

LECTURE 9

Development Tools

DEVELOPMENT TOOLS

Since you are all still in the earlier stages of your semester-long projects, now is a good time to cover some useful modules and tools for managing larger Python projects (especially ones that involve multiple people).

- virtualenv
- documenting
- logging
- unit testing

VIRTUALENV

virtualenv is a tool for creating isolated Python environments.

Let's say you are working on two projects which require Twisted, a python-based networking package. One of the projects requires Twisted 14.x, but the other requires Twisted 13.x. Which should you install? Solve the problem by creating a custom Python environment for each project!

To install virtualenv via pip, just open the command-line and type the following:

```
$ sudo pip install virtualenv
```

Note that all of the details in this lecture are based on Ubuntu 14.04.

VIRTUALENV

To create a virtual environment for a project:

```
$ cd my_project_folder  
$ virtualenv venv
```

Essentially, we're creating a copy of the Python interpreter (as well as a copy of the pip and setuptools libraries) inside of a folder called venv. We can specify a different Python version in the following way:

```
$ virtualenv -p /usr/bin/python2.7 venv
```

Use the `--no-site-packages` option to *not* include globally installed packages.

VIRTUALENV

After creating a new virtual environment, the next step is to activate it.

```
$ source venv/bin/activate  
(venv) $
```

The name of the virtual environment will appear before your prompt after activation. Anything you install at this point will install to your isolated environment, not to the global site packages.

```
(venv) $ pip install twisted
```

VIRTUALENV

To deactivate a virtual environment:

```
(venv) $ deactivate
$
```

Now, we're back to using the default Python interpreter and globally installed packages. You can delete a virtual environment by simply deleting the folder created, in this case called "venv".

VIRTUALENV

For distributing your project and/or for easy set-up, freeze the current virtual environment.

```
(venv) $ pip freeze > requirements.txt
```

This creates a list of the installed packages and versions inside of requirements.txt. This can be used to rebuild the environment later. This is useful for allowing another developer to run your project without having to figure out what packages and which versions were used.

VIRTUALENV

Putting all of this together, the typical use of a virtual environment is as follows:

```
$ virtualenv venv --no-site-packages
$ source venv/bin/activate
(venv) $ pip install -r requirements.txt
...
(venv) $ deactivate
```


VIRTUALENV

You are encouraged to use virtual environments in your projects and to keep an updated requirements.txt with your project.

You are also encouraged to house all of your code in a repository. This makes sharing code between group members much easier. You can just pull the code and create a virtual environment with the included requirements.txt file. Easy!

DOCUMENTATION

Being able to properly document code, especially large projects with multiple contributors, is incredibly important.

Code that is poorly-documented is sooner thrown-out than agonized over. So make sure your time is well-spent and document your code for whoever may need to see it in the future!

Python, as to be expected, has a selection of unique Python-based documenting tools and as a Python developer, it is important that you be familiar with at least one of them.

SPHINX

The Sphinx documentation tool is by far the most popular way to document Python code and is, in fact, used to generate the official Python documentation which is notoriously thorough and well-kept. There exist some other options like Pycco, Epydoc, and MkDocs – but Sphinx truly is the standard.

Getting Sphinx is super easy:

```
$ sudo pip install sphinx
```

WHAT IS SPHINX?

- Introduced in 2008 for documenting Python (and is written in Python).
- Converts reStructuredText to HTML, PDF, man page, etc.
- reStructuredText was developed to be a lightweight markup language specifically for documenting Python.
- But it's not just Python! C/C++ also supported with more languages on the way.

SPHINX FEATURES

- Output formats: HTML, LaTeX (for printable PDF versions), ePub, Texinfo, man pages, plain text.
- Extensive cross-references: semantic markup and automatic links for functions, classes, citations, glossary terms and similar pieces of information.
- Hierarchical structure: easy definition of a document tree, with automatic links to siblings, parents and children.
- Automatic indices: general index as well as a language-specific module indices.
- Code handling: automatic highlighting using the Pygments highlighter.
- Extensions: automatic testing of code snippets, inclusion of docstrings from Python modules (API docs), and more.
- Contributed extensions: 50+ extensions contributed by community – most available from PyPI.

SETTING UP SPHINX

Sphinx documentation requires a variety of source files which are housed within a single source directory.

The central component of the source directory is the `conf.py` file which configures all aspects of a Sphinx project.

The easiest way to get a Sphinx project started is to issue the following command:

```
$ sphinx-quickstart
```

This will guide you through generating a source directory and a default typical `conf.py`.

