sphinx demo

Release 0.1.1

anita kean

January 28, 2010

CONTENTS

This is the sphinx-generated pdf of this demo. The source-files for the demo are available for download from http://nzpug.org/MeetingsAuckland/November2009.

Sphinx was released in early 2008 as a tool to process the python documentation

- It renders text with relatively little markup into html/pdf
- `LaTeX`'s math-mode is available.
- It renders code with language-specific highlighting (defaults to python)
- provides most of the features of html and latex markup:
  - linking
  - referencing (sections, arbitrary locations)
  - indexing (terms, objects)
  - navigation (sections, optionally numbered)
  - inclusion (images, tables, text, code, objects)
  - exclusion (tag-based, e.g. for html/pdf)
  - math-mode (e.g. $e^{i\theta} = \cos(\theta) + i\sin(\theta)$)
  - citation
  - footnotes
- By default, has a `Show Source` link in its Table-of-Contents panel where you can *view un-rendered text*.
- It is being actively developed.
- It has an active Usenet newsgroup sphinx-dev.
- It is used by many projects - you can learn from these.

The code depends on

- `docutils` (restructured text parser),
- `jinja2` (templating tool) and
- `pygments` (highlighter).
- **latex** (fonts, mathematical formatting) - with the texlive package you'll need `latex-recommended`, `latex-extra` and `fonts-recommended`.
  - see the Sphinx Extensions documentation for how to use ReportLab's `rst2pdf` instead.

It has a set of extensions which includes:

autodoc extract docstrings from documented code

intersphinx link documentation to other python documentation

doctest enable testing docstrings with doctest

and a set of builders which includes:

html build html documentation

latex build latex documentation - which can then be written out to pdf with such tools as **pdflatex**

doctest run docstrings through doctest

linkcheck check all hyperlinks in document tree

CHAPTER

ONE

GENERAL SPHINX USAGE

This documentation contains an introduction to restructured text markup and to using `sphinx` to build documentation.

With a few exceptions, `sphinx` markup is a superset of docutils restructured text (ReST). You may prefer to start with the more detailed `docutils` ReSt documentation (e.g. `docutils` ReSt quickref).

Your best resource is other sphinx documentation on your computer: - once you have `sphinx` itself installed, browse its `doc` directory to see the source documentation.

1.1 getting started

A sphinx installation provides you with two command-line tools:

sphinx-quickstart initiate a sphinx project

sphinx-build build the documentation (to html or pdf) from newly-edited source-files

1.1.1 sphinx-quickstart

At the command-line, in a directory where you want to develop the documentation, type:

```
$ sphinx-quickstart
```

for an interactive program which queries setup options and writes out a file `conf.py` in the root source directory of your project. Edit this file by hand afterwards to change configuration as you prefer.

The default settings work well for a first attempt, but this package comes with the source files in a separate directory from the build directory.

i.e. in response to the **sphinx-quickstart** question:

```
You have two options for placing the build directory for Sphinx output.
Either, you use a directory "_build" within the root path, or you separate
"source" and "build" directories within the root path.
> Separate source and build directories (y/N) [n]:
```

– I answered `y`.

By default, sphinx installs a `Makefile` (linux) or `make.bat` (MS) in the top directory. This provides common incantations of **sphinx-build** for the edit-build cycle.

Extensions are offered in **sphinx-quickstart** - you can elect to include these now, or do so later [1]. We'll illustrate the following extensions:

- `autodoc` - to pull in docstrings from code
- `intersphinx` - to link with other python documentation
- `pngmath` (or `jsmath`) - to enable `LaTeX` math-mode rendered in html [2]

After unpacking this archive, you'll have directory structure:

```
.
|-- Makefile          (make directory)
|-- README
|-- build
|   |-- doctrees
|   |-- html          (view html output here)
|   `-- latex         (issue make all-pdf, view pdf/latex/tex here)
|-- make.bat
|-- source            (root source directory)
|   |-- _static
|   |-- _templates
|   |-- conf.py
|   |-- .. (files and directories)
|-- src
|   __init__.py
|   `-- odict1.py
|   `-- odict2.py
|   `-- odict3.py
```

If you start a new project with the **sphinx-quickstart** command, you'll have directory structure:

```
.
|-- Makefile
|-- build
|-- make.bat
|-- source
|   |-- _static
|   |-- _templates
|   |-- conf.py
|   `-- index.rst
```

Henceforth, reference will be made to the

`root source directory` where the file `conf.py` is.

`make directory` where the file `Makefile` is.

Filenames referenced in `sphinx` are relative to either to the current file (e.g. `subdirectory/filename`, `../../filename`), or to the `root source directory` (`/_static/subdirectory/../filename`)

1.1.2 sphinx-build

If **man** (for `manual`) is available on your system (linux, Mac), [3] then

[1] by editing the `extensions` variable in the file `conf.py`
[2] the `sphinx.ext.jsmath` extension is very large and doesn't come with the sphinx installation - it needs to be downloaded separately.
[3] what about Windows?

```
$ man sphinx_build
```

shows **sphinx-build** usage.

```
$ sphinx-build  -b html -d build/doctrees source build/html

$ sphinx-build  -b latex -d build/doctrees source build/latex

$ sphinx-build  -b doctest -d build/doctrees source
```

Alternatively, use the convenience of **make** and the `Makefile` (linux, Mac) or `make.bat` (Windows) written by **sphinx-quickstart**:

```
$ make help                      (to see all possibilities)
$ make html                      (to render docs as html)
Build finished. The HTML pages are in build/html.

$ browse build/html/index.html   (to view the html-docs)
```

for the html, or

```
$ make latex                     (to render docs as tex)
$ cd build/latex
$ make all-pdf                   (to process tex file to pdf)
```

for the tex/pdf, and

```
$ make doctest
```

to run doctest on the docstrings of your documented code.

In addition, you'll see from **make help** that you can also run:

```
$ make linkcheck
```

to have sphinx check all the links in your documents.

Also, if you want to remove old build code and start afresh,

```
$ make clean
```

Footnote

root-directory location of the sphinx configuration file `conf.py` from the **sphinx-quickstart** output.

1.2 building a documentation set

Once you've run **sphinx-quickstart**, you have a basic skeleton for a `sphinx` project. Using the default settings (except for electing separate directories for the `source` and `build`), you'll have the following set of files:

```
.
|-- Makefile
|-- build
|-- make.bat
|-- source
|   |-- _static
|   |-- _templates
|   |-- conf.py
|   `-- index.rst
```

You'll be in the same directory as the `Makefile`, and can immediately see the html and pdf of this basic skeleton.

For the html:

```
$ make html
```

Point your browser to `build/html/index.html`.

You're viewing a minimal project: it contains only one file, the `index.html` file rendered from the default `index.rst` in the root source directory.

Note: Click on the `Show Source` link in the navigation panel to see the source of the `index.html`. (Then check your `source/index.rst` to see that this is the same!)

And for the pdf:

```
$ make latex
$ cd build/latex
$ make all-pdf
```

Now view the file `<your_project_name_here>.pdf`.

