prettytable - Tutorial.wiki

Export to GitHub

PrettyTable is best explained by example. It is designed to let you write something nice and simple like this:

``` from prettytable import PrettyTable

x = PrettyTable(["City name", "Area", "Population", "Annual Rainfall"]) x.align["City name"] = "l" # Left align city names x.padding_width = 1 # One space between column edges and contents (default) 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]) print x ```

in order to get something that looks 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 | | Sydney | 2058 | 4336374 | 1214.8 | | Melbourne | 1566 | 3806092 | 646.9 | |
Perth | 5386 | 1554769 | 869.4 | +-----------+------+------------+-----------------+
```

PrettyTable makes it easy to display data in a nice and neat but non-graphical format that you can use in command line programs, automatically generated emails, etc. Note that the results won't look right unless you use a monospace font.

## Getting data into tables

### Building tables manually

There are several ways to manually enter data into a table.

You can use x.add_row to build a table up row-by-row, like in the example at the start of this document.
```

You can also use x.add_column to build the table up column-by-column instead of row-by-row, if you're so inclined:

```
x = PrettyTable() 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])
```

You can even build a table using a combination of add_row and add_column if you're careful. It's probably not a good idea, but PrettyTable will work as long as you do it properly.

Building tables from existing sources

The quickest way to get data into a PrettyTable object is to build a table from data which is already stored in some table-like format.
Building from CSV files

If you have some data in a comma separated value (CSV) file, say "mytable.csv", you can make a PrettyTable of it as follows:

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

The variable pt will then be a fully populated PrettyTable object.

Note that the first row of the CSV file will be interpreted as the field names and used for the table header. If you don't want this behaviour, you can specify your own field names by passing a "field_names" keyword argument. In this case, all rows of the CSV file will be entered as table data.

Building from HTML code

If you have some data in a string of HTML code using `<table>`, say in the variable html_string, you can make a PrettyTable of it as follows

```
from prettytable import from_html pts = from_html(html_string)
```

The variable pts will then be a list of PrettyTable objects. There will be one PrettyTable for each `<table>` in the HTML code.

If you know for a fact that your HTML code will only contain one table, then instead of using from_html to get a list of length one, you can use the from_html_one function which will return just one PrettyTable object directly. If you give from_html_one a string containing several `<table>`s, it will throw an Exception.

Building from SQL databases

If you have some data in a database which has a Python API compatible with the DB-API2 specification (for example, the sqlite3 API in the standard library), you can bild a PrettyTable out of the results of a SELECT statement as follows

```

# db_cur is a Cursor object for your database

from prettytable import from_db_cursor db_cur.execute("SELECT * FROM mytable") pt = from_db_cursor(db_cur) ```

## Printing and getting strings

### ASCII tables

Once you've put data into a table, you can print the whole thing out like this `print x # Python 2.x print(x) # Python 3.x`

If you don't want to print a table but just get a string which you can, e.g., write to a file or put in an email, you can use: `string = x.get_string()` The get_string method also takes a lot of arguments you can use to change the appearance of your table. If you want to take advantage of these abilities when printing a table, you can just print the returned string:

`print x.get_string(border=False, padding_width=5) # Python 2.x print(x.get_string(border=False, padding_width=5)) # Python 3.x`

Note that while these are called "ASCII tables", this is just refering to the visual style of the tables. The get_string() method returns a unicode object, not an ASCII-encoded byte string.

### HTML tables

Instead of an ASCII table, you can call get_html_string to get a HTML `<table>` structure.

By default, the `<table>` returned by get_html_string will be a fairly plain looking table in a web browser, and will not necessarily reflect e.g. your PrettyTable's settings regarding alignment, line ruling, padding etc. If you call get_html_string with the format=True keyword argument, PrettyTable will generate code which attempts to match these settings as best as possible within the constraints of the HTML table specification.

You can use the "attribute" keyword argument to pass a dictionary of HTML attributes that should appear in the opening `<table>` tag, e.g. if you want to get "`<table class="foo">`", so that you can use CSS to style your table instead of relying on PrettyTable's format=True option, then use:
```

```
x.get_html_string(attributes = {"class": "foo"})
```

Selecting subsets of data

If you're only interested in showing some of the fields in your table, you can do this:

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

to get a "sub table":

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

You can print only the first 3 rows of the table by doing this:

```
print x.get_string(start=0,end=3)
```

which gives you this:

```
+----------+------+------------+-----------------+ | City name | Area | Population | Annual Rainfall | +----------+------+------------+-----------------+ | Adelaide | 1295 | 1158259 | 600.5 | | Brisbane | 5905 | 1857594 | 1146.4 | | Darwin | 112 | 120900 | 1714.7 | +----------+------+------------+-----------------+
```

If you want to permanently get rid of all the rows other than the first 3, you can create a new PrettyTable object with only those rules by slicing the table as if it were a Python list:

```
new_table = old_table[0:3]
```

Sorting tables

You can sort the rows of your table by a particular column like this:

```
print x.get_string(sortby="Annual Rainfall")
```

which gives you this:

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

You can do

```
print x.get_string(sortby="Annual Rainfall", reversesort=True)
```

to sort the table in the reverse order (from most to least rainful).

If you are going to print the table several times and you always want to sort by a certain column, you can do

```
x.sortby = "Annual Rainfall" print x print x print x
```

and the sorting will happen each time. You can turn sorting off later with

```
x.sortby = None
```

Similarly, you can set reversesort to be permanently True or False in the same way.

If you want to sort using something other than the default comparison method, you can add a key function(http://wiki.python.org/moin/HowTo/Sorting/#Key_Functions) using the sort_key keyword argument.

Controlling table style

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).

Setting a table style

You can set the style for your table using the `set_style` method before any calls to `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_COLUMNS` - A borderless style that works well with command line programs for columnar data

Other preset styles may appear in future releases.

Manually changing table style

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.
- `header_style` - Controls capitalisation of field names in the header. Allowed values: "cap" (capitalise first letter of each word), "title" (title case), "upper" (all upper-case), "lower" (all lower-case) or None (don't change from original field name setting). Default is None.
- `hrules` - Controls printing of horizontal rules after rows. Allowed values: FRAME, 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
- `align` - Horizontal alignment (None, "l" (left), "c" (centre), "r" (right))
- `valign` - Vertical alignment (None, "t" (top), "m" (middle) or "b" (bottom))
- `int_format` - Controls formatting of integer data. This should be a string which can be placed between "%" and "d" in something like `print "%d" % 42`.
- `float_format` - Controls formatting of floating point data. This should be a string which can be placed between "%" and "f" in something like `print "%f" % 4.2`.
- `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:

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)
```

Making once-off style changes

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 call of `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
```

Per-column settings

Note that some settings can be controlled on a per-column basis if they are set using attribute assignment, not keyword arguments. If you set column alignment like this:

```
print x.get_string(align="l")
```

or this

```
x.align = "l"
```

then the alignment is set to left for all of the columns. But you can set the alignment for individual columns like this:

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

A subsequent use of

```
x.align = "l"
```

will overwrite these per-column settings, setting all columns to "l" again.