The ElementTree iterparse Function

The new **iterparse** interface allows you to track changes to the tree while it is being built. This interface was first added in the cElementTree library, and is also available in ElementTree 1.2.5 and later.

Recent versions of ~~lxml.etree~~ (dead link) also supports this API.

Usage

To use **iterparse**, just call the method and iterate over the object it returns. The result is an iterable that returns a stream of (event, element) tuples.

```
for event, elem in iterparse(source):
    ... elem is complete; process it ...
```

```
for event, elem in iterparse(source, events=("start", "end")):
    if event == "start":
        ... elem was just added to the tree ...
    else:
        ... elem is complete; process it ...
```

The **events** option specify what events you want to see (available events in this release are “start”, “end”, “start-ns”, and “end-ns”, where the “ns” events are used to get detailed namespace information). If the option is omitted, only “end” events are returned.

Note: The tree builder and the event generator are not necessarily synchronized; the latter usually lags behind a bit. This means that when you get a “start” event for an element, the builder may already have filled that element with content. You cannot rely on this, though — a “start” event can only be used to inspect the attributes, not the element content. For more details, see this message.

Incremental Parsing

Note that **iterparse** still builds a tree, just like **parse**, but you can safely rearrange or remove parts of the tree while parsing. For example, to parse large files, you can get rid of elements as soon as you’ve processed them:

```
for event, elem in iterparse(source):
    if elem.tag == "record":
        ... process record elements ...
        elem.clear()
```

The above pattern has one drawback; it does not clear the root element, so you will end up with a single element with lots of empty child elements. If your files are huge, rather than just large, this might be a problem. To work around this, you need to get your hands on the root element. The easiest way to do this is to enable **start** events, and save a reference to the first element in a variable:

```
# get an iterable
context = iterparse(source, events=("start", "end"))

# turn it into an iterator
context = iter(context)

# get the root element
event, root = context.next()

for event, elem in context:
    if event == "end" and elem.tag == "record":
        ... process record elements ...
        root.clear()
```

(future releases will make it easier to access the root element from within the loop)

Namespace Events

The namespace events contain information about namespace scopes in the source document. This can be used to keep track of active namespace prefixes, which are otherwise discarded by the parser. Here's how you can emulate the **namespaces** attribute in the FancyTreeBuilder class:

```
events = ("end", "start-ns", "end-ns")
namespaces = []
for event, elem in iterparse(source, events=events):
    if event == "start-ns":
        namespaces.insert(0, elem)
    elif event == "end-ns":
        namespaces.pop(0)
    else:
        ...
```

The **namespaces** variable in this example will contain a stack of (prefix, uri) tuples.

(Note how **iterparse** lets you replace instance variables with local variables. The code is not only easier to write, it is also a lot more efficient.)

For better performance, you can append and remove items at the right end of the list instead, and loop backwards when looking for prefix mappings.

Incremental Decoding

Here's a rather efficient and almost complete XML-RPC decoder (just add fault handling). This implementation is 3 to 4 times faster than the 170-line version I wrote for Python's **xmlrpclib** library…

```
from cElementTree import iterparse
from cStringIO import StringIO
import datetime, time

def make_datetime(text):
    return datetime.datetime(
        *time.strptime(text, "%Y%m%dT%H:%M:%S")[:6]
    )

unmarshallers = {
    "int": lambda x: int(x.text), "i4": lambda x: int(x.text),
    "boolean": lambda x: x.text == "1",
    "string": lambda x: x.text or "",
    "double": lambda x: float(x.text),
    "dateTime.iso8601": lambda x: make_datetime(x.text),
    "array": lambda x: [v.text for v in x],
    "struct": lambda x: dict((k.text or "", v.text) for k, v in x),
    "base64": lambda x: decodestring(x.text or ""),
    "value": lambda x: x[0].text,
}

def loads(data):
    params = method = None
    for action, elem in iterparse(StringIO(data)):
        unmarshal = unmarshallers.get(elem.tag)
        if unmarshal:
            data = unmarshal(elem)
            elem.clear()
            elem.text = data
        elif elem.tag == "methodCall":
            method = elem.text
        elif elem.tag == "params":
            params = tuple(v.text for v in elem)
    return params, method
```

Note that code uses the **text** attribute to temporarily hold unmarshalled Python objects. All standard ElementTree implementations support this, but some alternative implementations may not support non-text attribute values.

The same approach can be used to read Apple's **plist** format:

```
try:
    import cElementTree as ET
except ImportError:
    import elementtree.ElementTree as ET
import base64, datetime, re

unmarshallers = {

    # collections
    "array": lambda x: [v.text for v in x],
    "dict": lambda x:
        dict((x[i].text, x[i+1].text) for i in range(0, len(x), 2)),
    "key": lambda x: x.text or "",

    # simple types
    "string": lambda x: x.text or "",
    "data": lambda x: base64.decodestring(x.text or ""),
    "date": lambda x: datetime.datetime(*map(int, re.findall("\d+", x.text))),
    "true": lambda x: True,
    "false": lambda x: False,
    "real": lambda x: float(x.text),
    "integer": lambda x: int(x.text),

}

def load(file):
    parser = iterparse(file)
    for action, elem in parser:
        unmarshal = unmarshallers.get(elem.tag)
        if unmarshal:
            data = unmarshal(elem)
            elem.clear()
            elem.text = data
        elif elem.tag != "plist":
            raise IOError("unknown plist type: %r" % elem.tag)
    return parser.root[0].text
```

To round this off, here’s the obligatory RSS-reader-in-less-than-eight-lines example:

```
from urllib import urlopen
from cElementTree import iterparse

for event, elem in iterparse(urlopen("http://online.effbot.org/rss.xml")):
    if elem.tag == "item":
        print elem.findtext("link"), "-", elem.findtext("title")
        elem.clear() # won't need the children any more
```