
**The RestructuredText Book
Documentation**
Release 1.0

Daniel Greenfeld, Eric Holscher

Sep 05, 2018

Tutorial

1	Schedule	2
1.1	Getting Started: Overview & Installing Initial Project	2
1.2	Step 1: Getting started with RST	8
1.3	Step 2: Building References & API docs	13
1.4	Step 3: Keeping Documentation Up to Date	23
1.5	Finishing Up: Additional Extensions & Individual Exploration	27
1.6	Cheat Sheet	31
1.7	Crawler Step 1 Documentation	34
1.8	Crawler Step 2 Documentation	34
1.9	Crawler Step 3 Documentation	36
	Python Module Index	40

Welcome to the Introduction to Sphinx & Read the Docs. This tutorial will walk you through the initial steps writing reStructuredText and Sphinx, and deploying that code to Read the Docs.

Please provide feedback to [@ericholscher](#).

Schedule

- 9-9:30 Introduction to the Tutorial
- 9:30-9:50 *Getting Started: Overview & Installing Initial Project*
- 9:50-10:20 *Step 1: Getting started with RST*
- 10:20-10:40 *Break*
- 10:40-11:10 *Step 2: Building References & API docs*
- 11:10-11:40 *Step 3: Keeping Documentation Up to Date*
- 11:40-12:20 *Finishing Up: Additional Extensions & Individual Exploration*

Thanks for coming

1.1 Getting Started: Overview & Installing Initial Project

1.1.1 Concepts

Sphinx Philosophy

Sphinx is what is called a documentation generator. This means that it takes a bunch of source files in plain text, and generates a bunch of other awesome things, mainly HTML. For our use case you can think of it as a program that takes in plain text files in **reStructuredText** format, and outputs HTML.

```
reST -> Sphinx -> HTML
```

So as a user of Sphinx, your main job will be writing these text files. This means that you should be minimally familiar with `reStructuredText` as a language. It's similar to Markdown in a lot of ways, if you are already familiar with Markdown.

1.1.2 Tasks

Installing Sphinx

The first step is installing `Sphinx`. Sphinx is a python project, so it can be installed like any other python library. Every Operating System should have Python pre-installed, so you should just have to run:

```
pip install sphinx
```

Note: Advanced users can install this in a virtualenv if they wish. Also, `easy_install install sphinx` works fine if you don't have pip.

Get this repo

To do this tutorial, you need the actual repository. It contains the example code that we will be documenting.

You can clone it here:

```
git clone https://github.com/ericholscher/pycon-sphinx-tutorial
```

Getting Started

Now you are ready to start creating documentation. You should have a directory called `crawler`, which contains source code in its `src` directory. Inside the `crawler` you should create a `docs` directory, and move into it:

```
cd crawler
mkdir docs
cd docs
```

Then you can create the Sphinx project skeleton in this directory:

```
sphinx-quickstart
```

Have the *Project name* be `Crawler`, put in your own *Author name*, and put in `1.0` as the *Project version*. Otherwise you can accept the default options.

My output looks like this:

```
1 -> sphinx-quickstart
2 Welcome to the Sphinx 1.3.1 quickstart utility.
3
4 Please enter values for the following settings (just press Enter to
5 accept a default value, if one is given in brackets).
6
7 Enter the root path for documentation.
8 > Root path for the documentation [.]:
```

You have two options for placing the build directory for Sphinx output. Either, you use a directory `"_build"` within the root path, or you `↪separate`

```
12 "source" and "build" directories within the root path.
13 > Separate source and build directories (y/n) [n]:
14
15 Inside the root directory, two more directories will be created; "_
16 ↪templates"
17 for custom HTML templates and "_static" for custom stylesheets and _
18 ↪other static
19 files. You can enter another prefix (such as ".") to replace the _
20 ↪underscore.
```

```
18 > Name prefix for templates and static dir [_]:
19
20 The project name will occur in several places in the built _
21 ↪documentation.
```

```
21 > Project name: Crawler
22 > Author name(s): Eric Holscher
23
24 Sphinx has the notion of a "version" and a "release" for the
25 software. Each version can have multiple releases. For example, for
26 Python the version is something like 2.5 or 3.0, while the release is
27 something like 2.5.1 or 3.0a1. If you don't need this dual structure,
28 just set both to the same value.
29 > Project version: 1.0
30 > Project release [1.0]:
31
32 If the documents are to be written in a language other than English,
33 you can select a language here by its language code. Sphinx will then
34 translate text that it generates into that language.
35
36 For a list of supported codes, see
37 http://sphinx-doc.org/config.html#confval-language.
```

(continues on next page)

(continued from previous page)

```
38 > Project language [en]:
39
40 The file name suffix for source files. Commonly, this is either ".txt"
41 or ".rst". Only files with this suffix are considered documents.
42 > Source file suffix [.rst]:
43
44 One document is special in that it is considered the top node of the
45 "contents tree", that is, it is the root of the hierarchical structure
46 of the documents. Normally, this is "index", but if your "index"
47 document is a custom template, you can also set this to another
48   ↪filename.
49
50 > Name of your master document (without suffix) [index]:
51
52 Sphinx can also add configuration for epub output:
53 > Do you want to use the epub builder (y/n) [n]:
54
55 Please indicate if you want to use one of the following Sphinx
56   ↪extensions:
57 > autodoc: automatically insert docstrings from modules (y/n) [n]:
58 > doctest: automatically test code snippets in doctest blocks (y/n)
59   ↪[n]:
60 > intersphinx: link between Sphinx documentation of different projects
61   ↪(y/n) [n]:
62 > todo: write "todo" entries that can be shown or hidden on build (y/
63   ↪n) [n]:
64 > coverage: checks for documentation coverage (y/n) [n]:
65 > pngmath: include math, rendered as PNG images (y/n) [n]:
66 > mathjax: include math, rendered in the browser by MathJax (y/n) [n]:
67 > ifconfig: conditional inclusion of content based on config values (y/
68   ↪n) [n]:
69 > viewcode: include links to the source code of documented Python
70   ↪objects (y/n) [n]:
71
72 A Makefile and a Windows command file can be generated for you so that
73   ↪you
74 only have to run e.g. `make html' instead of invoking sphinx-build
75 directly.
76 > Create Makefile? (y/n) [y]:
77 > Create Windows command file? (y/n) [y]:
78
79 Creating file ./conf.py.
80 Creating file ./index.rst.
81 Creating file ./Makefile.
82 Creating file ./make.bat.
```

