
sampledoc Documentation

Release 1.0

John Hunter, Fernando Perez, Michael Droettboom

August 16, 2013

CONTENTS

1	Getting started	3
1.1	Installing your doc directory	3
2	Customizing the look and feel of the site	7
3	Sphinx extensions for embedded plots, math and more	9
3.1	ipython sessions	10
3.2	Using math	10
3.3	Inserting matplotlib plots	11
3.4	Inheritance diagrams	12
3.5	This file	13
4	Ipython Directive	19
4.1	Pseudo-Decorators	23
4.2	Sphinx source for this tutorial	23
5	Sphinx cheat sheet	29
5.1	Formatting text	29
5.2	Making a list	29
5.3	Making a table	30
5.4	Making links	30
5.5	This file	30
6	Emacs ReST support	33
6.1	Emacs helpers	33
7	Indices and tables	35

This is a tutorial introduction to quickly get you up and running with your own sphinx documentation system. We'll cover installing sphinx, customizing the look and feel, using custom extensions for embedding plots, inheritance diagrams, syntax highlighted ipython sessions and more. If you follow along the tutorial, you'll start with nothing and end up with this site – it's the bootstrapping documentation tutorial that writes itself!

The source code for this tutorial lives in mpl git (see [Fetching the data](#)) and you can grab a hardcopy of the sampledoc PDF

GETTING STARTED

1.1 Installing your doc directory

You may already have sphinx [sphinx](#) installed – you can check by doing:

```
python -c 'import sphinx'
```

If that fails grab the latest version of and install it with:

```
> sudo easy_install -U Sphinx
```

Now you are ready to build a template for your docs, using sphinx-quickstart:

```
> sphinx-quickstart
```

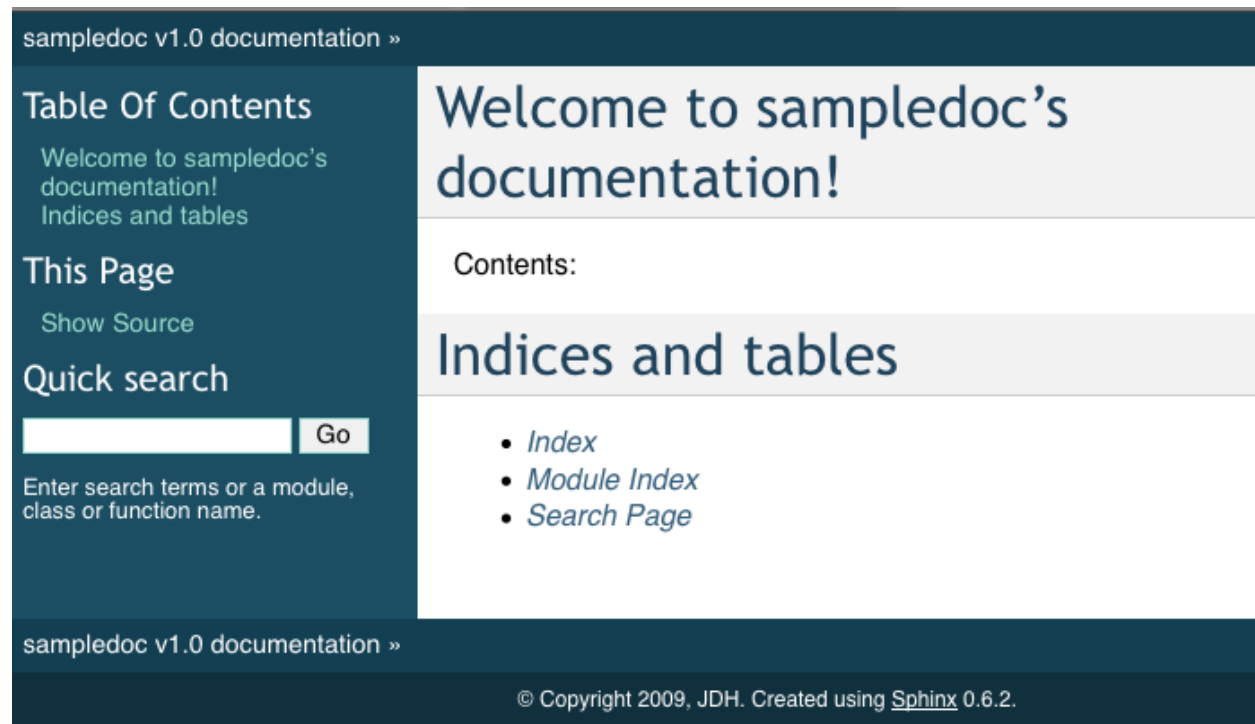
accepting most of the defaults. I choose “sampledoc” as the name of my project. cd into your new directory and check the contents:

```
home:~/tmp/sampledoc> ls
Makefile      _static      conf.py
_build        _templates   index.rst
```

The index.rst is the master ReST for your project, but before adding anything, let’s see if we can build some html:

```
make html
```

If you now point your browser to `_build/html/index.html`, you should see a basic sphinx site.



sampledoc v1.0 documentation »

Table Of Contents
Welcome to sampledoc's documentation!
Indices and tables

This Page
Show Source

Quick search

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

Welcome to sampledoc's documentation!

Contents:

Indices and tables

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

sampledoc v1.0 documentation »

© Copyright 2009, JDH. Created using [Sphinx](#) 0.6.2.

1.1.1 Fetching the data

Now we will start to customize our docs. Grab a couple of files from the [web site](#) or git. You will need `getting_started.rst` and `_static/basic_screenshot.png`. All of the files live in the “completed” version of this tutorial, but since this is a tutorial, we’ll just grab them one at a time, so you can learn what needs to be changed where. Since we have more files to come, I’m going to grab the whole git directory and just copy the files I need over for now. First, I’ll cd up back into the directory containing my project, check out the “finished” product from git, and then copy in just the files I need into my `sampledoc` directory:

```
home:~/tmp/sampledoc> pwd
/Users/jdhunter/tmp/sampledoc
home:~/tmp/sampledoc> cd ..
home:~/tmp> git clone https://github.com/matplotlib/sampledoc.git tutorial
Cloning into 'tutorial'...
remote: Counting objects: 87, done.
remote: Compressing objects: 100% (43/43), done.
remote: Total 87 (delta 45), reused 83 (delta 41)
Unpacking objects: 100% (87/87), done.
Checking connectivity... done
home:~/tmp> cp tutorial/getting_started.rst sampledoc/
home:~/tmp> cp tutorial/_static/basic_screenshot.png sampledoc/_static/
```

The last step is to modify `index.rst` to include the `getting_started.rst` file (be careful with the indentation, the “g” in “getting_started” should line up with the “.” in `:maxdepth:`):

```
Contents:

.. toctree::
   :maxdepth: 2

   getting_started.rst
```


and then rebuild the docs:

```
cd sampledoc
make html
```

When you reload the page by refreshing your browser pointing to `_build/html/index.html`, you should see a link to the “Getting Started” docs, and in there this page with the screenshot. *Voila!*

Note we used the image directive to include to the screenshot above with:

```
.. image:: _static/basic_screenshot.png
```

Next we’ll customize the look and feel of our site to give it a logo, some custom css, and update the navigation panels to look more like the [sphinx](#) site itself – see *Customizing the look and feel of the site*.

