FAH-Documentation-Test Documentation

Release 0.0.1

LPunizh

Dec 10, 2020

CONTENTS:

This is a test build for Folding@Home's User Guide Documentation.

Folding@home (FAH or F@h) is a distributed computing project for simulating protein dynamics, including the process of protein folding and the movements of proteins implicated in a variety of diseases. It brings together citizen scientists who volunteer to run simulations of protein dynamics on their personal computers. Insights from this data are helping scientists to better understand biology, and providing new opportunities for developing therapeutics.

CHAPTER

ONE

USER GUIDE

This is a test build for Folding@Home's User Guide Documentation.

User Guide Documentation.

1.1 General

This is a test build for Folding@Home's General Documentation.

General Documentation.

1.1.1 Installation Guides

1.1.2 Running Folding@Home

1.1.3 Troubleshooting

Do I need administrator privileges to install the FAH client?

In most cases, yes. If you do not have administrator privileges or cannot access an administrator account, this indicates that you do not own or manage the computer. Most clients need to have some administrative access to complete the installation. If you do not have the rights, you are advised to ask for permission before going any further.

It is against the FAH End User License Agreement (EULA) to circumvent these restrictions without the expressed permission of the computer's owner. If you are the owner and do not know how to use administrator privileges, please ask someone with more experience to help you.

How can I get installation help if I have a problem?

When one or more steps are difficult to comprehend, you can ask for help on the Folding Support Forum. Someone will provide the answer or help you to complete the installation. In some cases, you may have additional questions after the installation. Feel free to Search the Guides and FAQ pages for answers. You can also Search the Forum for answers or Tools, post a new question, or read through the FAH FAQs.

When asking for help in the forum, please let us know which install guide you are using so we can follow along. And if you know your computer specifications and the client info you are trying to install (i.e. operating system, client version, client type, driver version, etc.) please list these with your question. This information is not required but will help us answer your questions more quickly.

1.1.4 Rules & Policies

1.1.5 Stats, Teams and Usernames

1.1.6 Points

1.1.7 OpenSource

FAH is built from several open source tools, namely Gromacs (http://www.gromacs.org), TINKER (http://dasher.wustl.edu), and AMBER (http://ambermd.org/) for MD packages and MPICH for MPI (http://www-unix.mcs.anl.gov/mpi/mpich/). If you're interested in checking out these codes, you should feel free to download them and check them out. One can compile a SMP version of Gromacs by using the latest Gromacs with MPICH. This would reproduce the SMP clients we have on FAH.

DATA

We've partnered with the Simbios National Biomedical Computing Center to provide data for download. If you're curious, check out our first data set project page

https://simtk.org/home/foldvillin

We'll be releasing more data as time goes on. Our hope is that by making the raw data openly available, this will greatly supplement the published results.

FULL OPEN SOURCING OF THE CLIENT

We have not outsourced the client for several reasons, relating to client reliability and other issues. However, we've come up with a compromise — we have been developing a plug in architecture to allow people to write open source code that we can plug into our client. Visit the Folding Support Forum to discuss, ask questions, and show off your work.

ISN'T GROMACS A GPL'D CODE? WHERE'S THE SOURCE FOR YOUR MODS?

Folding@home has been granted a non-commercial, non-GPL license for Gromacs, so we are not required to release our source. We have analogous license for the other core codes. The copyright owners of any GPL code (in this case the owners are the Gromacs development team) can distribute the same piece of software with difference licenses in parallel. See the GPL FAQ for more info on this. However, we will release our patches back to the Gromacs tree (and have discussed this extensively with the Gromacs team).

We are also working to release our GPU code and other aspects of FAH mods in a new open library called OpenMM. You can learn more about that here: https://simtk.org/home/openmm.

It is important to note that we do release the scientific modifications back to the open source community, but do not release information which would enable donors to cheat on points, which some donors have done ruining the experience for many others.

1.1.8 Donation

WHAT WOULD MY DONATION OF FUNDS GO TO?

We are currently raising funds for new servers, to increase our server reliability. In general, we have needs both large (e.g. servers, graduate students, developers etc.) and small (memory, hard drives and backups). For example, we have to deal with over 500 terabytes of valuable scientific data, and more data comes in every day, all of which must be backed up and carefully preserved. This process is necessary, but quite expensive. Every little bit helps.

1.1.9 FLOPS

1.1.10 Press Info

1.1.11 Miscellaneous

1.2 FAQ

You'll find a large library of frequently asked questions about all things connected to Folding@home.

This section contains everything from guides and information about how to install and use the Folding@home software to the science behind our research. Look through the sidebar and find the topic you want to know more about.

