Table Management in Python - II

Kaustubh Vaghmare*

July 14, 2013

Abstract

This is the second and the final part of the document set called "Table Management in Python". The earlier document covered more about the module "asciitable". The primary focus was to read tables in various formats and write them out. The key advantage of the module is that it can be extended to include more types of files. This is what atpy does. It inherits "asciitable" functionality and adds several of its own classes and methods, giving you a module that allows you to not only read and write tables but also manipulate rows and columns. The knowledge that we have acquired in the part 1 of the document will allow us to spend less time in reading and writing tables as the options, commands, etc are similar. So, I urge you to read the first part of the document to get more out of the second!

1 Introduction

This document tries to extend the ideas already discussed in the part 1 of the document. The part 1 of the document stressed more on using asciitable and using its read() and write() functions to read tables of a variety of formats and also write them. But that does not end table management at all. In fact, it is the beginning. You generally need to do a lot more and I do not mean adding two columns to make a third column. This can be done too trivially since if Numpy is installed, asciitable by default gives you Numpy arrays on which you can perform various functions.

The beginning and the end of asciitable is to allow you to read and write ascii tables. The module is not meant to provide you with a library of tools to manipulate the structure of the tables. It also does not allow you to read tables that are not ASCII in nature, like FITS files or VOTables which are .xml files. So, at this point you might wonder why we spent so much time appreciating asciitable. Well, simply because you can extend it to define your own types of tables. Thus the knowledge of it was important. Further, if all you needed was to load the table and save it, this is enough.

There is a module called ATPy which allows one to do more manipulation with tables than just reading and writing them. We proceed to describe the basic features of this module below. Let us first install it. This can be done by downloading

`https://github.com/downloads/atpy/atpy/ATpy-0.9.6.tar.gz`

Then, one can extract it and run,

*Research Scholar at IUCAA, Pune; kaustubh@iucaa.ernet.in

```
sudo python setup.py install
```

To check if the installation worked, in an interactive Python shell, simply say

```
import atpy
```

If it works, we are ready to go. Please note that this may not work well if you have not installed asciitable, vo, etc which are Python modules. Please refer to detailed documentations for understanding which dependencies are compulsory and which are optional. We will now assume an installation of ATPy and proceed to explaining how you can exploit it.

2 Opening an Existing Table

There is a class called Table which is the centre of all the action. To load up a table, simply use the following statement.

```
tableobject = atpy.Table()
tableobject.read("NameOfFile.ext", type="Type")
```

If the ext speaks about the type of format, you will load it else you will have to manually specify the format. For eg. if your text file table is Table.dat, from the .dat extension it is not obvious what can be the format of this table. Unlike asciitable where it would auto-detect independent of the extension atpy is a little conservative and cautious. It is better to actually give the type of file.

```
tableobject.read("Table.dat", type="ascii")
```

The supported table types currently available by default include - fits, vo, ipac, hdf5, cds, mrt, daophot, rdb, ascii

With appropriate modules installed, it is possible to read in databases such as those created using SQLLite or MySQL. In this version of the document, we will not cover on how to deal with these files.

Everything except an ASCII table is well structured and rigid. However, when trying to access an ASCII file, one can run into a lot of problems. Firstly, there is no preferred extension, there is no preferred format. Some might want to put in a comma as a separator, some might want to put in a pipe and so on. So, when loading ASCII tables above, it is vital to provide arguments.

Now, in part 1 of the document, we have in great detail described how to load ASCII tables using asciitable.read() module. All arguments can be used since it is an extension of asciitable.

```
tableobject.read("Table.dat", type="ascii", delimiter="|", includes_names = [...
```

3 Accessing Data from a Loaded Table

We can start out by opening a table

```
t = Table("MyTable.dat", type="ascii" ...)
```

So, let us assume that column names are given in the table and have been recognised, else we already know that some default names are assigned which are col1, col2 and so on...

