Down-Translating XML: The Python Way

Alberto Simões
2Ai – School of Technology, IPCA, Barcelos, Portugal

José João Almeida
Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

Nowadays, the most used approach to process an XML file is based on the processing of a DOM structure and a set of operations that collects or edits information in the model using some kind of selectors (usually CSS-like or XPath).

Nevertheless, the process of performing a depth-first walk through the DOM, and synthesizing values, is a simple way to traverse and transform an entire XML document.

In this document we discuss the details on the implementation and usage of a Python package for XML document processing based on this structure. Given the existence of similar tools for other programming languages, we will mainly focus on the used approach, that takes advantage of the Python style guides and development patterns.

2012 ACM Subject Classification Software and its engineering → Extensible Markup Language (XML); Software and its engineering → Scripting languages

Keywords and phrases XML, Python, Depth-First Processing

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.15

Supplementary Material *Software (Python Package)*: `https://pypi.org/project/xmldt/`
Software (Source Code): `https://natura.di.uminho.pt/svn/main/python/XML-DT/`

Funding *Alberto Simões*: This project was funded by Portuguese national funds (PIDDAC), through the FCT – Fundação para a Ciência e Tecnologia and FCT/MCTES under the scope of the project UIDB/05549/2020.
José João Almeida: This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

The Down-Translation is the name used in other libraries, for Perl [1] and JavaScript [7], to refer to the approach of processing an XML Document Object Model (DOM) tree doing a depth-first traversal, and synthesizing values.

The idea is simple: define a mashup of a dispatch table and visitor patterns [3]. The programmer can define a function to process each type of node in the XML file. This function returns (or synthesizes) the value of processing its children. This returned value is used by its parent, and the values are synthesized up to the root of the document object model tree.

While the approach is known, in this article we describe the implementation of such a processor for the Python programming language. This implementation differs from the other two mentioned above as it does not use an explicit dispatch table, but uses Python's own symbol table for that purpose. This makes it easy to define XML processors as simple classes, that can even inherit from other XML processors, allowing more specific approaches to take advantage of other, more general, processors.

The article is structured as follows: Section 2 motivates for the use of Down-Translation approaches for XML Processing. Section 3 discusses the implementation decisions and presents some details on how Python features were used to provide an interesting programming interface. Section 4 shows some examples of DT Processors. The article concludes with an analysis of the obtained tool and future steps.

2 DOM vs DT processing

Currently, most libraries for processing XML documents focus on the construction of the DOM and allowing the programmer to search in the tree for specific structures. These structures are usually encoded using the Cascading Style Sheets (CSS) [4] syntax for element selectors, or by using the XPath [6] language path specification.

The programmer instructs the XML library to find the set of elements in the desired positions, and to extract the required information. Most of the time, these elements are not processed directly, but the children are visited following some other path, in order to find the relevant information. This process is repeated until all the desired information is obtained.

Whenever a selector or a XPath path is used, the complete DOM tree needs to be processed. Thus, every time a new query is performed, the library requires to process the tree and find the relevant elements.

With a Depth-First approach, the DOM tree is still required, but the goal it to traverse it only once, from its leafs to its root. During this process the programmer is able to construct whatever he needs as an answer. It can be the creation of other XML document, and therefore, returning a string or XML objects in each visitor, or to return a specific data structure that represents the desired information. As a third option, each visitor can store the desired information in any data structure, as side effects.

3 Python Implementation

In this section we will describe the details on the XmlDt module implementation. First we will describe how the dispatch table of visitors is implemented, as well as the visitors for special elements, like comments or textual data. Then follows the list of special methods that the programmer can use on each visitor. Finally, a discussion on the support of hooks and the chaining of visitors.

3.1 XML Element Dispatch Table

The Perl and JavaScript versions of the Down-Translation approach are based on dispatch tables. These are implemented using dictionaries (hash tables or objects, to use the terminology for each language). The dictionaries store a relation between each XML element names and an anonymous functions (or lambda expression) that is responsible for processing those elements. While this is a compact method to define such processors in these two languages, for Python it is not so clean, as Python lambda expressions is limited to a single statement, as the typical way of using lambda expressions in Python that require more than one statement is to implement a named function that is, then, called by the lambda expression. This approach would require the programmer to implement a named function for each element being processed.

Taking this into consideration, the module was implemented using Python's reflection capabilities. The XmlDt module offers a super-class that the user should derive to create each processor. The methods on this subclass are automatically assigned for processing the XML elements with the same name. This is performed during run-time, when XmlDt looks-up each XML element name in the class symbol table.

