

THE IMPORTANCE OF DOCUMENTING CODE, AND HOW YOU MIGHT MAKE YOURSELF DO IT

Erik Tollerud

@eteq

Space Telescope Science Institute

Giacconi Fellow

Astropy Coordinating Committee Member



WHY DOCUMENT?

```
def T2rgb(Ts, fn='smj10q.csv', n=None, modw=None):  
    ...
```

```
def Temp_to_rgb(temps, eye_response_fn='smj10q.csv', normed_at=None, modulate_with=None):  
    """  
    Converts an array of blackbody temperatures to RGB  
  
    Parameters  
    -----  
    temps : numpy array  
        The temperatures to convert  
    eye_response_fn : str  
        Path to the file with the human eye cone response functions  
    normed_at : float or None  
        The temperature at which the responses should be 1, or None to apply no  
        rescaling of the response functions.  
    modulate_with : numpy array or None  
        A rescaling factor to multiply the output cones by, or None to do no  
        rescaling of the output.  
  
    Returns  
    -----  
    rgb : numpy array  
        The output RGB values. Has shape (3, ... shape of 'temps' ...)  
    """  
    ...
```

WHY DOCUMENT?

- “Dirty” coding / “Science” mode
- Public coding / “Developer” mode

WHY DOCUMENT?

- “Dirty” coding / “Science” mode
 - Do it for you
- Public coding / “Developer” mode

WHY DOCUMENT?

- “Dirty” coding / “Science” mode
 - Do it for you
 - Do it for reproducibility
- Public coding / “Developer” mode

WHY DOCUMENT?

- “Dirty” coding / “Science” mode
 - Do it for you
 - Do it for reproducibility
- Public coding / “Developer” mode
 - Write for others: provide background

SOME RULES TO MAKE YOURSELF DO IT

- Always document as you code
- In Python: use docstrings!

```
def Temp_to_rgb(temps, eye_response_fn='smj10q.csv', normed_at=None, modulate_with=None):  
    """  
    Converts an array of blackbody temperatures to RGB  
    Parameters  
    """
```

- Write code in modular pieces, and document the interfaces

- Write docs so that future you understands them

- Use descriptive names (even if long)

