asciitable Documentation

Release 0.8.0

Tom Aldcroft

May 09, 2013

CONTENTS

An extensible ASCII table reader and writer for Python 2 and 3.

`Asciitable` can read and write a wide range of ASCII table formats via built-in Extension Reader Classes:

- `Basic`: basic table with customizable delimiters and header configurations
- `Cds`: CDS format table (also Vizier and ApJ machine readable tables)
- `CommentedHeader`: column names given in a line that begins with the comment character
- `Daophot`: table from the IRAF DAOphot package
- `FixedWidth`: table with fixed-width columns (*Fixed-width Gallery*)
- `Ipac`: IPAC format table
- `Latex`, `AASTex`: LaTeX tables (plain and AASTex)
- `Memory`: table already in memory (list of lists, dict of lists, etc)
- `NoHeader`: basic table with no header where columns are auto-named
- `Rdb`: tab-separated values with an extra line after the column definition line
- `Tab`: tab-separated values

At the top level `asciitable` looks like many other ASCII table interfaces since it provides default `read()` and `write()` functions with long lists of parameters to accommodate the many variations possible in commonly encountered ASCII table formats. Below the hood however `asciitable` is built on a modular and extensible class structure. The basic functionality required for reading or writing a table is largely broken into independent base class elements so that new formats can be accomodated by modifying the underlying class methods as needed.

Warning: This package is no longer being developed.
The asciitable package has been moved into the Astropy project and is now known as astropy.io.ascii. This new version is very compatible with asciitable and most existing code should work.
The astropy.io.ascii package is being actively developed and contains many new features and bug fixes relative to asciitable. Users are strongly encouraged to migrate to astropy.io.ascii. If you have any questions or problems please send mail to the AstroPy mailing list (astropy@scipy.org).

Author Tom Aldcroft (aldcroft@head.cfa.harvard.edu)

CHAPTER

ONE

FIXED-WIDTH GALLERY

Fixed-width tables are those where each column has the same width for every row in the table. This is commonly used to make tables easy to read for humans or FORTRAN codes. It also reduces issues with quoting and special characters, for example:

```
Col1   Col2    Col3 Col4
---- --------- ---- ----
 1.2   "hello"    1    a
 2.4 's worlds    2    2
```

There are a number of common variations in the formatting of fixed-width tables which `asciitable` can read and write. The most signicant difference is whether there is no header line (`FixedWidthNoHeader`), one header line (`FixedWidth`), or two header lines (`FixedWidthTwoLine`). Next, there are variations in the delimiter character, whether the delimiter appears on either end ("bookends"), and padding around the delimiter.

Details are available in the class API documentation, but the easiest way to understand all the options and their interactions is by example.

1.1 Reading

1.1.1 FixedWidth

Nice, typical fixed format table

```
>>> import asciitable
>>> table = """
... # comment (with blank line above)
... |  Col1  |  Col2   |
... |  1.2   | "hello" |
... |  2.4   |'s worlds|
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

Typical fixed format table with col names provided

```
>>> table = """
... # comment (with blank line above)
... |  Col1  |  Col2   |
... |  1.2   | "hello" |
... |  2.4   |'s worlds|
```

```
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth, names=('name1', 'name2'))
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('name1', '<f8'), ('name2', '|S9')])
```

Weird input table with data values chopped by col extent

```
>>> table = """
...   Col1  |  Col2 |
...   1.2       "hello"
...   2.4   sdf's worlds
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth)
rec.array([(1.2, '"hel'), (2.4, "df's wo")],
      dtype=[('Col1', '<f8'), ('Col2', '|S7')])
```

Table with double delimiters

```
>>> table = """
... || Name ||   Phone ||         TCP||
... |  John  | 555-1234 |192.168.1.10X|
... |  Mary  | 555-2134 |192.168.1.12X|
... |   Bob  | 555-4527 | 192.168.1.9X|
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', '|S4'), ('Phone', '|S8'), ('TCP', '|S12')])
```

Table with space delimiter

```
>>> table = """
...  Name  --Phone-    ----TCP-----
...  John  555-1234    192.168.1.10
...  Mary  555-2134    192.168.1.12
...   Bob  555-4527     192.168.1.9
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth, delimiter=' ')
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('Name', '|S4'), ('--Phone-', '|S8'), ('----TCP-----', '|S12')])
```

Table with no header row and auto-column naming.

Use header_start and data_start keywords to indicate no header line.

```
>>> table = """
... |  John  | 555-1234 |192.168.1.10|
... |  Mary  | 555-2134 |192.168.1.12|
... |   Bob  | 555-4527 | 192.168.1.9|
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidth,
...                 header_start=None, data_start=0)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', '|S4'), ('col2', '|S8'), ('col3', '|S12')])
```

Table with no header row and with col names provided.

Second and third rows also have hanging spaces after final "l". Use header_start and data_start keywords to indicate no header line.

```
>>> table = ["|  John  | 555-1234 |192.168.1.10|"
...          "|  Mary  | 555-2134 |192.168.1.12|  "
...          "|   Bob  | 555-4527 | 192.168.1.9|  "]
>>> asciitable.read(table, Reader=asciitable.FixedWidth,
...                 header_start=None, data_start=0,
...                 names=('Name', 'Phone', 'TCP'))
rec.array([('John', '555-1234', '192.168.1.10')],
      dtype=[('Name', '|S4'), ('Phone', '|S8'), ('TCP', '|S12')])
```

1.1.2 FixedWidthNoHeader

Table with no header row and auto-column naming. Use the FixedWidthNoHeader convenience class.

```
>>> table = """
... |  John  | 555-1234 |192.168.1.10|
... |  Mary  | 555-2134 |192.168.1.12|
... |   Bob  | 555-4527 | 192.168.1.9|
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidthNoHeader)
rec.array([('John', '555-1234', '192.168.1.10'),
       ('Mary', '555-2134', '192.168.1.12'),
       ('Bob', '555-4527', '192.168.1.9')],
      dtype=[('col1', '|S4'), ('col2', '|S8'), ('col3', '|S12')])
```

Table with no delimiter with column start and end values specified.

This uses the col_starts and col_ends keywords. Note that the col_ends values are inclusive so a position range of 0 to 5 will select the first 6 characters.

```
>>> table = """
... #    5   9     17  18      28    <== Column start / end indexes
... #    |   |       ||         |    <== Column separation positions
...   John   555- 1234 192.168.1.10
...   Mary   555- 2134 192.168.1.12
...    Bob   555- 4527  192.168.1.9
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidthNoHeader,
...                 names=('Name', 'Phone', 'TCP'),
...                 col_starts=(0, 9, 18),
...                 col_ends=(5, 17, 28),
...                 )
rec.array([('John', '555- 1234', '192.168.1.'),
       ('Mary', '555- 2134', '192.168.1.'),
       ('Bob', '555- 4527', '192.168.1')],
      dtype=[('Name', '|S4'), ('Phone', '|S9'), ('TCP', '|S10')])
```

1.1.3 FixedWidthTwoLine

Typical fixed format table with two header lines with some cruft

```
>>> table = """
...   Col1    Col2
...   ----  ---------
...    1.2xx"hello"
...   2.4   's worlds
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidthTwoLine)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

Restructured text table

```
>>> table = """
... ======= ===========
...   Col1    Col2
... ======= ===========
...   1.2   "hello"
...   2.4   's worlds
... ======= ===========
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidthTwoLine,
...                 header_start=1, position_line=2, data_end=-1)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

Text table designed for humans and test having position line before the header line.

```
>>> table = """
... +------+----------+
... | Col1 |   Col2   |
... +------|----------+
... |  1.2 | "hello"  |
... |  2.4 | 's worlds|
... +------+----------+
... """
>>> asciitable.read(table, Reader=asciitable.FixedWidthTwoLine, delimiter='+',
...                 header_start=1, position_line=0, data_start=3, data_end=-1)
rec.array([(1.2, '"hello"'), (2.4, "'s worlds")],
      dtype=[('Col1', '<f8'), ('Col2', '|S9')])
```

1.2 Writing

1.2.1 FixedWidth

Define input values ''dat'' for all write examples.

