Doconce Description

Author: Hans Petter Langtangen

Date: Mar 26, 2011

What Is Doconce?

Doconce is two things:

1. Doconce is a very simple and minimally tagged markup language that look like ordinary ASCII text (much like what you would use in an email), but the text can be transformed to numerous other formats, including HTML, Wiki, LaTeX, PDF, reStructuredText (reST), Sphinx, Epytext, and also plain text (where non-obvious formatting/tags are removed for clear reading in, e.g., emails). From reStructuredText you can go to XML, HTML, LaTeX, PDF, OpenOffice, and from the latter to RTF and MS Word.
2. Doconce is a working strategy for never duplicating information. Text is written in a single place and then transformed to a number of different destinations of diverse type (software source code, manuals, tutorials, books, wikis, memos, emails, etc.). The Doconce markup language support this working strategy. The slogan is: "Write once, include anywhere".

A wide range of markup languages exist. For example, reStructuredText and Sphinx have recently become popular. So why another one?

- Doconce can convert to plain *untagged* text, more desirable for computer programs and email.
- Doconce has less cluttered tagging of text.
- Doconce has better support for copying in parts of computer code, say in examples, directly from the source code files.
- Doconce has stronger support for mathematical typesetting, and has many features for being integrated with (big) LaTeX projects.
- Doconce is almost self-explanatory and is a handy starting point for generating documents in more complicated markup languages, such as Google Wiki, LaTeX, and Sphinx. A primary application of Doconce is just to make the initial versions of a Sphinx or Wiki document.

Doconce was particularly written for the following sample applications:

- Large books written in LaTeX, but where many pieces (computer demos, projects, examples) can be written in Doconce to appear in other contexts in other formats, including plain HTML, Sphinx, or MS Word.

- Software documentation, primarily Python doc strings, which one wants to appear as plain untagged text for viewing in Pydoc, as reStructuredText for use with Sphinx, as wiki text when publishing the software at googlecode.com, and as LaTeX integrated in, e.g., a thesis.
- Quick memos, which start as plain text in email, then some small amount of Doconce tagging is added, before the memos can appear as MS Word documents or in wikis.

Disclaimer: Doconce is a simple tool, largely based on interpreting and handling text through regular expressions. The possibility for tweaking the layout is obviously limited since the text can go to all sorts of sophisticated markup languages. Moreover, because of limitations of regular expressions, some formatting may face problems when transformed to other formats.

You can jump to the section The Doconce Software Documentation Strategy to see a recipe for how to use Doconce, unless you need some more motivation for the problem which Doconce tries to solve.

Motivation: Problems with Documenting Software

Duplicated Information. It is common to write some software documentation in the code (doc strings in Python, doxygen in C++, javadoc in Java) while similar documentation is often also included in a LaTeX or HTML manual or tutorial. Although the various types of documentation may start out to be the same, different physical files must be used since very different tagging is required for different output formats. Over time the duplicated information starts to diverge. Severe problems with such unsynchronized documentation was one motivation for developing the Doconce concept and tool.

Tagging Issues in Python Documentation. A problem with doc strings in Python is that they benefit greatly from some tagging, Epytext or reStructuredText, when transformed to HTML or PDF manuals. However, such tagging looks annoying in Pydoc, which just shows the pure doc string. For Pydoc we should have more minimal (or no) tagging (students and newbies are in particular annoyed by any unfamiliar tagging of ASCII text). On the contrary, manuals or tutorials in HTML and LaTeX need quite much tagging.

Solution. Accurate information is crucial and can only be maintained in a *single physical* place (file), which must be converted (filtered) to suitable formats and included in various documents (HTML/LaTeX manuals/tutorials, Pydoc/Epydoc/HappyDoc reference manuals).

A Common Format. There is no existing format and associated conversion tools that allow a "singleton" documentation file to be filtered to LaTeX, HTML, XML, PDF, Epydoc, HappyDoc, Pydoc, *and* plain untagged text. As we are involved with mathematical software, the LaTeX manuals should have nicely typeset mathematics, while Pydoc, Epydoc, and HappyDoc must show LaTeX math in verbatim mode. Unfortunately, Epytext is annoyed by even very simple LaTeX math (also in verbatim environments). To summarize, we need

1. A minimally tagged markup language with full support for for mathematics and verbatim computer code.

2. Filters for producing highly tagged formats (LaTeX, HTML, XML), medium tagged formats (reStructuredText, Epytext), and plain text with completely invivisble tagging.
3. Tools for inserting appropriately filtered versions of a "singleton" documentation file in other documents (manuals, tutorials, doc strings).

One answer to these points is the Doconce markup language, its associated tools, and a C-style preprocessor tool or the Mako template system. Then we can *write once, include anywhere*! And what we write is close to plain ASCII text.

But isn't reStructuredText exactly the format that fulfills the needs above? Yes and no. Yes, because reStructuredText can be filtered to a lot of the mentioned formats. No, because of the reasons listed in the section What Is Doconce?, but perhaps the strongest feature of Doconce is that it integrates well with LaTeX: Large LaTeX documents (book) can be made of many smaller Doconce units, typically describing examples and computer codes, glued with mathematical pieces written entirely in LaTeX and with heavy cross-referencing of equations, as is usual in mathematical texts. All the Doconce units can then be available also as stand-alone examples in wikis or Sphinx pages and thereby used in other occasions (including software documentation and teaching material). This is a promising way of composing future books of units that can be reused in many contexts and formats, currently being explored by the Doconce maintainer.

A final warning may be necessary: The Doconce format is a minimalistic formatting language. It is ideal when you start a new project when you are uncertain about which format to choose. At some later stage, when you need quite some sophisticated formatting and layout, you can perform the final filtering of Doconce into something more appropriate for future demands. The convenient thing is that the format decision can be posponed (maybe forever - which is the common experience of the Doconce developer).

Dependencies

If you make use of preprocessor directives in the Doconce source, either Preprocess or Mako must be installed. To make LaTeX documents (without going through the reStructuredText format) you also need ptex2tex and some style files that `ptex2tex` potentially makes use of. Going from reStructuredText to formats such as XML, OpenOffice, HTML, and LaTeX requires docutils. Making Sphinx documents requires of course Sphinx. All of the mentioned potential dependencies are pure Python packages which are easily installed.

