Table Management in Python - I

Kaustubh Vaghmare*

July 14, 2013

Abstract

Hi! Python is becoming a standard programming / scripting language of choice for astronomers and other scientists today. I believe the key power of Python and the key hurdle in mastering it, both, are the extensive volume of Python modules available today. Each module expands the ability of Python from a simple scripting / programming language to a language that was exclusively designed for a specific purpose such as table management, plotting, numerical computations, etc. Most often, there is only way to learn modules, the official documentation on the project web site. But truth be told, it helps to have a friendly tutorial on that module to get you started. Recently, I had to expand my horizons of programming abilities to include manipulating large and complex tables. To this end, I had to intensively study a couple of modules. These notes are a by-product of this journey. They are written in a highly informal tone and I recommend readers to freely offer suggestions / corrections as necessary. The document is split into two parts. The first covers only the "asciitable" module and is more concerned with reading and writing tables as opposed to manipulating them.

1 Introduction

If you have to deal with a lot of tables and you are using C, C++ or FORTRAN, you are doing it completely wrong! There is hardly any way in which a scientist who has to keep loading tables and performing row / column operations on it is going to have an efficient code that can do these manipulations. The preferred way to deal with tables is to use tools like TopCat, if you like the Click + Drag way or use a language like Python or IDL if you prefer having a code so that all operations are transparent.

Having made that statememt, now, this document - it is about Python and managing ASCII tables using Python. We are not going to cover Python and its basics. We are simply going to discuss table management using Python. To be more specific, we are going to dwell into a module called "asciitable" and play with tables.

For downloading instructions, refer to -

http://cxc.harvard.edu/contrib/asciitable/

Now, you may find instructions to get started with the module itself but this document is meant to be a little informal and easier to deal with. In other words, it is a tutorial rather than a mere description

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

of all methods. The purpose of this document will be to make it easier for you to understand the document thrown all over the place in the above link. So, let's start.

To use `asciitable` module, we proceed by saying

```
import asciitable
```

2 Opening Existing Tables

2.1 Simply Opening a Table!

Tables come in a variety of formats. Even saying it is in ASCII format is not enough, there can be many sub-varieties - CSV, tab separated, pipe separated and so on. Then, there is DAOPHOT format, IPAC format and so on. `asciitable` has a `read()` method which can try and guess the format of the table, thus in many cases it is not necessary to tell the precise nature of the file manually. But we will eventually learn how to do this too.

```
data = asciitable.read("MyFile.dat")
```

That's it! All the data in the table stored in the above file, if done in a standard way is loaded. `data` is now a complicated Numpy-like array which stores all data. Now, sometimes one specifies column names and sometimes not. If names are provided, one can access the data using the names of the columns and if not, some default column names will be assigned.

Let's say the file is of the form.

```
#Name, Age, Gender
Tom 19 Male
Angela 18 Female
Rick 24 Male
...
...
```

In such a case, the data which is now stored in an object called `data` can be accessed in the following way.

```
data['Name'] # gives the first column in this case.
data[2]  # gives the third row in this case.
```

In the case where the column names are not given, the first column will be called "col1", the second column name will be called "col2" and so on. So, in this case, the data in the first column can be accessed by saying,

```
data['col1']
```

That's it, you know how to open a table using the module 'asciitable'.

2.2 Opening Files - Advanced

Not happy with how the file is being opened when asciitable is allowed to guess its nature? You think you need to help it understand your file better? OK, this section is meant to help you achieve this.

1. My files do not have column names, but I would like to provide some during loading, col1, col2 are very inconvenient. Ok, we show you how.

   ```
   data = asciitable.read("MyTable.txt",
   names=['Name','Age',Gender'])
   ```

   Note: Even if some names are provided in your file, it is possible to override the same using the above command.

2. Hey, I think I may need to tell what delimiter is being used in my file specifically. Sure, it's simple.

   ```
   data = asciitable.read("MyTable.txt", delimiter="|")
   ```

3. I had already opened the table file as a file object called FileObject. Well, nothing much here, just supply the file object name during read.

   ```
   data = asciitable.read(FileObject, ...)
   ```

4. Hey, I think I will be happier if I specifically tell whether the file being opened is a CDS file or a DAOPHOT file and so on.

   ```
   data = asciitable.read("MyTable.txt",
   Reader=asciitable.CommentedHeader)
   ```

   Valid names include Basic, Cds, CommentedHeader, Daophot, Fixedwidth, Ipac, Latex, Aastex, Memory, NoHeader, Rdb and Tab. Please refer to original module documentation for what type each means.

5. You think you have it all covered, let's see - can you help me load only specific columns instead of all the columns. You know, be a little memory friendly when it is possible. Hmm... Sure, here we go.

   ```
   data = asciitable.read("MyTable.txt", include_names =
   ['MyChoice1',MyChoice2'])
   ```

   Tired of typing the names of the columns, think it is better to tell what not to take rather than what to take? This is how you do it.

   ```
   data = asciitable.read("MyTable.txt",
   exclude_names = ['NoNo1','Nono2'])
   ```

6. Hey, just want to load some specific lines, say 3rd to 7th line including the 7th line.

   ```
   data = asciitable.read("MyTable.txt",
   data_start=2, data_end=8)
   ```

   And if you want to load all but last 3 lines.

   ```
   data = asciitable.read("MyTable.txt", data_end = -3)
   ```

7. That's impressive, but do you support comment lines and if so, can you tell me how comments are recognised and can I change this behavior? Sure, any line or a part of line followed by a # (for ascii tables) will be ignored and will be called as a comment line. If you use something like %, for eg. we can help you with that.