QUICKSTART

By default, quickstart creates a source directory with `conf.py` and a master document, `index.rst`.

The main function of the master document is to serve as a welcome page, and to contain the root of the “table of contents tree” (or toctree), which connects multiple `rst` files into a single hierarchy.

```
ticket_app/docs$ sphinx-quickstart
```

Welcome to the Sphinx 1.2.3 quickstart utility.

Please enter values for the following settings (just press Enter to accept a default value, if one is given in brackets).

Enter the root path for documentation.

> Root path for the documentation [.]:

...

Finished: An initial directory structure has been created.

You should now populate your master file `./index.rst` and create other documentation source files. Use the Makefile to build the docs, like so:

make builder

where "builder" is one of the supported builders, e.g. `html`, `latex` or `linkcheck`.

```
ticket_app/docs$ ls
```

```
_build conf.py index.rst make.bat Makefile _static _templates
```

INDEX.RST

Let's take a look at the main index.rst they created for us.

The toctree directive allows us to specify other .rst files to include in our documentation.

```
Welcome to Ticket Scraper's documentation!
```

```
=====
```

```
Contents:
```

```
.. toctree::  
    :maxdepth: 2
```

```
Indices and tables
```

```
=====
```

```
* :ref:`genindex`  
* :ref:`modindex`  
* :ref:`search`
```


INDEX.RST

Let's take a look at the main index.rst they created for us.

The toctree directive allows us to specify other .rst files to include in our documentation.

We've currently added two local .rst files. These files themselves may also have toctree directives which are used to grow the tree.

* Leave off the extension and use '/' for directories.

```
Welcome to Ticket Scraper's documentation!
```

```
=====
```

```
Contents:
```

```
.. toctree::  
    :maxdepth: 2
```

```
    intro  
    tech
```

RESTRUCTUREDTEXT

- *Paragraphs*: chunks of text separated by one or more blank lines. All lines of the same paragraph must be left-aligned to the same level of indentation.
- Inline Markup
 - **emphasis** → *emphasis*
 - ****strong emphasis**** → **emphasis**
 - ```code snippet``` -- Note the backquotes.
 - Escape special characters (e.g. asterisks) with a backslash.
 - May not be nested and content must start and end with non-whitespace.
- Interpreted Text Roles → `:rolename: `content``
 - Standard roles: *emphasis*, **strong**, `literal`, *subscript*, *superscript*, etc.

RESTRUCTUREDTEXT

- Lists: just place list markup at start of paragraph and indent accordingly.

- '*' indicated bulleted list.
- '1', '2', '3', etc for manually numbered lists.
- '#' for auto-numbered lists.

- Preserve line breaks with '|'.

```
* this is  
* a list
```

```
* with a nested list  
* and some subitems
```

```
* and here the parent list continues
```

RESTRUCTUREDTEXT

Code snippets are commonly included in the documentation of a project – specifically when describing use.

End the preceding paragraph with the `::` marker and indent the code sample.

Remember! As in Python, whitespace is significant to Sphinx. Typically this means you'll need to indent but sometimes directives and markers cannot be preceded by whitespace – be careful.

This is a normal text paragraph. The next paragraph is a code sample::

It is not processed in any way, except that the indentation is removed.

It can span multiple lines.

This is a normal text paragraph again.

RESTRUCTUREDTEXT

- Section headers are created by overlining and underlining the header content with some punctuation character.
- The heading level of the punctuation character is determined by context, however the convention is:
 - # with overline, for parts.
 - * with overline, for chapters.
 - =, for sections.
 - -, for subsections.
 - ^, for subsubsections.

```
=====
This is a Section heading
=====
```

RESTRUCTUREDTEXT

- Besides toctree, many directives are supported.
 - “Admonition”-elements like attention, tip, and warning, the image and figure directives, etc.
 - [Here](#) are the available directives.
- Comments are constructed like so:

```
.. This is a comment.  
..  
    This whole indented block is a comment.  
    Still in the comment.
```

INTRO.RST

Now that we know more about reStructuredText, we can create intro.rst and add it to our documentation.

intro.rst

Introduction

=====

Goals

The goal of this documentation is to provide a clear and technologically informative basis for the structure, functionality, and design of the Taylor Swift Concert Ticket Price Showdown application.

TECH.RST

We can also make a simple tech.rst, which would supposedly describe the technologies used in our application.

tech.rst

Technologies

=====

BUILDING SPHINX

Now we have some very simple rst files to build our documentation with. Generally speaking, you can build Sphinx projects in the following way:

```
$ sphinx-build -b html sourcedir builddir
```

The `-b` option selects the format.

But we were smart enough to use Sphinx's quickstart tool so we have a makefile.

BUILDING SPHINX

The makefile provided by the quickstart tool allows you to specify a number of targets, including:

- html
- json
- latex
- latexpdf
- man
- xml
- etc.

