pyRXP Documentation

Release 1.16

Robin Becker <robin@reportlab.com>

Jul 17, 2018

Contents

Contents:

CHAPTER 1

1. Introduction

1.1 1.1 Who is this document aimed at?

This document is aimed at anyone who wants to know how to use the pyRXP parser extension from Python. It's assumed that you know how to use the Python programming language and understand its terminology. We make no attempt to teach XML in this document, so you should already know the basics (what a DTD is, some of the syntax etc.)

1.2 1.2 What is PyRXP?

PyRXP is a Python language wrapper around the excellent RXP parser. RXP is a validating namespace-aware XML parser written in C. It was released by ReportLab in 2003, at a time when the available XML parsing tools in Python were, frankly, a mess. At the time it was the fastest XML-parsing framework available to Python programmers, with the benefit of validation. Please bear in mind that much of the documentation was written at that time.

RXP was written by Richard Tobin at the Language Technology Group, Human Communication Research Centre, University of Edinburgh. PyRXP was written by Robin Becker at ReportLab.

ReportLab uses pyRXP to parse its own Report Markup Language formatting product, and for all inbound XML within our document generation solutions. Having a validating XML parser is a huge benefit, because it stops a large proportion of bad input from other systems early on, and forces producers to get things right, rather than leaning on us to write ad-hoc cleanups for other peoples' poor data.

The code is extremely mature and stable.

In recent years, libxml2 and lxml have become popular and offer much of the same functionality, under less restrictive licenses; these may also be a valid choice for your project.

This documentation describes pyRXP-1.16 being used with RXP 1.4.0, as well as ReportLab's emerging XML toolkit which uses it.

1.3 1.3 License terms

Edinburgh University have released RXP under the GPL. This is generally fine for in-house or open-source use. But if you want to use it in a closed-source commercial product, you may need to negotiate a separate license with them. By contrast, most Python software uses a less restrictive license; Python has its own license, and ReportLab uses the FreeBSD license for our PDF Toolkit, which means you CAN use it in commercial products.

We licensed RXP for our commercial products, but are releasing pyRXP under the GPL. (We did try to persuade Edinburgh to release under a Python style license, but they declined; otherwise pyRXP might have become the Python standard.)

If you want to use pyRXP for a commercial product, you need to purchase a license. We are authorised resellers for RXP and can sell you a commercial license to use it in your own products. PyRXP is ideal for embedded use being lightweight, fast and pythonic.

1.4 1.4 Why another XML toolkit?

This grew out of real world needs which others in the Python community may share. ReportLab make tools which read in some kind of data and make PDF reports. One common input format these days is XML. It's very convenient to express the interface to a system as an XML file. Some other system might send us some XML with tags like <invoice> and <customer>, and we turn these into nice looking invoices.

Also, we have a commercial product called Report Markup Language - we sell a converter to turn RML files into PDF. This has to parse XML, and do it fast and accurately.

Typically we want to get this XML into memory as fast as possible. And, if the performance penalties are not too great, we'd like the option to validate it as well. Validation is useful because we can stop bad data at the point of input; if someone else sends our system an XML 'invoice packet' which is not valid according to the agreed DTD, and gets a validation error, they will know what's going on. This is a lot more helpful than getting a strange and unrelated-sounding error during the formatting stage.

We tried to use all the parsers we could find. We found that almost all of them were constructing large object models in Python code, which took a long time and a lot of memory. Even the fastest C-based parser, expat (which was not yet a standard part of Python at the time) calls back into Python code on every start and end tag, which defeats most of the benefit. Aaron Watters of ReportLab sat down for a couple of days in 2000 and produced his own parser, rparsexml, which uses string.find and got pretty much the same speed as pyexpat. We evolved our own representation of a tree in memory; which became the cornerstone of our approach; and when we found RXP we found it easy to make a wrapper around it to produce the "tuple tree".

We have now reached the point in our internal bag-of-tools where XML parsing is a standard component, running entirely at C-like speeds, and we don't even think much about it any more. Which means we must be doing something right and it's time to share it :-)

1.5 1.5 Design Goals

This is part of an XML framework which we will polish up and release over time as we find the time to document it. The general components are:

- A standard in-memory representation of an XML document (the *tuple tree* below)
- Various parsers which can produce this - principally pyRXP, but expat wrapping is possible
- A 'lazy wrapper' around this which gives a very friendly Pythonic interface for navigating the tree
- A lightweight transformation tool which does a lot of what XSLT can do, but again with Pythonic syntax

In general we want to get the whole structure of an XML document into memory as soon as possible. Having done so, we're going to traverse through it and move the data into our own object model anyway; so we don't really care what kind of "node objects" we're dealing with and whether they are DOM-compliant. Our goals for the whole framework are:

- Fast - XML parsing should not be an overhead for a program
- Validate when needed, with little or no performance penalty
- Construct a complete tree in memory which is easy and natural to access
- An easy lightweight wrapping system with some of the abilities of XSLT without the complexity

Note that pyRXP is just the main parsing component and not the framework itself.

1.6 1.6 Design non-goals

It's often much more helpful to spell out what a system or component will NOT do. Most of all we are NOT trying to produce a standards-compliant parser.

- Not a SAX parser
- Not a DOM parser
- Does not capture full XML structure

Why not? Aren't standards good?

It's great that Python has support for SAX and DOM, but these are basically Java (or at least cross-platform) APIs. If you're doing Python, it's possible to make things simpler, and we've tried. Let's imagine you have some XML containing an *invoice* tag, that this in turn contains *lineItems* tags, and each of these has some text content and an *amount* attribute. Wouldn't it be nice if you could write some Python code this simple?

```
invoice = pyRXP.Parser().parse(myInvoiceText)
for lineItem in invoice:
    print invoice.amount
```

Likewise, if a node is known to contain text, it would be really handy to just treat it as a string. We have a preprocessor we use to insert data into HTML and RML files which lets us put Python expressions in curly braces, and we often do things like

```
<html><head><title>My web page</title></head>
<body>
<h1>Statement for {{xml.customer.DisplayName}}</h1>
<!-- etc etc -->
</body>
</html>
<h1></h1>
```

Try to write the equivalent in Java and you'll have loads of method calls to getFirstElement(), getNextElement() and so on. Python has beautifully compact and readable syntax, and we'd rather use it. So we're not bothering with SAX and DOM support ourselves. (Although if other people want to contribute full DOM and SAX wrappers for pyRXP, we'll accept the patches).

