ATpy

Release 0.9.7

Eli Bressert and Thomas Robitaille

July 30, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

ATpy is a high-level Python package providing a way to manipulate tables of astronomical data in a uniform way. The two main features of ATpy are:

- It provides a Table class that contains data stored in a NumPy structured array, along with meta-data to describe the columns, and methods to manipulate the table (e.g. adding/removing/renaming columns, selecting rows, changing values).
- It provides built-in support for reading and writing to several common file/database formats, including FITS, VO, and IPAC tables, and SQLite, MySQL and PostgreSQL databases, with a very simple API.

In addition, ATpy provides a TableSet class that can be used to contain multiple tables, and supports reading and writing to file/database formats that support this (FITS, VO, and SQL databases).

Finally, ATpy provides support for user-written read/write functions for file/database formats not supported by default. We encourage users to send us custom read/write functions to read commonly used formats, and would be happy to integrate them into the main distribution.

CHAPTER

TWO

OBTAINING AND INSTALLING

2.1 Requirements

ATpy requires the following:

- Python 2.6 or later
- Numpy 1.5 or later
- Astropy 0.2 or later

The following packages are optional, but are required to read/write to certain formats:

- h5py 1.3.0 or later (for HDF5 tables)
- MySQL-python 1.2.2 or later (for MySQL tables)
- PyGreSQL 3.8.1 or later (for PostGreSQL tables)

2.2 Stable version

The latest stable release of ATpy can be downloaded from PyPI. To install ATpy, use the standard installation procedure:

```
tar xvzf ATpy-X-X.X.tar.gz
cd ATpy-X.X.X/
python setup.py install
```

2.3 Developer version

Advanced users wishing to use the latest development ("unstable") version can check it out with:

```
git clone git://github.com/atpy/atpy.git
```

which can then be installed with:

```
cd atpy
python setup.py install
```

CHAPTER

THREE

CONSTRUCTING A TABLE

The `Table` class is the basic entity in ATpy. It consists of table data and metadata. The data is stored using a NumPy structured array. The metadata includes units, null values, and column descriptions, as well as comments and keywords.

Data can be stored in the table using many of the NumPy types, including booleans, 8, 16, 32, and 64-bit signed and unsigned integers, 32 and 64-bit floats, and strings. Not all file formats and databases support reading and writing all of these types – for more information, see *Supported Formats*.

3.1 Creating a table

The simplest way to create an instance of the `Table` is to call the class with no arguments:

```
>>> t = atpy.Table()
```

3.2 Populating the table

A table can be populated either manually or by reading data from a file or database. Reading data into a table erases previous content. Data can be manually added once a table has been read in from a file.

3.2.1 Reading data from a file

The `read(...)` method can be used to read in a table from a file. To date, ATpy supports the following file formats:

- FITS tables (`type=fits`)
- VO tables (`type=vo`)
- IPAC tables (`type=ipac`)
- HDF5 (`type=hdf5`)

Now that ATpy has integrates with asciitable, the following formats are also supported:

- CDS (`type=cds` or `type=mrt`)
- DAOPhot (`type=daophot`)
- RDB (`type=rdb`)
- Arbitrary ASCII tables (`type=ascii`)

When reading a table from a file, the only required argument is the filename. For example, to read a VO table called `example.xml`, the following should be used:

```
>>> t.read('example.xml')
Auto-detected input type: VO table
```

The `read()` method will in most cases correctly identify the format of the file from the extension. As seen above, the default behavior is to specifically tell the user what format is being assumed, but this can be controlled via the `verbose` argument.

In some cases, `read()` will fail to determine the input type. In this case, or to override the automatically selected type, the input type can be specified using the `type` argument:

```
>>> t.read('example.xml', type='vo')
```

The `read` method supports additional file-format-dependent options. These are described in more detail in *Supported Formats*.

In cases where multiple tables are available in a table file, ATpy will display a message to the screen with instructions of how to specify which table to read in. Alternatively, see *Table Sets* for information on how to read all tables into a single `TableSet` instance.

As a convenience, it is possible to create a `Table` instance and read in data in a single command:

```
>>> t = Table('example.xml')
```

Any arguments given to `Table` are passed on to the `read` method, so the above is equivalent to:

```
>>> t = Table()
>>> t.read('example.xml')
```

As of 0.9.6, it is now possible to specify URLs starting with `http://` or `ftp://` and the file will automatically be downloaded. Furthermore, it is possible to specify files compressed in gzip or bzip format for all I/O formats.

3.2.2 Reading data from a database

Reading a table from a database is very similar to reading a table from a file. The main difference is that for databases, the first argument should be the database type, To date, ATpy supports the following database types:

- SQLite (`sqlite`)
- MySQL (`mysql`)
- PostGreSQL (`postgres`)

The remaining arguments depend on the database type. For example, an SQLite database can be read by specifying the database filename:

```
>>> t.read('sqlite','example.db')
```

For MySQL and PostGreSQL databases, it is possible to specify the database, table, authentication, and host parameters. The various options are descried in more detail in *Supported Formats*. As for files, the `verbose` and `type` arguments can be used.

As for reading in from files, one can read in data from a database while initializing the `Table` object:

```
>>> t = Table('sqlite','example.db')
```

Note: It is possible to specify a full SQL query using the `query` argument. Any valid SQL is allowed. If this is used, the table name should nevertheless be specified using the `table` argument.

3.2.3 Adding columns to a table

It is possible to add columns to an empty or an existing table. Two methods exist for this. The first, `add_column`, allows users to add an existing array to a column. For example, the following can be used to add a column named `time` where the variable `time_array` is a NumPy array:

```
>>> t.add_column('time', time_array)
```

The `add_column` method also optionally takes metadata about the column, such as units, or a description. For example:

```
>>> t.add_column('time', time_array, unit='seconds')
```

indicates that the units of the column are seconds. It is also possible to convert the datatype of an array while adding it to a table by using the `dtype` argument. For example, the following stores the column from the above examples as 32-bit floating point values:

```
>>> t.add_column('time', time_array, unit='seconds', dtype=np.float32)
```

In some cases, it is desirable to add an empty column to a table, and populate it element by element. This can be done using the `add_empty_column` method. The only required arguments for this method are the name and the data type of the column:

```
>>> t.add_empty_column('id', np.int16)
```

If the column is the first one being added to an empty table, the `shape` argument should be used to specify the number of rows. This should either be an integer giving the number of rows, or a tuple in the case of vector columns (see *Vector Columns* for more details)

3.2.4 Vector Columns

As well as using one-dimensional columns is also possible to specify so-called vector columns, which are essentially two-dimensional arrays. Only FITS and VO tables support reading and writing these. The `add_column` method accepts two-dimensional arrays as input, and uses these to define vector columns. Empty vector columns can be created by using the `add_empty_column` method along with the `shape` argument to specify the full shape of the column. This should be a tuple of the form `(n_rows, n_elements)`.