```
ticket_app/docs$ make html
sphinx-build -b html -d _build/doctrees . _build/html
Running Sphinx v1.2.3
loading pickled environment... done
building [html]: targets for 2 source files that are out of date
updating environment: 0 added, 2 changed, 0 removed
reading sources... [100%] tech
looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] tech
writing additional files... genindex search
copying static files... done
copying extra files... done
dumping search index... done
dumping object inventory... done
build succeeded.
```

Build finished. The HTML pages are in `_build/html`.

BUILDING SPHINX

The screenshot shows a web browser displaying a Sphinx-generated documentation page. The browser's address bar shows the file path: `file:///home/caitlin/Teaching/Python/Demos/ticket_app/docs/_build/html/index.html`. The page has a dark blue header and footer. The main content area is white. On the left, there is a dark blue sidebar with white text. The main content area has a light blue header section, a white content section, and a light blue footer section. The sidebar contains links to the Table of Contents, Next topic, This Page, and Quick search. The main content area features a large heading, a list of contents, a heading for indices and tables, and a list of links. The footer contains copyright information.

file:///home/caitlin/Teaching/Python/Demos/ticket_app/docs/_build/html/index.html Search

Ticket Scraper 1.0 documentation » next | index

Table Of Contents

Welcome to Ticket Scraper's documentation!
Indices and tables

Next topic

Introduction

This Page

Show Source

Quick search

Enter search terms or a module, class or function name.

Welcome to Ticket Scraper's documentation!

Contents:

- Introduction
 - Goals
- Technologies

Indices and tables

- [Index](#)
- [Module Index](#)
- [Search Page](#)

Ticket Scraper 1.0 documentation » next | index

© Copyright 2015, Caitlin. Created using [Sphinx](#) 1.2.3.

BUILDING SPHINX

The screenshot shows a web browser displaying a Sphinx-generated documentation page. The browser's address bar shows the file path: `file:///home/caitlin/Teaching/Python/Demos/ticket_app/docs/_build/html/intro.html#goals`. The page has a dark blue header and footer. The header contains the text "Ticket Scraper 1.0 documentation »" on the left and navigation links "previous | next | index" on the right. The main content area is divided into a left sidebar and a main body. The sidebar, which has a dark blue background, contains several sections: "Table Of Contents" with links to "Introduction" and "Goals" (where "Goals" is highlighted); "Previous topic" with a link to "Welcome to Ticket Scraper's documentation!"; "Next topic" with a link to "Technologies"; "This Page" with a link to "Show Source"; and "Quick search" with a search input field and a "Go" button. The main body has a light gray background and contains the title "Introduction" followed by a section titled "Goals". The "Goals" section contains the text: "The goal of this documentation is to provide a clear and technologically informative basis for the structure, functionality, and design of the Taylor Swift Concert Ticket Price Showdown application." The footer contains the text "Ticket Scraper 1.0 documentation »" on the left and "previous | next | index" on the right, and a copyright notice "© Copyright 2015, Caitlin. Created using [Sphinx](#) 1.2.3." in the center.

file:///home/caitlin/Teaching/Python/Demos/ticket_app/docs/_build/html/intro.html#goals Search

Ticket Scraper 1.0 documentation » previous | next | index

Table Of Contents

- Introduction
 - Goals

Previous topic

Welcome to Ticket Scraper's documentation!

Next topic

Technologies

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Introduction

Goals

The goal of this documentation is to provide a clear and technologically informative basis for the structure, functionality, and design of the Taylor Swift Concert Ticket Price Showdown application.

Ticket Scraper 1.0 documentation » previous | next | index

© Copyright 2015, Caitlin. Created using [Sphinx](#) 1.2.3.

DOCUMENTING CODE

The manual way to document a Python object is like this:

```
.. function:: get_posts(event_name, location)
```

```
    Accesses the Craigslist page of search results for a  
    given event and location. Returns a list of post  
    information tuples, sorted by price.
```

You can reference an object by using the :func: notation, where the object name is in backquotes.

```
The :func:`get_posts` function can be called...
```

There are also directives for classes, methods, etc. Any object can be documented.

SCRAPER.RST

Let's add an rst file to document our scraping code.

Ticket Scraping

=====

Ticket scraping is performed by the `ticket_scraper.py` module. Initial calls should be made to the `:func:`get_posts`` function, which will guide the scraping process and returns a list of relevant posts.