Note: The name of the root html page is `index.html`, but the name of your pdf will be the project name you gave when you ran **sphinx-quickstart**. Look in your `conf.py` file for the line which defines the variable `latex_documents`:

```
latex_documents = [
   ('index', 'sphinx_demo.tex', u'sphinx demo Documentation',
             u'your name here', 'manual'),
          ]
```

The **make all-pdf** command will produce a file called (in this case) `sphinx_demo.pdf`. Edit the `conf.py` file to change this name as you wish.

Recommended work-setup:

You are working simultaneously on at least three files in three different locations:

```
./Makefile                    (make directory)
./build/html/index.html       (html directory)
./build/latex/project.pdf     (pdf directory)
./source/                     (root source  directory)
```

It helps to have:

- a workspace in the *make directory* where you run **make** commands to regenerate the html and latex
- another workspace in the *source directory* tree with an editor open on the page you're currently writing,
- another workspace where you browse the html page
- possibly another in the build/latex directory where you run the **make all-pdf** and refresh a view of the pdf file

The GNU tool **screen** provides a sane way to get about these tasks.

The first file to edit is the base file: `source/index.rst` which was written out by **sphinx-quickstart**:

```
 1
 2 Main title for sphinx demo
 3 ==========================
 4
 5
 6 .. toctree::
 7    :maxdepth: 2
 8
 9 Indices and tables
10 ==================
11
12 * :ref:`genindex`
13 * :ref:`modindex`
14 * :ref:`search`
```

- **It has a `toctree` directive (line 6):** to list the links that should appear on the Table of Contents tree,

  but it's empty - no files are included in this web-page project yet.

- **`toctree` has option `maxdepth` of 2 (line 7):** any pages listed in the tree will be exposed two levels down (section, subsection) in the navigation tree which `sphinx` will produce.

To extend the project, open `source/index.rst` in an editor, and insert some text:

```
Main title for sphinx demo
==========================

Here's the first paragraph of the main page of
the sphinx demo.

All text is written as restructured text.

Here's a list:
*  first item
*  second item

and here's a literal block::

   literal block
```

```
   here
   text is not interpreted


By making an entry in the Table of Contents tree (``toctree``)
at the bottom of this file - the name of an rst-file which appears
in the same directory - a link will appear to that page.
So we simultaneously create a page for that link, and
insert some rst.   We'll call the page ``page1.rst``.

.. toctree::
   :maxdepth: 2

   self
   page1

Indices and tables
==================

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

There are now two entries in the `toctree`:

- `self`
- `page1.rst`

sitemap

The `self` entry in the `toctree` is optional, but its presence generates a `sitemap` for the html pages - with links at top and bottom of screen for navigation up the document-tree.

Next, create a new file `page1.rst` in the same directory and put some related material in it. Here's an example:

```
Second page of the documentation
================================

This page links from the main :file:`index.html`

This page is going to link to separate page titled
Side Topic.

Create a directory in the root source directory called
side_topic and create two files there:

* :file:`side_topic/index.rst`
* :file:`side_topic/page_a.rst`

Then in the Table of Contents of this file, we'll list the file
:file:`side_topic/index,
and in :file:`side_topic/index, list page_a in its table of contents.

.. toctree::
   :maxdepth: 1
```

```
   side_topic/index
.. any other files you like here
```

Notice that this file also has its own Table of Contents Tree - it lists the file `side_topic/index`. This assumes a directory `side_topic` exists, and a file `index.rst` exists in that directory. The link to that page in html Table of Contents will be the main title on that page:

```
Side Topic
==========


.. toctree::
   :maxdepth: 2

   page_a

Indices and tables
==================

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

If this `side_topic` were to be expanded, a series of files might exist in that directory, all of which were referenced by `index.rst` there.

Your root source directory will now look like:

```
.
|-- _static
|-- _templates
|-- conf.py
|-- index.rst
|-- page1.rst
`-- side_topic
    |-- index.rst
    `-- page_a.rst
```

and after you build the html, your `build/html` directory structure will contain corresponding structure.

1.3 some markup features

What follows is a mixture of restructured text and sphinx markup to illustrate some useful features. Refer to the DocUtils Primer and Sphinx documentation for full resources.

`sphinx` markup extends the restructured text of `docutils`.

Restructured Text (ReST)

Text is white-space sensitive. Tab-indentations are three spaces.
In addition:

- text enclosed in single back-ticks is *interpreted*
- text enclosed in double back-ticks is simply `emphasized`

Most markup consists of *directives* and *roles*.

- blank lines define the beginning and end of directives
- change in indentation level can also signify change in markup

directive text of the form:

```
.. directive_type::   name
   :directive_option:  option_argument
   :another_directive_option:

   body of directive
   ...
   ...

continuation after directive ...
```

directive names include the general features: `toctree` `code-block` `image` `figure` `glossary` `include` `index` `literalinclude` `math` `note` `only` `seealso` `table` `topic` `warning`

See for example the form of equation-numbered *math-equations* and *csv-tables*.

See Also:

The section *Python Markup* covers some Python-specific directives used in marking up Python code.

role (inline): - of the form:

```
... :role_name_here:`text to be interpreted here` ...
```

Role names include:

`math` (see *math (latex)*) inline math-mode,

`obj`, `meth`, `class`, ... to reference python objects documented with the corresponding directive – see the source of *Python Markup* for an illustration

`ref` - cross-reference to text indexed with `index` directive or *link label* (see *links and cross-references*)

1.3.1 basic markup features

sections

You'll see from the structure of the source pages of this demo that sections are demarcated by titles followed by a line of underscoring.

Sections can be referenced if a label is placed on the lines preceding it:

```
.. _section_label_name:

section here
------------
```

comments

The presence of two periods at the beginning of any line, an no other combination of symbols which makes it a directive, determines a comment. The text following is ignored till the occurrence of the next blank line:

```
..  commented text here
```

inclusion

Text be be read into the current file either in-line, or pre-formatted.

in-line inclusion The `..  include::` directive causes in-line inclusion:

```
.. include:: /_static/some_file.txt
```

You'll find plenty of occurrences of it in the sources of this demo.

pre-formatted inclusion to include a file, pre-formatted (no interpretation - e.g. quotation of formatted text or code):

```
.. literalinclude:: /_static/fib.py
```

This produces:

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""Fibonacci sequences using generators

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""

def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print n,
```

(the root directory which contains the file `conf.py` also contains `_static`, so the file is named absolutely - relative to that directory)

To see that file with line-numbers, add the *:linenos:* option:

```
.. literalinclude:: /_static/fib.py
   :linenos:
```

which gives:

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""Fibonacci sequences using generators

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""

def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print n,
```

To extract just lines 16, 17 of this file, add the `:lines:` option:

```
.. literalinclude:: /_static/fib.py
   :lines: 16-17
```

to get

```
for n in fibonacci(1000):
    print n,
```

We can also select specific python object definitions with the `:pyobject:` option:

```
.. literalinclude:: /_static/fib.py
   :pyobject: fibonacci
```

```
def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b
```

literal blocks

Sphinx interprets anything following the form `::` `<newline>` as pre-formatted text:

```
Text in a literal block::

   The indented paragraph following this is highlighted and presented as
   is.
```

results in :

Text in a literal block:

```
The indented paragraph following this is highlighted and presented as
is.
```

(Notice one of the colons has been removed.)

