Literate Programming with xmltangle

Table of Contents

What is xmltangle

xmltangle is part of a system of Literate Programming. Literate Programming is a programming style in which the programmer writes an essay about his program which contains the program in its entirety. He then can either typeset his essay (known as weaving), or build the code (known as tangling). The tangle program allows the user to write their code in the way that it's easiest to think of and describe rather than how the compiler expects it. The tangle program can be instructed how to piece together your program properly.

xmltangle uses processing instructions to handle the tangling. That way, your documentation can be written using any XML DTD (actually, it doesn't even need a DTD). You can typeset your essay using the standard XML stylesheet methods, such as CSS, XSLT, or DSSSL. This program handles the tangling.

A Simple Example

This section contains example XML that constitutes a Literate Program. This program will simply ask the user to type in a number, and then add up all of the numbers 1 through the number which the user typed.

```
<?xml version="1.0" ?>
<article>
<title>Simple Literate Program</title>

<sect1>
<title>Introduction</title>

<para>
This program is made to illustrate the concepts of literate programming.
This program receives as input a number, and then adds up all of the
numbers between one and that number, and then prints the output.
The outline for the program is as follows:

<programlisting>
<?lp-file id="mainlisting" file="example.py"?>
<?lp-options preserve-newlines="no"?>
<?lp-section-id?>Main Listing<?lp-section-id-end?> =
<?lp-code?>
#!/usr/bin/python

<?lp-ref?>{Import needed modules}<?lp-ref-end?>
<?lp-ref?>{Initialize variables}<?lp-ref-end?>
<?lp-ref?>{Read in a line of input}<?lp-ref-end?>
<?lp-ref?>{Add up numbers from one to the number given}<?lp-ref-end?>
<?lp-ref?>{Print results}<?lp-ref-end?>
<?lp-code-end?>
<?lp-options preserve-newlines="yes"?>
</programlisting>
</para>

</sect1>

<sect1>
<title>Adding up the numbers</title>

<para>
Since adding up the numbers is the purpose of the program, we will
dive into that section first.  One of the benefits of literate program-
ming
is that you can go in any order you please.  Therefore, you can explain
the most important concepts first, even if they appear later in the source
file.
</para>
```

```
<para>
The code in this section will generate a range from 1 to
number + 1, and then iterate through and add them all to the
variable total.

<programlisting>
<?lp-section-id?>Add up numbers from one to the number given<?lp-section-
id-end?> =
<?lp-code?>
for num in range(1, number + 1):
 total = total + num
<?lp-code-end?>
</programlisting>

We now have a variable total that has not been initialized.  Although we
could do it here because Python allows that, we will put all variable
initializations in the "Initialize variables" section to show some additional
features of Literate Programming.  The total variable needs to be initialized
to 0.

<programlisting>
<?lp-section-id?>Initialize variables<?lp-section-id-end?> =
<?lp-code?>
total = 0
<?lp-code-end?>
</programlisting>

</para>

</sect1>

<sect1>
<title>Input and Output</title>

<para>
The input routine for this program is very basic - it simply attempts to
read a number from standard input and store it in the number variable.
To start with, we will initialize number to an invalid value.

<programlisting>
<?lp-section-id?>Initialize variables<?lp-section-id-end?> +=
<?lp-code?>
number = 0
<?lp-code-end?>
</programlisting>

Notice that this has been appended to the "Initialize variables" code segment
that was started in the previous section.  Now, we need to read in the
number.

<programlisting>
<?lp-section-id?>Read in a line of input<?lp-section-id-end?> =
<?lp-code?>
print """Please enter a positive integer, and I will add all numbers be-
tween one and
that number"""
textline = sys.stdin.readline()
try:
 number = string.atoi(textline)
 if number == 0:
  raise InvalidNumberError
except:
 print "You typed an invalid number.  Exitting..."
 sys.exit(1)
<?lp-code-end?>
```

```
</programlisting>

That code needs to have the string module and the sys module imported.
Therefore, we will add the imports to the beginning of the source file.

<programlisting>
<?lp-section-id?>Import needed modules<?lp-section-id-end?> =
<?lp-code?>
import sys, string
<?lp-code-end?>
</programlisting>

Finally, when the code is finished, we need to print the results.

<programlisting>
<?lp-section-id?>Print results<?lp-section-id-end?>
<?lp-code?>
print "When you add all the numbers from one to ", number, ", the re-
sult is ", total, "."
<?lp-code-end?>
</programlisting>

</para>

</sect1>

</article>
```

Obviously, that program was a bit long for what it intends to do. However, it's meant only as an illustration of how to do literate programming. In the next section, we'll go over the main processing instructions mentioned in the previous example.