CUSTOMIZING THE LOOK AND FEEL OF THE SITE

The [sphinx](#) site itself looks better than the sites created with the default css, so here we'll invoke T. S. Eliot's maxim "Talent imitates, but genius steals" and grab their css and part of their layout. As before, you can either get the required files `_static/default.css`, `_templates/layout.html` and `_static/logo.png` from the website or git (see [Fetching the data](#)). Since I did a git clone before, I will just copy the stuff I need from there:

```
home:~/tmp/sampledok> cp ../sampledoc_tut/_static/default.css _static/
home:~/tmp/sampledok> cp ../sampledoc_tut/_templates/layout.html _templates/
home:~/tmp/sampledok> cp ../sampledoc_tut/_static/logo.png _static/
home:~/tmp/sampledok> ls _static/ _templates/
_static/:
basic_screenshot.png      default.css               logo.png

_templates/:
layout.html
```

Sphinx will automatically pick up the css and layout html files since we put them in the default places with the default names, but we have to manually include the logo in our `layout.html`. Let's take a look at the layout file: the first part puts a horizontal navigation bar at the top of our page, like you see on the [sphinx](#) and [matplotlib](#) sites, the second part includes a logo that when we click on it will take us *home* and the last part moves the vertical navigation panels to the right side of the page:

```
{% extends "!layout.html" %}

{% block rootrellink %}
    <li><a href="{{ pathto('index') }}">home</a>|&nbsp;</li>
    <li><a href="{{ pathto('search') }}">search</a>|&nbsp;</li>
    <li><a href="{{ pathto('contents') }}">documentation </a> &raquo;</li>
{% endblock %}

{% block relbar1 %}

<div style="background-color: white; text-align: left; padding: 10px 10px 15px 15px">
<a href="{{ pathto('index') }}"></a>
</div>
{{ super() }}
{% endblock %}

{# put the sidebar before the body #}
```

```
{% block sidebar1 %}{{ sidebar() }}{% endblock %}
{% block sidebar2 %}{% endblock %}
```

Once you rebuild the site with a `make html` and reload the page in your browser, you should see a fancier site that looks like this

The screenshot shows a web page for 'sampledoc tutorial'. At the top, the word 'sampledoc' is written in a large, stylized font with a colorful, textured fill. Below it is a horizontal navigation bar with links: 'home', 'search', 'documentation', 'next', and 'index'. The main content area is titled 'sampledoc tutorial' and contains a 'Contents:' section with a bulleted list of links: 'Getting started', 'Installing your doc directory', 'Customizing the look and feel of the site', 'Sphinx extensions for embedded plots, math and more', 'ipython sessions', 'Using math', 'Inserting matplotlib plots', 'Inheritance diagrams', and 'This file'. To the right of the main content is a sidebar with a 'Table Of Contents' section containing links to 'sampledoc tutorial' and 'Indices and tables', a 'Next topic' section, a 'Getting started' section, a 'This Page' section, a 'Show Source' section, and a 'Quick search' section.

sampledoc

[home](#) | [search](#) | [documentation](#) » [next](#) | [index](#)

sampledoc tutorial

Contents:

- [Getting started](#)
 - [Installing your doc directory](#)
- [Customizing the look and feel of the site](#)
- [Sphinx extensions for embedded plots, math and more](#)
 - [ipython sessions](#)
 - [Using math](#)
 - [Inserting matplotlib plots](#)
 - [Inheritance diagrams](#)
 - [This file](#)

Table Of Contents

[sampledoc tutorial](#)

[Indices and tables](#)

Next topic

[Getting started](#)

This Page

[Show Source](#)

Quick search

SPHINX EXTENSIONS FOR EMBEDDED PLOTS, MATH AND MORE

Sphinx is written in python, and supports the ability to write custom extensions. We've written a few for the matplotlib documentation, some of which are part of matplotlib itself in the `matplotlib.sphinxext` module, some of which are included only in the `sphinx doc` directory, and there are other extensions written by other groups, eg `numpy` and `ipython`. We're collecting these in this tutorial and showing you how to install and use them for your own project. First let's grab the python extension files from the `sphinxext` directory from git (see [Fetching the data](#)), and install them in our `sampledoc` project `sphinxext` directory:

```
home:~/tmp/sampledoc> mkdir sphinxext
home:~/tmp/sampledoc> cp ../sampledoc_tut/sphinxext/*.py sphinxext/
home:~/tmp/sampledoc> ls sphinxext/
apigen.py      docscrape_sphinx.py      ipython_console_highlighting.py
docscrape.py   inheritance_diagram.py    numpydoc.py
```

In addition to the builtin matplotlib extensions for embedding pyplot plots and rendering math with matplotlib's native math engine, we also have extensions for syntax highlighting ipython sessions, making inheritance diagrams, and more.

We need to inform sphinx of our new extensions in the `conf.py` file by adding the following. First we tell it where to find the extensions:

```
# If your extensions are in another directory, add it here. If the
# directory is relative to the documentation root, use
# os.path.abspath to make it absolute, like shown here.
sys.path.append(os.path.abspath('sphinxext'))
```

And then we tell it what extensions to load:

```
# Add any Sphinx extension module names here, as strings. They can
# be extensions coming with Sphinx (named 'sphinx.ext.*') or your
# custom ones.
extensions = [
    'matplotlib.sphinxext.mathmpl',
    'matplotlib.sphinxext.only_directives',
    'matplotlib.sphinxext.plot_directive',
    'matplotlib.sphinxext.ipython_directive',
    'sphinx.ext.autodoc',
    'sphinx.ext.doctest',
    'ipython_console_highlighting',
    'inheritance_diagram',
    'numpydoc']
```

Now let's look at some of these in action. You can see the literal source for this file at [extensions-literal](#).