An isolated `::` also signals a literal block, - but now both colons are removed in output.

code

By default, code in literal blocks is highlighted as Python code.

The following python code is thus highlighted simply by appearing in an indented region preceded by `::`:

```
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
"""Fibonacci sequences using generators

This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""


def fibonacci(max):
    a, b = 0, 1
    while a < max:
        yield a
        a, b = b, a+b

for n in fibonacci(1000):
    print n,
```

See Also:

literal includes for including code from files, numbering it and including code selectively.

Non-python code is highlighted with the `.. code-block:: <language_here>` directive, (the trailing argument specifying the language).

So:

```
.. code-block:: sh

   echo $PATH
   for file in *;
      do
         ...
      done;
```

will be highlighted as shell-code:

```
echo $PATH
for file in *;
   do
      ...
   done;
```

lists

itemized lists Just as you type lists in raw text with asterisks, so also in ReST:

```
*  item 1 here
*  item two follows, being careful to keep the same
   indentation as previous line
*  item 3
```

gives

- item 1 here
- item two follows, being careful to keep the same indentation as previous line
- item 3

Similarly,

```
#.  item 1 here
#.  item two follows, being careful to keep the same
    indentation as previous line
#.  item 3
```

gives

1. item 1 here
2. item two follows, being careful to keep the same indentation as previous line
3. item 3

And finally there's the description list:

```
item one
   description of item 1 here
item two
   description of item two here
   again continuting at same indentation
item three
   description of item three here
```

gives

item one description of item 1 here

item two description of item two here again continuting at same indentation

item three description of item three here

math (latex)

Note: One or the other of sphinx extensions `sphinx.ext.pngmath` and `sphinx.ext.jsmath` (but not both) must be enabled to see LaTeX math markup rendered in your html pages.

Either you elected the extension in your `sphinx-quickstart`, or you edit the configuration file conf.py associated with this sphinx project:

```
# If extensions (or modules to document with autodoc) are in another directory,
# Add any :mod:`sphinx` extension module names here, as strings. They can be extensions
extensions = [..., 'sphinx.ext.pngmath',...]
```

The `sphinx.ext.jsmath` extension is very large, and doesn't come with the sphinx package. You need to download it separately.

inline As with other markup, there is a directive and a role:

to insert $c = \pm\sqrt{a^2 + b^2}$ in your code, just prepend it with the :math: mode tag:

```
to insert :math:`c= \pm \sqrt{a^2+b^2}` in your code, ...
```

paragraph-mode The quadratic formula gives in general, two solutions to the equation $ax^2 + bx + c = 0$:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{1.1}$$

in your code, just prepend it with the `..math::` directive:

```
.. math::  x = \frac{-b\pm \sqrt{b^2-4ac}}{2a}
   :label:  quadratic
```

The `:label:` option creates a label and allows us to refer to the equation elsewhere (see (1.1)).

See Also:

`sphinx.ext.mathbase` (in contrast with the other modules in `sphinx.ext`, this extension does not need to be listed in your `conf.py`)

footnotes

Footnote labels are established with references enclosed in square brackets and trailing underscore::

```
place a footnote here [1]_ please.
```

The footnote label is either a number or a `#` (hash). Occurrences in the text of `[#]_` or `[3]_` link to the set of footnotes found in following incantation of

```
.. rubric:: Footnotes
.. [#] first footnote here
.. [3] third footnote
```

citations

These are of the same form as footnotes, but are any text except hashes and numbers. They can be referenced from any file in the document tree. See the sphinx documentation for more details ([Ref]) (or the source of this page)

Citations will end up in the bibliography of the built document.

glossary

A definition list under the glossary directive indexes the terms defined there. See the section *cross-references* for referring to these terms:

```
demo glossary
^^^^^^^^^^^^^

.. glossary::

   term 1
      definition for term 1 here
      can be several lines long at same indentation level

   term 2
      another term for the glossary.
```

This producess the

demo glossary

term 1 definition for term 1 here can be several lines long at same indentation level

term 2 another term for the glossary.

and we can now refer to *term 1* and *term 2*.

1.3.2 paragraph-mode constructs

tables

Tables can entered literally:

```
=====  ===== ===== ======
  a    b     c       d
=====  ===== ===== ======
   1     2     3     4
   5     6     7     8
   9     10    11    12
=====  ===== ===== ======
```

yield

a	b	c	d
1	2	3	4
5	6	7	8
9	10	11	12

Alternatively, source from csv (either inline or from a file):

```
.. csv-table:: integers
   :header: a,b,c,d
   :widths: 10,10,10,10

   0,2,3,4
```

```
   5,6,7,8
   9,10,11,12
```

to produce

Table 1.1: integers

a	b	c	d
0	2	3	4
5	6	7	8
9	10	11	12

To source from a file, use the option `:file: path/to/filename/here.csv`.

Another simple alternative is the list-entry table, generated with:

```
.. list-table:: Caption
   :header-rows: 1

   *  - Left
      - Right
   *  - 1
      - 2
```

which produces :

Table 1.2: Caption

Left	Right
1	2

images

The directive is:

```
.. image:: filename.type
```

Given that the pdflatex-builder will prefer a pdf and the html-builder an image-format, using the form:

```
.. image:: filename.*
```

allows builders to select their preferred format, if available.

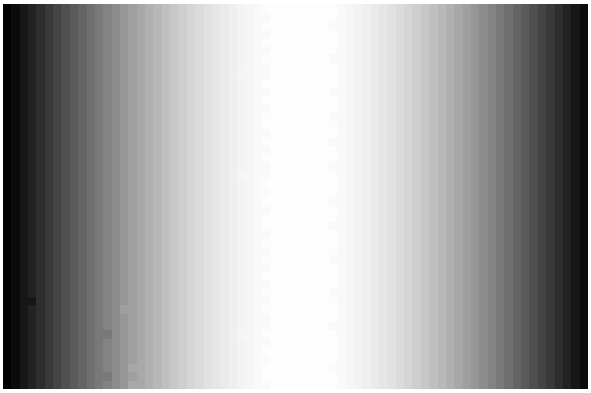
As with other inclusions, path name is relative to the current directory or to the *root directory*:

```
.. image:: path/to/filename.type
   or
.. image:: /path/to/filename.type
```

For example, the text

```
.. image:: /_static/gradient_fx_cos.*
   :alt:  gradient image
   :height: 2in
```

produces



figures

Using the figure directive:

```
.. figure:: filename.type

   caption here
```

you can cross-reference images with the same syntax as with sections: place a link just above the directive:

```
.. _figure_label_here:
.. figure:: filename.type

   caption here

reference then made to :ref:`figure_label_here` or alternatively to
:ref:`alternate label <figure_label_here>`
```

Figure 1.1: caption here

reference is then made to *caption here* or alternatively to *alternate label*.

minipages

If you want text in a box, the directives available to you are:

```
.. note::
.. seealso
.. topic
.. warning
```

The text which follows must be indented. E.g.:

```
.. note:: In order to insert a warning in your restructured text,
   precede the next line with a blank line, then the following
   text

.. warning:: Here's the warning.


Continue on with regular text after another blank line.
And if you want your own title for the box, use the ``topic``
directive, as follows:

.. topic::  Topic here

   A paragraph of text for this box.

If there are related pages, you might want to add the  directive:

.. seealso:: See the related page too.
```

This produces:

Note: In order to insert a warning in your restructured text, precede the next line with a blank line, then the following text

Warning: Here's the warning.