```
>>> table = """
... | Col1 |  Col2     |  Col3 | Col4 |
... | 1.2  | "hello"   |  1    | a    |
... | 2.4  | 's worlds |  2    | 2    |
... """
>>> dat = asciitable.read(table, Reader=asciitable.FixedWidth)
```

Write a table as a normal fixed width table.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidth)
| Col1 |      Col2 | Col3 | Col4 |
|  1.2 |   "hello" |    1 |    a |
|  2.4 | 's worlds |    2 |    2 |
```

Write a table as a fixed width table with no padding.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidth, delimiter_pad=None)
|Col1|     Col2|Col3|Col4|
| 1.2|  "hello"|   1|   a|
| 2.4|'s worlds|   2|   2|
```

Write a table as a fixed width table with no bookend.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidth, bookend=False)
Col1 |      Col2 | Col3 | Col4
 1.2 |   "hello" |    1 |    a
 2.4 | 's worlds |    2 |    2
```

Write a table as a fixed width table with no delimiter.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidth, bookend=False, delimiter=None)
Col1       Col2  Col3  Col4
 1.2    "hello"     1     a
 2.4  's worlds     2     2
```

Write a table as a fixed width table with no delimiter and formatting.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidth,
...                  formats={'Col1': '%-8.3f', 'Col2': '%-15s'})
|     Col1 |            Col2 | Col3 | Col4 |
| 1.200    | "hello"         |    1 |    a |
| 2.400    | 's worlds       |    2 |    2 |
```

1.2.2 FixedWidthNoHeader

Write a table as a normal fixed width table.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthNoHeader)
| 1.2 |   "hello" | 1 | a |
| 2.4 | 's worlds | 2 | 2 |
```

Write a table as a fixed width table with no padding.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthNoHeader, delimiter_pad=None)
|1.2|  "hello"|1|a|
|2.4|'s worlds|2|2|
```

Write a table as a fixed width table with no bookend.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthNoHeader, bookend=False)
1.2 |   "hello" | 1 | a
2.4 | 's worlds | 2 | 2
```

Write a table as a fixed width table with no delimiter.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthNoHeader, bookend=False,
...                  delimiter=None)
```

```
1.2    "hello"  1  a
2.4  's worlds  2  2
```

1.2.3 FixedWidthTwoLine

Write a table as a normal fixed width table.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthTwoLine)
Col1      Col2 Col3 Col4
---- --------- ---- ----
 1.2   "hello"    1    a
 2.4 's worlds    2    2
```

Write a table as a fixed width table with space padding and '=' position_char.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthTwoLine,
...                  delimiter_pad=' ', position_char='=')
Col1        Col2   Col3   Col4
====   =========   ====   ====
 1.2     "hello"      1      a
 2.4   's worlds      2      2
```

Write a table as a fixed width table with no bookend.

```
>>> asciitable.write(dat, Writer=asciitable.FixedWidthTwoLine, bookend=True, delimiter='|')
|Col1|     Col2|Col3|Col4|
|----|---------|----|----|
| 1.2|  "hello"|   1|   a|
| 2.4|'s worlds|   2|   2|
```

CHAPTER

TWO

REQUIREMENTS

- asciitable passes its nosetests for the following platform / Python version combinations. Other combinations may work but have not been tried.

OS	Python version
Linux	2.4, 2.6, 2.7, 3.2
MacOS 10.6	2.7
Windows XP	2.7

- Though not required NumPy is recommended.
- NumPy versions 1.2 and 1.3 (Python 2) and 1.5 (Python 3) have been tested in previous releases, while current testing uses NumPy 1.6.

CHAPTER

THREE

DOWNLOAD

The latest release of the `asciitable` package is available on the Python Package Index at http://pypi.python.org/pypi/asciitable.

The latest git repository version is available at https://github.com/taldcroft/asciitable or with:

```
git clone git://github.com/taldcroft/asciitable.git
```

CHAPTER

FOUR

INSTALLATION AND TEST

The asciitable package includes a number of component modules that must be made available to the Python interpreter.

4.1 Easy way

The easy way to install asciitable is using `pip install` or `easy_install`. Either one will work, but `pip` is the more "modern" alternative. The following will download and install the package:

```
pip install [--user] asciitable
   ** OR **
easy_install [--user] asciitable
```

The `--user` option will install asciitable in a local user directory instead of within the Python installation directory structure. See the discussion on where packages get installed for more information. The `--user` option requires Python 2.6 or later.

4.2 Less easy way

Download and untar the package tarball, then change into the source directory:

```
tar zxf asciitable-<version>.tar.gz
cd asciitable-<version>
```

If you have the nose module installed then at this point you can run the test suite:

```
nosetests     # Python 2
nosetests3    # Python 3
```

There are several methods for installing. Choose ONE of them.

Python site-packages

If you have write access to the python site-packages directory you can do:

```
python setup.py install
```

Local user library

If you running python 2.6 or later the following command installs the asciitable module to the appropriate local user directory:

```
python setup.py install --user
```

CHAPTER

FIVE

READING TABLES

The majority of commonly encountered ASCII tables can be easily read with the `read()` function:

```
import asciitable
data = asciitable.read(table)
```

where `table` is the name of a file, a string representation of a table, or a list of table lines. By default `read()` will try to guess the table format by trying all the supported formats. If this does not work (for unusually formatted tables) then one needs give asciitable additional hints about the format, for example:

```
data = asciitable.read('t/nls1_stackinfo.dbout', data_start=2, delimiter='|')
data = asciitable.read('t/simple.txt', quotechar="'")
data = asciitable.read('t/simple4.txt', Reader=asciitable.NoHeader, delimiter='|')
table = ['col1 col2 col3', '1 2 hi', '3 4.2 there']
data = asciitable.read(table, delimiter=" ")
```

The `read()` function accepts a number of parameters that specify the detailed table format. Different Reader classes can define different defaults, so the descriptions below sometimes mention "typical" default values. This refers to the `Basic` reader and other similar Reader classes.

5.1 Commonly used parameters for `read()`

table [input table] There are four ways to specify the table to be read:

- Name of a file (string)
- Single string containing all table lines separated by newlines
- File-like object with a callable read() method
- List of strings where each list element is a table line

The first two options are distinguished by the presence of a newline in the string. This assumes that valid file names will not normally contain a newline.

Reader [Reader class (default= `BasicReader`)] This specifies the top-level format of the ASCII table, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be a Reader class. For basic usage this means one of the built-in Extension Reader Classes.

numpy [return a NumPy record array (default=True)] By default the output from `read()` is a NumPy record array object. This powerful container efficiently supports both column-wise and row access to the table and comes with the full NumPy stack of array manipulation methods.

If NumPy is not available or desired then set `numpy=False`. The output of `read()` will then be a dictionary of `Column` objects with each key for the corresponding named column.

guess: try to guess table format (default=True) If set to True then `read()` will try to guess the table format by cycling through a number of possible table format permuations and attemping to read the table in each case. See the Guess table format section for further details.

delimiter [column delimiter string] A one-character string used to separate fields which typically defaults to the space character. Other common values might be "\s" (whitespace), "," or "|" or "\t" (tab). A value of "\s" allows any combination of the tab and space characters to delimit columns.

comment [regular expression defining a comment line in table] If the `comment` regular expression matches the beginning of a table line then that line will be discarded from header or data processing. For the `Basic` Reader this defaults to "\s*#" (any whitespace followed by #).

quotechar [one-character string to quote fields containing special characters] This specifies the quote character and will typically be either the single or double quote character. This is can be useful for reading text fields with spaces in a space-delimited table. The default is typically the double quote.

header_start [line index for the header line not counting comment lines] This specifies in the line index where the header line will be found. Comment lines are not included in this count and the counting starts from 0 (first non-comment line has index=0). If set to None this indicates that there is no header line and the column names will be auto-generated. The default is dependent on the Reader.

data_start: line index for the start of data not counting comment lines This specifies in the line index where the data lines begin where the counting starts from 0 and does not include comment lines. The default is dependent on the Reader.

data_end: line index for the end of data (can be negative to count from end) If this is not None then it allows for excluding lines at the end that are not valid data lines. A negative value means to count from the end, so -1 would exclude the last line, -2 the last two lines, and so on.

converters: dict of data type converters See the Converters section for more information.

names: list of names corresponding to each data column Define the complete list of names for each data column. This will override names found in the header (if it exists). If not supplied then use names from the header or auto-generated names if there is no header.

include_names: list of names to include in output From the list of column names found from the header or the `names` parameter, select for output only columns within this list. If not supplied then include all names.

exclude_names: list of names to exlude from output Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values: fill value specifier of lists This can be used to fill missing values in the table or replace strings with special meaning. See the Replace bad or missing values section for more information and examples.

fill_include_names: list of column names, which are affected by `fill_values`. If not supplied, then `fill_values` can affect all columns.

fill_exclude_names: list of column names, which are not affected by `fill_values`. If not supplied, then `fill_values` can affect all columns.