3.1 ipython sessions

Michael Droettboom contributed a sphinx extension which does `pygments` syntax highlighting on `ipython` sessions. Just use `ipython` as the language in the `sourcecode` directive:

```
.. sourcecode:: ipython

    In [69]: lines = plot([1,2,3])

    In [70]: setp(lines)
             alpha: float
             animated: [True | False]
             antialiased or aa: [True | False]
             ...snip
```

and you will get the syntax highlighted output below.

```
In [69]: lines = plot([1,2,3])

In [70]: setp(lines)
alpha: float
animated: [True | False]
antialiased or aa: [True | False]
...snip
```

This support is included in this template, but will also be included in a future version of `Pygments` by default.

3.2 Using math

In sphinx you can include inline math $x \leftarrow y$ $x \forall y$ $x - y$ or display math

$$W_{\delta_1 \rho_1 \sigma_2}^{3\beta} = U_{\delta_1 \rho_1}^{3\beta} + \frac{1}{8\pi 2} \int_{\alpha_2}^{\alpha_2} d\alpha'_2 \left[\frac{U_{\delta_1 \rho_1}^{2\beta} - \alpha'_2 U_{\rho_1 \sigma_2}^{1\beta}}{U_{\rho_1 \sigma_2}^{0\beta}} \right] \quad (3.1)$$

To include math in your document, just use the `math` directive; here is a simpler equation:

```
.. math::

    W^{\{3\beta\}}_{\{\delta_1 \rho_1 \sigma_2\}} \approx U^{\{3\beta\}}_{\{\delta_1 \rho_1\}}
```

which is rendered as

$$W_{\delta_1 \rho_1 \sigma_2}^{3\beta} \approx U_{\delta_1 \rho_1}^{3\beta} \quad (3.2)$$

This documentation framework includes a Sphinx extension, `sphinxext/mathmpl.py`, that uses `matplotlib` to render math equations when generating HTML, and `LaTeX` itself when generating a PDF. This can be useful on systems that have `matplotlib`, but not `LaTeX`, installed. To use it, add `mathmpl` to the list of extensions in `conf.py`.

Current SVN versions of Sphinx now include built-in support for math. There are two flavors:

- `pngmath`: uses `dvipng` to render the equation
- `jsmath`: renders the math in the browser using Javascript

To use these extensions instead, add `sphinx.ext.pngmath` or `sphinx.ext.jsmath` to the list of extensions in `conf.py`.

All three of these options for math are designed to behave in the same way.

See the `matplotlib` [mathtext guide](#) for lots more information on writing mathematical expressions in `matplotlib`.

3.3 Inserting matplotlib plots

Inserting automatically-generated plots is easy. Simply put the script to generate the plot in the `pyplots` directory, and refer to it using the `plot` directive. First make a `pyplots` directory at the top level of your project (next to `:conf.py`) and copy the `ellipses.py` file into it:

```
home:~/tmp/sampledoc> mkdir pyplots
home:~/tmp/sampledoc> cp ../sampledoc_tut/pyplots/ellipses.py pyplots/
```

You can refer to this file in your sphinx documentation; by default it will just inline the plot with links to the source and PF and high resolution PNGS. To also include the source code for the plot in the document, pass the `include-source` parameter:

```
.. plot:: pyplots/ellipses.py
   :include-source:
```

In the HTML version of the document, the plot includes links to the original source code, a high-resolution PNG and a PDF. In the PDF version of the document, the plot is included as a scalable PDF.

```
from pylab import *
from matplotlib.patches import Ellipse

delta = 45.0 # degrees

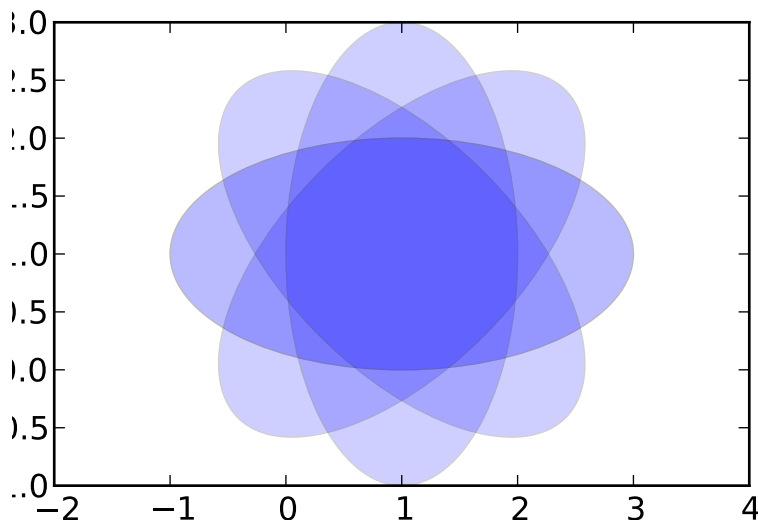
angles = arange(0, 360+delta, delta)
ells = [Ellipse((1, 1), 4, 2, a) for a in angles]

a = subplot(111, aspect='equal')

for e in ells:
    e.set_clip_box(a.bbox)
    e.set_alpha(0.1)
    a.add_artist(e)

xlim(-2, 4)
ylim(-1, 3)

show()
```

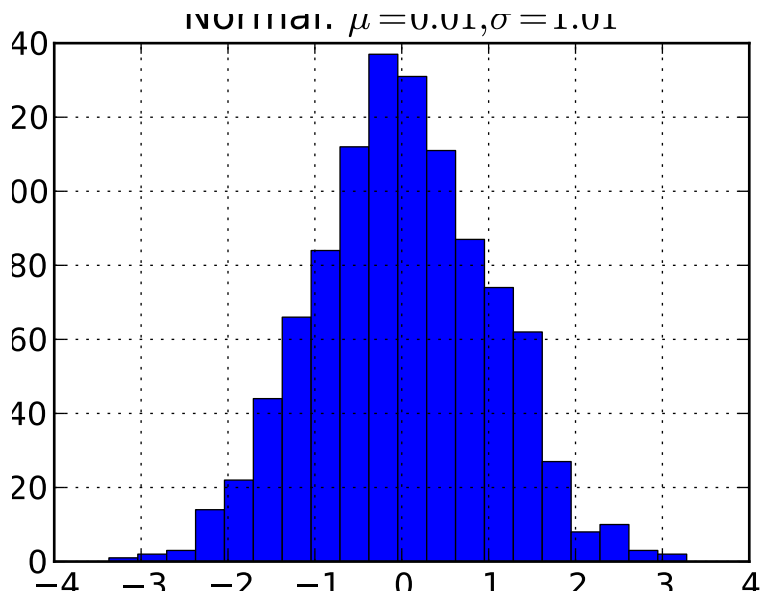


You can also inline code for plots directly, and the code will be executed at documentation build time and the figure inserted into your docs; the following code:

```
.. plot::

    import matplotlib.pyplot as plt
    import numpy as np
    x = np.random.randn(1000)
    plt.hist( x, 20)
    plt.grid()
    plt.title(r'Normal:  $\mu=0.01$ ,  $\sigma=1.01$ '%(x.mean(), x.std()))
    plt.show()
```

produces this output:



See the matplotlib [pyplot tutorial](#) and the [gallery](#) for lots of examples of matplotlib plots.

