table*, *key*, *aggregation=None*, *value=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import aggregate, look
>>> look(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 = aggregate(table1, 'foo', len)
>>> look(table2)
+-------+---------+
| 'key' | 'value' |
+=======+=========+
| 'a'   |       2 |
+-------+---------+
| 'b'   |       3 |
+-------+---------+
| 'c'   |       1 |
+-------+---------+

>>> # aggregate single field
... table3 = aggregate(table1, 'foo', sum, 'bar')
>>> look(table3)
+-------+---------+
| 'key' | 'value' |
+=======+=========+
| 'a'   |      10 |
+-------+---------+
```

```
| 'b'   |      13 |
+-------+---------+
| 'c'   |       4 |
+-------+---------+

>>> # alternative signature for single field aggregation using keyword args
... table4 = aggregate(table1, key=('foo', 'bar'), aggregation=list, value=('bar', 'baz'))
>>> look(table4)
+----------+-------------------------+
| 'key'    | '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
>>> from petl import strjoin
>>> aggregation = OrderedDict()
>>> aggregation['count'] = len
>>> aggregation['minbar'] = 'bar', min
>>> aggregation['maxbar'] = 'bar', max
>>> aggregation['sumbar'] = 'bar', sum
>>> aggregation['listbar'] = 'bar' # default aggregation function is list
>>> aggregation['listbarbaz'] = ('bar', 'baz'), list
>>> aggregation['bars'] = 'bar', strjoin(', ')
>>> table5 = aggregate(table1, 'foo', aggregation)
>>> look(table5)
+-------+---------+----------+----------+----------+-----------+-------------------------------
| 'key' | 'count' | 'minbar' | 'maxbar' | 'sumbar' | 'listbar' | 'listbarbaz'
+=======+=========+==========+==========+==========+===========+===============================
| 'a'   |       2 |        3 |        7 |       10 | [3, 7]    | [(3, True), (7, False)]
+-------+---------+----------+----------+----------+-----------+-------------------------------
| 'b'   |       3 |        2 |        9 |       13 | [2, 2, 9] | [(2, True), (2, False), (9, Fal
+-------+---------+----------+----------+----------+-----------+-------------------------------
| 'c'   |       1 |        4 |        4 |        4 | [4]       | [(4, True)]
+-------+---------+----------+----------+----------+-----------+-------------------------------

>>> # can also use list or tuple to specify multiple field aggregation
... aggregation = [('count', len),
...                ('minbar', 'bar', min),
...                ('maxbar', 'bar', max),
...                ('sumbar', 'bar', sum),
...                ('listbar', 'bar'), # default aggregation function is list
...                ('listbarbaz', ('bar', 'baz'), list),
...                ('bars', 'bar', strjoin(', '))]
>>> table6 = aggregate(table1, 'foo', aggregation)
>>> look(table6)
+-------+---------+----------+----------+----------+-----------+-------------------------------
| 'key' | 'count' | 'minbar' | 'maxbar' | 'sumbar' | 'listbar' | 'listbarbaz'
+=======+=========+==========+==========+==========+===========+===============================
```

```
| 'a'   |       2 |        3 |        7 |       10 | [3, 7]    | [(3, True), (7, False)]
+-------+---------+----------+----------+----------+-----------+--------------------------
| 'b'   |       3 |        2 |        9 |       13 | [2, 2, 9] | [(2, True), (2, False), (9, Fal
+-------+---------+----------+----------+----------+-----------+--------------------------
| 'c'   |       1 |        4 |        4 |        4 | [4]       | [(4, True)]
+-------+---------+----------+----------+----------+-----------+--------------------------

>>> # can also use suffix notation
... table7 = aggregate(table1, 'foo')
>>> table7['count'] = len
>>> table7['minbar'] = 'bar', min
>>> table7['maxbar'] = 'bar', max
>>> table7['sumbar'] = 'bar', sum
>>> table7['listbar'] = 'bar' # default aggregation function is list
>>> table7['listbarbaz'] = ('bar', 'baz'), list
>>> table7['bars'] = 'bar', strjoin(', ')
>>> look(table7)
+-------+---------+----------+----------+----------+-----------+--------------------------
| 'key' | 'count' | 'minbar' | 'maxbar' | 'sumbar' | 'listbar' | 'listbarbaz'
+=======+=========+==========+==========+==========+===========+==========================
| 'a'   |       2 |        3 |        7 |       10 | [3, 7]    | [(3, True), (7, False)]
+-------+---------+----------+----------+----------+-----------+--------------------------
| 'b'   |       3 |        2 |        9 |       13 | [2, 2, 9] | [(2, True), (2, False), (9, Fal
+-------+---------+----------+----------+----------+-----------+--------------------------
| 'c'   |       1 |        4 |        4 |        4 | [4]       | [(4, True)]
+-------+---------+----------+----------+----------+-----------+--------------------------
```

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 `sort()` function.

petl.**rangeaggregate**(*table*, *key*, *width*, *aggregation=None*, *value=None*, *minv=None*, *maxv=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Group rows into bins then apply aggregation functions. E.g.:

```
>>> from petl import rangeaggregate, look, strjoin
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 3     |
+-------+-------+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
| 'c'   | 4     |
+-------+-------+
| 'd'   | 3     |
+-------+-------+

>>> # aggregate whole rows
... table2 = rangeaggregate(table1, 'bar', 2, len)
>>> look(table2)
+---------+---------+
```

```
| 'key'    | 'value' |
+==========+=========+
| (1, 3)   | 2       |
+----------+---------+
| (3, 5)   | 3       |
+----------+---------+
| (5, 7)   | 0       |
+----------+---------+
| (7, 9)   | 1       |
+----------+---------+
| (9, 11)  | 1       |
+----------+---------+

>>> # aggregate single field
... table3 = rangeaggregate(table1, 'bar', 2, list, 'foo')
>>> look(table3)
+----------+-----------------+
| 'key'    | 'value'         |
+==========+=================+
| (1, 3)   | ['b', 'b']      |
+----------+-----------------+
| (3, 5)   | ['a', 'd', 'c'] |
+----------+-----------------+
| (5, 7)   | []              |
+----------+-----------------+
| (7, 9)   | ['a']           |
+----------+-----------------+
| (9, 11)  | ['b']           |
+----------+-----------------+

>>> # aggregate single field - alternative signature using keyword args
... table4 = rangeaggregate(table1, key='bar', width=2, aggregation=list, value='foo')
>>> look(table4)
+----------+-----------------+
| 'key'    | 'value'         |
+==========+=================+
| (1, 3)   | ['b', 'b']      |
+----------+-----------------+
| (3, 5)   | ['a', 'd', 'c'] |
+----------+-----------------+
| (5, 7)   | []              |
+----------+-----------------+
| (7, 9)   | ['a']           |
+----------+-----------------+
| (9, 11)  | ['b']           |
+----------+-----------------+

>>> # aggregate multiple fields
... from collections import OrderedDict
>>> aggregation = OrderedDict()
>>> aggregation['foocount'] = len
>>> aggregation['foojoin'] = 'foo', strjoin('')
>>> aggregation['foolist'] = 'foo' # default is list
>>> table5 = rangeaggregate(table1, 'bar', 2, aggregation)
>>> look(table5)
+----------+------------+-----------+-----------------+
| 'key'    | 'foocount' | 'foojoin' | 'foolist'       |
+==========+============+===========+=================+
```

```
| (1, 3)  | 2          | 'bb'      | ['b', 'b']      |
+---------+------------+-----------+-----------------+
| (3, 5)  | 3          | 'adc'     | ['a', 'd', 'c'] |
+---------+------------+-----------+-----------------+
| (5, 7)  | 0          | ''        | []              |
+---------+------------+-----------+-----------------+
| (7, 9)  | 1          | 'a'       | ['a']           |
+---------+------------+-----------+-----------------+
| (9, 11) | 1          | 'b'       | ['b']           |
+---------+------------+-----------+-----------------+
```

Changed in version 0.12.

Changed signature to simplify and make consistent with `aggregate()`.

petl.**rangecounts**(*table*, *key*, *width*, *minv=None*, *maxv=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Group rows into bins then count the number of rows in each bin. E.g.:

```
>>> from petl import rangecounts, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 3     |
+-------+-------+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
| 'c'   | 4     |
+-------+-------+
| 'd'   | 3     |
+-------+-------+

>>> table2 = rangecounts(table1, 'bar', 2)
>>> look(table2)
+---------+---------+
| 'key'   | 'value' |
+=========+=========+
| (1, 3)  | 2       |
+---------+---------+
| (3, 5)  | 3       |
+---------+---------+
| (5, 7)  | 0       |
+---------+---------+
| (7, 9)  | 1       |
+---------+---------+
| (9, 11) | 1       |
+---------+---------+
```

See also `rangeaggregate()`.

petl.**rowreduce**(*table*, *key*, *reducer*, *fields=None*, *missing=None*, *presorted=False*, *buffersize=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.:

```
>>> from petl import rowreduce, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 3     |
+-------+-------+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
| 'c'   | 4     |
+-------+-------+

>>> def reducer(key, rows):
...     return [key, sum(row[1] for row in rows)]
...
>>> table2 = rowreduce(table1, key='foo', reducer=reducer, fields=['foo', 'barsum'])
>>> look(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 `aggregate()` and `fold()`.

..versionchanged:: 0.12

Was previously deprecated, now resurrected as it is a useful function in it's own right.

`petl.`**`recordreduce`**(*table*, *key*, *reducer*, *fields=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Deprecated since version 0.9.

Use `rowreduce()` instead.

`petl.`**`rangerowreduce`**(*table*, *key*, *width*, *reducer*, *fields=None*, *minv=None*, *maxv=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import rangerowreduce, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
```

```
| 'a'   | 3     |
+-------+-------+
| 'a'   | 7     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'b'   | 1     |
+-------+-------+
| 'b'   | 9     |
+-------+-------+
| 'c'   | 4     |
+-------+-------+

>>> def reducer(key, rows):
...     return [key[0], key[1], ''.join(row[0] for row in rows)]
...
>>> table2 = rangerowreduce(table1, 'bar', 2, reducer=reducer, fields=['frombar', 'tobar', 'foos
>>> look(table2)
+-----------+---------+--------+
| 'frombar' | 'tobar' | 'foos' |
+===========+=========+========+
| 1         | 3       | 'bb'   |
+-----------+---------+--------+
| 3         | 5       | 'ac'   |
+-----------+---------+--------+
| 5         | 7       | ''     |
+-----------+---------+--------+
| 7         | 9       | 'a'    |
+-----------+---------+--------+
| 9         | 11      | 'b'    |
+-----------+---------+--------+
```

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 `rangeaggregate()` and `rangecounts()`.

..versionchanged:: 0.12

Was previously deprecated, now resurrected as it is a useful function in it's own right.

petl.**rangerecordreduce**(*table*, *key*, *width*, *reducer*, *fields=None*, *minv=None*, *maxv=None*, *failonerror=False*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Reduce records grouped into bins under the given key via an arbitrary function.

Deprecated since version 0.9.

Use `rangeaggregate()` instead.

petl.**mergeduplicates**(*table*, *key*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import mergeduplicates, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | 2.7   |
```

```
+-------+-------+-------+
| 'B'   | 2     | None  |
+-------+-------+-------+
| 'D'   | 3     | 9.4   |
+-------+-------+-------+
| 'B'   | None  | 7.8   |
+-------+-------+-------+
| 'E'   | None  | 42.0  |
+-------+-------+-------+
| 'D'   | 3     | 12.3  |
+-------+-------+-------+
| 'A'   | 2     | None  |
+-------+-------+-------+

>>> table2 = mergeduplicates(table1, 'foo')
>>> look(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 *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 `sort()` function.

Changed in version 0.3.

Previously conflicts were reported as a list, this is changed to a tuple in version 0.3.

Changed in version 0.10.

Renamed from 'mergereduce' to 'mergeduplicates'. Conflicts now reported as instance of Conflict.

petl.**mergesort**(**tables*, ***kwargs*)

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

```
>>> from petl import mergesort, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 9     |
+-------+-------+
| 'C'   | 2     |
+-------+-------+
| 'D'   | 10    |
+-------+-------+
| 'A'   | 6     |
+-------+-------+
| 'F'   | 1     |
```

```
+-------+-------+

>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'B'   | 3     |
+-------+-------+
| 'D'   | 10    |
+-------+-------+
| 'A'   | 10    |
+-------+-------+
| 'F'   | 4     |
+-------+-------+

>>> table3 = mergesort(table1, table2, key='foo')
>>> look(table3)
+-------+-------+
| '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* - field name or tuple of fields to sort by (defaults to *None* - lexical sort)
- *reverse* - *True* if sort in reverse (descending) order (defaults to *False*)
- *presorted* - *True* if inputs are already sorted by the given key (defaults to *False*)
- *missing* - value to fill with when input tables have different fields (defaults to *None*)
- *header* - specify a fixed header for the output table
- *buffersize* - limit the number of rows in memory per input table when inputs are not presorted

New in version 0.9.