5.2 Advanced parameters for `read()`

`read()` can accept a few more parameters that allow for code-level customization of the reading process. These will be discussed in the Advanced table reading section.

data_Splitter: Splitter class to split data columns

header_Splitter: Splitter class to split header columns

Inputter: Inputter class

Outputter: Outputter class

5.3 Replace bad or missing values

`Asciitable` can replace string values in the input data before they are converted. The most common use case is probably a table which contains string values that are not a valid representation of a number, e.g. `"..."` for a missing value or `""`. If `Asciitable` cannot convert all elements in a column to a numeric type, it will format the column as strings. To avoid this, `fill_values` can be used at the string level to fill missing values with the following syntax, which replaces `<old>` with `<new>` before the type conversion is done:

```
fill_values = <fill_spec> | [<fill_spec1>, <fill_spec2>, ...]
<fill_spec> = (<old>, <new>, <optional col name 1>, <optional col name 2>, ...)
```

Within the `<fill_spec>` tuple the `<old>` and `<new>` values must be strings. These two values are then followed by zero or more column names. If column names are included the replacement is limited to those columns listed. If no columns are specified then the replacement is done in every column, subject to filtering by `fill_include_names` and `fill_exclude_names` (see below).

The `fill_values` parameter in `read()` takes a single `<fill_spec>` or a list of `<fill_spec>` tuples. If several `<fill_spec>` apply to a single occurence of `<old>` then the first one determines the `<new>` value. For instance the following will replace an empty data value in the `x` or `y` columns with "1e38" while empty values in any other column will get "-999":

```
asciitable.read(table, fill_values=[('', '1e38', 'x', 'y'), ('', '-999')])
```

The following shows an example where string information needs to be exchanged before the conversion to float values happens. Here `no_rain` and `no_snow` is replaced by `0.0`:

```
table = ['day  rain     snow',     # column names
         #---  -------  --------
         'Mon  3.2      no_snow',
         'Tue  no_rain  1.1',
         'Wed  0.3      no_snow']
asciitable.read(table, fill_values=[('no_rain', '0.0'), ('no_snow', '0.0')])
```

Sometimes these rules apply only to specific columns in the table. Columns can be selected with `fill_include_names` or excluded with `fill_exclude_names`. Also, column names can be given directly with fill_values:

```
asciidata = ['text,no1,no2', 'text1,1,1.',',2,']
asciitable.read(asciidata, fill_values = ('', 'nan','no1','no2'), delimiter = ',')
```

Here, the empty value '' in column `no2` is replaced by `nan`, but the `text` column remains unaltered.

If the `numpy` module is available, then the default output is a NumPy masked array, where all values, which were replaced by `fill_values` are masked. See the description of the `NumpyOutputter` class for information on disabling masked arrays.

5.4 Guess table format

If the `guess` parameter in `read()` is set to True (which is the default) then `read()` will try to guess the table format by cycling through a number of possible table format permutations and attemping to read the table in each case. The first format which succeeds and will be used to read the table. To succeed the table must be successfully parsed by the Reader and satisfy the following column requirements:

- At least two table columns
- No column names are a float or int number
- No column names begin or end with space, comma, tab, single quote, double quote, or a vertical bar (|).

These requirements reduce the chance for a false positive where a table is successfully parsed with the wrong format. A common situation is a table with numeric columns but no header row, and in this case `asciitable` will auto-assign column names because of the restriction on column names that look like a number.

The order of guessing is shown by this Python code:

```
for Reader in (Rdb, Tab, Cds, Daophot, Ipac):
    read(Reader=Reader)
for Reader in (CommentedHeader, BasicReader, NoHeader):
    for delimiter in ("|", ",", " ", "\\s"):
        for quotechar in ('"', "'"):
            read(Reader=Reader, delimiter=delimiter, quotechar=quotechar)
```

Note that the `FixedWidth` derived-readers are not included in the default guess sequence (this causes problems), so to read such tables one must explicitly specify the reader class with the `Reader` keyword.

If none of the guesses succeed in reading the table (subject to the column requirements) a final try is made using just the user-supplied parameters but without checking the column requirements. In this way a table with only one column or column names that look like a number can still be successfully read.

The guessing process respects any values of the Reader, delimiter, and quotechar parameters that were supplied to the read() function. Any guesses that would conflict are skipped. For example the call:

```
dat = asciitable.read(table, Reader=NoHeader, quotechar="'")
```

would only try the four delimiter possibilities, skipping all the conflicting Reader and quotechar combinations.

Guessing can be disabled in two ways:

```
import asciitable
data = asciitable.read(table)                # guessing enabled by default
data = asciitable.read(table, guess=False)   # disable for this call
asciitable.set_guess(False)                  # set default to False globally
data = asciitable.read(table)                # guessing disabled
```

5.5 Converters

`Asciitable` converts the raw string values from the table into numeric data types by using converter functions such as the Python `int` and `float` functions. For example `int("5.0")` will fail while float("5.0") will succeed and return 5.0 as a Python float.

5.5.1 Without NumPy

The default set of converters for the `BaseOutputter` class is defined as such:

```
default_converters = [asciitable.convert_list(int),
                      asciitable.convert_list(float),
                      asciitable.convert_list(str)]
```

These take advantage of the `convert_list()` function which returns a 2-element tuple. The first element is function that will convert a list of values to the desired type. The second element is an `asciitable` class that specifies the type of data produced. This element should be one of `StrType`, `IntType`, or `FloatType`.

The conversion code steps through each applicable converter function and tries to call the function with a column of string values. If it succeeds without throwing an exception it will then break out, but otherwise move on to the next conversion function.

Use the `converters` keyword argument in order to force a specific data type for a column. This should be a dictionary with keys corresponding to the column names. Each dictionary value is a list similar to the `default_converter`. For example:

```
# col1 is int, col2 is float, col3 is string
converters = {'col1': [asciitable.convert_list(int)],
              'col2': [asciitable.convert_list(float)],
              'col3': [asciitable.convert_list(str)]}
read('file.dat', converters=converters)
```

Note that it is also possible to specify a list of converter functions that will be tried in order:

```
converters = {'col1': [asciitable.convert_list(float),
                       asciitable.convert_list(str)]}
read('file.dat', converters=converters)
```

5.5.2 With NumPy

If the `numpy` module is available then the `NumpyOutputter` is selected by default. In this case the default converters are:

```
default_converters = [asciitable.convert_numpy(numpy.int),
                      asciitable.convert_numpy(numpy.float),
                      asciitable.convert_numpy(numpy.str)]
```

These take advantage of the `convert_numpy()` function which returns a 2-element tuple `(converter_func, converter_type)` as described in the previous section. The type provided to `convert_numpy()` must be a valid numpy type, for example `numpy.int`, `numpy.uint`, `numpy.int8`, `numpy.int64`, `numpy.float`, `numpy.float64`, `numpy.str`.

The converters for each column can be specified with the `converters` keyword:

```
converters = {'col1': [asciitable.convert_numpy(numpy.uint)],
              'col2': [asciitable.convert_numpy(numpy.float32)]}
read('file.dat', converters=converters)
```

5.6 Advanced table reading

This section is not finished. It will discuss ways of making custom reader functions and how to write custom `Reader`, `Splitter`, `Inputter` and `Outputter` classes. For now please look at the examples and especially the code for the existing Extension Reader Classes.

5.6.1 Examples

Define a custom reader functionally

```
def read_rdb_table(table):
    reader = asciitable.Basic()
    reader.header.splitter.delimiter = '\t'
    reader.data.splitter.delimiter = '\t'
```

```
    reader.header.splitter.process_line = None
    reader.data.splitter.process_line = None
    reader.data.start_line = 2

    return reader.read(table)
```

Define custom readers by class inheritance

```
# Note: Tab and Rdb are already included in asciitable for convenience.
class Tab(asciitable.Basic):
    def __init__(self):
        asciitable.Basic.__init__(self)
        self.header.splitter.delimiter = '\t'
        self.data.splitter.delimiter = '\t'
        # Don't strip line whitespace since that includes tabs
        self.header.splitter.process_line = None
        self.data.splitter.process_line = None
        # Don't strip data value spaces since that is significant in TSV tables
        self.data.splitter.process_val = None
        self.data.splitter.skipinitialspace = False

class Rdb(asciitable.Tab):
    def __init__(self):
        asciitable.Tab.__init__(self)
        self.data.start_line = 2
```

Create a custom splitter.process_val function

```
# The default process_val() normally just strips whitespace.
# In addition have it replace empty fields with -999.
def process_val(x):
    """Custom splitter process_val function: Remove whitespace at the beginning
    or end of value and substitute -999 for any blank entries."""
    x = x.strip()
    if x == '':
        x = '-999'
    return x

# Create an RDB reader and override the splitter.process_val function
rdb_reader = asciitable.get_reader(Reader=asciitable.Rdb)
rdb_reader.data.splitter.process_val = process_val
```

CHAPTER

SIX

WRITING TABLES

`Asciitable` is able to write ASCII tables out to a file or file-like object using the same class structure and basic user interface as for reading tables.