(continues on next page)

(continued from previous page)

```
75 Finished: An initial directory structure has been created.
76
77 You should now populate your master file ./index.rst and create other_
  ↳documentation
78 source files. Use the Makefile to build the docs, like so:
79     make builder
80 where "builder" is one of the supported builders, e.g. html, latex or_
  ↳linkcheck.
```

Your file system should now look similar to this:

```
crawler
├── src
├── docs
│   ├── index.rst
│   ├── conf.py
│   ├── Makefile
│   ├── make.bat
│   ├── _build
│   ├── _static
│   └── _templates
```

We have a top-level `docs` directory in the main project directory. Inside of this is:

index.rst: This is the index file for the documentation, or what lives at `/`. It normally contains a *Table of Contents* that will link to all other pages of the documentation.

conf.py: Allows for customization of Sphinx. You won't need to use this too much yet, but it's good to be familiar with this file.

Makefile & make.bat: This is the main interface for local development, and shouldn't be changed.

_build: The directory that your output files go into.

_static: The directory to include all your static files, like images.

_templates: Allows you to override Sphinx templates to customize look and feel.

Building docs

Let's build our docs into HTML to see how it works. Simply run:

```
# Inside top-level docs/ directory.
make html
```

This should run Sphinx in your shell, and output HTML. At the end, it should say something about the documents being ready in `_build/html`. You can now open them in your browser by typing:


```
# On OS X
open _build/html/index.html
```

You can also view it by running a web server in that directory:

```
# Inside docs/_build/html directory.
python -m SimpleHTTPServer

# For python 3
python3 -m http.server
```

Then open your browser to <http://localhost:8000>.

This should display a rendered HTML page that says **Welcome to Crawler’s documentation!** at the top.

Note: `make html` is the main way you will build HTML documentation locally. It is simply a wrapper around a more complex call to Sphinx, which you can see as the first line of output.

Custom Theme

You’ll notice your docs look a bit different than mine.

First, you need to install the theme:

```
$ pip install sphinx_rtd_theme
```

Then you need to update a few settings in your `conf.py`.

```
import sphinx_rtd_theme

html_theme = 'sphinx_rtd_theme'

html_theme_path = [sphinx_rtd_theme.get_html_theme_path()]
```

If you rebuild your documentation, you will see the new theme:

```
make html
```

Warning: Didn’t see your new theme? That’s because Sphinx is smart, and only rebuilds pages that have changed. It might have thought none of your pages changed, so it didn’t rebuild anything. Fix this by running a `make clean html`, which will force a full rebuild.

1.1.3 Extra Credit

Have some extra time left? Check out these other cool things you can do with Sphinx.

Understanding `conf.py`

Sphinx is quite configurable, which can be a bit overwhelming. However, the `conf.py` file is quite well documented. You can read through it and get some ideas about what all it can do.

A few of the more useful settings are:

- `project`
- `html_theme`
- `extensions`
- `exclude_patterns`

This is all well documented in the Sphinx [Configuration doc](#).

Moving on

Now it is time to move on to *Step 1: Getting started with RST*.

1.2 Step 1: Getting started with RST

Now that we have our basic skeleton, let's document the project. As you might have guessed from the name, we'll be documenting a basic web crawler.

For this project, we'll have the following pages:

- Index Page
- Support
- Installation
- Cookbook
- Command Line Options
- API

Let's go over the concepts we'll cover, and then we can talk more about the pages to create.

1.2.1 Concepts

A lot of these RST syntax examples are covered in the Sphinx [reStructuredText Primer](#).

Sections

```
Title
=====

Section
-----

Subsection
~::~::~::~
```

Every Sphinx document has multiple level of headings. Section headers are created by underlining the section title with a punctuation character, at least as long as the text.

They give structure to the document, which is used in navigation and in the display in all output formats.

Code Samples

```
You can use ``backticks`` for showing ``highlighted`` code.
```

If you want to make sure that text is shown in monospaced fonts for code examples or concepts, use double backticks around it. It looks like `this` on output.

Hyperlink Syntax

```
`A cool website`_
.. A cool website: http://sphinx-doc.org
```

The link text is set by putting a `_` after some text. The ``` is used to group text, allowing you to include multiple words in your link text. You should use the ```, even when the link text is only one word. This keeps the syntax consistent.

The link target is defined at the bottom of the section with `.. <link text>: <target>`.

Code Example Syntax

```
A cool bit of code::  
  
    Some cool Code  
  
.. code-block:: rst  
  
    A bit of **rst** which should be *highlighted* properly.
```

The syntax for displaying code is `::`. When it is used at the end of a sentence, Sphinx is smart and displays one `:` in the output, and knows there is a code example in the following indented block.

Sphinx, like Python, uses meaningful whitespace. Blocks of content are structured based on the indentation level they are on. You can see this concept with our `code-block` directive later.

Table of Contents Tree

```
.. toctree::  
   :maxdepth: 2  
  
   install  
   support
```

Now would be a good time to introduce the `toctree`. One of the main concepts in Sphinx is that it allows multiple pages to be combined into a cohesive hierarchy. The `toctree` directive is a fundamental part of this structure.

The above example will output a Table of Contents in the page where it occurs. The `maxdepth` argument tells Sphinx to include 2 levels of headers in its output. It will output the 2 top-level headers of the pages listed. This also tells Sphinx that the other pages are sub-pages of the current page, creating a “tree” structure of the pages:

```
index  
├── install  
└── support
```

Note: The TOC Tree is also used for generating the navigation elements inside Sphinx. It is quite important, and one of the most powerful concepts in Sphinx.

1.2.2 Tasks

Create Installation page

Installation documentation is really important. Anyone who is coming to the project will need to install it. For our example, we are installing a basic Python script, so it will be pretty easy.