The programmer's work is just to define a subclass for XmlDt, and implement methods for each XML element needs processing. These methods are visitors of the XML element nodes. They receive the processor object (as they are methods) as well as the element they are visiting.

Listing 1 Simple XmlDt to replace `<b>` and `<i>` elements by `<emph>` elements.

```
class ReplaceBoldItalic (XmlDt):
   def i(self, element):
      element.tag = "emph"
      return element.xml

   def b(self, element):
      element.tag = "emph"
      return element.xml

processor = ReplaceBoldItalic()
print(processor(filename="something.xml"))
```

While interesting, this approach has two main drawbacks, that needed to be considered:

- The super-class should not pollute the symbol table, as any name used for a method might be used in an XML element. To minimize this problem, all methods inside the XmlDt package are prefixed with an underscore. While those are valid names for elements on XML files, they are not common.

  Note that Python has two distinct use of the underscore in method named. When using only one underscore in the beginning of the method name, it is used to denote that the method is private (while this is not enforced, it is the common *pythonic* way of describing that property). There is also the possibility of prefixing the methods with two underscores, and add two underscores at the end of the method name. These methods are special methods that Python defines, and allow the implementation of special functionalities, like the definition of a constructor or an index-based accessing function.

  XmlDt uses these two types of methods. The first for internal methods, that are not expected to be sub-classed. The second, for internal methods that include basic implementation/behavior, but that can be extended by the programmer (thus, following the same *pythonic* approach for other similar situations).
- From the W3C standard, the valid characters for XML element names are all valid as method names, except the period and the dash/hyphen. To address this problem, a Python decorator[1] was implemented. Python decorators are special methods invoked at compile time. They decor or annotate methods, and allow to add extra information to the method. This decorator allows the programmer to implement the visitor with any valid name, and assign it to another tag name. It can be used for any tag, but it mostly useful when tag names include any of these two characters.

A simple example of a XmlDt processor, that processes an XML document and replaces all `<b>` and `<i>` element names into `<emph>` elements is presented in Listing 1

When requiring to process an element whose name is not a valid Python method name, the decorator mentioned before should be used. The decorator is named `dt_tag` and must precede a method. This decoration specifies the XML element name that will be processed by the visitor described. Listing 2 shows a simple example of how this decoration can be used, to process the `<TEI.2>` root element.

[1] `https://peps.python.org/pep-0318/`

Listing 2 Method decorated with the name of the XML element the visitor should process.

```
class processTEI (XmlDt):
   @XmlDt.dt_tag("TEI.2")
   def process_tei_two(self, element):
      # do something
      return element.xml
```

3.2 Special element handlers

There are some other handlers that are useful, but that are not related directly to XML elements, like the processing of text content of elements, processing comments, or even, the possibility to define a default visitor (that will be called whenever a visitor is not found for a specific XML element).

These specific methods have default behaviors defined in the XmlDt module, just like other double underscore methods that are available builtin in Python. That is the reason we chose this syntax for visitors that have a default behavior, but that can be overridden by the programmer.

PCData the XmlDt module implements the method `__pcdata__` that processes text nodes in the DOM tree. While there is a default visitor, that returns the text with no change, the programmer can implement its own visitor. This visitor receives the processor object and the text, as a string.

CData by default XmlDt does not distinguishes between PCData and CData elements.[2] Nevertheless, it is possible to configure XmlDt to distinguish them. In this case, the `__cdata__` visitor is called whenever such an element is found.

Comments it is not common to process an XML document and take actions accordingly with its comments, but given that the underlying library has access to them, XmlDt allows the programmer to perform actions on such elements, overriding the `__comment__` method. By default, this method has no action.

Default when processing large XML documents, it is common that a specific set of elements have specific visitors that need to be hand tailored, but for the majority of the elements, their visitors are always the same. With this in mind, the `__default__` method can be overridden to define the default behavior for elements that do not have a visitor.

As an example, consider a processing tool that processes an XML document replacing each word by its stem[3]. Listing 3 shows how this can be performed Using the NLTK library [2] and XmlDt using two special methods, one to process text contents, and the other to construct back XML elements from each node in the original tree.

3.3 XmlDt methods

There are some specific functionalities that can be useful for the programmer. While one of the goals was not to pollute the name space for the user module, we still decided to define standard methods, making them start with the `dt_` prefix. If by some reason these names are required as element names, the user can still use the `@dt_tag` decorator referred previously.

---

[2] In this sense, follows the same approach of the `lxml` (`https://pypi.org/project/lxml/`) library, that is the module used by XmlDt to construct the DOM tree.

[3] A stem is a kind of root for similar words [5].

Listing 3 Stemming the contents of an XML file.

```
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
from xmldt import XmlDt

ps = PorterStemmer()

class XmlStemmer (XmlDt):
   def __pcdata__(dt, txt):
      words = word_tokenize(txt)
      return "␣".join([ps.stem(w) for w in words])

   def __default__(dt, el):
      return el.xml
```

It might be important to stress here why we have two distinct naming conventions. These are methods offered by the library, that are not supposed to be overridden by the programmer. The double underscore methods are offered as default methods that can be overwritten.

The most relevant methods and getters are:

`dt_in_context` This method is used to check the current visitor's path, allowing the user to define custom behaviors accordingly with the element context. It receives one or more element names and return a boolean value if the current node's path include those elements.

`dt_parent` This is an accessor to the current parent's element, allowing the addition of further information in that element. There is a similar method to access to grand-parent element: `dt_gparent`.

`dt_root` Similar to the accessor shown above, this allows the direct access to the document root element.

`dt_path` Returns the current path of elements (does not include the element itself). This is not a list of element names, but a list of the elements themselves.

3.4 Hooks and Chaining

To allow sub-classes to expand the way methods are searched in the module symbol table, XmlDt uses the concept of a hook list, where the programmer can include new code to decide which method to call for processing a specific element. This is useful, for instance, in the HtmlDt class (described in section 4.2), where this feature is used to allow the programmer to define methods to handle specific element classes (the `class` attribute from HTML).

While this feature is quite useful, it raises a problem: what visitor to call when an element has more than one class, or if there is a visitor function for the specific class and for the element name itself. To allow multiple visitors to be called, there is a special method, `dt_chain(el)`, that, whenever is called with the current element, makes XmlDt to call the next available visitor.

The order of processing elements is as follows: first all hook methods are checked. After hooks, the main processor symbol table is searched. Finally, the default processor is used. When more than one hook method is registered, the used order is relevant to understand how processors are searched.

Listing 4 Implementation of a XML grep command line tool.

```
from xmldt import XmlDt
import sys
import re

if len(sys.argv) != 3:
   print("Usage: dtgrep pattern filename")
   sys.exit(1)

_, pattern, filename = sys.argv

class Grep (XmlDt):
   def __pcdata__(self, text):
      if re.search(pattern, text):
          xpath = "/".join([e.tag for e in self.dt_path]
          print(f"{xpath}: {text}")
      return text

grep = Grep()
grep(filename=filename)
```

4 Using XmlDt

In this section we present three different usages of XmlDt:

1. A simple `grep` command line tool, that searches for a pattern in elements contents, and present the user with information about the hit;
2. A subclass of XmlDt, named HtmlDt, for processing HTML files, and using some functionalities especially useful for this kind of documents;
3. A tool to help the process of creating a XmlDt processor based on sample XML files.

4.1 XML Grep

This is a straightforward and typical use of XmlDt. The user specified a pattern and a XML file, and the elements which content match the supplied pattern are presented as output.

The program presented in Listing 4 includes the relevant parts of dealing with XmlDt. The details on processing command line arguments were simplified: the first argument for the command line tool (thus, `argv[1]`, is the regular expression, and the second is the file to process.

This solution uses the `__pcdata__` handler to perform the search and, whenever a match is found, to print the path to that element as well as the element contents.

4.2 Processing HTML

One of the benefits of implementing the tool as a class is the possibility to create sub-classes that inherit the default behavior of the main tool and add new features.

This allowed the creation of the HtmlDt module, that inherits the default functionalities of XmlDt and adds some specific tools to easy the parsing of HTML documents.

The implementation of HtmlDt defines:

- the constructor for the parser including the attribute required by `lxml` to process HTML files;
- a decorator, named `html_class`, that allows the programmer to specify a function that will process HTML elements that belong to a specific HTML *class*;

While the first item has a very simple implementation, just overriding the default constructors, the second requires some Python *reflection*, that is, the ability to query and change details on the program while it is executing. In this case, a hook is added to XmlDt that searches in the Python symbol table for methods decorated with the `html_class` property. The hook runs before any other method from the dispatch table, and checks if the *class* attribute is present in the element being currently processed. Whenever an element with a known class is found that specific processor is called.

Listing 5 shows an example of using the `html_class` decorator, to specify that, whenever an element is found with the *bold* class, the element should be changed to the `<b>` element.

Listing 5 Example of usage of the HtmlDt module.

```
class AddBold (HtmlDt):

    @HtmlDt.html_class("bold")
    def set_bold(self, e):
         e.tag = "b"
         return e.xml
```

While the HtmlDt class has less than 40 lines of code, the details of handling the decorators is not simple, and are outside of the scope of this article. The most interesting part if the hook that processes classes, and is shown in Listing 6. This hook is registered in the constructor of the HtmlDt module. The hook, itself, receives and element and looks for the *class* attribute. Existing, its value is replaced by a list of all the used classes (splitting the class list by the space character, as defined by the HTML standard). Then each class if visited, once at a time, following the order of use. If there is a specific method for that class, then it is added to a list, that is returned at the end of the hook. This list is, at last, the sequence of functions to be called by the XmlDt class.

Listing 6 Handler for class-based hooks.

```
    def _class_hook(self, element):
        handlers = []
        if "class" in element:
            element["class"] = re.split(r'\s+', element["class"])
            for c in element["class"]:
                if c in self._classes:
                    handlers.append(self._classes[c])
        return handlers
```

4.3 Base Processor Generator

The XmlDt package distribution includes a small utility that allows to create bare processors based on specific XML instances. The tool receives a list of XML files, processes them, and generates a Python script with the dispatch table for all the required visitors.

The tool has a diverse set of options, that will not be discussed here. Nevertheless, its basic structure is quite simple. Given the ability to describe processors based on generic XML objects (namely the `__default__` handler), it is possible to gather information about every tag element present on the supplied XML files, as well as what attributes are used for each of these elements. Listing 7 shows a simplified version of the default element handler.

Listing 7 Simplified `__default__` handler for processor generation.

```
def __default__(self, element):

    # register tag
    if element.tag not in self.data:
       self.data[element.tag] = {":count": 0}

    # count number of occurrences for this tag
    self.data[element.tag][":count"] += 1

    # save information about existing attributes
    for key in element.attrs.keys():
        if key not in self.data[element.tag]:
           self.data[element.tag][key] = 0

        # count attribute usage
        self.data[e.tag][key] += 1
```

After processing the set of sample files, and storing all the information in the `data` property, the tool uses a text template to generate the processor. It also includes details on the number of occurrences of each element and attribute in the XML files as Python comments. This information can be useful for the programmer to structure the processor.

5 Conclusion

In this short paper we describe a specific Python tool for processing XML documents. It is based on two distinct patterns from software development: the dispatch table and the visitor patterns.

The usage of these patterns with the Down-Translation approach, and the power of the Python language, namely in terms of reflection, allows the creation of elegant XML processors.

The tool is available both at PyPi[4] and is open source[5].

While the current status of the tool makes it usable and useful, a set of improvements are in place for development:

- Add support to process URL directly;
- Add support to handlers based on the element identifier and in the element language, taking advantage of the `@id` and `@xml:lang` attributes;
- Allow proper debug of the order handlers are called when more than one can be used for a specific element (for instance, an element might be processed by a handler based on tag name, or HTML class, or identifier, or language);
- Add support for CSS selectors for handlers.

[4] `https://pypi.org/project/xmldt/`
[5] `https://natura.di.uminho.pt/svn/main/python/XML-DT/`

References

1 José João Almeida and José Carlos Ramalho. XML::DT a perl down-translation module. In *XML-Europe'99, Granada - Espanha*, May 1999.

2 Steven Bird, Edward Loper, and Ewan Klein. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.

3 E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994. URL: https://books.google.pt/books?id=6oHuKQe3TjQC.

4 Ian Hickson, John Williams, Daniel Glazman, Peter Linss, Elika Etemad, and Tantek Çelik. Selectors level 3. W3C recommendation, W3C, November 2018. URL: https://www.w3.org/TR/2018/REC-selectors-3-20181106/.

5 Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

6 Jonathan Robie, Josh Spiegel, and Michael Dyck. XML path language (XPath) 3.1. W3C recommendation, W3C, March 2017. URL: https://www.w3.org/TR/2017/REC-xpath-31-20170321/.

7 Alberto Simões. XML Parsing in JavaScript. In Ricardo Queirós, Mário Pinto, Alberto Simões, José Paulo Leal, and Maria João Varanda, editors, *6th Symposium on Languages, Applications and Technologies (SLATE 2017)*, volume 56 of *OpenAccess Series in Informatics (OASIcs)*, pages 9:1–9:10, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/OASIcs.SLATE.2017.9.