The Doconce Software Documentation Strategy

- Write software documentation, both tutorials and manuals, in the Doconce format. Use many files - and never duplicate information!
- Use `#include` statements in source code (especially in doc strings) and in LaTeX documents for including documentation files. These documentation files must be filtered to an appropriate

format by the program `doconce` before being included. In a Python context, this means plain text for computer source code (and Pydoc); Epytext for Epydoc API documentation, or the Sphinx dialect of reStructuredText for Sphinx API documentation; LaTeX for LaTeX manuals; and possibly reStructuredText for XML, Docbook, OpenOffice, RTF, Word.

- Run the preprocessor `preprocess` on the files to produce native files for pure computer code and for various other documents.

Consider an example involving a Python module in a `basename.p.py` file. The `.p.py` extension identifies this as a file that has to be preprocessed) by the `preprocess` program. In a doc string in `basename.p.py` we do a preprocessor include in a comment line, say (use triple quotes in the doc string in case the `doc1` documentation includes code snippets with doc strings with the usual triple double quotes): `# #include docstrings/doc1.dst.txt`. The file `docstrings/doc1.dst.txt` is a file filtered to a specific format (typically plain text, reStructedText, or Epytext) from an original "singleton" documentation file named `docstrings/doc1.do.txt`. The `.dst.txt` is the extension of a file filtered ready for being included in a doc string (`d` for doc, `st` for string).

For making an Epydoc manual, the `docstrings/doc1.do.txt` file is filtered to `docstrings/doc1.epytext` and renamed to `docstrings/doc1.dst.txt`. Then we run the preprocessor on the `basename.p.py` file and create a real Python file `basename.py`. Finally, we run Epydoc on this file. Alternatively, and nowadays preferably, we use Sphinx for API documentation and then the Doconce `docstrings/doc1.do.txt` file is filtered to `docstrings/doc1.rst` and renamed to `docstrings/doc1.dst.txt`. A Sphinx directory must have been made with the right `index.rst` and `conf.py` files. Going to this directory and typing `make html` makes the HTML version of the Sphinx API documentation.

The next step is to produce the final pure Python source code. For this purpose we filter `docstrings/doc1.do.txt` to plain text format (`docstrings/doc1.txt`) and rename to `docstrings/doc1.dst.txt`. The preprocessor transforms the `basename.p.py` file to a standard Python file `basename.py`. The doc strings are now in plain text and well suited for Pydoc or reading by humans. All these steps are automated by the `insertdocstr.py` script. Here are the corresponding Unix commands:

```
# make Epydoc API manual of basename module:
cd docstrings
doconce format epytext doc1.do.txt
mv doc1.epytext doc1.dst.txt
cd ..
preprocess basename.p.py > basename.py
epydoc basename

# make Sphinx API manual of basename module:
cd doc
doconce format sphinx doc1.do.txt
mv doc1.rst doc1.dst.txt
cd ..
preprocess basename.p.py > basename.py
```

```
cd docstrings/sphinx-rootdir  # sphinx directory for API source
make clean
make html
cd ../..

# make ordinary Python module files with doc strings:
cd docstrings
doconce format plain doc1.do.txt
mv doc1.txt doc1.dst.txt
cd ..
preprocess basename.p.py > basename.py

# can automate inserting doc strings in all .p.py files:
insertdocstr.py plain .
# (runs through all .do.txt files and filters them to plain format and
# renames to .dst.txt extension, then the script runs through all
# .p.py files and runs the preprocessor, which includes the .dst.txt
# files)
```

Demos

The current text is generated from a Doconce format stored in the:

```
docs/manual/manual.do.txt
```

file in the Doconce source code tree. We have made a demo web page where you can compare the Doconce source with the output in many different formats: HTML, LaTeX, plain text, etc.

The file `make.sh` in the same directory as the `manual.do.txt` file (the current text) shows how to run `doconce format` on the Doconce file to obtain documents in various formats.

Another demo is found in:

```
docs/tutorial/tutorial.do.txt
```

In the `tutorial` directory there is also a `make.sh` file producing a lot of formats, with a corresponding web demo of the results.

From Doconce to Other Formats

Transformation of a Doconce document to various other formats applies the script `doconce format`:

```
Unix/DOS> doconce format format mydoc.do.txt
```

The `preprocess` program is always used to preprocess the file first, and options to `preprocess` can be added after the filename. For example:

```
Unix/DOS> doconce format LaTeX mydoc.do.txt -Dextra_sections
```

The variable `FORMAT` is always defined as the current format when running `preprocess`. That is, in the last example, `FORMAT` is defined as `LaTeX`. Inside

the Doconce document one can then perform format specific actions through tests like `#if FORMAT == "LaTeX"`.

Inline comments in the text are removed from the output by:

```
Unix/DOS> doconce format LaTeX mydoc.do.txt remove_inline_comments
```

One can also remove such comments from the original Doconce file by running a helper script in the `bin` folder of the Doconce source code:

```
Unix/DOS> doconce remove_inline_comments mydoc.do.txt
```

This action is convenient when a Doconce document reaches its final form.

HTML

Making an HTML version of a Doconce file `mydoc.do.txt` is performed by:

```
Unix/DOS> doconce format HTML mydoc.do.txt
```

The resulting file `mydoc.html` can be loaded into any web browser for viewing.

LaTeX

Making a LaTeX file `mydoc.tex` from `mydoc.do.txt` is done in two steps: .. Note: putting code blocks inside a list is not successful in many

Step 1. **Filter the doconce text to a pre-LaTeX form `mydoc.p.tex` for `ptex2tex`:**

```
Unix/DOS> doconce format LaTeX mydoc.do.txt
```

LaTeX-specific commands ("newcommands") in math formulas and similar can be placed in files `newcommands.tex`, `newcommands_keep.tex`, or `newcommands_replace.tex` (see the section Macros (Newcommands)). If these files are present, they are included in the LaTeX document so that your commands are defined.

Step 2. Run `ptex2tex` (if you have it) to make a standard LaTeX file:

```
Unix/DOS> ptex2tex mydoc
```

or just perform a plain copy:

```
Unix/DOS> cp mydoc.p.tex mydoc.tex
```

Doconce generates a `.p.tex` file with some preprocessor macros. For example, to enable font Helvetica instead of the standard Computer Modern font:

```
Unix/DOS> ptex2tex -DHELVETICA mydoc
```

The title, authors, and date are by default typeset in a non-standard way to enable a nicer treatment of multiple authors having institutions in common. The standard LaTeX "maketitle" heading is also available through:

```
Unix/DOS> ptex2tex -DTRAD_LATEX_HEADING mydoc
```

The `ptex2tex` tool makes it possible to easily switch between many different fancy formattings of computer or verbatim code in LaTeX documents. After any `!bc sys` command in the Doconce source you can insert verbatim block styles as defined in your `.ptex2tex.cfg` file, e.g., `!bc sys cod` for a code snippet, where `cod` is set to a certain environment in `.ptex2tex.cfg` (e.g., `CodeIntended`). There are over 30 styles to choose from.