Include the following in your `install.rst`, on the same level as `index.rst`, properly marked up:

```
1 Installation
2
3 At the command line:
4
5 easy_install crawler
6
7 Or, if you have pip installed:
8
9 pip install crawler
```

Note: Live Preview: *Installation*

Create Support page

It's always important that users can ask questions when they get stuck. There are many ways to handle this, but normal approaches are to have an IRC channel and mailing list.

Go ahead and put this in your `support.rst`, but add the proper RST markup:

```
1 Support
2
3 The easiest way to get help with the project is to join the #crawler
4 channel on Freenode.
5 We hang out there and you can get real-time help with your projects.
6 The other good way is to open an issue on Github.
7
8 The mailing list at https://groups.google.com/forum/#!forum/crawler
9 is also available for support.
10
11 Freenode: irc://freenode.net
12 Github: http://github.com/example/crawler/issues
13
```

Note: Live Preview: *Support*

You can now open the `support.html` file directly, but it isn't showing on the navigation..

Add TocTree

Now you need to tie all these files together. As we mentioned above, the *Table of Contents Tree* is the best way to do this. Go ahead and complete the `toctree` directive in your `index.rst` file, adding the new `install` and `support`.

Sanity Check

Your filesystem should now look something like this:

```
crawler
├── src
├── docs
│   ├── index.rst
│   ├── support.rst
│   ├── install.rst
│   ├── Makefile
│   └── conf.py
```

Build Docs

Now that you have a few pages of content, go ahead and build your docs again:

```
make html
```

If you open up your `index.html`, you should see the basic structure of your docs from the included `toctree` directive.

1.2.3 Extra Credit

Have some extra time left? Check out these other cool things you can do with Sphinx.

Make a manpage

The beauty of Sphinx is that it can output in multiple formats, not just HTML. All of those formats share the same base format though, so you only have to change things in one place. So you can

generate a manpage for your docs:

```
make man
```

This will place a manpage in `_build/man`. You can then view it with:

```
man _build/man/crawler.1
```

Create a single page document

Some people prefer one large HTML document, instead of having to look through multiple pages. This is another area where Sphinx shines. You can write your documentation in multiple files to make editing and updating easier. Then if you want to distribute a single page HTML version:

```
make singlehtml
```

This will combine all of your HTML pages into a single page. Check it out by opening it in your browser:

```
open _build/singlehtml/index.html
```

Note: You'll notice that it included the documents in the order that your *TOC Tree* was defined.

Play with RST

RST takes a bit of practice to wrap your head around. Go over to <http://rst.ninjs.org>, which is a live preview.

Note: Use the *Cheat Sheet* for lots more ideas!

Looking for some ideas of what the syntax contains? The [reStructuredText Primer](#) in the Sphinx docs is a great place to start.

Moving on

Now it is time to move on to *Step 2: Building References & API docs*.

1.3 Step 2: Building References & API docs

Note: Finish at 11:15

1.3.1 Concepts

Referencing

Another important Sphinx feature is that it allows referencing across documents. This is another powerful way to tie documents together.

The simplest way to do this is to define an explicit reference object:

```
.. _reference-name:
```

```
Cool section
```

```
-----
```

Which can then be referenced with `:ref::`

```
:ref:`reference-name`
```

Which will then be rendered with the title of the section *Cool section*.

Sphinx also supports `:doc:`docname`` for linking to a document.

Semantic Descriptions and References

Sphinx also has much more powerful semantic referencing capabilities, which knows all about software development concepts.

Say you're creating a CLI application. You can define an option for that program quite easily:

```
.. option:: -i <regex>, --ignore <regex>
```

```
    Ignore pages that match a specific pattern.
```

That can also be referenced quite simply:

```
:option:`-i`
```

Sphinx includes a large number of these semantic types, including:

- `Module`
- `Class`
- `Method`

External References

Sphinx also includes a number of pre-defined references for external concepts. Things like PEP's and RFC's:

```
You can learn more about this at :pep:`8` or :rfc:`1984`.
```

You can read more about this in the Sphinx inline-markup docs.

Automatically generating this markup

Of course, Sphinx wants to make your life easy. It includes ways to automatically create these object definitions for your own code. This is called `autodoc`, which allows you do to syntax like this:

```
.. automodule:: crawler
```

and have it document the full Python module importable as `crawler`. You can also do a full range of auto functions:

```
.. autoclass::  
.. autofunction::  
.. autoexception::
```

Warning: The module must be importable by Sphinx when running. We'll cover how to do this in the Tasks below.

You can read more about this in the Sphinx `autodoc` docs.

1.3.2 Tasks

Referencing Code

Let's go ahead and add a cookbook to our documentation. Users will often come to your project to solve the same problems. Including a Cookbook or Examples section will be a great resource for this content.

In your `cookbook.rst`, add the following:

```
1 Cookbook  
2  
3 Crawl a web page
```

(continues on next page)

(continued from previous page)

```
4
5 The most simple way to use our program is with no arguments.
6 Simply run:
7
8 python main.py -u <url>
9
10 to crawl a webpage.
11
12 Crawl a page slowly
13
14 To add a delay to your crawler,
15 use -d:
16
17 python main.py -d 10 -u <url>
18
19 This will wait 10 seconds between page fetches.
20
21 Crawl only your blog
22
23 You will want to use the -i flag,
24 which will ignore URLs matching the passed regex::
25
26 python main.py -i "^blog" -u <url>
27
28 This will only crawl pages that contain your blog URL.
```

Note: Live Preview: [Cookbook](#)

Remember, you will need to use `:option:` blocks here. This is because they are referencing a command line option for our program.

Adding Reference Targets

Now that we have pointed at our CLI options, we need to actually define them. In your `cli.rst` file, add the following:

```
1 Command Line Options
2
3 These flags allow you to change the behavior of Crawler.
4 Check out how to use them in the Cookbook.
5
6 -d <sec>, --delay <sec>
7
```

(continues on next page)

(continued from previous page)

```
8 Use a delay in between page fetchs so we don't overwhelm the remote_
  ↪server.
9 Value in seconds.
10
11 Default: 1 second
12
13 -i <regex>, --ignore <regex>
14
15 Ignore pages that match a specific pattern.
16
17 Default: None
```

Note: Live Preview: *Command Line Options*

Here you are documenting the actual options your code takes.