- Use a consistent style/format. (Ideally one that fits with your tools..)

```
1 FUNCTION FXPAR, HDR, NAME, ABORT, COUNT=MATCHES, COMMENT=COMMENTS, $  
2 START=START, PRECHECK=PRECHECK, POSTCHECK=POSTCHECK, $  
3 NOCONTINUE = NOCONTINUE, $  
4 DATATYPE=DATATYPE  
5  
6 ; NAME  
7 ; FXPAR()  
8 ; PURPOSE:  
9 ; Obtain the value of a parameter in a FITS header.  
10 ; EXPLANATION:  
11 ; The first 8 characters of each element of HDR are searched for a match to  
12 ; NAME. If the keyword is one of those allowed to take multiple values  
13 ; ("HISTORY", "COMMENT", or "BLANK"), then the value is taken  
14 ; as the next 32 characters. Otherwise, it is assumed that the next  
15 ; character is '=', and the value (and optional comment) is then parsed  
16 ; from the last 71 characters. An error occurs if there is no parameter  
17 ; with the given name.  
18 ;  
19 ; If the value is too long for one line, it may be continued on to the  
20 ; the next input card, using the CONTINUE Long String Keyword convention.  
21 ; For more info, http://fits.gsfc.nasa.gov/registry/continue\_keyword.html  
22 ;  
23 ; Complex numbers are recognized if two numbers separated by one or more  
24 ; space characters.  
25 ;  
26 ; If a numeric value has no decimal point (or E or D) it is returned as  
27 ; type LONG. If it contains more than 8 numerals, or contains the  
28 ; character 'D', then it is returned as type DOUBLE. Otherwise it is  
29 ; returned as type FLOAT. If an integer is too large to be stored as  
30 ; type LONG, then it is returned as DOUBLE.  
31 ;  
32 ; CALLING SEQUENCE:  
33 ; Result = FXPAR( HDR, NAME [, ABORT, COUNT=, COMMENT=, /NOCONTINUE ] )  
34 ;  
35 ;  
36 ; Result = FXPAR(HEADER, 'DATE') ; Finds the value of DATE  
37 ; Result = FXPAR(HEADER, 'NAXIS*') ; Returns array dimensions as  
38 ; ; vector  
39 ;  
40 ; REQUIRED INPUTS:  
41 ; HDR = FITS header string array (e.g., as returned by FXREAD). Each  
42 ; element should have a length of 80 characters  
43 ; NAME = String name of the parameter to return. If NAME is of the  
44 ; form 'keyword*' then an array is returned containing values  
45 ; of keywordN where N is an integer. The value of keywordN  
will be placed in RESULT(N-1). The data type of RESULT will
```

```
63 class FLRW(Cosmology):  
64     """ A class describing an isotropic and homogeneous  
65     (Friedmann-Lemaître-Robertson-Walker) cosmology.  
66  
67     This is an abstract base class -- you can't instantiate  
68     examples of this class, but must work with one of its  
69     subclasses such as "LambdaCDM" or "wCDM".  
70  
71     Parameters  
72  
73     H0 : float or scalar ~astropy.units.Quantity  
74         Hubble constant at z = 0. If a float, must be in (km/sec/Mpc)  
75  
76     Om0 : float  
77         Omega matter density of non-relativistic matter in units of the  
78         critical density at z=0. Note that this does not include  
79         massive neutrinos.  
80  
81     Ode0 : float  
82         Omega dark energy density of dark energy in units of the critical  
83         density at z=0.  
84  
85     Tcmb0 : float or scalar ~astropy.units.Quantity  
86         Temperature of the CMB at z=0. If a float, must be in (K). Default: 2.725.  
87         Setting this to zero will turn off both photons and neutrinos (even  
88         massive ones)  
89  
90     Neff : float  
91         Effective number of Neutrino species. Default: 3.04.  
92  
93     m_nu : ~astropy.units.Quantity  
94         Mass of each neutrino species. If this is a scalar Quantity, then all  
95         neutrino species are assumed to have that mass. Otherwise, the mass of  
96         each species. The actual number of neutrino species (and hence the  
97         number of elements of m_nu if it is not scalar) must be the floor of  
98         Neff. Usually this means you must provide three neutrino masses unless  
99         you are considering something like a sterile neutrino.  
100  
101  
102     name : str  
103         Optional name for this cosmological object.  
104  
105     Ob0 : float  
106         Omega baryon density of baryon matter in units of the critical  
107         density at z = 0.  
108  
109     Notes  
110     -----  
111     Class instances are static -- you can't change the values  
112     of the parameters. That is, all of the attributes above are  
113     read only.  
114  
115     """  
116     def __init__(self, H0=70, Om0=0.3, Ode0=0.7, Tcmb0=2.725, Neff=3.04,  
117                 m_nu=None, Ob0=None, name=None, Ode=None):  
118         # All cosmological parameters are read-only  
119         self._H0 = float(H0)  
120         if self._H0 < 0:  
121             raise ValueError("Hubble constant can not be negative")  
122         self._Om0 = float(Om0)  
123         if Om0 < 0:  
124             raise ValueError("Omega matter density can not be negative")  
125         self._Ode0 = float(Ode0)  
126         if Ode0 < 0:  
127             raise ValueError("Omega dark energy density can not be negative")  
128         self._Tcmb0 = float(Tcmb0)  
129         if Tcmb0 < 0:  
130             raise ValueError("CMB temperature can not be negative")  
131         self._Neff = float(Neff)  
132         if Neff < 0:  
133             raise ValueError("Effective number of degrees of freedom can not be negative")  
134         self._m_nu = m_nu  
135         self._Ob0 = float(Ob0)  
136         if Ob0 < 0:  
137             raise ValueError("Omega baryon density can not be negative")  
138         self._name = name
```