Overview

1.2.1 Can I run Folding@home on a machine I don't own?

Please only run Folding@home on machines you either own or on which you have the permission of the owner to run our software. If there is any doubt (eg you want to run on computers at work), we suggest you get written approval (eg get your superior to sign a letter giving authorization); we have found that written documentation of this sort is important if there is ever any dispute of whether permission was indeed granted. Please do not assume that permission is granted by the owner. Any other use of Folding@home violates our end user license agreement (EULA), and just isn't a good idea in general.

1.2.2 What are the minimum system requirements?

All computers can contribute to Folding@home. However, if the computer is too slow (e.g. wasn't built in the last 5 years or so), the computer might not be fast enough to make the deadlines of typical work units. A Pentium 4 or newer equivalent computer (with SSE) is able to complete work units before they expire.

Folding part time (less than 24 hours a day) will increase the minimum system requirements to make the deadlines. In this case, it is up to the donor to determine how many hours a day are required to fold and still meet the deadlines with each computer.

1.2.3 What happens if there is a suspected license (EULA) violation?

We will attempt to contact the donor if there is some suspicion of a EULA violation. Many donors use their email as their donor name and this is helpful. If do not have any information on hand and we have been presented with a sufficiently strong case that there was a EULA violation, we will zero the points of the donor and not allow clients to run under the name of that donor. This decision can be reversed if there is sufficient information to exonerate the donor. The donor should contact one of the Pande Group members or Forum Moderators at our forum to get in touch with us in such a situation.

1.2.4 What has been "folded" so far, and how much have I folded?

We've simulated a wide variety of proteins and other molecules. We divide the simulations into packets called "Work Units," (WUs) each of which is sent to a computer for processing. We then assemble all the WUs from a project into a completed simulation. We keep many types of statistics of users and work accomplished in our Stats section. You can check your Individual stats, Team stats, and overall Project stats. Please also review the Results and Awards sections.

1.2.5 What has the project completed so far?

We have been able to fold several proteins into the 1.5 millisecond time range with experimental validation of our folding kinetics. This is a timescale a thousand times longer than any previous atomic-level simulation, and represents a fundamental advance over previous work. We are now simulating important proteins used in structural biology studies of folding as well as proteins involved in disease. We've been able to perform detailed simulations of many of these proteins at biologically-relevant timescales, giving us insights that had previously been unobtainable. We've also identified several potential drug candidates which may be able to fight Alzheimer's, cancer, and infection by viruses. Peer-reviewed scientific papers detailing our results are posted on our Results page, and many of them are published in top journals such as Science, Nature, PNAS, and JMB. These publications are highly detailed and often technical, but summaries of their findings can be found on Results page as well as the Folding@home article on Wikipedia.

1.2.6 What is distributed computing?

Distributed Computing is a method of computer processing in which different parts of a program, or different portions of data, are processing simultaneously on two or more computers. We are able to use this approach to significantly accelerate our research. Folding@home is one of the largest, most powerful, and most widely distributed computing networks.

1.2.7 What is Folding@home? What is protein folding?

Folding@home is a distributed computing project, that very simply stated, studies protein folding and misfolding. Protein folding is explained in more detail in the scientific background section. It also helps us develop drugs to combat disease.

1.2.8 Who "owns" the results? What will happen to them?

Folding@home is run by an academic institution (specifically the Pande Group, at Stanford University's Chemistry Department), which is a nonprofit institution dedicated to science research and education. We will not sell the data or make any money from it. Moreover, we will make the data available for others to use. In particular, the results from Folding@home will be made available on several levels. Most importantly, analysis of the simulations will be submitted to scientific journals for publication, and these journal articles will be posted on the web page after publication.

Following the publications of these scientific articles, w e will make the raw data of the folding runs available to other researchers upon request. The data sets from some of our most prominent simulations are already publicly available. We've also striven to share our key technologies with other scientists, to assist their research as well.

1.2.9 Why don't you post the source code?

Most of the critical parts of FAH are publicly available. The Tinker and Gromacs source codes can be downloaded and run. Unlike many computer projects, the paramount concern is not functionality, but the scientific integrity, and posting the source code in a way that would allow people to reverse engineer the code to produce bogus scientific results would make the whole project pointless. However, we stress that the vast majority of our code is already open source. We have an Open Source FAQ with more details.

1.2.10 Why not just use a supercomputer?

Modern supercomputers are essentially clusters of hundreds of processors linked by fast networking. The speed of these processors is comparable to (and often slower than) those found in PCs! Supercomputers are not only very expensive to operate, but they are often simultaneously shared by many different research groups, and it is a challenge to scale a molecular simulation to all of their processors. Protein folding dynamics is statistical in nature, so a single long simulation from a supercomputer would not be sufficient to fully understand the folding process. Folding@home is one of the most powerful computing systems on the planet, and we use novel methods to utilize its network to statistically analyse the dynamics of protein folding. Hence, the calculations performed on Folding@home would not be possible by any other means! This is possible since PC processors are now very fast and there are hundreds of millions of PCs sitting idle in the world.