As a very simple example:

```
x = np.array([1, 2, 3])
y = x**2
asciitable.write({'x': x, 'y': y}, 'outfile.dat', names=['x', 'y'])
```

6.1 Input data formats

A number of data formats for the input table are supported:

- Existing ASCII table with metadata (`BaseReader` object)
- Data from asciitable.read() (`DictLikeNumpy` object)
- NumPy structured array or record array
- Sequence of sequences (row-oriented list of lists)
- Dict of sequences (column oriented dictionary of lists)

6.1.1 Existing ASCII table with metadata

The example below highlights that the `get_reader()` function returns a `Reader` object that supports keywords and table metadata. The `Reader` object can then be an input to the `write()` function and allow for any associated metadata to be written.

Note that in the current release there is no support for actually writing the available keywords or other metadata, but the infrastructure is available and this is the top priority for development.

```
# Get a Reader object
table = asciitable.get_reader(Reader=asciitable.Daophot)

# Read a table from a file.  Return a NumPy record array object and also
# update column and metadata attributes in the "table" Reader object.
data = table.read('t/daophot.dat')

# Write the data in a variety of ways using as input both the NumPy record
# array and the higher-level Reader object.
asciitable.write(table, "table.dat", Writer=asciitable.Tab )
```

```
asciitable.write(table, open("table.dat", "w"), Writer=asciitable.NoHeader )
asciitable.write(table, sys.stdout, Writer=asciitable.CommentedHeader )
asciitable.write(table, sys.stdout, Writer=asciitable.Rdb, exclude_names=['CHI'] )

asciitable.write(table, sys.stdout, formats={'XCENTER': '%12.1f',
                                             'YCENTER': lambda x: round(x, 1)},
                                    include_names=['XCENTER', 'YCENTER'])
```

6.1.2 Data from asciitable.read()

`Asciitable.read` returns a data object that can be an input to the `write()` function. If NumPy is available the default data object type is a NumPy record array. However it is possible to use `asciitable` without NumPy in which case a `DictLikeNumpy` object is returned. This object supports the most basic column and row indexing API of a NumPy structured array. This object can be used as input to the `write()` function.

```
table = asciitable.get_reader(Reader=asciitable.Daophot, numpy=False)
data = table.read('t/daophot.dat')

asciitable.write(data, sys.stdout)
```

6.1.3 NumPy structured array

A NumPy structured array (aka record array) can serve as input to the `write()` function.

```
data = numpy.zeros((2,), dtype=('i4,f4,a10'))
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]
asciitable.write(data, sys.stdout)
```

6.1.4 Sequence of sequences

A doubly-nested structure of iterable objects (e.g. lists or tuples) can serve as input to `write()`. The outer layer represents rows while the inner layer represents columns.

```
data = [[1, 2,   3      ],
        [4, 5.2, 6.1    ],
        [8, 9,   'hello']]
asciitable.write(data, 'table.dat')
asciitable.write(data, 'table.dat', names=['x', 'y', 'z'], exclude_names=['y'])
```

6.1.5 Dict of sequences

A dictionary containing iterable objects can serve as input to `write()`. Each dict key is taken as the column name while the value must be an iterable object containing the corresponding column values. Note the difference in output between this example and the previous example.

```
data = {'x': [1, 2, 3],
        'y': [4, 5.2, 6.1],
        'z': [8, 9, 'hello world']}
asciitable.write(data, 'table.dat', names=['x', 'y', 'z'])
```

Specifying the `names` argument is necessary if the order of the columns matters. The specified values must match the keys in the `data` dict.

6.2 Commonly used parameters for `write()`

The `write()` function accepts a number of parameters that specify the detailed output table format. Different Reader classes can define different defaults, so the descriptions below sometimes mention "typical" default values. This refers to the `Basic` reader and other similar Reader classes.

Some Reader classes, e.g. `Latex` or `AASTex` accept aditional keywords, that can customize the output further. See the documentation of these classes for details.

output [output specifier] There are two ways to specify the output for the write operation:

- Name of a file (string)
- File-like object (from open(), StringIO, etc)

table [input table] The are five possible formats for the data table that is to be written:

- `Asciitable` Reader object (returned by `get_reader()`) which has been used to read a table
- Output from `read()` (`DictLikeNumpy`)
- NumPy structured array or record array
- List of lists: e.g. `[[2, 3], [4, 5], [6, 7]]` (3 rows, 2 columns)
- Dict of lists: e.g. `{'c1': [2, 3, 4], 'c2': [5, 6, 7]}` (3 rows, 2 columns)

Writer [Writer class (default= `Basic`)] This specifies the top-level format of the ASCII table to be written, for example if it is a basic character delimited table, fixed format table, or a CDS-compatible table, etc. The value of this parameter must be a Reader class. For basic usage this means one of the built-in Extension Reader Classes. Note: Reader classes and Writer classes are synonymous, in other words Reader classes can also write, but for historical reasons they are called Reader classes.

delimiter [column delimiter string] A one-character string used to separate fields which typically defaults to the space character. Other common values might be "," or "|" or "\t".

comment [string defining a comment line in table] For the `Basic` Reader this defaults to "#".

formats: dict of data type converters For each key (column name) use the given value to convert the column data to a string. If the format value is string-like then it is used as a Python format statement, e.g. '%0.2f' % value. If it is a callable function then that function is called with a single argument containing the column value to be converted. Example:

```
asciitable.write(table, sys.stdout, formats={'XCENTER': '%12.1f',
                                             'YCENTER': lambda x: round(x, 1)},
```

names: list of names corresponding to each data column Define the complete list of names for each data column. This will override names determined from the data table (if available), except in the case of input as a Dict of sequences where it specifies the *order* columns. If not supplied then use names from the data table or auto-generated names.

include_names: list of names to include in output From the list of column names found from the data table or the `names` parameter, select for output only columns within this list. If not supplied then include all names.

exclude_names: list of names to exlude from output Exclude these names from the list of output columns. This is applied *after* the `include_names` filtering. If not specified then no columns are excluded.

fill_values: fill value specifier of lists This can be used to fill missing values in the table or replace values with special meaning. The syntax is the same as used on input. See the Replace bad or missing values section for more information on the syntax. When writing a table, all values are converted to strings, before any value is replaced. Thus, you need to provide the string representation (stripped of whitespace) for each value. Example:

```
asciitable.write(table, sys.stdout, fill_values = [('nan', 'no data'),
                                                   ('-999.0', 'no data')])
```

fill_include_names: list of column names, which are affected by `fill_values`. If not supplied, then `fill_values` can affect all columns.

fill_exclude_names: list of column names, which are not affected by `fill_values`. If not supplied, then `fill_values` can affect all columns.

CHAPTER

SEVEN

BASE CLASS ELEMENTS

The key elements in `asciitable` are:

- `Column`: Internal storage of column properties and data ()
- `Inputter`: Get the lines from the table input.
- `Splitter`: Split the lines into string column values.
- `Header`: Initialize output columns based on the table header or user input.
- `Data`: Populate column data from the table.
- `Outputter`: Convert column data to the specified output format, e.g. NumPy structured array.

Each of these elements is an inheritable class with attributes that control the corresponding functionality. In this way the large number of tweakable parameters is modularized into managable groups. Where it makes sense these attributes are actually functions that make it easy to handle special cases.

CHAPTER

EIGHT

ASCIITABLE API

An extensible ASCII table reader and writer.

Copyright Smithsonian Astrophysical Observatory (2010)

Author Tom Aldcroft (aldcroft@head.cfa.harvard.edu)

8.1 Functions

asciitable.**read**(*table*, *numpy=True*, *guess=None*, ***kwargs*)

Read the input `table`. If `numpy` is True (default) return the table in a numpy record array. Otherwise return the table as a dictionary of column objects using plain python lists to hold the data. Most of the default behavior for various parameters is determined by the Reader class.