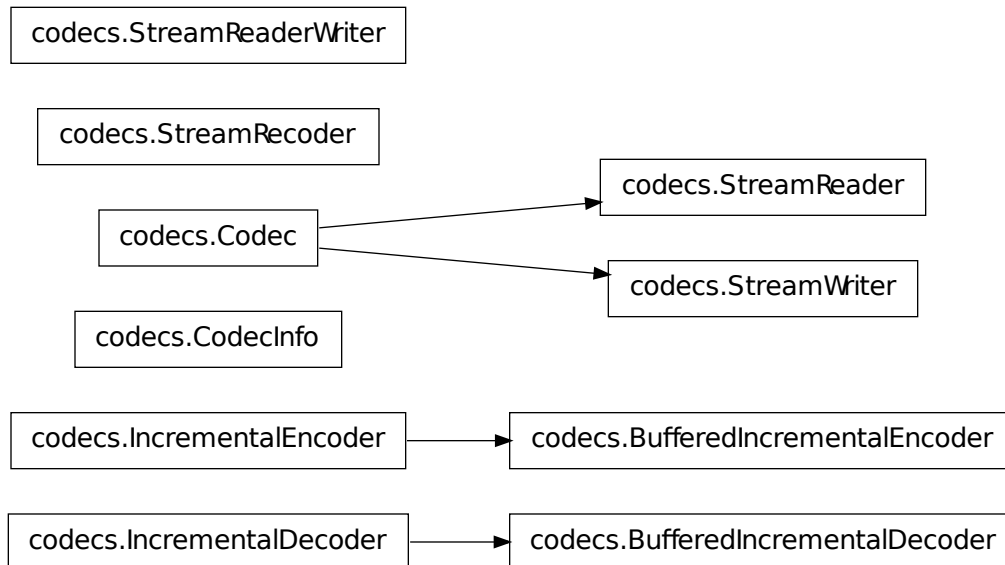
3.4 Inheritance diagrams

Inheritance diagrams can be inserted directly into the document by providing a list of class or module names to the `inheritance-diagram` directive.

For example:

```
.. inheritance-diagram:: codecs
```

produces:



See the [Ipython Directive](#) for a tutorial on embedding stateful, matplotlib aware ipython sessions into your rest docs with multiline and doctest support.

3.5 This file

```
.. _extensions:
```

```
*****
Sphinx extensions for embedded plots, math and more
*****
```

Sphinx is written in python, and supports the ability to write custom extensions. We've written a few for the matplotlib documentation, some of which are part of matplotlib itself in the `matplotlib.sphinxext` module, some of which are included only in the sphinx doc directory, and there are other extensions written by other groups, eg numpy and ipython. We're collecting these in this tutorial and showing you how to install and use them for your own project. First let's grab the python extension files from the `:file:'sphinxext'` directory from git (see `:ref:'fetching-the-data'`), and install them in our `:file:'sampledoc'` project `:file:'sphinxext'` directory::

```
home:~/tmp/sampledoc> mkdir sphinxext
home:~/tmp/sampledoc> cp ../sampledoc_tut/sphinxext/*.py sphinxext/
home:~/tmp/sampledoc> ls sphinxext/
apigen.py      docscrape_sphinx.py      ipython_console_highlighting.py
docscrape.py   inheritance_diagram.py    numpydoc.py
```

In addition to the builtin matplotlib extensions for embedding pyplot

plots and rendering math with matplotlib's native math engine, we also have extensions for syntax highlighting ipython sessions, making inheritance diagrams, and more.

We need to inform sphinx of our new extensions in the `:file:'conf.py'` file by adding the following. First we tell it where to find the extensions::

```
# If your extensions are in another directory, add it here. If the
# directory is relative to the documentation root, use
# os.path.abspath to make it absolute, like shown here.
sys.path.append(os.path.abspath('sphinxext'))
```

And then we tell it what extensions to load::

```
# Add any Sphinx extension module names here, as strings. They can
# be extensions coming with Sphinx (named 'sphinx.ext.*') or your
# custom ones.
extensions = [
    'matplotlib.sphinxext.mathmpl',
    'matplotlib.sphinxext.only_directives',
    'matplotlib.sphinxext.plot_directive',
    'matplotlib.sphinxext.ipython_directive',
    'sphinx.ext.autodoc',
    'sphinx.ext.doctest',
    'ipython_console_highlighting',
    'inheritance_diagram',
    'numpydoc']
```

Now let's look at some of these in action. You can see the literal source for this file at `:ref:'extensions-literal'`.

`.. _ipython-highlighting:`

ipython sessions
=====

Michael Droettboom contributed a sphinx extension which does `'pygments <http://pygments.org>'` syntax highlighting on `'ipython <http://ipython.scipy.org>'` sessions. Just use ipython as the language in the `'sourcecode'` directive::

```
.. sourcecode:: ipython

    In [69]: lines = plot([1,2,3])

    In [70]: setp(lines)
             alpha: float
             animated: [True | False]
             antialiased or aa: [True | False]
             ...snip
```

and you will get the syntax highlighted output below.

```
.. sourcecode:: ipython

    In [69]: lines = plot([1,2,3])
```

```
In [70]: setp(lines)
        alpha: float
        animated: [True | False]
        antialiased or aa: [True | False]
        ...snip
```

This support is included in this template, but will also be included in a future version of Pygments by default.

```
.. _using-math:
```

```
Using math
=====
```

In sphinx you can include inline math `:math:`x\rightarrow y`` for all `y\ x-y`` or display math

```
.. math::
```

$$W^{3\beta}_{\{\delta_1 \rho_1 \sigma_2\}} = U^{3\beta}_{\{\delta_1 \rho_1\}} + \frac{1}{8\pi^2} \int^{\alpha}$$

To include math in your document, just use the math directive; here is a simpler equation::

```
.. math::
```

$$W^{3\beta}_{\{\delta_1 \rho_1 \sigma_2\}} \approx U^{3\beta}_{\{\delta_1 \rho_1\}}$$

which is rendered as

```
.. math::
```

$$W^{3\beta}_{\{\delta_1 \rho_1 \sigma_2\}} \approx U^{3\beta}_{\{\delta_1 \rho_1\}}$$

This documentation framework includes a Sphinx extension, `:file:`sphinxext/mathmpl.py``, that uses matplotlib to render math equations when generating HTML, and LaTeX itself when generating a PDF. This can be useful on systems that have matplotlib, but not LaTeX, installed. To use it, add ```mathmpl``` to the list of extensions in `:file:`conf.py``.