Try it out

Let's go ahead and build the docs and see what happens. Do a:

```
make html
```

Here you will see that the `:option:` blocks magically become links to the definition. This is your first taste of Semantic Markup. With Sphinx, we are able to simply say that something is a `option`, and then it handles everything for us; linking between the definition and the usage.

Importing Code

Being able to define options and link to them is pretty neat. Wouldn't it be great if we could do that with actual code too? Sphinx makes this easy, let's take a look.

We'll go ahead and create an `api.rst` that will hold our API reference:

```
1 Crawler Python API
2
3 Getting started with Crawler is easy.
4 The main class you need to care about is crawler.main.Crawler
5
6 crawler.main
7
8 automodule: crawler.main
```

Note: Live Preview: *Crawler Python API*

Remember, you'll need to use the `.. autoclass::` directive to pull in your source code. This will render the docstrings of your Python code nicely.

Requirements

In order to build your code, it needs to be able to import it. This means it needs all of the required Python modules you import in the code.

If you have third party dependencies, that means that you have to have them installed in your Python environment. Luckily, for most cases you can actually mock these variables using `autodoc_mock_imports`.

In your `conf.py` go ahead and add:

```
autodoc_mock_imports = ['bs4', 'requests']
```

This will allow your docs to import the example code without requiring those modules be installed.

Tell Sphinx about your code

When Sphinx runs `autodoc`, it imports your Python code to pull off the docstrings. This means that Sphinx has to be able to see your code. We'll need to add our `PYTHONPATH` to our `conf.py` so it can import the code.

If you open up your `conf.py` file, you should see something close to this on line 18:

```
# If extensions (or modules to document with autodoc) are in another_
→directory,
# add these directories to sys.path here. If the directory is relative_
→to the
# documentation root, use os.path.abspath to make it absolute, like_
→shown here.
#sys.path.insert(0, os.path.abspath('.'))
```

As it notes, you need to let it know the path to your Python source. In our example it will be `../src/`, so go ahead and put that in this setting.

Note: You should always use relative paths here. Part of the value of Sphinx is having your docs build on other people's computers, and if you hard code local paths that won't work!

Try it out

Now go ahead and regenerate your docs and look at the magic that happened:

```
make html
```

Your Python docstrings have been magically imported into the project.

Tie it all together

Now let's link directly to that for users who come in to the project. Update your `index.rst` to look like:

```
1 Crawler Step 2 Documentation
2
3 User Guide
4
5 toctree:
6
7 install
8 support
9 cookbook
10
11 Programmer Reference
12
13 toctree:
14
15 cli
16 api
```

Note: Live Preview: [Crawler Step 2 Documentation](#)

One last time, let's rebuild those docs:

```
make html
```

Warning: You now have awesome documentation! :)

Now you have a beautiful documentation reference that is coming directly from your code. This means that every time you change your code, it will automatically be reflected in your documentation.

The beauty of this approach is that it allows you to keep your prose and reference documentation in the same place. It even lets you semantically reference the code from inside the docs. This is amazingly powerful and a great way to write documentation.

1.3.3 Extra Credit

Have some extra time left? Let's look through the code to understand what's happening here more.

Look through intersphinx

Intersphinx allows you to bring the power of Sphinx references to multiple projects. It lets you pull in references, and semantically link them across projects. For example, in this guide we reference the Sphinx docs a lot, so we have this intersphinx setting:

```
intersphinx_mapping = {
    'sphinx': ('http://sphinx-doc.org/', None),
}
```

Which allows us to add a prefix to references and have them resolve:

```
:ref:`sphinx:inline-markup`
```

We can also ignore the prefix, and Sphinx will fall back to intersphinx references if none exist in the current project:

```
:ref:`inline-markup`
```

You can read more about this in the [intersphinx docs](#).

Understand the code

A lot of the magic that is happening in *Importing Code* above is actually in the source code.

Check out the code for `crawler/main.py`:

```
1 """
2 Main Module
3 """
4 import time
5 from optparse import OptionParser
6 # Python 3 compat
7 try:
8     from urlparse import urlparse
9 except ImportError:
```

(continues on next page)

(continued from previous page)

```
10     from urllib.parse import urlparse
11
12 import requests
13 from bs4 import BeautifulSoup
14
15 from utils import log, should_ignore
16
17
18 class Crawler(object):
19
20     """
21     Main Crawler object.
22
23     Example::
24
25         c = Crawler('http://example.com')
26         c.crawl()
27
28     :param delay: Number of seconds to wait between searches
29     :param ignore: Paths to ignore
30
31     """
32
33     def __init__(self, url, delay, ignore):
34         self.url = url
35         self.delay = delay
36         if ignore:
37             self.ignore = ignore.split(',')
38         else:
39             self.ignore = []
40
41     def get(self, url):
42         """
43         Get a specific URL, log its response, and return its content.
44
45         :param url: The fully qualified URL to retrieve
46         """
47         response = requests.get(url)
48         log(url, response.status_code)
49         return response.content
50
51     def crawl(self):
52         """
53         Crawl the URL set up in the crawler.
54
```

(continues on next page)

(continued from previous page)

```

55     This is the main entry point, and will block while it runs.
56     """
57     html = self.get(self.url)
58     soup = BeautifulSoup(html, "html.parser")
59     for tag in soup.findAll('a', href=True):
60         link = tag['href']
61         parsed = urlparse(link)
62         if parsed.scheme:
63             to_get = link
64         else:
65             to_get = self.url + link
66         if should_ignore(self.ignore, to_get):
67             print('Ignoring URL: {url}'.format(url=to_get))
68             continue
69         self.get(to_get)
70         time.sleep(self.delay)
71
72
73 def run_main():
74     """
75     A small wrapper that is used for running as a CLI Script.
76     """
77
78     parser = OptionParser()
79     parser.add_option("-u", "--url", dest="url", default="http://docs.
80     ↪readthedocs.org/en/latest/",
81                       help="URL to fetch")
82     parser.add_option("-d", "--delay", dest="delay", type="int",
83     ↪default=1,
84                       help="Delay between fetching")
85     parser.add_option("-i", "--ignore", dest="ignore", default='',
86                       help="Ignore a subset of URL's")
87
88     (options, args) = parser.parse_args()
89
90     c = Crawler(url=options.url, delay=options.delay, ignore=options.
91     ↪ignore)
92     c.crawl()
93
94 if __name__ == '__main__':
95     run_main()

```

As you can see, we're heavily using RST in our docstrings. This gives us the same power as we have in Sphinx, but allows it to live within the code base.