CHAPTER

TWO

CONTACT US

2.1 Do you have technical problems with the client(app)?

Tech support is found at:

- Discord
- Foldingforum.org

Do you want to report a client issue or bug?

- Github

2.2 Do you want to report a security vulnerability?

Email Joseph: joseph@cauldrondevelopment.com

2.3 Does your company want to help us?

Email Anton: thynell@stanford.edu

2.4 Media inquiry?

Email Anton: thynell@stanford.edu

CHAPTER

THREE

RESTRUCTUREDTEXT (RST) TUTORIAL

3.1 Learn to write beautiful documents

Overview

3.1.1 Introduction

reStructuredText (one word) is a plain-text markup language for writing technical documents, books, websites, and more. It is easy to read and write because it is just regular text and all you need is a simple text editor. Even Notepad would suffice. Despite it being written in plain-text, it is powerful enough to create professional technical documentation, books, and websites.

There are software tools that will convert your plain-text document in to the desired output format. For example: HTML, PDF, ePub, or custom. It is capable of auto-generating table-of-contents, hyperlinks between documents, creating headings, tables, and many other elements. It is also extendable and customizable.

In this tutorial, we will walk through everything you need to know to excel with reStructuredText. We will start with th e basics, but if you follow through to the end, you will have a deep understanding of how it works and how to extend the source code to suit your own needs.

The Python community were the first adopters since it's design specification was written as a Python Enhancement Proposal (PEP) by David Goodger in 2001.

PEP-257: Docstring Conventions PEP-258: Docutils Design Specification http://docutils.sourceforge.net/docs/peps/pep-0257.html http://docutils.sourceforge.net/docs/peps/pep-0258.html

Do not be misled in to believing reStructuredText is only good for Python documentation. Python does use reStructuredText as the standard documentation, but it can be used for any kind of documentation and even writing books and making HTML pages. GitHub supports reStructuredText and will automatically process a README.rst and provide the HTML output if someone lands on your project.

This is aimed at technical writers, programmers, and authors who want an easy yet powerful way to write documents and books. If you are already familiar with Markdown, reStructuredText should come naturally. It is just as easy to get started with, but has many more powerful features available. If you are currently using Markdown, I highly recommend you give reStructuredText a chance. The learning curve is small if you already know very basic Markdown syntax.

http://docutils.sourceforge.net/docs/index.html

3.1.2 Create a basic .rst document

Before getting in to all of the features, let's look at a very basic reStructuredText file. reStructuredText files typically have `.rst` or `.txt` extensions. Here is a simple document:

```
================
My Project Readme
================
-------------------------
Clever subtitle goes here
-------------------------

Introduction
============

This is an example reStructuredText document that starts at the very top
with a title and a sub-title. There is one primary header, Introduction.
There is one example subheading below.
The document is just plain text so it is easily readable even before
being converted to HTML, man page, PDF or other formats.

Subheading
----------

The basic syntax is not that different from Markdown, but it also
has many more powerful features that Markdown doesn't have. We aren't
taking advantage of those yet though.

- Bullet points
- Are intuitive
- And simple too
```

That's a basic reStructuredText document. We haven't looked at any of the more powerful features like hyperlinking or adding images yet. You can save this file as `README.rst` or whatever you want and then store it. If you are familiar with Markdown syntax, you'll notice it's not very different. GitHub also supports `.rst` files so if you include a `README.rst`, GitHub will convert it to HTML when people land on the repository.

3.1.3 Tools

There are two primary tools for converting reStructuredText in to finished products. First, there is `docutils` which is contains the core parser and writer tools. Then there is `Sphinx` which is built on top of docutils and intended for larger projects. Sphinx adds even more functionality and can be used to create very professional looking documents.

docutils

docutils is a Python package that contains classes and scripts that can parse, format, and output to various formats like HTML.

Install docutils

Install the Python **docutils** package in the terminal using `pip` with:

```
python -m pip install docutils
```

Convert RST documents

There are several tools that come with the package, but some primary ones are:

```
rst2html4
rst2html5
rst2man
rst2xml
rst2latex
```

These tools will output to stdout that can be piped to a file like this:

```
rst2html5 mydoc.rst > mydoc.html
```

You can create custom functions called directives to enhance your markup. You can also create custom writers to output the parsed document in different ways.

Sphinx

Sphinx is built on top of `docutils`. While tools like `rst2html5` that come with `docutils` will turn a `.rst` file in to a `.html`, it is generally good for a single page. Sphinx is good for larger documentation or writing projects. You can have multiple `.rst` files in your project to organize and link between them. Sphinx is much more powerful and can be used to publish books and websites using reStructuredText.

Sphinx has a few output options. Among the options are building a website of multiple HTML documents that link together. When you view documentation on https://readthedocs.org or you read the official Python documentation, those are Sphinx generated pages. You can also build it as a single- page HTML document. It also offers plain-text, PDF, epub, and LaTeX builders.