Now, there are two ways in which you can access the data and we are already sure of which one you will prefer to use. One is...

```
t.col1
```

and

```
t.['col1']
```

And to access an entire row, you can proceed as follows -

```
t[0]
```

And to access a particular row for a particular column -

```
t.col1[0]
```

Now that we have seen how it is easy to access the data by merely referring to the column name, let us ask ourselves an important question - how do we check what columns are loaded in a given file? Simple, just say

```
t.columns
```

This returns a dictionary of column names and the data types. If you want to see a nice description of your current table instead of getting a usable Python object like a dictionary, just say -

```
t.describe()
```

You can access keywords and comments, if any, associated with the table by saying the following. Please remember that keywords work only for FITS tables where a header is defined.

```
t.keywords
t.comments
```

Let us now revisit how to access rows. It is possible to simply say, t[0] but a better way to do the same is by using the t.row() method.

```
t.row(1) # is equivalent to t[1]
```

By default, this returns a Numpy structured array but sometimes it is useful to get row entries as a list and in such situations it is possible to do the following,

```
t.row(1, python_types=True)
```

Now, let us introduce ourselves to two more methods which will make row access incredibly powerful and it is also the way you would want to typically use when playing around with the data.

```
a = t.rows([1,3,4]) # returns 2nd, 4th and 5th rows as table "a".
```

Remember that the above command will return a new table and not display the selected rows. Then every single method you are learning can be used on "a". Okay, next we shall discuss an even more powerful method to do this job called where().

```
a = t.where( (t.col1 == 3) & (t.col2 != 4) )
```

This returns all the rows which satsify the condition mentioned inside where() and stores them as a new table. This simply means that you are almost capable of querying the table and returning the subset relevant to you.

Let us wrap up this section on how to access table data by introducing two more functions. One is called len() which when you say len(t) gives you the total numbers of rows in the table and the other is t.shape which gives you a tuple comprising of number of rows and number of columns.

That's it, we have learned how to open a table and we have also learned how to access data from it. By now, it should be clear that atpy is already one step ahead by giving you better ways to access your tables than simply referring to them as arrays. Let us now learn to change the very shape of the table itself!

4 Changing Tables

We have loaded existing tables, we have accessed data from them and now it is time to express anger and frustration about the way the data has been organised and change it! Speaking of anger, let us first learn how to destroy columns inside a table.

```
t.remove_columns(["col1", "col2", ... ])
t.keep_columns(["col1", "col2", ... ])
```

Simple as that. Unhappy with the way a column has been named by your collaborator. Don't sweat, just change the name.

```
t.rename_column("old", "new")
```

If you want to concatenate two tables of exactly the same number of columns, say t1 and t2 and store the result in t1, then you can say,

```
t1.append(t2)
```

It will return an error if you attempt to join two tables that have dissimilar columns

And finally, to sort the entire table according to the values in a column, proceed as follows.

```
t.sort("colname")
```

If you need a tie breaker i.e. a second reference column for sorting those rows which have exactly the same entry for the sort column, then provide a list of column names.

```
t.sort( ["col1", "col2", ...] )
```

Done.

5 Table Construction

We have done quite a lot when it comes to opening existing tables, changing them and and so on. But hey, someone had to make those tables in the first place which means that you need to know

how to make tables rather than simply opening existing tables. Fear not, this section will take you there.

We need to now create an instance of the Table class and this can be done by saying,

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

At this point, you can say t.read(...) in order to read data from a file but this is something we have already done, so let us skip this part and focus on how to create tables using arrays that are hanging around in the memory. Let us called them a1, a2, a3 ... - these are just vectors waiting to become columns.

Let us now make columns in the current table.

```
t.add_column("ColumnName", a1)
t.add_column("ColumnName2", a2)
t.add_column("ColumnName3", a3)
```

That's it, you have added three vectors and made them the three columns of your table. The datatype is detected automatically. But it is also possible to assign a unit to the particular column and your own enforced data type.

```
t.add_column("ColumnName", a1, unit="second", dtype=np.float32)
```

And if you prefer adding an empty column so that some other piece of code in your program can then add elements one by one, then you would proceed as follows.

```
t.add_empty_column("colName", dtype=np.int16)
```

And then you can run some piece of code where you would manipulate each element of this empty column. eg.

```
t.colName[1] = 16
```

Finally, a comment on whether an array of many elements can become a single row entry for a column, in other words, can we have vector columns? The answer is - depends on your precise format. If the format is "vo" or "fits", this is possible and can be done by simply passing an appropriate 2-d array. In case you want to create an empty column that supports vectors, say

```
t.add_empty_column("colName", dtype=np.int16, shape=(nrows, ncolumns))
```

Further, to add metadata information or keyword information, proceed as follows.

```
t.add_comment("This table is meant for those who don't want to live!")
t.add_keyword("Key1",666)
```

Once you have played with your table "t" to your heart's content and are now finally ready to save it as a file, say -

```
t.write("Filename.ext", type ="type", ...)
```

All the arguments in read() are applicable here. The one additional argument that works here is called overwrite=True or overwrite=False which can control if you want to clobber the files already existing.