

# Building informational webs with PyStyler (Version 2.0)



## Abstract

PyStyler is a tool for the construction and maintenance of HTML-based informational webs. It automates the insertion of navigational links and allows for the enforcement of a consistent style and insertion of stereotyped design features.

John W. Shipman, john@nmt.edu  
New Mexico Tech Computer Center  
Socorro, NM 87801

---

## Contents

1. Introduction to PyStyler . . . . .	3
2. Overview of the web creation process . . . . .	3
3. Web page style . . . . .	4
3.1 Avoiding navigational shock . . . . .	4
3.2 Default "Next" and "Previous" links . . . . .	4
3.3 Default "Up" and "See also" links . . . . .	5
4. Building your directory structure . . . . .	6
4.1 The PathMap file . . . . .	6
4.1.1 What is a "topic?" . . . . .	7
4.1.2 What is a "short name?" . . . . .	7
4.1.3 Topic definition lines in the PathMap file . . . . .	8
4.2. The Plan file . . . . .	8
4.2.1 Control lines in the Plan file . . . . .	9
4.3. The Template file . . . . .	10
4.3.1 The %author; format code . . . . .	11
4.3.2 The %body; format code . . . . .	11
4.3.3 The %nextTitle; format code . . . . .	11
4.3.4 The %nextURL; format code . . . . .	11
4.3.5 The %prevTitle; format code . . . . .	11

4.3.6	The %prevURL; format code . . . . .	12
4.3.7	The %seeAlso; format code . . . . .	12
4.3.8	The %title; format code . . . . .	12
4.3.9	The %upTitle; format code . . . . .	12
4.3.10	The %upURL; format code . . . . .	12
4.3.11	The %url; format code . . . . .	12
4.3.12	The %updated; format code . . . . .	12
4.3.13	Conditional expansion in the Template file . . . . .	12
4.3.14	The %ifNext; construct . . . . .	13
4.3.15	The %ifPrev; construct . . . . .	13
4.3.16	The %ifSee; construct . . . . .	13
4.3.17	The %ifUp; consruct . . . . .	14
5.	Overview of page creation . . . . .	14
6.	The page header section . . . . .	14
6.1	The <author> header tag: Author credit . . . . .	15
6.2	The <next> tag: Designate the next page . . . . .	15
6.3	The <prev> tag: Designate the previous page . . . . .	15
6.4	The <see> tag: Designate see-also pages . . . . .	16
6.5	The <template> tag: Designate an alternate template . . . . .	16
6.6	The <updated> tag: Modification timestamping . . . . .	17
7.	The page body section . . . . .	17
7.1	Linking with the <g> tag . . . . .	17
7.2	Linking with the <r> tag . . . . .	18
7.3	Linking with the <rr> tag . . . . .	18
7.4	An example of a complete source page . . . . .	19
8.	Using the RCS source code control system . . . . .	19
9.	Running PyStyler . . . . .	20
10.	StylIndex, an automatic index builder . . . . .	21
11.	Conversion of existing webs . . . . .	22
12.	planview.cgi, an automatic site map viewer . . . . .	23
12.1	Installation of planview.cgi . . . . .	24

---

## 1. Introduction to PyStyler

---

The PyStyler program can help you build and maintain your World Wide Web pages when you start having more than a few dozen of them. In particular, PyStyler automates these common chores of web maintenance:

- You can enforce a uniform page style, such as standard title styles and page credits, without having to add these features by hand to each page.
- The work required to maintain navigational features (such as next-page and previous-page links) is greatly reduced.
- If you decide to change the directory structure you use to store your pages, it is less work to find and fix all the references to pages that have moved.

We will start by discussing the creation of a new web from scratch. A later section will discuss the conversion of existing webs.

## 2. Overview of the web creation process

---

Skills you will need to maintain a PyStyler-based web include: a basic understanding of Unix commands and directories; ability to use some general-purpose text editor such as emacs or pico; and knowledge of HTML.

Here is the general work flow involved in creating and maintaining a PyStyler-based web.

1. Create a specific Unix directory where all the files will live.
2. Using an ordinary text editor, create certain data files that describe the structure of your web.
3. Create a file for each page in the web—using a mixture of ordinary HTML and special notation for PyStyler. These *source files* must have names ending with `.g`, not `.html`.
4. Bring the files created under the RCS source code control system (optional).
5. Run the PyStyler program, which will create the `.html` file corresponding to each `.g` file automatically, a process called *expansion*. If this operation produces any error messages, they will be written to file `webstyler.log`. Read this file, fix the input files and run PyStyler again.

When PyStyler runs to completion and reports no problems, it will create all the actual HTML files of your web, and you can browse them normally.

### 3. Web page style

---

Before we discuss the mechanics of using PyStyler, we need to define a few terms.

By *navigation*, we mean the features of a web page that give the reader some places to go from the current page.

#### 3.1 Avoiding navigational shock

The concept of *navigational shock* is best illustrated by an example.

Suppose you are reading a page on the web, and you see a link with the text “see *How to play the flute*” and you click on that link, and it takes you to a short story entitled *Fungo bats at bay*. Wouldn’t you find it to be a rather unpleasant, disorienting surprise?

The best way to avoid navigational shock is to use the same link text as the title of the page you’re linking to. For example, if you have a link whose text is “How to play the flute,” and it takes you to a page entitled “How to play the flute,” the reader will go where they expect to go.

It’s also okay to use link text that isn’t identical to the target page’s title, so long as the reader isn’t likely to be surprised. We’ll call this a *link variant*.

For example, if you’re in the middle of a series of pages on mastodon hunting, and every page’s title starts “Mastodon hunting: ...,” and you want to link to a page whose title is “Mastodon hunting: sometimes the mastodon gets you,” your link text might read “see the page on when the mastodon wins.” That might not be too bad a case of navigational shock.

Whether such a link variant constitutes navigational shock is a matter of judgement. The editor of a web might want to monitor such cases, and PyStyler provides an optional report that lists all these link variants.

Keep in mind that a link text should supply enough information so that the reader can make an informed decision about whether or not they want to go there. This ties in with some of the bedrock rules of technical writing, like using descriptive titles.

#### 3.2 Default “Next” and “Previous” links

It is also convenient to provide *standard navigational links* from each page, such as links to “Next” and “Previous” topics for material that is serially organized. These standard links are typically at the top or bottom of a page, sometimes both.

The maintenance of standard links is a nuisance if you have to do it by hand. For example, if you add a new page in the middle of a sequence, you have to fix the “Next” link on the preceding page and the “Previous” link on the following page to point to the new page. If these links aren’t maintained, they can confuse or strand the reader.

PyStyler provides features that will automatically fill in standard navigational links of certain common types. These links go by default to places that are determined by the outline structure of your web.

For example, suppose a fragment of your web has this outline structure:

- 4.2 Using a parachute
  - 4.2.1 Get out of the plane
  - 4.2.2 Pull the ripcord

### 4.2.3 Roll when you land

Suppose that topic 4.2 is a bullet list that has links to topics 4.2.1, 4.2.2, and 4.2.3. If someone wants to read all the lower-level topics in sequence, it's convenient for 4.2.1 to have a "Next" link that points to 4.2.2, and then page 4.2.2 will have a "Next" link that points to 4.2.3.

Someone who comes to 4.2.3 from elsewhere might appreciate a "Previous" link on that page that points back to 4.2.2, so they can back up to the beginning of the sequence—it might be important! And 4.2.2 has a "Previous" link that points back to 4.2.1.

Of course, you won't want to be stuck with this particular navigational sequence in all cases; PyStyler just builds those links as a default. You can override these defaults, and specify that a given page's "Next" link can point somewhere else, or that there be no "Next" link at all.

### 3.3 Default "Up" and "See also" links

So far, we've given the reader a way to move down the tree (that is, further into the outline) by explicit links from 4.2 to 4.2.1 and 4.2.2. We've given them a way to move laterally along the sequence back and forth between 4.2.1 and 4.2.2.

However, we should also give the reader a way to move back up the tree, and to make jumps to related topics.

One clear way to present options to the reader is to have a navigational link at the end of the page labeled "See also," followed by one or more links elsewhere. For example, a page on carburetors might say, "See also: fuel pumps, intake manifolds, and air cleaners."

As you wander the Web, you'll see pages with "Up" links. The problem with this is that "Up" refers to an outline structure that is not readily perceptible to a reader who got there by jumping into the middle of your structure from elsewhere. So the reader may find it difficult to predict where "Up" will go.

I prefer the technique that Dr. Jonathan Price recommends: that you provide an "Up" link, but that you call it a "See also" link. If the page above this one has a meaningful title, the reader will realize that it is a more inclusive topic, and can make an informed decision whether to go there.

PyStyler lumps these two categories together: the "See also" link out of a PyStyler page, by default, will point to the next higher topic in the outline (from 4.2.2 to 4.2, for example). You can add more see-also topics to any page, and you can also suppress the "up" link on a page if you prefer.

Now that we've defined how the default navigation works, let's move on to the mechanics of putting up your web.

## 4. Building your directory structure

---

You must designate a specific directory as your *starting directory*.

- This must be your current working directory when PyStyler is run.
- You must create certain specific files in this directory that PyStyler needs for its operation.
- The pages of your web can be stored in the starting directory, or they can be distributed to any number of directories, but PyStyler requires that all the files associated with your web live either in the starting directory or below it in the directory tree.

If your web is fairly small, you can keep all the files in the starting directory. As your web grows, you can migrate files to, or create them in, subdirectories of the starting directory, and subdirectories of those subdirectories and so on.

There are three special files that control the operation of PyStyler. You must create these files in the starting directory:

- The `PathMap` file supplies PyStyler with information about your directory structure.
- The `Plan` file describes the *outline* structure of your web. This may not be the same as your directory structure.
- The `Template` file describes the standard page layout that will be enforced on all the pages of your web, including the standard navigational features.

### 4.1 The `PathMap` file

---

Once you have created your starting directory, the next step is to create a file in that directory named `PathMap` (capitalized just so). Use a text editor to create this file.

This file must start with a line that gives the URL of your starting page, minus its file name. For example, if the URL of your home page is:

```
http://www.foo.com/research/homepage.html
```

then the first line of your `PathMap` file must contain this string:

```
http://www.foo.com/research/
```

The remaining lines of this file, if any, define the *topic names* used in your web. Before we discuss the format of these lines, let's fill in the background.

With ordinary HTML, links refer to specific path names—so if you move an HTML file, you have to find all the files that refer to it and fix their links. This can be a tedious and error-prone operation.

### 4.1.1 What is a “topic?”

One of the reasons PyStyler was written was to make life easier when you have to reorganize the directory structure of your web. As webs grow, sometimes so many files will be added to a directory that it becomes unwieldy—to the point where the output from `ls` doesn't even fit on one screen. The obvious solution is to create one or more subdirectories and migrate most of the pages down into these new subdirectories.

Suppose for the moment that a cross-reference, from some page *a* to some other page *b*, uses the full directory path of *b*. If *b* were moved to a different path, then, all the cross-references to that page would have to be found and fixed.

So, in order to reduce the pain of moving pages around in the directory structure, PyStyler uses the concept of a *topic*. Each directory other than the starting directory is given a short, unique *topic name*. References in one place in your web that link to some other place in your web must use this topic name, instead of the path name, and PyStyler will automatically fill in the current path name of the reference.

The PathMap file describes where each topic currently lives. Then, if at some future time you decide to move a topic to a different directory, all you have to do (other than physically move the files) is to change one line in PathMap and rerun PyStyler, and all of your references to files in that topic will automatically be changed to point to its new location.

If you have a very small web, and all the files are in the starting directory, your PathMap file need have only the one line giving the base URL (described above).

For nontrivial webs, though, it is convenient to define a set of topic names, and use the PathMap file to describe where each topic lives, as a relative pathname (relative to the starting directory).

### 4.1.2 What is a “short name?”

A short name consists of a topic name, followed by a slash (/), and then the name of the source file, without its `.g` extension.

If the file is in the starting directory, omit the topic name and slash. Here are some examples of short names:

```
breeds/alley
homepage
```

The first example refers to file `alley.g` in topic `breeds`. The second example refers to file `homepage.g` in the starting directory.

You can use anchor references in a short name. For example, if file `breeds/alley` contains an anchor `#mongrel`, then `breeds/alley#mongrel` is a valid short name and refers to that location in that file.

### 4.1.3 Topic definition lines in the PathMap file

All lines of the PathMap file (after the first line giving the base URL of the web) constitute a catalog of the currently defined set of topic names. Each line starts with the topic name, followed by one or more spaces or tabs, followed by a relative pathname describing how to get from the starting directory to the directory where the files for that topic live.

For example, suppose your web starts at this URL:

```
http://www.meow.com/cats/homepage.html
```

Suppose also that its files are located in directories `breeds`, `food`, `food/dry`, and `food/canned`. Your complete PathMap file might look like this:

```
http://www.meow.com/cats/  
breeds           breeds  
canned           food/canned  
dry_food         food/dry  
food             food
```

If your PathMap file gets fairly large, you may want to alphabetize the topic entries so you can find them easily.

## 4.2. The Plan file

---

While the PathMap file describes where the files of your web live in the directory structure, the Plan file gives a completely different view of your web—as an outline.

In the section above on default navigation, you will note that these links assume the existence of an outline. That is one function of the Plan file: it enforces a formal outline structure on the web. It is *not* required that you make this outline visible to the reader—that is up to you.

The Plan file also serves as an inventory of every page in your web, so PyStyler will know which `.g` files it has to read to generate the corresponding `.html` files.

The Plan file uses the “outline mode” of the emacs text editor. Each nonblank line describes one piece of the outline. In emacs, use the sequence `M-x outline-mode` to enable outline mode. See the author for documentation on the features of this mode.

- Each line must start with one or more asterisk (\*) characters that describe how deep the corresponding topic is in the outline. The first line of the file must have exactly one asterisk, and it describes the root or starting page of your web. Remaining lines must start with two or more asterisks. You are not allowed to jump levels; for example, a level-4 entry may not follow a level-2 entry.
- Following the asterisks, place the title of the page. Note that the title of each page is given in the Plan file, and *not* in the source file.
- After the end of the title, type a vertical bar (|) to separate the title from the next field.
- At the end of the line, type the *short name* of the source file that contains the text for this page (see *What is a short name?*, above).

You can use blanks liberally between these fields to make the file more readable.

For example, here are the first few lines from the Plan file corresponding to the example PathMap above:

```
* The cats page      | homepage
** Breeds of cats   | breeds/homepage
*** The Abyssinian  | breeds/abyssinian
*** The Alley       | breeds/alley
```

The first line says that the starting point for this web is a page entitled “The cats page.” The source file for this page is in file `homepage.g` in the starting directory. The URL of the actual page produced by PyStyler from this page will be `http://www.meow.com/cats/homepage.html`.

The second line describes a second-level topic entitled “Breeds of cats,” and its source file is in directory `breeds/homepage.g`. The third line is for a page entitled “The Abyssinian,” and its source file is `breeds/abyssinian.g`, and so on.

#### 4.2.1. Control lines in the Plan file

Interspersed with the topic lines in the Plan file, you may also place control lines that specify options for a group of topics. Control lines start with a dollar sign (\$).

At the moment, there is only one control line, `$template`. The purpose of this line is to specify a different template for one or more topics (see below for a discussion of template files). There are two forms:

```
$template filename
$template
```

The first form tells PyStyler to start using the given *filename* as the template for the following topic lines. The second form tells PyStyler to revert to the preceding template.

These pairs of control lines can be nested. Here is an extended example:

```
* The cats page      | homepage
$template breed.tpl
** Breeds of cats   | breeds/homepage
*** The Abyssinian  | breeds/abyssinian
*** The Alley       | breeds/alley
...
$template longhair.tpl
*** Persians        | breeds/persian
$template
*** Siamese         | breeds/siamese
$template
** Cat food         | foods/homepage
...
```

In the above example, the default template file, `Template`, would be used for topics `homepage` and `foods/homepage`. Template file `breed.tpl` would be used for `breeds/homepage`; that same template would also be used for `breeds/siamese`, because the `$template` line before it ends the use of template `breed.tpl`.

### 4.3. The Template file

---

Now that you have created a PathMap file that describes where the topics' directories are, and a Plan file enumerating all the files and their outline structure, it is time to create the Template file, which specifies the standard page layout.

The Template file is useful for giving your pages a uniform "look and feel." It also specifies how you want your standard navigational links to look.

Create the Template file using a normal text editor. This file is sort of a general frame into which the body of each page is inserted. There are two kinds of elements:

- Most of the text in the Template file is placed literally on each page. For example, if you want a graphic logo to be placed at the beginning of every page, you might start the Template file with something like:

```

```

and that bit of HTML would be the first thing in each .html file of your web.

- You can also type special *format codes* in the Template file that cause the insertion of certain features onto each page. All format codes start with a percent sign (%) and end with a semicolon (;). The only required element is the %body; format code, which means "put the body of the page here."

Before we look at all the various format codes that you can use in your Template file, let's look at an example of a complete Template file to get the idea.

```
<title>%title;</title>
<h1>%title;</h1>

%body;

<hr>
<b>Next:</b> %nextURL;%nextTitle;</a><br>
<b>See also:</b> %seeAlso;<br>
<b>Previous:</b> %prevURL;%prevTitle;</a><br>
<b>Home:</b>
<a href="http://www.foo.bar/">Back to my home page</a><br>
```

This Template file shows a basic page style. The page starts with <title> and <h1> tags, each enclosing the page's title. The %title; format code means "insert the current page's title here."

The position of the %body; format code determines where the text of each page will be placed. All the rest of the text after this point causes navigational links and other text to be placed below the page body.

The <hr> tag sets off the body from the navigational links with a horizontal rule.

The next line causes a "Next page" link to be automatically generated at that position. For example, if a page named part1.g was followed by a page named part2.g in the Plan outline, and the title of part2.g is "Part II", then the part1.html file built by PyStyler from this Template would contain a line that looks like this:

```
<b>Next:</b> <a href="part2.html">Part II</a><br>
```

**Note:** The way you capitalize format codes is ignored. Format codes `%nexturl`; and `%NextURL`; and `%NEXTURL`; are all treated as identical.

This brief example should give you a general idea of what is going on inside a `Template` file. Now let's discuss the different format codes and their meaning.

#### 4.3.1 The `%author`; *format code*

You can specify the automatic insertion of an author credit on each page by using the `%author`; format code in your `Template`.

If your pages are written by more than one author, you can place information in each source file about who wrote it by using the special `<author>` tag (see the section on page creation below for more details). If the `%author`; format code appears in the `Template`, the author credit for that page will appear at that position.

For example, suppose the `Template` file contained this line:

```
<address>%author;</address>
```

and source file `redbeans.g` contains this author credit:

```
<author>Paul Prudhomme, <tt>paul@kpaul.com</tt></author>
```

then the output file `redbeans.html` would have this text at the position of the `%author`; item:

```
<address>Paul Prudhomme, <tt>paul@kpaul.com</tt><address>
```

#### 4.3.2 The `%body`; *format code*

This format code means "put the actual body of the page here." It must appear exactly once in the `Template` file. Stuff before this code in the `Template` will appear at the top of the page, and stuff below it will appear at the bottom of the page.

#### 4.3.3 The `%nextTitle`; *format code*

This format code expands to the title of the "Next" page after the current page. For example, if page `topic1.g` is followed by a page whose title is "Break forth," then the text `Break forth` will appear in file `topic1.html` at the position corresponding to the `%nextTitle`; format code.

#### 4.3.4 The `%nextURL`; *format code*

This format code expands to an HTML `<a>` tag pointing to the current page's "Next" page. For example, if the "Next" page for file `topic1.g` is file `topic2.g`, then the text

```
<a href="topic2.html">
```

will appear in file `topic1.html` at the position corresponding to the `%nextURL`; format code.

#### 4.3.5 The `%prevTitle`; *format code*

This is like the `%nextTitle`; code, but it expands to the title of the "Previous" page relative to each page.

#### 4.3.6 The `%prevURL`; *format code*

Like the `%nextURL`; *format code*, the `%prevURL`; *format code* expands to an `<a>` tag, but pointing at the “Previous” page instead of the “Next” page.

#### 4.3.7 The `%seeAlso`; *format code*

This *format code* expands to a list containing: a link to the page above this one in the outline, if any, with that page’s title as its link text; followed by any additional “see-also” pages specified in each page, if any—see the section on page writing under the `<see>` tag.

#### 4.3.8 The `%title`; *format code*

This *format code* expands to the title of the current page. For example, if the `Plan` file says that the title of page `manx.g` is “The Manx,” then in file `manx.html` the text “The Manx” will appear at each position where `%title`; occurs in the `Template`.

#### 4.3.9 The `%upTitle`; *format code*

For those who prefer a simple “next higher topic” link instead of a “see also” link, the `%upTitle`; *format code* expands to the title of the page above the current page in the topic hierarchy, if any. If it occurs in the root (starting) page of the whole web, it expands to no text. See the `%ifUp`; *format code*, below, for a way to deal with this special case.

#### 4.3.10 The `%upURL`; *format code*

Like the `%nextURL`; *format code*, the `%upURL`; *format code* expands to an `<a>` tag. This tag links to the page for the parent topic of the current page, if any. Remember that you must provide link text and a closing `</a>` tag. The `%upTitle`; *format code* is a good choice for the link text.

#### 4.3.11 The `%url`; *format code*

This *format code* expands to the URL of the current page. For example, if the `PathMap` file says that our base URL is `http://www.meow.com/cats/`, then output file `breeds/tabby.html` will contain this text at the position corresponding to any `%url`; *format codes* in the `Template`:

```
http://www.meow.com/cats/breeds/tabby.html
```

#### 4.3.12 The `%updated`; *format code*

If you would like to provide information on each page as to when that page was last updated, you can use the `%updated`; *format code*. This feature requires that you use the RCS source code control system (discussed below). When writing each source page, you must insert an RCS “Date” string inside a special `<updated>` tag. The `%updated`; *format code* expands to the date extracted from that tag on each page, automating the process of timestamping each page.

#### 4.3.13 *Conditional expansion in the Template file*

In some situations, you will want the text of a page to be different in different situations. `PyStyler` provides *conditional expansion* for these cases.

For example, suppose your `Template` calls for the word “Next:” to appear in boldface type, followed by the link to the next page (using `%nextURL`;). However, on pages that don’t have a “Next” page, you don’t want the boldfaced “Next:” to appear.

#### 4.3.14 The `%ifNext;` construct

There are two ways to use the `%ifNext;` construct. The first way is like this:

```
%ifNext;  
  true-text  
%endif;
```

In this form, the *true-text* appears only on pages that have a “Next” page.

```
%ifNext;  
  true-text  
%else;  
  false-text  
%endif;
```

This form places *true-text* on pages that have a “Next” page, and it places *false-text* on pages that don’t have a “Next” page.

Here is an example:

```
%ifnext;  
  <b>Next:</b> %nextURL;%nextTitle;</a>  
%endif;
```

For a page that has no “Next” page, this part of the Template will generate no text.

Suppose, however, that you have a source file whose “Next” page is a file in the same directory whose name is `frogs.g` and whose title is “Types of frogs.” Then the above section of the Template would expand to:

```
<b>Next:</b> <a href="frogs.html">Types of frogs</a>
```

Here’s an example that uses the `%else;` part. Suppose you are building a “button bar” on the top of your page that has the word “Next” in it, and you want that word to be a link to the next page if there is one, but you don’t want it to be a link otherwise. Part of your Template might read like this:

```
%ifnext;  
  %nextURL;Next</a>  
%else;  
  Next  
%endif;
```

#### 4.3.15 The `%ifPrev;` construct

The `%ifPrev;` construct has the same form as the `%ifNext;` construct, only its expansion depends on whether the source page has a “Previous” page or not.

#### 4.3.16 The `%ifSee;` construct

Structurally the same as `%ifNext;`, the expansion of the `%ifSee;` construct depends on whether current source page has any “See also” pages. As discussed below in the section on page preparation, there can be any number of “See also” links on a page. The expansion of `%ifSee;` produces the *true-text* if there are any see-also targets, or it produces the *false-text* (if any) if there are none.

### 4.3.17 The `%ifUp;` construct

Similar to `%ifNext;` and `%ifPrev;`, the expansion of the `%ifUp;` conditional depends on whether the current page is or is not the root page of the entire web. If the page is not the root, the expansion produces the *true-text*. If the page is the root and there is an `%else;` part, the expansion produces the *false-text*.

## 5. Overview of page creation

---

Once you have prepared the `PathMap`, `Plan`, and `Template` files, the next step is to write the actual web pages.

Each page must live in a separate file whose name ends with a `.g` extension. All such pages must be enumerated in the `Plan` file. When `PyStyler` runs, it will go and find each source file, read it, and use it to build the final `.html` file.

For example, if the `Plan` file includes an entry for a file called `breeds/tabby`, and the `PathMap` file states that topic `breeds` is found in directory `catalog/breeds`, `PyStyler` will try to read file `catalog/breeds/tabby.g` and write file `catalog/breeds/tabby.html`.

Use a normal text editor to prepare the source files for your pages. Each source file consists of two parts:

- An optional *header section*, enclosed between `<head>` and `</head>` tags. This section contains special tags (not HTML, though they look like HTML) that affect the way `PyStyler` expands that page.
- The remainder of the source file can be just straight HTML, or it can be a mixture of HTML and special tags that `PyStyler` will process. This part of the source file is called the **body**, and it will appear in the output `.html` file at the position corresponding to the place in the `Template` file where the `%body;` format code appears.

## 6. The page header section

---

If you don't need separate author credits on each page, and you aren't using the `%updated;` tag for automatic timestamping, and you are happy with the default navigation, you don't need a header section.

Otherwise, your file should start with a `<head>` tag, followed by one or more of the special header tags, followed by a `</head>` tag. This part of the source file is called the *page header section*.

The special header tags—`<author>`, `<next>`, `<prev>`, `<see>`, `<template>`, and `<updated>`—are described below.

## 6.1 The `<author>` header tag: Author credit

There are two ways to insert an author credit onto the pages of your web. If the author is always the same, you can just put the author credit in your Template.

If, however, the author credits vary among pages, you should provide author information in each source file's header section. Then you can use the `%author;` format code in the Template, and the author credit will appear on each page at the position corresponding to that format code.

To provide this author information, enclose it within a pair of `<author>...</author>` tags in the page's header section. For example:

```
<author>Joe Beets,  
<tt>mudhead@morse.science.edu</tt></author>
```

Note that the author credit can include HTML markup tags.

## 6.2 The `<next>` tag: Designate the next page

By default, the "Next" link of a given page points to the next topic (if any) at that same level in the outline, as described in the Plan file. For example, if topic 4.2.1 is followed by 4.2.2, then the "Next" link of 4.2.1 points to 4.2.2 by default. (These numbers don't actually exist in the file, but they are used for purposes of illustration.) This means that some files, such as the starting file or the last file, will not have a default "Next" link.

When you're writing a page, you may want to specify that the "Next" link on that page goes somewhere other than to the default location. Or you may want to specify a "Next" link for a page that has no default "Next" page. To do this, use a tag of this form in the page header:

```
<next id="shortname">
```

where *shortname* is the short name of the topic you want to use as this page's "Next" topic.

For example, suppose you are writing a source file named `foo.g` and you want that page's next link to go a file named `bar.g` under the topic `buzzwords`. You would place this tag in the page header section, anywhere between the `<head>` tag and the `</head>` tag:

```
<next id="buzzwords/bar">
```

On some pages, you might want to override the default "Next" link and specify that there is **no** "Next" link at all. You can do this by including this tag in that page's header section:

```
<next none>
```

## 6.3 The `<prev>` tag: Designate the previous page

The default "Previous" link on each page is the previous topic at the same level, if any. So the "Previous" link of topic 4.2.3 would point by default back to topic 4.2.2. Pages with no previous topic at the same level have no default "Previous" link.

You can override the default "Previous" link on any page. Use a tag of the form:

```
<prev id="shortname">
```

There is also a form that suppresses the "Previous" link on a page:

```
<prev none>
```

## 6.4 The `<see>` tag: Designate see-also pages

The `<see>` tag can be used to add new links to the list of “See also” topics for a page. For each topic to be added, use a tag of this form in the page header:

```
<see id="shortname">
```

There is also a form for suppressing all “See also” links, including the default link to the parent (next higher) topic:

```
<see none>
```

If you want a page to have some see-also links but not the default link to the parent topic, you can use a `<see none>` tag first to remove the parent link, then use one or more tags of the `<see id="shortname">` form to specify the links you do want.

## 6.5 The `<template>` tag: Designate an alternate template

If you want some of your pages to use a different template than the one given in the `Template` file, you can specify an alternate template file in those pages using this header tag:

```
<template id="pathname">
```

where *pathname* is the pathname of a template file, relative to the starting directory. Alternate template files use the same format as the main `Template` file.

*Note.* This tag is now **deprecated**. It will still work, but you should use `$template` control lines in the `Plan` file to specify alternate templates.

The reason for this deprecation is somewhat complicated. In the first version of `PyStyler`, running the program rewrote every single `.html` file every time. Usually, though, you are only changing a few pages at a time. So, for performance reasons, the second version (2.0) checks the timestamps on the existing `.html` file, if any, and only rewrites those whose `.g` files have been changed since the corresponding `.html` file was written. Checking the timestamps of two files is a much faster operation than reading the `.g` file and re-writing the `.html` file.

This dependency of the `.html` file on the `.g` file, though, is not the only dependency. Clearly, if a given page’s template has changed, the page must be rewritten in that case as well. If a given page uses the default template, `PyStyler` will rewrite the `.html` file if the template is newer. If the page’s template is specified in the `Plan` file, again we will rewrite the `.html` file if the template is newer.

However, if `PyStyler` has to go and read the `.g` file to see if it contains a `<template>` tag, this annuls a good part of the performance increase of just checking timestamps. So if `PyStyler` finds that the timestamp of the `.html` file is newer than that of the `.g` file and the file’s template (whether the default template or one specified in the `Plan` file), it does not read the `.g` file to see if it contains a `<template>` tag.

This, however, leads to the following “gotcha:” If your `.g` file contains a `<template>` tag, and you change the template, the `.html` file will not be rewritten! This is why `PyStyler` has a “force” command line option, `-f`, to force rewriting of all `.html` files.

## 6.6 The `<updated>` tag: *Modification timestamping*

It is a great courtesy to the reader when each of your pages shows the date and time of its last modification. PyStyler can make the updating of these timestamps automatic, in combination with the source code control system RCS, discussed below.

Assuming that all your source files are under RCS control, you can get automatic timestamping by including a tag of this form in each page header section:

```
<updated date="$Date$">
```

RCS will automatically place a date into this tag every time you check in your files. Then, place an `%updated;` format code somewhere in your `Template`, and the timestamp will appear at that position on each page.

## 7. The page body section

---

You write each page body using pretty much standard HTML notation. However, you can use certain special PyStyler-only tags that make it easier to build cross-links *within* your web. These tags make it possible to point to a page without knowing its exact location within your directory structure.

These special tags can also help you avoid or control navigational shock. As discussed above, the best way to avoid navigational shock is to use as link text the title of the page you're linking to.

### 7.1 Linking with the `<g>` tag

The `<g>` tag was invented for linking to glossary entries. It looks like this:

```
<g id="shortname">
```

This expands to:

```
<a href="filename">title</a>
```

where *filename* is the actual file name of the page linked to, and *title* is its title as given in the `Plan` file. Note that if the target page gets moved, the next run of PyStyler will change the *filename* to reflect that target page's new file name.

This form of link is very handy with glossary pages whose title is the term being defined. For example, suppose you have a glossary topic that contains a source file call `fubar.g` that defines what the term "fubar" stands for, and whose title is simply "fubar." You might refer to it like this:

```
...Walt was <g id="glossary/fubar"> after...
```

The generated HTML text might read:

```
...Walt was <a href="glossary/fubar">fubar</a> after...
```

## 7.2 Linking with the `<r>` tag

The `<r>` tag works like the `<g>` tag except that it puts paired single quotes around the link text. This is a useful way of suggesting to the reader that the link points to a page with that title. The general form is:

```
<r id="shortname">
```

This expands to:

```
<a href="filename">'title'</a>
```

Here's an example. Suppose the Plan file says that the title for short name `food/homepage` is "An overview of cat food." You might encode a link to this page as:

```
If your cat is hungry,  
see <r id="food/homepage">.
```

which expands to:

```
If your cat is hungry,  
see <a href="food/homepage">'An overview of cat food'</a>.
```

*Note.* The actual file reference that appears after `HREF=` may be different, depending on the location of the reference. For example, if the referring page is in a directory two directories down from the starting directory, the reference might look something like `HREF="../../food/homepage"`. But you won't need to worry about the actual mechanics of this, so long as `PyStyler` is working correctly.

## 7.3 Linking with the `<rr>` tag

The `<g>` and `<r>` tags minimize navigational shock by guaranteeing that the link text is the same as the title of the target page. However, sometimes it reads better if the link text is not identical to the title. For this, use the `<rr>` tag, which is a paired tag:

```
<rr id="shortname">link-text</rr>
```

where the *link-text* is whatever text you want to use for a link. Unlike the `<r>` tag, single quotes are not inserted automatically.

Here is an example:

```
Mastodons are dangerous (see <rr id="mastodon/theywin">  
the page on when the mastodon wins</rr>).
```

The HTML generated from this might look like this:

```
Mastodons are dangerous (see <a href="mastodon/theywin.html">  
the page on when the mastodon wins</a>).
```

## 7.4 An example of a complete source page

Here is an example of a finished source file, ready to go:

```
<head>
  <author>John Shipman,
    <tt>tcc-doc@nmt.edu</tt></author>
  <updated date="$Date: 1996/01/06 22:37:39 $">
</head>

<p>Cat information includes:
<ul>
  <li><r id="breeds/homepage">
  <li><r id="food/homepage">
  <li>Also see the <rr id="art/show">art show.</rr>
</ul>
```

## 8. Using the RCS source code control system

---

It is not strictly necessary to archive all your source files using the RCS source code control system, unless you are using the %updated; template format code.

RCS has these benefits for those who write and maintain webs:

- It provides automatic timestamping of modifications with the %updated; format code.
- If more than one person is working on your web, RCS will help control access so that only one person is working on a file at a time. Files must be checked out, modified, and checked back in before someone else can modify them.
- You can get a complete history of all the changes made to a file, with an indication of who made them. You can also retrieve any earlier version of a file.
- Using RCS provides a quick recourse in case you accidentally delete a file: you can always retrieve the last version checked in. It might not be current, but it may be lots better than nothing at all.

For more information about RCS, see the Tech Computer Center help pages at:

<http://www.nmt.edu/tcc/help/tool/rcs/rcs.html>

## 9. Running PyStyler

---

Once you have constructed your PathMap, Plan, and Template files, and written all the source files mentioned in Plan, PyStyler will build all your .html files if you follow these steps.

1. Change your current working directory to the starting directory of your web.
2. Type this command:

```
pystyler
```

If there are any errors in any of the input files, PyStyler will write error messages to its standard output and also to a file named `pystyler.log`. Fix the errors and rerun the program. When PyStyler reports no errors, your web should be ready for browsing.

PyStyler has these command line options:

```
pystyler [-n] [-w newroot] [-f]
```

where:

- n Produces a “navigational shock report.” This report lists all of the uses of `<rr>` tags, starting with the page title and then, for each such link, it shows the actual text used for the link and then shows the file name and line number of the reference.
- w *newroot* is the path name of a directory. PyStyler will put all of the generated HTML files in and under that directory instead of in the same directory structure where the .g files live. If you are using subdirectories, you don’t have to create all the subdirectories of *newroot*; PyStyler will do that for you automatically. The idea here is that you may want to have a “test” version and a “production” version of the whole tree. You can tinker with structure and content in the test version without affecting the production version, and then rebuild the production version once you have it looking the way you want.
- f Normally, PyStyler will not rewrite any .html file whose timestamp is newer than both its corresponding .g file and the Template file. The -f option forces all .html files to be rewritten regardless of timestamps.

For an example of a sizable web produced by PyStyler, have a look at the NM Tech Computer Center help system:

```
http://www.nmt.edu/tcc/help/
```

## 10. StylIndex, *an automatic index builder*

---

Once your web reaches the size where it gets hard to find things just by clicking around on the navigational links, it can be very handy to have an index.

The StylIndex program works with PyStyler to generate an index to all keywords used in the titles of your web. It comes with a `.cgi` script that allows the user to search for any keyword.

So that the user doesn't have to wait for the entire index to download, the index is broken into 27 pieces—a short starting page, and one page for each letter of the alphabet. The starting page has a search form on it, and a “thumb index”—the letters of the alphabet, as links, so that the user can click on the initial letter of a word. Each of the 26 letter pages also starts with the search form, and starts and end with a thumb index.

There is a working example of a StylIndex index at:

```
http://www.nmt.edu/tcc/help/i/index.html
```

To build an index using StylIndex, follow these steps.

- a. Create a new directory where all the index files will live. For example, if you want to call this directory `inx`, change your current directory to the starting directory of the web and type this Unix command:

```
mkdir inx
```

- b. Add a line to your PathMap file assigning a short name to that directory. To continue the example, you would add a line to PathMap that looks like this:

```
inx      inx
```

- a. Add a new topic to your plan file for the index start page, followed by 26 topics for the letters of the alphabet. The start page can have any short name you like, but each of the 26 letter pages must have the same short name with an underbar (`_`) and the lowercased letter appended.

To continue the example, suppose your index is at level 2 in your Plan file. The lines added to your Plan file might look like this:

```
** Index          | inx/index
*** Index: A     | inx/index_a
*** Index: B     | inx/index_b
...
*** Index: Z     | inx/index_z
```

- b. If you would like to include some text before the search form on each page, explaining how the index works, type that text into a file in the index directory using a text editor. This file is referred to below as the “boilerplate” file.
- c. Using a text editor, create a file called `exclusion` in the index directory, containing all the words that should not be indexed. Obvious candidates are the letters of the alphabet and words like `an`, `of`, and `the`.
- d. Run the StylIndex program using this command:

```
stylindex [option ...] shortbase [boilerplate]
```

where:

<i>shortbase</i>	is the short name of the index's starting page in the Plan file; in the example, that would be i/index.
<i>boilerplate</i>	is the optional name of the "boilerplate" file mentioned in the previous step, if you have one.
<i>option</i>	Options include:
-D <i>dir</i>	The directory name, relative to the starting directory of the web, where the index data files should live. If you omit this option, these files (keywords, exclusion, and index.cgi) will all live in the starting directory.
-p	If given, the index will be made in the "permuted" style, with the keywords running down the center of the screen. The default is to build a book-style index, with the keywords on the left.

To continue the example, assuming your boilerplate file is named boiler, you would type this command to build the index files:

```
stylindex inx/index boiler -D inx
```

When you type this command, StylIndex will read the PathMap and Plan files and use them to build the .g files comprising the index—inx/index.g, inx/index\_a.g, ..., inx/index\_z.g.

- e. Build your web normally as described above in the section *Running PyStyler*.
- f. Look over the generated index pages with your favorite browser, and see if there are any terms that should not be indexed. If so, add those terms to the exclusion file and rerun StylIndex and PyStyler.
- g. Once your index looks okay, add links to it from the rest of your web. You may want to put an *Index* link in the page-end navigation links of your template, and an *Index* button on your page-top button bar if you have one.

## 11. Conversion of existing webs

---

Assuming you have read and understood the above material on constructing a new web, here is a suggested plan of attack for the conversion of existing HTML webs to PyStyler.

1. Use an entirely new directory tree for the PyStyler-converted web, so you can leave the old web in place until the new one is complete.
2. If you are happy with your current directory structure, a good start for a PathMap file would be to treat each directory as a topic. Give each one a unique name and make up your PathMap file by alphabetizing the topics. If your old web is fairly small, you can start without a topic structure, put all your files in the starting directory, and create a PathMap file with only the first line giving the base URL.
3. Design an outline structure for your web. Take advantage of the default navigational linking to structure serial topics so that the reader can use the "Next" links to move through them in sequence. Then build a Plan file that includes a line for every file in your web.

4. Decide how you want your standard page style to look, and build a `Template` file that describes this style.
5. Create all the directories corresponding to topics in your `PathMap` file.
6. Make copies of the `.html` files from your old web in the corresponding directories of the new web. Rename each of these files with the same name, except with a `.g` extension.
7. For each file mentioned in your `Plan`, use a text editor to rework that file so that it uses `PyStyler`'s features. Delete any page header or trailer information that will be filled in automatically by the `Template`. Then inspect every `<a>` link in the page. If it points at another file in your web, replace that link with either a `<g>` link, an `<r>` link, or an `<rr>` link, as appropriate. Links pointing to files outside your web can be left as they are.
8. Run `PyStyler`, look at the error messages, fix the problem, and rerun until there are no errors.

## 12. `planview.cgi`, *an automatic site map viewer*

---

The `planview.cgi` script is a CGI script that you can use in your web to allow your readers to browse the `Plan` file as if were an outline. This kind of functionality is sometimes referred to as a "site map."

To use this script, place in your `Template` or in any `.g` file a link that looks like this:

```
<a href="planview.cgi?h=h&aa=p&zz=e">...</a>
```

where:

- |          |   |
|----------|---|
| <i>h</i> | is the absolute path name of your starting directory, where the <code>Plan</code> file resides.   |
| <i>p</i> | is the name of a "prologue file" that is copied to the beginning of the page generated by the script. This file is optional, and if it is not to be used, omit the entire " <code>&amp;aa=p</code> " part of the tag. |
| <i>e</i> | is the name of an "epilogue file" that is appended to the end of the generated page. This too is optional; omit the " <code>&amp;zz=e</code> " part if it is not used.  |

The generated page has up to four parts:

1. The contents of the prologue file, if used.
2. A link to the root file of the web.
3. A "tree browser" section. The appearance and operation of this section will be familiar to anyone who has ever used Microsoft's Windows Explorer.

This section initially displays all of the second-level topics in your `Plan` file, with their titles displayed as links that take you directly to the relevant page. Each title is preceded by a clickable element that looks like (+). If the user clicks on that element, the page is regenerated with that node of the tree "opened," that is, with the subtopics of that topic shown under it and indented. The parent element that was clicked on will now be shown with a clickable element that looks like (-), and if the user clicks on that element, the page will be regenerated with that node "closed," that is, with its subtopics not displayed.

4. The contents of the epilogue file, if used.

Here is an example showing what the tree browser section of the generated page might look like for a web with three second-level topics:

- (+) Topic 1's title
- (+) Topic 2's title
- (+) Topic 3's title

All the titles are clickable and will take you to the page with that title.

Suppose that the user clicks on the (+) element for topic 2, and further suppose that topic 2 has four subtopics. The page might then look something like this:

- (+) Topic 1's title
- (-) Topic 2's title
  - (+) Topic 2.1's title
  - (+) Topic 2.2's title
  - (+) Topic 2.3's title
  - (+) Topic 2.4's title
- (+) Topic 3's title

Clicking on the (-) would restore the page to its original appearance.

## **12.1 Installation of `planview.cgi`**

---

You will need to place two files in your web's starting directory:

1. File `planview.py` is a Python script that does the actual generation of the site map. You may instead install the byte-compiled form of this script, called `planview.pyc`; this will load a bit faster.
2. The `planview.cgi` file is a short Bourne shell script that invokes the above script.

At the moment this works only on New Mexico Tech Computer Center machines, since these scripts rely on a number of standard modules from the author's personal Python library. Contact the author if you would like it installed elsewhere.

---

Written by John W. Shipman (`tcc-doc@nmt.edu`). This version printed 2004-05-20. Copyright © 2001 by the New Mexico Institute of Mining and Technology.

Revision: 1.11 Date: 2004/05/11 22:04:22