3.2.5 Writing the data to a file

Writing data to files or databases is done through the `write` method. The arguments to this method are very similar to that of the `read` data. The only main difference is that the `write` method can take an `overwrite` argument that specifies whether or not to overwrite existing files.

3.3 Adding meta-data

Comments and keywords can be added to a table using the `add_comment()` and `add_keyword()` methods:

```
>>> t.add_comment("This is a great table")
>>> t.add_keyword("meaning", 42)
```

CHAPTER

FOUR

ACCESSING TABLE DATA

4.1 Accessing the data

The table data is stored in a NumPy structured array, which can be accessed by passing the column name a key. This returns the column in question as a NumPy array:

```
t['column_name']
```

For convenience, columns with names that satisfy the python variable name requirements (essentially starting with a letter and containing no symbols apart from underscores) can be accessed directly as attributes of the table:

```
t.column_name
```

Since the returned data is a NumPy array, individual elements can be accessed using:

```
t['column_name'][row_number]
```

or:

```
t.column_name[row_number]
```

Both notations can be used to set data in the table, for example:

```
t.column_name[row_number] = 1
```

and:

```
t['column_name'][row_number] = 1
```

are equivalent, and will set the element at `row_number` to 1

4.2 Accessing the metadata

The column metadata is stored in the `columns` attribute. To see an overview of the metadata, simply use:

```
>>> t.columns
```

The metadata for a specific column can then be accessed by specifying the column name as a key:

```
>>> t.columns['some_column']
```

or using the column number:

```
>>> t.columns[column_number]
```

The attributes of a column object are `dtype`, `unit`, `description`, `null`, and `format`.

Note: While the unit, description and format for a column can be modified using the columns attribute, the dtype and null values should not be modified in this way as the changes will not propagate to the data array.

It is also possible to view a description of the table by using the `describe` method of the `Table` instance:

```
>>> t.describe()
```

In addition to the column metadata, the comments and keywords are available via the `keywords` and `comments` attributes of the `Table` instance, for example:

```
>>> instrument = t.keywords['instrument']
```

The `keywords` attribute is a dictionary, and the `comments` attribute is a list.

4.3 Accessing table rows

The `row(...)` method can be used to access a specific row in a table:

```
>>> row = t.row(row_number)
```

This returns the row as a NumPy record. The row can instead be returned as a tuple of elements with Python types, by using the `python_types` argument:

```
>>> row = t.row(row_number, python_types=True)
```

Two more powerful methods are available: `rows` and `where`. The `rows` method can be used to retrieve specific rows from a table as a new `Table` instance:

```
>>> t_new = t.rows([1,3,5,2,7,8])
```

Alternatively, the `where` method can be given a boolean array to determine which rows should be selected. This is in fact very powerful as the boolean array can actually be written as selection conditions:

```
>>> t_new = t.where((t.id > 10) & (t.ra < 45.4) & (t.flag == 'ok'))
```

4.4 Global Table properties

One can access the number of rows in a table by using the python `len` function:

```
>>> len(t)
```

In addition, the number of rows and columns can also be accessed with the `shape` attribute:

```
>>> t.shape
```

where the first number is the number of rows, and the second is the number of columns (note that a vector column counts as a single column).

CHAPTER

FIVE

MODIFYING TABLES

5.1 Manipulating table columns

Columns can be renamed or removed. To do this, one can use the `remove_column`, `remove_columns`, `keep_columns` and `rename_column` methods. For example, to rename a column `time` to `space`, one can use:

```
>>> t.rename_column('time','space')
```

The `keep_columns` essentially acts in the opposite way to `remove_columns` - it is used to specify which subset of the columns to not remove, which can be useful for extracting specific columns from a large table. For more information, see the *Full API for Table class*.

5.2 Sorting tables

To sort a table, use the `sort()` method, along with the name of the column to sort by:

```
>>> t.sort('time')
```

5.3 Combining tables

Given two `Table` instances with the same column metadata, and the same number of columns, one table can be added to the other via the `append` method:

```
>>> t1 = Table(...)
>>> t2 = Table(...)
>>> t1.append(t2)
```

CHAPTER

SIX

TABLE SETS

A `TableSet` instance contains a Python list of individual instances of the `Table` class. The advantage of using a `TableSet` instead of building a Python list of `Table` instances manually is that ATpy allows reading and writing of groups of tables to file formats that support it (e.g. FITS and VO table files or SQL databases).

6.1 Initialization

The easiest way to create a table set object is to call the `TableSet` class with no arguments:

```
tset = TableSet()
```

6.2 Manually adding a table to a set

An instance of the `Table` class can be added to a set by using the `append()` method:

```
tset.append(t)
```

where `t` is an instance of the `Table()` class.

6.3 Reading in tables from a file or database

The `read()` method can be used to read in multiple tables from a file or database. This method automatically determines the file or database type and reads in the tables. For example, all the tables in a VO table can be read in using:

```
tset.read('somedata.xml')
```

while all the tables in a FITS file can be read in using:

```
tset.read('somedata.fits')
```

As for the `Table()` class, in some cases, `read()` will fail to determine the input type. In this case, or to override the automatically selected type, the input type can be specified using the type argument:

```
tset.read('somedata.fits.gz', type='fits')
```

Any arguments passed to `TableSet()` when creating a table instance are passed to the `read()` method. This can be used to create a `TableSet()` instance and fill it with data in a single line. For example, the following:

```
tset = TableSet('somedata.xml')
```

is equivalent to:

```
tset = TableSet()
tset.read('somedata.xml')
```

6.4 Accessing a single table

Single tables can be accessed through the `TableSet.tables` python list. For example, the first table in a set can be accessed with:

```
tset.tables[0]
```

And all methods associated with single tables are then available. For example, the following shows how to run the `describe` method of the first table in a set:

```
tset.tables[0].describe()
```

6.5 Adding meta-data

As well as having keywords and comments associated with each `Table`, it is possible to have overall keywords and comments associated with a `TableSet`. Comments and keywords can be added to a table using the `add_comment()` and `add_keyword()` methods:

```
>>> tset.add_comment("This is a great table set")
>>> tset.add_keyword("version", 314)
```

CHAPTER

SEVEN

MASKING AND NULL VALUES

It is often useful to be able to define missing or invalid values in a table. There are currently two ways to do this in ATpy, *Null values*, and *Masking*. The preferred way is to use Masking, but this requires at least NumPy 1.4.1 in most cases, and the latest svn version of NumPy for SQL database input/output. Therefore, for version 0.9.4 of ATpy, the default is to use the Null value method. To opt-in to using masked arrays, specify the `masked=True` argument when creating a `Table` instance:

```
>>> t = Table('example.fits.gz', masked=True)
```

In future, once NumPy 1.5.0 is out, we will switch over to using masked arrays by default, and will slowly phase out the Null value method.