Sphinx also adds a few custom directives (the `..` prefixed functions) like the `toctree` which allows you to embed the table of contents and link to another document.

If you are going for "serious" documentation, Sphinx is the choice.

Install Sphinx

Sphinx is a Python package build on top of `docutils` and can be installed with `pip` like this:

```
python -m pip install sphinx
```

Start a new project

Once sphinx is installed, you don't generally invoke `sphinx-build` directly to build a project the way you call something like `rst2html`. Instead, you call `sphinx-quickstart` which will generate a new project with its own build script. For example, this command will create a new directory called `docs` and put the project inside of it:

```
sphinx-quickstart docs
```

It will prompt you for a project name and an author name, as well as many other questions. You can select all of the defaults if you are unsure.

The new project will have an `index.rst` for you to start editing, as well as a `Makefile` and a `make.bat` so you can build it on Windows. It will also have a directory for templates and building if you want to customize the output. You can add custom CSS and HTML.

Edit the `index.rst` and add other pages and subdirectories as needed. When you are done editing, you can make/build the project with the `make` command.

Build the project

Once you are ready to build the reStructuredText documents in to their final form, you call `make`:

```
make  # Will print all options
make.bat # In Windows, sphinx-quickstart creates a make.bat in project root

# Build the documents to various formats
make html
make singlehtml
make epub
make man
make latex
make text

# Python specific (not covered here)
make doctest   # Run unit tests embedded in docstrings
make coverage  # Check documentation coverage of code
```

They will end up in the `_build` directory.

Pandoc

Pandoc is a universal document converter. It can take sources in reStructuredText, Markdown, LaTeX, Microsoft Word docx, Open Document odt, HTML, epub, and many others and convert it in to various output formats including HTML, docx, odt, ppt, epub, PDF, or other markup formats like Markdown, RST, AsciiDoc.

Pandoc is intended to be a uinversal may behave slightly differently from dedicated RST tools like `docutils` and `Sphinx` which uses `docutils`. I would only recommend using Pandoc if you have special requirements to convert to a format that is not suported by `docutils` and `sphinx`.

Install pandoc

Full installation instructions at Pandoc.org

Releases for Windows, Mac, Linux are available for download on GitHub at https://github.com/jgm/pandoc/releases.

- **Windows**: Download the installer from the pandoc releases page.
- **Mac**: Use brew to install: `brew install pandoc`
- **Linux**: Download the linux.tar.gz file and extract it. The `bin/` directory contains the `pandoc` executable. Alternatively, in Debian based distributions, there is a `.deb` package available for download on that GitHub releases page.

Readthedocs.org

The website https://readthedocs.org/ is a great website that will build and host your documentation for you. You can use Sphinx, Mkdcocs, or generate your own documents. I recommend using Sphinx. When you use `sphinx-quickstart` to generate a `docs/` directory in your project, ReadTheDocs.org will know what to do. Sign up and tell ReadTheDocs.org about your repository, and set up a webhook to automatically build and host your documentation any time there is a git push.

They will host multiple versions of documentation, you can use their theme or custom themes, and it even makes PDF and ePub versions available.

3.1.4 Syntax examples

This is a mashup of common syntax. It's like a cheatsheet for quick reference. There is some freedom with reStructuredText that allows you to pick different characters for creating headers and bulleted lists. As long as you are consistent throughout your document it will interpret the headers automatically. This example uses my preferred characters and styling for headings.

Overline and underline combined is separate from just underline. The lines can be created with any of the following characters, based on preference. You just need to be consistent within a single document, `' " . , : ; ! ? - ) ] } / \ >`.

Try saving the contents of this example to `sample.rst` and build it to HTML to see how it looks yourself with:

```
rst2html5 sample.rst > sample.html
```

Some of the elements covered in this example are:

- Headings
- Comments
- Images
- Lists
- Preformatted text
- Code blocks
- Links
- Footnotes
- Transitions/lines/horizontal rules
- Tables

- Preserving line breaks

Here is the sample reStructuredText:

```
""""""""""""""""""
Document Title
""""""""""""""""""
..........
Subtitle
..........

.. contents:: Overview
   :depth: 3

==================
Section 1
==================

Text can be *italicized* or **bolded**  as well as ``monospaced``.
You can \*escape certain\* special characters.

----------------------
Subsection 1 (Level 2)
----------------------

Some section 2 text

Sub-subsection 1 (level 3)
--------------------------

Some more text.

=========
Examples
=========

--------
Comments
--------

.. This is a comment
   Special notes that are not shown but might come out as HTML comments

------
Images
------

Add an image with:

.. image:: screenshots/file.png
   :height: 100
   :width: 200
   :alt: alternate text

You can inline an image or other directive with the |customsub| command.

.. |customsub| image:: image/image.png
              :alt: (missing image text)
```