Current SVN versions of Sphinx now include built-in support for math. There are two flavors:

- `pngmath`: uses dvipng to render the equation
- `jsmath`: renders the math in the browser using Javascript

To use these extensions instead, add ```sphinx.ext.pngmath``` or ```sphinx.ext.jsmath``` to the list of extensions in `:file:`conf.py``.

All three of these options for math are designed to behave in the same way.

See the matplotlib `mathtext` guide <http://matplotlib.sourceforge.net/users/mathtext.html> for lots more information on writing mathematical expressions in matplotlib.

```
.. _pyplots:
```

```
Inserting matplotlib plots
=====
```

Inserting automatically-generated plots is easy. Simply put the script to generate the plot in the `:file:'pyplots'` directory, and refer to it using the `''plot''` directive. First make a `:file:'pyplots'` directory at the top level of your project (next to `:'conf.py'`) and copy the `:file:'ellipses.py'` file into it::

```
home:~/tmp/sampledoc> mkdir pyplots
home:~/tmp/sampledoc> cp ../sampledoc_tut/pyplots/ellipses.py pyplots/
```

You can refer to this file in your sphinx documentation; by default it will just inline the plot with links to the source and PF and high resolution PNGS. To also include the source code for the plot in the document, pass the `''include-source''` parameter::

```
.. plot:: pyplots/ellipses.py
   :include-source:
```

In the HTML version of the document, the plot includes links to the original source code, a high-resolution PNG and a PDF. In the PDF version of the document, the plot is included as a scalable PDF.

```
.. plot:: pyplots/ellipses.py
   :include-source:
```

You can also inline code for plots directly, and the code will be executed at documentation build time and the figure inserted into your docs; the following code::

```
.. plot::

    import matplotlib.pyplot as plt
    import numpy as np
    x = np.random.randn(1000)
    plt.hist( x, 20)
    plt.grid()
    plt.title(r'Normal:  $\mu=0.2f$ ,  $\sigma=0.2f$ '%(x.mean(), x.std()))
    plt.show()
```

produces this output:

```
.. plot::

    import matplotlib.pyplot as plt
    import numpy as np
    x = np.random.randn(1000)
    plt.hist( x, 20)
    plt.grid()
    plt.title(r'Normal:  $\mu=0.2f$ ,  $\sigma=0.2f$ '%(x.mean(), x.std()))
    plt.show()
```

See the matplotlib `'pyplot tutorial`
<http://matplotlib.sourceforge.net/users/pyplot_tutorial.html>' and
the `'gallery` <<http://matplotlib.sourceforge.net/gallery.html>>' for
lots of examples of matplotlib plots.

Inheritance diagrams
=====

Inheritance diagrams can be inserted directly into the document by
providing a list of class or module names to the
`'inheritance-diagram'` directive.

For example::

```
.. inheritance-diagram:: codecs
```

produces:

```
.. inheritance-diagram:: codecs
```

```
.. _extensions-literal:
```

See the `:ref:'ipython_directive'` for a tutorial on embedding stateful,
matplotlib aware ipython sessions into your rest docs with multiline
and doctest support.

This file
=====

```
.. literalinclude:: extensions.rst
```


IPYTHON DIRECTIVE

The ipython directive is a stateful ipython shell for embedding in sphinx documents. It knows about standard ipython prompts, and extracts the input and output lines. These prompts will be renumbered starting at 1. The inputs will be fed to an embedded ipython interpreter and the outputs from that interpreter will be inserted as well. For example, code blocks like the following:

```
.. ipython::

    In [136]: x = 2

    In [137]: x**3
    Out[137]: 8
```

will be rendered as

```
In [1]: x = 2

In [2]: x**3
Out[2]: 8
```

Note: This tutorial should be read side-by-side with the Sphinx source for this document (see [Sphinx source for this tutorial](#)) because otherwise you will see only the rendered output and not the code that generated it. Excepting the example above, we will not in general be showing the literal rest in this document that generates the rendered output.

The state from previous sessions is stored, and standard error is trapped. At doc build time, ipython’s output and std err will be inserted, and prompts will be renumbered. So the prompt below should be renumbered in the rendered docs, and pick up where the block above left off.

```
In [3]: z = x*3    # x is recalled from previous block

In [4]: z
Out[4]: 6

In [5]: print z
6

In [6]: q = z[]    # this is a syntax error -- we trap ipy exceptions
File "<ipython-input-6-9e4a6c3ff9f7>", line 1
    q = z[]    # this is a syntax error -- we trap ipy exceptions
        ^
SyntaxError: invalid syntax
```

The embedded interpreter supports some limited markup. For example, you can put comments in your ipython sessions, which are reported verbatim. There are some handy “pseudo-decorators” that let you doctest the output. The

inputs are fed to an embedded ipython session and the outputs from the ipython session are inserted into your doc. If the output in your doc and in the ipython session don't match on a doctest assertion, an error will be

```
In [7]: x = 'hello world'

# this will raise an error if the ipython output is different
In [8]: x.upper()
Out[8]: 'HELLO WORLD'

# some readline features cannot be supported, so we allow
# "verbatim" blocks, which are dumped in verbatim except prompts
# are continuously numbered
In [9]: x.st<TAB>
x.startswith x.strip
```

Multi-line input is supported.

```
In [10]: url = 'http://ichart.finance.yahoo.com/table.csv?s=CROX\
.....: &d=9&e=22&f=2009&g=d&a=1&br=8&c=2006&ignore=.csv'
.....:

In [11]: print url.split('&')
['http://ichart.finance.yahoo.com/table.csv?s=CROX ', 'd=9', 'e=22', 'f=2009', 'g=d', 'a=1', 'br=8',
'f=2009', 'g=d', 'a=1', 'b=8', 'c=2006', 'ignore=.csv']

In [60]: import urllib
```

You can do doctesting on multi-line output as well. Just be careful when using non-deterministic inputs like random numbers in the ipython directive, because your inputs are ruin through a live interpreter, so if you are doctesting random output you will get an error. Here we “seed” the random number generator for deterministic output, and we suppress the seed line so it doesn't show up in the rendered output

```
In [12]: import numpy.random

In [14]: numpy.random.rand(10,2)
Out[14]:
array([[ 0.64524308,  0.59943846],
       [ 0.47102322,  0.8715456 ],
       [ 0.29370834,  0.74776844],
       [ 0.99539577,  0.1313423 ],
       [ 0.16250302,  0.21103583],
       [ 0.81626524,  0.1312433 ],
       [ 0.67338089,  0.72302393],
       [ 0.7566368 ,  0.07033696],
       [ 0.22591016,  0.77731835],
       [ 0.0072729 ,  0.34273127]])
```

