PTable Documentation

Release latest

May 02, 2015

Contents

TUTORIAL ON HOW TO USE THE PRETTYTABLE 0.6+ API

This tutorial is distributed with PrettyTable and is meant to serve as a "quick start" guide for the lazy or impatient. It is not an exhaustive description of the whole API, and it is not guaranteed to be 100% up to date. For more complete and update documentation, check the PrettyTable wiki at http://code.google.com/p/prettytable/w/list

Let's suppose you have a shiny new PrettyTable:

```
from prettytable import PrettyTable
x = PrettyTable()
```

and you want to put some data into it. You have a few options.

CHAPTER 1

Row by row

You can add data one row at a time. To do this you can set the field names first using the `field_names` attribute, and then add the rows one at a time using the `add_row` method:

```
x.field_names = ["City name", "Area", "Population", "Annual Rainfall"]
x.add_row(["Adelaide",1295, 1158259, 600.5])
x.add_row(["Brisbane",5905, 1857594, 1146.4])
x.add_row(["Darwin", 112, 120900, 1714.7])
x.add_row(["Hobart", 1357, 205556, 619.5])
x.add_row(["Sydney", 2058, 4336374, 1214.8])
x.add_row(["Melbourne", 1566, 3806092, 646.9])
x.add_row(["Perth", 5386, 1554769, 869.4])
```

CHAPTER 2

Column by column

You can add data one column at a time as well. To do this you use the `add_column` method, which takes two arguments - a string which is the name for the field the column you are adding corresponds to, and a list or tuple which contains the column data"

```
x.add_column("City name",
["Adelaide","Brisbane","Darwin","Hobart","Sydney","Melbourne","Perth"])
x.add_column("Area", [1295, 5905, 112, 1357, 2058, 1566, 5386])
x.add_column("Population", [1158259, 1857594, 120900, 205556, 4336374, 3806092,
1554769])
x.add_column("Annual Rainfall",[600.5, 1146.4, 1714.7, 619.5, 1214.8, 646.9,
869.4])
```

CHAPTER 3

Mixing and matching

If you really want to, you can even mix and match `add_row` and `add_column` and build some of your table in one way and some of it in the other. There's a unit test which makes sure that doing things this way will always work out nicely as if you'd done it using just one of the two approaches. Tables built this way are kind of confusing for other people to read, though, so don't do this unless you have a good reason.

CHAPTER 4

Importing data from a CSV file

If you have your table data in a comma separated values file (.csv), you can read this data into a PrettyTable like this:

```
from prettytable import from_csv
fp = open("myfile.csv", "r")
mytable = from_csv(fp)
fp.close()
```

CHAPTER 5

Importing data from a database cursor

If you have your table data in a database which you can access using a library which confirms to the Python DB-API (e.g. an SQLite database accessible using the sqlite module), then you can build a PrettyTable using a cursor object, like this:

```
import sqlite3
from prettytable import from_cursor

connection = sqlite3.connect("mydb.db")
cursor = connection.cursor()
cursor.execute("SELECT field1, field2, field3 FROM my_table")
mytable = from_cursor(cursor)
```

CHAPTER 6

Getting data out

There are three ways to get data out of a PrettyTable, in increasing order of completeness:

- The `del_row` method takes an integer index of a single row to delete.
- The `clear_rows` method takes no arguments and deletes all the rows in the table - but keeps the field names as they were so you that you can repopulate it with the same kind of data.
- The `clear` method takes no arguments and deletes all rows and all field names. It's not quite the same as creating a fresh table instance, though - style related settings, discussed later, are maintained.

CHAPTER 7

Displaying your table in ASCII form

PrettyTable's main goal is to let you print tables in an attractive ASCII form, like this:

```
+-----------+------+------------+-----------------+
| City name | Area | Population | Annual Rainfall |
+-----------+------+------------+-----------------+
| Adelaide  | 1295 |  1158259   |      600.5      |
| Brisbane  | 5905 |  1857594   |      1146.4     |
| Darwin    | 112  |   120900   |      1714.7     |
| Hobart    | 1357 |   205556   |      619.5      |
| Melbourne | 1566 |  3806092   |      646.9      |
| Perth     | 5386 |  1554769   |      869.4      |
| Sydney    | 2058 |  4336374   |      1214.8     |
+-----------+------+------------+-----------------+
```

You can print tables like this to `stdout` or get string representations of them.