If you want to set the default for masking to be on or off for a whole script, this can be done using the `set_masked_default` function:

```
import atpy
atpy.set_masked_default(True)
```

If you want to set the default for masking on a user-level, create a file named `~/.atpyrc` in your home directory, containing:

```
[general]
masked_default:yes
```

The `set_masked_default` function overrides the `.atpyrc` file, and the `masked=` argument in Table overrides both the `set_masked_default` function and the `.atpyrc` file.

7.1 Null values

The basic idea behind this method is to specify a special value in each column that will signify missing or invalid data. To specify the Null value for a column, use the `null` argument in `add_column`:

```
>>> t.add_column('time', time, null=-999.)
```

Following this, if the table is written out to a file or database, this null value will be stored.

This method is generally unreliable, especially for floating point values, and does not allow users to easily distinguish between invalid and missing values.

7.2 Masking

NumPy supports masked arrays, where specific elements of an array can be properly masked by using a *mask* - a boolean array. There are several advantages to using this:

- The mask is unrelated to the value in the cell - any cell can be masked, not just all cells with a specific value
- It is possible to distinguish between invalid (e.g. NaN) and missing values
- Values can easily be unmasked (although when writing to a file/database, the 'old' values are lost for masked elements).
- NumPy provides masked versions of many functions, for example `sum`, `mean`, or `median`, which means that it is easy to correctly compute statistics on masked arrays, ignoring the masked values.

To specify the mask of a column, use the `mask` argument in `add_column`. To do the equivalent to the example in *Null values*, use:

```
>>> t.add_column('time', time, mask=time==-999.)
```

When writing out to certain file/database formats, a masked value has to be given a specific value - this is called a *fill* value. To set the fill value, simply use the `fill` argument when adding data to a column:

```
>>> t.add_column('time', time, mask=time==-999., fill=-999.)
```

In the above example, if the table is written out to an IPAC table, the value of -999. will be used for masked values.

Note: When implementing this in ATpy, we discovered a few bugs in the masked structured implementation of NumPy, which have now been fixed. Therefore, we recommend using the latest svn version of NumPy if you want to use masked arrays.

CHAPTER

EIGHT

CUSTOM READING/WRITING

One of the new features introduced in ATpy 0.9.2 is the ability for users to write their own read/write functions and *register* them with ATpy. A read or write function needs to satisfy the following requirements:

- The first argument should be a `Table` instance (in the case of a single table reader/writer) or a `TableSet` instance (in the case of a table set reader/writer)
- The function can take any other arguments, with the exception of the keyword arguments `verbose` and `type`.
- The function should not return anything, but rather should operate directly on the table or table set instance passed as the first argument
- If the file format supports masking/null values, the function should take into account that there are two ways to mask values (see *Masking and null values*). The `Table` instance has a `_masked` attribute that specifies whether the user wants a Table with masked arrays, or with a null value. The function should take this into account. For example, in the built-in FITS reader, the table is populated with `add_column` in the following way:

```
if self._masked:
    self.add_column(name, data, unit=columns.units[i], \
        mask=data==columns.nulls[i])
else:
    self.add_column(name, data, unit=columns.units[i], \
        null=columns.nulls[i])
```

The reader/writer function can then fill the table by using the `Table` methods described in *Full API for Table class* (for a single table reader/writer) or *Full API for TableSet class* (for a table set reader/writer). In particular, a single table reader will likely contain calls to `add_column`, while a single table writer will likely contain references to the `data` attribute of `Table`.

Once a custom function is available, the user can register it using one of the four ATpy functions:

- `atpy.register_reader`: Register a reader function for single tables
- `atpy.register_set_reader`: Register a reader function for table sets
- `atpy.register_writer`: Register a writer function for single tables
- `atpy.register_set_writer`: Register a writer function for tables sets

The API for these functions is of the form `(ttype, function, override=True/False)`, where `ttype` is the code name for the format (like the build-in `fits`, `vo`, `ipac`, or `sql` types), function is the actual function to use, and override allows the user to override existing definitions (for example to provide an improved `ipac` reader).

For example, if a function is defined for reading HDF5 tables, which we can call hdf5.read, then one would first need to register this function after importing atpy:

```
>>> import atpy
>>> atpy.register_reader('hdf5', hdf5.read)
```

This type can then be used when reading in a table:

```
>>> t = atpy.Table('mytable.hdf5', type='hdf5')
```

It is also possible to register extensions for a specific type using `atpy.register_extensions`. This function expects a table type and a list of file extensions to associate with it. For example, by setting:

```
>>> atpy.register_extensions('hdf5', ['hdf5', 'hdf'])
```

One can then read in an HDF5 table without specifying the type:

```
>>> t = atpy.Table('mytable.hdf5')
```

We encourage users to send us examples of reader/writer functions for various formats, and would be happy in future to include readers and writers for commonly used formats in ATpy.

CHAPTER

NINE

SUPPORTED FORMATS

The following pages describe the file formats currently supported, and format-specific options. A full API is also included for advanced users.

9.1 FITS tables

Note: The Flexible Image Transport System (FITS) format is a widely used file format in Astronomy, that is used to store, transmit, and manipulate images and tables. FITS tables contain one or more header-data units (HDU) which can be either images or tables in ASCII or binary format. Tables can contain meta-data, stored in the header.

9.1.1 Overview

FITS tables are supported thanks to the pyfits module. Reading FITS tables is straightforward:

```
>>> t = atpy.Table('table.fits')
```

If more than one table is present in the file, the HDU can be specified:

```
>>> t = atpy.Table('table.fits', hdu=2)
```

To read in all HDUs in a file, use the `TableSet` class:

```
>>> t = atpy.TableSet('table.fits')
```

Compressed FITS files can be read easily:

```
>>> t = atpy.Table('table.fits.gz')
```

In the event that ATpy does not recognize a FITS table (for example if the file extension is obscure), the type can be explicitly given:

```
>>> t = atpy.Table('table', type='fits')
```

Note: As for all file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.1.2 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()`, `Table.write()`, `TableSet()/TableSet.read()`, and `TableSet.write()` respectively.

`atpy.fitstable.`**`read`**(*self*, *filename*, *hdu=None*, *memmap=False*, *verbose=True*)
Read a table from a FITS file

Required Arguments:

***filename*: [string]** The FITS file to read the table from

Optional Keyword Arguments:

***hdu*: [integer]** The HDU to read from the FITS file (this is only required if there are more than one table in the FITS file)

***memmap*: [bool]** Whether PyFITS should use memory mapping

`atpy.fitstable.`**`write`**(*self*, *filename*, *overwrite=False*)
Write the table to a FITS file

Required Arguments:

***filename*: [string]** The FITS file to write the table to

Optional Keyword Arguments:

***overwrite*: [True | False]** Whether to overwrite any existing file without warning

`atpy.fitstable.`**`read_set`**(*self*, *filename*, *memmap=False*, *verbose=True*)
Read all tables from a FITS file

Required Arguments:

***filename*: [string]** The FITS file to read the tables from

Optional Keyword Arguments:

***memmap*: [bool]** Whether PyFITS should use memory mapping

`atpy.fitstable.`**`write_set`**(*self*, *filename*, *overwrite=False*)
Write the tables to a FITS file

Required Arguments:

***filename*: [string]** The FITS file to write the tables to

Optional Keyword Arguments:

***overwrite*: [True | False]** Whether to overwrite any existing file without warning

9.2 VO tables

Note: Virtual Observatory (VO) tables are a new format developed by the International Virtual Observatory Alliance to store one or more tables. It is a format based on the Extensible Markup Language (XML).

