# 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

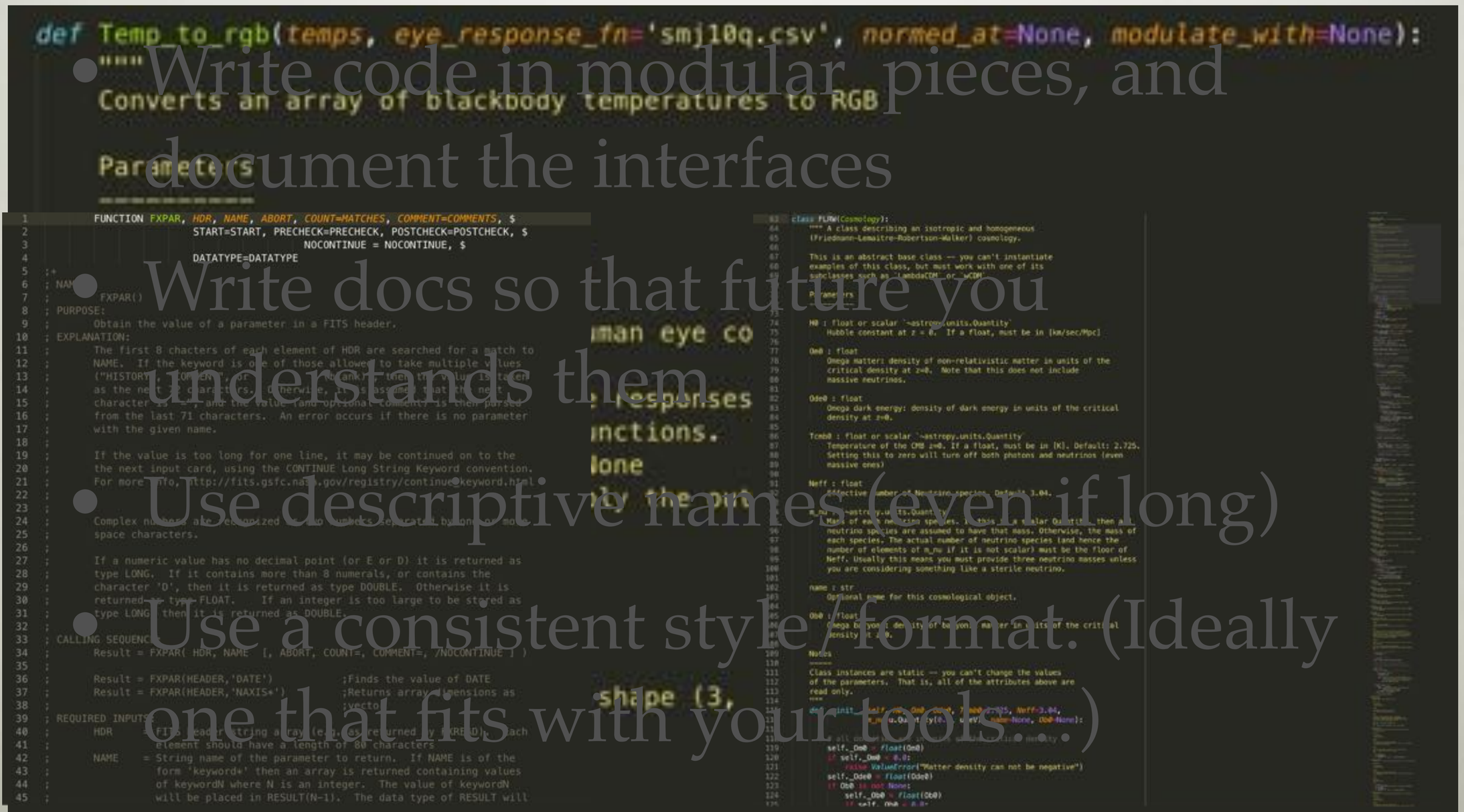
# Why Document?

- "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!
- 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…)



# One step Further: Use Tools

python

Sphinx
Python documentation generator

de
xy

STANDARD
C++

Doxygen

Java

JavaDoc

# Use Tools To Make It Readable

python

SPHINX

PYTHON DOCUMENTATION GENERATOR

```
class FLRW(Cosmology):
    """ A class describing an isotropic and homogeneous
    (Friedmann-Lemaitre-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 `~astropy.units.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.  Note that this does not include
        massive neutrinos.

    Ode0 : float
        Omega dark energy: density of dark energy in units of the critical
        density at z=0.

    Tcmb0 : float or scalar `~astropy.units.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 : `~astropy.units.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.
    """
    def __init__(self, H0, Om0, Ode0, Tcmb0=2.725, Neff=3.04,
                 m_nu=u.Quantity(0.0, u.eV), name=None, Ob0=None):

        # all densities are in units of the critical density
        self._Om0 = float(Om0)
        if self._Om0 < 0.0:
            raise ValueError("Matter density can not be negative")
        self._Ode0 = float(Ode0)
        if Ob0 is not None:
            self._Ob0 = float(Ob0)
            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-Lemaitre-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

reST

Sphinx

Python documentation generator

index.rst