7.1 Printing

To print a table in ASCII form, you can just do this:

```
print x
```

in Python 2.x or:

```
print(x)
```

in Python 3.x.

The old x.printt() method from versions 0.5 and earlier has been removed.

To pass options changing the look of the table, use the get_string() method documented below:

```
print x.get_string()
```

7.2 Stringing

If you don't want to actually print your table in ASCII form but just get a string containing what *would* be printed if you use "print x", you can use the `get_string` method:

```
mystring = x.get_string()
```

This string is guaranteed to look exactly the same as what would be printed by doing "print x". You can now do all the usual things you can do with a string, like write your table to a file or insert it into a GUI.

7.3 Controlling which data gets displayed

If you like, you can restrict the output of `print x` or `x.get_string` to only the fields or rows you like.

The `fields` argument to these methods takes a list of field names to be printed:

```
print x.get_string(fields=["City name", "Population"])
```

gives:

```
+-----------+------------+
| City name | Population |
+-----------+------------+
| Adelaide  |  1158259   |
| Brisbane  |  1857594   |
| Darwin    |   120900   |
| Hobart    |   205556   |
| Melbourne |  3806092   |
| Perth     |  1554769   |
| Sydney    |  4336374   |
+-----------+------------+
```

The `start` and `end` arguments take the index of the first and last row to print respectively. Note that the indexing works like Python list slicing - to print the 2nd, 3rd and 4th rows of the table, set `start` to 1 (the first row is row 0, so the second is row 1) and set `end` to 4 (the index of the 4th row, plus 1):

```
print x.get_string(start=1,end=4)
```

prints:

```
+-----------+------+------------+-----------------+
| City name | Area | Population | Annual Rainfall |
+-----------+------+------------+-----------------+
| Brisbane  | 5905 |    1857594 | 1146.4          |
| Darwin    | 112  |     120900 | 1714.7          |
| Hobart    | 1357 |     205556 | 619.5           |
+-----------+------+------------+-----------------+
```

7.4 Changing the alignment of columns

By default, all columns in a table are centre aligned.

7.4.1 All columns at once

You can change the alignment of all the columns in a table at once by assigning a one character string to the `align` attribute. The allowed strings are "l", "r" and "c" for left, right and centre alignment, respectively:

```
x.align = "r"
print x
```

gives:

```
+-----------+------+------------+-----------------+
| City name | Area | Population | Annual Rainfall |
+-----------+------+------------+-----------------+
|  Adelaide | 1295 |    1158259 |           600.5 |
|  Brisbane | 5905 |    1857594 |          1146.4 |
|    Darwin |  112 |     120900 |          1714.7 |
|    Hobart | 1357 |     205556 |           619.5 |
| Melbourne | 1566 |    3806092 |           646.9 |
|     Perth | 5386 |    1554769 |           869.4 |
|    Sydney | 2058 |    4336374 |          1214.8 |
+-----------+------+------------+-----------------+
```

7.4.2 One column at a time

You can also change the alignment of individual columns based on the corresponding field name by treating the `align` attribute as if it were a dictionary.

```
x.align["City name"] = "l"
x.align["Area"] = "c"
x.align["Population"] = "r"
x.align["Annual Rainfall"] = "c"
print x
```

gives:

```
+-----------+------+------------+-----------------+
| City name | Area | Population | Annual Rainfall |
+-----------+------+------------+-----------------+
| Adelaide  | 1295 |    1158259 |      600.5      |
| Brisbane  | 5905 |    1857594 |      1146.4     |
| Darwin    | 112  |     120900 |      1714.7     |
| Hobart    | 1357 |     205556 |      619.5      |
| Melbourne | 1566 |    3806092 |      646.9      |
| Perth     | 5386 |    1554769 |      869.4      |
| Sydney    | 2058 |    4336374 |      1214.8     |
+-----------+------+------------+-----------------+
```

7.5 Sorting your table by a field

You can make sure that your ASCII tables are produced with the data sorted by one particular field by giving `get_string` a `sortby` keyword argument, which > must be a string containing the name of one field.