```
.. function:: get_posts(event_name, location)
```

Accesses the Craigslist page of search results for the given event and location. Returns a list of post information tuples, sorted by price.

SCRAPER.RST

←

file:///home/caitlin/Teaching/Python/Demos/ticket_app/docs/_build/html/scraper.html#get_posts

↻

Search

↓

🏠

☆

📄

👤

ABP

⌵

📺

☰

Ticket Scraper 1.0 documentation »

previous | index

Previous topic

Technologies

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Ticket Scraping

Ticket scraping is performed by the `ticket_scraper.py` module. Initial calls should be made to the `get_posts()` function, which will guide the scraping process and returns a list of relevant posts.

`get_posts(event_name, location)`

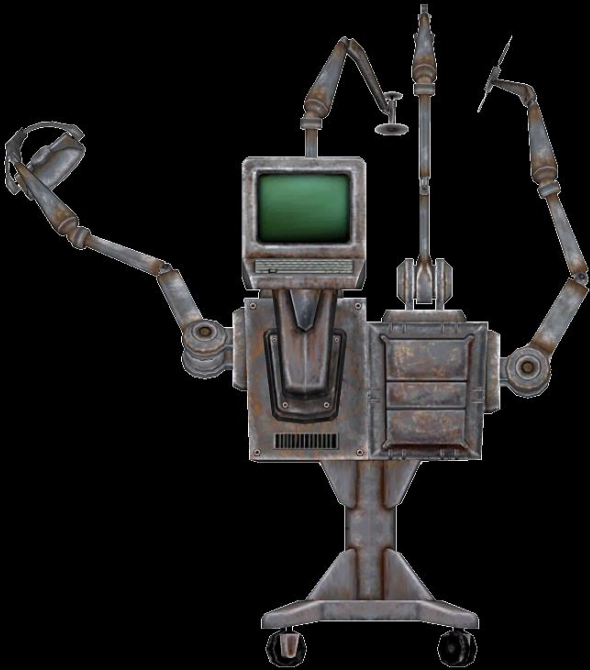
Accesses the Craigslist page of search results for the given event and location. Returns a list of post information tuples, sorted by price.

Ticket Scraper 1.0 documentation »

previous | index

© Copyright 2015, Caitlin. Created using [Sphinx](#) 1.2.3.

AUTODOC



The autodoc feature allows you to use your source code docstrings to generate the documentation.

You can autodoc on a function basis:

```
.. autofunction:: ticket_scraper.get_posts
```

Or on a module basis (and other bases, as well):

```
.. automodule:: ticket_scraper
   :members:
```

But make sure the path to your source code is available in conf.py

AUTODOC

Docstring comments can be in whichever format you choose, but there are some key structures which Sphinx will know how to render. For example:

```
def get_posts(event_name, location):
    ''' The get_posts function accesses the Craigslist page of search
    results for the given event and location. Returns a list of post
    information tuples, sorted by price.
    :param event_name: The keyword with which to search for the event.
    :type event_name: String.
    :param location: The area in which to search for the event.
    :type location: String.
    :returns: List of tuples of post information, sorted by price.
    :raises: None. '''
    ev = '+'.join(event_name.split())
    page = requests.get("http://" + str(location.lower()) + url + ev)
    tree = html.fromstring(page.text)
    posts_pages = get_pages(tree, location)
    post_info = get_post_info(posts_pages)
    return sorted(post_info, key=lambda post: int(post[4]))
```

THE NEW SCRAPER.RST

Ticket Scraping

=====

Ticket scraping is performed by the `ticket_scraper.py` module. Initial calls should be made to the `:func:`get_posts`` function, which will guide the scraping process and returns a list of relevant posts.