ONE STEP FURTHER: USE TOOLS



SPHINX

PYTHON DOCUMENTATION GENERATOR

de
xy

STANDARD



Doxygen



Java™



JavaDoc

USE TOOLS TO MAKE IT READABLE



SPHINX

PYTHON DOCUMENTATION GENERATOR

```
63 class FLRW(Cosmology):
64     """ A class describing an isotropic and homogeneous
65     (Friedmann-Lemaître-Robertson-Walker) cosmology.
66
67     This is an abstract base class -- you can't instantiate
68     examples of this class, but must work with one of its
69     subclasses such as LambdaCDM or wCDM.
70
71     Parameters
72     ~~~~~
73
74     H0 : float or scalar ~astropy.units.Quantity
75         Hubble constant at  $z = 0$ . If a float, must be in [km/sec/Mpc]
76
77     Om0 : float
78         Omega matter: density of non-relativistic matter in units of the
79         critical density at  $z=0$ . Note that this does not include
80         massive neutrinos.
81
82     Ode0 : float
83         Omega dark energy: density of dark energy in units of the critical
84         density at  $z=0$ .
85
86     Tcmb0 : float or scalar ~astropy.units.Quantity
87         Temperature of the CMB  $z=0$ . If a float, must be in [K]. Default: 2.725.
88         Setting this to zero will turn off both photons and neutrinos (even
89         massive ones)
90
91     Neff : float
92         Effective number of Neutrino species. Default 3.04.
93
94     m_nu : ~astropy.units.Quantity
95         Mass of each neutrino species. If this is a scalar Quantity, then all
96         neutrino species are assumed to have that mass. Otherwise, the mass of
97         each species. The actual number of neutrino species (and hence the
98         number of elements of m_nu if it is not scalar) must be the floor of
99         Neff. Usually this means you must provide three neutrino masses unless
100         you are considering something like a sterile neutrino.
101
102     name : str
103         Optional name for this cosmological object.
104
105     Ob0 : float
106         Omega baryons: density of baryonic matter in units of the critical
107         density at  $z=0$ .
108
109     Notes
110     ~~~~~
111     Class instances are static -- you can't change the values
112     of the parameters. That is, all of the attributes above are
113     read only.
114
115     """
116     def __init__(self, H0, Om0, Ode0, Tcmb0=2.725, Neff=3.04,
117                  m_nu=u.Quantity(0.0, u.eV), name=None, Ob0=None):
118
119         # all densities are in units of the critical density
120         self._Om0 = float(Om0)
121         if self._Om0 < 0.0:
122             raise ValueError("Matter density can not be negative")
123         self._Ode0 = float(Ode0)
124         if Ob0 is not None:
125             self._Ob0 = float(Ob0)
126         if self._Ob0 < 0.0:
```

```
class astropy.cosmology.FLRW(H0, Om0, Ode0, Tcmb0=2.725, Neff=3.04, m_nu=<Quantity 0.0 eV>,
name=None, Ob0=None) [edit on github][source]
```

Bases: `astropy.cosmology.core.Cosmology`

A class describing an isotropic and homogeneous (Friedmann-Lemaître-Robertson-Walker) cosmology.

This is an abstract base class -- you can't instantiate examples of this class, but must work with one of its subclasses such as `LambdaCDM` or `wCDM`.

Parameters: **H0** : float or scalar `Quantity`

Hubble constant at $z = 0$. If a float, must be in [km/sec/Mpc]

Om0 : float

Omega matter: density of non-relativistic matter in units of the critical density at $z=0$.

Ode0 : float

Omega dark energy: density of dark energy in units of the critical density at $z=0$.

Tcmb0 : float or scalar `Quantity`

Temperature of the CMB $z=0$. If a float, must be in [K]. Default: 2.725. Setting this to zero will turn off both photons and neutrinos (even massive ones)

Neff : float

Effective number of Neutrino species. Default 3.04.

m_nu : `Quantity`

Mass of each neutrino species. If this is a scalar `Quantity`, then all neutrino species are assumed to have that mass. Otherwise, the mass of each species. The actual number of neutrino species (and hence the number of elements of `m_nu` if it is not scalar) must be the floor of `Neff`. Usually this means you must provide three neutrino masses unless you are considering something like a sterile neutrino.

name : str

Optional name for this cosmological object.

Ob0 : float

Omega baryons: density of baryonic matter in units of the critical density at $z=0$.

Notes

Class instances are static -- you can't change the values of the parameters. That is, all of the attributes above are read only.