Another demonstration of multi-line input and output

```
In [15]: print x
hello world

In [16]: for i in range(10):
.....:     print i
.....:
.....:
0
1
2
```


3
4
5
6
7
8
9

Most of the “pseudo-decorators” can be used as options to ipython mode. For example, to setup matplotlib pylab but suppress the output, you can do. When using the matplotlib `use` directive, it should occur before any import of pylab. This will not show up in the rendered docs, but the commands will be executed in the embedded interpreter and subsequent line numbers will be incremented to reflect the inputs:

```
.. ipython::
   :suppress:

In [144]: from pylab import *

In [145]: ion()
```

Likewise, you can set `:doctest:` or `:verbatim:` to apply these settings to the entire block. For example,

```
In [19]: cd mpl/examples/
/home/jdhunter/mpl/examples
```

```
In [20]: pwd
```

```
In [21]: cd mpl/examples/<TAB>
mpl/examples/animation/      mpl/examples/misc/
mpl/examples/api/            mpl/examples/mplot3d/
mpl/examples/axes_grid/      mpl/examples/pylab_examples/
mpl/examples/event_handling/  mpl/examples/widgets
```

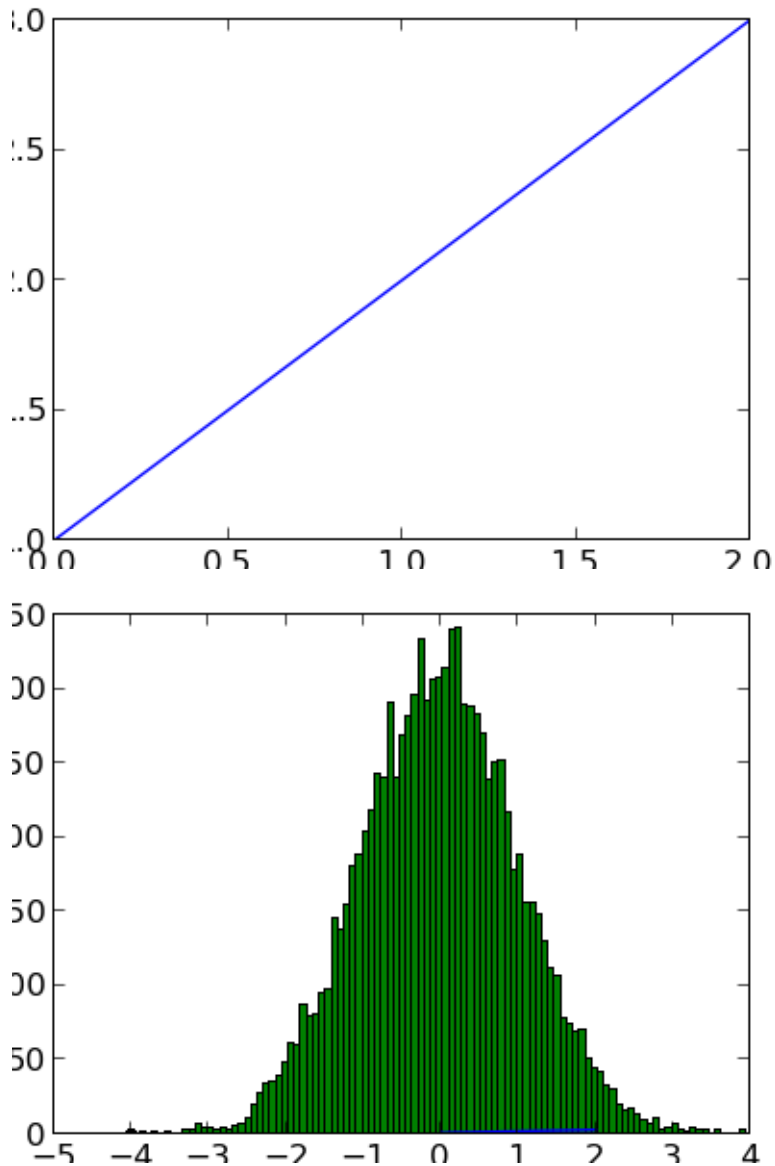
```
In [22]: cd mpl/examples/widgets/
/home/jdhunter/mpl/examples/widgets
```

```
In [23]: !wc *
 2    12    77 README.txt
40    97   884 buttons.py
26    90   712 check_buttons.py
19    52   416 cursor.py
180  404  4882 menu.py
16    45   337 multicursor.py
36   106   916 radio_buttons.py
48   226  2082 rectangle_selector.py
43   118  1063 slider_demo.py
40   124  1088 span_selector.py
450 1274 12457 total
```

You can create one or more pyplot plots and insert them with the `@savefig` decorator.

```
In [24]: plot([1,2,3]);

# use a semicolon to suppress the output
In [25]: hist(np.random.randn(10000), 100);
```

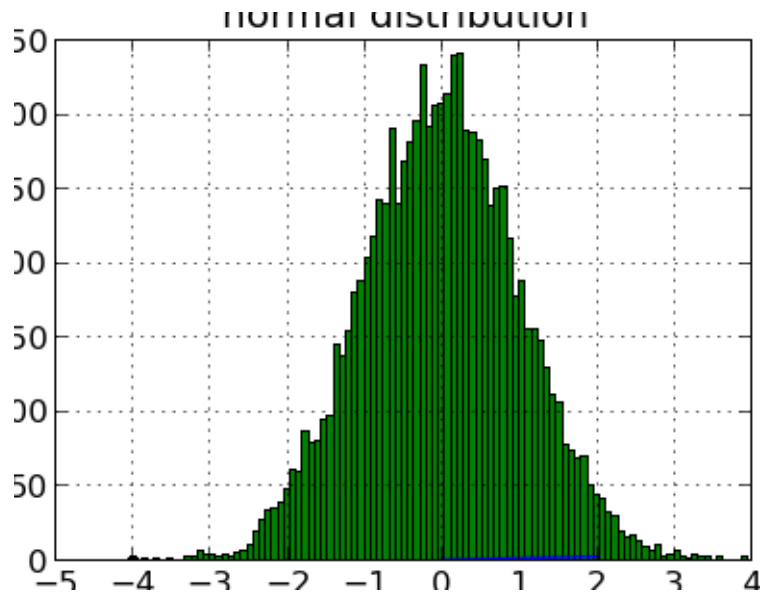


In a subsequent session, we can update the current figure with some text, and then resave

```
In [26]: ylabel('number')
Out[26]: <matplotlib.text.Text at 0x358e110>

In [27]: title('normal distribution')
Out[27]: <matplotlib.text.Text at 0x359b090>

In [28]: grid(True)
```



4.1 Pseudo-Decorators

Here are the supported decorators, and any optional arguments they take. Some of the decorators can be used as options to the entire block (eg `verbatim` and `suppress`), and some only apply to the line just below them (eg `savefig`).