Continue on with regular text after another blank line. And if you want your own title for the box, use the `topic` directive, as follows:

Topic here

A paragraph of text for this box.

If there are related pages, you might want to add the directive:

See Also:

See the related page too.

(end of output)

1.3.3 links, cross-references, indexing

www-links

Sphinx renders text of the form `http://www...` as a web-link without it requiring any markup. Alternate text appears for the link when of the form (note the trailing underscore - rst-format):

```
`Alternate text <http://www.google.com>`_
```

indexing

Preceding any line with an `index directive` of the form:

```
.. index:: item1, item2, item3, ...
```

inserts index entries for these items, which link back to this position. An `index directive` of the form:

```
.. index:: pair: itemA; itemB
```

inserts two entries in the *Index* of the form:

itemA itemB

and

itemB itemA

See the sphinx markup documentation for further constructs.

cross-references

Sphinx extends restructured text in cross-referencing between documents. Cross-references may be made to indexed items.

Items are indexed by being defined as

object	definition	reference	alternate-name
glossary terms	*glossary*	`:term:`item_here``	`:term:`renamed_item <item_here>``
python objects	*Python Markup*	`:obj:`item_here``	`:obj:`renamed_item <item_here>`` [4]
index entries	*indexing*	`:ref:`indexed_item_here``	`:ref:`renamed_item <indexed_item_here>``

labels

Cross-references may also be made to section and figures. This requires that a label is inserted in front of the section header or figure directive:

1. section labels, which precede the section title

   ```
   .. _label_here:

   section title
   -------------
   ```

   This creates a target (label) at this section title.

   We then link to the section title in this or other documents with text of the form

   ```
   :ref:`<label_here>`
   ```

[4] `obj` can be any role in the table *Python Markup*

The text appearing in the link will be `section title`, not the label text.

A label to text different from the section title requires the form:

```
:ref:`alternate title <label_here>`
```

A link to the *literal blocks* section is created in this way.

2. figures - labelled in the same way as sections
3. arbitrary text:

   Any text other than a section title or figure caption can be labelled in the same way, but when it is referenced it *must* have an explicit title:

   ```
   :ref:`explicit title here <label_here>`
   ```

And cross-references may be made to other documents with the `:doc:`relative_path_to_doc_here`` role.

CHAPTER

TWO

DOCUMENTING PYTHON

To document a python module with Sphinx, several approaches can be taken:

1. documentation is written in restructured-text that may contain marked-up docstrings
2. markup can be inserted into docstrings in the module code and documentation is based entirely on these docstrings
3. the two approaches are mixed.

If you look in the sources of the Python Documentation, you'll see that it uses the first method exclusively – it does not extract any module docstrings.

In constrast, the `sphinx.ext.autodoc` extension allows for the second of these approaches with very little restructured text in `sphinx` documents.

We will illustrate all three approaches on the same code, the `Ordered Dictionary` module `odict`.

The Ordered Dictionary module `odict` can be downloaded directly from Voidspace and is also available in the `pythonutils` package available from PythonUtils. But for the purposes of illustration, copies of the module are in the `src` directory of this archive.

To avoid incurring errors with duplicate documentation of the same module (multiple targets for references to the same object), three separate copies of the code are provided in three modules named `odict1`, `odict2`, and `odict3`.

- `odict1` will illustrate marking up your docstrings and using that inline
- `odict2` will be documented using `autoclass` to extract some docstrings from your module code and include those in restructured-text documentation
- `odict3` will be documented using `automodule` - documentation drawn solely from module docstrings.

Note: path configuration

If the module you are documenting is not already on your **PYTHONPATH**, sphinx needs to know where to find it.

In the *root source directory*, open the file `conf.py` in an editor and put the `odict` modules on your path by changing:

```
#sys.path.append(os.path.abspath(''))
```

to contain the path to your module. In the case of this package, that line has been replaced by:

```
sys.path.append(os.path.abspath('../src/'))
```

If you are documenting a module that's located on your python-path, creating a symbolic link to that location from the `src` directory is an easy way to access it during the documentation process. In the `src` directory (linux and Mac), issue the command:

```
$ ln -s /absolute/path/to/module   module
```

2.1 Python Markup

See Also:

`sphinx` documentation for module-specific markup

Before we add `sphinx` python directives to the documentation for `odict1`, we cannot cross-reference any members of its objects from sphinx documentation as no labels exist at their points of definition. And the module index of such documentation won't include any objects associated with this module.

Using *directives*, we designate labels for python objects so they can be cross-referenced with corresponding *roles*.

Table 2.1: Python directives and roles

directive	argument	Role
.. module::	name	*:mod:*
.. class::	name[(signature)]	*:class*
.. method::	name(signature)	*:meth:*
.. function::	name(signature)	*:func:*
.. attribute::	name	*:attr:*
.. exception::	name	*:exc:*
.. object::	name	*:obj:*
.. data::	name	*:data:*

Note: default role

To decrease the presence of markup in your documentation, you can use the `default_role` variable of your project's configuration file `conf.py` to designate the role assigned to any interpreted code (enclosed in single backticks) when no role is specified:

```
# The reST default role (used for this markup: 'text') to use for all documents.
#default_role = None
```

You can insert a value of the most-commonly referenced object here (`obj`, `class`, ...).

Suppose we want to document (and refer to) the `items` method of the `OrderedDict` class of the `odict1` module. The relevant lines of `odict1.py` are (extracted with `literalinclude`):

```
class OrderedDict(dict):
    """
    A class of dictionary that keeps the insertion order of keys.

    All appropriate methods return keys, items, or values in an ordered way.

    All normal dictionary methods are available. Update and comparison is
    restricted to other OrderedDict objects.

    Various sequence methods are available, including the ability to explicitly
    mutate the key ordering.

    __contains__ tests::
```

```
    >>> d = OrderedDict(((1, 3),))
    >>> 1 in d
    True
    >>> 4 in d
    False

__len__ tests::

    >>> len(OrderedDict())
    0
    >>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
    3

get tests::

    >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
    >>> d.get(1)
    3
    >>> d.get(4) is None
    1
    >>> d.get(4, 5)
    5
    >>> d
    OrderedDict([(1, 3), (3, 2), (2, 1)])

has_key tests::

    >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
    >>> d.has_key(1)
    1
    >>> d.has_key(4)
    0
"""

def __init__(self, init_val=(), strict=False):
    """
    Create a new ordered dictionary. Cannot init from a normal dict,
    nor from kwargs, since items order is undefined in those cases.

    If the ``strict`` keyword argument is ``True`` (``False`` is the
    default) then when doing slice assignment - the ``OrderedDict`` you are
    assigning from *must not* contain any keys in the remaining dict.

    >>> OrderedDict()
    OrderedDict([])
    >>> OrderedDict({1: 1})
    Traceback (most recent call last):
    TypeError: undefined order, cannot get items from dict
    >>> OrderedDict({1: 1}.items())
    OrderedDict([(1, 1)])
    >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
    >>> d
    OrderedDict([(1, 3), (3, 2), (2, 1)])
    >>> OrderedDict(d)
    OrderedDict([(1, 3), (3, 2), (2, 1)])
    """
    self.strict = strict
    dict.__init__(self)
```

```
        if isinstance(init_val, OrderedDict):
            self._sequence = init_val.keys()
            dict.update(self, init_val)
        elif isinstance(init_val, dict):
            # we lose compatibility with other ordered dict types this way
            raise TypeError('undefined order, cannot get items from dict')
        else:
            self._sequence = []
            self.update(init_val)
```

and the items method:

```
    def items(self):
        """
        ``items`` returns a list of tuples representing all the
        ``(key, value)`` pairs in the dictionary.

        >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
        >>> d.items()
        [(1, 3), (3, 2), (2, 1)]
        >>> d.clear()
        >>> d.items()
        []
        """
        return zip(self._sequence, self.values())
```