```
.. automodule:: ticket_scraper
   :members:
```

In `conf.py`: `sys.path.insert(0, os.path.abspath('.')) + "/app/"`)

THE NEW SCRAPER.RST

Ticket Scraper 1.0 documentation »

previous | modules | index

Previous topic

Technologies

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Ticket Scraping

Ticket scraping is performed by the `ticket_scraper.py` module. Initial calls should be made to the `ticket_scraper.get_posts()` function, which will guide the scraping process and returns a list of relevant posts.

`ticket_scraper.get_post_info(posts_pages)`

The `get_post_info` loops through a list of post pages and scrapes the required information. It returns a list of tuples, each containing information found in the corresponding post.

Parameters:	posts_pages (<i>lxml html tree.</i>) – A list of html trees for each of the search results.
Returns:	List of tuples of post information.
Raises:	None.

`ticket_scraper.get_posts(event_name, location)`

The `get_posts` function accesses the Craigslist page of search results for the given event and location. Returns a list of post information tuples, sorted by price.

Parameters:	<ul style="list-style-type: none">• event_name (<i>String.</i>) – The keyword with which to search for the event.• location (<i>String.</i>) – The metropolitan area in which to search for the event.
Returns:	List of tuples of post information, sorted by ticket price.
Raises:	None.

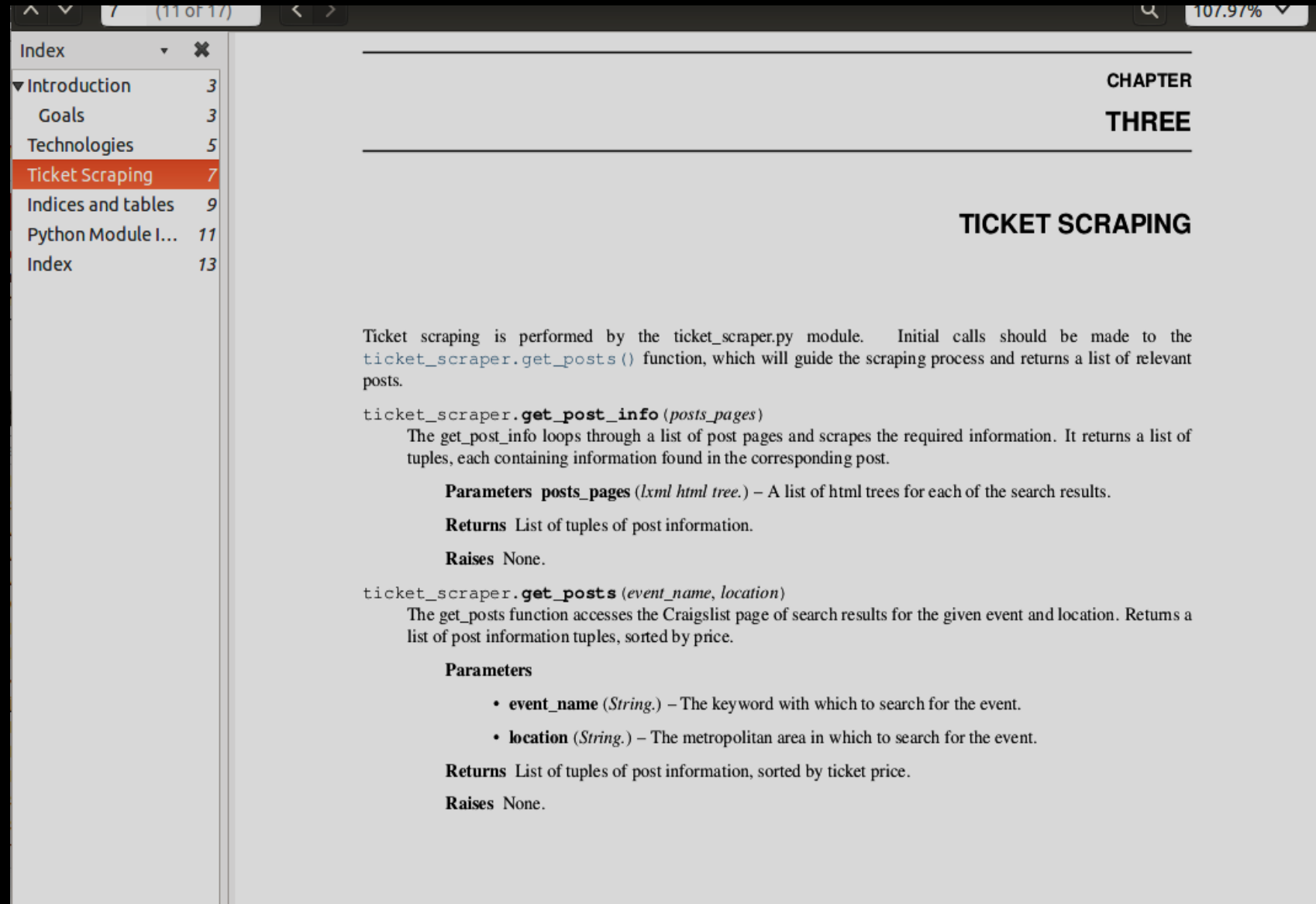
Ticket Scraper 1.0 documentation »

previous | modules | index

© Copyright 2015, Caitlin. Created using [Sphinx](#) 1.2.3.

AND A PDF!

For some light reading...



SPHINX

- Matplotlib has provided a nice Sphinx tutorial, especially for mathematical modules and customizing the documentation look. Check it out [here](#).
- A list of the major projects that use Sphinx can be found [here](#). You can check out their docs and then look for the “source” or “show source” option to see the rst that created the page.
- [readthedocs.org](#) is a docs-hosting website specifically geared towards Sphinx documentation.
- Bitbucket, at least, works well with rst files and you can include your Sphinx docs in your repository without worrying about readability.

LOGGING

Logging is an essential practice for non-trivial applications in which events are recorded for potential diagnostic use in the future.

Logging can be used to record the following kinds of items:

- Errors: any exceptions that are raised can be logged when they are caught.
- Significant events: for example, when an administrator logs into a system.
- Data Handled: for example, a request came into the system with x, y, z parameters.

Logging can provide useful information about the state of the program during a crash or help a developer understand why the program is exhibiting some kind of behavior.

LOGGING

The Python Standard Library actually comes with a standard logging module. It is a particularly good decision to use this module because it can include messages generated from any other package that also uses this library.

As usual, you can use the logging module by using the `import logging` statement.

LOGGING

Here's a simple example of some logging calls.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
```