1.7 1.7 How fast is it?

The examples file includes a crude benchmarking script. It measures speed and memory allocation of a number of different parsers and frameworks. This is documented later on. Suffice to say that we can parse hamlet in 0.15 seconds with full validation on a P500 laptop. Doing the same with the *minidom* in the Python distribution takes 33 times as long and allocates 8 times as much memory, and does not validate. It also appears to have a significant edge on Microsoft's XML parser and on FourThought's cDomlette. Using pyRXP means that XML parsing will typically take a tiny amount of time compared to whatever your Python program will do with the data later.

1.8 1.8 The Tuple Tree structure

Most 'tree parsers' such as DOM create 'node objects' of some sort. The DOM gives one consensus of what such an object should look like. The problem is that "objects" means "class instances in Python", and the moment you start to use such beasts, you move away from fast C code to slower interpreted code. Furthermore, the nodes tend to have magic attribute names like "parent" or "children", which one day will collide with structural names.

So, we defined the simplest structure we could which captured the structure of an XML document. Each tag is represented as a tuple of

```
(tagName, dict_of_attributes, list_of_children, spare)
```

The dict_of_attributes can be None (meaning no attributes) or a dictionary mapping attribute names to values. The list_of_children may either be None (meaning a singleton tag) or a list with elements that are 4-tuples or plain strings.

A great advantage of this representation - which only uses built-in types in Python - is that you can marshal it (and then zip or encrypt the results) with one line of Python code. Another is that one can write fast C code to do things with the structure. And it does not require any classes installed on the client machine, which is very useful when moving xml-derived data around a network.

This does not capture the full structure of XML. We make decisions before parsing about whether to expand entities and CDATA nodes, and the parser deals with it; after parsing we have most of the XML file's content, but we can't get back to the original in 100% of cases. For example following two representations will both (with default settings) return the string "Smith & Jones", and you can't tell from the tuple tree which one was in the file:

```
<provider>Smith &amp; Jones<provider>
```

Alternatively one can use

```
<provider><[CDATA[Smith & Jones]]>]<![CDATA[]><provider>
```

So if you want a tool to edit and rewrite XML files with perfect fidelity, our model is not rich enough. However, note that RXP itself DOES provide all the hooks and could be the basis for such a parser.

1.9 1.9 Can I get involved?

Sure! Join us on the Reportlab-users mailing list (*http://two.pairlist.net/mailman/listinfo/reportlab-users*), and feel free to contribute patches. The final section of this manual has a brief "wish list".

Because the Reportlab Toolkit is used in many mission critical applications and because tiny changes in parsers can have unintended consequences, we will keep checkin rights on sourceforge to a trusted few developers; but we will do our best to consider and process patches.

CHAPTER 2

2. Installation and Setup

We make available pre-built Windows binaries. On other platforms you can build it from source using distutils. pyRXP is a single extension module with no other dependencies outside Python itself.

2.1 2.1 Installing from PyPI

The easiest way to install pyRXP is by using the package on PyPI:

```
pip install pyRXP
```

2.2 2.2 Source Code installation

If you'd rather install from source code (available under the GPL), you can find it as a Mercurial repository on BitBucket:

```
hg clone https://bitbucket.org/rptlab/pyrxp
cd pyrxp
python setup.py install
```

2.3 2.2.1 Post installation tests

Whichever method you used to get pyRXP installed, you should run the short test suite to make sure there haven't been any problems.

Cd to the `test` directory and run the file `testRXPbasic.py`.

Running the test program should result in a message like this:

```
> python testRXPbasic.py
........................................
............
52 tests, no failures!
```

These are basic health checks, which are the minimum required to make sure that nothing drastic is wrong. This is the very least that you should do - you should not skip this step!

If you want to be more thorough, there is a much more comprehensive test suite which tests XML compliance. This is run by a file called test_xmltestsuite.py, also in the test directory. This depends on a set of more than 300 tests written by James Clark which you can download in the form of a zip file from

```
http://www.reportlab.com/ftp/xmltest.zip
```

or

```
ftp://ftp.jclark.com/pub/xml/xmltest.zip
```

You can simply drop this in the test directory and run the test_xmltestsuite file which will automatically unpack and use it.

2.4 2.3 Windows binary - pyRXP.pyd

ReportLab's FTP server has win32-dlls and amd64-dlls directories, both of which are sub-divided into Python versions, where you'll find the suitable pyd file. So, assuming you use Python 2.7 on a 64-bit Windows machine, the file you need to download is:

```
http://www.reportlab.com/ftp/amd64-dlls/2.7/pyRXP.pyd
```

Download the pyRXP DLL from the ReportLab FTP site. Save the pyRXP.pyd in the DLLs directory under your Python installation (eg this is the C:\Python27\DLLs directory for a standard Windows installation of Python 2.7).

2.5 2.4 Examples

If you installed pyRXP from source you'll find an `examples` directory, which includes a couple of substantial XML files with DTDs, a wrapper module called *xmlutils* which provides easy access to the tuple tree, and a simple benchmarking script, both documented in section 4.