```
.. include:: references.txt

.. _astropy-table:

*******************************
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 capabilities 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 :ref:`table_operations` like database joins, concatenation, and binning.
* Maintain a table index for fast retrieval of table items or ranges.
* Manipulate multidimensional columns.
* Handle non-native (mixin) column types within table.
* Methods for :ref:`read_write_tables` to files.
* Hooks for :ref:`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 :ref:`table_implementation_change` provides details about the
   change.  This includes discussion of the table architecture, key differences,
   and benefits of the change.

Getting Started
```

astropy:docs — astropy — Index — Modules — Search

Astropy v1.0.1 » Data Tables (`astropy.table`) « 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

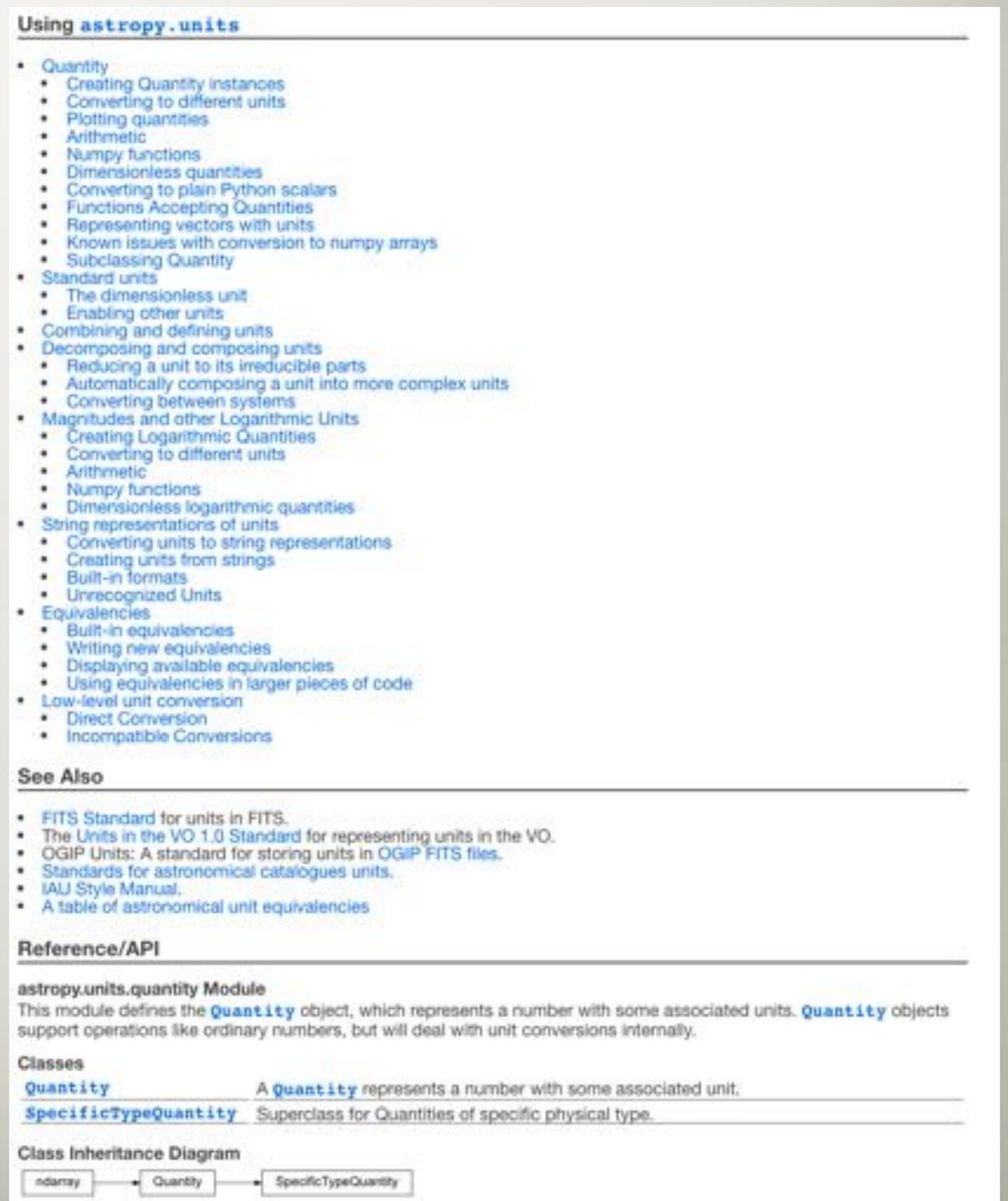
The basic workflow for creating a table, accessing table elements, and modifying the table is shown below. These examples show a very simple case, while the full `astropy.table` documentation is available from the Using table section.