Edit your `sphinx` restructured text documentation to contain the docstrings of `odict1.OrderedDict` as follows:

```
.. currentmodule:: odict1
.. class:: OrderedDict([init_val=(), strict=False])

   class of :class:`dict` that keeps the insertion order of keys.

   All appropriate methods return keys, items, or values in an ordered way.

   All normal dictionary methods are available. Update and comparison is
   restricted to other OrderedDict objects.

   Various sequence methods are available, including the ability to explicitly
   mutate the key ordering

   :param init_val:  OrderedDict or list of (key, value) tuples
   :param strict:    Boolean - slice-assignment must be to unique keys

   :returns:    object of type dict
```

and include the following edited docstrings of `odict1.OrdereDict.items` too (the method markup is indented relative to the class markup):

```
   .. method:: items()

       :returns: a list of tuples representing all the-
             ``(key, value)`` pairs in the dictionary.

      >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
      >>> d.items()
      [(1, 3), (3, 2), (2, 1)]
```

```
>>> d.clear()
>>> d.items()
[]
```

This will be rendered in your documentation as:

---

class **OrderedDict**(*[init_val=(), strict=False]*)

class of `dict` that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering

Parameters
- *init_val* – OrderedDict or list of (key, value) tuples
- *strict* – Boolean - slice-assignment must be to unique keys

Returns object of type dict

items()

returns a list of tuples representing all the- `(key, value)` pairs in the dictionary.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.items()
[(1, 3), (3, 2), (2, 1)]
>>> d.clear()
>>> d.items()
[]
```

Now we can refer, from any document in the document-tree, to the method `OrderedDict.items` (click on the link to be taken to its documentation) – or alternatively to `items` if that notation is unambiguous. These objects will now appear in the index, and be searchable.

In the sections on *autodoc*, we'll look at how this this process can be automated with the `sphinx.ext.autodoc` extension.

2.2 using intersphinx

With each build of the html docs (with a **make html** incantation in the make directory), a file called `objects.inv` is written to the `build/html` directory with entries of the form

```
# Sphinx inventory version 1
# Project: sphinx demo
# Version: 0.1.1
odict3 mod python/auto/odict_3.html
odict2.Items.index method python/auto/odict_2.html
odict2.Items.remove method python/auto/odict_2.html
odict3.OrderedDict.items method python/auto/odict_3.html
odict3.OrderedDict.sort method python/auto/odict_3.html
odict3.OrderedDict.copy method python/auto/odict_3.html
odict3.OrderedDict.iterkeys method python/auto/odict_3.html
```

(... - lines snipped)

If you elect to use the `sphinx.ext.intersphinx` extension when you `quickstart` your project, then the root-directory `conf.py` will have a line of the form

```
intersphinx_mapping = {'http://docs.python.org/': None,}
```

where the location for the main python documentation is specified. Consequently, references of the form `:class:`dict`` will be rendered as `dict` giving live links to the documentation for objects in the main python documentation.

Any objects that we refer to with the form:

object	reference
method	*:meth:`method_name`*
class	*:class:`class_name`*
module	*:mod:`module_name`*
object	*:obj:`object_name`*
...	...

will be linked to their documentation if it is available.

If you look in the `conf.py` for this archive, it has

```
# Example configuration for intersphinx: refer to the Python standard library.
intersphinx_mapping = {'http://docs.python.org/': None,
                       'http://sphinx.pocoo.org': None,
                       'http://jinja.pocoo.org/2/documentation': None,
```

With the `sphinx` documentation linked into this build, reference to `sphinx` objects – with roles of the form `:mod:`sphinx.ext.intersphinx`` – are rendered as live links (e.g. `sphinx.ext.intersphinx`) in these pages.

The addresses given are the locations of the `objects.inv` files for the sphinx-generated documentation of these projects.

More lines can be added to that dictionary to access documentation for other sphinx-documented packages.

2.2.1 accessing local documentation sources

If you want to use locally archived documentation for a package, the form of the entry in `intersphinx_mapping` is:

```
intersphinx_mapping = {'http://docs.python.org/': None,}
                       'http://sphinx.pocoo.org/':
                       '/path/to/local/docs/build/html/objects.inv',
                       ... }
```

which allows offline access to your documentation (exactly what I needed for this demo).

2.2.2 electing `sphinx.ext.intersphinx` after quickstart

If you didn't elect to use the `sphinx.ext.intersphinx` extension at *sphinx-quickstart*, then just add it to the extensions line in your `conf.py` file, and add the intersphinx_mapping variable too:

```
# Add any :mod:`sphinx` extension module names here, as strings. They can be
# extensions
# coming with :mod:`sphinx` (named 'sphinx.ext.*') or your custom ones.
extensions = ['sphinx.ext.autodoc', 'sphinx.ext.intersphinx', ...]
```

2.3 using autodoc

See Also:

`sphinx` documentation for the autodoc extension

In the configuration file associated with this sphinx project, the `autodoc`, `intersphinx` and `pngmath` sphinx extensions have been enabled.

We'll illustrate how to document `odict2` and `odict3` using these extensions.

2.3.1 using autoclass

The `autodoc` extension of sphinx (`sphinx.ext.autodoc`) provides a way to extract the docstrings of code into your documentation. If you mark-up your doc-strings in ReST using the method illustrated in the *Python markup* example, then they can be extracted with very little work using the `autoclass` directive of the `autodoc` extension.

Taking the `OrderedDict` class of `odict2` as an example, here are the class docstrings:

```
"""
A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is
restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly
mutate the key ordering.