For example, to print the example table we built earlier of Australian capital city data, so that the most populated city comes last, we can do this:

```
print x.get_string(sortby="Population")
```

to get

```
+-----------+------+------------+-----------------+
| City name | Area | Population | Annual Rainfall |
+-----------+------+------------+-----------------+
| Darwin    | 112  |   120900   |      1714.7     |
| Hobart    | 1357 |   205556   |      619.5      |
| Adelaide  | 1295 |  1158259   |      600.5      |
```

```
| Perth     | 5386 |  1554769   |      869.4      |
| Brisbane  | 5905 |  1857594   |      1146.4     |
| Melbourne | 1566 |  3806092   |      646.9      |
| Sydney    | 2058 |  4336374   |      1214.8     |
+-----------+------+------------+-----------------+
```

If we want the most populated city to come *first*, we can also give a `reversesort=True` argument.

If you *always* want your tables to be sorted in a certain way, you can make the setting long term like this:

```
x.sortby = "Population"
print x
print x
print x
```

All three tables printed by this code will be sorted by population (you could do `x.reversesort = True` as well, if you wanted). The behaviour will persist until you turn it off:

```
x.sortby = None
```

If you want to specify a custom sorting function, you can use the `sort_key` keyword argument. Pass this a function which accepts two lists of values and returns a negative or positive value depending on whether the first list should appeare before or after the second one. If your table has n columns, each list will have n+1 elements. Each list corresponds to one row of the table. The first element will be whatever data is in the relevant row, in the column specified by the `sort_by` argument. The remaining n elements are the data in each of the table's columns, in order, including a repeated instance of the data in the `sort_by` column.

CHAPTER 8

Changing the appearance of your table - the easy way

By default, PrettyTable produces ASCII tables that look like the ones used in SQL database shells. But if can print them in a variety of other formats as well. If the format you want to use is common, PrettyTable makes this very easy for you to do using the `set_style` method. If you want to produce an uncommon table, you'll have to do things slightly harder (see later).

8.1 Setting a table style

You can set the style for your table using the `set_style` method before any calls to `print` or `get_string`. Here's how to print a table in a format which works nicely with Microsoft Word's "Convert to table" feature:

```
from prettytable import MSWORD_FRIENDLY
x.set_style(MSWORD_FRIENDLY)
print x
```

In addition to `MSWORD_FRIENDLY` there are currently two other in-built styles you can use for your tables:

- `DEFAULT` - The default look, used to undo any style changes you may have made
- `PLAIN_COLUMN` - A borderless style that works well with command line programs for columnar data

Other styles are likely to appear in future releases.

CHAPTER 9

Changing the appearance of your table - the hard way

If you want to display your table in a style other than one of the in-built styles listed above, you'll have to set things up the hard way.

Don't worry, it's not really that hard!

9.1 Style options

PrettyTable has a number of style options which control various aspects of how tables are displayed. You have the freedom to set each of these options individually to whatever you prefer. The `set_style` method just does this automatically for you.

The options are these:

- `border` - A boolean option (must be `True` or `False`). Controls whether > > or not a border is drawn around the table.
- `header` - A boolean option (must be `True` or `False`). Controls whether > > or not the first row of the table is a header showing the names of all the > > fields.
- `hrules` - Controls printing of horizontal rules after rows. Allowed > > values: FRAME, HEADER, ALL, NONE - note that these are variables defined > > inside the `prettytable` module so make sure you import them or use > > `prettytable.FRAME` etc.
- `vrules` - Controls printing of vertical rules between columns. Allowed > > values: FRAME, ALL, NONE.
- `int_format` - A string which controls the way integer data is printed. > > This works like: `print "%<int_format>d" % data`
- `float_format` - A string which controls the way floating point data is > > printed. This works like: `print "%<int_format>f" % data`
- `padding_width` - Number of spaces on either side of column data (only used > > if left and right paddings are None).
- `left_padding_width` - Number of spaces on left hand side of column data.
- `right_padding_width` - Number of spaces on right hand side of column data.
- `vertical_char` - Single character string used to draw vertical lines. > > Default is `|`.
- `horizontal_char` - Single character string used to draw horizontal lines. > > Default is `-`.
- `junction_char` - Single character string used to draw line junctions. > > Default is `+`.