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

Convenience function to combine multiple tables (via `mergesort()`) then combine duplicate rows by merging

under the given key (via `mergeduplicates()`). E.g.:

```
>>> from petl import look, merge
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 1     | 'A'   | True  |
+-------+-------+-------+
| 2     | 'B'   | None  |
+-------+-------+-------+
| 4     | 'C'   | True  |
+-------+-------+-------+

>>> look(table2)
+-------+-------+--------+
| 'bar' | 'baz' | 'quux' |
+=======+=======+========+
| 'A'   | True  | 42.0   |
+-------+-------+--------+
| 'B'   | False | 79.3   |
+-------+-------+--------+
| 'C'   | False | 12.4   |
+-------+-------+--------+

>>> table3 = merge(table1, table2, key='bar')
>>> look(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 `mergesort()`, except *key* is required.

Changed in version 0.9.

Now uses `mergesort()`, should be more efficient for presorted inputs.

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

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

```
>>> from petl import melt, look
>>> look(table1)
+------+----------+-------+
| 'id' | 'gender' | 'age' |
+======+==========+=======+
| 1    | 'F'      | 12    |
+------+----------+-------+
| 2    | 'M'      | 17    |
+------+----------+-------+
| 3    | 'M'      | 16    |
+------+----------+-------+

>>> table2 = melt(table1, 'id')
>>> look(table2)
```

```
+------+------------+---------+
| '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
... look(table3)
+------+--------+----------+----------+
| 'id' | 'time' | 'height' | 'weight' |
+======+========+==========+==========+
| 1    | 11     | 66.4     | 12.2     |
+------+--------+----------+----------+
| 2    | 16     | 53.2     | 17.3     |
+------+--------+----------+----------+
| 3    | 12     | 34.5     | 9.4      |
+------+--------+----------+----------+

>>> table4 = melt(table3, key=['id', 'time'])
>>> look(table4)
+------+--------+------------+---------+
| '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 = melt(table3, key=['id', 'time'], variables=['height'])
>>> look(table5)
+------+--------+------------+---------+
| 'id' | 'time' | 'variable' | 'value' |
+======+========+============+=========+
| 1    | 11     | 'height'   | 66.4    |
+------+--------+------------+---------+
| 2    | 16     | 'height'   | 53.2    |
+------+--------+------------+---------+
| 3    | 12     | 'height'   | 34.5    |
+------+--------+------------+---------+
```

See also `recast()`.

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

Recast molten data. E.g.:

```
>>> from petl import recast, look
>>> look(table1)
+------+------------+---------+
| 'id' | 'variable' | 'value' |
+======+============+=========+
| 3    | 'age'      | 16      |
+------+------------+---------+
| 1    | 'gender'   | 'F'     |
+------+------------+---------+
| 2    | 'gender'   | 'M'     |
+------+------------+---------+
| 2    | 'age'      | 17      |
+------+------------+---------+
| 1    | 'age'      | 12      |
+------+------------+---------+
| 3    | 'gender'   | 'M'     |
+------+------------+---------+

>>> table2 = recast(table1)
>>> look(table2)
+------+-------+----------+
| 'id' | 'age' | 'gender' |
+======+=======+==========+
| 1    | 12    | 'F'      |
+------+-------+----------+
| 2    | 17    | 'M'      |
+------+-------+----------+
| 3    | 16    | 'M'      |
+------+-------+----------+

>>> # specifying variable and value fields
... look(table3)
+------+----------+--------+
| 'id' | 'vars'   | 'vals' |
+======+==========+========+
| 3    | 'age'    | 16     |
+------+----------+--------+
| 1    | 'gender' | 'F'    |
+------+----------+--------+
| 2    | 'gender' | 'M'    |
+------+----------+--------+
| 2    | 'age'    | 17     |
+------+----------+--------+
| 1    | 'age'    | 12     |
+------+----------+--------+
| 3    | 'gender' | 'M'    |
+------+----------+--------+

>>> table4 = recast(table3, variablefield='vars', valuefield='vals')
>>> look(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
... look(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 = recast(table6, key='id')
>>> look(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 = recast(table6, key='id', reducers={'weight': mean})
>>> look(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)
... look(table9)
+------+------------+---------+
```

```
| 'id' | 'variable' | 'value' |
+======+============+=========+
| 1    | 'gender'   | 'F'     |
+------+------------+---------+
| 2    | 'age'      | 17      |
+------+------------+---------+
| 1    | 'age'      | 12      |
+------+------------+---------+
| 3    | 'gender'   | 'M'     |
+------+------------+---------+

>>> table10 = recast(table9, key='id')
>>> look(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 `melt()`.

petl.**transpose**(*table*)

Transpose rows into columns. E.g.:

```
>>> from petl import transpose, look
>>> look(table1)
+------+----------+
| 'id' | 'colour' |
+======+==========+
| 1    | 'blue'   |
+------+----------+
| 2    | 'red'    |
+------+----------+
| 3    | 'purple' |
+------+----------+
| 5    | 'yellow' |
+------+----------+
| 7    | 'orange' |
+------+----------+

>>> table2 = transpose(table1)
>>> look(table2)
+----------+--------+-------+----------+----------+----------+
| 'id'     | 1      | 2     | 3        | 5        | 7        |
+==========+========+=======+==========+==========+==========+
| 'colour' | 'blue' | 'red' | 'purple' | 'yellow' | 'orange' |
+----------+--------+-------+----------+----------+----------+
```

See also `recast()`.

petl.**pivot**(*table*, *f1*, *f2*, *f3*, *aggfun*, *missing=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

Construct a pivot table. E.g.:

```
>>> from petl import pivot, look
>>> look(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       |
+----------+----------+---------+---------+

>>> table2 = pivot(table1, 'region', 'gender', 'units', sum)
>>> look(table2)
+----------+-------+--------+
| 'region' | 'boy' | 'girl' |
+==========+=======+========+
| 'east'   | 33    | 29     |
+----------+-------+--------+
| 'west'   | 35    | 23     |
+----------+-------+--------+

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

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

See also `recast()`.

petl.**hashjoin**(*left*, *right*, *key=None*, *cache=True*)

Alternative implementation of `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.

New in version 0.5.

Changed in version 0.16.

Added support for caching data from right hand table (only available when *key* is given).

`petl.`**`hashleftjoin`**(*left*, *right*, *key=None*, *missing=None*, *cache=True*)
Alternative implementation of `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.

New in version 0.5.

Changed in version 0.16.

Added support for caching data from right hand table (only available when *key* is given).

`petl.`**`hashlookupjoin`**(*left*, *right*, *key=None*, *missing=None*)
Alternative implementation of `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.

New in version 0.11.

`petl.`**`hashrightjoin`**(*left*, *right*, *key=None*, *missing=None*, *cache=True*)
Alternative implementation of `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.

New in version 0.5.

Changed in version 0.16.

Added support for caching data from left hand table (only available when *key* is given).

`petl.`**`hashantijoin`**(*left*, *right*, *key=None*)
Alternative implementation of `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.

New in version 0.5.

`petl.`**`hashcomplement`**(*a*, *b*)
Alternative implementation of `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.

New in version 0.5.

`petl.`**`hashintersection`**(*a*, *b*)
Alternative implementation of `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.

New in version 0.5.

petl.**flatten**(*table*)

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

```
>>> from petl import flatten, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'C'   | 7     | False |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
| 'C'   | 9     | True  |
+-------+-------+-------+

>>> list(flatten(table1))
['A', 1, True, 'C', 7, False, 'B', 2, False, 'C', 9, True]
```

See also `unflatten()`.

New in version 0.7.

petl.**unflatten**(**args*, ***kwargs*)

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

```
>>> from petl import unflatten, look
>>> input = ['A', 1, True, 'C', 7, False, 'B', 2, False, 'C', 9]
>>> table = unflatten(input, 3)
>>> look(table)
+------+------+-------+
| '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
... look(table1)
+---------+
| 'lines' |
+=========+
| 'A'     |
+---------+
| 1       |
+---------+
| True    |
+---------+
| 'C'     |
+---------+
| 7       |
+---------+
| False   |
+---------+
```

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

>>> table2 = unflatten(table1, 'lines', 3)
>>> look(table2)
+------+------+-------+
| 'f0' | 'f1' | 'f2'  |
+======+======+=======+
| 'A'  | 1    | True  |
+------+------+-------+
| 'C'  | 7    | False |
+------+------+-------+
| 'B'  | 2    | False |
+------+------+-------+
| 'C'  | 9    | None  |
+------+------+-------+
```

See also `flatten()`.

New in version 0.7.

petl.**annex**(**tables*, ***kwargs*)

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

```
>>> from petl import annex, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 9     |
+-------+-------+
| 'C'   | 2     |
+-------+-------+
| 'F'   | 1     |
+-------+-------+

>>> look(table2)
+-------+-------+
| 'foo' | 'baz' |
+=======+=======+
| 'B'   | 3     |
+-------+-------+
| 'D'   | 10    |
+-------+-------+

>>> table3 = annex(table1, table2)
>>> look(table3)
+-------+-------+-------+-------+
| 'foo' | 'bar' | 'foo' | 'baz' |
+=======+=======+=======+=======+
| 'A'   | 9     | 'B'   | 3     |
+-------+-------+-------+-------+
| 'C'   | 2     | 'D'   | 10    |
```

```
+-------+-------+-------+-------+
| 'F'   | 1     | None  | None  |
+-------+-------+-------+-------+
```

New in version 0.10.

petl.**fold**(*table*, *key*, *f*, *value=None*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)

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

```
>>> from petl import fold, look
>>> look(table1)
+------+---------+
| 'id' | 'count' |
+======+=========+
| 1    | 3       |
+------+---------+
| 1    | 5       |
+------+---------+
| 2    | 4       |
+------+---------+
| 2    | 8       |
+------+---------+

>>> import operator
>>> table2 = fold(table1, 'id', operator.add, 'count', presorted=True)
>>> look(table2)
+-------+---------+
| 'key' | 'value' |
+=======+=========+
| 1     | 8       |
+-------+---------+
| 2     | 12      |
+-------+---------+
```

See also `aggregate()`, `rowreduce()`.

New in version 0.10.

petl.**addrownumbers**(*table*, *start=1*, *step=1*)

Add a field of row numbers. E.g.:

```
>>> from petl import addrownumbers, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 9     |
+-------+-------+
| 'C'   | 2     |
+-------+-------+
| 'F'   | 1     |
+-------+-------+

>>> table2 = addrownumbers(table1)
>>> look(table2)
+-------+-------+-------+
| 'row' | 'foo' | 'bar' |
+=======+=======+=======+
| 1     | 'A'   | 9     |
+-------+-------+-------+
```

```
| 2     | 'C'   | 2     |
+-------+-------+-------+
| 3     | 'F'   | 1     |
+-------+-------+-------+
```

New in version 0.10.

petl.**addcolumn**(*table*, *field*, *col*, *index=None*, *missing=None*)
Add a column of data to the table. E.g.:

```
>>> from petl import addcolumn, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 1     |
+-------+-------+
| 'B'   | 2     |
+-------+-------+

>>> col = [True, False]
>>> table2 = addcolumn(table1, 'baz', col)
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'A'   | 1     | True  |
+-------+-------+-------+
| 'B'   | 2     | False |
+-------+-------+-------+
```

New in version 0.10.

petl.**addfield**(*table*, *field*, *value=None*, *index=None*)
Add a field with a fixed or calculated value. E.g.:

```
>>> from petl import addfield, look
>>> look(table1)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'M'   | 12    |
+-------+-------+
| 'F'   | 34    |
+-------+-------+
| '-'   | 56    |
+-------+-------+

>>> # using a fixed value
... table2 = addfield(table1, 'baz', 42)
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'M'   | 12    | 42    |
+-------+-------+-------+
| 'F'   | 34    | 42    |
+-------+-------+-------+
| '-'   | 56    | 42    |
```

```
+-------+-------+-------+

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

>>> # an expression string can also be used via expr
... from petl import expr
>>> table3 = addfield(table1, 'baz', expr('{bar} * 2'))
>>> look(table3)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'M'   | 12    | 24    |
+-------+-------+-------+
| 'F'   | 34    | 68    |
+-------+-------+-------+
| '-'   | 56    | 112   |
+-------+-------+-------+
```

Changed in version 0.10.

Renamed 'extend' to 'addfield'.

petl.**filldown**(*table*, **fields*, ***kwargs*)

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

```
>>> from petl import filldown, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 1     | 'a'   | None  |
+-------+-------+-------+
| 1     | None  | 0.23  |
+-------+-------+-------+
| 1     | 'b'   | None  |
+-------+-------+-------+
| 2     | None  | None  |
+-------+-------+-------+
| 2     | None  | 0.56  |
+-------+-------+-------+
| 2     | 'c'   | None  |
+-------+-------+-------+
| None  | 'c'   | 0.72  |
+-------+-------+-------+

>>> table2 = filldown(table1)
>>> look(table2)
+-------+-------+-------+
```

```
| '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 = filldown(table1, 'bar')
>>> look(table3)
+-------+-------+-------+
| '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 = filldown(table1, 'bar', 'baz')
>>> look(table4)
+-------+-------+-------+
| '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  |
+-------+-------+-------+
```

New in version 0.11.

petl.**fillright**(*table*, *missing=None*)

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

```
>>> from petl import fillright, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 1     | 'a'   | None  |
+-------+-------+-------+
| 1     | None  | 0.23  |
+-------+-------+-------+
| 1     | 'b'   | None  |
+-------+-------+-------+
| 2     | None  | None  |
+-------+-------+-------+
| 2     | None  | 0.56  |
+-------+-------+-------+
| 2     | 'c'   | None  |
+-------+-------+-------+
| None  | 'c'   | 0.72  |
+-------+-------+-------+

>>> table2 = fillright(table1)
>>> look(table2)
+-------+-------+-------+
| '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  |
+-------+-------+-------+
```

New in version 0.11.

petl.**fillleft**(*table*, *missing=None*)

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

```
>>> from petl import fillleft, look
>>> look(table1)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 1     | 'a'   | None  |
+-------+-------+-------+
| 1     | None  | 0.23  |
+-------+-------+-------+
| 1     | 'b'   | None  |
+-------+-------+-------+
```

```
| 2     | None  | None  |
+-------+-------+-------+
| None  | None  | 0.56  |
+-------+-------+-------+
| 2     | 'c'   | None  |
+-------+-------+-------+
| None  | 'c'   | 0.72  |
+-------+-------+-------+

>>> table2 = fillleft(table1)
>>> look(table2)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 1     | 'a'   | None  |
+-------+-------+-------+
| 1     | 0.23  | 0.23  |
+-------+-------+-------+
| 1     | 'b'   | None  |
+-------+-------+-------+
| 2     | None  | None  |
+-------+-------+-------+
| 0.56  | 0.56  | 0.56  |
+-------+-------+-------+
| 2     | 'c'   | None  |
+-------+-------+-------+
| 'c'   | 'c'   | 0.72  |
+-------+-------+-------+
```

New in version 0.11.

petl.**multirangeaggregate**(*table*, *keys*, *widths*, *aggregation*, *value=None*, *mins=None*, *maxs=None*)

Group rows at multiple levels then aggregate whole rows or specified values. E.g.:

```
>>> from petl import look, multirangeaggregate
>>> look(table1)
+-----+-----+-----+
| 'x' | 'y' | 'z' |
+=====+=====+=====+
| 1   | 3   | 9   |
+-----+-----+-----+
| 2   | 3   | 12  |
+-----+-----+-----+
| 4   | 2   | 17  |
+-----+-----+-----+
| 2   | 7   | 3   |
+-----+-----+-----+
| 1   | 6   | 1   |
+-----+-----+-----+

>>> table2 = multirangeaggregate(table1, keys=('x', 'y'), widths=(2, 2), aggregation=sum, mins=(
>>> look(table2)
+------------------+---------+
| 'key'            | 'value' |
+==================+=========+
| ((0, 2), (0, 2)) | 0       |
+------------------+---------+
| ((0, 2), (2, 4)) | 9       |
+------------------+---------+
```

```
| ((2, 4), (0, 2)) | 0       |
+------------------+---------+
| ((2, 4), (2, 4)) | 29      |
+------------------+---------+
```

New in version 0.12.

petl.**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., if the join key is present in the table:

```
>>> from petl import look, unjoin
>>> look(table1)
+-------+-------+----------+
| 'foo' | 'bar' | 'baz'    |
+=======+=======+==========+
| 'A'   | 1     | 'apple'  |
+-------+-------+----------+
| 'B'   | 1     | 'apple'  |
+-------+-------+----------+
| 'C'   | 2     | 'orange' |
+-------+-------+----------+

>>> table2, table3 = unjoin(table1, 'baz', key='bar')
>>> look(table2)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'A'   | 1     |
+-------+-------+
| 'B'   | 1     |
+-------+-------+
| 'C'   | 2     |
+-------+-------+

>>> look(table3)
+-------+----------+
| 'bar' | 'baz'    |
+=======+==========+
| 1     | 'apple'  |
+-------+----------+
| 2     | 'orange' |
+-------+----------+
```

An integer join key can also be reconstructed, e.g.:

```
>>> look(table4)
+-------+----------+
| 'foo' | 'bar'    |
+=======+==========+
| 'A'   | 'apple'  |
+-------+----------+
| 'B'   | 'apple'  |
+-------+----------+
| 'C'   | 'orange' |
+-------+----------+

>>> table5, table6 = unjoin(table4, 'bar')
```

```
>>> look(table5)
+-------+----------+
| 'foo' | 'bar_id' |
+=======+==========+
| 'A'   | 1        |
+-------+----------+
| 'B'   | 1        |
+-------+----------+
| 'C'   | 2        |
+-------+----------+

>>> look(table6)
+------+----------+
| 'id' | 'bar'    |
+======+==========+
| 1    | 'apple'  |
+------+----------+
| 2    | 'orange' |
+------+----------+
```

New in version 0.12.

petl.**distinct**(*table*, *presorted=False*, *buffersize=None*, *tempdir=None*, *cache=True*)
Return only distinct rows in the table. See also `duplicates()` and `unique()`.