LOGGING

There are five logging levels which should be used accordingly to reflect the severity of the event being logged.

Level	Use
Debug	For development.
Info	Messages confirming expected behavior.
Warning	Warnings about future issues or benign unexpected behavior.
Error	Something unexpected happened and software is not working properly as a result.
Critical	Something unexpected happened and software is not running anymore as a result.

The `logging.loglevel()` methods are the easiest ways to quickly log a message with a given severity level. You can also use `logging.log(level, msg)`.

LOGGING

Note that the info and debug messages are not logged. This is because the default logging level – the level at which messages must be logged – is warning and higher.

```
>>> import logging
>>> logging.critical("Imminent fatal error!")
CRITICAL:root:Imminent fatal error!
>>> logging.error("Could not perform some function but still running.")
ERROR:root:Could not perform some function but still running.
>>> logging.warning("Disk space low...")
WARNING:root:Disk space low...
>>> logging.info("Everything seems to be working ok.")
>>> logging.debug("Here's some info that might be useful to debug with.")
```

LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module `logtest.py` for example:

```
import logging


logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

LOGGING

It is not typical to log directly in standard output. That might be terrifying to your client. You should at least direct logging statements to a file. Take the module `logtest.py` for example:

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG)
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```



Log debug messages
and higher

LOGGING

After running `logtest.py`, we have a logfile called `example.log` with the following contents:

```
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
INFO:root:Everything seems to be working ok.
DEBUG:root:Here's some info that might be useful to debug with.
```

LOGGING

```
import logging
import sys

for arg in sys.argv:
    if arg[:11] == "--loglevel=":
        loglvl = arg[11:].upper()
    else:
        loglvl = "INFO"
```

```
logging.basicConfig(filename='example.log', level=getattr(logging, loglvl))
logging.critical("Imminent fatal error!")
logging.error("Could not perform some function but still running.")
logging.warning("Disk space low...")
logging.info("Everything seems to be working ok.")
logging.debug("Here's some info that might be useful to debug with.")
```

An even better approach would be to allow the loglevel to be set as an argument to the program itself.

LOGGING

Now, I can specify what the logging level should be without changing the code – this is a more desirable scenario. If I do not set the logging level, it will default to INFO.

```
~$ python logtest.py --loglevel=warning
~$ more example.log
CRITICAL:root:Imminent fatal error!
ERROR:root:Could not perform some function but still running.
WARNING:root:Disk space low...
```

LOGGING

The `logging.basicConfig()` function is the simplest way to do basic global configuration for the logging system.

Any calls to `info()`, `debug()`, etc will call `basicConfig()` if it has not been called already. Any subsequent calls to `basicConfig()` have no effect.

Argument	Effect
filename	File to which logged messages are written.
filemode	Mode to open the filename with. Defaults to 'a' (append).
format	Format string for the messages.
datefmt	Format string for the timestamps.
level	Log level messages and higher.
stream	For StreamHandler objects.

LOGGING

If your application contains multiple modules, you can still share a single log file. Let's say we have a module `driver.py` which uses another module `mymath.py`.