Note for Windows users:

If you only installed the DLL, you can download the examples from

```
http://www.reportlab.com/ftp/pyrxp_examples.zip
```

CHAPTER 3

3. Using pyRXP

3.1 3.1. Simple use without validation

3.1.1 3.1.1 The Parse method and callable instances of the parser

Firstly you have to import the pyRXP module (using Python's import statement). While we are here, pyRXP has a couple of attributes that are worth knowing about: `version` gives you a string with the version number of the pyRXP module itself, and `RXPVersion` gives you string with the version information for the rxp library embedded in the module.

```
>>> import pyRXPU
>>> pyRXPU.version
'1.16'
>>> pyRXPU.RXPVersion
'RXP 1.5.0 Copyright Richard Tobin, LTG, HCRC, University of Edinburgh'
```

Once you have imported pyRXP, you can instantiate a parser instance using the Parser class.

```
>>>rxp=pyRXPU.Parser()
```

To parse some XML, you use the `parse` method, passing a string as the first argument and receiving the parsed Tuple Tree as a result:

```
>>> rxp=pyRXPU.Parser()
>>> rxp.parse('<a>some text</a>')
(u'a', None, [u'some text'], None)
```

As a shortcut, you can call the instance directly:

```
>>> rxp=pyRXPU.Parser()
>>> rxp('<a>some text</a>')
(u'a', None, [u'some text'], None)
```

The current version of PyRXP only contains pyRXPU, which is the 16-bit Unicode aware version of pyRXP, and all returned strings are Unicode strings.

__Note__: Throughout this documentation, we'll use the explicit call syntax for clarity.

3.1.2 3.1.2 Basic usage

We'll start with some very simple examples and leave validation for later.

```
>>> rxp.parse('<tag>content</tag>')
(u'tag', None, [u'content'], None)
```

Each element ("tag") in the XML is represented as a tuple of 4 elements:

- 'tag': the tag name (aka element name).
- None: a dictionary of the tag's attributes (None here since it doesn't have any).
- ['content']: a list of the children elements of the tag.
- None: the fourth element is unused by default.

This tree structure is equivalent to the input XML, at least in information content. It is theoretically possible to recreate the original XML from this tree since no information has been lost.

A tuple tree for more complex XML snippets will contain more of these tuples, but they will all use the same structure as this one.

```
>>> rxp.parse('<tag1><tag2>content</tag2></tag1>')
(u'tag1', None, [(u'tag2', None, [u'content'], None)], None)
```

This may be easier to understand if we lay it out differently:

```
>>> rxp.parse('<tag1><tag2>content</tag2></tag1>')
(u'tag1',
 None,
     [(u'tag2',
       None,
       [u'content'],
       None)
     ],
None)
```

Tag1 is the name of the outer tag, which has no attributes. Its contents is a list. This contents contains Tag2, which has its own attribute dictionary (which is also empty since it has no attributes) and its content, which is the string 'content'. It has the closing null element, then the list for Tag2 is closed, Tag1 has its own final null element and it too is closed.

The XML that is passed to the parser must be balanced. Any opening and closing tags must match. They wouldn't be valid XML otherwise.

3.1.3 3.1.3 Empty tags and the ExpandEmpty flag

Look at the following three examples. The first one is a fairly ordinary tag with contents. The second and third can both be considered as empty tags - one is a tag with no content between its opening and closing tag, and the other is the singleton form which by definition has no content.

```
>>> rxp.parse('<tag>my contents</tag>')
(u'tag', None, [u'my contents'], None)

>>> rxp.parse('<tag></tag>')
(u'tag', None, [], None)

>>> rxp.parse('<tag/>')
(u'tag', None, None, None)
```

Notice how the contents list is handled differently for the last two examples. This is how we can tell the difference between an empty tag and its singleton version. If the content list is empty then the tag doesn't have any content, but if the list is None, then it can't have any content since it's the singleton form which can't have any by definition.

Another example:

```
>>>rxp.parse('<outerTag><innerTag>bb</innerTag>aaa<singleTag/></outerTag>')
(u'outerTag', None, [(u'innerTag', None, [u'bb'], None), u'aaa', (u'singleTag',
None, None, None)], None)
```

Again, this is more understandable if we show it like this:

```
(u'outerTag',
 None,
     [(u'innerTag',
       None,
       [u'bb'],
       None),
          u'aaa',
              (u'singleTag',
               None,
               None,
               None)
      ],
 None)
```

In this example, the tuple contains the outerTag (with no attribute dictionary), whose list of contents are the innerTag, which contains the string 'bb' as its contents, and the singleton singleTag whose contents list is replaced by a null.

The way that these empty tags are handled can be changed using the `ExpandEmpty` flag. If `ExpandEmpty` is set to 0, these singleton forms come out as None, as we have seen in the examples above. However, if you set it to 1, the empty tags are returned as standard tags of their sort.

This may be useful if you will need to alter the tuple tree at some future point in your processing. Lists and dictionaries are mutable, but None isn't and therefore can't be changed.

Some examples. This is what happens if we accept the default behaviour:

```
>>> rxp.parse('<a>some text</a>')
(u'a', None, [u'some text'], None)
```

Explicitly setting ExpandEmpty to 1 gives us these:

```
>>> rxp.parse('<a>some text</a>', ExpandEmpty=1)
(u'a', {}, [u'some text'], None)
```

Notice how the None from the first example is being returned as an empty dictionary in the second version. `ExpandEmpty` makes the sure that the attribute list is always a dictionary. It also makes sure that a self-closed tag returns an empty list.

A very simple example of the singleton or ‘self-closing’ version of a tag.

```
>>> rxp.parse('<b/>', ExpandEmpty=0)
(u'b', None, None, None)
```

```
>>> rxp.parse('<b/>', ExpandEmpty=1)
(u'b', {}, [], None)
```

Again, notice how the Nones have been expanded.