New in version 0.12.

petl.**rowgroupmap**(*table*, *key*, *mapper*, *fields=None*, *missing=None*, *presorted=False*, *buffersize=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.

New in version 0.12.

petl.**groupcountdistinctvalues**(*table*, *key*, *value*)
Group by the *key* field then count the number of distinct values in the *value* field.

New in version 0.14.

petl.**groupselectfirst**(*table*, *key*)
Group by the *key* field then return the first row within each group.

New in version 0.14.

petl.**groupselectmin**(*table*, *key*, *value*)
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.

New in version 0.14.

petl.**groupselectmax**(*table*, *key*, *value*)
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 (maximum) value.

New in version 0.14.

CHAPTER 5

Load - writing tables to files and databases

The following functions write data from a table to 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.

`petl.`**`tocsv`**(*table*, *source=None*, *dialect=<class csv.excel at 0x2ad5050>*, ***kwargs*)

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

```
>>> from petl import tocsv, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> tocsv(table, 'test.csv')
>>> # look what it did
... from petl import fromcsv
>>> look(fromcsv('test.csv'))
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '2'   |
+-------+-------+
| 'c'   | '2'   |
+-------+-------+
```

The *filename* argument is the path of 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.

Supports transparent writing to .gz and .bz2 files.

petl.**appendcsv**(*table*, *source=None*, *dialect=<class csv.excel at 0x2ad5050>*, ***kwargs*)
Append data rows to an existing CSV file. E.g.:

```
>>> # look at an existing CSV file
... from petl import look, fromcsv
>>> testcsv = fromcsv('test.csv')
>>> look(testcsv)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '2'   |
+-------+-------+
| 'c'   | '2'   |
+-------+-------+

>>> # append some data
... look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'd'   | 7     |
+-------+-------+
| 'e'   | 42    |
+-------+-------+
| 'f'   | 12    |
+-------+-------+

>>> from petl import appendcsv
>>> appendcsv(table, 'test.csv')
>>> # look what it did
... look(testcsv)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '2'   |
+-------+-------+
| 'c'   | '2'   |
+-------+-------+
| 'd'   | '7'   |
+-------+-------+
| 'e'   | '42'  |
+-------+-------+
| 'f'   | '12'  |
+-------+-------+
```

The *filename* argument is the path of 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 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. See also the `cat()` function.

Supports transparent writing to `.gz` and `.bz2` files.

`petl.`**`totsv`**(*table*, *source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `tocsv()` but with different default dialect (tab delimited).

Supports transparent writing to `.gz` and `.bz2` files.

New in version 0.9.

`petl.`**`appendtsv`**(*table*, *source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `appendcsv()` but with different default dialect (tab delimited).

Supports transparent writing to `.gz` and `.bz2` files.

New in version 0.9.

`petl.`**`toucsv`**(*table*, *source=None*, *dialect=<class csv.excel at 0x2ad5050>*, *encoding='utf-8'*, ***kwargs*)
Write the table to a CSV file via the given encoding. Like `tocsv()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

`petl.`**`appenducsv`**(*table*, *source=None*, *dialect=<class csv.excel at 0x2ad5050>*, *encoding='utf-8'*, ***kwargs*)
Append the table to a CSV file via the given encoding. Like `appendcsv()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

`petl.`**`toutsv`**(*table*, *source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `toucsv()` but with different default dialect (tab delimited).

New in version 0.19.

`petl.`**`appendutsv`**(*table*, *source=None*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Convenience function, as `appenducsv()` but with different default dialect (tab delimited).

New in version 0.19.

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

```
>>> from petl import topickle, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> topickle(table, 'test.dat')
>>> # look what it did
... from petl import frompickle
>>> look(frompickle('test.dat'))
+-------+-------+
| '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.

Supports transparent writing to `.gz` and `.bz2` files.

petl.**appendpickle**(*table*, *source=None*, *protocol=-1*)

Append data to an existing pickle file. E.g.:

```
>>> from petl import look, frompickle
>>> # inspect an existing pickle file
... testdat = frompickle('test.dat')
>>> look(testdat)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> # append some data
... from petl import appendpickle
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'd'   | 7     |
+-------+-------+
| 'e'   | 42    |
+-------+-------+
| 'f'   | 12    |
+-------+-------+

>>> appendpickle(table, 'test.dat')
>>> # look what it did
... look(testdat)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+
| 'd'   | 7     |
+-------+-------+
| 'e'   | 42    |
+-------+-------+
```

```
| 'f'   | 12    |
+-------+-------+
```

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. See also the `cat()` function.

Supports transparent writing to `.gz` and `.bz2` files.

`petl.`**`tosqlite3`**(*table*, *filename_or_connection*, *tablename*, *create=False*, *commit=True*)

Load data into a table in an `sqlite3` database. Note that if the database table exists, it will be truncated, i.e., all existing rows will be deleted prior to inserting the new data. E.g.:

```
>>> from petl import tosqlite3, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> # by default, if the table does not already exist, it will be created
... tosqlite3(table, 'test.db', 'foobar')
>>> # look what it did
... from petl import fromsqlite3
>>> look(fromsqlite3('test.db', 'select * from foobar'))
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| u'a'  | 1     |
+-------+-------+
| u'b'  | 2     |
+-------+-------+
| u'c'  | 2     |
+-------+-------+
```

If the table does not exist and `create=True` then a table will be created using the field names in the table header. However, note that no type specifications will be included in the table creation statement and so column type affinities may be inappropriate.

Changed in version 0.10.2.

Either a database file name or a connection object can be given as the second argument.

Changed in version 0.21.

Default value for `create` argument changed to `False`.

`petl.`**`appendsqlite3`**(*table*, *filename_or_connection*, *tablename*, *commit=True*)

Load data into an existing table in an `sqlite3` database. Note that the database table will be appended, i.e., the new data will be inserted into the table, and any existing rows will remain. E.g.:

```
>>> from petl import appendsqlite3, look
>>> look(moredata)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'd'   | 7     |
```

```
+-------+-------+
| 'e'   | 9     |
+-------+-------+
| 'f'   | 1     |
+-------+-------+

>>> appendsqlite3(moredata, 'test.db', 'foobar')
>>> # look what it did
... from petl import look, fromsqlite3
>>> look(fromsqlite3('test.db', 'select * from foobar'))
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| u'a'  | 1     |
+-------+-------+
| u'b'  | 2     |
+-------+-------+
| u'c'  | 2     |
+-------+-------+
| u'd'  | 7     |
+-------+-------+
| u'e'  | 9     |
+-------+-------+
| u'f'  | 1     |
+-------+-------+
```

Changed in version 0.10.2.

Either a database file name or a connection object can be given as the second argument.

petl.**todb**(*table*, *dbo*, *tablename*, *schema=None*, *commit=True*)

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.:

```
>>> from petl import look, todb
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+
```

... using `sqlite3`:

```
>>> import sqlite3
>>> connection = sqlite3.connect('test.db')
>>> # assuming table "foobar" already exists in the database
... todb(table, connection, 'foobar')
```

... using `psycopg2`:

```
>>> import psycopg2
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> # assuming table "foobar" already exists in the database
... todb(table, connection, 'foobar')
```

... using `MySQLdb`:

```
>>> import MySQLdb
>>> connection = MySQLdb.connect(passwd="moonpie", db="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
... 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.

Changed in version 0.10.2.

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

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

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

Load data into an existing database table via a DB-API 2.0 connection or cursor. Note that the database table will be appended, i.e., the new data will be inserted into the table, and any existing rows will remain. E.g.:

```
>>> from petl import look, appenddb
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+
```

... using `sqlite3`:

```
>>> import sqlite3
>>> connection = sqlite3.connect('test.db')
>>> # assuming table "foobar" already exists in the database
... appenddb(table, connection, 'foobar')
```

... using `psycopg2`:

```
>>> import psycopg2
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> # assuming table "foobar" already exists in the database
... appenddb(table, connection, 'foobar')
```

... using `MySQLdb`:

```
>>> import MySQLdb
>>> connection = MySQLdb.connect(passwd="moonpie", db="thangs")
>>> # tell MySQL to use standard quote character
... connection.cursor().execute('SET SQL_MODE=ANSI_QUOTES')
>>> # load data, appending rows to table "foobar"
... appenddb(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.

Changed in version 0.10.2.

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

```
>>> import psycopg2
>>> connection = psycopg2.connect("dbname=test user=postgres")
>>> cursor = connection.cursor()
>>> appenddb(table, cursor, 'foobar')
```

petl.**totext**(*table*, *source=None*, *template=None*, *prologue=None*, *epilogue=None*)

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

```
>>> from petl import totext, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> prologue = """{| class="wikitable"
... |-
... ! foo
... ! bar
... """
>>> template = """|-
... | {foo}
... | {bar}
... """
>>> epilogue = "|}"
>>> totext(table, 'test.txt', template, prologue, epilogue)
>>>
>>> # see what we did
... with open('test.txt') as f:
...     print f.read()
...
{| class="wikitable"
|-
! foo
! bar
|-
| a
| 1
|-
| b
| 2
|-
| c
| 2
|}
```

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

Supports transparent writing to `.gz` and `.bz2` files.

petl.**appendtext**(*table*, *source=None*, *template=None*, *prologue=None*, *epilogue=None*)
Append the table to a text file.

New in version 0.19.

petl.**toutext**(*table*, *source=None*, *encoding='utf-8'*, *template=None*, *prologue=None*, *epilogue=None*)
Write the table to a text file via the given encoding. Like `totext()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

petl.**appendutext**(*table*, *source=None*, *encoding='utf-8'*, *template=None*, *prologue=None*, *epilogue=None*)
Append the table to a text file via the given encoding. Like `appendtext()` but accepts an additional `encoding` argument which should be one of the Python supported encodings. See also `codecs`.

New in version 0.19.

petl.**tojson**(*table*, *source=None*, *prefix=None*, *suffix=None*, **args*, ***kwargs*)
Write a table in JSON format, with rows output as JSON objects. E.g.:

```
>>> from petl import tojson, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
+-------+-------+

>>> tojson(table, 'example.json')
>>> # check what it did
... with open('example.json') as f:
...     print f.read()
...
[{"foo": "a", "bar": 1}, {"foo": "b", "bar": 2}, {"foo": "c", "bar": 2}]
```

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

Supports transparent writing to `.gz` and `.bz2` files.

New in version 0.5.

petl.**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.:

```
>>> from petl import tojsonarrays, look
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
| 'c'   | 2     |
```

```
+-------+-------+

>>> tojsonarrays(table, 'example.json')
>>> # check what it did
... with open('example.json') as f:
...     print f.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.

Supports transparent writing to `.gz` and `.bz2` files.

New in version 0.11.

CHAPTER 6

Utility functions

petl.**header**(*table*)
Return the header row for the given table. E.g.:

```
>>> from petl import header
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> header(table)
['foo', 'bar']
```

See also `fieldnames()`.

petl.**data**(*table*, **sliceargs*)
Return a container supporting iteration over data rows in a given table. I.e., like `iterdata()` only a container is returned so you can iterate over it multiple times.

Changed in version 0.10.

Now returns a container, previously returned an iterator. See also `iterdata()`.

petl.**iterdata**(*table*, **sliceargs*)
Return an iterator over the data rows for the given table. E.g.:

```
>>> from petl import data
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> it = iterdata(table)
>>> it.next()
['a', 1]
>>> it.next()
['b', 2]
```

Changed in version 0.3.

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

Changed in version 0.10.

Renamed from "data".

petl.**dataslice**(*table*, **args*)
Deprecated since version 0.3.

Use `data()` instead, it supports slice arguments.

petl.**fieldnames**(*table*)
Return the string values of all fields for the given table. If the fields are strings, then this function is equivalent to `header()`, i.e.:

```
>>> from petl import header, fieldnames
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> header(table)
['foo', 'bar']
>>> fieldnames(table)
['foo', 'bar']
>>> header(table) == fieldnames(table)
True
```

Allows for custom field objects, e.g.:

```
>>> class CustomField(object):
...     def __init__(self, id, description):
...         self.id = id
...         self.description = description
...     def __str__(self):
...         return self.id
...     def __repr__(self):
...         return 'CustomField(%r, %r)' % (self.id, self.description)
...
>>> table = [[CustomField('foo', 'Get some foo.'), CustomField('bar', 'A lot of bar.')],
...           ['a', 1],
...           ['b', 2]]
>>> header(table)
[CustomField('foo', 'Get some foo.'), CustomField('bar', 'A lot of bar.')]
>>> fieldnames(table)
['foo', 'bar']
```

petl.**nrows**(*table*)

Count the number of data rows in a table. E.g.:

```
>>> from petl import nrows
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> nrows(table)
2
```

Changed in version 0.10.