(continues on next page)

(continued from previous page)

```
-----
Lists
-----

- Bullet are made like this
- Point levels must be consistent
    * Sub-bullets
        + Sub-sub-bullets
- Lists

Term
    Definition for term
Term2
    Definition for term 2

:List of Things:
    item1 - these are 'field lists' not bulleted lists
    item2
    item 3

:Something: single item
:Someitem: single item

-----------------
Preformatted text
-----------------

A code example prefix must always end with double colon like it's presenting␣
↪something::

    Anything indented is part of the preformatted block
   Until
  It gets back to
 Allll the way left

Now we're out of the preformatted block.

------------
Code blocks
------------

There are three equivalents: ``code``, ``sourcecode``, and ``code-block``.

.. code:: python

   import os
   print(help(os))

.. sourcecode::

  # Equivalent

.. code-block::

  # Equivalent
```

(continues on next page)

(continued from previous page)

```
-----
Links
-----

Web addresses by themselves will auto link, like this: https://www.devdungeon.com

You can also inline custom links: `Google search engine <https://www.google.com>`_

This is a simple link_ to Google with the link defined separately.

.. _link: https://www.google.com

This is a link to the `Python website`_.

.. _Python website: http://www.python.org/

This is a link back to `Section 1`_. You can link based off of the heading name
within a document.

---------
Footnotes
---------

Footnote Reference [1]_

.. [1] This is footnote number one that would go at the bottom of the document.

Or autonumbered [#]

.. [#] This automatically becomes second, based on the 1 already existing.

-----------------
Lines/Transitions
-----------------

Any 4+ repeated characters with blank lines surrounding it becomes an hr line, like␣
↪this.

====================================

------
Tables
------

+--------+--------+--------+
| Time   | Number | Value  |
+========+========+========+
| 12:00  | 42     | 2      |
+--------+--------+--------+
| 23:00  | 23     | 4      |
+--------+--------+--------+

----------------------
Preserving line breaks
----------------------

Normally you can break the line in the middle of a paragraph and it will
```

(continues on next page)

(continued from previous page)

```
ignore the newline. If you want to preserve the newlines, use the ``|`` prefix
on the lines. For example:

| These lines will
| break exactly
| where we told them to.
```

3.1.5 Splitting up a document in to multiple files

Sphinx has a special directive for building linking pages together and embedding a table of contents from another page. The `toctree` directive will essentially import the headings/table of contents from the file specified. It is good for creating a master landing page that links to sub-documents. Here is an example of its usage snipped from the Python official documentation at https://github.com/python/cpython/tree/master/Doc:

```
.. toctree::

   whatsnew/index.rst
   tutorial/index.rst
   faq/index.rst
   glossary.rst

   about.rst
   bugs.rst
   copyright.rst
   license.rst
```

And inside each one of those directories/.rst files, you can put more `toctree` elements if you have nested levels of complexity.

3.1.6 Create a custom writer

The `docutils` package comes with several writers, including html4, html5, and odf. If you aren't satisfied with the existing output formats, you can create a custom `Writer`. You can subclass the existing writers if you want to extend or modify them. We will look at how to extend the `docutils.writers.html5_polyglot.Writer` class to override different methods and modify the output.

We'll make a custom writer that outputs slightly modified HTML and we will print out just the body with no HTML boilerplate. The goal is to generate HTML that can be inserted in to a content management system like Drupal where it is assumed the outer HTML template and CSS styling is already available and all you need is the content section.

Base classes

`docutils` provides the base classes and tools needed. We will need to get more familiar with the writer and translator classes. They are intimately tied together and we will be creating subclasses of both.

I found a great tutorial on this topic at http://www.arnebrodowski.de/blog/write-your-own-restructuredtext-writer.html that covers this topic. I recommend giving it a read.

After inspecting the source code for `docutils`, this is how the class structure is set up for the html writer and translator.

Writer class heirarchy for the HTML writers:

- **`docutils.writers.Writer`**

  - **docutils.writers._html_base.Writer**
    - docutils.writers.html5_polyglot.Writer
    - docutils.writers.html4css1.Writer

Translator class heirarchy for HTML translators:

- **docutils.nodes.NodeVisitor**
  - docutils.nodes.GenericNodeVisitor
  - **docutils.writers._html_base.HTMLTranslator**
    - docutils.writers.html5_polyglot.HTMLTranslator
    - docutils.writers.html4css1.HTMLTranslator

You need to define a custom Writer and Translator. The translator defines the logic used by the writer on how to output/wrap each node. You can subclass an existing writer. These are some of the existing writer and translator classes related to HTML output.

Note that this is only a few of the writer classes, there are several other subclasses in the `docutils.writers` package, but these are the ones we're interested in since we're looking for custom HTML output. You can subclass any one of these, depending on how much logic you want to inherit. We should choose wether to subclass the lowest level class like `docutils.writers.Writer` or the highest level one available in the `html5_polyglot` module, `docutils.writers.html5_polyglot.Writer`.