Attributes Summary

H0 Return the Hubble constant as an `Quantity` at $z=0$

USE TOOLS TO TELL A STORY



SPHINX

PYTHON DOCUMENTATION GENERATOR

```
index.rst
1 .. include:: references.txt
2
3 .. _astropy-table:
4
5 *****
6 Data Tables ('astropy.table')
7 *****
8
9 Introduction
10 =====
11
12 'astropy.table' provides functionality for storing and manipulating
13 heterogeneous tables of data in a way that is familiar to 'numpy' users. A few
14 notable capabilities of this package are:
15
16 * Initialize a table from a wide variety of input data structures and types.
17 * Modify a table by adding or removing columns, changing column names,
18   or adding new rows of data.
19 * Handle tables containing missing values.
20 * Include table and column metadata as flexible data structures.
21 * Specify a description, units and output formatting for columns.
22 * Interactively scroll through long tables similar to using ``more``.
23 * Create a new table by selecting rows or columns from a table.
24 * Perform :ref:`table\_operations` like database joins, concatenation, and binning.
25 * Maintain a table index for fast retrieval of table items or ranges.
26 * Manipulate multidimensional columns.
27 * Handle non-native (mixin) column types within table.
28 * Methods for :ref:`read\_write\_tables` to files.
29 * Hooks for :ref:`subclassing\_table` and its component classes.
30
31 Currently 'astropy.table' is used when reading an ASCII table using
32 'astropy.io.ascii'. Future releases of AstroPy are expected to use
33 the |Table| class for other subpackages such as 'astropy.io.votable' and 'astropy.io.fits'.
34
35 .. Note::
36
37 Starting with version 1.0 of astropy the internal implementation of the
38 |Table| class changed so that it no longer uses numpy structured arrays as
39 the core table data container. Instead the table is stored as a collection
40 of individual column objects. *For most users there is NO CHANGE to the
41 interface and behavior of |Table| objects.*
42
43 The page on :ref:`table\_implementation\_change` provides details about the
44 change. This includes discussion of the table architecture, key differences,
45 and benefits of the change.
46
47 Getting Started
```

astropy:docs

astropy v1.0.1 - Data Tables (`astropy.table`)

Index Modules Search

previous next

Page Contents

- Data Tables (`astropy.table`)
- Introduction
- Getting Started
- Using `table`
 - Construct table
 - Access table
 - Modify table
 - Table operations
 - Masking
 - I/O with tables
 - Mixin columns
 - Implementation
- Reference/API
 - `astropy.table` Module
 - Functions
 - Classes
 - Class Inheritance Diagram

Data Tables (`astropy.table`)

Introduction

`astropy.table` provides functionality for storing and manipulating heterogeneous tables of data in a way that is familiar to `numpy` users. A few notable features of this package are:

- Initialize a table from a wide variety of input data structures and types.
- Modify a table by adding or removing columns, changing column names, or adding new rows of data.
- Handle tables containing missing values.
- Include table and column metadata as flexible data structures.
- Specify a description, units and output formatting for columns.
- Interactively scroll through long tables similar to using `more`.
- Create a new table by selecting rows or columns from a table.
- Perform [Table operations](#) like database joins and concatenation.
- Manipulate multidimensional columns.
- Methods for [Reading and writing Table objects](#) to files
- Hooks for [Subclassing Table](#) and its component classes

Currently `astropy.table` is used when reading an ASCII table using `astropy.io.ascii`. Future releases of AstroPy are expected to use the `Table` class for other subpackages such as `astropy.io.votable` and `astropy.io.fits`.

Note

Starting with version 1.0 of astropy the internal implementation of the `Table` class changed so that it no longer uses numpy structured arrays as the core table data container. Instead the table is stored as a collection of individual column objects. For most users there is NO CHANGE to the interface and behavior of `|Table|` objects.

The page on [Table implementation change in 1.0](#) provides details about the change. This includes discussion of the table architecture, key differences, and benefits of the change.

Getting Started

The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full `astropy.table` documentation is available from the [Using table](#) section.

First create a simple table with three columns of data named `a`, `b`, and `c`. These columns have integer, float, and string values respectively:

```
>>> from astropy.table import Table
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'), meta={'name': 'first table'})
```


USE TOOLS TO COMBINE THE TWO

Narrative docs +
Reference is a powerful
combination!