```
import logging
import mymath

def main():
    logging.basicConfig(filename='example.log', level=logging.DEBUG)
    logging.info("Starting.")
    x = mymath.add(2, 3)
    logging.info("Finished with result " + str(x))

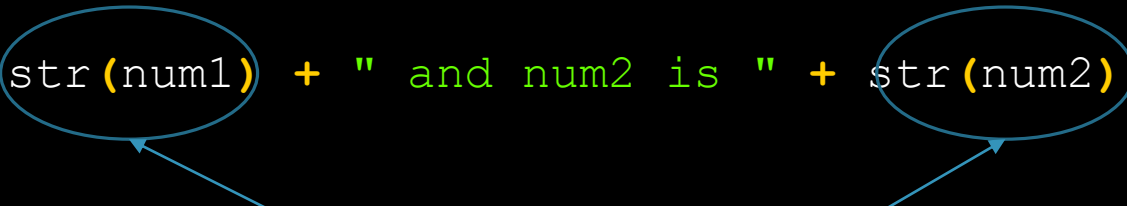
if __name__ == "__main__":
    main()
```

LOGGING

The mymath.py module also logs some message but note that we do not have to reconfigure the logging module. All the messages will log to the same file.

```
import logging

def add(num1, num2):
    logging.debug("num1 is " + str(num1) + " and num2 is " + str(num2))
    logging.info("Adding.")
    return num1 + num2
```

A diagram consisting of two blue ovals. The left oval encircles the expression 'str(num1)' in the logging.debug line of the code. The right oval encircles the expression 'str(num2)' in the same line. Two blue arrows originate from the bottom of these ovals and point towards the text 'Note the logging of variable data here.' located below the code block.

Note the logging of variable data here.

LOGGING

This behavior gives us some insight into the reason why additional calls `basicConfig()` have no effect. The first `basicConfig()` call made during the execution of the application is used to direct logging for all modules involved – even if they have their own `basicConfig()` calls.

```
~$ python driver.py
~$ more example.log
INFO:root:Starting.
DEBUG:root:num1 is 2 and num2 is 3
INFO:root:Adding.
INFO:root:Finished with result 5
```

LOGGING

You can modify the format of the message string as well. Typically, it's useful to include the level, timestamp, and message content.

```
import logging

logging.basicConfig(filename='example.log', level=logging.DEBUG,
                    format='%(asctime)s:%(levelname)s:%(message)s')
logging.info("Some important event just happened.")
```

```
~? python logtest.py
```

```
~? more example.log
```

```
2015-6-11 11:41:42,612:INFO:Some important event just happened.
```

LOGGING

All of the various formatting options can be found [here](#).

This is really just a very basic usage of the logging module, but its definitely enough to log a small project.

Advanced logging features give you a lot more control over when and how things are logged – most notably, you could implement a rotating series of log files rather than one very large logfile which might be difficult to search through.

AUTOMATED TESTING

Obviously, after you write some code, you need to make sure it works. There are pretty much three ways to do this, as pointed out by Ned Batchelder:

- Automatically test your code.
- Manually test your code.
- Just ship it and wait for clients to complain about your code.

The last is...just not a good idea. The second can be downright infeasible for a large project. That leaves us with automated testing.

TESTING

Let's say we have the following module with two simple functions.

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

TESTING

The simplest way to test is to simply pop open the interpreter and try it out.

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

```
>>> import even  
>>> even.even(2)  
True  
>>> even.even(3)  
False  
>>> even.pos_even(2)  
True  
>>> even.pos_even(3)  
False  
>>> even.pos_even(-2)  
False  
>>> even.pos_even(-3)  
False
```


TESTING

This method is time-consuming and not repeatable. We'll have to redo these steps manually anytime we make changes to the code.

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

```
>>> import even  
>>> even.even(2)  
True  
>>> even.even(3)  
False  
>>> even.pos_even(2)  
True  
>>> even.pos_even(3)  
False  
>>> even.pos_even(-2)  
False  
>>> even.pos_even(-3)  
False
```

TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually “check” the correctness of the results.

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

test_even.py

```
import even  
print "even.even(2) = ", even.even(2)  
print "even.even(3) = ", even.even(3)  
print "even.pos_even(2) = ", even.pos_even(2)  
print "even.pos_even(3) = ", even.pos_even(3)  
print "even.pos_even(-2) = ", even.pos_even(-2)  
print "even.pos_even(-3) = ", even.pos_even(-3)
```

TESTING

We can store the testing statements inside of a module and run them anytime we want to test. But this requires us to manually “check” the correctness of the results.

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