Parameters

- **table** – input table (file name, list of strings, or single newline-separated string)
- **numpy** – use the `NumpyOutputter` class else use `BaseOutputter` (default=True)
- **guess** – try to guess the table format (default=True)
- **Reader** – Reader class (default= `BasicReader`)
- **Inputter** – Inputter class
- **Outputter** – Outputter class
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)

- **exclude_names** – list of names to exlude from output (applied after `include_names`)
- **fill_values** – specification of fill values for bad or missing table values
- **fill_include_names** – list of names to include in fill_values (default=None selects all names)
- **fill_exclude_names** – list of names to exlude from fill_values (applied after `fill_include_names`)

`asciitable.`**`get_reader`**(*Reader=None*, *Inputter=None*, *Outputter=None*, *numpy=True*, ***kwargs*)

Initialize a table reader allowing for common customizations. Most of the default behavior for various parameters is determined by the Reader class.

Parameters

- **Reader** – Reader class (default= `BasicReader`)
- **Inputter** – Inputter class
- **Outputter** – Outputter class
- **numpy** – use the NumpyOutputter class else use BaseOutputter (default=True)
- **delimiter** – column delimiter string
- **comment** – regular expression defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **header_start** – line index for the header line not counting comment lines
- **data_start** – line index for the start of data not counting comment lines
- **data_end** – line index for the end of data (can be negative to count from end)
- **converters** – dict of converters
- **data_Splitter** – Splitter class to split data columns
- **header_Splitter** – Splitter class to split header columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)
- **fill_values** – specification of fill values for bad or missing table values
- **fill_include_names** – list of names to include in fill_values (default=None selects all names)
- **fill_exclude_names** – list of names to exlude from fill_values (applied after `fill_include_names`)

`asciitable.`**`write`**(*table*, *output=<open file '<stdout>', mode 'w' at 0x7fea176451e0>*, *Writer=None*, ***kwargs*)

Write the input `table` to `filename`. Most of the default behavior for various parameters is determined by the Writer class.

Parameters

- **table** – input table (Reader object, NumPy struct array, list of lists, etc)
- **output** – output [filename, file-like object] (default = sys.stdout)
- **Writer** – Writer class (default= `Basic`)
- **delimiter** – column delimiter string
- **write_comment** – string defining a comment line in table

- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)

`asciitable.`**`get_writer`**(*Writer=None*, ***kwargs*)

Initialize a table writer allowing for common customizations. Most of the default behavior for various parameters is determined by the Writer class.

Parameters

- **Writer** – Writer class (default= `Basic`)
- **delimiter** – column delimiter string
- **write_comment** – string defining a comment line in table
- **quotechar** – one-character string to quote fields containing special characters
- **formats** – dict of format specifiers or formatting functions
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)

`asciitable.`**`convert_list`**(*python_type*)

Return a tuple `(converter_func, converter_type)`. The converter function converts a list into a list of the given `python_type`. This argument is a function that takes a single argument and returns a single value of the desired type. In general this will be one of `int`, `float` or `str`. The converter type is used to track the generic data type (int, float, str) that is produced by the converter function.

`asciitable.`**`convert_numpy`**(*numpy_type*)

Return a tuple `(converter_func, converter_type)`. The converter function converts a list into a numpy array of the given `numpy_type`. This type must be a valid numpy type, e.g. numpy.int, numpy.uint, numpy.int8, numpy.int64, numpy.float, numpy.float64, numpy.str. The converter type is used to track the generic data type (int, float, str) that is produced by the converter function.

`asciitable.`**`set_guess`**(*guess*)

Set the default value of the `guess` parameter for read()

Parameters **guess** – New default `guess` value (TruelFalse)

8.2 Core Classes

class `asciitable.`**`BaseReader`**

Bases: `object`

Class providing methods to read an ASCII table using the specified header, data, inputter, and outputter instances.

Typical usage is to instantiate a Reader() object and customize the `header`, `data`, `inputter`, and `outputter` attributes. Each of these is an object of the corresponding class.

There is one method `inconsistent_handler` that can be used to customize the behavior of `read()` in the event that a data row doesn't match the header. The default behavior is to raise an InconsistentTableError.

comment_lines
Return lines in the table that match header.comment regexp

inconsistent_handler(*str_vals*, *ncols*)
Adjust or skip data entries if a row is inconsistent with the header.

The default implementation does no adjustment, and hence will always trigger an exception in read() any time the number of data entries does not match the header.

Note that this will *not* be called if the row already matches the header.

Parameters

- **str_vals** – A list of value strings from the current row of the table.
- **ncols** – The expected number of entries from the table header.

Returns list of strings to be parsed into data entries in the output table. If the length of this list does not match `ncols`, an exception will be raised in read(). Can also be None, in which case the row will be skipped.

read(*table*)
Read the `table` and return the results in a format determined by the `outputter` attribute.

The `table` parameter is any string or object that can be processed by the instance `inputter`. For the base Inputter class `table` can be one of:

•File name

•String (newline separated) with all header and data lines (must have at least 2 lines)

•List of strings

Parameters **table** – table input

Returns output table

write(*table=None*)
Write `table` as list of strings.

Parameters **table** – asciitable Reader object

Returns list of strings corresponding to ASCII table

class asciitable.**BaseData**
Bases: `object`

Base table data reader.

Parameters

- **start_line** – None, int, or a function of `lines` that returns None or int
- **end_line** – None, int, or a function of `lines` that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

comment = None

default_formatter
alias of `str`

end_line = None

fill_exclude_names = None

fill_include_names = None

fill_values = []

formats = {}

get_data_lines(*lines*)
Set the `data_lines` attribute to the lines slice comprising the table data values.

get_str_vals()
Return a generator that returns a list of column values (as strings) for each data line.

masks(*cols*)
Set fill value for each column and then apply that fill value

In the first step it is evaluated with value from `fill_values` applies to which column using `fill_include_names` and `fill_exclude_names`. In the second step all replacements are done for the appropriate columns.

process_lines(*lines*)
Strip out comment lines and blank lines from list of `lines`

Parameters **lines** – all lines in table

Returns list of lines

splitter_class
alias of `DefaultSplitter`

start_line = None

write(*lines*)

write_spacer_lines = ['ASCIITABLE_WRITE_SPACER_LINE']

class `asciitable.`**BaseHeader**
Bases: `object`

Base table header reader

Parameters

- **auto_format** – format string for auto-generating column names
- **start_line** – None, int, or a function of `lines` that returns None or int
- **comment** – regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after `include_names`)

auto_format = 'col%d'

colnames
Return the column names of the table

comment = None

exclude_names = None

get_col_type(*col*)

get_cols (*lines*)
Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns. This list only includes the actual requested columns after filtering by the include_names and exclude_names attributes. See `self.names` for the full list.

Parameters **lines** – list of table lines

Returns None

get_type_map_key (*col*)

include_names = None

n_data_cols
Return the number of expected data columns from data splitting. This is either explicitly set (typically for fixedwidth splitters) or set to self.names otherwise.

names = None

process_lines (*lines*)
Generator to yield non-comment lines

splitter_class
alias of `DefaultSplitter`

start_line = None

write (*lines*)

write_spacer_lines = ['ASCIITABLE_WRITE_SPACER_LINE']

class `asciitable.`**BaseInputter**
Bases: `object`

Get the lines from the table input and return a list of lines. The input table can be one of:

- File name
- String (newline separated) with all header and data lines (must have at least 2 lines)
- File-like object with read() method
- List of strings

get_lines (*table*)
Get the lines from the `table` input.

Parameters **table** – table input

Returns list of lines

process_lines (*lines*)
Process lines for subsequent use. In the default case do nothing. This routine is not generally intended for removing comment lines or stripping whitespace. These are done (if needed) in the header and data line processing.

Override this method if something more has to be done to convert raw input lines to the table rows. For example the ContinuationLinesInputter derived class accounts for continuation characters if a row is split into lines.

class `asciitable.`**BaseOutputter**
Bases: `object`

Output table as a dict of column objects keyed on column name. The table data are stored as plain python lists within the column objects.

converters = {}

default_converters = [(<function converter at 0x15c0c08>, <class 'asciitable.core.IntType'>), (<function converter a

class asciitable.**BaseSplitter**
Bases: object

Base splitter that uses python's split method to do the work.

This does not handle quoted values. A key feature is the formulation of __call__ as a generator that returns a list of the split line values at each iteration.

There are two methods that are intended to be overridden, first process_line() to do pre-processing on each input line before splitting and process_val() to do post-processing on each split string value. By default these apply the string strip() function. These can be set to another function via the instance attribute or be disabled entirely, for example:

```
reader.header.splitter.process_val = lambda x: x.lstrip()
reader.data.splitter.process_val = None
```

Parameters **delimiter** – one-character string used to separate fields

delimiter = None

join(*vals*)

process_line(*line*)
Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end.

process_val(*val*)
Remove whitespace at the beginning or end of value.

class asciitable.**Column**(*name*, *index*)
Bases: object

Table column.