Using `astropy.units`

- [Quantity](#)
 - [Creating Quantity instances](#)
 - [Converting to different units](#)
 - [Plotting quantities](#)
 - [Arithmetic](#)
 - [Numpy functions](#)
 - [Dimensionless quantities](#)
 - [Converting to plain Python scalars](#)
 - [Functions Accepting Quantities](#)
 - [Representing vectors with units](#)
 - [Known issues with conversion to numpy arrays](#)
 - [Subclassing Quantity](#)
- [Standard units](#)
 - [The dimensionless unit](#)
 - [Enabling other units](#)
- [Combining and defining units](#)
- [Decomposing and composing units](#)
 - [Reducing a unit to its irreducible parts](#)
 - [Automatically composing a unit into more complex units](#)
 - [Converting between systems](#)
- [Magnitudes and other Logarithmic Units](#)
 - [Creating Logarithmic Quantities](#)
 - [Converting to different units](#)
 - [Arithmetic](#)
 - [Numpy functions](#)
 - [Dimensionless logarithmic quantities](#)
- [String representations of units](#)
 - [Converting units to string representations](#)
 - [Creating units from strings](#)
 - [Built-in formats](#)
 - [Unrecognized Units](#)
- [Equivalencies](#)
 - [Built-in equivalencies](#)
 - [Writing new equivalencies](#)
 - [Displaying available equivalencies](#)
 - [Using equivalencies in larger pieces of code](#)
- [Low-level unit conversion](#)
 - [Direct Conversion](#)
 - [Incompatible Conversions](#)

See Also

- [FITS Standard for units in FITS.](#)
- [The Units in the VO 1.0 Standard for representing units in the VO.](#)
- [OGIP Units: A standard for storing units in OGIP FITS files.](#)
- [Standards for astronomical catalogues units.](#)
- [IAU Style Manual.](#)
- [A table of astronomical unit equivalencies](#)

Reference/API

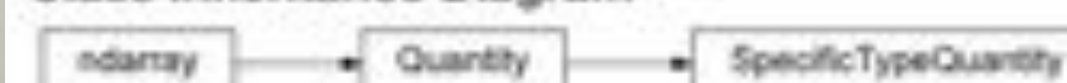
`astropy.units.quantity` Module

This module defines the `Quantity` object, which represents a number with some associated units. `Quantity` objects support operations like ordinary numbers, but will deal with unit conversions internally.

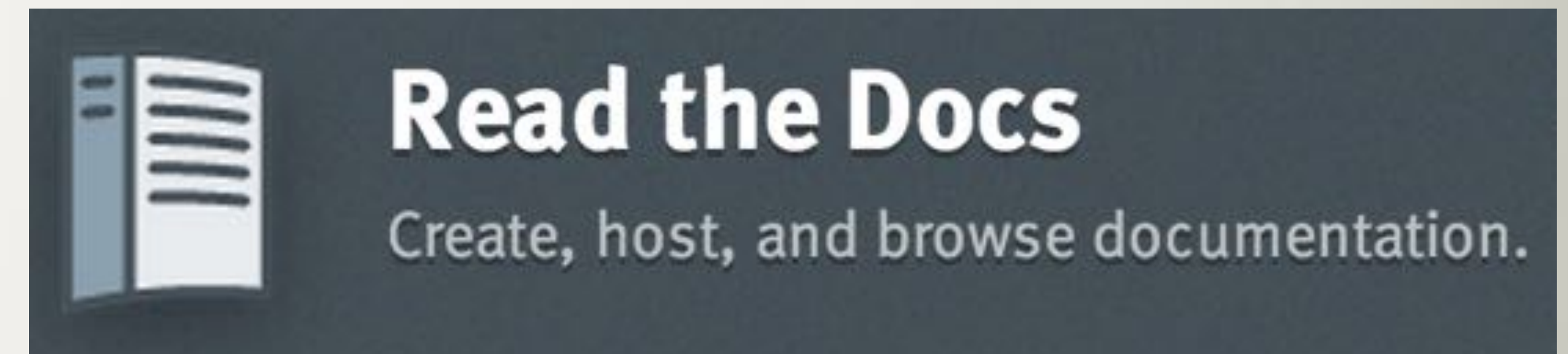
Classes

- | | |
|--------------------------------------|--|
| Quantity | A <code>Quantity</code> represents a number with some associated unit. |
| SpecificTypeQuantity | Superclass for Quantities of specific physical type. |