   ```
   data = asciitable.read("MyTable.txt", comment="\s*\%")
   ```

8. My header line i.e. the line in which the names of the columns are stored is positioned in an unusual place, can I tell you how to load that specific line as the header? Sure, you can!

   ```
   data = asciitable.read("MyTable.txt", header_start=3)
   ```

9. My file has missing values, please help. Sure, we will. But first, we need to tell you how things work. Every single element in a table starts out as a string when loading the file. Then a parser finds out if all elements in a column are numbers or not. If so, they are converted into proper operatable numbers. So, if you have something like ... or ? in situations where values are missing, the entire column can be converted to a string in such situations. So, if you have missing values, it is a good idea to declare them at the beginning.

   ```
   data = asciitable.read("MyTable.txt",
   fill_values = ("","0"))
   ```

   In the above, for all columns, wherever you find empty spaces, zeros are placed. Not happy, right? You want to be able to define different behaviors for different columns, so this is how you can proceed.

   ```
   data = asciitable.read("MyTable.txt",fill_values=
   ("","0","x","y"))
   ```

   Now, only x and y columns are processed in such a way. To really define different methods of dealing with missing values for different columns,

   ```
   data = asciitable.read("MyTable.txt",fill_values =
   [("","0","x","y"),("","-999","z")])
   ```

   If you want to to define one rule for two columns and define another universal rule for the rest of the columns, this can be achieved by saying,

   ```
   data = asciitable.read("MyTable.txt", fill_values =
   [("","0","x","y"),("","-999")])
   ```

10. Okay, now I'm just being deliberately insane but is there some alternate way in which you can store the data instead of Numpy arrays. Hahaha! Yes, of course! WHAT?! Yes, yes, you can...even though we are not sure why...

    ```
    data = asciitable.read("MyTable.txt", numpy=False)
    ```

    The above statement will then create a Python dictionary which stores the table.

11. Hmm... fine, I think that's it, no wait, can you tell me if there is a way to tell that a single value "Sid is Kid" is to be treated as a complete text and not as separate columns. Yes, if you can make sure your file has a quote or some mark around such values, you can specify this...

    ```
    data = asciitable.read("MyTable.txt", quotechar="'")
    ```

12. Okay, now tell me if it is possible to override the data types that asciitable assigns to various columns. For example, is it possible to force feed the data definitions while loading a table. Well, yes, you can!

    We begin by defining a dictionary as follows...

    ```
    converters = {'col1': [asciitable.convert_list(int)],
          'col2': [asciitable.convert_list(float)]
          'col3': [asciitable.convert_list(str)]}

    data = asciitable.read("MyTable.txt", converters=converters)
    ```

Well, that is how you can load, with so much flexibility, large amount of data into a single construct which can then be used for your purpose. That concludes our section on how to load different data types.

3 Saving New Tables

It is possible to have some data which exists in a tabular format and then save it in some format using asciitable. Just as read() was used in order to open an existing table, there is a method called write() available for saving files but let us start with a primary question - what can you save as a table?

The simplest form is a system of equal length arrays.

```
x = np.array([1,3,4])
y = x**2
asciitable.write({"x": x, "y":y },"NewFile.txt", names=["x","y"])
```

So, we have supplied a dictionary that forms a one-one correspondence between the names of the columns and the corresponding arrays which represent the data that is thrown into those columns. Then, we of course have the name of the file and a specific listing of all the columns.

If you performed some operation on a "data" already read using the read() function, it is much simple to execute this operation.

```
asciitable.write(data, "NewFile")
```

Remember, it is always possible to supply a FileObject as the target to which a table is written. For those who know advanced level Python programming, you might remember that there is a command sys.stdout which is a FileObject and whenever data is written out to such a file object, the output comes directly on the screen. So, we can say,

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

The next approach utilises lists of lists or lists of tuples. This goes like this,

```
data = [ [1,3,4], [5,6,7], [1,5,6] ]
asciitable.write(data, "NewFile.txt")
```

The number of elements in the inner list or tuple is regarded as a column while each sublist is a row entry. A similar result can be achieved using dictionaries as follows.

```
someDict = { "x":[1,3,4], "y":["Tom", "Dick", "harry"],
"z":[1,3,5] }
asciitable.write(someDict, "NewFile.txt")
```

And of course, you can use a triple quote string to create the table as follows,

```
data = """
# x, y,z
1,3,4
5,6,6
2,4,5"""
asciitable.write(data, "NewFIle.txt")
```

That is neat. Okay, as usual I have some questions for you. Shall we start? Yes, of course.

1. The first thing I notice immediately is that I am confused about which format the output will be in! An ASCII table, if so, what will be the delimiter? The most basic format is called "Basic" which gives you a simple ASCII whitespace separated table. But you can control the output format as follows.

   ```
   asciitable.write(data, Writer=WriterObjectType)
   ```

   The WriterObjectType is going to be similar to the reader object.

2. Can I specify a delimiter on my own? Yes -

   ```
   asciitable.write(data, delimiter="|")
   ```

3. Can I tell how the comment line is included in whatever I use to create the table.

   ```
   asciitable.write(data, comment="#")
   ```

4. If my input to the write() does not include any obvious names to the column, can I specify them now while writing the table. Yes, you can!

   ```
   asciitable.write(data, names=["Name","Age","Gender"])
   ```

5. Can I choose to output only some columns? Yes, it's the same as a the read() function,

   ```
   asciitable.write(data, include_names = ["Name","Age"])
   ```

   or

   ```
   asciitable.write(data, exclude_names = ["Gender"])
   ```

6. Can I define fill values?

   ```
   asciitable.write(data, fill_values =
   [ ("nan","N/A"), (-999.0,"Not Defined") ] )
   ```

That's it. You now know how to read table files and write them.