Documentation for Research Coding Projects

Matthew McKay

May 31, 2017

QuantEcon - Research School of Economics, ANU, Australia

Where to get this?

`https://github.com/mmcky/anu.macroreadinggroup.code-documentation.git`

Python

Table of Contents

'Good' Code Style (PEP 8)

Emphasis on **Readability**

This is good for **others** and good for **future self**

Python PEP 8

1. provides a set of rules and a standard for writing code
2. widely adopted by the Python Community
3. emphasis is on readability of code

Many editors assist with easily conforming to these types of guidelines and many can use Linters

Code Layout

Summary:

1. Indentation
2. Tabs or Spaces
3. Maximum Line Length
4. Should a line break before or after a binary operator?
5. Blank Lines
6. Source File Encoding
7. Importing

Naming Conventions

Summary:

1. Class Names are CapCase
2. Function Names are lowercase with words separated by an underscore
3. Function and Method Arguments
4. Constants are all CAPS

Comments

Summary:

1. Block Comments
2. Inline Comments
3. Documentation Strings

Docstrings and PEP 257

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.

This docstring becomes the `__doc__` special attribute of that object.

For consistency:

1. always use """triple double quotes""" around docstrings **(Most Common)**
2. Use r"""raw triple double quotes""" if you use any backslashes in your docstrings
3. Use u"""Unicode triple-quoted strings""" for unicode docstrings

PEP 257

The initial standard for writing docstrings in Projects

1. One Line Docstrings
2. Multi Line Docstrings
3. Handling Docstring Indentation

Docstring Example: One-line

One-line type docstrings

```
def add(a, b):
    """add two numbers together"""
    return a + b
```

the docstring is attached to the add function as the doc attribute `add.__doc__`

Docstring Example: Multi-line

```
def add(a, b, show=True):
    """add two numbers together

    Keyword arguments:
    show -- boolean, show the resulting value (default=True)
    """
    result = a + b
    if show:
        print("Result = {}".format(result))
    return result
```

Class Docstrings

```
class TestAdd(object):
  """This is a class docstring"""

  def __init__(self, a, b, show=True):
      """
      add two numbers together and store the result

      Keyword arguments:
      show -- boolean, show the resulting value (default=True)

      .. note:: this is the __init__ docstring
      """
      self.result = a + b
```

for documenting `__init__` method you typically pick a convention to use class or method docstring

Jupyter and Introspection

Jupyter

1. as an environment for documenting research, and
2. Jupyter as a tool for introspection

Jupyter as a documentation tool ...

Jupyter itself can be useful for documenting research projects as it provides modal type cells:

1. prose (with LaTeX math markup support)
2. programming
3. data, visualization and plotting

Useful feature of Jupyter ...

Jupyter (via IPython) has a number of useful query features for objects

```
import pandas as pd

df = pd.DataFrame([1,2,3,4,5])

df.<tab>                #Provides access to object methods
df.sort(<tab>)          #Provides method signature
df.sort(<tab><tab>)     #Provides full docstring
df.sort?                #Provides docstring in new window
df.sort??               #Provides docstring and full set of code for
```

Why use Documentation Standards?

Main: A lot of thought has gone into design and readability

Others:

1. Easier to work across projects in a community
2. Integrates with software that can build useful user-guides, notes, or manuals
3. use markup (via RST) while retaining readability
4. can be tailored to the needs of a specific community

Numpy/Scipy Style Docstrings

Focus: layout that can produce a well formatted reference guide

Uses a subset of re-structured text (RST) markup:

1. maintain readability in text editors
2. allows for more advanced formatting to be inferred from simple markup
3. allows the use of LaTeX for math
4. bibtex citations
5. use of sphinx directives to add warnings, notes etc.