This approach of having the docs live inside the code is great for some things. However, the power

of Sphinx allows you to mix docstrings and prose documentation together. This lets you keep the amount of

Moving on

Could it get better? In fact, it can and it will. Let's go on to *Step 3: Keeping Documentation Up to Date*.

1.4 Step 3: Keeping Documentation Up to Date

Now we have a wonderful set of documentation, so we want to make sure it stays up to date and correct.

There are two factors here:

- The documentation is up to date with the code
- The user is seeing the latest version of the docs

We will solve the first problem with Sphinx's `doctest` module. The second problem we will solve by deploying our docs to [Read the Docs](#).

1.4.1 Concepts

Testing your code

Sphinx ships with a `doctest` module which is quite powerful. It allows you to run tests against your code inside your docs. This means that you can verify all of the code examples work, so that your docs are always up to date with your code!

Warning: This only works for Python currently.

You can read the full Sphinx docs for `doctest`, but here is a basic example:

```
.. doctest::  
  
    >>> sum(2, 2)  
    4
```

When you run this example, Sphinx will validate the return is what is expected.

If you need any other code to be run, but not output to the user, you can use `testsetup`:

```
.. testsetup::

    import os

    x = 4
```

This will then be available in the examples that you actually show your user.

Hosting docs on Read the Docs

Read the Docs (<https://readthedocs.org>) is an open source doc hosting site. It's built in Django, and is free to use for open source projects. It hosts Sphinx documentation, automatically building it each time you make a commit.

Read the Docs gives you a number of additional features, over hosting Sphinx yourself:

- You can add Versions to your project for each tag & branch.
- You can alerts for when your doc build fails
- You can search across the full set of docs with Elastic Search

We'll be putting your docs up on Read the Docs at the end of this tutorial.

1.4.2 Tasks

Add doctests to our utils

The `utils` module inside `crawler` is a good candidate for testing. It has small, self-contained pieces of logic that will work great as doctests.

Open your `api.rst`, and update it to look like:

```
1 Crawler Python API
2
3 Getting started with Crawler is easy.
4 The main class you need to care about is crawler.main.Crawler
5
6 crawler.main
7
8 crawler.utils
9
10 crawler.utils.should_ignore
11
12 should_ignore(['blog/$'], 'http://ericholscher.com/blog/')
13 True
```

(continues on next page)

(continued from previous page)

```
14
15 # This test should fail
16 should_ignore(['home'], 'http://ericholscher.com/blog/')
17 True
18
19 crawler.utils.log
20
21 log('http://ericholscher.com/blog/', 200)
22 OK: 200 http://ericholscher.com/blog/
23
24 log('http://ericholscher.com/blog/', 500)
25 ERR: 500 http://ericholscher.com/blog/
26
27 # This test should fail
28 log('http://ericholscher.com/blog/', 500)
29 OK: 500 http://ericholscher.com/blog/
```

Note: Live Preview: [Crawler Python API](#)

Now go ahead and add the RST markup that is covered above in the *Concepts* section.

As you can see here, we are actually testing our logic. It also acts as documentation for your users, and is included in the output of your documentation.

These doctests do double duty, acting as **tests and documentation**.

Caveats

Note that we have to import our code in the `testsetup::` block. This is so that Sphinx can call the functions properly in our doctest blocks. This is hidden in the output of the docs though, so users won't be confused.

Note: You can also put doctest blocks directly in your docstrings. They will need to include full import paths though, as Sphinx can't guarantee the `testsetup::` directive will be called.

Test your docs

You can now go ahead and test your docs:

```
make doctest
```

Note: You will need to make sure to add the `sphinx.ext.doctest` to your extensions. Open up your `conf.py` file and make sure that you have it there.

It should provide output that looks similar to this:

```
Doctest summary
=====
 5 tests
 2 failures in tests
 0 failures in setup code
 0 failures in cleanup code
build finished with problems.
```

As you can see, some of the tests are broken! You should go ahead and fix the tests :)

Requirements

In order for Read the Docs to build your code, it needs to be able to import it. This means it needs all of the required Python modules you import in the code.

You can add a `requirements.txt` to the top-level of your project:

```
beautifulsoup4
requests
```

Read the Docs

Last but not least, once you've written your documentation you have to put it somewhere for the world to see! Read the Docs makes this quite simple, and is free for all open source projects.

- Register for an account at <http://readthedocs.org>
- Click the *Import Project* button
- Add the URL for a specific repository you want to build docs for
- Sit back and have a drink while Read the Docs does the rest.

It will:

- Pull down your code
- Install your `requirements.txt`
- Build HTML, PDF, and ePub of your docs
- Serve it up online at `http://<projectname>.readthedocs.org`

1.4.3 Extra Credit

Have some extra time left? Let's run the code and see if it actually works!

Explore doctests more

Sphinx's `doctest` module has more interesting options. You can do things that look more like normal unit tests, as well as specific “doctest-style” testing. Go in and re-write one of the existing tests to use the `testcode` directive instead of the `doctest` directive.

Run the crawler

Go ahead and run the crawler against the Read the Docs documentation:

```
# in crawler/src/crawler
python main.py -u https://docs.readthedocs.org/en/latest/
```

You should see your terminal start printing output, if your internet is working.

Can you add another command line option, and document it?