__contains__ tests::

    >>> d = OrderedDict(((1, 3),))
    >>> 1 in d
    True
    >>> 4 in d
    False

__len__ tests::

    >>> len(OrderedDict())
    0
    >>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
    3

get tests::

    >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
    >>> d.get(1)
```

```
        3
        >>> d.get(4) is None
        1
        >>> d.get(4, 5)
        5
        >>> d
        OrderedDict([(1, 3), (3, 2), (2, 1)])

    has_key tests::

        >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
        >>> d.has_key(1)
        1
        >>> d.has_key(4)
        0
    """
```

We insert the lines

```
.. currentmodule:: odict2
.. autoclass:: OrderedDict
```

in our documentation, to

1. declare that any objects we refer to are in the `odict2` module
2. extract the doc-strings of the `OrderedDict` class.

With these `sphinx` commands, the class docstrings are rendered as:

---

class **OrderedDict**(*init_val=(), strict=False*)

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
True
>>> 4 in d
False
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

Notice that the interactive python sessions (>>> code) are quoted in `literalblock` rst automatically, and the arguments of the `__init__` method appear in the class declaration. Also notice that with `intersphinx` enabled, we can see the documentation for the base `__init__` method (through the link) too.

If we add in the option `:members: __init__` to the `autoclass` directive, as:

```
.. autoclass:: OrderedDict
   :members: __init__
```

we are requesting that the method `OrderedDict.__init__` (or, if it's unambiguous, we could use the form `:meth:`~OrderedDict.__init__`` for `__init__`) docstrings be presented too.

From these two lines, `sphinx.ext.autodoc` produces:

---

class **OrderedDict**(*init_val=()*, *strict=False*)

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
True
>>> 4 in d
False
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

__init__(*init_val=(), strict=False*)

Create a new ordered dictionary. Cannot init from a normal dict, nor from kwargs, since items order is undefined in those cases.

If the `strict` keyword argument is `True` (`False` is the default) then when doing slice assignment - the `OrderedDict` you are assigning from *must not* contain any keys in the remaining dict.

```
>>> OrderedDict()
OrderedDict([])
>>> OrderedDict({1: 1})
Traceback (most recent call last):
TypeError: undefined order, cannot get items from dict
>>> OrderedDict({1: 1}.items())
OrderedDict([(1, 1)])
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
>>> OrderedDict(d)
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

Compare this with the `__init__` method code:

```
def __init__(self, init_val=(), strict=False):
    """
    Create a new ordered dictionary. Cannot init from a normal dict,
    nor from kwargs, since items order is undefined in those cases.

    If the ``strict`` keyword argument is ``True`` (``False`` is the
    default) then when doing slice assignment - the ``OrderedDict`` you are
    assigning from *must not* contain any keys in the remaining dict.

    >>> OrderedDict()
    OrderedDict([])
    >>> OrderedDict({1: 1})
```

```
        Traceback (most recent call last):
        TypeError: undefined order, cannot get items from dict
        >>> OrderedDict({1: 1}.items())
        OrderedDict([(1, 1)])
        >>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
        >>> d
        OrderedDict([(1, 3), (3, 2), (2, 1)])
        >>> OrderedDict(d)
        OrderedDict([(1, 3), (3, 2), (2, 1)])
        """
        self.strict = strict
        dict.__init__(self)
        if isinstance(init_val, OrderedDict):
            self._sequence = init_val.keys()
            dict.update(self, init_val)
        elif isinstance(init_val, dict):
            # we lose compatibility with other ordered dict types this way
            raise TypeError('undefined order, cannot get items from dict')
        else:
            self._sequence = []
            self.update(init_val)

### Special methods ###
```

We can selectively include whichever class-members we want to document as `:members:` arguments. Suppose we want the `__init__`, `insert` and `index` methods to be documented:

```
.. autoclass:: OrderedDict
   :members: __init__, insert, index
```

This gives

---

class **OrderedDict**(*init_val=(), strict=False*)

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
True
>>> 4 in d
False
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

__init__(*init_val=(), strict=False*)

Create a new ordered dictionary. Cannot init from a normal dict, nor from kwargs, since items order is undefined in those cases.

If the `strict` keyword argument is `True` (`False` is the default) then when doing slice assignment - the `OrderedDict` you are assigning from *must not* contain any keys in the remaining dict.

```
>>> OrderedDict()
OrderedDict([])
>>> OrderedDict({1: 1})
Traceback (most recent call last):
TypeError: undefined order, cannot get items from dict
>>> OrderedDict({1: 1}.items())
OrderedDict([(1, 1)])
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
>>> OrderedDict(d)
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

insert(*index, key, value*)

Takes `index`, `key`, and `value` as arguments.

Sets `key` to `value`, so that `key` is at position `index` in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.insert(0, 4, 0)
>>> d
OrderedDict([(4, 0), (1, 3), (3, 2), (2, 1)])
>>> d.insert(0, 2, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2)])
>>> d.insert(8, 8, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2), (8, 1)])
```

index(*key*)

Return the position of the specified key in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.index(3)
1
>>> d.index(4)
Traceback (most recent call last):
ValueError: list.index(x): x not in list
```

We can also add the `:undoc-members:` option to the `autoclass` directive to see the members which have no doc-strings.

show-inheritance

If we add in the `:show-inheritance:` option, through `sphinx.ext.intersphinx` we get a link to the documentation of the `dict` class that OrderedDict inherits from:

```
.. autoclass:: OrderedDict
   :members: __init__, insert, index
   :show-inheritance:
```

class **OrderedDict**(*init_val=()*, *strict=False*)

Bases: `dict`

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
True
>>> 4 in d
False
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

__init__(*init_val=(), strict=False*)

Create a new ordered dictionary. Cannot init from a normal dict, nor from kwargs, since items order is undefined in those cases.

If the `strict` keyword argument is `True` (`False` is the default) then when doing slice assignment - the `OrderedDict` you are assigning from *must not* contain any keys in the remaining dict.

```
>>> OrderedDict()
OrderedDict([])
>>> OrderedDict({1: 1})
Traceback (most recent call last):
TypeError: undefined order, cannot get items from dict
>>> OrderedDict({1: 1}.items())
OrderedDict([(1, 1)])
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
>>> OrderedDict(d)
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

insert(*index, key, value*)

Takes `index`, `key`, and `value` as arguments.

Sets `key` to `value`, so that `key` is at position `index` in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.insert(0, 4, 0)
>>> d
OrderedDict([(4, 0), (1, 3), (3, 2), (2, 1)])
>>> d.insert(0, 2, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2)])
>>> d.insert(8, 8, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2), (8, 1)])
```

index(*key*)

Return the position of the specified key in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.index(3)
1
>>> d.index(4)
Traceback (most recent call last):
ValueError: list.index(x): x not in list
```

using automethod

If we simply want to document particular members of `OrderedDict` without the class docstrings, use `automethod`:

```
.. currentmodule:: odict2
.. automethod::  OrderedDict.insert
```

insert(*index, key, value*)

Takes `index`, `key`, and `value` as arguments.

