

10

Documenting Your Project

Documentation is work that is often neglected by developers and sometimes by managers. This is often due to a lack of time towards the end of development cycles, and the fact that people think they are bad at writing. Some of them are bad, but the majority of them are able to produce fine documentation.

In any case, the result is a disorganized documentation made of documents that are written in a rush. Developers hate doing this kind of work most of the time. Things get even worse when existing documents need to be updated. Many projects out there are just providing poor, out-of-date documentation because the manager does not know how to deal with it.

But setting up a documentation process at the beginning of the project and treating documents as if they were modules of code makes documenting easier. Writing can even be fun when a few rules are followed.

This chapter provides a few tips to start documenting your project through:

- The seven rules of technical writing that summarize the best practices
- A reStructuredText primer, which is a plain text markup syntax used in most Python projects
- A guide for building good project documentation

The Seven Rules of Technical Writing

Writing good documentation is easier in many aspects than writing a code. Most developers think it is very hard, but by following a simple set of rules it becomes really easy.

We are not talking here about writing a book of poems but a comprehensive piece of text that can be used to understand a design, an API, or anything that makes up the code base.

Every developer is able to produce such material, and this section provides seven rules that can be applied in all cases.

- **Write in two steps:** Focus on ideas, and then on reviewing and shaping your text.
- **Target the readership:** Who is going to read it?
- **Use a simple style:** Keep it straight and simple. Use good grammar.
- **Limit the scope of the information:** Introduce one concept at a time.
- **Use realistic code examples:** Foos and bars should be dropped.
- **Use a light but sufficient approach:** You are not writing a book!
- **Use templates:** Help the readers to get habits.

These rules are mostly inspired and adapted from *Agile Documenting*, a book by Andreas Rüping that focuses on producing the best documentation in software projects.

Write in Two Steps

Peter Elbow, in *Writing with Power*, explains that it is almost impossible for any human being to produce a perfect text in one shot. The problem is that many developers write documentation and try to directly come up with a perfect text. The only way they succeed in this exercise is by stopping the writing after every two sentences to read them back, and do some corrections. This means that they are focusing both on the content and the style of the text.

This is too hard for the brain and the result is often not as good as it could be. A lot of time and energy is spent in polishing the style and shape of the text, before its meaning is completely thought through.

Another approach is to drop the style and organization of the text and focus on its content. All ideas are laid down on paper, no matter how they are written. The developer starts to write a continuous stream and does not pause when he or she makes grammatical mistakes, or for anything that is not about the content. For instance, it does not matter if the sentences are barely understandable as long as the ideas are written down. He or she just writes down what he wants to say, with a rough organization.

By doing this, the developer focuses on what he or she wants to say and will probably get more content out of his or her brain than he or she initially thought he or she would.

Another side-effect when doing free writing is that other ideas that are not directly related to the topic will easily go through the mind. A good practice is to write them down on a second paper or screen when they appear, so they are not lost, and then get back to the main writing.

The second step consists of reading back the whole text and polishing it so that it is comprehensible to everyone. Polishing a text means enhancing its style, correcting its faults, reorganizing it a bit, and removing any redundant information it has.

When the time dedicated to write documentation is limited, a good practice is to cut this time in two equal durations – one for writing the content, and one to clean and organize the text.



Focus on the content, and then on style and cleanliness.

Target the Readership

When starting a text, there is a simple question the writer should consider: *Who is going to read it?*

This is not always obvious, as a technical text explains how a piece of software works, and is often written for every person who might get and use the code. The reader can be a manager who is looking for an appropriate technical solution to a problem, or a developer who needs to implement a feature with it. A designer might also read it to know if the package fits his or her needs from an architectural point of view.

Let's apply a simple rule: Each text should have only one kind of readers.

This philosophy makes the writing easier. The writer precisely knows what kind of reader he or she is dealing with. He or she can provide a concise and precise documentation that is not vaguely intended for all kinds of readers.

A good practice is to provide a small introductory text that explains in one sentence what the documentation is about, and guides the reader to the appropriate part:

```
Atomisator is a product that fetches RSS feeds and saves them in a
database, with a filtering process.
If you are a developer, you might want to look at the API description
(api.txt)
If you are a manager, you can read the features list and the FAQ
(features.txt)
If you are a designer, you can read the architecture and
infrastructure notes (arch.txt)
```

By taking care of directing your readers in this way, you will probably produce better documentation.



Know your readership before you start to write.

Use a Simple Style

Seth Godin is one of the best-selling writers on marketing topics. You might want to read *Unleashing the Ideavirus*, which is available for free on the Internet (http://en.wikipedia.org/wiki/Unleashing_the_Ideavirus).