Minimal custom HTML writer

Here is a minimal example of how to create and use your own writer. In this case, we are simply inheriting the behavior of the `html5_polyglot` writer and translator that come with `docutils` package. We aren't modifying any of the behavior yet, but it's a good starting place.

Custom writer example:

```
"""Minimal writer/translator for customizing docutils output"""
from docutils.writers import html5_polyglot
from docutils.core import publish_string

class MyCustomHTMLTranslator(html5_polyglot.HTMLTranslator):
    pass

class MyCustomHTMLWriter(html5_polyglot.Writer):
    def __init__(self ):
        html5_polyglot.Writer.__init__(self)
        self.translator_class = MyCustomHTMLTranslator

if __name__ == '__main__':
    html_output = publish_string(source='Put reStructured text here.',
                                 writer=MyCustomHTMLWriter())
    print(html_output)
```

That example will run and output HTML. Now you are free to modify the behavior of the writer or the translator. To see how they work under the hood, look in to the source code of its parent class, and the parent of that class too. Find what functions you want to override and implement them in your class.

Full custom HTML writer

Since we want to lower the heading level by one, we should replace the method that is in charge or outputting those header tags. I found it in `docutils.writers._html_base.HTMLTranslator.visit_title()` so I implemented a `visit_title()` method in my own translator by copy and pasting the original one as a starting place. Here is the full code used.

Custom HTML writer example:

```
"""
A custom docutils writer that will convert reStructuredText (RST) to html5,
but slightly modified from the html5_polyglot writer. The goal is to output
only the HTML body with the intention of embedding it inside a larger HTML
document using a content management system (CMS) like Drupal, Wordpress,
or Django.

- It only outputs the body, from subtitle to end of document, no HTML
  boilerplate, no CSS, no title.
- It lowers the heading level by one. It assumes the h1 is being output
  as the document title by the CMS. The output starts with the subtitle,
  and goes to the end of the document.

Built using
http://www.arnebrodowski.de/blog/write-your-own-restructuredtext-writer.html
as a reference.

It has a translator and a writer.

The translator is a  defines how to wrap or output each type of node.
At it's core, the translator is actually a ``nodes.GenericNodeVisitor``
that visits each node and decides how to process it.

The writer is what gets passed to the ``publish_*`` functions in the end
that process the document and provide HTML output.
The writer contains a reference to which translator it will use.

To use this writer, see the __main__ section at the bottom.
You call ``docutils.core.publish_*`` and pass it your customer writer.
"""
from docutils.writers import html5_polyglot
from docutils import nodes
import os


class HTMLBodyTranslator(html5_polyglot.HTMLTranslator):
    """
    Contains all the logic on how to wrap various nodes with HTML.
    For each node type, you can write a ``visit_*`` and ``depart_*``
    method. Copy the existing method from
    ``docutils.writers.html5_polyglot.HTMLTranslator`` if there is one,
    and modify it from there.

    Get list of all node types::

      >>> import docutils.nodes
      >>> docutils.nodes.node_class_names
      >>> help(docutils.nodes)
```

(continues on next page)

(continued from previous page)

```
    node_class_names:
      Text
      abbreviation acronym address admonition attention attribution author
          authors
      block_quote bullet_list
      caption caution citation citation_reference classifier colspec comment
          compound contact container copyright
      danger date decoration definition definition_list definition_list_item
          description docinfo doctest_block document
      emphasis entry enumerated_list error
      field field_body field_list field_name figure footer
          footnote footnote_reference
      generated
      header hint
      image important inline
      label legend line line_block list_item literal literal_block
      math math_block
      note
      option option_argument option_group option_list option_list_item
          option_string organization
      paragraph pending problematic
      raw reference revision row rubric
      section sidebar status strong subscript substitution_definition
          substitution_reference subtitle superscript system_message
      table target tbody term tgroup thead tip title title_reference topic
          transition
      version
      warning
  """

  def visit_title(self, node):
      # Modifed code, copied from parent class
      check_id = 0  # TODO: is this a bool (False) or a counter?
      close_tag = '</p>\n'
      if isinstance(node.parent, nodes.topic):
          self.body.append(
                self.starttag(node, 'p', '', CLASS='topic-title first'))
      elif isinstance(node.parent, nodes.sidebar):
          self.body.append(
                self.starttag(node, 'p', '', CLASS='sidebar-title'))
      elif isinstance(node.parent, nodes.Admonition):
          self.body.append(
                self.starttag(node, 'p', '', CLASS='admonition-title'))
      elif isinstance(node.parent, nodes.table):
          self.body.append(
                self.starttag(node, 'caption', ''))
          close_tag = '</caption>\n'
      elif isinstance(node.parent, nodes.document):
          self.body.append(self.starttag(node, 'h1', '', CLASS='title'))
          close_tag = '</h1>\n'
          self.in_document_title = len(self.body)
      else:
          assert isinstance(node.parent, nodes.section)
          h_level = self.section_level + self.initial_header_level# - 1
          atts = {}
          if (len(node.parent) >= 2 and
              isinstance(node.parent[1], nodes.subtitle)):
```