Sets `key` to `value`, so that `key` is at position `index` in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.insert(0, 4, 0)
>>> d
OrderedDict([(4, 0), (1, 3), (3, 2), (2, 1)])
>>> d.insert(0, 2, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2)])
>>> d.insert(8, 8, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2), (8, 1)])
```

second example

Now we look at the `Items` class of `odict2`. The lines

```
.. autoclass::  Items
   :members:
```

request automatic extraction of docstrings for the whole class - the class docstrings and the docstrings of all members (methods, class attributes).

---

class Items(*main*)

Custom object for accessing the items of an OrderedDict.

Can be called like the normal `OrderedDict.items` method, but also supports indexing and sequence methods.

append(*item*)

Add an item to the end.

There's only one public method of this class which has docstrings - that's `Items.append`.

The `autoclass` invocation has produced output of the form

```
.. method:: Items.append
```

effectively defining a target (with the `method` directive) for the label `Items.append` that we can then link to.

If we add in the options `:undoc-members:` and to the `autoclass` directive, we'll see a listing of all the undocumented members too. The `:undoc-members:` option has added:

```
.. autoclass::  Items
   :members:
   :undoc-members:
   :inherited-members:
```

This gives automatic extraction of docstrings for the whole class.

---

class **Items**(*main*)

Custom object for accessing the items of an OrderedDict.

Can be called like the normal `OrderedDict.items` method, but also supports indexing and sequence methods.

append(*item*)

Add an item to the end.

count(*item*)

extend(*other*)

index(*item, *args*)

insert(*i, item*)

pop(*i=-1*)

remove(*item*)

reverse()

sort(**args, **kwds*)

2.3.2 generic documentation with automodule

In order to illustrate yet another alternative approach, we use the third copy of the file `odict.py` in a fictitious module `odict3`. We'll document the entire module using the `automodule` directive:

```
.. automodule:: odict3
   :members:
   :show-inheritance:
```

This will produce the page

A dict that keeps keys in insertion order

class **OrderedDict**(*init_val=(), strict=False*)

Bases: `dict`

A class of dictionary that keeps the insertion order of keys.

All appropriate methods return keys, items, or values in an ordered way.

All normal dictionary methods are available. Update and comparison is restricted to other OrderedDict objects.

Various sequence methods are available, including the ability to explicitly mutate the key ordering.

__contains__ tests:

```
>>> d = OrderedDict(((1, 3),))
>>> 1 in d
True
>>> 4 in d
False
```

__len__ tests:

```
>>> len(OrderedDict())
0
>>> len(OrderedDict(((1, 3), (3, 2), (2, 1))))
3
```

get tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.get(1)
3
>>> d.get(4) is None
1
>>> d.get(4, 5)
5
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

has_key tests:

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.has_key(1)
1
>>> d.has_key(4)
0
```

clear()

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.clear()
>>> d
OrderedDict([])
```

copy()

```
>>> OrderedDict(((1, 3), (3, 2), (2, 1))).copy()
OrderedDict([(1, 3), (3, 2), (2, 1)])
```

index(*key*)

Return the position of the specified key in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.index(3)
1
>>> d.index(4)
Traceback (most recent call last):
ValueError: list.index(x): x not in list
```

insert(*index, key, value*)

Takes `index`, `key`, and `value` as arguments.

Sets `key` to `value`, so that `key` is at position `index` in the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.insert(0, 4, 0)
>>> d
OrderedDict([(4, 0), (1, 3), (3, 2), (2, 1)])
>>> d.insert(0, 2, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2)])
>>> d.insert(8, 8, 1)
>>> d
OrderedDict([(2, 1), (4, 0), (1, 3), (3, 2), (8, 1)])
```

items()

`items` returns a list of tuples representing all the `(key, value)` pairs in the dictionary.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.items()
[(1, 3), (3, 2), (2, 1)]
>>> d.clear()
>>> d.items()
[]
```

iteritems()

```
>>> ii = OrderedDict(((1, 3), (3, 2), (2, 1))).iteritems()
>>> ii.next()
(1, 3)
>>> ii.next()
(3, 2)
>>> ii.next()
(2, 1)
>>> ii.next()
Traceback (most recent call last):
StopIteration
```

iterkeys()

```
>>> ii = OrderedDict(((1, 3), (3, 2), (2, 1))).iterkeys()
>>> ii.next()
1
>>> ii.next()
3
>>> ii.next()
2
>>> ii.next()
Traceback (most recent call last):
StopIteration
```

itervalues()

```
>>> iv = OrderedDict(((1, 3), (3, 2), (2, 1))).itervalues()
>>> iv.next()
3
```

```
>>> iv.next()
2
>>> iv.next()
1
>>> iv.next()
Traceback (most recent call last):
StopIteration
```

keys()
Return a list of keys in the `OrderedDict`.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.keys()
[1, 3, 2]
```

pop(*key, *args*)
No dict.pop in Python 2.2, gotta reimplement it

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.pop(3)
2
>>> d
OrderedDict([(1, 3), (2, 1)])
>>> d.pop(4)
Traceback (most recent call last):
KeyError: 4
>>> d.pop(4, 0)
0
>>> d.pop(4, 0, 1)
Traceback (most recent call last):
TypeError: pop expected at most 2 arguments, got 3
```

popitem(*i=-1*)
Delete and return an item specified by index, not a random one as in dict. The index is -1 by default (the last item).

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.popitem()
(2, 1)
>>> d
OrderedDict([(1, 3), (3, 2)])
>>> d.popitem(0)
(1, 3)
>>> OrderedDict().popitem()
Traceback (most recent call last):
KeyError: 'popitem(): dictionary is empty'
>>> d.popitem(2)
Traceback (most recent call last):
IndexError: popitem(): index 2 not valid
```

rename(*old_key, new_key*)
Rename the key for a given value, without modifying sequence order.

For the case where new_key already exists this raise an exception, since if new_key exists, it is ambiguous as to what happens to the associated values, and the position of new_key in the sequence.

```
>>> od = OrderedDict()
>>> od['a'] = 1
>>> od['b'] = 2
>>> od.items()
[('a', 1), ('b', 2)]
>>> od.rename('b', 'c')
>>> od.items()
[('a', 1), ('c', 2)]
>>> od.rename('c', 'a')
Traceback (most recent call last):
ValueError: New key already exists: 'a'
>>> od.rename('d', 'b')
Traceback (most recent call last):
KeyError: 'd'
```

reverse()
Reverse the order of the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.reverse()
>>> d
OrderedDict([(2, 1), (3, 2), (1, 3)])
```

setdefault(*key, defval=None*)

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.setdefault(1)
3
>>> d.setdefault(4) is None
True
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1), (4, None)])
>>> d.setdefault(5, 0)
0
>>> d
OrderedDict([(1, 3), (3, 2), (2, 1), (4, None), (5, 0)])
```

setitems(*items*)
This method allows you to set the items in the dict.

It takes a list of tuples - of the same sort returned by the `items` method.

```
>>> d = OrderedDict()
>>> d.setitems(((3, 1), (2, 3), (1, 2)))
>>> d
OrderedDict([(3, 1), (2, 3), (1, 2)])
```

setkeys(*keys*)
`setkeys` all ows you to pass in a new list of keys which will replace the current set. This must contain the same set of keys, but need not be in the same order.

If you pass in new keys that don't match, a `KeyError` will be raised.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.keys()
[1, 3, 2]
>>> d.setkeys((1, 2, 3))
```

```
>>> d
OrderedDict([(1, 3), (2, 1), (3, 2)])
>>> d.setkeys(['a', 'b', 'c'])
Traceback (most recent call last):
KeyError: 'Keylist is not the same as current keylist.'
```

setvalues(*values*)

You can pass in a list of values, which will replace the current list. The value list must be the same len as the OrderedDict.

(Or a `ValueError` is raised.)