VO tables are supported thanks to the vo module. Reading VO tables is straightforward:

```
>>> t = atpy.Table('table.vot')
```

If more than one table is present in the file, ATpy will give a list of available tables, identified by an ID (`tid`). The specific table to read can then be specified with the `tid=` argument:

```
>>> t = atpy.Table('table.vot', tid=2)
```

To read in all tables in a file, use the `TableSet` class:

```
>>> t = atpy.TableSet('table.vot')
```

In some cases, the VO table file may not be strictly standard compliant. When reading in a VO table, it is possible to specify an argument which controls whether to adhere strictly to standards and throw an exception if any errors are found (`pedantic=True`), or whether to relax the requirements and accept non-standard features (`pedantic=False`). The latter is the default.

Finally, when writing out a VO table, the default is to use ASCII VO tables (analogous to ASCII FITS tables). It is also possible to write tables out in binary VO format. To do this, use the `votype` argument:

```
>>> t.write('table.vot', votype='binary')
```

The default is `votype='ascii'`.

In the event that ATpy does not recognize a VO table (for example if the file extension is obscure), the type can be explicitly given:

```
>>> t = atpy.Table('table', type='vo')
```

Note: As for all file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.2.1 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()`/`Table.read()`, `Table.write()`, `TableSet()`/`TableSet.read()`, and `TableSet.write()` respectively.

`atpy.votable.`**`read`**(*self*, *filename*, *pedantic=False*, *tid=-1*, *verbose=True*)

Read a table from a VOT file

Required Arguments:

***filename*: [string]** The VOT file to read the table from

Optional Keyword Arguments:

***tid*: [integer]** The ID of the table to read from the VO file (this is only required if there are more than one table in the VO file)

***pedantic*: [True | False]** When *pedantic* is True, raise an error when the file violates the VO Table specification, otherwise issue a warning.

`atpy.votable.`**`write`**(*self*, *filename*, *votype='ascii'*, *overwrite=False*)

Write the table to a VOT file

Required Arguments:

***filename*: [string]** The VOT file to write the table to

Optional Keyword Arguments:

***votype*: ['ascii' | 'binary']** Whether to write the table as ASCII or binary

`atpy.votable.`**`read_set`**(*self*, *filename*, *pedantic=False*, *verbose=True*)
Read all tables from a VOT file

Required Arguments:

***filename*: [string]** The VOT file to read the tables from

Optional Keyword Arguments:

***pedantic*: [True | False]** When *pedantic* is True, raise an error when the file violates the VO Table specification, otherwise issue a warning.

`atpy.votable.`**`write_set`**(*self*, *filename*, *votype='ascii'*, *overwrite=False*)
Write all tables to a VOT file

Required Arguments:

***filename*: [string]** The VOT file to write the tables to

Optional Keyword Arguments:

***votype*: ['ascii' | 'binary']** Whether to write the tables as ASCII or binary tables

9.3 HDF5 tables

Note: The Hierarchical Data Format (HDF) is a format that can be used to store, transmit, and manipulate datasets (n-dimensional arrays or tables). Datasets can be collected into groups, which can be collected into larger groups. Datasets and groups can contain meta-data, in the form of attributes.

HDF5 tables are supported thanks to the h5py module. Reading HDF5 tables is straightforward:

```
>>> t = atpy.Table('table.hdf5')
```

If more than one table is present in the file, ATpy will give a list of available tables, identified by a path. The specific table to read can then be specified with the `table=` argument:

```
>>> t = atpy.Table('table.hdf5', table='Measurements')
```

In the case where a table is inside a group, or a hierarchy of groups, the table name may be a full path inside the file:

```
>>> t = atpy.Table('table.hdf5', table='Group1/Measurements')
```

To read in all tables in an HDF5 file, use the `TableSet` class:

```
>>> t = atpy.TableSet('table.hdf5')
```

When writing out an HDF5 table, the default is to write the uncompressed, but it is possible to turn on compression using the `compression` argument:

```
>>> t.write('table.hdf5', compression=True)
```

To write the table to a specific group within the file, use the `group` argument:

```
>>> t.write('table.hdf5', group='Group4')
```

Finally, it is possible to append tables to existing files, using the `append` argument. For example, the following two commands write out two tables to the same existing file:

```
>>> t1.write('existing_table.hdf', append=True)
>>> t2.write('existing_table.hdf', append=True)
```

In the event that ATpy does not recognize an HDF5 table (for example if the file extension is obscure), the type can be explicitly given:

```
>>> t = atpy.Table('table', type='hdf5')
```

Note: As for all file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.3.1 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()`, `Table.write()`, `TableSet()/TableSet.read()`, and `TableSet.write()` respectively.

`atpy.hdf5table.`**`read`**(*self*, *filename*, *table=None*, *verbose=True*)

Read a table from an HDF5 file

Required Arguments:

***filename*: [string]**

The HDF5 file to read the table from

OR

***file or group handle*: [h5py.highlevel.File | h5py.highlevel.Group]** The HDF5 file handle or group handle to read the table from

Optional Keyword Arguments:

***table*: [string]** The name of the table to read from the HDF5 file (this is only required if there are more than one table in the file)

`atpy.hdf5table.`**`write`**(*self*, *filename*, *compression=False*, *group=''*, *append=False*, *overwrite=False*, *ignore_groups=False*)

Write the table to an HDF5 file

Required Arguments:

***filename*: [string]**

The HDF5 file to write the table to

OR

***file or group handle*: [h5py.highlevel.File | h5py.highlevel.Group]** The HDF5 file handle or group handle to write the table to

Optional Keyword Arguments:

***compression*: [True | False]** Whether to compress the table inside the HDF5 file

***group*: [string]** The group to write the table to inside the HDF5 file

***append*: [True | False]** Whether to append the table to an existing HDF5 file

***overwrite*: [True | False]** Whether to overwrite any existing file without warning

***ignore_groups*: [True | False]** With this option set to True, groups are removed from table names. With this option set to False, tables are placed in groups that are present in the table name, and the groups are created if necessary.