Some more examples show how these work with slightly more complex XML which uses nested tags:

```
>>> rxp.parse('<a>some text<b>Hello</b></a>', ExpandEmpty=0)
(u'a', None, [u'some text', (u'b', None, [u'Hello'], None)], None)

>>> rxp.parse('<a>some text<b>Hello</b></a>', ExpandEmpty=1)
(u'a', {}, [u'some text', (u'b', {}, [u'Hello'], None)], None)
```

```
>>> rxp.parse('<a>some text<b></b></a>', ExpandEmpty=0)
(u'a', None, [u'some text', (u'b', None, [], None)], None)

>>> rxp.parse('<a>some text<b></b></a>', ExpandEmpty=1)
(u'a', {}, [u'some text', (u'b', {}, [], None)], None)
```

```
>>> rxp.parse('<a>some text<b/></a>', ExpandEmpty=0)
(u'a', None, [u'some text', (u'b', None, None, None)], None)

>>> rxp.parse('<a>some text<b/></a>', ExpandEmpty=1)
(u'a', {}, [u'some text', (u'b', {}, [], None)], None)
```

3.1.4 3.1.4 Processing instructions

Both the comment and processing instruction tag names are special - you can check for them relatively easily. This section processing instruction and the next one covers handling comments.

A processing instruction allows developers to place information specific to an outside application within the document. You can handle it using the `ReturnProcessingInstruction` attribute.

```
>>> rxp.parse(<a><?works document="hello.doc"?></a>')
(u'a', None, [], None)
>>> #vanishes - like a comment
>>> rxp.parse('<a><?works document="hello.doc"?></a>', ReturnProcessingInstructions=1)
(u'a', None, [(u'<?', {u'name': u'works'}, [u'document="hello.doc"'], None)], None)
>>>
```

pyRXP uses a module pseudo-constant called `piTagName` (it’s not an instance attribute) to check for processing instructions:

```
>>> pyRXP.piTagName
u'<?'
```

You can test against `piTagName` - but don’t try and change it. See the section on trying to change `commentTagName` for an example of what would happen.

```
>>> rxp.parse('<a><?works document="hello.doc"?></a>',
... ReturnProcessingInstructions=1)[2][0][0] is pyRXP.piTagName
True
```

This is a simple test and doesn't even have to process the characters. It allows you to process these lists looking for processing instructions (or comments if you are testing against commentTagName as shown in the next section)

3.1.5 3.1.5 Handling comments and the srcName attribute

NB The way ReturnComments works has changed between versions.

By default, PyRXP ignores comments and their contents are lost (this behaviour can be changed - see the section of Flags later for details).

```
>>> rxp.parse('<tag><!-- this is a comment about the tag --></tag>')
(u'tag', None, [], None)

>>> rxp.parse('<!-- this is a comment -->')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Document ends too soon
 in unnamed entity at line 1 char 27 of [unknown]
Document ends too soon
Parse Failed!
```

This causes an error, since the parser sees an empty string which isn't valid XML.

It is possible to set pyRXP to not swallow comments using the ReturnComments attribute.

```
>>> rxp.parse('<tag><!-- this is a comment about the tag --></tag>', ReturnComments=1)
(u'tag', None, [(u'<!--', None, [u' this is a comment about the tag '], None)], None)
```

Using ReturnComments, the comment are returned in the same way as an ordinary tag, except that the tag has a special name. This special name is defined in the module pseudo-constant commentTagName (again, not an instance attribute):

```
>>> rxp.commentTagName
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: commentTagName

>>> pyRXPU.commentTagName
u'<!--'
```

Please note that changing commentTagName won't work: what would be changed is simply the Python representation, while the underlying C object would remain untouched:

```
>>> import pyRXPU
>>> p=pyRXPU.Parser()
>>> pyRXPU.commentTagName = "##" # THIS WON'T WORK!
>>> pyRXPU.commentTagName
'##'
>>> #LOOKS LIKE IT WORKS - BUT SEE BELOW FOR WHY IT DOESN'T
>>> rxp.parse('<a><!-- this is another comment comment --></a>', ReturnComments = 1)
>>> # DOESN'T WORK!
```

(continues on next page)

(continued from previous page)

```
>>> (u'a', None, [(u'<!--', None, [u' this is another comment comment '], None)],␣
↪None)
>>> #SEE?
```

What it is useful for is to check against to see if you have been returned a comment:

```
>>> rxp.parse('<a><!-- comment --></a>', ReturnComments=1)
(u'a', None, [(u'<!--', None, [u' comment '], None)], None)
>>> rxp.parse('<a><!-- comment --></a>', ReturnComments=1)[2][0][0]
u'<!--'
>>> #this returns the comment name tag from the tuple tree...
>>> rxp.parse('<a><!-- comment --></a>', ReturnComments=1)[2][0][0] is pyRXP.
↪commentTagName
1
>>> #they're identical
>>> #it's easy to check if it's a special name
```

Using `ReturnComments` is useful, but there are circumstances where it fails. Comments which are outside the root tag (in the following snippet, that means which are outside the tag '<tag/>', ie the last element in the line) will still be lost:

```
>>> rxp.parse('<tag/><!-- this is a comment about the tag -->', ReturnComments=1)
(u'tag', None, None, None)
```

To get around this, you need to use the `ReturnList` attribute:

```
>>> rxp.parse('<tag/><!-- this is a comment about the tag -->', ReturnComments=1,␣
↪ReturnList=1)
[(u'tag', None, None, None), (u'<!--', None, [u' this is a comment about the tag '],␣
↪None)]
>>>
```

Since we've seen a number of errors in the preceding paragraphs, it might be a good time to mention the `srcName` attribute. The Parser has an attribute called `srcName` which is useful when debugging. This is the name by which pyRXP refers to your code in tracebacks. This can be useful - for example, if you have read the XML in from a file, you can use the `srcName` attribute to show the filename to the user. It doesn't get used for anything other than pyRXP Errors - SyntaxErrors and IOErrors still won't refer to your XML by name.

```
>>> rxp.srcName = 'mycode'
>>> rxp.parse('<a>aaa</a')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Expected > after name in end tag, but got <EOE>
 in unnamed entity at line 1 char 10 of mycode
Expected > after name in end tag, but got <EOE>
Parse Failed!
```

The XML that is passed to the parser must be balanced. Not only must the opening and closing tags match (they wouldn't be valid XML otherwise), but there must also be one tag that encloses all the others. If there are valid fragments that aren't enclosed by another valid tag, they are considered 'multiple elements' and a pyRXP Error is raised.

```
>>> rxp.parse('<a></a><b></b>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Document contains multiple elements
```

(continues on next page)

(continued from previous page)

```
 in unnamed entity at line 1 char 9 of mycode
Document contains multiple elements
Parse Failed!

>>> rxp.parse('<outer><a></a><b></b></outer>')
(u'outer', None, [(u'a', None, [], None), (u'b', None, [], None)], None)
```

3.2 3.2. Validating against a DTD

This section describes the default behaviours when validating against a DTD. Most of these can be changed - see the section on flags later in this document for details on how to do that.