Lately, he made an analysis on his blog to try to understand why his books sold so well. He made a list of all best sellers in the marketing area and compared the average number of words per sentences in each one of them.

He realized that his books had the lowest number of words per sentence (thirteen words). This simple fact, Seth explained, proved that readers prefer short and simple sentences, rather than long and stylish ones.

By keeping sentences short and simple, your writings will consume less brain power for their content to be extracted, processed, and then understood. Writing technical documentation aims to provide a software guide to readers. It is not a fiction story, and should be closer to your microwave notice than to the latest Stephen King novel.

A few tips to keep in mind are:

- Use simple sentences; they should not be longer than two lines.
- Each paragraph should be composed of three or four sentences, at the most, that express one main idea. Let your text breathe.
- Don't repeat yourself too much: Avoid journalistic styles where ideas are repeated again and again to make sure they are understood.
- Don't use several tenses. Present tense is enough most of the time.
- Do not make jokes in the text if you are not a really fine writer. Being funny in a technical book is really hard, and few writers master it. If you really want to distill some humor, keep it in code examples and you will be fine.



You are not writing fiction, so keep the style as simple as possible.

Limit the Scope of the Information

There's a simple sign of bad documentation in a software: You are looking for some information that you know is present somewhere, but you cannot find it. After spending some time reading the table of contents, you are starting to grep the files trying several word combinations, but cannot get what you are looking for.

This happens when writers are not organizing their texts in topics. They might provide tons of information, but it is just gathered in a monolithic or non-logical way. For instance, if a reader is looking for a big picture of your application, he or she should not have to read the API documentation: that is a low-level matter.

To avoid this effect, paragraphs should be gathered under a meaningful title for a given section, and the global document title should synthesize the content in a short phrase.

A table of contents could be made of all the section's titles.

A simple practice to compose your titles is to ask yourself: What phrase would I type in Google to find this section?

Use Realistic Code Examples

Foo and bar are bad citizens. When a reader tries to understand how a piece of code works with a usage example, having an unrealistic example will make it harder to understand.

Why not use a real-world example? A common practice is to make sure that each code example can be cut and pasted in a real program.

An example of bad usage is:

We have a parse function:

```
>>> from atomisator.parser import parse
```

Let's use it:

```
>>> stuff = parse('some-feed.xml')
>>> stuff.next()
{'title': 'foo', 'content': 'blabla'}
```

A better example would be when the parser knows how to return a feed content with the parse function, available as a top-level function:

```
>>> from atomisator.parser import parse
```

Let's use it:

```
>>> my_feed = parse('http://tarekziade.wordpress.com/feed')
>>> my_feed.next()
{'title': 'eight tips to start with python',
 'content': 'The first tip is..., ...'}
```

This slight difference might sound overkill, but in fact it makes your documentation a lot more useful. A reader can copy those lines into a shell, understands that `parse` uses a URL as a parameter, and that it returns an iterator that contains blog entries.



Code examples should be directly reusable in real programs.

Use a Light but Sufficient Approach

In most agile methodologies, documentation is not the first citizen. Making software that works is the most important thing, over detailed documentation. So a good practice, as Scott Ambler explains in his book *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*, is to define the real documentation needs, rather than creating an exhaustive set of documents.

For instance, a single document that explains how Atomisator works for administrators is sufficient. There is no other need for them than to know how to configure and run the tool. This document limits its scope to answer to one question: How do I run Atomisator on my server?

Besides readership and scope, limiting the size of each section written for the software to a few pages is a good idea. By making each section four pages long at the most, the writer will have to synthesize his or her thought. If it needs more, it probably means that the software is too complex to explain or use.



Working software over comprehensive documentation

The Agile Manifesto.

Use Templates

Every page on Wikipedia is similar. There are boxes on the left side that are used to summarize dates or facts. At the beginning of the document is a table of contents with links that refer to anchors in the same text. There is always a reference section at the end.

Users get used to it. For instance, they know they can have a quick look at the table of contents, and if they do not find the info they are looking for, they will go directly to the reference section to see if they can find another website on the topic. This works for any page on Wikipedia. You learn the *Wikipedia way* to be more efficient.

So using templates forces a common pattern for documents, and therefore makes people more efficient in using them. They get used to the structure and know how to read it quickly.

Providing a template for each kind of document also provides a quick start for writers.

In this chapter, we will see the various kinds of documents a piece of software can have, and use Paster to provide skeletons for them. But the first thing to do is to describe the markup syntax that should be used in Python documentation.