`atpy.hdf5table.`**`read_set`**(*self*, *filename*, *pedantic=False*, *verbose=True*)

Read all tables from an HDF5 file

Required Arguments:

***filename*: [string]** The HDF5 file to read the tables from

`atpy.hdf5table.`**`write_set`**(*self*, *filename*, *compression=False*, *group=''*, *append=False*, *overwrite=False*, *ignore_groups=False*, ***kwargs*)

Write the tables to an HDF5 file

Required Arguments:

***filename*: [string]**

The HDF5 file to write the tables to

OR

***file or group handle*: [h5py.highlevel.File | h5py.highlevel.Group]** The HDF5 file handle or group handle to write the tables to

Optional Keyword Arguments:

***compression*: [True | False]** Whether to compress the tables inside the HDF5 file

***group*: [string]** The group to write the table to inside the HDF5 file

***append*: [True | False]** Whether to append the tables to an existing HDF5 file

***overwrite*: [True | False]** Whether to overwrite any existing file without warning

***ignore_groups*: [True | False]** With this option set to True, groups are removed from table names. With this option set to False, tables are placed in groups that are present in the table name, and the groups are created if necessary.

9.4 IPAC tables

Note: IPAC tables are an ASCII table that can contain a single table. The format can contain meta-data that consists of keyword values and comments (analogous to FITS files), and the column headers are separated by pipe (|) symbols that indicate the position of the columns.

IPAC tables are natively supported in ATpy (no additional module is required). Reading IPAC tables is straightforward:

```
>>> t = atpy.Table('table.tbl')
```

and writing a table out in IPAC format is equally easy:

```
>>> t.write('table.tbl')
```

IPAC tables can have three different definitions with regard to the alignment of the columns with the pipe symbols in the header. The definition to use is controlled by the `definition` argument. The definitions are:

1. Any character below a pipe symbol belongs to the column on the left, and any characters below the first pipe symbol belong to the first column.
2. Any character below a pipe symbol belongs to the column on the right.
3. No characters should be present below the pipe symbols.

The default is `definition=3`.

Note: As for all file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.4.1 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()` and `Table.write()` respectively.

`atpy.ipactable.`**`read`**(*self*, *filename*, *definition=3*, *verbose=False*, *smart_typing=False*)

Read a table from a IPAC file

Required Arguments:

***filename*: [string]** The IPAC file to read the table from

Optional Keyword Arguments:

definition: [1 | 2 | 3]

The definition to use to read IPAC tables:

1: any character below a pipe symbol belongs to the column on the left, and any characters below the first pipe symbol belong to the first column.

2: any character below a pipe symbol belongs to the column on the right.

3: no characters should be present below the pipe symbols (default).

smart_typing: [True | False]

Whether to try and save memory by using the smallest integer type that can contain a column. For example, a column containing only values between 0 and 255 can be stored as an unsigned 8-bit integer column. The default is false, so that all integer columns are stored as 64-bit integers.

`atpy.ipactable.`**`write`**(*self*, *filename*, *overwrite=False*)

Write the table to an IPAC file

Required Arguments:

***filename*: [string]** The IPAC file to write the table to

9.5 ASCII tables

Note: There are probably as many ASCII table formats as astronomers (if not more). These generally store a single table, and can sometimes include meta-data.

9.5.1 Overview

Reading ASCII tables is supported thanks to the asciitable module, which makes it easy to read in arbitrary ASCII files.

By default, several pre-defined formats are available. These include CDS tables (also called Machine-Readable tables), DAOPhot tables, and RDB tables. To read these formats, simply use:

```
>>> t = atpy.Table('table.mrt', type='mrt')
>>> t = atpy.Table('table.cds', type='cds')
>>> t = atpy.Table('table.phot', type='daophot')
>>> t = atpy.Table('table.rdb', type='rdb')
```

The *type=* argument is optional for these formats, if they have appropriate file extensions, but due to the large number of ASCII file formats, it is safer to include it.

ATpy also allows full access to asciitable. If the `type='ascii'` argument is specified in `Table()`, all arguments are passed to `asciitable.read`, and the result is automatically stored in the ATpy `Table` instance. For more information on the arguments available in `asciitable.read`, see here.

Note: As for all file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.5.2 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()`.

`atpy.asciitables.`**`read_cds`**(*self*, *filename*, ***kwargs*)
Read data from a CDS table (also called Machine Readable Tables) file

Required Arguments:

***filename*: [string]** The file to read the table from

Keyword Arguments are passed to astropy.io.ascii

`atpy.asciitables.`**`read_daophot`**(*self*, *filename*, ***kwargs*)
Read data from a DAOphot table

Required Arguments:

***filename*: [string]** The file to read the table from

Keyword Arguments are passed to astropy.io.ascii

`atpy.asciitables.`**`read_rdb`**(*self*, *filename*, ***kwargs*)
Read data from an RDB table

Required Arguments:

***filename*: [string]** The file to read the table from

Keyword Arguments are passed to astropy.io.ascii

atpy.asciitables.**read_ascii**(*self*, *filename*, ***kwargs*)

Read a table from an ASCII file using astropy.io.ascii

Optional Keyword Arguments:

Reader - Reader class (default= BasicReader) Inputter - Inputter 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)

Note that the Outputter argument is not passed to astropy.io.ascii.

See the astropy.io.ascii documentation at http://docs.astropy.org/en/latest/io/ascii/index.html for more details.

atpy.asciitables.**write_ascii**(*self*, *filename*, ***kwargs*)

Read a table from an ASCII file using astropy.io.ascii

Optional Keyword Arguments:

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)

See the astropy.io.ascii documentation at http://docs.astropy.org/en/latest/io/ascii/index.html for more details.

9.6 SQL databases

Note: Structured Query Language (SQL) databases are wildly used in web infrastructure, and are also used to store large datasets in Science. Several flavors exist, the most popular of which are SQLite, MySQL, and PostGreSQL.

SQL databases are supported in ATpy thanks to the sqlite module built-in to Python, the MySQL-python module, and the PyGreSQL module. When reading from databases, the first argument in `Table` should be the database type (one of `sqlite`, `mysql`, and `postgres`). For SQLite databases, which are stored in a file, reading in a table is easy:

```
>>> t = atpy.Table('sqlite', 'mydatabase.db')
```

If more than one table is present in the file, the table name can be specified:

```
>>> t = atpy.Table('sqlite', 'mydatabase.db', table='observations')
```

For MySQL databases, standard MySQL parameters can be specified. These include `user`, `passwd`, `db` (the database name), `host`, and `port`. For PostGreSQL databases, standard PostGreSQL parameters can be specified. These include `user`, `password`, `database`, and `host`.