Moving on

Now we are at the last part of our Tutorial. Let's head on over to *Finishing Up: Additional Extensions & Individual Exploration*.

1.5 Finishing Up: Additional Extensions & Individual Exploration

If there is much time left in the session, take some time to play around and get to know Sphinx better. There is a large ecosystem of extensions, and lots of builtin features we haven't covered.

I'm happy to consult with you about interesting challenges you might be facing with docs.

Part of being a good user of Sphinx is knowing what all is there. Here are a few options for what to look at:

- [Developing extensions for Sphinx](#)
- [Read through all the existing extensions](#)
- [Breathe](#)
- [Explore the Read the Docs Admin Panel](#)

- Apply these docs to a project you have
- Show a neighbor what you've done & talk about the concepts learned.

Also, here are a number of more thought out examples of things you might do:

- *Markdown Support*
- *Generate i18n Files*
- *Play with Sphinx autoapi*
- *Add Django Support*
- *Tables*

1.5.1 Markdown Support

You can use Markdown and reStructuredText in the same Sphinx project. We support this natively on Read the Docs, and you can do it locally:

```
$ pip install recommonmark
```

Then in your `conf.py`:

```
from recommonmark.parser import CommonMarkParser

source_parsers = {
    '.md': CommonMarkParser,
}

source_suffix = ['.rst', '.md']
```

Note: Markdown doesn't support a lot of the features of Sphinx, like inline markup and directives. However, it works for basic prose content.

You can now add a Markdown file with a `.md` extension, and Sphinx will build it into the project. You can do things like include it in your normal TOC Tree, and Sphinx will search it.

Go ahead and add a new Markdown File with an `.md` extension. Since we haven't covered Markdown in this text, here is an example `community.md`:

```
# Community Standards

The Crawler community is quite large,
```

(continues on next page)

(continued from previous page)

```
and with that we have a specific set of standards that we apply in our
↳community.

All of our project spaces are covered by the [Django Community Code of
↳Conduct] (https://djangoproject.com/conduct/).

### Feedback

Any issues can be sent directly to our [project mailing
↳list] (mailto:community@crawler.com).
```

Add it to your `toctree` in your `index.rst` as well, and you will see it appear properly in Sphinx.

1.5.2 Generate i18n Files

Sphinx has support for i18n. If you do a `make gettext` on your project, you should get a `gettext` catalog for your documentation. Check for it in `_build/locale`.

You can then use these files to translate your documentation using most standard tools. You can read more about this in Sphinx's [Internationalization](#) doc.

1.5.3 Play with Sphinx autoapi

`sphinx-autoapi` is a tool that I am helping develop which will make doing API docs easier. It depends on parsing, instead of importing code. This means you don't need to change your `PYTHONPATH` at all, and we have a few other different design decisions.

First you need to install `autoapi`:

```
pip install sphinx-autoapi
```

Then add it to your Sphinx project's `conf.py`:

```
extensions = ['autoapi.extension']

# Document Python Code
autoapi_type = 'python'
autoapi_dir = '../src'
```

AutoAPI will automatically add itself to the last `TOCTree` in your top-level `index.rst`.

This is needed because we will be outputting `rst` files into the `autoapi` directory. This adds it into the global `TOCTree` for your project, so that it appears in the menus.

1.5.4 Add Django Support

Have a Django project laying around? Add Sphinx documentation to it! There isn't anything special for Django projects except for the `DJANGO_SETTINGS_MODULE`.

You can set it in your `conf.py`, similar to `autodoc`. Try this piece of code:

```
# Set this to whatever your settings file should default to.
os.environ.setdefault("DJANGO_SETTINGS_MODULE", "settings.test")
```

1.5.5 Tables

Tables can be a tricky part of a lot of lightweight markup languages. Luckily, RST has some really nice features around tables. It supports tables in a couple easier to use formats:

- [CSV](#)
- [List](#)

So for example, you can manage your tables in Google Docs, then export them as CSV in your docs.

An example of a CSV table:

```
.. csv-table::
   :header: "Treat", "Quantity", "Description"
   :widths: 15, 10, 30

   "Albatross", 2.99, "On a stick!"
   "Crunchy Frog", 1.49, "If we took the bones out, it wouldn't be
   crunchy, now would it?"
   "Gannet Ripple", 1.99, "On a stick!"
```

And a rendered example:

Treat	Quantity	Description
Albatross	2.99	On a stick!
Crunchy Frog	1.49	If we took the bones out, it wouldn't be crunchy, now would it?
Gannet Ripple	1.99	On a stick!

Go ahead and try it yourself!

1.6 Cheat Sheet

We have made a cheat sheet for helping you remember the syntax for RST & Sphinx programs.

1.6.1 RST Cheat Sheet

reStructuredText & Sphinx Cheatsheet

Section Headings

```

=====
Level 1 Heading
=====

Level 2 Heading
-----
    
```

Level 1 Heading
~::~::~::~::~

Level 2 Heading
+++++

- Heading structure is determined only by occurrence order.
- Heading overline is optional.
- Under/overlines use the following characters:

Recommended: = - ' : . ' " ~ ^ _ * + #
! \$ % & () , / ; < > ? @ [\] { | }

Styles

```

*Bold text*
**Italic text**
``Inline literal/code``
:sup:`super`\ Script
:sub:`sub`\ Script
    
```

Bullet Lists

```

* Unordered item
* Unordered item

1. Nested ordered item
2. Nested ordered item

    a. Nested ordered item

* Unordered item
    
```

Definition Lists

```

First term
  Definition of first term,
  it can span multiple lines.

Second term
  Definition of second term.

  Indent additional paragraphs.
    
```

Literal Code Blocks

```

Here is a literal block::

    Its contents are indented.

::

    The :: marker is omitted here
    
```

The :: marker will insert a ":" in the output in the above example. To omit the ":", precede the :: marker with white space, or use the marker on a line of its own.