The key attributes of a Column object are:

- **name** : column name
- **index** : column index (first column has index=0, second has index=1, etc)
- **type** : column type (NoType, StrType, NumType, FloatType, IntType)
- **str_vals** : list of column values as strings
- **data** : list of converted column values

class asciitable.**DefaultSplitter**
Bases: asciitable.core.BaseSplitter

Default class to split strings into columns using python csv. The class attributes are taken from the csv Dialect class.

Typical usage:

```
# lines = ..
splitter = asciitable.DefaultSplitter()
for col_vals in splitter(lines):
```

```
for col_val in col_vals:
    ...
```

Parameters

- **delimiter** – one-character string used to separate fields.
- **doublequote** – control how instances of *quotechar* in a field are quoted
- **escapechar** – character to remove special meaning from following character
- **quotechar** – one-character stringto quote fields containing special characters
- **quoting** – control when quotes are recognised by the reader
- **skipinitialspace** – ignore whitespace immediately following the delimiter

delimiter = ' '

doublequote = True

escapechar = None

join(*vals*)

process_line(*line*)
Remove whitespace at the beginning or end of line. This is especially useful for whitespace-delimited files to prevent spurious columns at the beginning or end. If splitting on whitespace then replace unquoted tabs with space first

process_val(*val*)
Remove whitespace at the beginning or end of value.

quotechar = '"'

quoting = 0

skipinitialspace = True

class asciitable.**DictLikeNumpy**(**args*, ***kwargs*)
Bases: dict

Provide minimal compatibility with numpy rec array API for BaseOutputter object:

```
table = asciitable.read('mytable.dat', numpy=False)
table.field('x')     # List of elements in column 'x'
table.dtype.names    # get column names in order
table[1]             # returns row 1 as a list
table[1][2]          # 3nd column in row 1
table['col1'][1]     # Row 1 in column col1
for row_vals in table:  # iterate over table rows
    print row_vals  # print list of vals in each row
```

class Dtype
Bases: object

DictLikeNumpy.**field**(*colname*)

DictLikeNumpy.**next**()

class asciitable.**InconsistentTableError**
Bases: exceptions.ValueError

class asciitable.**NumpyOutputter**

Bases: asciitable.core.BaseOutputter

Output the table as a numpy.rec.recarray

Missing or bad data values are handled at two levels. The first is in the data reading step where if data.fill_values is set then any occurences of a bad value are replaced by the correspond fill value. At the same time a boolean list mask is created in the column object.

The second stage is when converting to numpy arrays which by default generates masked arrays, if data.fill_values is set and plain arrays if it is not. In the rare case that plain arrays are needed set auto_masked (default = True) and default_masked (default = False) to control this behavior as follows:

auto_masked	default_masked	fill_values	output
–	True	–	masked_array
–	False	None	array
True	–	dict(..)	masked_array
False	–	dict(..)	array

To set these values use:

```
Outputter = asciitable.NumpyOutputter()
Outputter.default_masked = True
```

auto_masked_array = True

converters = {}

default_converters = [(<function converter at 0x15c0c08>, <class 'asciitable.core.IntType'>), (<function converter a

default_masked_array = False

8.3 Extension Reader Classes

The following classes extend the base Reader functionality to handle different table formats. Some, such as the Basic Reader class are fairly general and include a number of configurable attributes. Others such as Cds or Daophot are specialized to read certain well-defined but idiosyncratic formats.

- AASTex: AASTeX *deluxetable* used for AAS journals
- Basic: basic table with customizable delimiters and header configurations
- Cds: CDS format table (also Vizier and ApJ machine readable tables)
- CommentedHeader: column names given in a line that begins with the comment character
- Daophot: table from the IRAF DAOphot package
- FixedWidth: table with fixed-width columns (see also *Fixed-width Gallery*)
- FixedWidthNoHeader: table with fixed-width columns and no header
- FixedWidthTwoLine: table with fixed-width columns and a two-line header
- Ipac: IPAC format table
- Latex: LaTeX table with datavalue in the *tabular* environment
- NoHeader: basic table with no header where columns are auto-named
- Rdb: tab-separated values with an extra line after the column definition line
- Tab: tab-separated values

class asciitable.**AASTex**(***kwargs*)
Bases: asciitable.latex.Latex

Write and read AASTeX tables.

This class implements some AASTeX specific commands. AASTeX is used for the AAS (American Astronomical Society) publications like ApJ, ApJL and AJ.

It derives from Latex and accepts the same keywords (see Latex for documentation). However, the keywords header_start, header_end, data_start and data_end in latexdict have no effect.

class asciitable.**Basic**
Bases: asciitable.core.BaseReader

Read a character-delimited table with a single header line at the top followed by data lines to the end of the table. Lines beginning with # as the first non-whitespace character are comments. This reader is highly configurable.

```
rdr = asciitable.get_reader(Reader=asciitable.Basic)
rdr.header.splitter.delimiter = ' '
rdr.data.splitter.delimiter = ' '
rdr.header.start_line = 0
rdr.data.start_line = 1
rdr.data.end_line = None
rdr.header.comment = r'\s*#'
rdr.data.comment = r'\s*#'
```

Example table:

```
# Column definition is the first uncommented line
# Default delimiter is the space character.
apples oranges pears

# Data starts after the header column definition, blank lines ignored
1 2 3
4 5 6
```

class asciitable.**Cds**(*readme=None*)
Bases: asciitable.core.BaseReader

Read a CDS format table: http://vizier.u-strasbg.fr/doc/catstd.htx. Example:

```
Table: Spitzer-identified YSOs: Addendum
================================================================================
Byte-by-byte Description of file: datafile3.txt
--------------------------------------------------------------------------------
   Bytes Format Units  Label  Explanations
--------------------------------------------------------------------------------
   1-  3 I3     ---    Index  Running identification number
   5-  6 I2     h      RAh    Hour of Right Ascension (J2000)
   8-  9 I2     min    RAm    Minute of Right Ascension (J2000)
  11- 15 F5.2   s      RAs    Second of Right Ascension (J2000)
--------------------------------------------------------------------------------
  1 03 28 39.09
```

Basic usage

Use the asciitable.read() function as normal, with an optional readme parameter indicating the CDS ReadMe file. If not supplied it is assumed that the header information is at the top of the given table. Examples:

```
>>> import asciitable
>>> table = asciitable.read("t/cds.dat")
>>> table = asciitable.read("t/vizier/table1.dat", readme="t/vizier/ReadMe")
```

```
>>> table = asciitable.read("t/cds/multi/lhs2065.dat", readme="t/cds/multi/ReadMe")
>>> table = asciitable.read("t/cds/glob/lmxbrefs.dat", readme="t/cds/glob/ReadMe")
```

Using a reader object

When `Cds` reader object is created with a `readme` parameter passed to it at initialization, then when the `read` method is executed with a table filename, the header information for the specified table is taken from the `readme` file. An `InconsistentTableError` is raised if the `readme` file does not have header information for the given table.

```
>>> readme = "t/vizier/ReadMe"
>>> r = asciitable.get_reader(asciitable.Cds, readme=readme)
>>> table = r.read("t/vizier/table1.dat")
>>> # table5.dat has the same ReadMe file
>>> table = r.read("t/vizier/table5.dat")
```

If no `readme` parameter is specified, then the header information is assumed to be at the top of the given table.

```
>>> r = asciitable.get_reader(asciitable.Cds)
>>> table = r.read("t/cds.dat")
>>> #The following gives InconsistentTableError, since no
>>> #readme file was given and table1.dat does not have a header.
>>> table = r.read("t/vizier/table1.dat")
Traceback (most recent call last):
  ...
InconsistentTableError: No CDS section delimiter found
```

Caveats:

- Format, Units, and Explanations are available in the `Reader.cols` attribute.
- All of the other metadata defined by this format is ignored.