A reStructured text primer can be found here

Numpy Style Docstrings: Functions

```
def function(param, keyword_param=True):
  """(#1) a short description of the function

  (#2) deprecation warnings

  (#3) extended summary

  (#4) Parameters
  ---------------
  param : type
          description of param 1
  keyword_param : boolean, optional(default=True)
                  description of keyword parameter

  (#5) Returns and Yields (explanation of returned values)
  Returns
  -------
  result

  (#6) Raises (optional section dealing with exceptions)

  (#7) See Also

  (#8) Notes and References

  (#9) Examples (can also be doctests)
  """
```

Numpy Style Docstring: Typical Example

```
def func(arg1, arg2=True):
    """Summary line.

    Extended description of function.

    Parameters
    ----------
    arg1 : int
        Description of arg1
    arg2 : bool, optional(default=True)
        Description of arg2

    Returns
    -------
    bool
        Description of return value
```

Numpy Style Docstrings: Classes

A) Class Docstring

Use same sections as function (except Returns as **not** applicable)

The `__init__` constructor should be documented in the class docstring

An **Attributes** section can be located below Parameters to describe non-method attributes

A **Methods** section can be located below Attributes to document public methods

B) Method Docstrings

Documented in similar fashion to functions, but always exclude `self` from the list of parameters

NumPy Style Docstring Guide and Example

NumPy Guide

https://github.com/numpy/numpy/blob/master/doc/HOWTO_DOCUMENT.rst.txt#sections

Extended explanatory example (Napoleon sphinx extension)
with modules, functions, and classes

http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_numpy.html#example-numpy

Google Style Docstrings: Typical Example

Main difference is use of indentation to separate sections rather than underlines

```
def func(arg1, arg2=True):
    """Summary line.

    Extended description of function.

    Args:
        arg1 (int): Description of arg1
        arg2 (bool, default=True): Description of arg2

    Returns:
        bool: Description of return value

    """
```

Google Style Docstrings

Extended explanatory example (Napoleon sphinx extension)

with module, functions and classes

http://sphinxcontrib-napoleon.readthedocs.io/en/latest/example_google.html

Comparison: Numpy vs Google Style

NumPy style tends to require more vertical space, whereas Google style tends to use more horizontal space.

Google style tends to be easier to read for short and simple docstrings, whereas NumPy style tends be easier to read for long and in-depth docstrings.

`http://www.sphinx-doc.org/en/stable/ext/napoleon.html#google-vs-numpy`

My tips for writing good documentation

1. don't **duplicate** clearly written python code in your comments and documentation
2. concise
3. consistent
4. include examples
5. use descriptive variable names in your code to reduce comments
6. make sure comments have purpose

Sphinx and Compilation Systems

A benefit of using a standard for documentation is nice integration with build systems such as Sphinx

You can document projects using autodoc which builds your documentation from internal docstrings

This is mainly useful for **larger** projects or code libraries

Example: QuantEcon Project Documentation

QuantEcon.py documentation can be found:

http://quanteconpy.readthedocs.io/en/latest/

Creating a Sphinx Project

sphinx-apidoc

To auto-generate sphinx rst files for a project

```
sphinx-apidoc -o docs .
```

sphinx-quickstart

To start a sphinx documentation project it is best to use

```
sphinx-quickstart
```

Sphinx getting started tutorial

Resources

1. Python - PEP8
2. Python - Docstring Conventions
3. NumpyDoc how-to Guide
4. Google Style Python Docstrings
5. Sphinx
    - sphinxcontrib.napoleon package
    - Numpydoc
    - autodoc
6. Jupyter

Julia

Julia Documentation

Current state of the art: Documenter.jl

Features:

1. supports markdown
2. support for LaTeX math
3. Doctests, cross-references for docs, linter etc.

documentation standards still to solidify

Julia Example

```
"""
add two numbers together

# Keyword arguments:
* `show:boolean`: show the resulting value (default=true)
"""
function add(a, b; show=true)
   result = a + b
   if show == true
     println("Result = ", result)
   end
   return result
 end
```