(continues on next page)

(continued from previous page)

```
                atts['CLASS'] = 'with-subtitle'
            self.body.append(
                  self.starttag(node, 'h%s' % h_level, '', **atts))
            atts = {}
            if node.hasattr('refid'):
                atts['class'] = 'toc-backref'
                atts['href'] = '#' + node['refid']
            if atts:
                self.body.append(self.starttag({}, 'a', '', **atts))
                close_tag = '</a></h%s>\n' % (h_level)
            else:
                close_tag = '</h%s>\n' % (h_level)
        self.context.append(close_tag)

    # Required override
    def should_be_compact_paragraph(self, node):
        if(isinstance(node.parent, nodes.block_quote)):
            return 0


class HTMLBodyWriter(html5_polyglot.Writer):
    """
    A ``docutils`` writer that will output HTML intended to be used within
    a larger existing HTML document, like within a content management system
    blog post.

    Writer that inherits from ``distutils.writers.html5_polyglot.Writer``.
    but overrides the ``translator_class`` which makes a few tweaks
    like lowering the heading levels by one.
    """

    def __init__(self ):
        self.parts = {}
        self.translator_class = HTMLBodyTranslator


if __name__ == '__main__':  # rst2html5body.py
    """
    Take a filename from the first command-line argument,
    process it using the custom writer, and output the body section
    only to standard output.

    Example usage::

      rst2html5body.py readme.rst > readme.html

    Then use that output as the content for your blog post.
    """
    from docutils.core import publish_parts
    import sys

    # First argument provided on the command line is the RST file name
    with open(sys.argv[1]) as rst_file:
        rst_content = rst_file.read()

    # publish_parts() will return a dictionary with the different
    # parts of the document, like head, stylesheet, body, already
```

(continues on next page)

(continued from previous page)

```
    # processed and turned in to HTML, just separated for us.
    # There are other ``publish_*`` options like publish_cmdline,
    # publish_file, publish_string, and more.
    # If you want the final full standalone HTML document with all the
    # boilerplate,use ``publish_string()`` instead.
    output_document_parts = publish_parts(source=rst_content,
                                          writer=HTMLBodyWriter())

    # >>> output_parts.keys()  # List all of the parts available

    # ['whole', 'encoding', 'version', 'head_prefix', 'head', 'stylesheet',
    # 'body_prefix', 'body_pre_docinfo', 'docinfo', 'body', 'body_suffix',
    # 'title', 'subtitle', 'header', 'footer', 'meta', 'fragment',
    # 'html_prolog', 'html_head', 'html_title', 'html_subtitle', 'html_body']

    print(output_document_parts['stylesheet'])
    print(output_document_parts['body'])
```

The code above defines two classes and then provides a __main__ example of how to use it. The `Writer` class is very simple, and it just specifies which translator to use. The bulk of the logic lives in the translator class.

3.1.7 Create a custom directive

Directives are the special lines that start with two dots and are treated as special functions. Some examples we've already see are the table of contents `.. contents::` and images `.. image::`. Under the hood, they really are just calling Python functions. You can create your own custom directives to execute special logic or output dynamic content.

There are several ways you can call a directive, depending on how much input you need to provide to the directive. Here are some different examples:

```
.. mydirective::

.. mydirective2:: Argument1

.. mydirective3:: Somevalue1
   :param1: somevalue2
   :param2: somevalue3

.. mydirective4::

  This is part of directive4.
  All of this will get passed to directive4.
  Until the indentation returns all the way to the left.

Now we're out of the directive body.
```

Read all about the built-in reStructuredText directives at http://docutils.sourceforge.net/0.14/docs/ref/rst/directives.html

What if we made a:

```
.. beware:: dogs
.. image_carousel::
.. slideshow::
```

Start by looking at base directives Minimal example - extending an existing/base directive or full example - actually do something

Use an example Pygments is a Python package for highlighting source code. There is a `..code::` directive that uses Pygment.

In the Pygment source code repository it is in `external/rst-directive.py`. We can use that as a good example.

https://bitbucket.org/birkenfeld/pygments-main/src/7941677dc77d4f2bf0bbd6140ade85a9454b8b80/external/rst-directive.py?at=default&fileviewer=file-view-default

Base class

Built-in directives can be found in the `docutils.parsers.rst.directives` package. In there, you will find all of the functions to register directives, call directives, and all of the directive classes like the `Image` class that corresponds with the `..image::` directive.