Step 3. Compile `mydoc.tex` and create the PDF file:

```
Unix/DOS> latex mydoc
Unix/DOS> latex mydoc
Unix/DOS> makeindex mydoc   # if index
Unix/DOS> bibitem mydoc     # if bibliography
Unix/DOS> latex mydoc
Unix/DOS> dvipdf mydoc
```

If one wishes to use the `Minted_Python`, `Minted_Cpp`, etc., environments in `ptex2tex` for typesetting code, the `minted` LaTeX package is needed. This package is included by running `doconce format` with the `-DMINTED` option:

```
Unix/DOS> ptex2tex -DMINTED mydoc
```

In this case, `latex` must be run with the `-shell-escape` option:

```
Unix/DOS> latex -shell-escape mydoc
Unix/DOS> latex -shell-escape mydoc
Unix/DOS> makeindex mydoc   # if index
Unix/DOS> bibitem mydoc     # if bibliography
Unix/DOS> latex -shell-escape mydoc
Unix/DOS> dvipdf mydoc
```

The `-shell-escape` option is required because the `minted.sty` style file runs the `pygments` program to format code, and this program cannot be run from `latex` without the `-shell-escape` option.

Plain ASCII Text

We can go from Doconce "back to" plain untagged text suitable for viewing in terminal windows, inclusion in email text, or for insertion in computer source code:

```
Unix/DOS> doconce format plain mydoc.do.txt  # results in mydoc.txt
```

reStructuredText

Going from Doconce to reStructuredText gives a lot of possibilities to go to other formats. First we filter the Doconce text to a reStructuredText file `mydoc.rst`:

```
Unix/DOS> doconce format rst mydoc.do.txt
```

We may now produce various other formats:

```
Unix/DOS> rst2html.py  mydoc.rst > mydoc.html # HTML
Unix/DOS> rst2latex.py mydoc.rst > mydoc.tex  # LaTeX
Unix/DOS> rst2xml.py   mydoc.rst > mydoc.xml  # XML
Unix/DOS> rst2odt.py   mydoc.rst > mydoc.odt  # OpenOffice
```

The OpenOffice file `mydoc.odt` can be loaded into OpenOffice and saved in, among other things, the RTF format or the Microsoft Word format. That is, one can easily go from Doconce to Microsoft Word.

Sphinx

Sphinx documents can be created from a Doconce source in a few steps.

Step 1. Translate Doconce into the Sphinx dialect of the reStructuredText format:

```
Unix/DOS> doconce format sphinx mydoc.do.txt
```

Step 2. Create a Sphinx root directory with a `conf.py` file, either manually or by using the interactive `sphinx-quickstart` program. Here is a scripted version of the steps with the latter:

```
mkdir sphinx-rootdir
sphinx-quickstart <<EOF
sphinx-rootdir
n

-
Name of My Sphinx Document
Author
version
version
.rst
index
n
y
n
n
n
n
y
n
n
y
y
y
EOF
```

These statements are automated by the command:

```
Unix/DOS> doconce sphinx_dir mydoc.do.txt
```

Step 3. Move the `tutorial.rst` file to the Sphinx root directory:

```
Unix/DOS> mv mydoc.rst sphinx-rootdir
```

If you have figures in your document, the relative paths to those will be invalid when you work with `mydoc.rst` in the `sphinx-rootdir` directory. Either edit `mydoc.rst` so that figure file paths are correct, or simply copy your figure directory to `sphinx-rootdir` (if all figures are located in a subdirectory).

Step 4. Edit the generated `index.rst` file so that `mydoc.rst` is included, i.e., add `mydoc` to the `toctree` section so that it becomes:

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

   mydoc
```

(The spaces before `mydoc` are important!)
Step 5. Generate, for instance, an HTML version of the Sphinx source:

```
make clean   # remove old versions
make html
```

Many other formats are also possible.
Step 6. View the result:

```
Unix/DOS> firefox _build/html/index.html
```

Note that verbatim code blocks can be typeset in a variety of ways depending the argument that follows `!bc`: `cod` gives Python (`code-block:: python` in Sphinx syntax) and `cppcod` gives C++, but all such arguments can be customized both for Sphinx and LaTeX output.

Google Code Wiki

There are several different wiki dialects, but Doconce only support the one used by Google Code. The transformation to this format, called `gwiki` to explicitly mark it as the Google Code dialect, is done by:

```
Unix/DOS> doconce format gwiki mydoc.do.txt
```

You can then open a new wiki page for your Google Code project, copy the `mydoc.gwiki` output file from `doconce format` and paste the file contents into the wiki page. Press **Preview** or **Save Page** to see the formatted result.

When the Doconce file contains figures, each figure filename must be replaced by a URL where the figure is available. There are instructions in the file for doing this. Usually, one performs this substitution automatically (see next section).

Tweaking the Doconce Output

Occasionally, one would like to tweak the output in a certain format from Doconce. One example is figure filenames when transforming Doconce to reStructuredText. Since Doconce does not know if the `.rst` file is going to be filtered to LaTeX or HTML, it cannot know if `.eps` or `.png` is the most appropriate image filename. The solution is to use a text substitution command or code with, e.g., sed, perl, python, or scitools subst, to automatically edit the output file from Doconce. It is then wise to run Doconce and the editing commands from a script to automate all steps in going from Doconce to the final format(s). The `make.sh` files in `docs/manual` and `docs/tutorial` constitute comprehensive examples on how such scripts can be made.

The Doconce Markup Language

The Doconce format introduces four constructs to markup text: lists, special lines, inline tags, and environments.

Lists

An unordered bullet list makes use of the `*` as bullet sign and is indented as follows:

```
* item 1

 item 2

  * subitem 1, if there are more
    lines, each line must
    be intended as shown here

  * subitem 2,
    also spans two lines

 item 3
```

This list gets typeset as

- item 1
- item 2
  - subitem 1, if there are more lines, each line must be intended as shown here
  - subitem 2, also spans two lines
- item 3

In an ordered list, each item starts with an `o` (as the first letter in "ordered"):

```
o item 1

o item 2

  * subitem 1

  * subitem 2

o item 3
```

resulting in

1. item 1
2. item 2
   - subitem 1
   - subitem 2
3. item 3

Ordered lists cannot have an ordered sublist, i.e., the ordering applies to the outer list only.

In a description list, each item is recognized by a dash followed by a keyword followed by a colon:

```
- keyword1: explanation of keyword1

- keyword2: explanation
  of keyword2 (remember to indent properly
  if there are multiple lines)
```

The result becomes

keyword1: explanation of keyword1

keyword2: explanation of keyword2 (remember to indent properly if there are multiple lines)

Special Lines

The Doconce markup language has a concept called *special lines*. Such lines starts with a markup at the very beginning of the line and are used to mark document title, authors, date, sections, subsections, paragraphs., figures, etc.

Heading with Title and Author(s). Lines starting with `TITLE:`, `AUTHOR:`, and `DATE:` are optional and used to identify a title of the document, the authors, and the date. The title is treated as the rest of the line, so is the date, but the author text consists of the name and associated institution(s) with the syntax:

```
name at institution1 and institution2 and institution3
```

The `at` with surrounding spaces is essential for adding information about institution(s) to the author name, and the `and` with surrounding spaces is essential as delimiter between different institutions. Multiple authors require multiple `AUTHOR:` lines. All information associated with `TITLE:` and `AUTHOR:` keywords must appear on a single line. Here is an example:

```
TITLE: On an Ultimate Markup Language
AUTHOR: H. P. Langtangen at Center for Biomedical Computing, Simula Research Laborato
AUTHOR: Kaare Dump at Segfault, Cyberspace Inc.
AUTHOR: A. Dummy Author
DATE: November 9, 2016
```

Note the how one can specify a single institution, multiple institutions, and no institution. In some formats (including reStructuredText and Sphinx) only the author names appear. Some formats have "intelligence" in listing authors and institutions, e.g., the plain text format:

```
Hans Petter Langtangen [1, 2]
Kaare Dump [3]
A. Dummy Author

[1] Center for Biomedical Computing, Simula Research Laboratory
[2] Department of Informatics, University of Oslo
[3] Segfault, Cyberspace Inc.
```

Similar typesetting is done for LaTeX and HTML formats.

Section Headings. Section headings are recognized by being surrounded by equal signs (=) or underscores before and after the text of the headline. Different section levels are recognized by the associated number of underscores or equal signs (=):

- 7 underscores or equal signs for sections
- 5 for subsections
- 3 for subsubsections
- 2 underscrores (only! - it looks best) for paragraphs (paragraph heading will be inlined)

Headings can be surrounded by blanks if desired.

Here are some examples:

```
======= Example on a Section Heading =======

The running text goes here.

      ===== Example on a Subsection Heading =====
The running text goes here.

          ===Example on a Subsubsection Heading===

The running text goes here.

__A Paragraph.__ The running text goes here.
```

The result for the present format looks like this:

Example on a Section Heading

The running text goes here.

Example on a Subsection Heading

The running text goes here.

Example on a Subsubsection Heading

The running text goes here.

A Paragraph. The running text goes here.

Figures. Figures are recognized by the special line syntax:

```
FIGURE:[filename, height=xxx width=yyy scale=zzz] possible caption
```

The filename can be without extension, and Doconce will search for an appropriate file with the right extension. If the extension is wrong, say `.eps` when requesting an HTML format, Doconce tries to find another file, and if not, the given file is converted to a proper format (using ImageMagick's `convert` utility).

The height, width, and scale keywords (and others) can be included if desired and may have effect for some formats. Note the comma between the sespecifications and that there should be no space around the = sign.

Note also that, like for `TITLE:` and `AUTHOR:` lines, all information related to a figure line must be written on the same line. Introducing newlines in a long caption will destroy the formatting (only the part of the caption appearing on the same line as `FIGURE:` will be included in the formatted caption).

Figure 1: It can't get worse than this... (fig:impact)

Movies. Here is an example on the `MOVIE:` keyword for embedding movies. This feature works only for the `LaTeX` and `HTML` formats:

```
MOVIE: [filename, height=xxx width=yyy] possible caption
```

MOVIE: [figs/demo.mpeg, width=600, height=470]

The LaTeX format results in a file that requires the movie15 package in order to play movies in PDF via Acroread. The HTML format will play the movie right away, while for all other formats there is no movie support. The HTML format can also treat filenames of the form `myframes*.png`. In that case, a player for showing the sequence of frames is inserted in the HTML file.

Computer Code. Another type of special lines starts with `@@@CODE` and enables copying of computer code from a file directly into a verbatim environment, see the section Blocks of Verbatim Computer Code below.

Inline Tagging

Doconce supports tags for *emphasized phrases*, **boldface phrases**, and `verbatim text` (also called type writer text, for inline code) plus LaTeX/TeX inline mathematics, such as $v = \sin(x)$.

Emphasized text is typeset inside a pair of asterisk, and there should be no spaces between an asterisk and the emphasized text, as in:

```
*emphasized words*
```

Boldface font is recognized by an underscore instead of an asterisk:

```
_several words in boldface_ followed by *ephasized text*.
```

The line above gets typeset as **several words in boldface** followed by *ephasized text*.

Verbatim text, typically used for short inline code, is typeset between backquotes:

```
‘call myroutine(a, b)‘ looks like a Fortran call
while ‘void myfunc(double *a, double *b)‘ must be C.
```

The typesetting result looks like this: `call myroutine(a, b)` looks like a Fortran call while `void myfunc(double *a, double *b)` must be C.

It is recommended to have inline verbatim text on the same line in the Doconce file, because some formats (LaTeX and `ptex2tex`) will have problems with inline verbatim text that is split over two lines.

Watch out for mixing backquotes and asterisk (i.e., verbatim and emphasized code): the Doconce interpreter is not very smart so inline computer code can soon lead to problems in the final format. Go back to the Doconce source and modify it so the format to which you want to go becomes correct (sometimes a trial and error process - sticking to very simple formatting usually avoids such problems).

Web addresses with links are typeset as:

```
some URL like "MyPlace": "http://my.place.in.space/src"
```

which appears as some URL like MyPlace. The space after colon is optional. Link to a file is done by the URL keyword, a colon, and enclosing the filename in double quotes:

```
URL:"manual.do.txt"
"URL": "manual.do.txt"
url: "manual.do.txt"
"url":"manual.do.txt"
```

All these constructions result in the link manual.do.txt. To make the URL itself appear as link name, put an “URL”, URL, or the lower case version, before the text of the URL enclosed in double quotes:

```
Click on this link: URL:"http://some.where.net".
```

Doconce also supports inline comments in the text:

```
[name: comment]
```

where `name` is the name of the author of the command, and `comment` is a plain text text. (**hpl**: Note that there must be a space after the colon, otherwise the comment is not recognized.) The name and comment are visible in the output unless `doconce format` is run with a command-line specification of removing such comments (see the chapter From Doconce to Other Formats for an example). Inline comments (**hpl**: Here is a specific example on an inline comment. It can span several lines.) are helpful during development of a document since different authors and readers can comment on formulations, missing points, etc. All such comments can easily be removed from the `.do.txt` file (see the chapter From Doconce to Other Formats).

Inline mathematics is written as in LaTeX, i.e., inside dollar signs. Most formats leave this syntax as it is (including to dollar signs), hence nice math formatting is only obtained in LaTeX (Epytext has some inline math support that is utilized). However, mathematical expressions in LaTeX syntax often contains special formatting commands, which may appear annoying in plain text. Doconce therefore supports an extended inline math syntax where the writer can provide an alternative syntax suited for formats close to plain ASCII:

```
Here is an example on a linear system
${\bf A}{\bf x} = {\bf b}$|$Ax=b$,
where $\bf A$|$A$ is an $n\times n$|$nxn$ matrix, and
$\bf x$|$x$ and $\bf b$|$b$ are vectors of length $n$|$n$.
```

That is, we provide two alternative expressions, both enclosed in dollar signs and separated by a pipe symbol, the expression to the left is used in LaTeX, while the expression to the right is used for all other formats. The above text is typeset as "Here is an example on a linear system Ax=b, where A is an nxn matrix, and x and b are vectors of length n."

Cross-Referencing

References and labels are supported. The syntax is simple:

```
label{section:verbatim}   # defines a label
For more information we refer to Section ref{section:verbatim}.
```

This syntax is close that that of labels and cross-references in LaTeX. When the label is placed after a section or subsection heading, the plain text, Epytext, and StructuredText formats will simply replace the reference by the title of the (sub)section. All labels will become invisible, except those in math environments. In the reStructuredText and Sphinx formats, the end effect is the same, but the "label" and "ref" commands are first translated to the proper reStructuredText commands by `doconce format`. In the HTML and (Google Code) Wiki formats, labels become anchors and references become links, and with LaTeX "label" and "ref" are just equipped with backslashes so these commands work as usual in LaTeX.

It is, in general, recommended to use labels and references for (sub)sections, equations, and figures only. By the way, here is an example on referencing Figure fig:impact (the label appears in the figure caption in the source code of this document). Additional references to the sections LaTeX Blocks of Mathematical Text and Macros (Newcommands) are nice to demonstrate, as well as a reference to

equations, say Equation (my:eq1)--Equation (my:eq2). A comparison of the output and the source of this document illustrates how labels and references are handled by the format in question.

Hyperlinks to files or web addresses are handled as explained in the section Inline Tagging.

Index and Bibliography

An index can be created for the LaTeX and the reStructuredText or Sphinx formats by the `idx` keyword, following a LaTeX-inspired syntax:

```
idx{some index entry}
idx{main entry!subentry}
idx{‘verbatim_text‘ and more}
```

The exclamation mark divides a main entry and a subentry. Backquotes surround verbatim text, which is correctly transformed in a LaTeX setting to:

```
\index{verbatim\_text@\texttt{\rm\smaller verbatim\_text and more}}
```

Everything related to the index simply becomes invisible in plain text, Epytext, StructuredText, HTML, and Wiki formats. Note: `idx` commands should be inserted outside paragraphs, not in between the text as this may cause some strange behaviour of the formatting. Index items are naturally placed right after section headings, before the text begins. Index items related to the heading of a paragraph, however, should be placed above the paragraph heading and not in between the heading and the text.

Literature citations also follow a LaTeX-inspired style:

```
as found in cite{Larsen:86,Nielsen:99}.
```

Citation labels can be separated by comma. In LaTeX, this is directly translated to the corresponding `cite` command; in reStructuredText and Sphinx the labels can be clicked, while in all the other text formats the labels are consecutively numbered so the above citation will typically look like:

```
as found in [3][14]
```

if `Larsen:86` has already appeared in the 3rd citation in the document and `Nielsen:99` is a new (the 14th) citation. The citation labels can be any sequence of characters, except for curly braces and comma.

The bibliography itself is specified by the special keyword `BIBFILE:`, which is optionally followed by a BibTeX file, having extension `.bib`, a corresponding reStructuredText bibliography, having extension `.rst`, or simply a Python dictionary written in a file with extension `.py`. The dictionary in the latter file should have the citation labels as keys, with corresponding values as the full reference text for an item in the bibliography. Doconce markup can be used in this text, e.g.:

```
{
'Nielsen:99': """
K. Nielsen. *Some Comments on Markup Languages*.
URL:"http://some.where.net/nielsen/comments", 1999.
```

```
"""',
'Larsen:86':
"""
O. B. Larsen. On Markup and Generality.
Personal Press*. 1986.
"""
}
```

In the LaTeX format, the `.bib` file will be used in the standard way, in the reStructuredText and Sphinx formats, the `.rst` file will be copied into the document at the place where the `BIBFILE:` keyword appears, while all other formats will make use of the Python dictionary typeset as an ordered Doconce list, replacing the `BIBFILE:` line in the document.

Finally, we must test the citation command and bibliography by citing a book [Python:Primer:09], a paper [Osnes:98], and both of them simultaneously [Python:Primer:09] [Osnes:98].

(**somereader**: comments, citations, and references in the latex style is a special feature of doconce :-))

Tables

A table like

time	velocity	acceleration
0.0	1.4186	-5.01
2.0	1.376512	11.919
4.0	1.1E+1	14.717624

is built up of pipe symbols and dashes:

```
|--------------------------------|
|time  | velocity | acceleration |
|--------------------------------|
| 0.0  | 1.4186   | -5.01        |
| 2.0  | 1.376512 | 11.919       |
| 4.0  | 1.1E+1   | 14.717624    |
|--------------------------------|
```

The pipes and column values do not need to be aligned (but why write the Doconce source in an ugly way?).

Blocks of Verbatim Computer Code

Blocks of computer code, to be typeset verbatim, must appear inside a "begin code" `!bc` keyword and an "end code" `!ec` keyword. Both keywords must be on a single line and *start at the beginning of the line*. There may be an argument after the `!bc` tag to specify a certain `ptex2tex` environment (for instance, `!bc dat` corresponds to the data file environment in `ptex2tex`, and `!bc cod` is typically used for a code snippet, but any argument can be defined). If there is no

argument, one assumes the ccq environment, which is plain LaTeX verbatim in the default `.ptex2tex.cfg`. However, all these arguments can be redefined in the `.ptex2tex.cfg` file.

The argument after `!bc` is also used in a Sphinx context. Then argument is mapped onto a valid Pygments language for typesetting of the verbatim block by Pygments. This mapping takes place in an optional comment to be inserted in the Doconce source file, e.g.:

```
# sphinx code-blocks: pycod=python cod=py cppcod=c++ sys=console
```

Here, three arguments are defined: `pycod` for Python code, `cod` also for Python code, `cppcod` for C++ code, and `sys` for terminal sessions. The same arguments would be defined in `.ptex2tex.cfg` for how to typeset the blocks in LaTeX using various verbatim styles (Pygments can also be used in a LaTeX context).

By default, `pro` is used for complete programs in Python, `cod` is for a code snippet in Python, while `xcod` and `xpro` implies computer language specific typesetting where `x` can be `f` for Fortran, `c` for C, `cpp` for C++, and `py` for Python. The argument `sys` means by default `console` for Sphinx and `CodeTerminal` (ptex2tex environent) for LaTeX. All these definitions of the arguments after `!bc` can be redefined in the `.ptex2tex.cfg` configuration file for ptex2tex/LaTeX and in the `sphinx code-blocks` comments for Sphinx. Support for other languages is easily added.

The enclosing `!ec` tag of verbatim computer code blocks must be followed by a newline. A common error in list environments is to forget to indent the plain text surrounding the code blocks. In general, we recommend to use paragraph headings instead of list items in combination with code blocks (it usually looks better, and some common errors are naturally avoided).

Here is a verbatim code block with Python code (`pycod` style):

```
# regular expressions for inline tags:
inline_tag_begin = r'(?P<begin>(^|\s+))'
inline_tag_end = r'(?P<end>[.,?!;:)\s])'
INLINE_TAGS = {
    'emphasize':
    r'%s\*(?P<subst>[^ `][^*`]*)\*%s' % \
    (inline_tag_begin, inline_tag_end),
    'verbatim':
    r'%s`(?P<subst>[^ ][^`]*)`%s' % \
    (inline_tag_begin, inline_tag_end),
    'bold':
    r'%s_(?P<subst>[^ `][^_`]*)_%s' % \
    (inline_tag_begin, inline_tag_end),
}
```

And here is a C++ code snippet (`cppcod` style):

```
void myfunc(double* x, const double& myarr) {
    for (int i = 1; i < myarr.size(); i++) {
        myarr[i] = myarr[i] - x[i]*myarr[i-1]
    }
}
```

Computer code can be copied directly from a file, if desired. The syntax is then:

```
@@@CODE myfile.f
@@@CODE myfile.f fromto:subroutine\s+test@^C\s{5}END1
```

The first line implies that all lines in the file `myfile.f` are copied into a verbatim block, typset in a `!bc pro` environment. The second line has a *fromto:' directive, which implies copying code between two lines in the code, typset within a ¡bc cod* environment. (The `pro` and `cod` arguments are only used for LaTeX and Sphinx output, all other formats will have the code typeset within a plain `!bc` environment.) Two regular expressions, separated by the `@` sign, define the "from" and "to" lines. The "from" line is included in the verbatim block, while the "to" line is not. In the example above, we copy code from the line matching `subroutine test` (with as many blanks as desired between the two words) and the line matching `C END1` (C followed by 5 blanks and then the text END1). The final line with the "to" text is not included in the verbatim block.

Let us copy a whole file (the first line above):

```
C     a comment

      subroutine    test()
      integer i
      real*8 r
      r = 0
      do i = 1, i
         r = r + i
      end do
      return
C     END1

      program testme
      call test()
      return
```

Let us then copy just a piece in the middle as indicated by the `fromto:` directive above:

```
subroutine    test()
integer i
real*8 r
r = 0
do i = 1, i
   r = r + i
end do
return
```

(Remark for those familiar with `ptex2tex`: The from-to syntax is slightly different from that used in `ptex2tex`. When transforming Doconce to LaTeX, one first transforms the document to a `.p.tex` file to be treated by `ptex2tex`. However, the `@@@CODE` line is interpreted by Doconce and replaced by a *pro* or *cod* `ptex2tex` environment.)

LaTeX Blocks of Mathematical Text

Blocks of mathematical text are like computer code blocks, but the opening tag is `!bt` (begin TeX) and the closing tag is `!et`. It is important that `!bt` and `!et` appear on the beginning of the line and followed by a newline.

Here is the result of a `!bt` - `!et` block:

```
\begin{eqnarray}
{\partial u\over\partial t} &=& \nabla^2 u + f,\label{myeq1}\\
{\partial v\over\partial t} &=& \nabla\cdot(q(u)\nabla v) + g
\end{eqnarray}
```

This text looks ugly in all Doconce supported formats, except from LaTeX and Sphinx. If HTML is desired, the best is to filter the Doconce text first to LaTeX and then use the widely available tex4ht tool to convert the dvi file to HTML, or one could just link a PDF file (made from LaTeX) directly from HTML. For other textual formats, it is best to avoid blocks of mathematics and instead use inline mathematics where it is possible to write expressions both in native LaTeX format (so it looks good in LaTeX) and in a pure text format (so it looks okay in other formats).

Macros (Newcommands)

Doconce supports a type of macros via a LaTeX-style *newcommand* construction. The newcommands defined in a file with name `newcommand_replace.tex` are expanded when Doconce is filtered to other formats, except for LaTeX (since LaTeX performs the expansion itself). Newcommands in files with names `newcommands.tex` and `newcommands_keep.tex` are kept unaltered when Doconce text is filtered to other formats, except for the Sphinx format. Since Sphinx understands LaTeX math, but not newcommands if the Sphinx output is HTML, it makes most sense to expand all newcommands. Normally, a user will put all newcommands that appear in math blocks surrounded by `!bt` and `!et` in `newcommands_keep.tex` to keep them unchanged, at least if they contribute to make the raw LaTeX math text easier to read in the formats that cannot render LaTeX. Newcommands used elsewhere throughout the text will usually be placed in `newcommands_replace.tex` and expanded by Doconce. The definitions of newcommands in the `newcommands*.tex` files *must* appear on a single line (multi-line newcommands are too hard to parse with regular expressions).

Example. Suppose we have the following commands in `newcommand_replace.tex`:

```
\newcommand{\beqa}{\begin{eqnarray}}
\newcommand{\eeqa}{\end{eqnarray}}
\newcommand{\ep}{\thinspace . }
\newcommand{\uvec}{\vec u}
\newcommand{\mathbfx}[1]{{\mbox{\boldmath $#1$}}}
\newcommand{\Q}{\mathbfx{Q}}
```

and these in `newcommands_keep.tex`:

```
\newcommand{\x}{\mathbfx{x}}
\newcommand{\normalvec}{\mathbfx{n}}
\newcommand{\Ddt}[1]{\frac{D#1}{dt}}
```

The LaTeX block:

```
\beqa
\x\cdot\normalvec &=& 0,\label{my:eq1}\\
\Ddt{\uvec} &=& \Q \ep\label{my:eq2}
\eeqa
```

will then be rendered to:

```
\begin{eqnarray}
\x\cdot\normalvec &=& 0,\label{my:eq1}\\
\Ddt{\vec u} &=& {\mbox{\boldmath $Q$}} \thinspace . \label{my:eq2}
\end{eqnarray}
```

in the current format.

Preprocessing Steps

Doconce allows preprocessor commands for, e.g., including files, leaving out text, or inserting special text depending on the format. Two preprocessors are supported: Preprocess (http://code.google.com/p/preprocess) and Mako (http://www.makotemplates.org/). The former allows include and if-else statements much like the well-known preprocessor in C and C++ (but it does not allow sophisticated macro substitutions). The latter preprocessor is a very powerful template system. With Mako you can automatically generate various type of text and steer the generation through Python code embedded in the Doconce document. An arbitrary set of `name=value` command-line arguments (at the end of the command line) automatically define Mako variables that are substituted in the document.

Doconce will detect if Preprocess or Mako commands are used and run the relevant preprocessor prior to translating the Doconce source to a specific format.

Preprocess and Mako always have the variable `FORMAT` to be the desired output format of Doconce. It is then easy to test on the value of `FORMAT` and take different actions for different formats. For example, one may create special LaTeX output for figures, say with multiple plots within a figure, while other formats may apply a separate figure for each plot.

Missing Features

- Footnotes

Troubleshooting

Disclaimer. First of all, Doconce has hardly any support for syntax checking. This means that if you encounter Python errors while running `doconce format`, the reason for the error is most likely a syntax problem in your Doconce source file. You have to track down this syntax problem yourself.

However, the problem may well be a bug in Doconce. The Doconce software is incomplete, and many special cases of syntax are not yet discovered to give problems. Such special cases are also seldom easy to fix, so one important way of "debugging" Doconce is simply to change the formatting so that Doconce

treats it properly. Doconce is very much based on regular expressions, which are known to be non-trivial to debug years after they are created. The main developer of Doconce has hardly any time to work on debugging the code, but the software works well for his diverse applications of it.

Code or TeX Block Errors in reST. Sometimes reStructuredText (reST) reports an "Unexpected indentation" at the beginning of a code block. If you see a `!bc`, which should have been removed by `doconce format`, it is usually an error in the Doconce source, or a problem with the rst/sphinx translator. Check if the line before the code block ends in one colon (not two!), a question mark, an exclamation mark, a comma, a period, or just a newline/space after text. If not, make sure that the ending is among the mentioned. Then `!bc` will most likely be replaced and a double colon at the preceding line will appear (which is the right way in reST to indicate a verbatim block of text).

Strange Errors Around Code or TeX Blocks in reST. If `idx` commands for defining indices are placed inside paragraphs, and especially right before a code block, the reST translator (rst and sphinx formats) may get confused and produce strange code blocks that cause errors when the reST text is transformed to other formats. The remedy is to define items for the index outside paragraphs.

Error Message "Undefined substitution..." from reST. This may happen if there is much inline math in the text. reST cannot understand inline LaTeX commands and interprets them as illegal code. Just ignore these error messages.

Preprocessor Directives Do Not Work. Make sure the preprocessor instructions, in Preprocess or Mako, have correct syntax. Also make sure that you do not mix Preprocess and Mako instructions. Doconce will then only run Preprocess.

The LaTeX File Does Not Compile. If the problem is undefined control sequence involving:

```
\code{...}
```

the cause is usually a verbatim inline text (in backquotes in the Doconce file) spans more than one line. Make sure, in the Doconce source, that all inline verbatim text appears on the same line.

Verbatim Code Blocks Inside Lists Look Ugly. Read the the section Blocks of Verbatim Computer Code above. Start the `!bc` and `!ec` tags in column 1 of the file, and be careful with indenting the surrounding plain text of the list item correctly. If you cannot resolve the problem this way, get rid of the list and use paragraph headings instead. In fact, that is what is recommended: avoid verbatim code blocks inside lists (it makes life easier).

LaTeX Code Blocks Inside Lists Look Ugly. Same solution as for computer code blocks as described in the previous paragraph. Make sure the `!bt` and `!et` tags are in column 1 and that the rest of the non-LaTeX surrounding text is correctly indented. Using paragraphs instead of list items is a good idea also here.

Inconsistent Headings in reStructuredText. The `rst2*.py` and Sphinx converters abort if the headers of sections are not consistent, i.e., a subsection must come under a section, and a subsubsection must come under a subsection (you cannot have a subsubsection directly under a section). Search for `===`, count the number of equality signs (or underscores if you use that) and make sure they decrease by two every time a lower level is encountered.

Strange Nested Lists in gwiki. Doconce cannot handle nested lists correctly in the gwiki format. Use nonnested lists or edit the `.gwiki` file directly.

Lists in gwiki Look Ugly in the Sourc. Because the Google Code wiki format requires all text of a list item to be on one line, Doconce simply concatenates lines in that format, and because of the indentation in the original Doconce text, the gwiki output looks somewhat ugly. The good thing is that this gwiki source is seldom to be looked at - it is the Doconce source that one edits further.

Problems with Boldface and Emphasize. Two boldface or emphasize expressions after each other are not rendered correctly. Merge them into one common expression.

Strange Non-English Characters. Check the encoding of the `.do.txt` file with the Unix `file` command. If UTF-8, convert to latin-1 using the Unix command:

```
Unix> iconv -f utf-8 -t LATIN1 myfile.do.txt --output newfile
```

(Doconce has a feature to detect the encoding, but it is not reliable and therefore turned off.)

Debugging. Given a problem, extract a small portion of text surrounding the problematic area and debug that small piece of text. Doconce does a series of transformations of the text. The effect of each of these transformation steps are dumped to a logfile, named `_doconce_debugging.log`, if the to `doconce format` after the filename is `debug`. The logfile is inteded for the developers of Doconce, but may still give some idea of what is wrong. The section "Basic Parsing Ideas" explains how the Doconce text is transformed into a specific format, and you need to know these steps to make use of the logfile.

Header and Footer

Some formats use a header and footer in the document. LaTeX and HTML are two examples of such formats. When the document is to be included in another document (which is often the case with Doconce-based documents), the header and footer are not wanted, while these are needed (at least in a LaTeX context) if the document is stand-alone. We have introduce the convention that if `TITLE:` or `#TITLE:` is found at the beginning of the line (i.e., the document has, or has an intention have, a title), the header and footer are included, otherwise not.

Basic Parsing Ideas

The (parts of) files with computer code to be directly included in the document are first copied into verbatim blocks.

All verbatim and TeX blocks are removed and stored elsewhere to ensure that no formatting rules are not applied to these blocks.

The text is examined line by line for typesetting of lists, as well as handling of blank lines and comment lines. List parsing needs some awareness of the context. Each line is interpreted by a regular expression:

```
(?P<indent> *(?P<listtype>[*o-] )? *)(?P<keyword>[^:]+?:)?(?P<text>.*)\s?
```

That is, a possible indent (which we measure), an optional list item identifier, optional space, optional words ended by colon, and optional text. All lines are of

this form. However, some ordinary (non-list) lines may contain a colon, and then the keyword and text group must be added to get the line contents. Otherwise, the text group will be the line.

When lists are typeset, the text is examined for sections, paragraphs, title, author, date, plus all the inline tags for emphasized, boldface, and verbatim text. Plain subsitutions based on regular expressions are used for this purpose.

The final step is to insert the code and TeX blocks again (these should be untouched and are therefore left out of the previous parsing).

It is important to keep the Doconce format and parsing simple. When a new format is needed and this format is not obtained by a simple edit of the definition of existing formats, it might be better to convert the document to reStructuredText and then to XML, parse the XML and write out in the new format. When the Doconce format is not sufficient to getting the layout you want, it is suggested to filter the document to another, more complex format, say reStructuredText or LaTeX, and work further on the document in this format.

A Glimpse of How to Write a New Translator

This is the HTML-specific part of the source code of the HTML translator:

```
FILENAME_EXTENSION['HTML'] = '.html'  # output file extension
BLANKLINE['HTML'] = '<p>\n'           # blank input line => new paragraph
INLINE_TAGS_SUBST['HTML'] = {         # from inline tags to HTML tags
    # keep math as is:
    'math': None,  # indicates no substitution
    'emphasize':     r'\g<begin><em>\g<subst></em>\g<end>',
    'bold':          r'\g<begin><b>\g<subst></b>\g<end>',
    'verbatim':      r'\g<begin><tt>\g<subst></tt>\g<end>',
    'URL':           r'\g<begin><a href="\g<url>">\g<link></a>',
    'section':       r'<h1>\g<subst></h1>',
    'subsection':    r'<h3>\g<subst></h3>',
    'subsubsection': r'<h5>\g<subst></h5>',
    'paragraph':     r'<b>\g<subst></b>. ',
    'title':         r'<title>\g<subst></title>\n<center><h1>\g<subst></h1></center>'
    'date':          r'<center><h3>\g<subst></h3></center>',
    'author':        r'<center><h3>\g<subst></h3></center>',
    }

# how to replace code and LaTeX blocks by HTML (<pre>) environment:
def HTML_code(filestr):
    c = re.compile(r'^!bc(.*?)\n', re.MULTILINE)
    filestr = c.sub(r'<!-- BEGIN VERBATIM BLOCK \g<1>-->\n<pre>\n', filestr)
    filestr = re.sub(r'!ec\n',
                     r'</pre>\n<! -- END VERBATIM BLOCK -->\n', filestr)
    c = re.compile(r'^!bt\n', re.MULTILINE)
    filestr = c.sub(r'<pre>\n', filestr)
    filestr = re.sub(r'!et\n', r'</pre>\n', filestr)
    return filestr
CODE['HTML'] = HTML_code
```

```
# how to typeset lists and their items in HTML:
LIST['HTML'] = {
    'itemize':
    {'begin': '\n<ul>\n', 'item': '<li>', 'end': '</ul>\n\n'},
    'enumerate':
    {'begin': '\n<ol>\n', 'item': '<li>', 'end': '</ol>\n\n'},
    'description':
    {'begin': '\n<dl>\n', 'item': '<dt>%s<dd>', 'end': '</dl>\n\n'},
    }

# how to type set description lists for function arguments, return
# values, and module/class variables:
ARGLIST['HTML'] = {
    'parameter': '<b>argument</b>',
    'keyword': '<b>keyword argument</b>',
    'return': '<b>return value(s)</b>',
    'instance variable': '<b>instance variable</b>',
    'class variable': '<b>class variable</b>',
    'module variable': '<b>module variable</b>',
    }

# document start:
INTRO['HTML'] = """
<html>
<body bgcolor="white">
"""
# document ending:
OUTRO['HTML'] = """
</body>
</html>
"""
```

Typesetting of Function Arguments, Return Values, and Variables

As part of comments (or doc strings) in computer code one often wishes to explain what a function takes of arguments and what the return values are. Similarly, it is desired to document class, instance, and module variables. Such arguments/variables can be typeset as description lists of the form listed below and *placed at the end of the doc string*. Note that `argument`, `keyword argument`, `return`, `instance variable`, `class variable`, and `module variable` are the only legal keywords (descriptions) for the description list in this context. If the output format is Epytext (Epydoc) or Sphinx, such lists of arguments and variables are nicely formatted:

```
- argument x: x value (float),
  which must be a positive number.
- keyword argument tolerance: tolerance (float) for stopping
  the iterations.
```

```
- return: the root of the equation (float), if found, otherwise None.
- instance variable eta: surface elevation (array).
- class variable items: the total number of MyClass objects (int).
- module variable debug: True: debug mode is on; False: no debugging
  (bool variable).
```

The result depends on the output format: all formats except Epytext and Sphinx just typeset the list as a list with keywords.

module variable x: x value (float), which must be a positive number.

module variable tolerance: tolerance (float) for stopping the iterations.

[Python:Primer:09] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Springer, 2009.

[Osnes:98] H. Osnes and H. P. Langtangen. An efficient probabilistic finite element method for stochastic groundwater flow. *Advances in Water Resources*, vol 22, 185-195, 1998.