EasySAX: Sax Made Pythonic

About Me

- Paul Prescod
- ISOGEN Consulting Engineer
- Professional Services Arm of DataChannel[1]
- Co-author, XML Handbook[2]

Laying the foundations

Overview

- EasySAX merges ideas from
  - SAX[3]
  - DOM[4]
  - XSLT
  - DSSSL
- Probably most similar to "saxon" in Java world.

What is Python?

- A high-level, object oriented, dynamically typed programming language[5].
- Can be used for scripting, conversions, abstraction-building.
- A merge of the best ideas of Smalltalk, Perl and Java?
- What does it mean to be Pythonic?

Meditate on this

- An object is Pythonic if it has the Python nature.
  - There is no rule of thumb.
  - There is no motto.
  - There is no overriding design goal.
  - There is only the oneness with the problems you are trying to solve and the way you think about it.

But master....

- How do we design things that are Pythonic?
  - Aristotle's golden mean
  - Make them simple, but not too simple to get the job done.
  - Elegant, but not in a cutesy way.
  - Flexible, but not at the expense of clarity.
  - Dynamic, but not at the expense of maintainability.
  - Dictionaries are KUEL. Use them alot.

Reflect on this

- At runtime it is possible to ask questions about objects.
- Reflection is a powerful tool.
- It can be used in diabolical ways
- When used virtuously it eases readability and maintenance
- This is Pythonic.

Reflecting on SAX

What is SAX?

- SAX is a low-level API to XML
- Performance is a key consideration
- It is simple, but not easy.
- Increasingly, it is no longer simple.
- It is relatively, but not completely, complete.

Does SAX have the Python Nature?

- Complexity is Pythonic if it is hidden.
- Good performance is Pythonic.
- Standards conformance is Pythonic.
- Re-inventing wheels is **not** Pythonic.
- Therefore we must use SAX, but hide it.

What must be hidden?

- SAX character handling is inelegant.
- SAX gives you a pointer into a buffer.

```
def characters(self,ch,start,length):
    print ch[start:start+length]
```

- Pythonistas would just expect a string object.

```
def characters( self, chars ):
    print chars
```

Event Dispatching

- SAX requires you to dispatch your own element events:

```
class MyHandler( SaxHandler ):
 def startElement(self,typename,attrs):
    if typename=="html":
      handleHTML(attrs)
    elif typename=="title":
      handleTitle(attrs)
    ...
```

- Large switch statements do not have the Python nature.

Context

- SAX requires application programmer to take care of context.

```
class MyHandler( SaxHandler ):
    def startElement( self, typename, attrs ):
        if typename=="html":
            handleHTML(attrs)
        elif typename=="title":
            titleMode=1
           ...
    def characters(self,chars,start,length):
        if titleMode:
            print chars[start:length]
```

SAX and Namespaces

- Namespaces are the ultimate koan.
- Do we keep prefixes?
- How do we keep them?
- How do we do comparisons?
- How do we keep this all efficient?

Do SAX events suck?

- No, they merely have not achieved enlightenment.
- Through a series of reincarnations we can move them towards enlightenment.
- At the end, is what is left still SAX?
- Ponder.

Does the DOM suck?

- Design elegance aside … the DOM is useful
- Nevertheless, it's major weakness is not a design flaw:
  - tree models are inherently weak at handling very large documents
  - this can be mitigated with an object database like ZODB
  - but you still need a lot of disk space
- A Pythonic SAX must have minimal memory requirements

Towards a Pythonic SAX

First Principle

- Do not reinvent the wheel.
- Parsers can still "speak" SAX
- Applications can use something more Pythonic
- Raw SAX is still available for speed-critical M2M B2B XML EDI on WinCE HPCs

Second Principle

- Let's steal ideas wherever we can.
  - XSLT
  - DSSSL
  - DOM
  - Omnimark
  - Balise
  - ...

Stealing from the DOM

- It takes humility to steal ideas from the DOM.
- Therefore it is a productive exercise.
- DOM 2 has a way of handling namespaces.
- The DOM is really good at handling context:
  - node.parentNode
  - node.childNodes
  - node.childNodes[0]
  - node.attributes
  - node.getAttribute("abc")
  - node.parentNode.getAttribute("abc")

SAX, meet DOM

- Instead of dispatching strings and integers, we can dispatch nodes:

```
def startElement( self, elementNode ):
    ...
def endElement( self, elementNode ):
    ...
def text( self, textNode ):
    ...
def processingInstruction( self, piNode ):
    ...
def comment( self, commentNode ):
    ....
```

- This gives us a way to navigate around.

Leveraging context

- Given that we have context...let's flaunt it!

```
def handle_spam( self, textNode ):
    "figure/title/text()"
    print "Figure title:"+`textNode`

def handle_dead_parrot(self,textNode ):
    "section/title/text()"
    print "Section title:"+`textNode`
    print textNode.parentNode.\
          attributes["type"]
```

Let's not get crazy

- There are some rules...
- Not all of the DOM is available (see next slide)
- Handlers must be named handle_something
- "something" is a symbolic label, not a “tagname”
- Particular nodes are matched against the XPath

How much DOM can we afford?

- The "right" amount of DOM varies from application to application.
- In processing techdocs it is really useful to be able to have a complete DOM for (e.g.) tables and figures.
- Parent context is almost always useful and relatively cheap.
- Therefore: always remember parents.
- Otherwise, only build subtrees for regions of the document.

Selective Domination

```
def handle_applets(self,elementNode):
    "applet as tree"
    for node in elementNode.childNodes:
        print node

def handle_tables( self, elementNode ):
    "table as tree"
    # do something
    self.processChildren( elementNode )
    # do something else
```

ProcessChildren

- Recursively invoke handler on children
- Like DSSSL function of same name
- Like XSLT apply-templates
- Like Omnimark %c
- Coming soon...processMatchingChildren

Other DOM costs

- The DOM is large and getting larger.
- It is complicated and redundant.
- Most parsers don't generate most node types.
- Most apps are read-only
- It probably would not pass the Guido test.
- Let's just make a subset: "minidom".

Namespaces

- Namespaces can be registered before you start parsing.
- You can fiddle with the namespace list while parsing (but would you?)
- You use prefixes in XPaths, just as in XSLT:

```
class MyHandler( EasySAXHandler ):
    def __init__(self):
        self.registerNamespace( "xhtml",
            "http://www.microsoft.com" )

    def handle_tables( self, elementNode ):
        "xhtml:table as tree"
        # do something
        self.processChildren( elementNode )
        # do something else
```

Garbage Collection

- Children know about their parents.
- By default, parents do NOT know about their children.
- When you build a tree, the parents do know about children.
- The references from parents to children are destroyed when handler completes.
- "Weak references" would help here.

Credit where Due

- I wrote "minidom"
- James Clark wrote Expat
- Dr. Dieter Maurer <dieter@handshake.de> wrote the biggest component: the XPath[6] parser
- Thanks to his good design, I could adapt it without any help from him.
- XMetaL[7] wrote these slides.

Todo...

- Documentation, documentation, documentation.
- Tree pruning?
- User defined functions.
- XSLT "modes"?
- Other DOM facilities.
- Python "xml library"?

Referenced URLs

[1]http://www.datachannel.com

[2]www.xmlbooks.com

[3]www.megginson.com

[4]http://www.w3c.org/TR

[5]http://www.python.org

[6]http://www.dieter.handshake.de/pyprojects/pyxpath.html

[7]http://www.xmetal.com