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.setvalues((1, 2, 3))
>>> d
OrderedDict([(1, 1), (3, 2), (2, 3)])
>>> d.setvalues([6])
Traceback (most recent call last):
ValueError: Value list is not the same length as the OrderedDict.
```

sort(**args, **kwargs*)

Sort the key order in the OrderedDict.

This method takes the same arguments as the `list.sort` method on your version of Python.

```
>>> d = OrderedDict(((4, 1), (2, 2), (3, 3), (1, 4)))
>>> d.sort()
>>> d
OrderedDict([(1, 4), (2, 2), (3, 3), (4, 1)])
```

update(*from_od*)

Update from another OrderedDict or sequence of (key, value) pairs

```
>>> d = OrderedDict(((1, 0), (0, 1)))
>>> d.update(OrderedDict(((1, 3), (3, 2), (2, 1))))
>>> d
OrderedDict([(1, 3), (0, 1), (3, 2), (2, 1)])
>>> d.update({4: 4})
Traceback (most recent call last):
TypeError: undefined order, cannot get items from dict
>>> d.update((4, 4))
Traceback (most recent call last):
TypeError: cannot convert dictionary update sequence element "4" to a 2-item sequence
```

values(*values=None*)

Return a list of all the values in the OrderedDict.

Optionally you can pass in a list of values, which will replace the current list. The value list must be the same len as the OrderedDict.

```
>>> d = OrderedDict(((1, 3), (3, 2), (2, 1)))
>>> d.values()
[3, 2, 1]
```

class SequenceOrderedDict(*init_val=(), strict=True*)

Bases: `odict3.OrderedDict`

Experimental version of OrderedDict that has a custom object for `keys`, `values`, and `items`.

These are callable sequence objects that work as methods, or can be manipulated directly as sequences.

Test for `keys`, `items` and `values`.

```
>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.keys
[1, 2, 3]
>>> d.keys()
[1, 2, 3]
>>> d.setkeys((3, 2, 1))
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
>>> d.setkeys((1, 2, 3))
>>> d.keys[0]
1
>>> d.keys[:]
[1, 2, 3]
>>> d.keys[-1]
3
>>> d.keys[-2]
2
>>> d.keys[0:2] = [2, 1]
>>> d
SequenceOrderedDict([(2, 3), (1, 2), (3, 4)])
>>> d.keys.reverse()
>>> d.keys
[3, 1, 2]
>>> d.keys = [1, 2, 3]
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.keys = [3, 1, 2]
>>> d
SequenceOrderedDict([(3, 4), (1, 2), (2, 3)])
>>> a = SequenceOrderedDict()
>>> b = SequenceOrderedDict()
>>> a.keys == b.keys
1
>>> a['a'] = 3
>>> a.keys == b.keys
0
>>> b['a'] = 3
>>> a.keys == b.keys
1
>>> b['b'] = 3
>>> a.keys == b.keys
0
>>> a.keys > b.keys
0
>>> a.keys < b.keys
1
>>> 'a' in a.keys
1
>>> len(b.keys)
2
>>> 'c' in d.keys
0
>>> 1 in d.keys
```

```
1
>>> [v for v in d.keys]
[3, 1, 2]
>>> d.keys.sort()
>>> d.keys
[1, 2, 3]
>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)), strict=True)
>>> d.keys[::-1] = [1, 2, 3]
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
>>> d.keys[:2]
[3, 2]
>>> d.keys[:2] = [1, 3]
Traceback (most recent call last):
KeyError: 'Keylist is not the same as current keylist.'


>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.values
[2, 3, 4]
>>> d.values()
[2, 3, 4]
>>> d.setvalues((4, 3, 2))
>>> d
SequenceOrderedDict([(1, 4), (2, 3), (3, 2)])
>>> d.values[::-1]
[2, 3, 4]
>>> d.values[0]
4
>>> d.values[-2]
3
>>> del d.values[0]
Traceback (most recent call last):
TypeError: Can't delete items from values
>>> d.values[::2] = [2, 4]
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> 7 in d.values
0
>>> len(d.values)
3
>>> [val for val in d.values]
[2, 3, 4]
>>> d.values[-1] = 2
>>> d.values.count(2)
2
>>> d.values.index(2)
0
>>> d.values[-1] = 7
>>> d.values
[2, 3, 7]
>>> d.values.reverse()
>>> d.values
[7, 3, 2]
>>> d.values.sort()
>>> d.values
```

```
[2, 3, 7]
>>> d.values.append('anything')
Traceback (most recent call last):
TypeError: Can't append items to values
>>> d.values = (1, 2, 3)
>>> d
SequenceOrderedDict([(1, 1), (2, 2), (3, 3)])


>>> d = SequenceOrderedDict(((1, 2), (2, 3), (3, 4)))
>>> d
SequenceOrderedDict([(1, 2), (2, 3), (3, 4)])
>>> d.items()
[(1, 2), (2, 3), (3, 4)]
>>> d.setitems([(3, 4), (2 ,3), (1, 2)])
>>> d
SequenceOrderedDict([(3, 4), (2, 3), (1, 2)])
>>> d.items[0]
(3, 4)
>>> d.items[:-1]
[(3, 4), (2, 3)]
>>> d.items[1] = (6, 3)
>>> d.items
[(3, 4), (6, 3), (1, 2)]
>>> d.items[1:2] = [(9, 9)]
>>> d
SequenceOrderedDict([(3, 4), (9, 9), (1, 2)])
>>> del d.items[1:2]
>>> d
SequenceOrderedDict([(3, 4), (1, 2)])
>>> (3, 4) in d.items
1
>>> (4, 3) in d.items
0
>>> len(d.items)
2
>>> [v for v in d.items]
[(3, 4), (1, 2)]
>>> d.items.count((3, 4))
1
>>> d.items.index((1, 2))
1
>>> d.items.index((2, 1))
Traceback (most recent call last):
ValueError: list.index(x): x not in list
>>> d.items.reverse()
>>> d.items
[(1, 2), (3, 4)]
>>> d.items.reverse()
>>> d.items.sort()
>>> d.items
[(1, 2), (3, 4)]
>>> d.items.append((5, 6))
>>> d.items
[(1, 2), (3, 4), (5, 6)]
>>> d.items.insert(0, (0, 0))
>>> d.items
[(0, 0), (1, 2), (3, 4), (5, 6)]
```

```
>>> d.items.insert(-1, (7, 8))
>>> d.items
[(0, 0), (1, 2), (3, 4), (7, 8), (5, 6)]
>>> d.items.pop()
(5, 6)
>>> d.items
[(0, 0), (1, 2), (3, 4), (7, 8)]
>>> d.items.remove((1, 2))
>>> d.items
[(0, 0), (3, 4), (7, 8)]
>>> d.items.extend([(1, 2), (5, 6)])
>>> d.items
[(0, 0), (3, 4), (7, 8), (1, 2), (5, 6)]
```

(that's all from the *one automodule incantation*) to which all references to `odict3` should now link.

This is the sphinx-generated pdf of this demo. The source-files for the demo are available for download from http://nzpug.org/MeetingsAuckland/November2009.

CHAPTER

THREE

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[Ref] In the doc directory of your sphinx installation.

MODULE INDEX

O

INDEX

Symbols

A

C

D

E

I

K

L

M

O

P

R

S

T

U

V