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 `Quantity` represents a number with some associated unit. |
| `SpecificTypeQuantity` | Superclass for Quantities of specific physical type. |

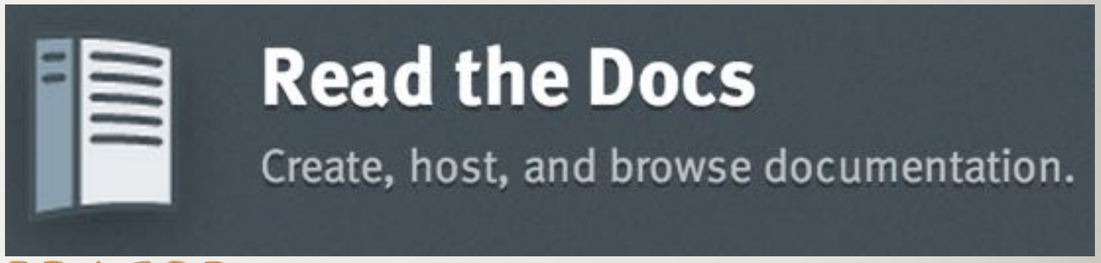
**Class Inheritance Diagram**

ndarray → Quantity → SpecificTypeQuantity

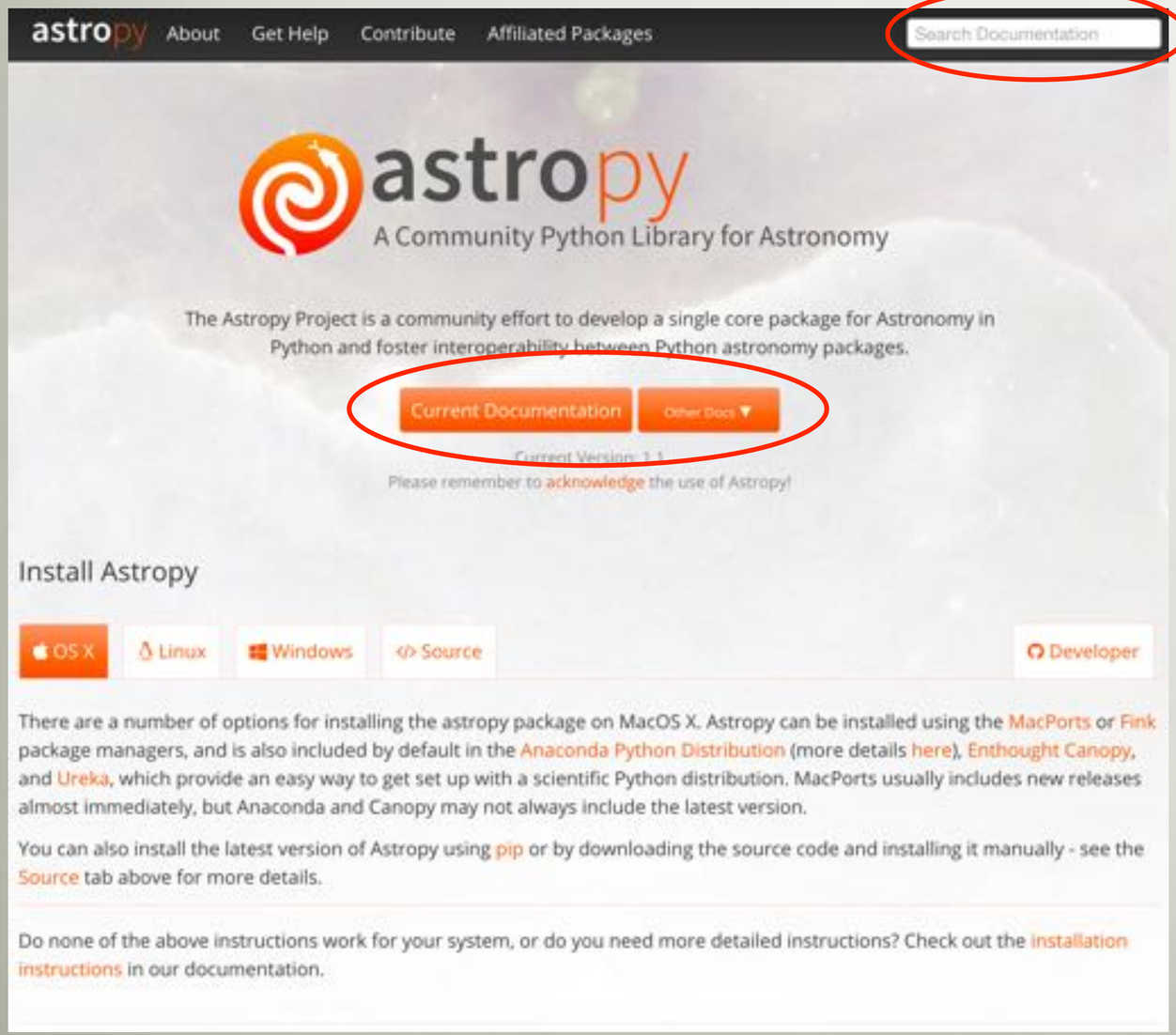
# For Public Code: Fit it into a Bigger Picture

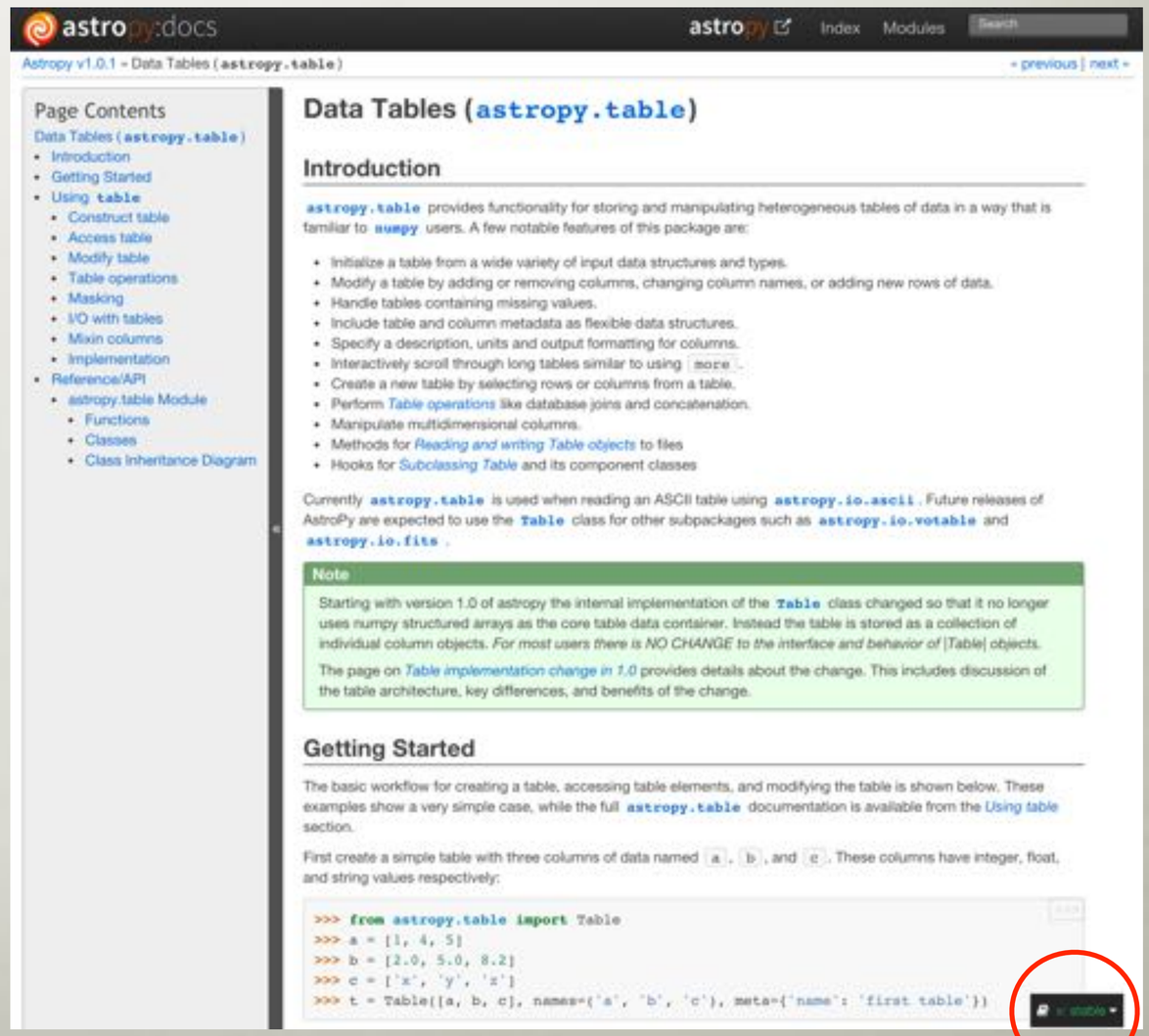
Sphinx

Python documentation generator




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