For the following examples, we're going to assume that you have a single directory with the DTD and any test files in it.

```
>>> dtd = open('tinydtd.dtd', 'r').read()

>>> print dtd
<!-- A tiny sample DTD for use with the PyRXP documentation -->
<!-- $Header $-->

<!ELEMENT a (b)>
<!ELEMENT b (#PCDATA)*>
```

This is just to show you how trivial the DTD is for this example. It's about as simple as you can get - two tags, both mandatory. Both a and b must appear in an xml file for it to conform to this DTD, but you can have as many b's as you want, and they can contain any content.

```
>>> fn=open('sample1.xml', 'r').read()

>>> print fn
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE a SYSTEM "tinydtd.dtd">

<a>
<b>This is the contents</b>
</a>
```

This is the simple example file. The first line is the XML declaration, and the *standalone="no"* part says that there should be an external DTD. The second line says where the DTD is, and gives the name of the root element - *a* in this case. If you put this in your XML document, pyRXP will attempt to validate it.

```
>> rxp.parse(fn)
(u'a',
 None,
 [u'\n', (u'b', None, [u'This tag is the contents'], None), '\n'],
 None)
>>>
```

This is a successful parse, and returns a tuple tree in the same way as we have seen where the input was a string.

If you have a reference to a non-existant DTD file in a file (or one that can't be found over a network), then any attempt to parse it will raise a pyRXP error.

```
>>> fn=open('sample2.xml', 'r').read()

>>> print fn
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE a SYSTEM "nonexistent.dtd">

<a>
<b>This is the contents</b>
</a>

>>> rxp.parse(fn)
C:\tmp\pyRXP_tests\nonexistent.dtd: No such file or directory
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.Error: Error: Couldn't open dtd entity file:///C:/tmp/pyRXP_tests/nonexistent.
↪dtd
 in unnamed entity at line 2 char 38 of [unknown]
```

This is a different kind of error to one where no DTD is specified:

```
>>> fn=open('sample4.xml', 'r').read()

>>> print fn
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<a>
<b>This is the contents</b>
</a>

>>> rxp.parse(fn,NoNoDTDWarning=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Document has no DTD, validating abandoned
 in unnamed entity at line 3 char 2 of [unknown]
Document has no DTD, validating abandoned
Parse Failed!
```

If you have errors in your XML and it does not validate against the DTD, you will get a different kind of pyRXPError.

```
>>> fn=open('sample3.xml', 'r').read()

>>> print fn
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<!DOCTYPE a SYSTEM "tinydtd.dtd">

<x>
<b>This is the contents</b>
</x>

>>> rxp.parse(fn)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.Error: Error: Start tag for undeclared element x
 in unnamed entity at line 4 char 3 of [unknown]
>>>
```

Whether PyRXP validates against a DTD, together with a number of other behaviours is decided by how the various flags are set.

By default, `ErrorOnValidityErrors` is set to 1, as is `NoNoDTDWarning`. If you want the XML you are parsing to actually validate against your DTD, you should have both of these set to 1 (which is the default value), otherwise instead of raising a pyRXP error saying the XML that doesn't conform to the DTD (which may or may not exist) this will be silently ignored. You should also have `Validate` set to 1, otherwise validation won't even be attempted.

Note that the first examples in this chapter - the ones without a DTD - only worked because we had carefully chosen what seem like the sensible defaults. It is set to validate, but not to complain if the DTD is missing. So when you feed it something without a DTD declaration, it notices the DTD is missing but continues in non-validating mode. There are numerous flags set out below which affect the behaviour.

3.3 3.3 Interface Summary

The python module exports the following:

`error`

a python exception

`version`

the string version of the module

`RXPVersion`

the version string of the rxp library embedded in the module

`parser_flags`

a dictionary of parser flags - the values are the defaults for parsers

`Parser(**kwargs)`

Create a parser

`piTagName`

special tagname used for processing instructions

`commentTagName`

special tagname used for comments

`recordLocation`

a special do nothing constant that can be used as the 'fourth' argument and causes location information to be recorded in the fourth position of each node.

3.4 3.4 Parser Object Attributes and Methods

`parse(src, **kwargs)`

We have already seen that this is the main interface to the parser. It returns ReportLab's standard tuple tree representation of the xml source. The string *src* contains the xml.

The keyword arguments can modify the instance attributes for this call only. For example, we can do

```
>>> rxp.parse('<a>some text</a>', ReturnList=1, ReturnComments=1)
```

instead of

```
>>> rxp.ReturnList=1
>>> rxp.ReturnComments=1
>>> rxp.parse('<a>some text</a>')
```

Any other parses using rxp will be unaffected by the values of `ReturnList` and `ReturnComments` in the first example, whereas all parses using p will have `ReturnList` and `ReturnComments` set to 1 after the second.

`srcName`

A name used to refer to the source text in error and warning messages. It is initially set as '<unknown>'. If you know that the data came from "spam.xml" and you want error messages to say so, you can set this to the filename.

`warnCB`

Warning callback. Should either be None, 0, or a callable object (e.g. a function) with a single argument which will receive warning messages. If None is used then warnings are thrown away. If the default 0 value is used then warnings are written to the internal error message buffer and will only be seen if an error occurs.

`eoCB`

Entity-opening callback. The argument should be None or a callable method with a single argument. This method will be called when external entities are opened. The method should return a (possibly modified) URI. So, you could intercept a declaration referring to *http://some.slow.box/somefile.dtd* and point at at the local copy you know you have handy, or implement a DTD-caching scheme.