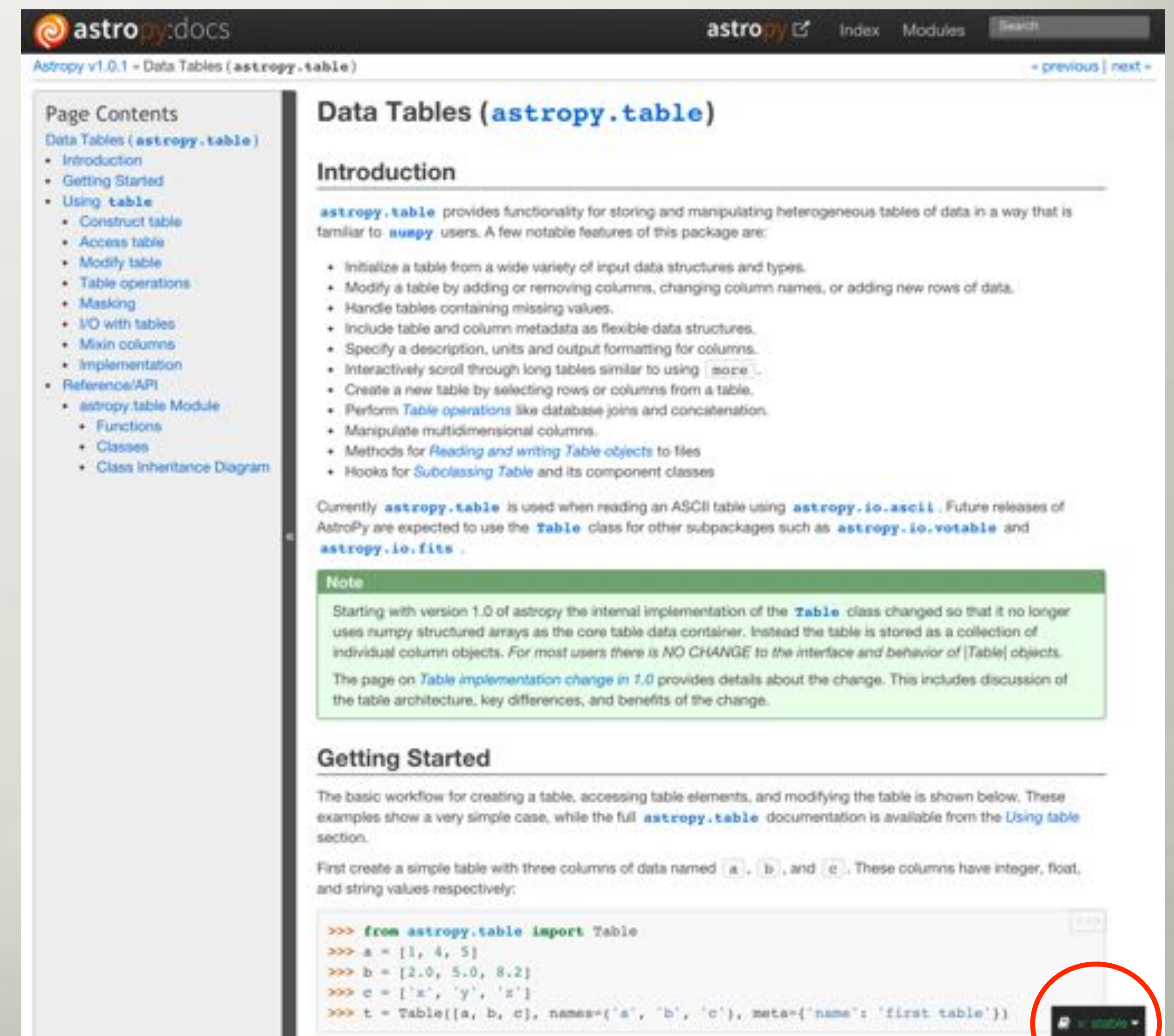
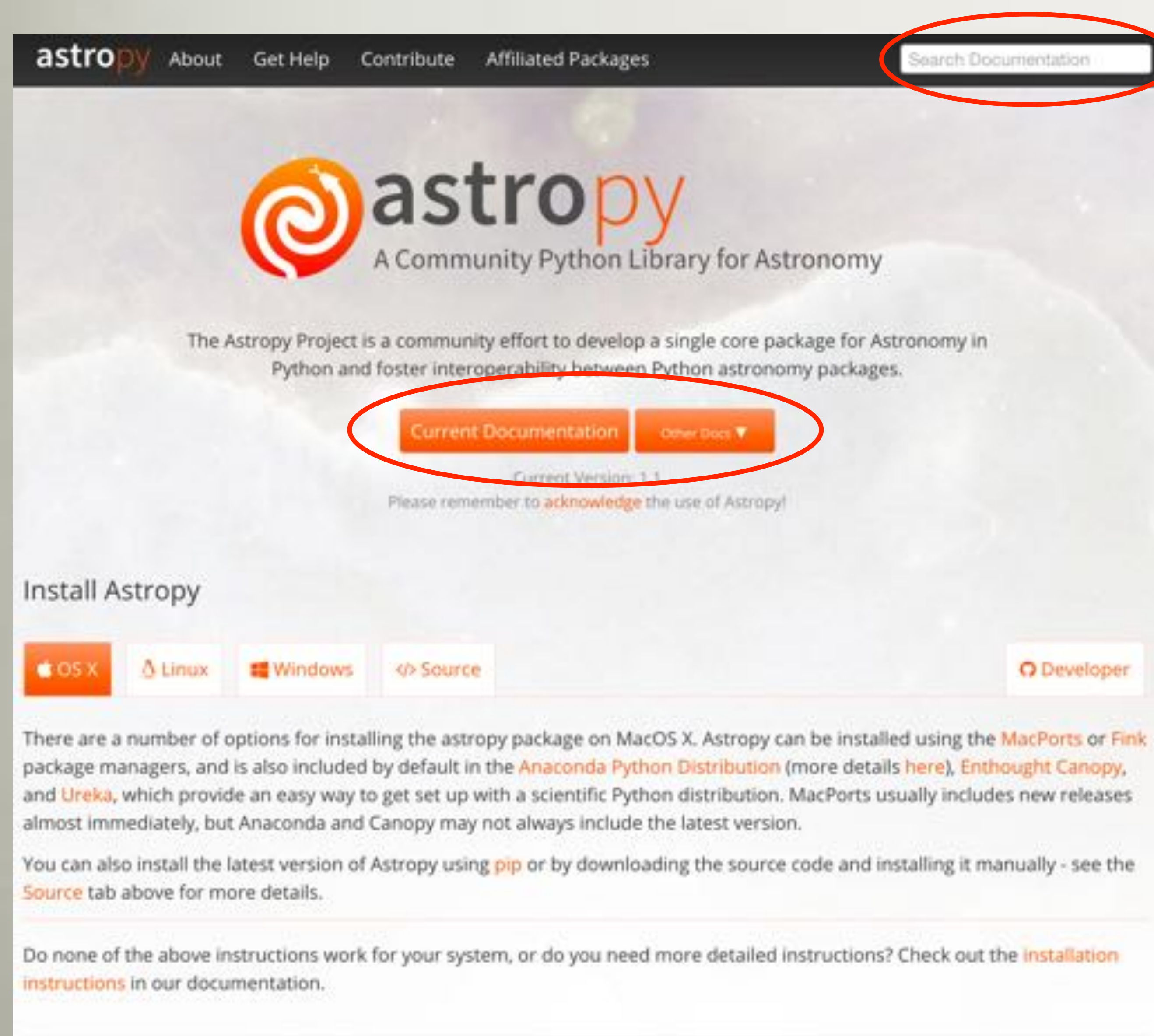
Class Inheritance Diagram



FOR PUBLIC CODE: FIT IT INTO A BIGGER PICTURE



<http://readthedocs.org>



IF YOU REMEMBER NOTHING ELSE, REMEMBER:

- Avoid “I’ll go back and document it after it’s working.” Trust me: you won’t.
- Write code with discrete chunks, and document the interfaces.
- Leads you to modular code