```
$ python test_even.py  
even.even(2) = True  
even.even(3) = False  
even.pos_even(2) = True  
even.pos_even(3) = False  
even.pos_even(-2) = False  
even.pos_even(-3) = False
```

TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

test_even.py

```
import even  
assert even.even(2) == True  
assert even.even(3) == False  
assert even.pos_even(2) == True  
assert even.pos_even(3) == False  
assert even.pos_even(-2) == False  
assert even.pos_even(-3) == False
```

TESTING

Let's use assert statements to our advantage. Now, when we test, we only need to see if there were any AssertionError exceptions raised. No output means all tests passed!

even.py

```
def even(num):  
    if abs(num)%2 == 0:  
        return True  
    return False  
  
def pos_even(num):  
    if even(num):  
        if num < 0:  
            return False  
        return True  
    return False
```

```
$ python test_even.py  
$
```

However, one error will halt our testing program entirely so we can only pick up one error at a time. We could nest assertions into try/except statements but now we're starting to do a lot of work for testing.

There must be a better way!

UNITTEST

The unittest module in the Standard Library is a framework for writing unit tests, which specifically test a *small* piece of code in isolation from the rest of the codebase.

Test-driven development is advantageous for the following reasons:

- Encourages modular design.
- Easier to cover every code path.
- The actual process of testing is less time-consuming.

UNITTEST

Here's an example of the simplest usage of unittest.

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

if __name__ == "__main__":
    unittest.main()
```

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False
```

```
$ python test_even.py
```

```
.
-----
Ran 1 test in 0.000s
```

OK

UNITTEST

Here's an example of the simplest usage of unittest.

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

if __name__ == "__main__":
    unittest.main()
```

All tests are defined in methods (which must start with “test_”) of some custom class that derives from unittest.TestCase.

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False
```

```
$ python test_even.py
```

```
.
-----
Ran 1 test in 0.000s
```

OK

UNITTEST BEHIND THE SCENES

By calling `unittest.main()` when we run the module, we are giving control to the `unittest` module. It will create a new instance of `EvenTest` for every test method we have so that they can be performed in isolation.

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        assert even.even(2) == True

if __name__ == "__main__":
    unittest.main()
```



What unittest does:

```
testcase = EvenTest()

try:
    testcase.test_is_two_even()
except AssertionError:
    [record failure]
else:
    [record success]
```

UNITTEST

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

if __name__ == '__main__':
    unittest.main()
```

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num > 0:
            return False
        return True
    return False
```

We've added some new tests along with some new source code. A couple things to notice: our source code has a logical error in it and we're no longer manually asserting. We're using unittest's nice assert methods.

UNITTEST

```
$ python test_even.py
.F
```

```
=====
FAIL: test_two_positive (__main__.EvenTest)
-----
```

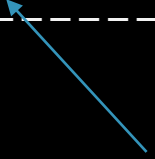
```
Traceback (most recent call last):
```

```
  File "test_even.py", line 8, in test_two_positive
    self.assertTrue(even.pos_even(2))
```

```
AssertionError: False is not true
-----
```

```
Ran 2 tests in 0.000s
```

```
FAILED (failures=1)
```



Extra information given to us by unittest's special assertion method.

UNITTEST

The `unittest` module defines a ton of assertion methods:

- `assertEqual(f, s)`
- `assertNotEqual(f, s)`
- `assertIn(f, s)`
- `assertIs(f, s)`
- `assertGreater(f, s)`
- `assertRaises(exc, f, ...)`
- **etc.**

UNITTEST

test_even.py

```
import unittest
import even

class EvenTest(unittest.TestCase):
    def test_is_two_even(self):
        self.assertTrue(even.even(2))
    def test_two_positive(self):
        self.assertTrue(even.pos_even(2))

if __name__ == '__main__':
    unittest.main()
```

even.py

```
def even(num):
    if abs(num)%2 == 0:
        return True
    return False

def pos_even(num):
    if even(num):
        if num < 0:
            return False
        return True
    return False
```

UNITTEST

```
$ python test_even.py
```

```
..
```

```
-----
```

```
Ran 2 tests in 0.000s
```

```
OK
```

```
$
```

UNITTEST

We incrementally add unit test functions and run them – when they pass, we add more code and develop the unit tests to assert correctness. Do not remove unit tests as you pass them.

Also, practice unit testing as much as you can. Do not wait until it is absolutely necessary.

As with logging, there is a lot more to unit testing that we're not covering here so definitely look up the docs and read articles about unit testing in Python to learn more about the advanced features.

Ned Batchelder also has a relevant unit testing talk from PyCon '14. Check it out [here](#).