`@suppress`

execute the ipython input block, but suppress the input and output block from the rendered output. Also, can be applied to the entire `.. ipython` block as a directive option with `:suppress:`.

`@verbatim`

insert the input and output block in verbatim, but auto-increment the line numbers. Internally, the interpreter will be fed an empty string, so it is a no-op that keeps line numbering consistent. Also, can be applied to the entire `.. ipython` block as a directive option with `:verbatim:`.

`@savefig` OUTFILE [IMAGE_OPTIONS]

save the figure to the static directory and insert it into the document, possibly binding it into a minipage and/or putting code/figure label/references to associate the code and the figure. Takes args to pass to the image directive (*scale*, *width*, etc can be kwargs); see [image options](#) for details.

`@doctest`

Compare the pasted in output in the ipython block with the output generated at doc build time, and raise errors if they don't match. Also, can be applied to the entire `.. ipython` block as a directive option with `:doctest:`.

4.2 Sphinx source for this tutorial

```
.. _ipython_directive:
```

```
=====
Ipython Directive
```

=====

The `ipython` directive is a stateful `ipython` shell for embedding in sphinx documents. It knows about standard `ipython` prompts, and extracts the input and output lines. These prompts will be renumbered starting at `1`. The inputs will be fed to an embedded `ipython` interpreter and the outputs from that interpreter will be inserted as well. For example, code blocks like the following::

```
.. ipython::

    In [136]: x = 2

    In [137]: x**3
    Out[137]: 8
```

will be rendered as

```
.. ipython::

    In [136]: x = 2

    In [137]: x**3
    Out[137]: 8
```

```
.. note::

    This tutorial should be read side-by-side with the Sphinc source
    for this document (see :ref:`ipython_literal`) because otherwise
    you will see only the rendered output and not the code that
    generated it. Excepting the example above, we will not in general
    be showing the liuteral rest in this document that generates the
    rendered output.
```

The state from previous sessions is stored, and standard error is trapped. At doc build time, `ipython`'s output and `std err` will be inserted, and prompts will be renumbered. So the prompt below should be renumbered in the rendered docs, and pick up where the block above left off.

```
.. ipython::

    In [138]: z = x*3    # x is recalled from previous block

    In [139]: z
    Out[139]: 6

    In [140]: print z
    -----> print(z)
    6

    In [141]: q = z[]    # this is a syntax error -- we trap ipy exceptions
    -----
    File "<ipython console>", line 1
      q = z[]    # this is a syntax error -- we trap ipy exceptions
        ^
    SyntaxError: invalid syntax
```

The embedded interpreter supports some limited markup. For example, you can put comments in your ipython sessions, which are reported verbatim. There are some handy "pseudo-decorators" that let you doctest the output. The inputs are fed to an embedded ipython session and the outputs from the ipython session are inserted into your doc. If the output in your doc and in the ipython session don't match on a doctest assertion, an error will be

```
.. ipython::
```

```
In [1]: x = 'hello world'

# this will raise an error if the ipython output is different
@doctest
In [2]: x.upper()
Out[2]: 'HELLO WORLD'

# some readline features cannot be supported, so we allow
# "verbatim" blocks, which are dumped in verbatim except prompts
# are continuously numbered
@verbatim
In [3]: x.st<TAB>
x.startswith x.strip
```

Multi-line input is supported.

```
.. ipython::
```

```
In [130]: url = 'http://ichart.finance.yahoo.com/table.csv?s=CROX\
.....: &d=9&e=22&f=2009&g=d&a=1&br=8&c=2006&ignore=.csv'

In [131]: print url.split('&')
-----> print(url.split('&'))
['http://ichart.finance.yahoo.com/table.csv?s=CROX', 'd=9', 'e=22',
'f=2009', 'g=d', 'a=1', 'b=8', 'c=2006', 'ignore=.csv']

In [60]: import urllib
```

You can do doctesting on multi-line output as well. Just be careful when using non-deterministic inputs like random numbers in the ipython directive, because your inputs are ruin through a live interpreter, so if you are doctesting random output you will get an error. Here we "seed" the random number generator for deterministic output, and we suppress the seed line so it doesn't show up in the rendered output

```
.. ipython::
```

```
In [133]: import numpy.random

@suppress
In [134]: numpy.random.seed(2358)

@doctest
In [135]: numpy.random.rand(10,2)
```

```
Out[135]:
array([[ 0.64524308,  0.59943846],
       [ 0.47102322,  0.8715456 ],
       [ 0.29370834,  0.74776844],
       [ 0.99539577,  0.1313423 ],
       [ 0.16250302,  0.21103583],
       [ 0.81626524,  0.1312433 ],
       [ 0.67338089,  0.72302393],
       [ 0.7566368 ,  0.07033696],
       [ 0.22591016,  0.77731835],
       [ 0.0072729 ,  0.34273127]])
```

Another demonstration of multi-line input and output

```
.. ipython::
```

```
In [106]: print x
-----> print(x)
jdjh
```

```
In [109]: for i in range(10):
.....:     print i
.....:
.....:
0
1
2
3
4
5
6
7
8
9
```

Most of the "pseudo-decorators" can be used as options to ipython mode. For example, to setup matplotlib pylab but suppress the output, you can do. When using the matplotlib ``use`` directive, it should occur before any import of pylab. This will not show up in the rendered docs, but the commands will be executed in the embedded interpreter and subsequent line numbers will be incremented to reflect the inputs::

```
.. ipython::
   :suppress:
```

```
In [144]: from pylab import *
```

```
In [145]: ion()
```

```
.. ipython::
   :suppress:
```

```
In [144]: from pylab import *
```

```
In [145]: ion()
```

Likewise, you can set ``:doctest:`` or ``:verbatim:`` to apply these settings to the entire block. For example,

```
.. ipython::  
    :verbatim:
```

```
In [9]: cd mpl/examples/  
/home/jdhunter/mpl/examples
```

```
In [10]: pwd  
Out[10]: '/home/jdhunter/mpl/examples'
```

```
In [14]: cd mpl/examples/<TAB>  
mpl/examples/animation/      mpl/examples/misc/  
mpl/examples/api/            mpl/examples/mplot3d/  
mpl/examples/axes_grid/      mpl/examples/pylab_examples/  
mpl/examples/event_handling/  mpl/examples/widgets
```

```
In [14]: cd mpl/examples/widgets/  
/home/jdhunter/mpl/examples/widgets
```

```
In [15]: !wc *  
   2   12   77 README.txt  
  40   97  884 buttons.py  
  26   90  712 check_buttons.py  
  19   52  416 cursor.py  
 180  404 4882 menu.py  
   16   45  337 multicursor.py  
   36  106  916 radio_buttons.py  
   48  226 2082 rectangle_selector.py  
   43  118 1063 slider_demo.py  
   40  124 1088 span_selector.py  
  450 1274 12457 total
```