For example, to read a table called `velocities` from a MySQL database called `measurements`, with a user `monty` and password `spam`, one would use:

```
>>> t = atpy.Table('mysql', user='monty', passwd='spam',
                   db='measurements', table='velocities')
```

To read in all the tables in a database, simply use the `TableSet` class, e.g:

```
>>> t = atpy.TableSet('sqlite', 'mydatabase.db')
```

or

```
>>> t = atpy.TableSet('mysql', user='monty', passwd='spam',
                      db='measurements')
```

It is possible to retrieve only a subset of a table, or the result of any standard SQL query, using the `query` argument. For example, the following will retrieve all entries where the `quality` variable is positive:

```
>>> t = atpy.Table('mysql', user='monty', passwd='spam',
                   db='measurements', table='velocities',
                   query='SELECT * FROM velocities WHERE quality > 0;' )
```

Any valid SQL command should work, including commands used to merge different tables.

Writing tables or table sets to databases is simple, and is done through the `write` method. As before, database parameters may need to be specified, e.g.:

```
>>> t.write('sqlite', 'mydatabase.db')
```

or

```
>>> t.write('mysql', user='monty', passwd='spam',
            db='measurements')
```

Note: As for file formats, the `verbose` argument can be specified to control whether warning messages are shown when reading (the default is `verbose=True`), and the `overwrite` argument can be used when writing to overwrite a file (the default is `overwrite=False`).

9.6.1 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()`, `Table.write()`, `TableSet()/TableSet.read()`, and `TableSet.write()` respectively.

`atpy.sqltable.`**`read`**(*self*, *dbtype*, **args*, ***kwargs*)

Required Arguments:

***dbtype*: ['sqlite' | 'mysql' | 'postgres']** The SQL database type

Optional arguments (only for Table.read() class):

***table*: [string]** The name of the table to read from the database (this is only required if there are more than one table in the database). This is not required if the query= argument is specified, except if using an SQLite database.

***query*: [string]** An arbitrary SQL query to construct a table from. This can be any valid SQL command provided that the result is a single table.

The remaining arguments depend on the database type:

•SQLite:

Arguments are passed to sqlite3.connect(). For a full list of available arguments, see the help page for sqlite3.connect(). The main arguments are listed below.

Required arguments:

***dbname*: [string]** The name of the database file

•MySQL:

Arguments are passed to MySQLdb.connect(). For a full list of available arguments, see the documentation for MySQLdb. The main arguments are listed below.

Optional arguments:

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***passwd*: [string]** The user password (default is blank)

***db*: [string]** The name of the database to connect to (no default)

***port* [integer]** The port to connect to (default is 3306)

•PostGreSQL:

Arguments are passed to pgdb.connect(). For a full list of available arguments, see the help page for pgdb.connect(). The main arguments are listed below.

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***password*: [string]** The user password (default is blank)

***database*: [string]** The name of the database to connect to (no default)

atpy.sqltable.**write**(*self*, *dbtype*, **args*, ***kwargs*)

Required Arguments:

***dbtype*: ['sqlite' | 'mysql' | 'postgres']** The SQL database type

Optional arguments (only for Table.read() class):

***table*: [string]** The name of the table to read from the database (this is only required if there are more than one table in the database). This is not required if the query= argument is specified, except if using an SQLite database.

***query*: [string]** An arbitrary SQL query to construct a table from. This can be any valid SQL command provided that the result is a single table.

The remaining arguments depend on the database type:

•SQLite:

Arguments are passed to sqlite3.connect(). For a full list of available arguments, see the help page for sqlite3.connect(). The main arguments are listed below.

Required arguments:

***dbname*: [string]** The name of the database file

•MySQL:

Arguments are passed to MySQLdb.connect(). For a full list of available arguments, see the documentation for MySQLdb. The main arguments are listed below.

Optional arguments:

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***passwd*: [string]** The user password (default is blank)

***db*: [string]** The name of the database to connect to (no default)

***port* [integer]** The port to connect to (default is 3306)

•PostGreSQL:

Arguments are passed to pgdb.connect(). For a full list of available arguments, see the help page for pgdb.connect(). The main arguments are listed below.

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***password*: [string]** The user password (default is blank)

***database*: [string]** The name of the database to connect to (no default)

`atpy.sqltable.`**`read_set`**(*self*, *dbtype*, **args*, ***kwargs*)

Required Arguments:

***dbtype*: ['sqlite' | 'mysql' | 'postgres']** The SQL database type

Optional arguments (only for Table.read() class):

***table*: [string]** The name of the table to read from the database (this is only required if there are more than one table in the database). This is not required if the query= argument is specified, except if using an SQLite database.

***query*: [string]** An arbitrary SQL query to construct a table from. This can be any valid SQL command provided that the result is a single table.

The remaining arguments depend on the database type:

•SQLite:

Arguments are passed to sqlite3.connect(). For a full list of available arguments, see the help page for sqlite3.connect(). The main arguments are listed below.

Required arguments:

***dbname*: [string]** The name of the database file

•MySQL:

Arguments are passed to MySQLdb.connect(). For a full list of available arguments, see the documentation for MySQLdb. The main arguments are listed below.

Optional arguments:

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***passwd*: [string]** The user password (default is blank)

***db*: [string]** The name of the database to connect to (no default)

***port* [integer]** The port to connect to (default is 3306)

•PostGreSQL:

Arguments are passed to pgdb.connect(). For a full list of available arguments, see the help page for pgdb.connect(). The main arguments are listed below.

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***password*: [string]** The user password (default is blank)

***database*: [string]** The name of the database to connect to (no default)

atpy.sqltable.**write_set**(*self*, *dbtype*, **args*, ***kwargs*)

Required Arguments:

***dbtype*: ['sqlite' | 'mysql' | 'postgres']** The SQL database type

Optional arguments (only for Table.read() class):

***table*: [string]** The name of the table to read from the database (this is only required if there are more than one table in the database). This is not required if the query= argument is specified, except if using an SQLite database.

***query*: [string]** An arbitrary SQL query to construct a table from. This can be any valid SQL command provided that the result is a single table.

The remaining arguments depend on the database type:

•SQLite:

Arguments are passed to sqlite3.connect(). For a full list of available arguments, see the help page for sqlite3.connect(). The main arguments are listed below.

Required arguments:

***dbname*: [string]** The name of the database file

•MySQL:

Arguments are passed to MySQLdb.connect(). For a full list of available arguments, see the documentation for MySQLdb. The main arguments are listed below.

Optional arguments:

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***passwd*: [string]** The user password (default is blank)

***db*: [string]** The name of the database to connect to (no default)

***port* [integer]** The port to connect to (default is 3306)

•PostGreSQL:

Arguments are passed to pgdb.connect(). For a full list of available arguments, see the help page for pgdb.connect(). The main arguments are listed below.