Targets and Links

```

Anchor target .. _anchorbyref:
              .. _Anchor link by text:
External target .. _External link name: http://example.com
Footnote target .. [1] A footnote
Citation target .. [cit1] A global citation

External links `External link <http://example.com>`_
               `External link name`_ or `Example <External link name>`_

Internal links `Anchor link by text`_ or `Anchor <Anchor link by text>`_
               `Anchor link by ref <anchorbyref>`_
               :ref:`anchorbyref`

Footnote Reference a footnote [1]_
Citation or a global citation [cit1]_

Section link Section Heading
             -----
             `Link <Section Heading>`_
    
```

Tables

```

=====  =====  =====
Simple table Header 2 Header 3
=====  =====  =====
Column 1   Column 2 Column 3
Horizontal column span ...
...        ...        ...
=====  =====  =====

+-----+-----+-----+
| Grid | Header 2 | Header 3 |
| table |           |           |
+-----+-----+-----+
| Column 1 | Column 2 | Vertical |
|           | column |           |
+-----+-----+-----+
| Horizontal span | span |
+-----+-----+-----+
    
```

Simple tables and grid tables can be replaced with external CSV files, using the csv-table directive.

Images and Figures

```

.. image:: image.png
   :height: 100px
   :width: 100px
   :align: bottom
   :target: target_

.. figure:: image.png
   :height: 100px

Figures are images with captions. They support all image options.
    
```

Comments

```

.. This is a single line comment, comments can span multiple lines as well.
    
```

1.6.2 Sphinx Cheat Sheet

Directives

reStructuredText directives consist of a **directive type**, **arguments**, and any number of **options**. Some directives expect a block of indented content.

```
.. type:: arguments
   :option: option value

   Directive content
```

```
.. container:: [container class]
.. csv-table:: [table title]
   :header: CSV data for headers
   :widths: number [, number]
   :file: filename
   :encoding: encoding
   :header-rows: number
   :delim: character, "tab", or "space"
   :quote: character
   :escape: character
.. rubric:: title
```

Table of Contents

```
.. toctree::
   :maxdepth: number of title levels
   :glob:
   :hidden:
   :numbered:
   :caption: caption text
   :titlesonly:

   [Include paths]
```

Python Domain

Domain directives expect a domain **directive type**, the **construct name** as a directive argument, and optionally, **domain directive options**.

```
.. py:method:: MyClass.foo(args)
   :param args: Arguments
   :type args: list

   Link to :py:meth:`MyClass.foo`.
```

Python Domain Directives	Python Domain References
.. py:module:: <i>module</i>	:py:mod:` <i>module</i> `
.. py:class:: <i>signature</i>	:py:class:` <i>signature</i> `
.. py:function:: <i>signature</i>	:py:func:` <i>signature</i> `
.. py:data:: <i>name</i>	:py:data:` <i>name</i> `
.. py:exception:: <i>name</i>	:py:exc:` <i>name</i> `
.. py:attribute:: <i>name</i>	:py:attr:` <i>name</i> `
.. py:method:: <i>signature</i>	:py:meth:` <i>signature</i> `
.. py:staticmethod:: <i>signature</i>	
.. py:classmethod:: <i>signature</i>	
.. py:decorator:: <i>signature</i>	
.. py:currentmodule:: <i>module</i>	:py:const:` <i>name</i> `

Python Domain Directive Options

```
:param [type] name: description
:type name: type
:raises class: description
:var name: description
:vartype name: type
:returns description:
:rtype type:
```

There are a number of other Sphinx language domains for representing code constructs in reference documentation: rst, c, cpp, js, ruby, php, dotnet, scala, go, lisp, coffee, and others.

Content Block Directives

```
.. topic:: [title]
.. sidebar:: [title]
   :subtitle: subtitle
.. admonition:: title
.. attention::
.. caution::
.. danger::
.. error::
.. hint::
.. important::
.. note::
.. tip::
.. warning::
.. seealso::
.. deprecated:: [version]
.. versionadded:: [version]
.. versionchanged:: [version]
.. math::
.. raw:: output format
```

```
.. topic:: Examples

   A topic block

.. note::
   This is a note.

.. versionchanged:: 1.3
   Something changed.

.. math::
   a_1 = b_1 + c_1

.. raw:: html

   <b>HTML output</b>
```

Code Examples

```
.. highlight:: language
   :linethreshold: number
.. codeblock:: [language]
   :linenos:
   :emphasize-lines: number [, number]
   :caption: caption text
   :name: block target name

   [Code example, indented]
.. literalinclude:: filename
   :language: language
   :emphasize-lines: number [, number]
   :linenos:
   :encoding: encoding
   :diff: filename
   :dedent: number
```

Some of the language lexers supported by the code example language option: none, python, js, php, ruby, perl, c, cpp, csharp, go, scala, lisp, coffee, dart, julia, lua, html, css, sass, json, yaml, diff, sql, bash, and shell-session

References

```
:role:`title <target>` Link to target with link text title
:role:`!title` Don't create link or reference
:role:`~module.Object` Use last element, Object, for link text
```

Cross-reference roles	Other inline markup roles
:any:` <i>name</i> `	:abbr:` <i>long (abbreviation)</i> `
:doc:`document name`	:command:` <i>name</i> `
:download:`filename`	:kbd:` <i>keystrokes</i> `
:envvar:` <i>name</i> `	:program:` <i>program</i> `
:keyword:`python keyword`	:pep:` <i>number</i> `
:option:`CLI option`	:rfc:` <i>number</i> `

Brought to you by **Read the Docs**.
 We offer private hosting <http://readthedocs.com> and open source hosting <http://readthedocs.org> for *Sphinx* documentation projects.

1.7 Crawler Step 1 Documentation

Our Crawler will make your life as a web developer easier. You can learn more about it in our documentation.

1.7.1 Installation

At the command line:

```
easy_install crawler
```

Or, if you have pip installed:

```
pip install crawler
```

1.7.2 Support

The easiest way to get help with the project is to join the `#crawler` channel on [Freenode](#). We hang out there and you can get real-time help with your projects. The other good way is to open an issue on [Github](#).

The mailing list at <https://groups.google.com/forum/#!forum/crawler> is also available for support.

1.8 Crawler Step 2 Documentation

Our Crawler will make your life as a web developer easier. You can learn more about it in our documentation.