A reStructuredText Primer

reStructuredText is also called reST (see <http://docutils.sourceforge.net/rst.html>). It is a plain text markup language widely used in the Python community to document packages. The great thing about reST is that the text is still readable since the markup syntax does not obfuscate the text like LaTeX would.

Here's a sample of such a document:

```
=====  
Title  
=====  
  
Section 1  
=====  
  
This word has emphasis.  
  
Section 2  
=====  
  
Subsection  
:~::~:~::~:~::~:~::~:~::~:  
  
Text.
```

reST comes in `docutils`, a package that provides a suite of scripts to transform a reST file to various formats, such as HTML, LaTeX, XML, or even S5, Eric Meyer's slide show system (see <http://meyerweb.com/eric/tools/s5>).

Writers can focus on the content and then decide how to render it, depending on the needs. For instance, Python itself is documented into reST, which is then rendered in HTML to build <http://docs.python.org>, and in various other formats.

The minimum elements one should know to start writing reST are:

- Section structure
- Lists
- Inline markup
- Literal block
- Links

This section is a really fast overview of the syntax. A quick reference is available for more information at: <http://docutils.sourceforge.net/docs/user/rst/quickref.html>, which is a good place to start working with reST.

To install reStructuredText, install `docutils`:

```
$ easy_install docutils
```

You will get a set of scripts starting with `rst2`, to be able to render reST in various formats.

For instance, the `rst2html` script will produce HTML output given an reST file:

```
$ more text.txt
Title
=====
content.
$ rst2html.py text.txt > text.html
$ more text.html
<?xml version="1.0" encoding="utf-8" ?>
...
<html ...>
<head>
...
</head>
<body>
<div class="document" id="title">
<h1 class="title">Title</h1>
<p>content.</p>
</div>
</body>
</html>
```

Section Structure

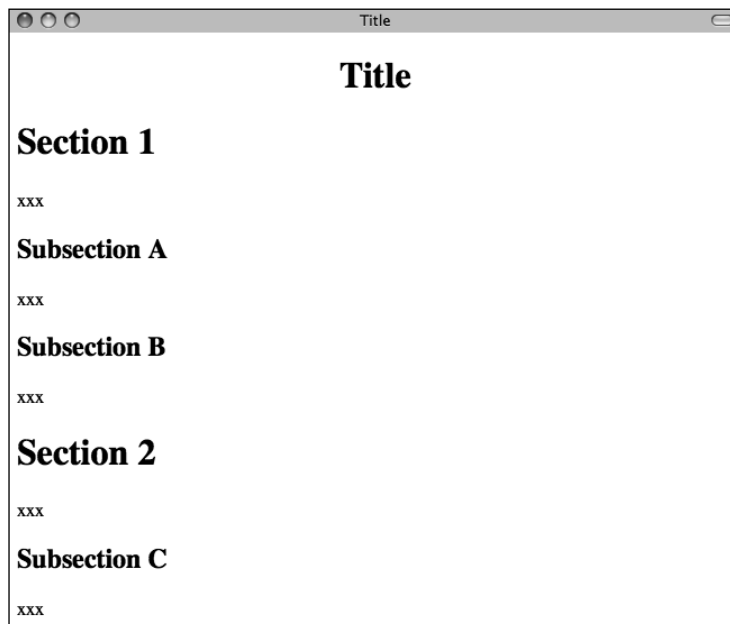
The document's title and its sections are underlined using non-alphanumeric characters. They can be overlined and underlined, and a common practice is to use this double markup for the title, and keep a simple underline for sections.

The most used characters to underline a section title are in the following order of precedence: `=`, `-`, `_`, `:`, `#`, `+`, `^`.

When a character is used for a section, it is associated with its level and it has to be used consistently throughout the document.

For example:

```
=====  
Title  
=====  
Section 1  
=====  
xxx  
Subsection A  
-----  
xxx  
Subsection B  
-----  
xxx  
Section 2  
=====  
xxx  
Subsection C  
-----  
xxx
```



The HTML output of this file will look like the illustration shown above.

Lists

reST provides bullet, and enumerated and definition lists with auto-enumeration features:

Bullet list:

- one
- two
- three

Enumerated list:

1. one
2. two
- #. auto-enumerated

Definition list:

- one
 one is a number.
- two
 two is also a number.

Inline Markup


Text can be styled using an inline markup:

- `*emphasis*`: Italics
- `**strong emphasis**`: Boldface
- ```inline literal```: Inline preformatted text
- ``a text with a link`_`: This will be replaced by a hyperlink as long as it is provided in the document (see in the Links section).

Literal Block

When you need to present some code examples, a literal block can be used. Two colons are used to mark the block, which is an indented paragraph:

```
This is a code example
::
    >>> 1 + 1
    2
Let's continue our text
```

 Don't forget to add a blank line after `::` and after the block, otherwise it will not be rendered.

Notice that the colon characters can be put in a text line. In that case, they will be replaced by a single colon in the various rendering formats:

```
This is a code example::
    >>> 1 + 1
    2
Let's continue our text
```

If you don't want to keep a single colon, you can insert a space between `example` and `::`. In that case, `::` will be interpreted and totally removed.

Links

A text can be changed into an external link with a special line starting with two dots, as long as it is provided in the document:

```
Try `Plone CMS`, it is great ! It is based on Zope_.
.. `_Plone CMS`: http://plone.org
.. `_Zope`: http://zope.org
```

A usual practice is to group the external links at the end of the document. When the text to be linked contains spaces, it has to be surrounded with ``` characters.

Internal links can also be used by adding a marker in the text:

```
This is a code example
.. _example:
::
    >>> 1 + 1
    2
Let's continue our text, or maybe go back to
the example_.
```

Sections are also targets that can be used:

```
====
Title
====
Section 1
=====
xxx
```

```
Subsection A
-----
xxx
Subsection B
-----
-> go back to `Subsection A`_
Section 2
=====
xxx
```

Building the Documentation

An easier way to guide your readers and your writers is to provide each one of them with helpers and guidelines, as we have learned in the previous section of this chapter.

From a writer's point of view, this is done by having a set of reusable templates together with a guide that describes how and when to use them in a project. It is called a **documentation portfolio**.

From a reader point of view, being able to browse the documentation with no pain, and getting used to finding the info efficiently, is done by building a **document landscape**.

Building the Portfolio

There are many kinds of documents a software project can have, from low-level documents that refer directly to the code, to design papers that provide a high-level overview of the application.

For instance, Scott Ambler defines an extensive list of document types in his book *Agile Modeling* (see <http://www.agilemodeling.com/essays/agileArchitecture.htm>). He builds a portfolio from early specifications to operations documents. Even the project management documents are covered, so the whole documenting needs are built with a standardized set of templates.

Since a complete portfolio is tightly related to the methodologies used to build the software, this chapter will only focus on a common subset that you can complete with your specific needs. Building an efficient portfolio takes a long time, as it captures your working habits.

A common set of documents in software projects can be classified in three categories:

- **Design:** All documents that provide architectural information, and low-level design information, such as class diagrams, or database diagrams
- **Usage:** Documents on how to use the software; this can be in the shape of a cookbook and tutorials, or a module-level help
- **Operations:** Provide guidelines on how to deploy, upgrade, or operate the software

Design

The purpose of design documentation is to describe how the software works and how the code is organized. It is used by developers to understand the system but is also a good entry point for people who are trying to understand how the application works.

The different kinds of design documents a software can have are:

- Architecture overview
- Database models
- Class diagrams with dependencies and hierarchy relations
- User interface wireframes
- Infrastructure description

Mostly, these documents are composed of some diagrams and a minimum amount of text. The conventions used for the diagrams are very specific to the team and the project, and this is perfectly fine as long as it is consistent.



UML provides thirteen diagrams that cover most aspects in a software design. The class diagram is probably the most used one, but it is possible to describe every aspect of software with it. See http://en.wikipedia.org/wiki/Unified_Modeling_Language#Diagrams.

Following a specific modeling language such as UML is not often fully done, and teams just make up their own way throughout their common experience. They pick up good practice from UML or other modeling languages, and create their own recipes.

For instance, for **architecture overview diagrams**, some designers just draw boxes and arrows on a whiteboard without following any particular design rules and take a picture of it. Others work with simple drawing programs such as Dia (<http://www.gnome.org/projects/dia>) or Microsoft Visio (not open source, so not free), since it is enough to understand the design. For example, all architecture diagrams presented in the *Chapter 6* of this book were made with OmniGraffle.

Database model diagrams depend on the kind of database you are using. There are complete data modeling software applications that provide drawing tools to automatically generate tables and their relations. But this is overkill in Python most of the time. If you are using an ORM such as SQLAlchemy (for instance), simple boxes with lists of fields, together with table relations as shown in *Chapter 6* are enough to describe your mappings before you start to write them.

Class diagrams are often simplified UML class diagrams: There is no need in Python to specify the protected members of a class, for instance. So the tools used for an architectural overview diagram fit this need too.

User interface diagrams depend on whether you are writing a web or a desktop application. Web applications often describe the center of the screen, since the header, footer, left, and right panels are common. Many web developers just handwrite those screens and capture them with a camera or a scanner. Others create prototypes in HTML and make screen snapshots. For desktop applications, snapshots on prototype screens, or annotated mock-ups made with tools such as Gimp or Photoshop are the most common way.

Infrastructure overview diagrams are like architecture diagrams, but they focus on how the software interacts with third-party elements, such as mail servers, databases, or any kind of data streams.

Common Template

The important point when creating such documents is to make sure the target readership is perfectly known, and the content scope is limited. So a generic template for design documents can provide a light structure with a little advice for the writer.

Such a structure can include:

- Title
- Author
- Tags (keywords)
- Description (abstract)
- Target (Who should read this?)
- Content (with diagrams)
- References to other documents

The content should be three or four screens (a 1024x768 average screen) at the most, to be sure to limit the scope. If it gets bigger, it should be split into several documents or summarized.

The template also provides the author's name and a list of tags to manage its evolutions and ease its classification. This will be covered later in the chapter.

Paster is the right tool to use to provide templates for documentation. `pbp.skels` implements the design template described, and can be used exactly like code generation. A target folder is provided and a few questions are answered:

```
$ paster create -t pbp_design_doc design
Selected and implied templates:
  pbp.skels#pbp_design_doc  A Design document

Variables:
  egg:      design
  package:  design
  project:  design

Enter title ['Title']: Database specifications for atomisator.db
Enter short_name ['recipe']: mappers
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: database mapping sql
Creating template pbp_design_doc
Creating directory ./design
  Copying +short_name+.txt_tmpl to ./design/mappers.txt
```

The result can then be completed:

```
=====
Database specifications for atomisator.db
=====

:Author: Tarek
:Tags: database mapping sql
:abstract:
    Write here a small abstract about your design document.
.. contents ::
Who should read this ?
:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:~::~:
Explain here who is the target readership.
Content
```

.....

Write your document here. Do not hesitate to split it in several sections.

References

.....

Put here references, and links to other documents.

Usage

Usage documentation describes how a particular part of the software works. This documentation can describe low-level parts such as how a function works, but also high-level parts such as command-line arguments for calling the program. This is the most important part of documentation in framework applications, since the target readership is mainly the developers that are going to reuse the code.

The three main kinds of documents are:

- **Recipe:** A short document that explains how to do something. This kind of document targets one readership and focuses on one specific topic.
- **Tutorial:** A step-by-step document that explains how to use a feature of the software. This document can refer to recipes, and each instance is intended to one readership.
- **Module helper:** A low-level document that explains what a module contains. This document could be shown (for instance) when you call the `help` built-in over a module.

Recipe

A recipe answers a very specific problem and provides a solution to resolve it.

For example, ActiveState provides a Python Cookbook online (a cookbook is a collection of recipes), where developers can describe how to do something in Python (see <http://aspn.activestate.com/ASPN/Python/Cookbook>).

These recipes must be short and are structured like this:

- Title
- Submitter
- Last updated
- Version
- Category
- Description

- Source (the source code)
- Discussion (the text explaining the code)
- Comments (from the web)

Often, they are one-screen long and do not go into great details. This structure perfectly fits a software's needs and can be adapted in a generic structure, where the target readership is added and the category replaced by tags:

- Title (short sentence)
- Author
- Tags (keywords)
- Who should read this?
- Prerequisites (other documents to read, for example)
- Problem (a short description)
- Solution (the main text, one or two screens)
- References (links to other documents)

The date and version are not useful here, since we will see later that the documentation is managed like source code in the project.

Like the `design` template, `pbp.skels` provide a `pbp_recipe_doc` template that can be used to generate this structure:

```
$ paster create -t pbp_recipe_doc recipes
Selected and implied templates:
  pbp.skels#pbp_recipe_doc  A recipe

Variables:
  egg:      recipes
  package:  recipes
  project:  recipes

Enter title (use a short question): How to use atomisator.db
Enter short_name ['recipe'] : atomisator-db
Enter author (Author name) ['John Doe']: Tarek
Enter keywords ['tag1 tag2']: atomisator db
Creating template pbp_recipe_doc
Creating directory ./recipes
  Copying +short_name+.txt_tmpl to ./recipes/atomisator-db.txt
```

The `pbp_tutorial_doc` template is provided in `pbp.skels` as well with this structure, which is similar to the design template.

Module Helper

The last template that can be added in our collection is the module helper template. A module helper refers to a single module and provides a description of its contents, together with usage examples.

Some tools can automatically build such documents by extracting the `docstrings` and computing module help using `pydoc`, like *Epydoc* (see <http://epydoc.sourceforge.net>). So it is possible to generate an extensive documentation based on API introspection. This kind of documentation is often provided in Python frameworks. For instance Plone provides an <http://api.plone.org> server that keeps an up-to-date collection of module helpers.

The main problems with this approach are:

- There is no smart selection performed over the modules that are really interesting to document.
- The code can be obfuscated by the documentation.

Furthermore, a module documentation provides examples that sometimes refer to several parts of the module, and are hard to split between the functions' and classes' `docstrings`. The module `docstring` could be used for that purpose by writing a text at the top of the module. But this ends in having a hybrid file composed of a block of text, then a block of code. This is rather obfuscating when the code represents less than 50% of the total length. If you are the author, this is perfectly fine. But when people try to read the code (not the documentation), they will have to jump the `docstrings` part.

Another approach is to separate the text in its own file. A manual selection can then be operated to decide which Python module will have its module helper file. The documents can then be separated from the code base and allowed to live their own life, as we will see in the next part. This is how Python is documented.

Many developers will disagree on the fact that doc and code separation is better than `docstrings`. This approach means that the documentation process is fully integrated in the development cycle; otherwise it will quickly become obsolete. The `docstrings` approach solves this problem by providing proximity between the code and its usage example, but doesn't bring it to a higher level: a document that can be used as part of a plain documentation.

The template for Module Helper is really simple, as it contains just a little metadata before the content is written. The target is not defined since it is the developers who wish to use the module:

- Title (module name)
- Author
- Tags (words)
- Content



The next chapter will cover Test-Driven Development using doctests and module helpers.

Operations

Operation documents are used to describe how the software can be operated. For instance:

- Installation and deployment documents
- Administration documents
- "Frequently Asked Questions" documents that help the users when a failure occurs
- Documents that explain how people can ask for help or provide feedback

These documents are very specific, but they can probably use the tutorial template defined in the earlier section.

Make Your Own Portfolio

The templates that we discussed earlier are just a basis that you can use to document your software. From there, as explained in the chapter dedicated to Paster, you can tune it and add other templates to build your own document portfolio.

Keep in mind the light but sufficient approach for project documentation: Each document added should have a clearly defined target readership and should fill a real need. Documents that don't add a real value should not be written.

Building the Landscape

The document portfolio built in the previous section provides a structure at document level, but does not provide a way to group and organize it to build the documentation the readers will have. This is what Andreas Rüping calls a document landscape, referring to the mental map the readers use when they browse documentation. He came up with the conclusion that the best way to organize documents is to build a logical tree.

In other words, the different kinds of documents composing the portfolio need to find a place to live within a tree of directories. This place must be obvious to the writers when they create the document and to the readers when they are looking for it.

A great helper in browsing documentation is index pages at each level that can drive writers and readers.

Building a document landscape is done in two steps:

- Building a tree for the producers (the writers)
- Building a tree for the consumers (the readers), on the top of the producers' one

This distinction between producers and consumers is important since they access the documents in different places and different formats.

Producer's Layout

From a producer's point of view, each document is processed exactly like a Python module. It should be stored in the version control system and worked like code.

Writers do not care about the final appearance of their prose and where it is available. They just want to make sure that they are writing a document, so it is the single source of truth on the topic covered.

reStructuredText files stored in a folder tree are available in the version control system together with the software code, and are a convenient solution to build the documentation landscape for producers.

If we look back at the folder structure presented in Chapter 6 for Atomisator, the `docs` folder can be used as the root of this tree.

The simplest way to organize the tree is to group documents by nature:

```
$ cd atomisator
$ find docs
docs
docs/source
docs/source/design
docs/source/operations
docs/source/usage
docs/source/usage/cookbook
docs/source/usage/modules
docs/source/usage/tutorial
```

Notice that the tree is located in a `source` folder because the `docs` folder will be used as a root folder to set up a special tool in the next section.

From there, an `index.txt` file can be added at each level (besides the root), explaining what kind of documents the folder contains, or summarizing what each sub-folder contains. These index files can define a listing of the documents they contain. For instance, the operation folder can contain a list of operations documents available:

```
=====
Operations
=====

This section contains operations documents:

- How to install and run Atomisator
- How to install and manage a PostgreSQL database
  for Atomisator
```

So that people do not forget to update them, we can have lists generated automatically.

Consumer's Layout

From a consumer's point of view, it is important to work out the index files and to present the whole documentation in a format that is easy to read and looks good. Web pages are the best pick and are easy to generate from reStructuredText files.

Sphinx (<http://sphinx.pocoo.org>) is a set of scripts and `docutils` extensions that can be used to generate an HTML structure from our text tree. This tool is used (for instance) to build the Python documentation, and many projects are now using it for their documentation. Among its built-in features, it produces a really nice browsing system, together with a light but sufficient client-side JavaScript search engine. It also uses `pygments` for rendering code examples, which produces really nice syntax highlights.

Sphinx can be easily configured to stick with the document landscape defined in the earlier section.

To install it, just call `easy_install`:

```
$ sudo easy_install-2.5 Sphinx
Searching for Sphinx
Reading http://cheeseshop.python.org/pypi/Sphinx/
...
Finished processing dependencies for Sphinx
```

This installs a few scripts such as `sphinx-quickstart`. This script will generate a script together with a `Makefile`, which can be used to generate the web documentation every time it is needed. Let's run this script in the `docs` folder and answer its questions:

```
$ sphinx-quickstart
Welcome to the Sphinx quickstart utility.

Enter the root path for documentation.
> Root path for the documentation [.] :
> Separate source and build directories (y/n) [n] : y
> Name prefix for templates and static dir [.] :
> Project name: Atomisator
> Author name(s): Tarek Ziadé
> Project version: 0.1.0
> Project release [0.1.0]:
> Source file suffix [.rst]: .txt
> Name of your master document (without suffix) [index]:
> Create Makefile? (y/n) [y]: y

Finished: An initial directory structure has been created.

You should now populate your master file ./source/index.txt and create
other documentation
source files. Use the sphinx-build.py script to build the docs, like so:

    make <builder>
```

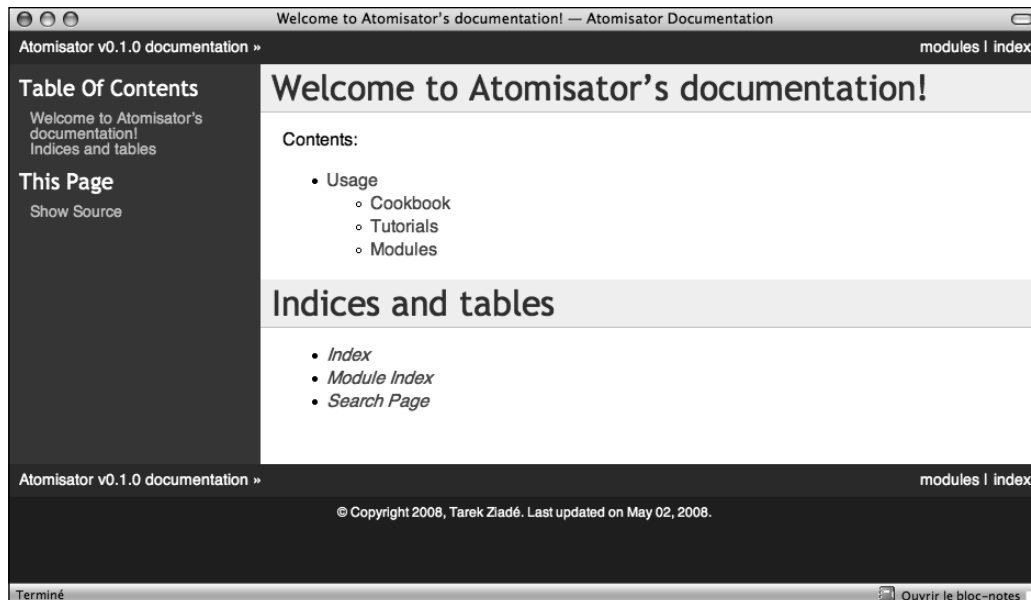
This adds a `conf.py` file in the source folder that contains the configuration defined through the answers, and an `index.txt` file at the root, together with a Makefile in docs.

Running `make html` will then generate a tree in build:

```
$ make html
mkdir -p build/html build/doctrees
sphinx-build.py -b html -d build/doctrees -D latex_paper_size= source
build/html
Sphinx v0.1.61611, building html
trying to load pickled env... done
building [html]: targets for 0 source files that are out of date
updating environment: 0 added, 0 changed, 0 removed
creating index...
writing output... index
finishing...
writing additional files...
copying static files...
dumping search index...
build succeeded.
```

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

The documentation will then be available in `build/html`, starting at `index.html`.



Besides the HTML versions of the documents, the tool also builds automatic pages such as a module list and an index. Sphinx provides a few `docutils` extensions to drive these features. The main ones are:

- A directive that builds a table of contents
- A marker that can be used to register a document as a module helper
- A marker to add an element in the index

Working on the Index Pages

Sphinx provides a `toctree` directive that can be used to inject a table of contents in a document, with links to other documents. Each line must be a file with its relative path, starting from the current document. Glob-style names can also be provided to add several files that match the expression.

For example, the index file in the `cookbook` folder, which we have previously defined in the producer's landscape, can look like this:

```

=====
Cookbook
=====

Welcome to the CookBook.

Available recipes:
.. toctree::
   :glob:
   *
```

With this syntax, the HTML page will display a list of all `reStructuredText` documents available in the `cookbook` folder. This directive can be used in all index files to build a browseable documentation.

Registering Module Helpers

For module helpers, a marker can be added so that it is automatically listed and available in the module's index page:

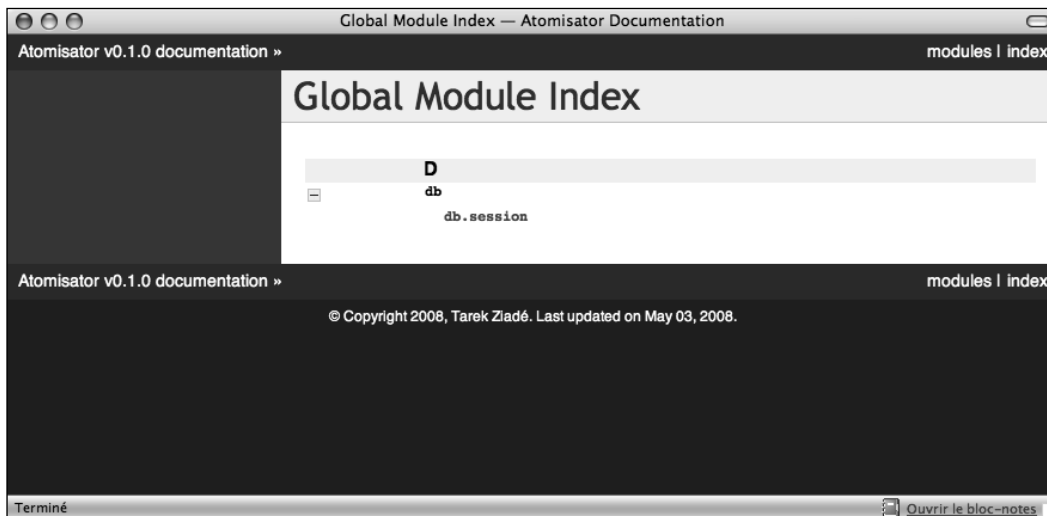
```

=====
session
=====

.. module:: db.session
The module session...
```

Notice that the `db` prefix here can be used to avoid module collision. Sphinx will use it as a module category and will group all modules that start with `db.` in this category.

For Atomisator `db`, `feed`, `main`, and `parser` can be used in order to group the entries, as shown in the figure:



In your documentation, you can use this feature when you have a lot of modules.



Notice that the module helper template that we created earlier (`pbp_module_doc`) can be changed to add the `module` directive by default.

Adding Index Markers

Another option can be used to fill the index page by linking the document to an entry:

```
=====  
session  
=====  
  
.. module:: db.session  
  
.. index::  
    Database Access  
    Session  
  
The module session...
```


Two new entries, `Database Access` and `Session` will be added in the index page.

Cross-references

Finally, Sphinx provides an inline markup to set cross-references. For instance, a link to a module can be done like this:

```
:mod:`db.session`
```

Where `:mod:` is the module marker's prefix and ``db.session`` is the name of the module to be linked to (as registered previously), keep in mind that `:mod:` as well as the previous elements are the specific directives introduced in reStructuredText by Sphinx.


 Sphinx provides a lot more features that you can discover in its website.
 For instance, the `autodoc` feature is a great option to automatically extract your doctests to build the documentation.
 See <http://sphinx.pocoo.org>.

Summary

This chapter explained in detail how to:

- Use a few rules for efficient writing
- Use reStructuredText, the Pythonistas LaTeX
- Build a document portfolio and landscape
- Use Sphinx to generate nice web documentation

The hardest thing to do when documenting a project is to keep it accurate and up to date. Making the documentation part of the code repository makes it a lot easier. From there, every time a developer changes a module, he or she should change the corresponding documentation as well.

This can be quite difficult in big projects, and adding a list of related documents in the header of the modules can help in that case.

A complementary approach to make sure the documentation is always accurate is to combine the documentation with tests through doctests.

This is covered in the next chapter, which presents Test-Driven Development principles, and then Document-Driven Development.