Renamed from 'rowcount' to 'nrows'.

petl.**look**(*table*, **sliceargs*, ***kwargs*)

Format a portion of the table as text for inspection in an interactive session. E.g.:

```
>>> from petl import look
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 2     |
+-------+-------+
```

Any irregularities in the length of header and/or data rows will appear as blank cells, e.g.:

```
>>> table = [['foo', 'bar'], ['a'], ['b', 2, True]]
>>> look(table)
+-------+-------+------+
| 'foo' | 'bar' |      |
```

```
+=======+=======+======+
| 'a'   |       |      |
+-------+-------+------+
| 'b'   | 2     | True |
+-------+-------+------+
```

Changed in version 0.3.

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

Changed in version 0.8.

The properties *n* and *p* can be used to look at the next and previous rows respectively. I.e., try `>>> look(table)` then `>>> _.n` then `>>> _.p`.

Changed in version 0.13.

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, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> look(table, style='simple')
=====  =====
'foo'  'bar'
=====  =====
'a'        1
'b'        2
=====  =====

>>> look(table, style='minimal')
'foo'  'bar'
'a'        1
'b'        2
```

The default style can also be changed, e.g.:

```
>>> look.default_style = 'simple'
>>> look(table)
=====  =====
'foo'  'bar'
=====  =====
'a'        1
'b'        2
=====  =====

>>> look.default_style = 'grid'
>>> look(table)
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   |     1 |
+-------+-------+
| 'b'   |     2 |
+-------+-------+
```

See also `lookall()` and `see()`.

petl.**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.

petl.**see**(*table*, **sliceargs*)
Format a portion of a table as text in a column-oriented layout for inspection in an interactive session. E.g.:

```
>>> from petl import see
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> see(table)
'foo': 'a', 'b'
'bar': 1, 2
```

Useful for tables with a larger number of fields.

Changed in version 0.3.

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

petl.**values**(*table*, *field*, **sliceargs*, ***kwargs*)
Return a container supporting iteration over values in a given field or fields. I.e., like `itervalues()` only a container is returned so you can iterate over it multiple times.

Changed in version 0.7.

Now returns a container, previously returned an iterator. See also `itervalues()`.

petl.**itervalues**(*table*, *field*, **sliceargs*, ***kwargs*)
Return an iterator over values in a given field or fields. E.g.:

```
>>> from petl import itervalues
>>> table = [['foo', 'bar'], ['a', True], ['b'], ['b', True], ['c', False]]
>>> foo = itervalues(table, 'foo')
>>> foo.next()
'a'
>>> foo.next()
'b'
>>> foo.next()
'b'
>>> foo.next()
'c'
>>> foo.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

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

If rows are uneven, the value of the keyword argument *missing* is returned.

More than one field can be selected, e.g.:

```
>>> table = [['foo', 'bar', 'baz'],
...          [1, 'a', True],
...          [2, 'bb', True],
...          [3, 'd', False]]
>>> foobaz = itervalues(table, ('foo', 'baz'))
>>> foobaz.next()
(1, True)
>>> foobaz.next()
(2, True)
>>> foobaz.next()
(3, False)
>>> foobaz.next()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
StopIteration
```

Changed in version 0.3.

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

Changed in version 0.7.

In previous releases this function was known as 'values'. Also in this release the behaviour with short rows is changed. Now for any value missing due to a short row, `None` is returned by default, or whatever is given by the *missing* keyword argument.

`petl.`**`valueset`**(*table*, *field*, *missing=None*)

Deprecated since version 0.3.

Use `set(values(table, *fields))` instead, see also `values()`.

`petl.`**`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.:

```
>>> from petl import valuecount
>>> table = (('foo', 'bar'), ('a', 1), ('b', 2), ('b', 7))
>>> n, f = valuecount(table, 'foo', 'b')
>>> n
2
>>> f
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.`**`valuecounts`**(*table*, **fields*, ***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.:

```
>>> from petl import valuecounts, look
>>> table = [['foo', 'bar'], ['a', True], ['b'], ['b', True], ['c', False]]
>>> look(valuecounts(table, 'foo'))
+---------+---------+-------------+
| 'value' | 'count' | 'frequency' |
+=========+=========+=============+
| 'b'     | 2       | 0.5         |
+---------+---------+-------------+
| 'a'     | 1       | 0.25        |
+---------+---------+-------------+
| 'c'     | 1       | 0.25        |
+---------+---------+-------------+

>>> look(valuecounts(table, 'bar'))
+---------+---------+--------------------+
| 'value' | 'count' | 'frequency'        |
+=========+=========+====================+
| True    | 2       | 0.6666666666666666 |
+---------+---------+--------------------+
| False   | 1       | 0.3333333333333333 |
+---------+---------+--------------------+
```

If more than one field is given, a report of value counts for each field is given, e.g.:

```
>>> look(valuecounts(table, 'foo', 'bar'))
+---------+---------+---------+-------------+
| 'field' | 'value' | 'count' | 'frequency' |
+=========+=========+=========+=============+
| 'foo'   | 'b'     |       2 |         0.5 |
+---------+---------+---------+-------------+
| 'foo'   | 'a'     |       1 |        0.25 |
+---------+---------+---------+-------------+
| 'foo'   | 'c'     |       1 |        0.25 |
+---------+---------+---------+-------------+
| 'bar'   |    True |       2 |         0.5 |
+---------+---------+---------+-------------+
| 'bar'   | None    |       1 |        0.25 |
+---------+---------+---------+-------------+
| 'bar'   |   False |       1 |        0.25 |
+---------+---------+---------+-------------+
```

If rows are short, the value of the keyword argument *missing* is counted.

petl.**valuecounter**(*table*, *field*, *missing=None*)

Find distinct values for the given field and count the number of occurrences. Returns a `dict` mapping values to counts. E.g.:

```
>>> from petl import valuecounter
>>> table = [['foo', 'bar'], ['a', True], ['b'], ['b', True], ['c', False]]
>>> c = valuecounter(table, 'foo')
>>> c['a']
1
>>> c['b']
2
>>> c['c']
1
>>> c
Counter({'b': 2, 'a': 1, 'c': 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.**dicts**(*table*, **sliceargs*, ***kwargs*)

Return a container supporting iteration over rows as dicts. I.e., like `iterdicts()` only a container is returned so you can iterate over it multiple times.

New in version 0.15.

petl.**iterdicts**(*table*, **sliceargs*, ***kwargs*)

Return an iterator over the data in the table, yielding each row as a dictionary of values indexed by field name. E.g.:

```
>>> from petl import dicts
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2]]
>>> it = dicts(table)
>>> it.next()
{'foo': 'a', 'bar': 1}
>>> it.next()
{'foo': 'b', 'bar': 2}
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Short rows are padded, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b']]
>>> it = dicts(table)
>>> it.next()
{'foo': 'a', 'bar': 1}
>>> it.next()
{'foo': 'b', 'bar': None}
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

New in version 0.15.

petl.**namedtuples**(*table*, **sliceargs*, ***kwargs*)
View the table as a container of named tuples. I.e., like `iternamedtuples()` only a container is returned so you can iterate over it multiple times.

New in version 0.15.

petl.**iternamedtuples**(*table*, **sliceargs*, ***kwargs*)
Return an iterator over the data in the table, yielding each row as a named tuple.

New in version 0.15.

petl.**records**(*table*, **sliceargs*, ***kwargs*)
Return a container supporting iteration over rows as records. I.e., like `iterrecords()` only a container is returned so you can iterate over it multiple times. See also `dicts()`.

Changed in version 0.15.

Previously returned dicts, now returns hybrid objects which behave like tuples/dicts/namedtuples.

petl.**iterrecords**(*table*, **sliceargs*, ***kwargs*)
Return an iterator over the data in the table, where rows support value access by index or field name. See also `iterdicts()`.

Changed in version 0.15.

Previously returned dicts, now returns hybrid objects which behave like tuples/dicts/namedtuples.

petl.**columns**(*table*, *missing=None*)
Construct a `dict` mapping field names to lists of values. E.g.:

```
>>> from petl import columns
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> cols = columns(table)
>>> cols['foo']
['a', 'b', 'b']
>>> cols['bar']
[1, 2, 3]
```

See also `facetcolumns()`.

petl.**facetcolumns**(*table*, *key*, *missing=None*)
Like `columns()` but stratified by values of the given key field. E.g.:

```
>>> from petl import facetcolumns
>>> table = [['foo', 'bar', 'baz'],
...          ['a', 1, True],
...          ['b', 2, True],
...          ['b', 3]]
```

```
>>> fc = facetcolumns(table, 'foo')
>>> fc['a']
{'baz': [True], 'foo': ['a'], 'bar': [1]}
>>> fc['b']
{'baz': [True, None], 'foo': ['b', 'b'], 'bar': [2, 3]}
>>> fc['c']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'c'
```

New in version 0.8.

petl.**isunique**(*table*, *field*)

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

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

Changed in version 0.10.

Renamed from "unique". See also `petl.unique()`.

petl.**isordered**(*table*, *key=None*, *reverse=False*, *strict=False*)

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

```
>>> from petl import isordered, look
>>> look(table)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'a'   | 1     | True  |
+-------+-------+-------+
| 'b'   | 3     | True  |
+-------+-------+-------+
| 'b'   | 2     |       |
+-------+-------+-------+

>>> isordered(table, key='foo')
True
>>> isordered(table, key='foo', strict=True)
False
>>> isordered(table, key='foo', reverse=True)
False
```

New in version 0.10.

petl.**limits**(*table*, *field*)

Find minimum and maximum values under the given field. E.g.:

```
>>> from petl import limits
>>> t1 = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> minv, maxv = limits(t1, 'bar')
>>> minv
1
```

```
>>> maxv
3
```

The *field* argument can be a field name or index (starting from zero).

petl.**stats**(*table*, *field*)
Calculate basic descriptive statistics on a given field. E.g.:

```
>>> from petl import stats
>>> 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]]
>>> stats(table, 'bar')
{'count': 3, 'errors': 2, 'min': 1.0, 'max': 3.0, 'sum': 6.0, 'mean': 2.0}
```

The *field* argument can be a field name or index (starting from zero).

petl.**lenstats**(*table*, *field*)
Convenience function to report statistics on value lengths under the given field. E.g.:

```
>>> from petl import lenstats
>>> table1 = [['foo', 'bar'],
...           [1, 'a'],
...           [2, 'aaa'],
...           [3, 'aa'],
...           [4, 'aaa'],
...           [5, 'aaaaaaaaaa']]
>>> lenstats(table1, 'bar')
{'count': 5, 'errors': 0, 'min': 1.0, 'max': 11.0, 'sum': 20.0, 'mean': 4.0}
```

petl.**stringpatterns**(*table*, *field*)
Profile string patterns in the given field, returning a table of patterns, counts and frequencies. E.g.:

```
>>> from petl import stringpatterns, look
>>> 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']]
>>> foopats = stringpatterns(table, 'foo')
>>> look(foopats)
+------------+---------+---------------------+
| 'pattern'  | 'count' | 'frequency'         |
+============+=========+=====================+
| 'Aa. Aaa'  | 3       | 0.5                 |
+------------+---------+---------------------+
| 'Aaa. Aaa' | 2       | 0.3333333333333333  |
+------------+---------+---------------------+
| 'Aa. Aaaa' | 1       | 0.16666666666666666 |
+------------+---------+---------------------+

>>> barpats = stringpatterns(table, 'bar')
>>> look(barpats)
+---------------+---------+---------------------+
| 'pattern'     | 'count' | 'frequency'         |
```

```
+===============+=========+=====================+
| '999-9999'    | 3       | 0.5                 |
+---------------+---------+---------------------+
| '999 9999'    | 2       | 0.3333333333333333  |
+---------------+---------+---------------------+
| '999-9999-AA' | 1       | 0.16666666666666666 |
+---------------+---------+---------------------+
```

New in version 0.5.

petl.**stringpatterncounter**(*table*, *field*)
Profile string patterns in the given field, returning a `dict` mapping patterns to counts.

New in version 0.5.

petl.**rowlengths**(*table*)
Report on row lengths found in the table. E.g.:

```
>>> from petl import look, rowlengths
>>> 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]]
>>> look(rowlengths(table))
+----------+---------+
| 'length' | 'count' |
+==========+=========+
| 3        | 3       |
+----------+---------+
| 2        | 2       |
+----------+---------+
| 4        | 1       |
+----------+---------+
```

Useful for finding potential problems in data files.

petl.**typecounts**(*table*, *field*, ***kwargs*)
Count the number of values found for each Python type and return a table mapping class names to counts and frequencies. E.g.:

```
>>> from petl import look, typecounts
>>> 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]]
>>> look(typecounts(table, 'foo'))
+-----------+---------+-------------+
| 'type'    | 'count' | 'frequency' |
+===========+=========+=============+
| 'str'     | 4       | 0.8         |
+-----------+---------+-------------+
| 'unicode' | 1       | 0.2         |
+-----------+---------+-------------+

>>> look(typecounts(table, 'bar'))
```

```
+-----------+---------+-------------+
| 'type'    | 'count' | 'frequency' |
+===========+=========+=============+
| 'unicode' | 3       | 0.6         |
+-----------+---------+-------------+
| 'int'     | 2       | 0.4         |
+-----------+---------+-------------+

>>> look(typecounts(table, 'baz'))
+-----------+---------+-------------+
| 'type'    | 'count' | 'frequency' |
+===========+=========+=============+
| 'int'     | 1       | 0.25        |
+-----------+---------+-------------+
| 'float'   | 1       | 0.25        |
+-----------+---------+-------------+
| 'unicode' | 1       | 0.25        |
+-----------+---------+-------------+
| 'str'     | 1       | 0.25        |
+-----------+---------+-------------+
```

The *field* argument can be a field name or index (starting from zero).

Changed in version 0.6.

Added frequency.

petl.**typecounter**(*table*, *field*)

Count the number of values found for each Python type. E.g.:

```
>>> from petl import typecounter
>>> 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]]
>>> typecounter(table, 'foo')
Counter({'str': 4, 'unicode': 1})
>>> typecounter(table, 'bar')
Counter({'unicode': 3, 'int': 2})
>>> typecounter(table, 'baz')
Counter({'int': 1, 'float': 1, 'unicode': 1, 'str': 1})
```

The *field* argument can be a field name or index (starting from zero).

petl.**typeset**(*table*, *field*)

Return a set containing all Python types found for values in the given field. E.g.:

```
>>> from petl import typeset
>>> 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]]
>>> typeset(table, 'foo')
set([<type 'str'>, <type 'unicode'>])
>>> typeset(table, 'bar')
set([<type 'int'>, <type 'unicode'>])
```

```
>>> typeset(table, 'baz')
set([<type 'float'>, <type 'str'>])
```

The *field* argument can be a field name or index (starting from zero).

petl.**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.:

```
>>> from petl import look, parsecounts
>>> 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]]
>>> look(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).

petl.**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.:

```
>>> from petl import parsecounter
>>> 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 = 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.**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)
>>> isodate('2002-02-30')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1032, in parser
    return parser
```

```
  File "/usr/lib/python2.7/_strptime.py", line 440, in _strptime
    datetime_date(year, 1, 1).toordinal() + 1
ValueError: day is out of range for month
```

Can be used with `parsecounts()`, e.g.:

```
>>> from petl import look, parsecounts, dateparser
>>> table = [['when', 'who'],
...          ['2002-12-25', 'Alex'],
...          ['2004-09-12', 'Gloria'],
...          ['2002-13-25', 'Marty'],
...          ['2002-02-30', 'Melman']]
>>> parsers={'date': dateparser('%Y-%m-%d')}
>>> look(parsecounts(table, 'when', parsers))
+--------+---------+----------+
| 'type' | 'count' | 'errors' |
+========+=========+==========+
| 'date' | 2       | 2        |
+--------+---------+----------+
```

Changed in version 0.6.