`fourth`

This argument should be None (default) or a callable method with no arguments. If callable, will be called to get or generate the 4th item of every 4-item tuple or list in the returned tree. May also be the special value `pyRXP.recordLocation` to cause the 4th item to be set to a location information tuple ((startname,startline,startchar),(endname,endline,endchar)).

3.5 3.5 List of Flags

Flag attributes corresponding to the rxp flags; the values are the module standard defaults. `pyRXP.parser_flags` returns these as a dictionary if you need to refer to these inline.

Flag (1=on, 0=off)	Default
AllowMultipleElements	0
AllowUndeclaredNSAttributes	0
CaseInsensitive	0
ErrorOnBadCharacterEntities	1
ErrorOnUndefinedAttributes	0
ErrorOnUndefinedElements	0
ErrorOnUndefinedEntities	1
ErrorOnUnquotedAttributeValues	1
ErrorOnValidityErrors	1
ExpandCharacterEntities	1
ExpandEmpty	0
ExpandGeneralEntities	1
IgnoreEntities	0
IgnorePlacementErrors	0
MaintainElementStack	1
MakeMutableTree	0

Continued on next page

Table 1 – continued from previous page

MergePCData	1
NoNoDTDWarning	1
NormaliseAttributeValues	1
ProcessDTD	0
RelaxedAny	0
ReturnComments	0
ReturnProcessingInstructions	0
ReturnDefaultedAttributes	1
ReturnList	0
ReturnNamespaceAttributes	0
ReturnUTF8 (pyRXPU)	0
SimpleErrorFormat	0
TrustSDD	1
Validate	1
WarnOnRedefinitions	0
XMLExternalIDs	1
XMLLessThan	0
XMLMiscWFErrors	1
XMLNamespaces	0
XMLPredefinedEntities	1
XMLSpace	0
XMLStrictWFErrors	1
XMLSyntax	1

3.6 3.6 Flag explanations and examples

With so many flags, there is a lot of scope for interaction between them. These interactions are not documented yet, but you should be aware that they exist.

3.6.1 AllowMultipleElements

Default: 0

Description:

A piece of XML that does not have a single root-tag enclosing all the other tags is described as having multiple elements. By default, this will raise a pyRXP error. Turning this flag on will ignore this and not raise those errors.

Example:

```
>>> rxp.AllowMultipleElements = 0
>>> rxp.parse('<a></a><b></b>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Document contains multiple elements
 in unnamed entity at line 1 char 9 of [unknown]
Document contains multiple elements

>>> rxp.AllowMultipleElements = 1
>>> rxp.parse('<a></a><b></b>')
('a', None, [], None)
```

3.6.2 AllowUndeclaredNSAttributes

Default: 0

Description:

[to be added]

Example:

[to be added]

3.6.3 CaseInsensitive

Default: 0

Description:

This flag controls whether the parse is case sensitive or not.

Example:

```
>>> rxp.CaseInsensitive=1
>>> rxp.parse('<a></A>')
('A', None, [], None)

>>> rxp.CaseInsensitive=0
>>> rxp.parse('<a></A>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Mismatched end tag: expected </a>, got </A>
 in unnamed entity at line 1 char 7 of [unknown]
Mismatched end tag: expected </a>, got </A>
```

3.6.4 ErrorOnBadCharacterEntities

Default: 1

Description:

If this is set, character entities which expand to illegal values are an error, otherwise they are ignored with a warning.

Example:

```
>>> rxp.ErrorOnBadCharacterEntities=0
>>> rxp.parse('<a>&#999;</a>')
(u'a', None, [u''], None)

>>> rxp.ErrorOnBadCharacterEntities=1
>>> rxp.parse('<a>&#999;</a>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: 0x3e7 is not a valid 8-bit XML character
 in unnamed entity at line 1 char 10 of [unknown]
0x3e7 is not a valid 8-bit XML character
```

3.6.5 ErrorOnUndefinedAttributes

Default: 0

Description:

If this is set and there is a DTD, references to undeclared attributes are an error.

See also: *ErrorOnUndefinedElements*

3.6.6 ErrorOnUndefinedElements

Default: 0

Description:

If this is set and there is a DTD, references to undeclared elements are an error.

See also: *ErrorOnUndefinedAttributes*

3.6.7 ErrorOnUndefinedEntities

Default: 1

Description:

If this is set, undefined general entity references are an error, otherwise a warning is given and a fake entity constructed whose value looks the same as the entity reference.

Example:

```
>>> rxp.ErrorOnUndefinedEntities=0
>>> rxp.parse('<a>&dud;</a>')
(u'a', None, [u'&dud;'], None)

>>> rxp.ErrorOnUndefinedEntities=1
>>> rxp.parse('<a>&dud;</a>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Undefined entity dud
 in unnamed entity at line 1 char 9 of [unknown]
Undefined entity dud
```

3.6.8 ErrorOnUnquotedAttributeValues

Default: 1

Description:

[to be added]

3.6.9 ErrorOnValidityErrors

Default: 1

Description:

If this is on, validity errors will be reported as errors rather than warnings. This is useful if your program wants to rely on the validity of its input.

3.6.10 ExpandEmpty

Default: 0

Description:

If false, empty attribute dicts and empty lists of children are changed into the value None in every 4-item tuple or list in the returned tree.

3.6.11 ExpandCharacterEntities

Default: 1

Description:

If this is set, entity references are expanded. If not, the references are treated as text, in which case any text returned that starts with an ampersand must be an entity reference (and provided `MergePCData` is off, all entity references will be returned as separate pieces).