You can set the style options to your own settings in two ways:

9.2 Setting style options for the long term

If you want to print your table with a different style several times, you can set your option for the "long term" just by changing the appropriate attributes. If you never want your tables to have borders you can do this:

```
x.border = False
print x
print x
print x
```

Neither of the 3 tables printed by this will have borders, even if you do things like add extra rows inbetween them. The lack of borders will last until you do:

```
x.border = True
```

to turn them on again. This sort of long term setting is exactly how `set_style` works. `set_style` just sets a bunch of attributes to pre-set values for you.

Note that if you know what style options you want at the moment you are creating your table, you can specify them using keyword arguments to the constructor. For example, the following two code blocks are equivalent:

```
x = PrettyTable()
x.border = False
x.header = False
x.padding_width = 5
```

x = PrettyTable(border=False, header=False, padding_width=5)

9.3 Changing style options just once

If you don't want to make long term style changes by changing an attribute like in the previous section, you can make changes that last for just one `get_string` by giving those methods keyword arguments. To print two "normal" tables with one borderless table between them, you could do this:

```
print x
print x.get_string(border=False)
print x
```

CHAPTER 10

Displaying your table in HTML form

PrettyTable will also print your tables in HTML form, as `<table>`s. Just like in ASCII form, you can actually print your table - just use `print_html()` - or get a string representation - just use `get_html_string()`. HTML printing supports the `fields`, `start`, `end`, `sortby` and `reversesort` arguments in exactly the same way as ASCII printing.

10.1 Styling HTML tables

By default, PrettyTable outputs HTML for "vanilla" tables. The HTML code is quite simple. It looks like this:

```
<table>
    <tr>
        <th>City name</th>
        <th>Area</th>
        <th>Population</th>
        <th>Annual Rainfall</th>
    </tr>
    <tr>
        <td>Adelaide</td>
        <td>1295</td>
        <td>1158259</td>
        <td>600.5</td>
    <tr>
        <td>Brisbane</td>
        <td>5905</td>
        <td>1857594</td>
        <td>1146.4</td>
    ...
    ...
    ...
</table>
```

If you like, you can ask PrettyTable to do its best to mimick the style options that your table has set using inline CSS. This is done by giving a `format=True` keyword argument to either the `print_html` or `get_html_string` methods. Note that if you *always* want to print formatted HTML you can do:

```
x.format = True
```

and the setting will persist until you turn it off.

Just like with ASCII tables, if you want to change the table's style for just one `print_html` or one `get_html_string` you can pass those methods keyword arguments - exactly like `print` and `get_string`.

10.2 Setting HTML attributes

You can provide a dictionary of HTML attribute name/value pairs to the `print_html` and `get_html_string` methods using the `attributes` keyword argument. This lets you specify common HTML attributes like `name`, `id` and `class` that can be used for linking to your tables or customising their appearance using CSS. For example:

```
x.print_html(attributes={"name":"my_table", "class":"red_table"})
```

will print:

```
<table name="my_table" class="red_table">
    <tr>
        <th>City name</th>
        <th>Area</th>
        <th>Population</th>
        <th>Annual Rainfall</th>
    </tr>
    ...
    ...
    ...
</table>
```

CHAPTER 11

Miscellaneous things

11.1 Copying a table

You can call the `copy` method on a PrettyTable object without arguments to return an identical independent copy of the table.

If you want a copy of a PrettyTable object with just a subset of the rows, you can use list slicing notation:

```
new_table = old_table[0:5]
```