Code contribution to enhance the parsing to include metadata in a Reader.meta attribute would be welcome.

write (*table=None*)
Not available for the Cds class (raises NotImplementedError)

class asciitable.**CommentedHeader**
Bases: `asciitable.core.BaseReader`

Read a file where the column names are given in a line that begins with the header comment character. The default delimiter is the <space> character.:

```
# col1 col2 col3
# Comment line
1 2 3
4 5 6
```

class asciitable.**Daophot**
Bases: `asciitable.core.BaseReader`

Read a DAOphot file. Example:

```
#K MERGERAD   = INDEF                   scaleunit  %-23.7g
#K IRAF = NOAO/IRAFV2.10EXPORT version %-23s
#K USER = davis name %-23s
#K HOST = tucana computer %-23s
#
#N ID    XCENTER   YCENTER   MAG         MERR          MSKY           NITER    \
#U ##    pixels    pixels    magnitudes  magnitudes    counts         ##       \
#F %-9d  %-10.3f   %-10.3f   %-12.3f     %-14.3f       %-15.7g        %-6d
```

```
#
#N         SHARPNESS   CHI         PIER  PERROR                                \
#U         ##          ##          ##    perrors                               \
#F         %-23.3f     %-12.3f     %-6d  %-13s
#
14       138.538   256.405   15.461      0.003         34.85955       4        \
-0.032      0.802       0     No_error
```

The keywords defined in the #K records are available via the Daophot reader object:

```
reader = asciitable.get_reader(Reader=asciitable.DaophotReader)
data = reader.read('t/daophot.dat')
for keyword in reader.keywords:
    print keyword.name, keyword.value, keyword.units, keyword.format
```

read(*table*)

write(*table=None*)

class asciitable.**FixedWidth**(*col_starts=None*, *col_ends=None*, *delimiter_pad=' '*, *bookend=True*)
Bases: asciitable.core.BaseReader

Read or write a fixed width table with a single header line that defines column names and positions. Examples:

```
# Bar delimiter in header and data

|  Col1 |   Col2      |  Col3 |
|  1.2  | hello there |     3 |
|  2.4  | many words  |     7 |

# Bar delimiter in header only

Col1 |   Col2      | Col3
1.2    hello there    3
2.4    many words     7

# No delimiter with column positions specified as input

Col1       Col2Col3
 1.2hello there   3
 2.4many words    7
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

class asciitable.**FixedWidthNoHeader**(*col_starts=None*, *col_ends=None*, *delimiter_pad=' '*, *bookend=True*)
Bases: asciitable.fixedwidth.FixedWidth

Read or write a fixed width table which has no header line. Column names are either input (names keyword) or auto-generated. Column positions are determined either by input (col_starts and col_stops keywords) or by splitting the first data line. In the latter case a delimiter is required to split the data line.

Examples:

```
# Bar delimiter in header and data

|  1.2  | hello there |     3 |
|  2.4  | many words  |     7 |

# Compact table having no delimiter and column positions specified as input

1.2hello there3
2.4many words 7
```

This class is just a convenience wrapper around `FixedWidth` but with `header.start_line = None` and `data.start_line = 0`.

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

class asciitable.**FixedWidthTwoLine**(*position_line=1*, *position_char='-'*, *delimiter_pad=None*, *bookend=False*)

Bases: asciitable.fixedwidth.FixedWidth

Read or write a fixed width table which has two header lines. The first header line defines the column names and the second implicitly defines the column positions. Examples:

```
# Typical case with column extent defined by ---- under column names.

 col1    col2         <== header_start = 0
-----  ------------   <== position_line = 1, position_char = "-"
  1     bee flies     <== data_start = 2
  2     fish swims

# Pretty-printed table

+------+------------+
| Col1 |   Col2     |
+------+------------+
|  1.2 | "hello"    |
|  2.4 | there world|
+------+------------+
```

See the *Fixed-width Gallery* for specific usage examples.

Parameters

- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = "-")
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

class asciitable.**Ipac**

Bases: asciitable.core.BaseReader

Read an IPAC format table: http://irsa.ipac.caltech.edu/applications/DDGEN/Doc/ipac_tbl.html:

```
\name=value
\ Comment
|  column1 |  column2 | column3 | column4  |    column5       |
|  double  |  double  |   int   |   double |     char         |
|   unit   |   unit   |   unit  |    unit  |     unit         |
|   null   |   null   |   null  |    null  |     null         |
 2.0978     29.09056   73765     2.06000    B8IVpMnHg
```

Or:

```
|-----ra---|----dec---|---sao---|------v---|----sptype--------|
  2.09708   29.09056     73765    2.06000   B8IVpMnHg
```

Caveats:

- Data type, Units, and Null value specifications are ignored.
- Keywords are ignored.
- The IPAC spec requires the first two header lines but this reader only requires the initial column name definition line

Overcoming these limitations would not be difficult, code contributions welcome from motivated users.

write (*table=None*)
Not available for the Ipac class (raises NotImplementedError)

class asciitable.**Latex**(*ignore_latex_commands=['hline', 'vspace', 'tableline'], latexdict={}, caption='', col_align=None*)
Bases: asciitable.core.BaseReader

Write and read LaTeX tables.

This class implements some LaTeX specific commands. Its main purpose is to write out a table in a form that LaTeX can compile. It is beyond the scope of this class to implement every possible LaTeX command, instead the focus is to generate a syntactically valid LaTeX tables. This class can also read simple LaTeX tables (one line per table row, no \multicolumn or similar constructs), specifically, it can read the tables that it writes.

Reading a LaTeX table, the following keywords are accepted:

ignore_latex_commands [] Lines starting with these LaTeX commands will be treated as comments (i.e. ignored).

When writing a LaTeX table, the some keywords can customize the format. Care has to be taken here, because python interprets \ in a string as an escape character. In order to pass this to the output either format your strings as raw strings with the r specifier or use a double \\. Examples:

```
caption = r'My table \label{mytable}'
caption = 'My table \\label{mytable}'
```

latexdict [Dictionary of extra parameters for the LaTeX output]

- **tabletype** [used for first and last line of table.] The default is \begin{table}. The following would generate a table, which spans the whole page in a two-column document:

```
asciitable.write(data, sys.stdout, Writer = asciitable.Latex,
                 latexdict = {'tabletype': 'table*'})
```

- **col_align** [Alignment of columns] If not present all columns will be centered.

- **caption** [Table caption (string or list of strings)] This will appear above the table as it is the standard in many scientific publications. If you prefer a caption below the table, just write the full LaTeX command as `latexdict['tablefoot'] = r'\caption{My table}'`
- **preamble, header_start, header_end, data_start, data_end, tablefoot: Pure LaTeX** Each one can be a string or a list of strings. These strings will be inserted into the table without any further processing. See the examples below.
- **units** [dictionary of strings] Keys in this dictionary should be names of columns. If present, a line in the LaTeX table directly below the column names is added, which contains the values of the dictionary. Example:

  import asciitable import asciitable.latex import sys data = {'name': ['bike', 'car'], 'mass': [75,1200], 'speed': [10, 130]} asciitable.write(data, sys.stdout, Writer = asciitable.Latex,

  latexdict = {'units': {'mass': 'kg', 'speed': 'km/h'}})

  If the column has no entry in the *units* dictionary, it defaults to ' '.

Run the following code to see where each element of the dictionary is inserted in the LaTeX table:

```
import asciitable
import asciitable.latex
import sys
data = {'cola': [1,2], 'colb': [3,4]}
asciitable.write(data, sys.stdout, Writer = asciitable.Latex,
                 latexdict = asciitable.latex.latexdicts['template'])
```

Some table styles are predefined in the dictionary `asciitable.latex.latexdicts`. The following generates in table in style preferred by A&A and some other journals:

```
asciitable.write(data, sys.stdout, Writer = asciitable.Latex,
                 latexdict = asciitable.latex.latexdicts['AA'])
```

As an example, this generates a table, which spans all columns and is centered on the page:

```
asciitable.write(data, sys.stdout, Writer = asciitable.Latex,
                 col_align = '|lr|',
                 latexdict = {'preamble': r'egin{center}', 'tablefoot': r'\end{center}',
                              'tabletype': 'table*'})
```

caption [Set table caption] Shorthand for:

```
latexdict['caption'] = caption
```

col_align [Set the column alignment.] If not present this will be auto-generated for centered columns. Shorthand for:

```
latexdict['col_align'] = col_align
```

write(*table=None*)

class asciitable.**Memory**

Bases: asciitable.core.BaseReader

Read a table from a data object in memory. Several input data formats are supported:

Output of asciitable.read():

```
table = asciitable.get_reader(Reader=asciitable.Daophot)
data = table.read('t/daophot.dat')
```

```
mem_data_from_table = asciitable.read(table, Reader=asciitable.Memory)
mem_data_from_data = asciitable.read(data, Reader=asciitable.Memory)
```

Numpy structured array:

```
data = numpy.zeros((2,), dtype=[('col1','i4'), ('col2','f4'), ('col3', 'a10')])
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]
mem_data = asciitable.read(data, Reader=asciitable.Memory)
```

Numpy masked structured array:

```
data = numpy.ma.zeros((2,), dtype=[('col1','i4'), ('col2','f4'), ('col3', 'a10')])
data[:] = [(1, 2., 'Hello'), (2, 3., "World")]
data['col2'] = ma.masked
mem_data = asciitable.read(data, Reader=asciitable.Memory)