***host*: [string]** The host to connect to (default is localhost)

***user*: [string]** The user to conenct as (default is current user)

***password*: [string]** The user password (default is blank)

***database*: [string]** The name of the database to connect to (no default)

9.7 Online queries

It is possible to query online databases and automatically return the results as a `Table` instance. There are several mechanisms for accessing online catalogs:

9.7.1 Virtual Observatory

An interface to the virtual observatory is provided via the vo module. To list the catalogs available, use the `list_catalogs()` method from `atpy.vo_conesearch`:

```
>>> from atpy.vo_conesearch import list_catalogs
>>> list_catalogs()
           USNO-A2
           USNO-B1
        USNO NOMAD
          USNO ACT
```

A specific catalog can then be queried with a conesearch by specifying a catalog, and the coordinates and radius (in degrees) to search:

```
>>> t = atpy.Table(catalog='USNO-B1', ra=233.112, dec=23.432, radius=0.3, type='vo_conesearch')
```

How long this query takes will depend on the speed of your network, the load on the server being queried, and the number of rows in the result. For advanced users, it is also possible to query catalogs not listed by `list_catalogs()` - for more details, see the *Full API for advanced users*.

9.7.2 IRSA Query

In addition to supporting Virtual Observatory queries, ATpy supports queries to the NASA/IPAC Infrared Science Archive (IRSA). The interface is similar to that of the VO. To list the catalogs available, use the `list_catalogs()` method from `atpy.irsa_service`:

```
>>> from atpy.irsa_service import list_catalogs
>>> list_catalogs()
            fp_psc  2MASS All-Sky Point Source Catalog (PSC)
            fp_xsc  2MASS All-Sky Extended Source Catalog (XSC)
            lga_v2  The 2MASS Large Galaxy Atlas
       fp_scan_dat  2MASS All-Sky Survey Scan Info
               ...  ...
```

The first column is the catalog code used in the query. A specific catalog can then be queried by specifying a query type, a catalog, and additional arguments as required. The different kinds of search are:

- `Cone`: This is a cone search. Requires `objstr`, a string containing either coordinates or an object name (see here for more information), and `radius`, with units given by `units` (`'arcsec'` by default). For example:

  ```
  >>> t = atpy.Table('Cone', 'fp_psc', objstr='m13', \
                     radius=100., type='irsa')
  ```

- `Box`: This is a box search. Requires `objstr`, a string containing either coordinates or an object name (see here for more information), and `size` in arcseconds. For example:

  ```
  >>> t = atpy.Table('Box', 'fp_psc', objstr='T Tau', \
                     size=200., units='deg', type='irsa')
  ```

- `Polygon`: This is a polygon search. Requires `polygon`, which should be a list of tuples of (ra, dec) in decimal degrees:

```
>>> t = atpy.Table('polygon','fp_psc', \
                   polygon=[(11.0, 45.0), (12.0, 45.0), (11.5, 46.)], \
                   type='irsa')
```

As for the VO query, how long these queries takes will depend on the speed of your network, the load on the IRSA server, and the number of rows in the result.

9.7.3 Full API for advanced users

Note: The following functions should not be called directly - the arguments should be passed to `Table()/Table.read()` using either `type=vo_conesearch` or `type=irsa`.

`atpy.vo_conesearch.`**`read`**(*self*, *catalog=None*, *ra=None*, *dec=None*, *radius=None*, *verb=1*, *pedantic=False*, ***kwargs*)

Query a VO catalog using the STScI vo module

This docstring has been adapted from the STScI vo conesearch module:

catalog [None | string | VOSCatalog | list]

May be one of the following, in order from easiest to use to most control:

- None: A database of conesearch catalogs is downloaded from STScI. The first catalog in the database to successfully return a result is used.
- catalog name: A name in the database of conesearch catalogs at STScI is used. For a list of acceptable names, see vo_conesearch.list_catalogs().
- url: The prefix of a url to a IVOA Cone Search Service. Must end in either ? or &.
- A VOSCatalog instance: A specific catalog manually downloaded and selected from the database using the APIs in the STScI vo.vos_catalog module.
- Any of the above 3 options combined in a list, in which case they are tried in order.

***pedantic* [bool]** When pedantic is True, raise an error when the returned VOTable file violates the spec, otherwise issue a warning.

***ra* [float]** A right-ascension in the ICRS coordinate system for the position of the center of the cone to search, given in decimal degrees.

***dec* [float]** A declination in the ICRS coordinate system for the position of the center of the cone to search, given in decimal degrees.

***radius* [float]** The radius of the cone to search, given in decimal degrees.

***verb* [int]** Verbosity, 1, 2, or 3, indicating how many columns are to be returned in the resulting table. Support for this parameter by a Cone Search service implementation is optional. If the service supports the parameter, then when the value is 1, the response should include the bare minimum of columns that the provider considers useful in describing the returned objects. When the value is 3, the service should return all of the columns that are available for describing the objects. A value of 2 is intended for requesting a medium number of columns between the minimum and maximum (inclusive) that are considered by the provider to most typically useful to the user. When the verb parameter is not provided, the server should respond as if verb = 2. If the verb parameter is not supported by the service, the service should ignore the parameter and should always return the same columns for every request.

Additional keyword arguments may be provided to pass along to the server. These arguments are specific to the particular catalog being queried.

atpy.irsa_service.**read**(*self*, *spatial*, *catalog*, *objstr=None*, *radius=None*, *units='arcsec'*, *size=None*, *polygon=None*)

Query the NASA/IPAC Infrared Science Archive (IRSA)

Required Arguments:

***spatial* ['Cone' | 'Box' | 'Polygon']** The type of query to execute

***catalog* [string]** One of the catalogs listed by `atpy.irsa_service.list_catalogs()`

Optional Keyword Arguments:

***objstr* [str]** This string gives the position of the center of the cone or box if performing a cone or box search. The string can give coordinates in various coordinate systems, or the name of a source that will be resolved on the server (see here for more details). Required if spatial is 'Cone' or 'Box'.

***radius* [float]** The radius for the cone search. Required if spatial is 'Cone'

***units* ['arcsec' | 'arcmin' | 'deg']** The units for the cone search radius. Defaults to 'arcsec'.

***size* [float]** The size of the box to search in arcseconds. Required if spatial is 'Box'.

***polygon* [list of tuples]** The list of (ra, dec) pairs, in decimal degrees, outlinining the polygon to search in. Required if spatial is 'Polygon'

Unless stated otherwise below, all table types support the following types:

- booleans
- 8-, 16-, 32-, and 64- bit signed integer numbers.
- 8-, 16-, 32-, and 64- bit unsigned integer numbers.
- 32- and 64-bit floating point numbers.
- ASCII strings

CHAPTER

TEN

FULL API FOR TABLE CLASS

10.1 Table initialization and I/O

`Table.`**`reset`**`()`
Empty the table

`Table.`**`read`**`(`**args*, ***kwargs*`)`
Read in a table from a file/database.

Optional Keyword Arguments (independent of table type):

***verbose*: [True | False]** Whether to print out warnings when reading (default is True)

***type*: [string]** The read method attempts to automatically guess the file/database format based on the arguments supplied. The type can be overridden by setting this argument.

`Table.`**`write`**`(`**args*, ***kwargs*`)`
Write out a table to a file/database.