All reStructuredText directives inherit from the base class `docutils.parsers.rst.Directive` which is defined in `docutils.parsers.rst.__init__`.

..code:: python

    # Defined in docutils.parsers.rst.__init__.py class Directive(object):

The docstring on this class is actually quite thorough you can access it easily using pydoc from the command line:

```
python -m pydoc docutils.parsers.rst.Directive
```

Or from the interactive python interpreter:

```
>>> import docutils.parsers.rst
>>> help(docutils.parsers.rst.Directive)
```

Refer to those sources for a full list of options. We'll look at a simple example to get you started.

References: # - http://docutils.sourceforge.net/docs/howto/rst-directives.html # - https://bitbucket.org/birkenfeld/pygments-main/src/7941677dc77d4f2bf0bbd6140ade85a9454b8b80/external/rst-directive.py?at=default&fileviewer=file-view-default

Custom directive:

```
from docutils import nodes
from docutils.parsers.rst import directives, Directive


class MyCustomDirective(Directive):
required_arguments = 1
optional_arguments = 0
final_argument_whitespace = True
option_spec = dict([(key, directives.flag) for key in VARIANTS])
has_content = True

def run(self):
    self.assert_has_content()
    self.arguments[0]
    self.options

    if error_condition:
```

(continues on next page)

(continued from previous page)

```
        raise self.error('Error message.')

    # Does format='html' mean it will get ignored in text or odf output?
    # We create a node to return
    return [nodes.raw('', parsed, format='html')]

directives.register_directive('mydirective', MyCustomDirective)


if __name__ == '__main__':
reStructuredText_source = """
=====================
Custom directive test
=====================

.. mydirective::

My directive should have run now.
"""

# Since the directive was registered already, it should get used when
# the parser runs when it encounters the ``.. mydirective::`` text.
html_output = publiish_string(source=reStructuredText_source, writer=html5_polyglot.
↪Writer)

print(html_output)
```

Base classes

`docutils.parsers.rst.Directive`

Minimal custom directive

There are two steps:

- Create a customer directive class that inherits from `docutils.parsers.rst.Directive`
- Register the directive with `docutils.parsers.rst.directives.directives.register_directive()`

Once you have your directive registered, anytime you call one of the `docutils.core.publish_*` functions, it will process your directive if it sees on in the reStructuredText it parses.

Custom directive example:

```
class MyCustomDirective(Directive)
  run():
    print('it ran!')
    return(my custom node? a text node? a code bloc node? an image? etc)
    process without outputting anything?

directives.register('shortcutname', MyCustomDirective)
```

Full custom directive

The previous example was a minimal skeleton. This example will actually do something.

3.1.8 Sublime Text RST Plugin

There is a Sublime Text package that helps when writing reStrucutredText. It helps with navigation, formatting, collapsing blocks, and more.

You can find more information about the plugin at: https://packagecontrol.io/packages/Restructured%20Text%20%28RST%29%20Snippets

The plugin comes with several features, including:

- Auto over/underline formatting for headers
- Smart bullet lists
- Quick build/preview shortcut
- Section folding
- Jump between headers

Setup plugin

Use the Sublime Text package manager to install the plugin. Install that first if you do not already have it.

Then, to install the package, in Sublime Text:

- Press `Ctrl-Shift-P`
- Search for `RST Snippets`
- Highlight the plugin in the search results and press enter to install.

Use the plugin

There are several features in the plugin but I will cover a few basics. See full documentation at https://packagecontrol.io/packages/Restructured%20Text%20%28RST%29%20Snippets for full details.

- When writing headers, just over/underline it with 3 characters and then press tab and it will auto complete the underlines to match the length of the title.
- When making a bullet list, press enter and it will automatically add the next bullet point on the next line. You can also press tab to indent one more level and it will swap out the bullet character to match the next level.
- Quick build/preview shortcut - Press `Ctrl-Shift-R` to build and preview the document.
- Folding and unfolding - Press `Shift-Tab` while the cursor is on a heading
- Jump between headers with `Alt-Up/Down`

3.1.9 Reference Links

- reStructuredText Wikipedia - https://en.wikipedia.org/wiki/ReStructuredText
- Docutils - http://docutils.sourceforge.net/
- reStructuredText directives - http://docutils.sourceforge.net/0.14/docs/ref/rst/directives.html
- Sphinx - http://www.sphinx-doc.org/en/master/
- Pandoc - https://pandoc.org/
- PyPI - https://pypi.org/
- Python Documentation - https://www.python.org/doc/
- ReadTheDocs - python docs
- GitHub - https://github.com/
- Sublime Text Plugin - https://packagecontrol.io/packages/Restructured%20Text%20%28RST%29%20Snippets

CHAPTER

FOUR

NOTED STYLE/FONT

- Bullet
- **Bold**
- *Italic*
- `Monospace`
- **You can *escape certain* special characters.** Tab