In the current version all masked values will be converted to nan.
```

Sequence of sequences:

```
data = [[1, 2,   3      ],
        [4, 5.2, 6.1    ],
        [8, 9,   'hello']]
mem_data = asciitable.read(data, Reader=asciitable.Memory, names=('c1','c2','c3'))
```

Dict of sequences:

```
data = {'c1': [1, 2, 3],
        'c2': [4, 5.2, 6.1],
        'c3': [8, 9, 'hello']}
mem_data = asciitable.read(data, Reader=asciitable.Memory, names=('c1','c2','c3'))
```

read(*table*)

write(*table=None*)
Not available for the Memory class (raises NotImplementedError)

class asciitable.**NoHeader**
Bases: asciitable.basic.Basic

Read a table with no header line. Columns are autonamed using header.auto_format which defaults to "col%d". Otherwise this reader the same as the Basic class from which it is derived. Example:

```
# Table data
1 2 "hello there"
3 4 world
```

class asciitable.**Rdb**
Bases: asciitable.basic.Tab

Read a tab-separated file with an extra line after the column definition line. The RDB format meets this definition. Example:

```
col1 <tab> col2 <tab> col3
N <tab> S <tab> N
1 <tab> 2 <tab> 5
```

In this reader the second line is just ignored.

class asciitable.**Tab**
Bases: asciitable.basic.Basic

Read a tab-separated file. Unlike the `Basic` reader, whitespace is not stripped from the beginning and end of lines. By default whitespace is still stripped from the beginning and end of individual column values.

Example:

```
col1 <tab> col2 <tab> col3
# Comment line
1 <tab> 2 <tab> 5
```

8.4 Other extension classes

These classes provide support for extension readers.

class `asciitable.cds.`**`CdsData`**
Bases: `asciitable.core.BaseData`

CDS table data reader

`process_lines`(*lines*)
Skip over CDS header by finding the last section delimiter

`splitter_class`
alias of `FixedWidthSplitter`

class `asciitable.cds.`**`CdsHeader`**(*readme=None*)
Bases: `asciitable.core.BaseHeader`

`col_type_map` = {'i': <class 'asciitable.core.IntType'>, 'a': <class 'asciitable.core.StrType'>, 'e': <class 'asciitable.core.

`get_cols`(*lines*)
Initialize the header Column objects from the table `lines` for a CDS header.

Parameters **lines** – list of table lines

Returns list of table Columns

`get_type_map_key`(*col*)

class `asciitable.basic.`**`CommentedHeaderHeader`**
Bases: `asciitable.core.BaseHeader`

Header class for which the column definition line starts with the comment character. See the `CommentedHeader` class for an example.

`process_lines`(*lines*)
Return only lines that start with the comment regexp. For these lines strip out the matching characters.

`write`(*lines*)

class `asciitable.`**`ContinuationLinesInputter`**
Bases: `asciitable.core.BaseInputter`

Inputter where lines ending in `continuation_char` are joined with the subsequent line. Example:

```
col1 col2 col3
1       2 3
4 5       6
```

`continuation_char` = '\\'

`process_lines`(*lines*)

class asciitable.daophot.**DaophotHeader**
Bases: asciitable.core.BaseHeader

Read the header from a file produced by the IRAF DAOphot routine.

get_cols(*lines*)
Initialize the header Column objects from the table lines for a DAOphot header. The DAOphot header is specialized so that we just copy the entire BaseHeader get_cols routine and modify as needed.

Parameters **lines** – list of table lines

Returns list of table Columns

class asciitable.**FixedWidthSplitter**
Bases: asciitable.core.BaseSplitter

Split line based on fixed start and end positions for each col in self.cols.

This class requires that the Header class will have defined col.start and col.end for each column. The reference to the header.cols gets put in the splitter object by the base Reader.read() function just in time for splitting data lines by a data object.

Note that the start and end positions are defined in the pythonic style so line[start:end] is the desired substring for a column. This splitter class does not have a hook for process_lines since that is generally not useful for fixed-width input.

bookend = False

delimiter_pad = ''

join(*vals*, *widths*)

class asciitable.**FixedWidthHeader**
Bases: asciitable.core.BaseHeader

Fixed width table header reader.

The key settable class attributes are:

Parameters

- **auto_format** – format string for auto-generating column names
- **start_line** – None, int, or a function of lines that returns None or int
- **comment** – regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns
- **names** – list of names corresponding to each data column
- **include_names** – list of names to include in output (default=None selects all names)
- **exclude_names** – list of names to exlude from output (applied after include_names)
- **position_line** – row index of line that specifies position (default = 1)
- **position_char** – character used to write the position line (default = "-")
- **col_starts** – list of start positions for each column (0-based counting)
- **col_ends** – list of end positions (inclusive) for each column
- **delimiter_pad** – padding around delimiter when writing (default = None)
- **bookend** – put the delimiter at start and end of line when writing (default = False)

get_cols (*lines*)
Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns. This list only includes the actual requested columns after filtering by the include_names and exclude_names attributes. See `self.names` for the full list.

Parameters **lines** – list of table lines

Returns None

get_fixedwidth_params (*line*)
Split `line` on the delimiter and determine column values and column start and end positions. This might include null columns with zero length (e.g. for header row = "| col1 || col2 | col3 |" or header2_row = "—— ———- ——"). The null columns are stripped out. Returns the values between delimiters and the corresponding start and end positions.

Parameters **line** – input line

Returns (vals, starts, ends)

get_line (*lines*, *index*)

position_line = None

write (*lines*)

class asciitable.**FixedWidthData**
Bases: `asciitable.core.BaseData`

Base table data reader.

Parameters

- **start_line** – None, int, or a function of `lines` that returns None or int
- **end_line** – None, int, or a function of `lines` that returns None or int
- **comment** – Regular expression for comment lines
- **splitter_class** – Splitter class for splitting data lines into columns

splitter_class
alias of `FixedWidthSplitter`

write (*lines*)

class asciitable.ipac.**IpacData**
Bases: `asciitable.core.BaseData`

IPAC table data reader

comment = '[\\\\]'

splitter_class
alias of `FixedWidthSplitter`

class asciitable.ipac.**IpacHeader**
Bases: `asciitable.core.BaseHeader`

IPAC table header

col_type_map = {'real': <class 'asciitable.core.FloatType'>, 'int': <class 'asciitable.core.IntType'>, 'float': <class 'ascii

comment = '\\\\'

get_cols(*lines*)
Initialize the header Column objects from the table `lines`.

Based on the previously set Header attributes find or create the column names. Sets `self.cols` with the list of Columns. This list only includes the actual requested columns after filtering by the include_names and exclude_names attributes. See `self.names` for the full list.

Parameters **lines** – list of table lines

Returns list of table Columns

process_lines(*lines*)
Generator to yield IPAC header lines, i.e. those starting and ending with delimiter character.

splitter_class
alias of `BaseSplitter`

class asciitable.latex.**LatexHeader**
Bases: `asciitable.core.BaseHeader`

header_start = '\\begin{tabular}'

start_line(*lines*)

write(*lines*)

class asciitable.latex.**LatexData**
Bases: `asciitable.core.BaseData`

data_end = '\\end{tabular}'

data_start = None

end_line(*lines*)

start_line(*lines*)

write(*lines*)

class asciitable.latex.**LatexSplitter**
Bases: `asciitable.core.BaseSplitter`

Split LaTeX table date. Default delimiter is *&*.

delimiter = '&'

join(*vals*)
Join values together and add a few extra spaces for readability

process_line(*line*)
Remove whitespace at the beginning or end of line. Also remove at end of line

process_val(*val*)
Remove whitespace and {} at the beginning or end of value.

class asciitable.latex.**AASTexHeader**
Bases: `asciitable.latex.LatexHeader`

In a *deluxetable* some header keywords differ from standard LaTeX.

This header is modified to take that into account.

header_start = '\\tablehead'

start_line(*lines*)

write(*lines*)

class asciitable.latex.**AASTexData**
Bases: asciitable.latex.LatexData

In a *deluxetable* the data is enclosed in *startdata* and *enddata*

data_end = '\\enddata'

data_start = '\\startdata'

start_line(*lines*)

write(*lines*)

class asciitable.latex.**AASTexHeaderSplitter**
Bases: asciitable.latex.LatexSplitter

extract column names from a *deluxetable*

This splitter expects the following LaTeX code **in a single line**:

ablehead{colhead{col1} & ... & colhead{coln}}

join(*vals*)

process_line(*line*)
extract column names from tablehead

CHAPTER

NINE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX