Introducing SAX

An application developer can choose any one of a number of strategies to read and use an XML document. In some very simple examples a script containing a number of regular expressions might do the job, but normally a more rigorous technique is required. The Simple API for XML – SAX - is one of the two key techniques for analysing and processing XML documents, the other being the more complicated Document Object Model – DOM.

SAX is not a formal standard in the usual sense of the term. It is not ratified by the W3C, or any other organisational body. SAX emerged in late 1998 as a de-facto standard API, and SAX version 2 (known as SAX2) has been a rock solid standard since its appearance in 2001. The original SAX (which was never called SAX1 but could have been) is now obsolete. This article refers to SAX2.

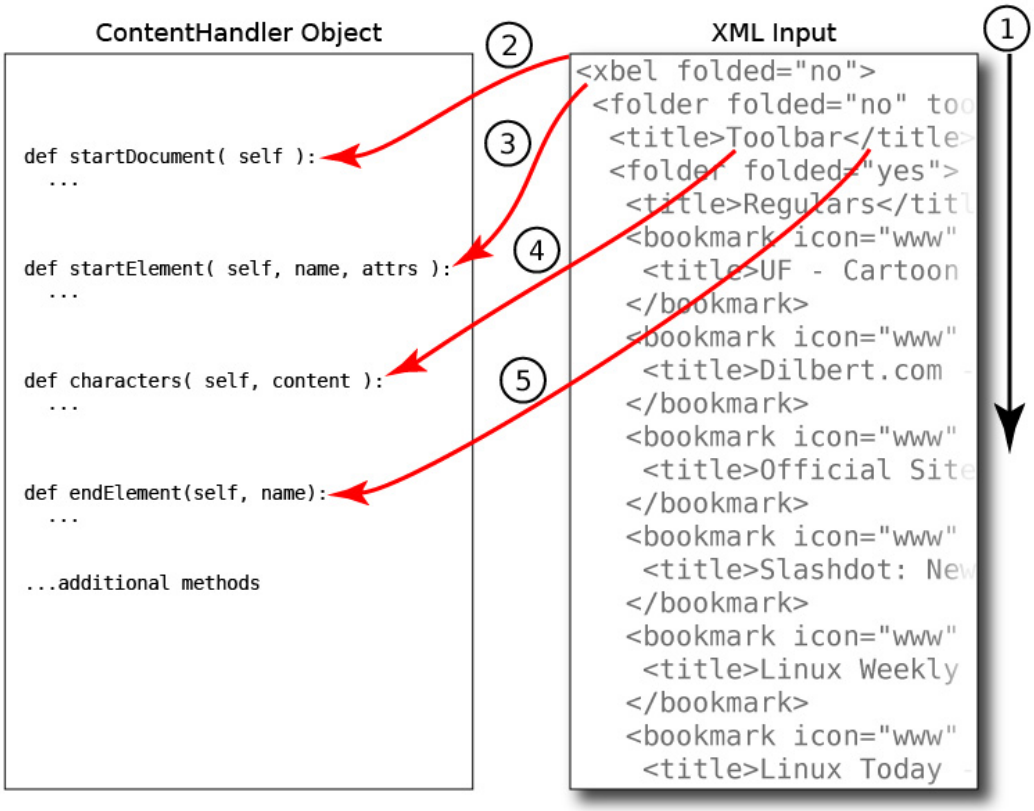
SAX is defined by an interface, which means all implementations, regardless of language, will share the same class names for the key components, and those classes will have the same methods with the same formal arguments. SAX maintainer David Brownell keeps the definitive version of the interface implemented in Java, and most SAX documents and examples are still Java based. However, the SAX2 API has been a standard library in Python since version 2.0 of the language, and it is this implementation this article concentrates on.

Event based Parsing

SAX uses an event based parsing technique which prevents it from having to build a potentially huge document tree in memory, which is the technique DOM uses. Although not ultimately as flexible as a tree, event based parsing means that SAX can process large a XML document on low end hardware, and therefore achieve results which wouldn't be possible using DOM.

Event based parsing involves the XML parser reading the data file and generating "events" when it sees interesting points in the XML structure. The word "event" is possibly something of a misnomer, since it implies something dynamic or asynchronous has happened. In fact, all that really happens is that the parser calls the standard methods of a SAX "ContentHandler" object when certain points of interest are seen in the XML.

For example, referring to Figure 1, the parser processes the XML exactly once, from top to bottom (1). When it sees the start of the XML document it calls the content handler object's startDocument method (2). Each time it sees an element's opening tag it calls the startElement method (3). For data characters in the XML it calls the characters method (4), and for each end tag it calls the content handler's endElement method (5). The SAX API actually defines about a dozen of these "events" which can be encountered during a parse of an XML file, and which can be handled by a method in the content handler object. (Note that the numbers on the diagram are only for identification purposes in this text. They don't represent the order in which real events occur – that is entirely dependent on the nature of the XML file being parsed.)



Building a Parser and Content Handler in Python

Using an XML parser and a SAX ContentHandler from Python is simple – write the content handler class to implement the application logic, create a parser object and give it an instance of the content handler class, then call the parser with the XML file to examine. Despite its pivotal nature to the whole operation, the parser itself is a transitory object. Once the parse is completed the parser's job is done and it normally has no further use. The simplest parse looks like this:

```
#!/usr/bin/python

from xml.sax          import make_parser
from xml.sax.handler import ContentHandler

parser  = make_parser()
handler = ContentHandler()

parser.setContentHandler( handler )

parser.parse( "testfile.xml" )
```

Here I use an instance of the ContentHandler class as my content handler. ContentHandler contains an empty implementation of each of the SAX interface methods, so used by itself it's useless. In normal code its purpose is to act as a base class that can be extended with code that implements application logic. This is where the magic behind SAX happens, so let's have a closer look at how a content handler is written.

The Content Handler Object

Basing an application's content handler class on the ContentHandler from the Python SAX library isn't essential, but it is the recommended route unless you intend to implement every method of the SAX content handling interface. Basing your own content handler on the ContentHandler class gives you an empty method for each method in the interface, so you only need to reimplement the ones you need.

Once the parser is running, the content handler is solely responsible for keeping track of what is happening in the XML. The parser can't back up or give any clues about where it is or what it has seen, so keeping track of what is happening is up to the content handler. The content handler sees the parts of the XML document very much in isolation from each other, so it is normal practice to use a set of flags in the content handler object in order to maintain context.

For example, suppose an application is looking for some telephone number text held inside an telephone element. The telephone number will character data, so it will be up to the `characters` method to extract the required string. But the `characters` method will be called once for every text string in the XML document, so how does it know which one is the telephone number? The answer lies with the `startElement` method, which will be called at the start of each element. When startElement sees an opening telephone tag, it must set a flag which the characters method can later check. If the flag is set the characters method knows that its parameter text is a telephone number. The endElement method then gets the responsibility of turning the flag off when it sees the closing telephone tag. The code would look something like this:

```
class TelephoneLocator( ContentHandler ):
  _telephoneFlag = False

  def startElement( self, name, attrs ):
    if name == "telephone":
      self._telephoneFlag = True

  def endElement( self, name ):
    if name == "telephone":
      self._telephoneFlag = False

  def characters( self, content ):
    if self._telephoneFlag:
      # content contains the telephone number text
```

An extra complication is that there is no guarantee that the characters method will be given all the text data in one go. It is perfectly valid for a parser to call `characters` with the first half of a text string, then immediately call it again with the second half. For this reason the characters method should only collect and build text strings; it's the endElement method which should actually deal with the string, because it's not until the end tag is reached that it can be certain the string has ended. (For mixed content models the startElement method can also identify the end of a string.) With this in mind, the content handler code should really look more like this:

```
class TelephoneLocator( ContentHandler ):
  _telephoneFlag = False
  _telephoneNo   = ""

  def startElement( self, name, attrs ):
    if name == "telephone":
      self._telephoneFlag = True
```

```
def endElement( self, name ):
  if name == "telephone":
    self._telephoneFlag = False
    # This is the point where we can use self._telephoneNo

def characters( self, content ):
  if self._telephoneFlag:
    self._telehoneNo = self._telephoneNo + content
```

Extending this example a little, let's say that the XML will carry a name element containing the name of an individual, followed by the telephone element. Now other information needs to be collected and acted upon. Firstly, the name to compare against the search criteria needs to be collected as a text string just like the telephone number. This means another flag must be set by startElement when the name tag is seen, and the characters method gets the additional responsibility of collecting the text data that is found when the nameFlag is set. When endElement sees the closing name tag, it needs to compare the name string with the search criteria to see if the telephone number that is going to follow is the required one. All this might seem rather complicated, but it's actually a lot easier to implement in code than it is to explain in words. Also, this sort of design is typical of SAX content handlers, so once you've done one or two it becomes natural.

The deeper into the XML the required data is, the more flags and partially built text strings are required, so it is not uncommon to have to build an object internal to the content handler which has the sole job of keeping track of the context. Without such an object the content handler can get very cluttered.

Another frequent requirement in content handler code is whitespace handling. Most text strings that come from an XML document into the characters method have whitespace attached to them. This is normally just an artefact of indenting in the XML file, so the easiest way to deal with it is simply to strip it off the start and ends of the string and forget about it. Sometimes, however, it is required, especially in programs that need to exactly recreate the input XML. Under these circumstances the varying nature of whitespace in XML documents needs to be taken into consideration, for example by using regular expressions instead of straight string comparisons.

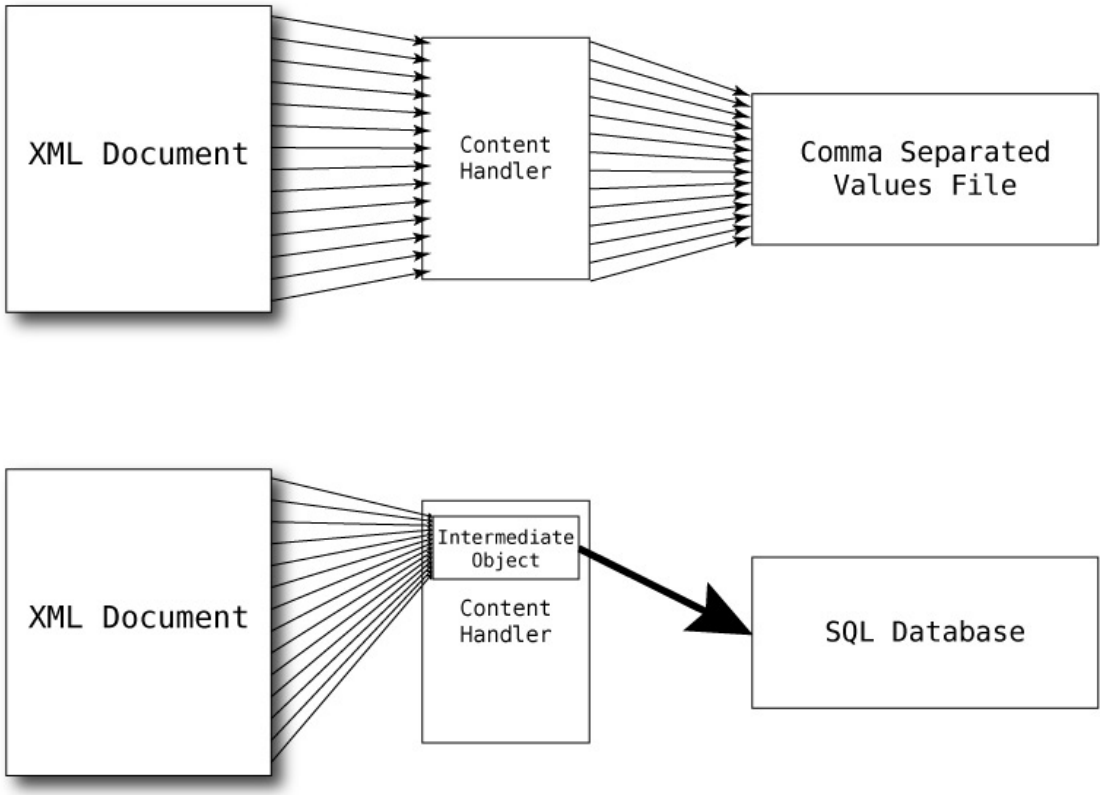
Content Handler Techniques

When writing a SAX application that processes an XML document, the developer has to make a choice regarding when to do the actual data processing. The options are to do it on the fly, directly from the content handler, or to store up the relevant information and do it all after the parse is completed.

SAX is often used instead of DOM because of its low memory requirements, so the most popular approach is to perform actions on the fly. This means that as the XML structures are read in, the content handler processes them and immediately generates the required output for that section of data. The top half of Figure 2 shows how this approach is suited to generating a comma separated values (CSV) file. Each line of CSV can be written out as soon as enough information has been collected. If an error is encountered halfway through the XML, the error can be caught and the partially completed CSV file can be deleted.

Sometimes, however, processing on the fly doesn't suit the application. For example, if you are looking to build a set of SQL updates from the XML document you could do them one at a time, on the fly, but that might have repercussions on your database – either stressing the server with hundreds of queries, or presenting rollback problems if your program hits a problem halfway down. Under these circumstances it might be

better to collect the required data into a Python dictionary (or object if necessary), and, once the parse has successfully completed, build and issue a single SQL update. See the bottom half of Figure 2. Obviously memory limitations come back into play if you use this approach, but a Python dictionary will be much more memory efficient than an entire DOM tree.



When using this technique the data will get built up inside the content handler object, which leaves the question of how to get to it after the parse is completed. There are two options. The first is to put the relevant code into the content handler's endDocument method. This method is called by the parser when the end of the XML document is reached. The second option is to add more methods to your content handler class to help access the information stored. A content handler class obviously implements the SAX ContentHandler interface, but there's nothing to stop you adding a `queryCollectedData` method to allow you to retrieve information after the parse has completed. The content handler has a specific purpose but you can treat it as a regular object as well.

Conclusion

SAX is the simplest of the XML handling techniques, and is ideally suited to collecting and processing data from large XML documents, especially on limited hardware. Imaginative use of SAX – either in the form of a thoughtfully implemented content handler, or other tricks such as using 2 or 3 consecutive SAX parses to complete a job - can sometimes allow a task to be achieved in a much simpler fashion, and on much cheaper hardware, than would be possible using a DOM approach.

Sidebar: Processing Instructions

Processing instructions are instructions in the XML that can control, or at the very least, attempt to influence, the processor that is working on the XML file. Processing instructions start with <?, followed immediately by the name of the "target", which is

the processor to which the instruction is intended. The processing instruction then contains further text (the instruction's "parameters" if you like) before ending with ?>. Although there is no mandated control over what processing instructions contain, there are a few de-facto standard ones like the <?xml-stylesheet ?> instruction that is always used to attach a stylesheet to an XML file. It is also conventional to add "parameters" to processing instructions in name="value" format, but this isn't actually a requirement.

Since your Python script is the processor for your XML, you can, if you want, add processing instructions in the XML which your script can capture, examine, and act on. Processing instructions are caught by the XML parser, which calls the content handler's processingInstruction method with two parameters – target and data.

You might, for example, decide to control your SAX script with a processing instruction that you give the target name "conversions", and that takes a data set in name="value" format:

```
<?conversions currency="USD" weight="lb" distance="mile" ?>
```

When your processingInstruction method is called, it can check the target string to see if it is "conversions". If it is, it can split the data into name/values pairs and from them set flags in the content handler to later invoke code that converts currencies, weight and distances to the units requested.

Bio

Derek Fountain is a freelance writer and software developer specialising in Linux and open source scripting languages. He lives in Perth, Western Australia.