See also: *ExpandGeneralEntities*, *ErrorOnBadCharacterEntities*

Example:

```
>>> rxp.ExpandCharacterEntities=1
>>> rxp.parse('<a>&#109;</a>')
(u'a', None, [u'm'], None)

>>> rxp.ExpandCharacterEntities=0
>>> rxp.parse('<a>&#109;</a>')
(u'a', None, [u'&#109;'], None)
```

3.6.12 ExpandGeneralEntities

Default: 1

Description:

If this is set, entity references are expanded. If not, the references are treated as text, in which case any text returned that starts with an ampersand must be an entity reference (and provided `MergePCData` is off, all entity references will be returned as separate pieces).

See also: *ExpandCharacterEntities*

Example:

```
>>> rxp.ExpandGeneralEntities=0
>>> rxp.parse('<a>&amp;</a>')
(u'a', None, [u'&amp;'], None)

>>> rxp.ExpandGeneralEntities=1
>>> rxp.parse('<a>&amp;</a>')
(u'a', None, [u'&#38;'], None)
```

3.6.13 IgnoreEntities

Default: 0

Description:

If this flag is on, normal entity substitution takes place. If it is off, entities are passed through unaltered.

Example:

```
>>> rxp.IgnoreEntities=0
>>> rxp.parse('<a>&amp;</a>')
(u'a', None, [u'&#38;'], None)

>>> rxp.IgnoreEntities=1
>>> rxp.parse('<a>&amp;</a>')
(u'a', None, [u'&amp;'], None)
```

3.6.14 IgnorePlacementErrors

Default: 0

Description:

[to be added]

3.6.15 MaintainElementStack

Default: 1

Description:

[to be added]

3.6.16 MakeMutableTree

Default: 0

Description:

If false, nodes in the returned tree are 4-item tuples; if true, 4-item lists.

3.6.17 MergePCData

Default: 1

Description:

If this is set, text data will be merged across comments and entity references.

3.6.18 NoNoDTDWarning

Default: 1

Description:

Usually, if `Validate` is set, the parser will produce a warning if the document has no DTD. This flag suppresses the warning (useful if you want to validate if possible, but not complain if not).

3.6.19 NormaliseAttributeValues

Default: 1

Description:

If this is set, attributes are normalised according to the standard. You might want to not normalise if you are writing something like an editor.

3.6.20 ProcessDTD

Default: 0

Description:

If `TrustSDD` is set and a DOCTYPE declaration is present, the internal part is processed and if the document was not declared standalone or if `Validate` is set the external part is processed. Otherwise, whether the DOCTYPE is automatically processed depends on `ProcessDTD`; if `ProcessDTD` is not set the DOCTYPE is not processed.

See also: *TrustSDD*

3.6.21 RelaxedAny

Default: 0

Description:

[to be added]

3.6.22 ReturnComments

Default: 0

Description:

If this is set, comments are returned as nodes with tag name `pyRXPU.commentTagName`, otherwise they are ignored.

Example:

```
>>> rxp.ReturnComments = 1
>>> rxp.parse('<a><!-- this is a comment --></a>')
('a', None, [('<!--', None, [' this is a comment '], None)], None)
>>> rxp.ReturnComments = 0
>>> rxp.parse('<a><!-- this is a comment --></a>')
('a', None, [], None)
```

See also: *ReturnList*

3.6.23 ReturnDefaultedAttributes

Default: 1

Description:

If this is set, the returned attributes will include ones defaulted as a result of ATTLIST declarations, otherwise missing attributes will not be returned.

3.6.24 ReturnList

Default: 0

Description:

If both `ReturnComments` and `ReturnList` are set to 1, the whole list (including any comments) is returned from a parse. If `ReturnList` is set to 0, only the first tuple in the list is returned (ie the actual XML content rather than any comments before it).

Example:

```
>>> rxp.ReturnComments=1
>>> rxp.ReturnList=1
>>> rxp.parse('<!-- comment --><a>Some Text</a><!-- another comment -->')
[(u'<!--', None, [u' comment '], None), (u'a', None, [u'Some Text'], None), ('<!--',
 None, [u' another comment '], None)]
>>> rxp.ReturnList=0
>>> rxp.parse('<!-- comment --><a>Some Text</a><!-- another comment -->')
(u'a', None, [u'Some Text'], None)
>>>
```

See also: *ReturnComments*

3.6.25 ReturnNamespaceAttributes

Default: 0

Description:

[to be added]

3.6.26 ReturnProcessingInstructions

Default: 0

Description:

If this is set, processing instructions are returned as nodes with tagname `pyRXPU.piTagname`, otherwise they are ignored.

3.6.27 SimpleErrorFormat

Default: 0

Description:

This causes the output on errors to get shorter and more compact.

Example:

```
>>> rxp.SimpleErrorFormat=0
>>> rxp.parse('<a>causes an error</b>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: Error: Mismatched end tag: expected </a>, got </b>
 in unnamed entity at line 1 char 22 of [unknown]
Mismatched end tag: expected </a>, got </b>

>>> rxp.SimpleErrorFormat=1
>>> rxp.parse('<a>causes an error</b>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: [unknown]:1:22: Mismatched end tag: expected </a>, got </b>
Mismatched end tag: expected </a>, got </b>
```

3.6.28 TrustSDD

Default: 1

Description:

If `TrustSDD` is set and a DOCTYPE declaration is present, the internal part is processed and if the document was not declared standalone or if `Validate` is set the external part is processed.

See also: *ProcessDTD*

3.6.29 Validate

Default: 1

Description:

If this is on, the parser will validate the document. If it's off, it won't. It is not usually a good idea to set this to 0.

3.6.30 WarnOnRedefinitions

Default: 0

Description:

If this is on, a warning is given for redeclared elements, attributes, entities and notations.

3.6.31 XMLExternalIDs

Default: 1

Description:

[to be added]

3.6.32 XMLLessThan

Default: 0

Description:

[to be added]

3.6.33 XMLMiscWFErrors

Default: 1

Description:

To do with well-formedness errors.

See also: *XMLStrictWFErrors*

3.6.34 XMLNamespaces

Default: 0

Description:

If this is on, the parser processes namespace declarations (see below). Namespace declarations are *not* returned as part of the list of attributes on an element. The namespace value will be prepended to names in the manner suggested by James Clark ie if *xmlns:foo='foovalue'* is active then *foo:name–>{fovalue}name*.