You can create one or more pyplot plots and insert them with the ``@savefig`` decorator.

```
.. ipython::
```

```
@savefig plot_simple.png width=4in  
In [151]: plot([1,2,3]);  
  
# use a semicolon to suppress the output  
@savefig hist_simple.png width=4in  
In [151]: hist(np.random.randn(10000), 100);
```

In a subsequent session, we can update the current figure with some text, and then resave

```
.. ipython::
```

```
In [151]: ylabel('number')
```

```
In [152]: title('normal distribution')
```

```
@savefig hist_with_text.png width=4in
```

```
In [153]: grid(True)
```

Pseudo-Decorators

=====

Here are the supported decorators, and any optional arguments they take. Some of the decorators can be used as options to the entire block (eg ```verbatim``` and ```suppress```), and some only apply to the line just below them (eg ```savefig```).

`@suppress`

execute the ipython input block, but suppress the input and output block from the rendered output. Also, can be applied to the entire ```..ipython``` block as a directive option with ```:suppress:```.

`@verbatim`

insert the input and output block in verbatim, but auto-increment the line numbers. Internally, the interpreter will be fed an empty string, so it is a no-op that keeps line numbering consistent. Also, can be applied to the entire ```..ipython``` block as a directive option with ```:verbatim:```.

`@savefig OUTFILE [IMAGE_OPTIONS]`

save the figure to the static directory and insert it into the document, possibly binding it into a minipage and/or putting code/figure label/references to associate the code and the figure. Takes args to pass to the image directive (`*scale*`, `*width*`, etc can be kwargs); see `image options` <http://docutils.sourceforge.net/docs/ref/rst/directives.html#image> for details.

`@doctest`

Compare the pasted in output in the ipython block with the output generated at doc build time, and raise errors if they don't match. Also, can be applied to the entire ```..ipython``` block as a directive option with ```:doctest:```.

`.. _ipython_literal:`

Sphinx source for this tutorial

=====

`.. literalinclude:: ipython_directive.rst`

SPHINX CHEAT SHEET

Here is a quick and dirty cheat sheet for some common stuff you want to do in sphinx and ReST. You can see the literal source for this file at *cheatsheet -literal*.

5.1 Formatting text

You use inline markup to make text *italics*, **bold**, or `monotype`.

You can represent code blocks fairly easily:

```
import numpy as np
x = np.random.rand(12)
```

Or literally include code:

```
from pylab import *
from matplotlib.patches import Ellipse

delta = 45.0 # degrees

angles = arange(0, 360+delta, delta)
ells = [Ellipse((1, 1), 4, 2, a) for a in angles]

a = subplot(111, aspect='equal')

for e in ells:
    e.set_clip_box(a.bbox)
    e.set_alpha(0.1)
    a.add_artist(e)

xlim(-2, 4)
ylim(-1, 3)

show()
```

5.2 Making a list

It is easy to make lists in rest

5.2.1 Bullet points

This is a subsection making bullet points

- point A
- point B
- point C

5.2.2 Enumerated points

This is a subsection making numbered points

1. point A
2. point B
3. point C

5.3 Making a table

This shows you how to make a table – if you only want to make a list see [Making a list](#).

Name	Age
John D Hunter	40
Cast of Thousands	41
And Still More	42

5.4 Making links

It is easy to make a link to [yahoo](#) or to some section inside this document (see [Making a table](#)) or another document.

You can also reference classes, modules, functions, etc that are documented using the sphinx [autodoc](#) facilities. For example, see the module `matplotlib.backend_bases` documentation, or the class `LocationEvent`, or the method `mpl_connect()`.

5.5 This file

```
.. _cheat-sheet:
```

```
*****  
Sphinx cheat sheet  
*****
```

```
Here is a quick and dirty cheat sheet for some common stuff you want  
to do in sphinx and ReST. You can see the literal source for this  
file at :ref:`cheatsheet  
-literal`.
```

```
.. _formatting-text:
```

Formatting text
=====

You use inline markup to make text *italics*, **bold**, or `'monotype'`.

You can represent code blocks fairly easily::

```
import numpy as np
x = np.random.rand(12)
```

Or literally include code:

```
.. literalinclude:: pyplots/ellipses.py
```

```
.. _making-a-list:
```

Making a list
=====

It is easy to make lists in rest

Bullet points

This is a subsection making bullet points

- * point A

- * point B

- * point C

Enumerated points

This is a subsection making numbered points

- #. point A

- #. point B

- #. point C

```
.. _making-a-table:
```

Making a table
=====

This shows you how to make a table -- if you only want to make a list see :ref:`making-a-list`.

=====	=====
Name	Age
=====	=====
John D Hunter	40
Cast of Thousands	41
And Still More	42

```
=====
```

```
.. _making-links:
```

Making links

```
=====
```

It is easy to make a link to `yahoo <<http://yahoo.com>>`_ or to some section inside this document (see :ref:`making-a-table`) or another document.

You can also reference classes, modules, functions, etc that are documented using the sphinx `autodoc <<http://sphinx.pocoo.org/ext/autodoc.html>>`_ facilites. For example, see the module :mod:`matplotlib.backend_bases` documentation, or the class :class:`~matplotlib.backend_bases.LocationEvent`, or the method :meth:`~matplotlib.backend_bases.FigureCanvasBase.mpl_connect`.

```
.. _cheatsheet-literal:
```

This file

```
=====
```

```
.. literalinclude:: cheatsheet.rst
```

EMACS REST SUPPORT

6.1 Emacs helpers

There is an emacs mode `rst.el` which automates many important ReST tasks like building and updating table-of-contents, and promoting or demoting section headings. Here is the basic `.emacs` configuration:

```
(require 'rst)
(setq auto-mode-alist
      (append ' (("\\.txt$" . rst-mode)
                  ("\\.rst$" . rst-mode)
                  ("\\.rest$" . rst-mode)) auto-mode-alist))
```

Some helpful functions:

C-c TAB - `rst-toc-insert`

Insert table of contents at point

C-c C-u - `rst-toc-update`

Update the table of contents at point

C-c C-l `rst-shift-region-left`

Shift region to the left

C-c C-r `rst-shift-region-right`

Shift region to the right

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*