Optional Keyword Arguments (independent of table type):

***verbose*: [True | False]** Whether to print out warnings when writing (default is True)

***type*: [string]** The read method attempts to automatically guess the file/database format based on the arguments supplied. The type can be overridden by setting this argument.

10.2 Meta-data

`Table.`**`add_comment`**`(`*comment*`)`
Add a comment to the table

Required Argument:

***comment*: [string]** The comment to add to the table

`Table.`**`add_keyword`**`(`*key*, *value*`)`
Add a keyword/value pair to the table

Required Arguments:

***key*: [string]** The name of the keyword

***value*: [string | float | integer | bool]** The value of the keyword

`Table.`**`describe`**`()`
Prints a description of the table

10.3 Column manipulation

Table.**add_column**(*name*, *data*, *unit=''*, *null=''*, *description=''*, *format=None*, *dtype=None*, *column_header=None*, *before=None*, *after=None*, *position=None*, *mask=None*, *fill=None*)

Add a column to the table

Required Arguments:

***name*: [string]** The name of the column to add

***data*: [numpy array]** The column data

Optional Keyword Arguments:

***unit*: [string]** The unit of the values in the column

***null*: [same type as data]** The values corresponding to 'null', if not NaN

***description*: [string]** A description of the content of the column

***format*: [string]** The format to use for ASCII printing

***dtype*: [numpy type]** Numpy type to convert the data to. This is the equivalent to the dtype= argument in numpy.array

***column_header*: [ColumnHeader]** The metadata from an existing column to copy over. Metadata includes the unit, null value, description, format, and dtype. For example, to create a column 'b' with identical metadata to column 'a' in table 't', use:

```
>>> t.add_column('b', column_header=t.columns[a])
```

***before*: [string]** Column before which the new column should be inserted

***after*: [string]** Column after which the new column should be inserted

***position*: [integer]** Position at which the new column should be inserted (0 = first column)

***mask*: [numpy array]** An array of booleans, with the same dimensions as the data, indicating whether or not to mask values.

***fill*: [value]** If masked arrays are used, this value is used as the fill value in the numpy masked array.

Table.**add_empty_column**(*name*, *dtype*, *unit=''*, *null=''*, *description=''*, *format=None*, *column_header=None*, *shape=None*, *before=None*, *after=None*, *position=None*)

Add an empty column to the table. This only works if there are already existing columns in the table.

Required Arguments:

***name*: [string]** The name of the column to add

***dtype*: [numpy type]** Numpy type of the column. This is the equivalent to the dtype= argument in numpy.array

Optional Keyword Arguments:

***unit*: [string]** The unit of the values in the column

***null*: [same type as data]** The values corresponding to 'null', if not NaN

***description*: [string]** A description of the content of the column

***format*: [string]** The format to use for ASCII printing

***column_header*: [ColumnHeader]** The metadata from an existing column to copy over. Metadata includes the unit, null value, description, format, and dtype. For example, to create a column 'b' with identical metadata to column 'a' in table 't', use:

```
>>> t.add_column('b', column_header=t.columns[a])
```

***shape*: [tuple]** Tuple describing the shape of the empty column that is to be added. If a one element tuple is specified, it is the number of rows. If a two element tuple is specified, the first is the number of rows, and the second is the width of the column.

***before*: [string]** Column before which the new column should be inserted

***after*: [string]** Column after which the new column should be inserted

***position*: [integer]** Position at which the new column should be inserted (0 = first column)

`Table.`**`remove_columns`**(*remove_names*)
Remove several columns from the table

Required Argument:

***remove_names*: [list of strings]** A list containing the names of the columns to remove

`Table.`**`keep_columns`**(*keep_names*)
Keep only specific columns in the table (remove the others)

Required Argument:

***keep_names*: [list of strings]** A list containing the names of the columns to keep. All other columns will be removed.

`Table.`**`rename_column`**(*old_name*, *new_name*)
Rename a column from the table

Require Arguments:

***old_name*: [string]** The current name of the column.

***new_name*: [string]** The new name for the column

`Table.`**`set_primary_key`**(*key*)
Set the name of the table column that should be used as a unique identifier for the record. This is the same as primary keys in SQL databases. A primary column cannot contain NULLs and must contain only unique quantities.

Required Arguments:

***key*: [string]** The column to use as a primary key

10.4 Table manipulation and selection

`Table.`**`sort`**(*keys*)
Sort the table according to one or more keys. This operates on the existing table (and does not return a new table).

Required arguments:

***keys*: [string | list of strings]** The key(s) to order by

Table.**row**(*row_number*, *python_types=False*)

Returns a single row

Required arguments:

***row_number*: [integer]** The row number (the first row is 0)

Optional Keyword Arguments:

***python_types*: [True | False]** Whether to return the row elements with python (True) or numpy (False) types.

Table.**rows**(*row_ids*)

Select specific rows from the table and return a new table instance

Required Argument:

***row_ids*: [list | np.int array]** A python list or numpy array specifying which rows to select, and in what order.

Returns:

A new table instance, containing only the rows selected

Table.**where**(*mask*)

Select matching rows from the table and return a new table instance

Required Argument:

***mask*: [np.bool array]** A boolean array with the same length as the table.

Returns:

A new table instance, containing only the rows selected

CHAPTER

ELEVEN

FULL API FOR TABLESET CLASS

11.1 TableSet initialization and I/O

TableSet.**reset**()
Empty the table set

TableSet.**read**(**args*, ***kwargs*)
Read in a table set from a file/database.

Optional Keyword Arguments (independent of table type):

***verbose*: [True | False]** Whether to print out warnings when reading (default is True)

***type*: [string]** The read method attempts to automatically guess the file/database format based on the arguments supplied. The type can be overridden by setting this argument.

TableSet.**write**(**args*, ***kwargs*)
Write out a table set to a file/database.

Optional Keyword Arguments (independent of table type):

***verbose*: [True | False]** Whether to print out warnings when writing (default is True)

***type*: [string]** The read method attempts to automatically guess the file/database format based on the arguments supplied. The type can be overridden by setting this argument.

11.2 Meta-data

TableSet.**add_comment**(*comment*)
Add a comment to the table set

Required Argument:

***comment*: [string]** The comment to add to the table

TableSet.**add_keyword**(*key*, *value*)
Add a keyword/value pair to the table set

Required Arguments:

***key*: [string]** The name of the keyword

***value*: [string | float | integer | bool]** The value of the keyword

TableSet.**describe**()
Describe all the tables in the set

11.3 TableSet manipulation and I/O

TableSet.**append**(*table*)

Append a table to the table set

Required Arguments:

***table*: [a table instance]** This can be a table of any type, which will be converted to a table of the same type as the parent set (e.g. adding a single VOTable to a FITSTableSet will convert the VOTable to a FITSTable inside the set)

PYTHON MODULE INDEX

a