Added *strict* keyword argument. If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised. Allows for, e.g., incremental parsing of mixed format fields.

`petl.`**`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)
>>> isotime('12:00:99')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1046, in parser

  File "/usr/lib/python2.7/_strptime.py", line 328, in _strptime
    data_string[found.end():])
ValueError: unconverted data remains: 9
>>> isotime('25:00:00')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1046, in parser

  File "/usr/lib/python2.7/_strptime.py", line 325, in _strptime
    (data_string, format))
ValueError: time data '25:00:00' does not match format '%H:%M:%S'
```

Can be used with `parsecounts()`, e.g.:

```
>>> from petl import look, parsecounts, timeparser
>>> table = [['when', 'who'],
...          ['00:00:00', 'Alex'],
...          ['12:02:45', 'Gloria'],
...          ['25:01:01', 'Marty'],
...          ['09:70:00', 'Melman']]
>>> parsers={'time': timeparser('%H:%M:%S')}
```

```
>>> look(parsecounts(table, 'when', parsers))
+--------+---------+----------+
| 'type' | 'count' | 'errors' |
+========+=========+==========+
| 'time' | 2       | 2        |
+--------+---------+----------+
```

Changed in version 0.6.

Added *strict* keyword argument. If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised. Allows for, e.g., incremental parsing of mixed format fields.

`petl.`**`datetimeparser`**(*fmt*, *strict=True*)

Return a function to parse strings as `datetime.datetime` objects using a given format. E.g.:

```
>>> from petl import datetimeparser
>>> isodatetime = datetimeparser('%Y-%m-%dT%H:%M:%S')
>>> isodatetime('2002-12-25T00:00:00')
datetime.datetime(2002, 12, 25, 0, 0)
>>> isodatetime('2002-12-25T00:00:99')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1018, in parser
    return datetime.strptime(value.strip(), format)
  File "/usr/lib/python2.7/_strptime.py", line 328, in _strptime
    data_string[found.end():])
ValueError: unconverted data remains: 9
```

Can be used with `parsecounts()`, e.g.:

```
>>> from petl import look, parsecounts, datetimeparser
>>> table = [['when', 'who'],
...          ['2002-12-25T00:00:00', 'Alex'],
...          ['2004-09-12T01:10:11', 'Gloria'],
...          ['2002-13-25T00:00:00', 'Marty'],
...          ['2002-02-30T07:09:00', 'Melman']]
>>> parsers={'datetime': datetimeparser('%Y-%m-%dT%H:%M:%S')}
>>> look(parsecounts(table, 'when', parsers))
+------------+---------+----------+
| 'type'     | 'count' | 'errors' |
+============+=========+==========+
| 'datetime' | 2       | 2        |
+------------+---------+----------+
```

Changed in version 0.6.

Added *strict* keyword argument. If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised. Allows for, e.g., incremental parsing of mixed format fields.

`petl.`**`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('Y')
True
>>> mybool('yes')
```

```
True
>>> mybool('No')
False
>>> mybool('nO')
False
>>> mybool('true')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1175, in parser
    raise ValueError('value is not one of recognised boolean strings: %r' % value)
ValueError: value is not one of recognised boolean strings: 'true'
>>> mybool('foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 1175, in parser
    raise ValueError('value is not one of recognised boolean strings: %r' % value)
ValueError: value is not one of recognised boolean strings: 'foo'
```

Can be used with `parsecounts()`, e.g.:

```
>>> from petl import look, parsecounts, boolparser
>>> table = [['who', 'vote'],
...          ['Alex', 'yes'],
...          ['Gloria', 'N'],
...          ['Marty', 'hmmm'],
...          ['Melman', 'nope']]
>>> mybool = boolparser(true_strings=['yes', 'y'], false_strings=['no', 'n'])
>>> parsers = {'bool': mybool}
>>> look(parsecounts(table, 'vote', parsers))
+--------+---------+----------+
| 'type' | 'count' | 'errors' |
+========+=========+==========+
| 'bool' | 2       | 2        |
+--------+---------+----------+
```

Changed in version 0.6.

Added *strict* keyword argument. If `strict=False` then if an error occurs when parsing, the original value will be returned as-is, and no error will be raised. Allows for, e.g., incremental parsing of mixed format fields.

`petl.`**`parsenumber`**(*v*, *strict=False*)

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.

New in version 0.4.

Changed in version 0.7: Set `strict=True` to get an exception if parsing fails.

`petl.`**`lookup`**(*table*, *keyspec*, *valuespec=None*, *dictionary=None*)

Load a dictionary with data from the given table. E.g.:

```
>>> from petl import lookup
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = lookup(table, 'foo', 'bar')
>>> lkp['a']
[1]
>>> lkp['b']
[2, 3]
```

If no *valuespec* argument is given, defaults to the whole row (as a tuple), e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = lookup(table, 'foo')
>>> lkp['a']
[('a', 1)]
>>> lkp['b']
[('b', 2), ('b', 3)]
```

Compound keys are supported, e.g.:

```
>>> t2 = [['foo', 'bar', 'baz'],
...       ['a', 1, True],
...       ['b', 2, False],
...       ['b', 3, True],
...       ['b', 3, False]]
>>> lkp = lookup(t2, ('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, e.g.:

```
>>> import shelve
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = shelve.open('mylookup.dat')
>>> lkp = lookup(table, 'foo', 'bar', lkp)
>>> lkp.close()
>>> exit()
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shelve
>>> lkp = shelve.open('mylookup.dat')
>>> lkp['a']
[1]
>>> lkp['b']
[2, 3]
```

petl.**lookupone**(*table*, *keyspec*, *valuespec=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.:

```
>>> from petl import lookupone
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['c', 2]]
>>> lkp = lookupone(table, 'foo', 'bar')
>>> lkp['a']
1
>>> lkp['b']
2
>>> lkp['c']
2
```

If the specified key is not unique and strict=False (default), the first value wins, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = lookupone(table, 'foo', 'bar', strict=False)
```

```
>>> lkp['a']
1
>>> lkp['b']
2
```

If the specified key is not unique and strict=True, will raise DuplicateKeyError, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = lookupone(table, 'foo', strict=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 451, in lookupone
petl.util.DuplicateKeyError
```

Compound keys are supported, e.g.:

```
>>> t2 = [['foo', 'bar', 'baz'],
...       ['a', 1, True],
...       ['b', 2, False],
...       ['b', 3, True]]
>>> lkp = lookupone(t2, ('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, e.g.:

```
>>> from petl import lookupone
>>> import shelve
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['c', 2]]
>>> lkp = shelve.open('mylookupone.dat')
>>> lkp = lookupone(table, 'foo', 'bar', dictionary=lkp)
>>> lkp.close()
>>> exit()
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shelve
>>> lkp = shelve.open('mylookupone.dat')
>>> lkp['a']
1
>>> lkp['b']
2
>>> lkp['c']
2
```

Changed in version 0.11.

Changed so that strict=False is default and first value wins.

petl.**dictlookup**(*table*, *keyspec*, *dictionary=None*)

Load a dictionary with data from the given table, mapping to dicts. E.g.:

```
>>> from petl import dictlookup
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
```

```
>>> lkp = dictlookup(table, 'foo')
>>> lkp['a']
[{'foo': 'a', 'bar': 1}]
>>> lkp['b']
[{'foo': 'b', 'bar': 2}, {'foo': 'b', 'bar': 3}]
```

Compound keys are supported, e.g.:

```
>>> t2 = [['foo', 'bar', 'baz'],
...       ['a', 1, True],
...       ['b', 2, False],
...       ['b', 3, True],
...       ['b', 3, False]]
>>> lkp = dictlookup(t2, ('foo', 'bar'))
>>> lkp[('a', 1)]
[{'baz': True, 'foo': 'a', 'bar': 1}]
>>> lkp[('b', 2)]
[{'baz': False, 'foo': 'b', 'bar': 2}]
>>> lkp[('b', 3)]
[{'baz': True, 'foo': 'b', 'bar': 3}, {'baz': False, 'foo': 'b', 'bar': 3}]
```

Data can be loaded into an existing dictionary-like object, including persistent dictionaries created via the `shelve` module, e.g.:

```
>>> import shelve
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = shelve.open('mydictlookup.dat')
>>> lkp = dictlookup(table, 'foo', dictionary=lkp)
>>> lkp.close()
>>> exit()
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shelve
>>> lkp = shelve.open('mydictlookup.dat')
>>> lkp['a']
[{'foo': 'a', 'bar': 1}]
>>> lkp['b']
[{'foo': 'b', 'bar': 2}, {'foo': 'b', 'bar': 3}]
```

Changed in version 0.15.

Renamed from *recordlookup*.

petl.**dictlookupone**(*table*, *keyspec*, *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.:

```
>>> from petl import dictlookupone
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['c', 2]]
>>> lkp = dictlookupone(table, 'foo')
>>> lkp['a']
{'foo': 'a', 'bar': 1}
>>> lkp['b']
{'foo': 'b', 'bar': 2}
>>> lkp['c']
{'foo': 'c', 'bar': 2}
```

If the specified key is not unique and strict=False (default), the first dict wins, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = dictlookupone(table, '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, e.g.:

```
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['b', 3]]
>>> lkp = dictlookupone(table, 'foo', strict=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "petl/util.py", line 451, in lookupone
petl.util.DuplicateKeyError
```

Compound keys are supported, e.g.:

```
>>> t2 = [['foo', 'bar', 'baz'],
...       ['a', 1, True],
...       ['b', 2, False],
...       ['b', 3, True]]
>>> lkp = dictlookupone(t2, ('foo', 'bar'), strict=False)
>>> lkp[('a', 1)]
{'baz': True, 'foo': 'a', 'bar': 1}
>>> lkp[('b', 2)]
{'baz': False, 'foo': 'b', 'bar': 2}
>>> lkp[('b', 3)]
{'baz': True, 'foo': 'b', 'bar': 3}
```

Data can be loaded into an existing dictionary-like object, including persistent dictionaries created via the `shelve` module, e.g.:

```
>>> import shelve
>>> lkp = shelve.open('mydictlookupone.dat')
>>> table = [['foo', 'bar'], ['a', 1], ['b', 2], ['c', 2]]
>>> lkp = dictlookupone(table, 'foo', dictionary=lkp)
>>> lkp.close()
>>> exit()
$ python
Python 2.7.1+ (r271:86832, Apr 11 2011, 18:05:24)
[GCC 4.5.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import shelve
>>> lkp = shelve.open('mydictlookupone.dat')
>>> lkp['a']
{'foo': 'a', 'bar': 1}
>>> lkp['b']
{'foo': 'b', 'bar': 2}
>>> lkp['c']
{'foo': 'c', 'bar': 2}
```

Changed in version 0.11.

Changed so that strict=False is default and first value wins.

Changed in version 0.15.

Renamed from *recordlookupone*.

petl.**expr**(*s*)

Construct a function operating on a record (i.e., a dictionary representation of a data row, indexed by field name).

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.**strjoin**(*s*)

Return a function to join sequences using *s* as the separator.

petl.**randomtable**(*numflds=5*, *numrows=100*, *wait=0*)

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.:

```
>>> from petl import randomtable, look
>>> t = randomtable(5, 10000)
>>> look(t)
+--------------------+--------------------+--------------------+--------------------+------
| 'f0'               | 'f1'               | 'f2'               | 'f3'               | 'f4'
+====================+====================+====================+====================+======
| 0.37981479583619415 | 0.5651754962690851 | 0.5219839418441516 | 0.400507081757018  | 0.187
+--------------------+--------------------+--------------------+--------------------+------
| 0.8523718373108918 | 0.9728988775985702 | 0.539819811070272  | 0.5253127991162814 | 0.032
+--------------------+--------------------+--------------------+--------------------+------
| 0.15767415808765595 | 0.8723372406647985 | 0.8116271113050197 | 0.19606663402788693 | 0.029
+--------------------+--------------------+--------------------+--------------------+------
| 0.29027126477145737 | 0.9458013821235983 | 0.0558711583090582 | 0.8388382491420909 | 0.533
+--------------------+--------------------+--------------------+--------------------+------
| 0.7299727877963395 | 0.7293822340944851 | 0.953624640847381  | 0.7161554959575555 | 0.868
+--------------------+--------------------+--------------------+--------------------+------
| 0.7057077618876934 | 0.5222733323906424 | 0.26527912571554013 | 0.41069309093677264 | 0.706
+--------------------+--------------------+--------------------+--------------------+------
| 0.9447075997744453 | 0.3980291877822444 | 0.5748113148854611 | 0.037655670603881974 | 0.308
+--------------------+--------------------+--------------------+--------------------+------
| 0.21559911346698513 | 0.8353039675591192 | 0.5558847892537019 | 0.8561403358605812 | 0.011
+--------------------+--------------------+--------------------+--------------------+------
| 0.27334411287843097 | 0.10064946027523636 | 0.7476185996637322 | 0.26201984851765325 | 0.630
+--------------------+--------------------+--------------------+--------------------+------
| 0.8348722928576766 | 0.40319578510057763 | 0.3658094978577834 | 0.9829576880714145 | 0.617
+--------------------+--------------------+--------------------+--------------------+------
```

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.

New in version 0.6.

See also `dummytable()`.

petl.**dummytable**(*numrows=100, fields=[('foo', <functools.partial object at 0x2ab6628>), ('bar', <functools.partial object at 0x2ab6680>), ('baz', <built-in method random of Random object at 0x1276580>)], wait=0*)

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.:

```
>>> from petl import dummytable, look
>>> t1 = dummytable(10000)
>>> look(t1)
```

```
+-------+-----------+----------------------+
| 'foo' | 'bar'     | 'baz'                |
+=======+===========+======================+
| 98    | 'oranges' | 0.017443519200384117 |
+-------+-----------+----------------------+
| 85    | 'pears'   | 0.6126183086894914   |
+-------+-----------+----------------------+
| 43    | 'apples'  | 0.8354915052285888   |
+-------+-----------+----------------------+
| 32    | 'pears'   | 0.9612740566307508   |
+-------+-----------+----------------------+
| 35    | 'bananas' | 0.4845179128370132   |
+-------+-----------+----------------------+
| 16    | 'pears'   | 0.150174888085586    |
+-------+-----------+----------------------+
| 98    | 'bananas' | 0.22592589109877748  |
+-------+-----------+----------------------+
| 82    | 'bananas' | 0.4887849296756226   |
+-------+-----------+----------------------+
| 75    | 'apples'  | 0.8414305202212253   |
+-------+-----------+----------------------+
| 78    | 'bananas' | 0.025845900016858714 |
+-------+-----------+----------------------+
```

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.

Data generation functions can be specified via the *fields* keyword argument, or set on the table via the suffix notation, e.g.:

```
>>> import random
>>> from functools import partial
>>> t2 = dummytable(10000, fields=[('foo', random.random), ('bar', partial(random.randint, 0, 50
>>> t2['baz'] = partial(random.choice, ['chocolate', 'strawberry', 'vanilla'])
>>> look(t2)
+---------------------+-------+--------------+
| 'foo'               | 'bar' | 'baz'        |
+=====================+=======+==============+
| 0.04595169186388326 | 370   | 'strawberry' |
+---------------------+-------+--------------+
| 0.29252999472988905 | 90    | 'chocolate'  |
+---------------------+-------+--------------+
| 0.7939324498894116  | 146   | 'chocolate'  |
+---------------------+-------+--------------+
| 0.4964898678468417  | 123   | 'chocolate'  |
+---------------------+-------+--------------+
| 0.26250784199548494 | 327   | 'strawberry' |
+---------------------+-------+--------------+
| 0.748470693146964   | 275   | 'strawberry' |
+---------------------+-------+--------------+
| 0.8995553034254133  | 151   | 'strawberry' |
+---------------------+-------+--------------+
| 0.26331484411715367 | 211   | 'chocolate'  |
+---------------------+-------+--------------+
| 0.4740252948218193  | 364   | 'vanilla'    |
+---------------------+-------+--------------+
| 0.166428545780258   | 59    | 'vanilla'    |
+---------------------+-------+--------------+
```

Changed in version 0.6.

Now supports different field types, e.g., non-numeric. Previous functionality is available as `randomtable()`.

`petl.`**`diffheaders`**(*t1*, *t2*)

Return the difference between the headers of the two tables as a pair of sets. E.g.:

```
>>> from petl import diffheaders
>>> table1 = [['foo', 'bar', 'baz'],
...           ['a', 1, .3]]
>>> table2 = [['baz', 'bar', 'quux'],
...           ['a', 1, .3]]
>>> add, sub = diffheaders(table1, table2)
>>> add
set(['quux'])
>>> sub
set(['foo'])
```

New in version 0.6.

`petl.`**`diffvalues`**(*t1*, *t2*, *f*)

Return the difference between the values under the given field in the two tables, e.g.:

```
>>> from petl import diffvalues
>>> table1 = [['foo', 'bar'],
...           ['a', 1],
...           ['b', 3]]
>>> table2 = [['bar', 'foo'],
...           [1, 'a'],
...           [3, 'c']]
>>> add, sub = diffvalues(table1, table2, 'foo')
>>> add
set(['c'])
>>> sub
set(['b'])
```

New in version 0.6.

`petl.`**`heapqmergesorted`**(*key=None*, **iterables*)

Return a single iterator over the given iterables, sorted by the given *key* function, assuming the input iterables are already sorted by the same function. (I.e., the merge part of a general merge sort.) Uses `heapq.merge()` for the underlying implementation. See also `shortlistmergesorted()`.

New in version 0.9.

`petl.`**`shortlistmergesorted`**(*key=None*, *reverse=False*, **iterables*)

Return a single iterator over the given iterables, sorted by the given *key* function, assuming the input iterables are already sorted by the same function. (I.e., the merge part of a general merge sort.) Uses `min()` (or `max()` if `reverse=True`) for the underlying implementation. See also `heapqmergesorted()`.

New in version 0.9.

`petl.`**`progress`**(*table*, *batchsize=1000*, *prefix=''*, *out=<open file '<stderr>', mode 'w' at 0x7f2e3d7391e0>*)

Report progress on rows passing through. E.g.:

```
>>> from petl import dummytable, progress, tocsv
>>> d = dummytable(100500)
>>> p = progress(d, 10000)
>>> tocsv(p, 'output.csv')
10000 rows in 0.57s (17574 rows/second); batch in 0.57s (17574 rows/second)
20000 rows in 1.13s (17723 rows/second); batch in 0.56s (17876 rows/second)
```

```
30000 rows in 1.69s (17732 rows/second); batch in 0.56s (17749 rows/second)
40000 rows in 2.27s (17652 rows/second); batch in 0.57s (17418 rows/second)
50000 rows in 2.83s (17679 rows/second); batch in 0.56s (17784 rows/second)
60000 rows in 3.39s (17694 rows/second); batch in 0.56s (17769 rows/second)
70000 rows in 3.96s (17671 rows/second); batch in 0.57s (17534 rows/second)
80000 rows in 4.53s (17677 rows/second); batch in 0.56s (17720 rows/second)
90000 rows in 5.09s (17681 rows/second); batch in 0.56s (17715 rows/second)
100000 rows in 5.66s (17675 rows/second); batch in 0.57s (17625 rows/second)
100500 rows in 5.69s (17674 rows/second)
```

See also `clock()`.

New in version 0.10.

`petl.`**`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.:

```
>>> from petl import dummytable, clock, convert, progress, tocsv
>>> t1 = dummytable(100000)
>>> c1 = clock(t1)
>>> t2 = convert(c1, 'foo', lambda v: v**2)
>>> c2 = clock(t2)
>>> p = progress(c2, 10000)
>>> tocsv(p, 'dummy.csv')
10000 rows in 1.17s (8559 rows/second); batch in 1.17s (8559 rows/second)
20000 rows in 2.34s (8548 rows/second); batch in 1.17s (8537 rows/second)
30000 rows in 3.51s (8547 rows/second); batch in 1.17s (8546 rows/second)
40000 rows in 4.68s (8541 rows/second); batch in 1.17s (8522 rows/second)
50000 rows in 5.89s (8483 rows/second); batch in 1.21s (8261 rows/second)
60000 rows in 7.30s (8221 rows/second); batch in 1.40s (7121 rows/second)
70000 rows in 8.59s (8144 rows/second); batch in 1.30s (7711 rows/second)
80000 rows in 9.78s (8182 rows/second); batch in 1.18s (8459 rows/second)
90000 rows in 10.98s (8193 rows/second); batch in 1.21s (8279 rows/second)
100000 rows in 12.30s (8132 rows/second); batch in 1.31s (7619 rows/second)
100000 rows in 12.30s (8132 rows/second)
>>> # time consumed retrieving rows from t1
... c1.time
5.4099999999999895
>>> # time consumed retrieving rows from t2
... c2.time
8.740000000000006
>>> # actual time consumed by the convert step
... c2.time - c1.time
3.330000000000016
```

See also `progress()`.

New in version 0.10.

`petl.`**`rowgroupby`**(*table*, *key*, *value=None*)

Convenient adapter for `itertools.groupby()`. E.g.:

```
>>> from petl import rowgroupby, look
>>> look(table)
+-------+-------+-------+
| 'foo' | 'bar' | 'baz' |
+=======+=======+=======+
| 'a'   | 1     | True  |
+-------+-------+-------+
```

```
| 'b'   | 3     | True  |
+-------+-------+-------+
| 'b'   | 2     |       |
+-------+-------+-------+

>>> # group entire rows
... for key, group in rowgroupby(table, 'foo'):
...     print key, list(group)
...
a [('a', 1, True)]
b [('b', 3, True), ('b', 2)]
>>> # group specific values
... for key, group in rowgroupby(table, 'foo', 'bar'):
...     print key, list(group)
...
a [1]
b [3, 2]
```

N.B., assumes the input table is already sorted by the given key.

New in version 0.10.

petl.**nthword**(*n*, *sep=None*)

Construct a function to return the nth word in a string. E.g.:

```
>>> from petl import nthword
>>> s = 'foo bar'
>>> f = nthword(0)
>>> f(s)
'foo'
>>> g = nthword(1)
>>> g(s)
'bar'
```

New in version 0.10.

petl.**cache**(*table*, *n=10000*)

Wrap the table with a cache that caches up to *n* rows as they are initially requested via iteration.

New in version 0.16.

CHAPTER 7

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 `fromcsv()` etc., with the exception of `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 `tocsv()` etc., with the exception of `StdinSource`, `URLSource` and `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` module for full details.

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

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

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

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

class `petl.`**`StdinSource`**

class `petl.`**`StdoutSource`**

class `petl.`**`URLSource`**(**args*, ***kwargs*)

class `petl.`**`StringSource`**(*s=None*)

class `petl.`**`PopenSource`**(**args*, ***kwargs*)

CHAPTER 8

Further Reading

8.1 petl.fluent - Alternative notation for combining transformations

New in version 0.6.

The module `petl.fluent` provides all of the functions present in the root `petl` module, but with modifications to allow them to be used in a fluent style. E.g.:

```
>>> from petl.fluent import *
>>> t0 = dummytable()
>>> t0.look()
+-------+-----------+----------------------+
| 'foo' | 'bar'     | 'baz'                |
+=======+===========+======================+
| 61    | 'oranges' | 0.41684297441746143  |
+-------+-----------+----------------------+
| 42    | 'bananas' | 0.5424838757229734   |
+-------+-----------+----------------------+
| 55    | 'pears'   | 0.044730394239418825 |
+-------+-----------+----------------------+
| 63    | 'apples'  | 0.6553751878324998   |
+-------+-----------+----------------------+
| 57    | 'pears'   | 0.33151097448517963  |
+-------+-----------+----------------------+
| 57    | 'apples'  | 0.2152565282912028   |
+-------+-----------+----------------------+
| 45    | 'bananas' | 0.1478840303008977   |
+-------+-----------+----------------------+
| 79    | 'pears'   | 0.14301990499723238  |
+-------+-----------+----------------------+
| 11    | 'pears'   | 0.16801320344526383  |
+-------+-----------+----------------------+
| 96    | 'oranges' | 0.3004187573856759   |
+-------+-----------+----------------------+

>>> t1 = t0.convert('bar', 'upper').extend('quux', 42).extend('spong', expr('{foo} * {quux}')).select
>>> t1.look()
+-------+----------+----------------------+--------+---------+
| 'foo' | 'bar'    | 'baz'                | 'quux' | 'spong' |
+=======+==========+======================+========+=========+
| 63    | 'APPLES' | 0.6553751878324998   | 42     | 2646    |
+-------+----------+----------------------+--------+---------+
| 57    | 'APPLES' | 0.2152565282912028   | 42     | 2394    |
```

```
+-------+----------+----------------------+--------+---------+
| 87    | 'APPLES' | 0.9045902500660937   | 42     | 3654    |
+-------+----------+----------------------+--------+---------+
| 5     | 'APPLES' | 0.6915135568859515   | 42     | 210     |
+-------+----------+----------------------+--------+---------+
| 28    | 'APPLES' | 0.8440288073976338   | 42     | 1176    |
+-------+----------+----------------------+--------+---------+
| 32    | 'APPLES' | 0.047452310539432774 | 42     | 1344    |
+-------+----------+----------------------+--------+---------+
| 93    | 'APPLES' | 0.8100969279893147   | 42     | 3906    |
+-------+----------+----------------------+--------+---------+
| 94    | 'APPLES' | 0.8216793407511486   | 42     | 3948    |
+-------+----------+----------------------+--------+---------+
| 94    | 'APPLES' | 0.7911584363109934   | 42     | 3948    |
+-------+----------+----------------------+--------+---------+
| 34    | 'APPLES' | 0.18846546302867728  | 42     | 1428    |
+-------+----------+----------------------+--------+---------+
```

Alternatively, if you don't want to import all petl function names into the root namespace, you can just import the module and use the `wrap()` function, e.g.:

```
>>> import petl.fluent as etl
>>> l = [['foo', 'bar'], ['a', 1], ['b', 3]]
>>> tbl = etl.wrap(l)
>>> tbl.look()
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 3     |
+-------+-------+

>>> tbl.cut('foo').look()
+-------+
| 'foo' |
+=======+
| 'a'   |
+-------+
| 'b'   |
+-------+

>>> tbl.tocsv('test.csv')
>>> etl.fromcsv('test.csv').look()
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '3'   |
+-------+-------+
```

Changed in version 0.21.

The recommended import and wrapping pattern (described above) has changed, see https://github.com/alimanfoo/petl/issues/230 for more details.

8.1.1 petl executable

New in version 0.10.

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

```
$ virtualenv petl
$ . petl/bin/activate
$ pip install petl
$ petl "dummytable().tocsv()" > dummy.csv
$ cat dummy.csv | petl "fromcsv().cut('foo', 'baz').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 petl.fluent functions imported.

8.2 petl.interactive - Optimisations for Use in Interactive Mode

New in version 0.5.

The module petl.interactive provides all of the functions present in the root petl module, but with a couple of optimisations for use within an interactive session.

The main optimisation is that some caching is done by default, such that the first 10000 rows of any table are cached in memory the first time they are requested. This usually provides a better experience when building up a transformation pipeline one step at a time, where you are examining the outputs of each intermediate step as its written via look() or see(). I.e., as each new step is added and the output examined, as long as less than 10000 rows are requested, only that new step will actually be executed, and none of the upstream transformations will be repeated, because the outputs from previous steps will have been cached.

The default cache size can be changed by setting petl.interactive.cachesize to an integer value.

Also, by default, the look() function is used to generate a representation of tables. So you don't need to type, e.g., >>> look(mytable), you can just type >>> mytable. The default representation function can be changed by setting petl.interactive.representation, e.g., petl.interactive.representation = petl.see, or petl.interactive.representation = None to disable this behaviour.

Finally, this module extends petl.fluent so you can use the fluent style if you wish, e.g.:

```
>>> import petl.interactive as etl
>>> l = [['foo', 'bar'], ['a', 1], ['b', 3]]
>>> tbl = etl.wrap(l)
>>> tbl
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | 1     |
+-------+-------+
| 'b'   | 3     |
+-------+-------+

>>> tbl.cut('foo')
+-------+
| 'foo' |
+=======+
| 'a'   |
+-------+
| 'b'   |
```

```
+-------+

>>> tbl.tocsv('test.csv')
>>> etl.fromcsv('test.csv')
+-------+-------+
| 'foo' | 'bar' |
+=======+=======+
| 'a'   | '1'   |
+-------+-------+
| 'b'   | '3'   |
+-------+-------+
```

Changed in version 0.21.

The recommended import and wrapping pattern (described above) has changed, see https://github.com/alimanfoo/petl/issues/230 for more details.

8.3 petl.push - Branching Pipelines

New in version 0.10.

8.3.1 Introduction

This module provides some functions for setting up branching data transformation pipelines.

The general pattern is to define the pipeline, connecting components together via the `pipe()` method call, then pushing data through the pipeline via the `push()` method call at the top of the pipeline. E.g.:

```
>>> from petl import fromcsv
>>> source = fromcsv('fruit.csv')
>>> from petl.push import *
>>> p = partition('fruit')
>>> p.pipe('orange', tocsv('oranges.csv'))
>>> p.pipe('banana', tocsv('bananas.csv'))
>>> p.push(source)
```

The pipe operator can also be used to connect components in the pipeline, by analogy with the use of the pipe character in unix/linux shells, e.g.:

```
>>> from petl import fromcsv
>>> source = fromcsv('fruit.csv')
>>> from petl.push import *
>>> p = partition('fruit')
>>> p | ('orange', tocsv('oranges.csv')
>>> p | ('banana', tocsv('bananas.csv')
>>> p.push(source)
```

8.3.2 Push Functions

petl.push.**partition**(*discriminator*)

Partition rows based on values of a field or results of applying a function on the row. E.g.:

```
>>> from petl.push import partition, tocsv
>>> p = partition('fruit')
```

```
>>> p.pipe('orange', tocsv('oranges.csv'))
>>> p.pipe('banana', tocsv('bananas.csv'))
>>> p.push(sometable)
```

In the example above, rows where the value of the 'fruit' field equals 'orange' are piped to the 'oranges.csv' file, and rows where the 'fruit' field equals 'banana' are piped to the 'bananas.csv' file.

petl.push.**sort**(*key=None*, *reverse=False*, *buffersize=None*)
Sort rows based on some key field or fields. E.g.:

```
>>> from petl.push import sort, tocsv
>>> p = sort('foo')
>>> p.pipe(tocsv('sorted_by_foo.csv'))
>>> p.push(sometable)
```

petl.push.**duplicates**(*key*)
Report rows with duplicate key values. E.g.:

```
>>> from petl.push import duplicates, tocsv
>>> p = duplicates('foo')
>>> p.pipe(tocsv('foo_dups.csv'))
>>> p.pipe('remainder', tocsv('foo_uniq.csv'))
>>> p.push(sometable)
```

N.B., assumes data are already sorted by the given key.

petl.push.**unique**(*key*)
Report rows with unique key values. E.g.:

```
>>> from petl.push import unique, tocsv
>>> p = unique('foo')
>>> p.pipe(tocsv('foo_uniq.csv'))
>>> p.pipe('remainder', tocsv('foo_dups.csv'))
>>> p.push(sometable)
```

N.B., assumes data are already sorted by the given key. See also `duplicates()`.

petl.push.**diff**()
Find rows that differ between two tables. E.g.:

```
>>> from petl.push import diff, tocsv
>>> p = diff()
>>> p.pipe('+', tocsv('added.csv'))
>>> p.pipe('-', tocsv('subtracted.csv'))
>>> p.pipe(tocsv('common.csv'))
>>> p.push(sometable, someothertable)
```

petl.push.**tocsv**(*filename*, *dialect=<class csv.excel at 0x2ad5050>*, ***kwargs*)
Push rows to a CSV file. E.g.:

```
>>> from petl.push import tocsv
>>> p = tocsv('example.csv')
>>> p.push(sometable)
```

petl.push.**totsv**(*filename*, *dialect=<class csv.excel_tab at 0x2ad50b8>*, ***kwargs*)
Push rows to a tab-delimited file. E.g.:

```
>>> from petl.push import totsv
>>> p = totsv('example.tsv')
>>> p.push(sometable)
```

petl.push.**topickle**(*filename*, *protocol=-1*)
Push rows to a pickle file. E.g.:

```
>>> from petl.push import topickle
>>> p = topickle('example.pickle')
>>> p.push(sometable)
```

8.4 Case Study 1 - Comparing Tables

This case study illustrates some of the petl functions available for doing some simple profiling and comparison of data from two tables.

8.4.1 Introduction

The files used in this case study can be downloaded from the following link:

- http://aliman.s3.amazonaws.com/petl/petl-case-study-1-files.zip

Download and unzip the files:

```
$ wget http://aliman.s3.amazonaws.com/petl/petl-case-study-1-files.zip
$ unzip petl-case-study-1-files.zip
```

The first file (*snpdata.csv*) contains a list of locations in the genome of the malaria parasite *P. falciparum*, along with some basic data about genetic variations found at those locations.

The second file (*popdata.csv*) is supposed to contain the same list of genome locations, along with some additional data such as allele frequencies in different populations.

The main point for this case study is that the first file (*snpdata.csv*) contains the canonical list of genome locations, and the second file (*popdata.csv*) contains some additional data about the same genome locations and therefore should be consistent with the first file. We want to check whether this second file is in fact consistent with the first file.

8.4.2 Preparing the data

Start the Python interpreter and import petl functions:

```
$ python
Python 2.7.2+ (default, Oct  4 2011, 20:03:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from petl import *
```

To save some typing, let *a* be the table of data extracted from the first file (*snpdata.csv*), and let *b* be the table of data extracted from the second file (*popdata.csv*), using the fromcsv() function:

```
>>> a = fromcsv('snpdata.csv', delimiter='\t')
>>> b = fromcsv('popdata.csv', delimiter='\t')
```

Examine the header from each file using the header() function:

```
>>> header(a)
('Chr', 'Pos', 'Ref', 'Nref', 'Der', 'Mut', 'isTypable', 'GeneId', 'GeneAlias', 'GeneDescr')
>>> header(b)
('Chromosome', 'Coordinates', 'Ref. Allele', 'Non-Ref. Allele', 'Outgroup Allele', 'Ancestral Allele'
```

There is a common set of 9 fields that is present in both tables, and we would like focus on comparing these common fields, however different field names have been used in the two files. To simplify comparison, use `rename()` to rename some fields in the second file:

```
>>> b_renamed = rename(b, {'Chromosome': 'Chr', 'Coordinates': 'Pos', 'Ref. Allele': 'Ref', 'Non-Ref
>>> header(b_renamed)
('Chr', 'Pos', 'Ref', 'Nref', 'Outgroup Allele', 'Ancestral Allele', 'Der', 'Ref. Aminoacid', 'Non-Re
```

Use `cut()` to extract only the fields we're interested in from both tables:

```
>>> common_fields = ['Chr', 'Pos', 'Ref', 'Nref', 'Der', 'Mut', 'GeneId', 'GeneAlias', 'GeneDescr']
>>> a_common = cut(a, common_fields)
>>> b_common = cut(b_renamed, common_fields)
```

Inspect the data:

```
>>> look(a_common)
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'Chr'  | 'Pos'   | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'  | 'GeneAlias' | 'GeneDescr'
+========+=========+=======+========+=======+=======+===========+=============+====================
| 'MAL1' | '91099' | 'G'   | 'A'    | '-'   | 'S'   | 'PFA0095c' | 'MAL1P1.10' | 'rifin'
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '91104' | 'A'   | 'T'    | '-'   | 'N'   | 'PFA0095c' | 'MAL1P1.10' | 'rifin'
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93363' | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93382' | 'T'   | 'G'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93384' | 'G'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93390' | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93439' | 'T'   | 'C'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '93457' | 'C'   | 'T'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '94008' | 'T'   | 'C'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------
| 'MAL1' | '94035' | 'C'   | 'T'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protei
+--------+---------+-------+--------+-------+-------+-----------+-------------+--------------------

>>> look(b_common)
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'Chr'  | 'Pos'   | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'  | 'GeneAlias'     | 'GeneDescr'
+========+=========+=======+========+=======+=======+===========+=================+================
| 'MAL1' | '91099' | 'G'   | 'A'    | '-'   | 'SYN' | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '91104' | 'A'   | 'T'    | '-'   | 'NON' | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '93363' | 'T'   | 'A'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '93382' | 'T'   | 'G'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '93384' | 'G'   | 'A'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '93390' | 'T'   | 'A'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
| 'MAL1' | '93439' | 'T'   | 'C'    | '-'   | 'SYN' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+-----------+-----------------+----------------
```

```
| 'MAL1' | '93457' | 'C'   | 'T'    | '-'   | 'SYN' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | '94008' | 'T'   | 'C'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | '94035' | 'C'   | 'T'    | '-'   | 'NON' | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium expo
+--------+---------+-------+--------+-------+-------+------------+-----------------+-----------------
```

The `fromcsv()` function does not attempt to parse any of the values from the underlying CSV file, so all values are reported as strings. However, the 'Pos' field should be interpreted as an integer.

Also, the 'Mut' field has a different representation in the two tables, which needs to be converted before the data can be compared.

Use the `convert()` function to convert the type of the 'Pos' field in both tables and the representation of the 'Mut' field in table *b*:

```
>>> a_conv = convert(a_common, 'Pos', int)
>>> b_conv1 = convert(b_common, 'Pos', int)
>>> b_conv2 = convert(b_conv1, 'Mut', {'SYN': 'S', 'NON': 'N'})
>>> b_conv = b_conv2
>>> look(a_conv)
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'Chr'  | 'Pos' | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'   | 'GeneAlias' | 'GeneDescr'
+========+=======+=======+========+=======+=======+============+=============+======================
| 'MAL1' | 91099 | 'G'   | 'A'    | '-'   | 'S'   | 'PFA0095c' | 'MAL1P1.10' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 91104 | 'A'   | 'T'    | '-'   | 'N'   | 'PFA0095c' | 'MAL1P1.10' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93363 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93382 | 'T'   | 'G'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93384 | 'G'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93390 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93439 | 'T'   | 'C'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 93457 | 'C'   | 'T'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 94008 | 'T'   | 'C'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------
| 'MAL1' | 94035 | 'C'   | 'T'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein,
+--------+-------+-------+--------+-------+-------+------------+-------------+----------------------

>>> look(b_conv)
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'Chr'  | 'Pos' | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'   | 'GeneAlias'     | 'GeneDescr'
+========+=======+=======+========+=======+=======+============+=================+=================
| 'MAL1' | 91099 | 'G'   | 'A'    | '-'   | 'S'   | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | 91104 | 'A'   | 'T'    | '-'   | 'N'   | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | 93363 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | 93382 | 'T'   | 'G'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
| 'MAL1' | 93384 | 'G'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-----------------
```

```
| 'MAL1' | 93390 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93439 | 'T'   | 'C'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93457 | 'C'   | 'T'    | '-'   | 'S'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 94008 | 'T'   | 'C'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 94035 | 'C'   | 'T'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
```

Now the tables are ready for comparison.

8.4.3 Looking for missing or unexpected rows

Because both tables should contain the same list of genome locations, they should have the same number of rows. Use `rowcount()` to compare:

```
>>> rowcount(a_conv)
103647
>>> rowcount(b_conv)
103618
>>> 103647-103618
29
```

There is some discrepancy. First investigate by comparing just the genomic locations, defined by the 'Chr' and 'Pos' fields, using `complement()`:

```
>>> a_locs = cut(a_conv, 'Chr', 'Pos')
>>> b_locs = cut(b_conv, 'Chr', 'Pos')
>>> locs_only_in_a = complement(a_locs, b_locs)
>>> rowcount(locs_only_in_a)
29
>>> look(locs_only_in_a)
+---------+---------+
| 'Chr'   | 'Pos'   |
+=========+=========+
| 'MAL1'  | 216961  |
+---------+---------+
| 'MAL10' | 538210  |
+---------+---------+
| 'MAL10' | 548779  |
+---------+---------+
| 'MAL10' | 1432969 |
+---------+---------+
| 'MAL11' | 500289  |
+---------+---------+
| 'MAL11' | 1119809 |
+---------+---------+
| 'MAL11' | 1278859 |
+---------+---------+
| 'MAL12' | 51827   |
+---------+---------+
| 'MAL13' | 183727  |
+---------+---------+
| 'MAL13' | 398404  |
+---------+---------+
```

```
>>> locs_only_in_b = complement(b_locs, a_locs)
>>> rowcount(locs_only_in_b)
0
```

So it appears that 29 locations are missing from table *b*. Export these missing locations to a CSV file using `tocsv()`:

```
>>> tocsv(locs_only_in_a, 'missing_locations.csv', delimiter='\t')
```

A shorthand for finding the difference between two tables is the `diff()` function:

```
>>> locs_only_in_b, locs_only_in_a = diff(a_locs, b_locs)
```

An alternative method for finding rows in one table where some key value is not present in another table is to use the `antijoin()` function:

```
>>> locs_only_in_a = antijoin(a_conv, b_conv, key=('Chr', 'Pos'))
>>> rowcount(locs_only_in_a)
29
>>> look(locs_only_in_a)
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'Chr'   | 'Pos'   | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'    | 'GeneAlias'            | 'Gene
+=========+=========+=======+========+=======+=======+=============+========================+======
| 'MAL1'  | 216961  | 'T'   | 'C'    | 'C'   | 'S'   | 'PFA0245w'  | 'MAL1P1.40'            | 'nove
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL10' | 538210  | 'T'   | 'A'    | '-'   | 'N'   | 'PF10_0133' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL10' | 548779  | 'A'   | 'C'    | '-'   | 'N'   | 'PF10_0136' | '-'                    | 'Init
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL10' | 1432969 | 'T'   | 'C'    | '-'   | 'N'   | 'PF10_0355' | '-'                    | 'Eryt
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL11' | 500289  | 'C'   | 'A'    | 'A'   | 'N'   | 'PF11_0528' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL11' | 1119809 | 'G'   | 'C'    | 'C'   | 'N'   | 'PF11_0300' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL11' | 1278859 | 'A'   | 'G'    | '-'   | 'N'   | 'PF11_0341' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL12' | 51827   | 'T'   | 'G'    | '-'   | 'N'   | 'PFL0030c'  | '2277.t00006,MAL12P1.6' | 'eryt
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL13' | 183727  | 'G'   | 'C'    | '-'   | 'N'   | 'MAL13P1.18' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
| 'MAL13' | 398404  | 'G'   | 'T'    | '-'   | 'S'   | 'MAL13P1.41' | '-'                    | 'hypo
+---------+---------+-------+--------+-------+-------+-------------+------------------------+------
```

8.4.4 Finding conflicts

We'd also like to compare the values given in the other fields, to find any discrepancies between the two tables.

First, examine all fields for discrepancies, using the `cat()` and `conflicts()` functions:

```
>>> ab = cat(a_conv, b_conv)
>>> ab_conflicts = conflicts(ab, key=('Chr', 'Pos'))
>>> rowcount(ab_conflicts)
185978
>>> look(ab_conflicts)
+--------+-------+-------+--------+-------+-------+-----------+-----------------+-------------------
| 'Chr'  | 'Pos' | 'Ref' | 'Nref' | 'Der' | 'Mut' | 'GeneId'  | 'GeneAlias'     | 'GeneDescr'
+========+=======+=======+========+=======+=======+===========+=================+===================
| 'MAL1' | 91099 | 'G'   | 'A'    | '-'   | 'S'   | 'PFA0095c' | 'MAL1P1.10'     | 'rifin'
```

```
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 91099 | 'G'   | 'A'    | '-'   | 'S'   | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 91104 | 'A'   | 'T'    | '-'   | 'N'   | 'PFA0095c' | 'MAL1P1.10'     | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 91104 | 'A'   | 'T'    | '-'   | 'N'   | 'PFA0095c' | 'MAL1P1.10,RIF' | 'rifin'
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93363 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93363 | 'T'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'hypothetical prot
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93382 | 'T'   | 'G'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93382 | 'T'   | 'G'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'hypothetical prot
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93384 | 'G'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'Plasmodium export
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
| 'MAL1' | 93384 | 'G'   | 'A'    | '-'   | 'N'   | 'PFA0100c' | 'MAL1P1.11'     | 'hypothetical prot
+--------+-------+-------+--------+-------+-------+------------+-----------------+-------------------
```

From a glance at the conflicts above, it appears there are discrepancies in the 'GeneAlias' and 'GeneDescr' fields. There may also be conflicts in other fields, so we need to investigate further.

Find conflicts, one field at a time, starting with the 'Ref', 'Nref', 'Der' and 'Mut' fields:

```
>>> ab_ref = cut(ab, 'Chr', 'Pos', 'Ref')
>>> ab_ref_conflicts = conflicts(ab_ref, key=('Chr', 'Pos'))
>>> rowcount(ab_ref_conflicts)
0
>>> ab_nref = cut(ab, 'Chr', 'Pos', 'Nref')
>>> ab_nref_conflicts = conflicts(ab_nref, key=('Chr', 'Pos'))
>>> rowcount(ab_nref_conflicts)
0
>>> ab_der = cut(ab, 'Chr', 'Pos', 'Der')
>>> ab_der_conflicts = conflicts(ab_der, key=('Chr', 'Pos'))
>>> rowcount(ab_der_conflicts)
0
>>> ab_mut = cut(ab, 'Chr', 'Pos', 'Mut')
>>> ab_mut_conflicts = conflicts(ab_mut, key=('Chr', 'Pos'))
>>> rowcount(ab_mut_conflicts)
3592
>>> look(ab_mut_conflicts)
+--------+--------+-------+
| 'Chr'  | 'Pos'  | 'Mut' |
+========+========+=======+
| 'MAL1' | 99099  | '-'   |
+--------+--------+-------+
| 'MAL1' | 99099  | 'N'   |
+--------+--------+-------+
| 'MAL1' | 99211  | '-'   |
+--------+--------+-------+
| 'MAL1' | 99211  | 'N'   |
+--------+--------+-------+
| 'MAL1' | 197903 | 'N'   |
+--------+--------+-------+
| 'MAL1' | 197903 | 'S'   |
+--------+--------+-------+
| 'MAL1' | 384429 | 'N'   |
+--------+--------+-------+
```

```
| 'MAL1' | 384429 | 'S'    |
+--------+--------+--------+
| 'MAL1' | 513268 | 'N'    |
+--------+--------+--------+
| 'MAL1' | 513268 | 'S'    |
+--------+--------+--------+

>>> tocsv(ab_mut_conflicts, 'mut_conflicts.csv', delimiter='\t')
```

So there are some conflicts in the 'Mut' field, and we've saved them to a file for later reference.

We'd like also to find conflicts in the 'GeneId', 'GeneAlias' and 'GeneDescr' fields. Here, while we're focusing on finding conflicts in one field at a time, we'll include information from the other fields in the output, to provide extra diagnostic information when investigating the cause of any conflicts, making use of the *include* keyword argument to the `conflicts()` function:

```
>>> ab_gene = cut(ab, 'Chr', 'Pos', 'GeneId', 'GeneAlias', 'GeneDescr')
>>> ab_geneid_conflicts = conflicts(ab_gene, key=('Chr', 'Pos'), include='GeneId')
>>> rowcount(ab_geneid_conflicts)
5434
>>> look(ab_geneid_conflicts)
+--------+--------+------------+----------------------+-----------------------------------------
| 'Chr'  | 'Pos'  | 'GeneId'   | 'GeneAlias'          | 'GeneDescr'
+========+========+============+======================+=========================================
| 'MAL1' | 190628 | 'PFA0215w' | 'MAL1P1.70'          | 'hypothetical protein, conserved'
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190628 | 'PFA0220w' | 'PFA0215w,MAL1P1.34b' | 'ubiquitin carboxyl-terminal hydrolase, puta
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190668 | 'PFA0215w' | 'MAL1P1.70'          | 'hypothetical protein, conserved'
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190668 | 'PFA0220w' | 'PFA0215w,MAL1P1.34b' | 'ubiquitin carboxyl-terminal hydrolase, puta
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190786 | 'PFA0215w' | 'MAL1P1.70'          | 'hypothetical protein, conserved'
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190786 | 'PFA0220w' | 'PFA0215w,MAL1P1.34b' | 'ubiquitin carboxyl-terminal hydrolase, puta
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190808 | 'PFA0215w' | 'MAL1P1.70'          | 'hypothetical protein, conserved'
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190808 | 'PFA0220w' | 'PFA0215w,MAL1P1.34b' | 'ubiquitin carboxyl-terminal hydrolase, puta
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190854 | 'PFA0215w' | 'MAL1P1.70'          | 'hypothetical protein, conserved'
+--------+--------+------------+----------------------+-----------------------------------------
| 'MAL1' | 190854 | 'PFA0220w' | 'PFA0215w,MAL1P1.34b' | 'ubiquitin carboxyl-terminal hydrolase, puta
+--------+--------+------------+----------------------+-----------------------------------------

>>> tocsv(ab_geneid_conflicts, 'geneid_conflicts.csv', delimiter='\t')
>>> ab_genealias_conflicts = conflicts(ab_gene, key=('Chr', 'Pos'), include='GeneAlias')
>>> rowcount(ab_genealias_conflicts)
39144
>>> look(ab_genealias_conflicts)
+--------+--------+------------+----------------------------------------+-------------------------
| 'Chr'  | 'Pos'  | 'GeneId'   | 'GeneAlias'                            | 'GeneDescr'
+========+========+============+========================================+=========================
| 'MAL1' | 91099  | 'PFA0095c' | 'MAL1P1.10'                            | 'rifin'
+--------+--------+------------+----------------------------------------+-------------------------
| 'MAL1' | 91099  | 'PFA0095c' | 'MAL1P1.10,RIF'                        | 'rifin'
+--------+--------+------------+----------------------------------------+-------------------------
| 'MAL1' | 91104  | 'PFA0095c' | 'MAL1P1.10'                            | 'rifin'
+--------+--------+------------+----------------------------------------+-------------------------
```

```
| 'MAL1' | 91104  | 'PFA0095c' | 'MAL1P1.10,RIF'                          | 'rifin'
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 99099  | 'PFA0110w' | 'MAL1P1.13,Pf155'                        | 'ring-infected erythrocyt
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 99099  | 'PFA0110w' | 'MAL1P1.13,Pf155,RESA'                   | 'ring-infected erythrocyt
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 99211  | 'PFA0110w' | 'MAL1P1.13,Pf155'                        | 'ring-infected erythrocyt
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 99211  | 'PFA0110w' | 'MAL1P1.13,Pf155,RESA'                   | 'ring-infected erythrocyt
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 111583 | 'PFA0125c' | 'JESEBL,jesebl,eba-181,EBA181,MAL1P1.16' | 'erythrocyte binding anti
+--------+--------+------------+------------------------------------------+-------------------------
| 'MAL1' | 111583 | 'PFA0125c' | 'jesebl,JESEBL,eba-181,MAL1P1.16'        | 'erythrocyte binding anti
+--------+--------+------------+------------------------------------------+-------------------------

>>> tocsv(ab_genealias_conflicts, 'genealias_conflicts.csv', delimiter='\t')
>>> ab_genedescr_conflicts = conflicts(ab_gene, key=('Chr', 'Pos'), include='GeneDescr')
>>> rowcount(ab_genedescr_conflicts)
177730
>>> look(ab_genedescr_conflicts)
+--------+-------+------------+-------------+-----------------------------------------------------
| 'Chr'  | 'Pos' | 'GeneId'   | 'GeneAlias' | 'GeneDescr'
+========+=======+============+=============+=====================================================
| 'MAL1' | 93363 | 'PFA0100c' | 'MAL1P1.11' | 'Plasmodium exported protein (PHISTa), unknown functio
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93363 | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein, conserved in P. falciparum'
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93382 | 'PFA0100c' | 'MAL1P1.11' | 'Plasmodium exported protein (PHISTa), unknown functio
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93382 | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein, conserved in P. falciparum'
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93384 | 'PFA0100c' | 'MAL1P1.11' | 'Plasmodium exported protein (PHISTa), unknown functio
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93384 | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein, conserved in P. falciparum'
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93390 | 'PFA0100c' | 'MAL1P1.11' | 'Plasmodium exported protein (PHISTa), unknown functio
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93390 | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein, conserved in P. falciparum'
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93439 | 'PFA0100c' | 'MAL1P1.11' | 'Plasmodium exported protein (PHISTa), unknown functio
+--------+-------+------------+-------------+-----------------------------------------------------
| 'MAL1' | 93439 | 'PFA0100c' | 'MAL1P1.11' | 'hypothetical protein, conserved in P. falciparum'
+--------+-------+------------+-------------+-----------------------------------------------------

>>> tocsv(ab_genedescr_conflicts, 'genedescr_conflicts.csv', delimiter='\t')
```

So each of these three fields has a different set of conflicts, and we've saved the output to CSV files for later reference.

8.5 Related Work

8.5.1 continuum.io

- http://continuum.io

In development, a major revision of NumPy to better support a range of data integration and processing use cases.

8.5.2 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.

8.5.3 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.

8.5.4 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.

8.5.5 pydataframe (Python package)

- http://code.google.com/p/pydataframe/

An implemention of an almost R like DataFrame object.

8.5.6 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.

8.5.7 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, crosstable/pivot.

Does not rely on streaming/iterative processing of data, and has a persistence capability based on zodb for handling larger datasets.

8.5.8 csvkit (Python package)

- http://pypi.python.org/pypi/picalo/
- http://csvkit.rtfd.org/

A set of command-line utilities for transforming tabular data from CSV (delimited) files. Includes csvclean, csvcut, csvjoin, csvsort, csvstack, csvstat, csvgrep, csvlook.

8.5.9 csvutils (Python package)

- http://pypi.python.org/pypi/csvutils

8.5.10 python-pipeline (Python package

- http://code.google.com/p/python-pipeline/

8.5.11 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.

8.5.12 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.

8.5.13 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

8.5.14 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.

8.5.15 Talend

- http://www.talend.com

8.5.16 Jaspersoft ETL

- http://www.jaspersoft.com/jasperetl

8.5.17 CloverETL

- http://www.cloveretl.com/

8.5.18 Apatar

- http://apatar.com/

8.5.19 Jitterbit

- http://www.jitterbit.com/

8.5.20 Scriptella

- http://scriptella.javaforge.com/

8.5.21 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

8.5.22 Flat File Checker (FlaFi)

- http://www.flat-file.net/

8.5.23 Orange

- http://orange.biolab.si/

8.5.24 North Concepts Data Pipeline

- http://northconcepts.com/data-pipeline/

8.5.25 SAS Clinical Data Integration

- http://www.sas.com/industry/pharma/cdi/index.html

8.5.26 R Reshape Package

- http://had.co.nz/reshape/

8.5.27 TableFu

- http://propublica.github.com/table-fu/

8.5.28 python-tablefu

- https://github.com/eyeseast/python-tablefu

8.5.29 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

8.5.30 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.

8.5.31 OpenETL

- https://launchpad.net/openetl
- http://bazaar.launchpad.net/~openerp-commiter/openetl/OpenETL/files/head:/lib/openetl/component/transform/

8.5.32 Data River

- http://www.datariver.it/

8.5.33 Ruffus

- http://www.ruffus.org.uk/

8.5.34 PyF

- http://pyfproject.org/

8.5.35 PyDTA

- http://presbrey.mit.edu/PyDTA

8.5.36 Google Fusion Tables

- http://www.google.com/fusiontables/Home/

8.5.37 pivottable (Python package)

- http://pypi.python.org/pypi/pivottable/0.8

8.5.38 PrettyTable (Python package)

- http://pypi.python.org/pypi/PrettyTable

8.5.39 PyTables (Python package)

- http://www.pytables.org/

8.5.40 plyr

- http://plyr.had.co.nz/

8.5.41 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

8.5.42 SwiftRiver

- http://ushahidi.com/products/swiftriver-platform

8.5.43 Data Science Toolkit

- http://www.datasciencetoolkit.org/about

8.5.44 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.

8.5.45 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://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://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 9

Indices and tables

- *genindex*
- *modindex*
- *search*

p

petl, **??**
petl.fluent, **??**
petl.interactive, **??**
petl.push, **??**