kwiver-doctest Documentation

Release 1.0

Keith Fieldhouse

December 29, 2015

Contents

Contents:

CHAPTER 1

Getting Started

We have adopted Sphinx as the documentation engine for the KWIVER project. Sphinx's focus on making *writing* documentation as easy as possible while still providing excellent support for *generating* documentation was espeially attractive. This project serves as an example Sphinx documented project and contains meta-documentation about how the documentation process for KWIVER projects works.

1.1 Environment

Sphinx is a Python based tool and requires a number of Python modules in addition to the Sphinx module itself. At the KWIVER project, we frequently use the Miniconda project from Continuum to provide out Python environment. This provides a cross-platorm (Windows, Linux and Mac OS X), consistent environment that's easy to install and maintain.

Miniconda provides it's own package manager, conda which can be used to install most of the packages required for Sphinx based documnetation. Conda also supports the creation of Python "sandboxes" or virtual environments. We typically keep a "Sphinx" environment available, which can be created this way:

```
conda create -n Sphinx sphinx sphinx_rtd_theme
```

Which will install the Sphinx tools (and all of Sphinx's dependencies) and the Sphinx ReadTheDocs theme (which is the current default KWIVER theme)

Once you've created the Sphinx environment you activate it this way:

```
source activate Sphinx
```

1.2 Quickstart

Sphinx provides a command that initializes a project with a Sphinx configuration file and stubs for some key documentation files called `sphinx-quickstart`. We create a `docs` directory within our KWIVER projects that contains these files. While you're at it, you may wish to create a `.gitignore` file containing `docs/_build` (at least) to avoid seeing the projects' documentation build artificats in your `git status` results.

When you run *sphinx-quickstart* in the `docs` directory it will ask a you a series of questions. In general you'll have to decide on the answers to may of these based on the needs of your project but there are some key settings that are useful:

- We use `.rst` as the source file suffix
- We turn on the EPub builder
- We turn on *autodoc*, *intersphinx* and *viewcode*

- We use `index.rst` as our anchor document

Once you've run `sphinx-quickstart`, you can edit `index.rst` to begin writing your documentation. We find the Sphinx reStructuredText Primer to be a useful introduction to the documentation format used by Sphinx.

For KWIVER projects, we typically edit the `conf.py` file to change `html_theme` to `sphinx_rtd_theme`.

1.3 Preview

Since reStructuredText is a mark up syntax that you work with in a text editor, you will need some means to see what your rendered documentation will look like. While you can simply run `make html` in your `docs` directory and open the resulting `.html` file, this can become somewhat tedious. If you install the `livereload` module in your Sphinx environment (`pip install livereload` should do the trick) you can use the following Python script:

```
from livereload import Server, shell
server = Server()
server.watch("*.rst", shell('make html', cwd='.'))   #'*
server.serve(root='_build/html')
```

Save this in your `docs` directory as `sphinx_server.py` and run it with this command:

```
python sphinx_server.py
```

Then, you can browse to `http://localhost:5500/` to see your rendered documentation. The `livereload` module will notice whenever you save a new version of one of your `*.rst` files and will re-run sphinx to provide an updated view of you rendered documentation.

CHAPTER 2

Documenting Code

2.1 Python Code

Sphinx started as a Python documentation tool and as a result, has strong capabilities in this area. In particular, it is capable of extracting Python "docstrings" and inserting them into your overall documentation collection as you dictate.

In the KWIVER project we use docstrings to document individual module, classes, members and functions. To include this text in our documentation, we need to make sure that Sphinx can "import" our modules without side effects. Primarily this means that there should be no executable code (beyond function, class and variable definitions) in your modules. If you want to make your module executable on the command line for convenience or testing purposes, use the following construct to guard that code:

```
if __name__ == "__main__":
    # executable code when your module is called directly on the command line goes here
```

You'll also need to make sure that Sphinx can find your modules by making sure their locations are on the Python path. You can do this by editing `conf.py` in your `docs` directory. Since Sphinx's configuration file is an actual Python file, you can use `sys.path` to adjust the Python path. Typically for KWIVER projects we keep python code in the the `python` directory and python based commands in the `bin` directory, both of which are peers of the `docs` directory. Given this, we can add the following lines to top of our `conf.py` file:

```
import sys
import os

sys.path.insert(0,"../python")
sys.path.insert(0,"../bin")
```

Sphinx runs with the `docs` directory as its current working directory, so these relative paths work.

2.2 Module Documentation Example

To include a module's documentation you use Sphinx's `automodule` command like this:

```
.. automodule:: kwiver_doctest
   :members:
```

What follows is documentation found in the `kwiver_doctest.py` module included with this repository.

2.2.1 `kwiver_doctest` Module

The module level documentation can contain reStructuredText in it just like the `.rst` files that make up a documenation collection.

`kwiver_doctest.`**`sample_function`**`(`*foo*`)`

This is sample function documentation

Parameters **`foo`** (*string*) – A sample parameter

Returns True on success, or False on failure

Return type bool

Raises AttributeError, KeyError

Function documentation has special tags. Click on the "source" link associated with this function to see how this function was documented. See the Sphinx info field documentation for further details:

2.3 Command Documentation Example

For the KWIVER project, we use the `argparse` module to parse our command line arguments. Among other things, this allows us to use the sphinx-argparse extension which will automatically document commands based on the help text included when the parser is built. In order to use it you'd invoke it like this:

```
.. argparse::
   :ref: kwiver-doctest-command.cli_parser
   :prog: kwiver-doctest-command
```

Which results in output like this:

Compute compute something useful based on input and argurments.

```
usage: kwiver-doctest-command [-h] [-o OUTPUT_FILEPATH] [-v] [-c CONFIG]
                              input_file
```

Positional arguments:

input_file	Input Data file

Options:

-o, --output-filepath	Path to a file to output feature vector to. Otherwise the feature vector is printed to standard out. Output is saved in numpy binary format (.npy suffix recommended).
-v=False, --verbose=False	Print additional debugging messages. All logging goes to standard error.
-c, --config	Configuration file

In order for this to work, you command mus tbe on the Python path that you set up in `conf.py` and there must be a symbol (either a function call or a variable) at the root level of the module that Sphinx can use to access the `argparse` object so that it can introspect the help text. If you use a function (like we have here with `cli_parser()`) make sure that the function *only* creates the `argparse` object becuase it will be exectued within the Sphinx process when the documentation is generated.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index