1.8.1 Installation

At the command line:

```
easy_install crawler
```

Or, if you have pip installed:

```
pip install crawler
```

1.8.2 Support

The easiest way to get help with the project is to join the `#crawler` channel on [Freenode](#). We hang out there and you can get real-time help with your projects. The other good way is to open an issue on [Github](#).

The mailing list at <https://groups.google.com/forum/#!forum/crawler> is also available for support.

1.8.3 Cookbook

Crawl a web page

The most simple way to use our program is with no arguments. Simply run:

```
python main.py -u <url>
```

to crawl a webpage.

Crawl a page slowly

To add a delay to your crawler, use `-d`:

```
python main.py -d 10 -u <url>
```

This will wait 10 seconds between page fetches.

Crawl only your blog

You will want to use the `-i` flag, which will ignore URLs matching the passed regex:

```
python main.py -i "^blog" -u <url>
```

This will only crawl pages that contain your blog URL.

1.8.4 Command Line Options

These flags allow you to change the behavior of **Crawler**. Check out how to use them in the *Cookbook*.

`-d <sec>`, `--delay <sec>`

Use a delay in between page fetches so we don't overwhelm the remote server. Value in seconds.

Default: 1 second

-i <regex>, **--ignore** <regex>
Ignore pages that match a specific pattern.
Default: None

1.8.5 Crawler Python API

Getting started with Crawler is easy. The main class you need to care about is *Crawler*

crawler.main

Main Module

class `crawler.main.Crawler` (*url*, *delay*, *ignore*)
Main Crawler object.

Example:

```
c = Crawler('http://example.com')
c.crawl()
```

Parameters

- **delay** – Number of seconds to wait between searches
- **ignore** – Paths to ignore

crawl ()

Crawl the URL set up in the crawler.

This is the main entry point, and will block while it runs.

get (*url*)

Get a specific URL, log its response, and return its content.

Parameters *url* – The fully qualified URL to retrieve

`crawler.main.run_main` ()

A small wrapper that is used for running as a CLI Script.

1.9 Crawler Step 3 Documentation

Our Crawler will make your life as a web developer easier. You can learn more about it in our documentation.

1.9.1 Installation

At the command line:

```
easy_install crawler
```

Or, if you have pip installed:

```
pip install crawler
```

1.9.2 Support

The easiest way to get help with the project is to join the `#crawler` channel on [Freenode](#). We hang out there and you can get real-time help with your projects. The other good way is to open an issue on [Github](#).

The mailing list at <https://groups.google.com/forum/#!forum/crawler> is also available for support.

1.9.3 Cookbook

Crawl a web page

The most simple way to use our program is with no arguments. Simply run:

```
python main.py -u <url>
```

to crawl a webpage.

Crawl a page slowly

To add a delay to your crawler, use `-d`:

```
python main.py -d 10 -u <url>
```

This will wait 10 seconds between page fetches.

Crawl only your blog

You will want to use the `-i` flag, which will ignore URLs matching the passed regex:

```
python main.py -i "^blog" -u <url>
```

This will only crawl pages that contain your blog URL.

1.9.4 Command Line Options

These flags allow you to change the behavior of **Crawler**. Check out how to use them in the *Cookbook*.

-d <sec>, **--delay** <sec>

Use a delay in between page fetchs so we don't overwhelm the remote server. Value in seconds.

Default: 1 second

-i <regex>, **--ignore** <regex>

Ignore pages that match a specific pattern.

Default: None

1.9.5 Crawler Python API

Getting started with Crawler is easy. The main class you need to care about is *Crawler*

crawler.main

Main Module

class `crawler.main.Crawler` (*url*, *delay*, *ignore*)

Main Crawler object.

Example:

```
c = Crawler('http://example.com')
c.crawl()
```

Parameters

- **delay** – Number of seconds to wait between searches
- **ignore** – Paths to ignore

crawl ()

Crawl the URL set up in the crawler.

This is the main entry point, and will block while it runs.

get (*url*)

Get a specific URL, log its response, and return its content.

Parameters **url** – The fully qualified URL to retrieve

`crawler.main.run_main()`

A small wrapper that is used for running as a CLI Script.

crawler.utils

`utils.should_ignore(url)`

Returns True if the URL should be ignored

Parameters

- **ignore_list** – The list of regexs to ignore.
- **url** – The fully qualified URL to compare against.

```
>>> should_ignore(['blog/$'], 'http://ericholscher.com/blog/')
True
```

```
# This test should fail
>>> should_ignore(['home'], 'http://ericholscher.com/blog/')
True
```

`utils.log(status)`

Log information about a response to the console.

Parameters

- **url** – The URL that was retrieved.
- **status** – A status code for the *Response*.

```
>>> log('http://ericholscher.com/blog/', 200)
OK: 200 http://ericholscher.com/blog/
```

```
>>> log('http://ericholscher.com/blog/', 500)
ERR: 500 http://ericholscher.com/blog/
```

```
# This test should fail
>>> log('http://ericholscher.com/blog/', 500)
OK: 500 http://ericholscher.com/blog/
```

Python Module Index

C

`crawler.main`, 38

Index

Symbols

- d <sec>, -delay <sec>
 - command line option, [35, 38](#)
- i <regex>, -ignore <regex>
 - command line option, [35, 38](#)

C

Code Example

Syntax, [9](#)

command line option

-d <sec>, -delay <sec>, [35, 38](#)

-i <regex>, -ignore <regex>, [35, 38](#)

crawl() (crawler.main.Crawler method), [36, 38](#)

Crawler (class in crawler.main), [36, 38](#)

crawler.main (module), [36, 38](#)

G

get() (crawler.main.Crawler method), [36, 38](#)

H

Hyperlink

Syntax, [9](#)

L

log() (crawler.utils method), [39](#)

R

run_main() (in module crawler.main), [36, 38](#)

S

should_ignore() (crawler.utils method), [39](#)

Syntax

Code Example, [9](#)

Hyperlink, [9](#)

TOC Tree, [10](#)

T

TOC Tree

Syntax, [10](#)