See also: *XMLSpace*

3.6.35 XMLPredefinedEntities

Default: 1

Description:

If this is on, pyRXP recognises the standard preset XML entities &amp; &lt; &gt; &quot; and &apos;) . If this is off, all entities including the standard ones must be declared in the DTD or an error will be raised.

Example:

```
>>> rxp.XMLPredefinedEntities=1
>>> rxp.parse('<a>&amp;</a>')
(u'a', None, [u'&'], None)

>>> rxp.XMLPredefinedEntities=0
>>> rxp.parse('<a>&amp;</a>')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
pyRXP.error: [unknown]:1:9: Undefined entity amp
Undefined entity amp
```

3.6.36 XMLSpace

Default: 0

Description:

If this is on, the parser will keep track of xml:space attributes

See also: *XMLNamespaces*

3.6.37 XMLStrictWFErrors

Default: 1

Description:

If this is set, various well-formedness errors will be reported as errors rather than warnings.

3.6.38 XMLSyntax

Default: 1

Description:

[to be added]

CHAPTER 4

4. The examples and utilities

The zip file of examples contains a couple of validatable documents in xml, the samples used in this manual, and two utility modules: one for benchmarking and one for navigating through tuple trees.

4.1 4.1 Benchmarking

benchmarks.py is a script aiming to compare performance of various parsers. We include it to make our results reproducable. It is not a work of art and if you think you can make it fairer or better, tell us how! Here's an example run.

```
> python benchmarks.py
Interactive benchmark suite for Python XML tree-parsers.
Using sample XML file 444220 bytes long
Parsers available:
    1.  pyRXP
    2.  pyRXP_nonvalidating
    3.  rparsexml
    4.  expat
    5.  minidom
    6.  msxml30
    7.  4dom
    8.  cdomlette
Parser number (or x to exit) > 1
Shall we do memory tests?  i.e. you look at Task Manager? y/n > y
testing pyRXP
Pre-parsing: please input python process memory in kb > 5104
Post-parsing: please input python process memory in kb > 10752
counted 12618 tags, 8157 attributes
pyRXP: init 0.0000, parse 0.0300, traverse 0.0200, mem used 5648kb, mem factor 13.02
```

Instead of the traditional example (hamlet), we took as our example an early version of the Report Markup Language user guide, which is about half a megabyte. Hamlet's XML has almost no attributes; ours contains lots of attributes, many of which will need conversion to numbers one day, and so it provides a more rounded basis for benchmarks

We measure several factors. First there is speed. Obviously this depends on your PC. The script exits after each test so you get a clean process. We measure (a) the time to load the parser and any code it needs into memory (important if doing CGI); (b) time to produce the tree, using whatever the parser natively produces; and (c) time to traverse the tree counting the number of tags and attributes. Note, (c) might be important with a 'very lazy' parser which searched the source text on every request. Also, later on we will be able to look at the difference between traversing a raw tuple tree and some objects with friendlier syntax.

Next is memory. Actually you have to measure that! If anyone can give us the API calls on Windows and other platforms to find out the current process size, we'd be grateful! What we are interested in is how big the structure is in memory. The above shows that the memory allocated is 9.86 times as big as the original XML text. That sounds a lot, but it's actually much less than most DOM parsers.

By contrast, here's the result for the *minidom* parser included in the official Python distro:

```
minidom: init 0.0100, parse 0.2600, traverse 0.0000, mem used 47320kb, mem factor 109.
↪08
```

Even though minidom uses pyexpat (which is in C) to parse the XML, it's several times slower and uses 8 times more memory. And of course it does not validate.

4.2 4.2 xmlutils and the TagWrapper

Finally, we've included a 'tag wrapper' class which makes it easy to navigate around the tuple tree. This is randomly selected from many such modules we have used in various projects; the next task for us is to pick ONE and publish it! Essentially, it uses lazy evaluation to try and figure out which part of the XML you want. If you ask for 'tag.spam', it will check if (a) there is an attribute called spam, or (b) if there is a child tag whose tag name is 'spam'. And you can iterate over child nodes as a sequence. And, the str() method of a tag which just contains text is just the text. The snippets below should make it clear what we are doing.

```
>>> srcText = open('rml_a.xml').read()
>>> tree = pyRXP.Parser().parse(srcText)
>>> import xmlutils
>>> tw = xmlutils.TagWrapper(tree)
>>> tw
TagWrapper<document>
>>> tw.filename
'RML_UserGuide_1_0.pdf'
>>> len(tw.story)  # how many tags in the story?
1566
>>> tw.template.pageSize
'(595, 842)'

>>> for elem in tw.story:
...     if elem.tagName == 'h1':
...         print elem
...
 RML User Guide

Part I - The Basics
Part II - Advanced Features
Part III - Tables
Appendix A - Colors recognized by RML
Appendix B - Glossary of terms and abbreviations
Appendix C - Letters used by the Greek tag
Appendix D - Command reference
```

(continues on next page)

(continued from previous page)

```
Generic Flowables (Story Elements)
Graphical Drawing Operations
Graphical State Change Operations
Style Elements
Page Layout Tags
Special Tags
>>>
```

We are NOT saying this is a particularly good or complete wrapper; but we do intend to standardize on one such wrapper module in the near future, because it makes access to XML information much more ‘pythonic’ and pleasant. It could be used with tuple trees generated by any parser. Please let us know if you have any suggestions on how it should behave.

CHAPTER 5

5. Future Directions

pyRXP is mature and unlikely to change further. At the time of writing in 2013, *libxml2/libxslt* and the very popular *lxml* package which use them, seem to have “picked up the mantle of”cornered the market” for full features XML processing in Python; and the standard library now has *cElementTree* so can do lightweight parsing quickly.

We expect to be using it for several years to come and will attempt to support any bugs found.

ReportLab Thornton House Thornton Road Wimbledon London, UK SW19 4NG