Basic Processing Instructions

The processing of xmltangle is controlled by processing instructions. These instructions serve to designate what function different pieces of text have. Literate Programs are broken up into sections, which themselves can contain references to other sections to be included. Each section is identified by it's section ID. Section ID's are all lowercase, and only contain letters. However, within your document, they can look however you want - they can be mixed case and include punctuation, spaces, etc. However, all letters are converted to lower-case internally, and all spaces, punctuation, and numbers are stripped out. That means that the following section ID's are identical:

- My SEcTION
- {My section}
- My, Se c t i o n 2

All of them convert to `mysection` internally. There are three reasons to use a section ID. First, if you are creating a new section of code, you have to first give it's section ID. You can do that by saying <?lp-section-id?>My Section<?lp-section-id-end?>. This makes the current section be `mysection`. This also creates storage space for your code internally. Now, when you want to add code to this section, you begin it with <?lp-code?>. It then records your code in that section until it hits <?lp-code-end?>. You can add as many code segments as you want. If you want to add code to a different section, just close the current code segment, and then specify which section you want to work on with <?lp-section-id?> and <?lp-section-id-end?>. You can switch back and forth between sections as often as you like. Remember, however, that since you

are using processing instructions, the text you put here will be formatted as if the processing instructions didn't exist at all.

Being able to switch back and forth between code sections is a good feature, but what makes Literate Programming unique is that you can embed sections within each other. For example, if I think an algorithm would be obscured by all of the error checking code, I can simply put the error checking code in a different section, and simply refer to it. That way, the algorithm is more concise, and all of the error checking still gets done. To refer to another section in your code, you use <?lp-ref?> and <?lp-ref-end?>, and simply insert the ID of the section you want to include here between these processing instructions. I usually also but braces {} around the section ID so it is obvious that it's not part of the real code when it is typeset, like this

```
<?lp-ref?>{My Section}<?lp-ref-end?>
```

Obviously, you cannot have circular references.

Finally, in order to write out these sections to files, we have to specify which files contain which sections. Therefore, we have the single processing instruction <?lp-file?> which handles this. Each file can have exactly one section ID (of course, that section ID can include many others using <?lp-ref?>). To set the file `myfile.py` to point to the section ID `mysection`, you can simply do

```
<?lp-file id="mysection" file="myfile.py"?>
```

Finally, there is the <?lp-options?> processing instruction, which sets processing options. The only one used in this program is `preserve-newlines`, which, when set to `no`, will ignore any newline immediately following <?lp-code?>. Normally, you want it set to yes, because of stylistic issues which I won't go into here. However, on the top-level section of an interpretted program, you need it set to `no` to make sure that the `#!` line goes at the very top of the file. You usually want to set it back to `yes` immediately afterwards. Alternatively, you could simply start your code immediately after the <?lp-code?> processing instruction, like this:

```
<?lp-code?>#!/usr/bin/python
...
```

However, I think that looks ugly.

xmltangle quick reference

Note that all section ID's are normalized into the lowercase, characters-only form automatically.

lp-section-id
lp-section-id-end

This switches the "current" section being processed, and sets up internal storage for the data if it is the first time it is used.

lp-code
lp-code-end

This creates a segment of code within a section. It is an error to use this before having issued an lp-section-id/lp-section-id-end pair.

lp-ref
lp-ref-end

Inserts another section into the current one. The other section does not have to be defined yet. Simply put the section ID between these two processing instructions.

lp-file

This sets what section ID is the top-level section for a given file. Has two parameters - id and file.

lp-options

This allows you to set processing options. The following are available.

preserve-newlines

When set to `no`, a newline immediately following <?lp-code?> will be ignored. When set to `yes` it includes that newline in the code. Defaults to `yes`.

strict-options

If set to `yes`, it will give warnings for options that don't exist. Otherwise, if you specify an option that doesn't exist, it will just ignore it.

lp-block
lp-block-end

This is a convenience instruction, which allows xmltangle to automatically increase indentation upon tangling. After an lp-block instruction, all lines are indented by 1 tab. After two such instructions, they are indented by 2 tabs. This is especially useful in languages like Python, where indentation has meaning. This way, you don't have to memorize where you are on indentation in subordinate code pieces. You can just back everything to the left margin, and let your previous lp-block statements set the correct tabbing.