

# *pyrang*: A single-sourcing tool for Python-XML applications



John W. Shipman

2011-02-08 16:30

## Abstract

Describes a script for extracting the names of XML elements and attributes so that Python scripts can use those names in symbolic form.

This publication is available in Web form<sup>1</sup> and also as a PDF document<sup>2</sup>. Please forward any comments to [tcc-doc@nmt.edu](mailto:tcc-doc@nmt.edu).

## Table of Contents

1. Introduction: Why <i>pyrang</i> ? .....	1
2. Operation of <i>pyrang</i> .....	3
3. Setting up Makefile rules for <i>pyrang</i> .....	4
4. <i>pyrang</i> internals .....	5
4.1. Code prologue .....	5
4.2. Module imports .....	5
4.3. Manifest constants .....	5
4.4. <code>main()</code> : The main program .....	6
4.5. <code>fatal()</code> : Write a message and stop .....	7
4.6. <code>processInput()</code> : Read the schema .....	8
4.7. <code>findNames()</code> : Recursive tree walker .....	9
4.8. <code>addName()</code> : Add one name to the name table .....	10
4.9. <code>pythonizeName()</code> : Sanitize an XML name for Python use .....	11
4.10. <code>writeOutput()</code> : Generate the Python file .....	12
4.11. <code>class Args</code> : Command line argument object .....	12
4.12. <code>Args.__init__()</code> : Constructor .....	13
4.13. Code epilogue .....	14

## 1. Introduction: Why *pyrang*?

The author has written several Python-language applications that process XML files using the DOM (Document Object Model), as described in *Python and the XML Document Object Model with 4Suite*<sup>3</sup>.

These Python scripts need to refer to XML element and attribute names in order to process them. Suppose, for example, that in an XML application to represent sports team rosters, a `team` element has `player`

<sup>1</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/pyrang/>

<sup>2</sup> <http://www.nmt.edu/tcc/help/lang/python/examples/pyrang/pyrang.pdf>

<sup>3</sup> <http://www.nmt.edu/tcc/help/pubs/pyxml4/>

child elements. If the variable `teamNode` is an XML DOM Element node, we might use this DOM call to get a list of those child elements:

```
playerList = teamNode.xpath ( "player" )
```

However, the author prefers to avoid using string constants in code, for two reasons:

1. Stylistically, it is a good idea to avoid, as much as possible, the use of constants in code. If you see the constant 20, for example, the obvious question is: why 20?

The professional way to use constants is to define a name for the constant, and then document that definition with an explanation of what the value represents.

The author uses names in all caps for such “manifest constants.” In a C-like language, these would typically be declared using the `#define` construct. Python doesn't have read-only variables, so we just use an ordinary variable.

For XML element names, he prefers a modest Hungarian notation, adding a characteristic suffix of “\_N” for element names (generic identifiers) and “\_A” for attribute names. So, for example, the manifest constant name for the `player` element would be `PLAYER_N`.

Furthermore, “intercapitalized” names such as “nSnakes” should have underbars inserted at each lowercase-to-uppercase transition (e.g., “N\_SNAKES”).

## Warning

XML allows names to contain three characters that are not valid in Python names: hyphen, period, and colon. We'll translate hyphens to underbars, but don't use names with periods and colons with this program.

2. In the life of the vast majority of applications, the design changes over time. In an XML application, it is particularly likely that element and attribute names will be added or changed. When the schema changes, the programmer must find, check, and possibly repair all references to a changed name in the code.

If we define a manifest constant for each element and attribute name, then a simple string search suffices to find all the references.

So we can rewrite the above example as:

```
PLAYER_N = "player" # Declared at the top of the source file
...
playerList = teamNode.xpath ( PLAYER_N )
```

A more subtle problem in maintainability is that now there are two places where XML element and attribute names are defined: in the schema that defines the XML document type, and in Python programs that process documents of that type. When the schema changes, the programmer has to remember to make parallel changes to the Python code. If the two versions get out of synchronization, Bad Things May Happen.

So we see that having these parallel versions violates the principle of *single-sourcing*: that is, there should be a single, reference version of any software entity.

The purpose of *pyrang*, then, is to automate single-sourcing of XML element and attribute names. You must have these software tools installed:

## Python

This script should work with any version of the Python programming language from 2.2 on.

## Relax NG

Relax NG is the author's preferred schema language. For more information, see *Relax NG Compact Syntax (RNC)*<sup>4</sup>.

## trang

The author prefers to write Relax NG schemas using the RNC (Relax NG Compact Syntax) notation. However, there is currently no easy way to access such a schema from Python.

Fortunately, there is an easy short-cut. James Clark's open-source tool *trang* can translate RNC schemas into RNG format, which is an XML document type. There are several good packages making it easy to access XML files.

See the *trang* page<sup>5</sup> for downloads and documentation.

## 4Suite

The *4Suite* package is a package for Python-XML applications. For more information, see *Python and the XML Document Object Model (DOM) with 4Suite*<sup>6</sup>.

## make

The standard Unix *make* utility automates the rebuilding of the Python definitions whenever the schema changes. This utility is driven by the file named *Makefile* in your development directory.

This document has these major sections:

- Section 2, "Operation of *pyrang*" (p. 3): How to run the *pyrang* script.
- Section 3, "Setting up *Makefile* rules for *pyrang*" (p. 4): How to set up *pyrang* in your *Makefile*.
- Section 4, "*pyrang* internals" (p. 5): The actual code for *pyrang*, in lightweight literate programming form.

Files referenced or created in this document:

- *pyrang*: The script for *pyrang*.
- *sysargs.py*: The author's module for processing command line arguments.

## 2. Operation of *pyrang*

---

This script operates on a Relax NG schema in the XML format. It produces a file containing Python declarations corresponding to the element and attribute names from the schema.

If the schema is expressed in Relax NG Compact Syntax (RNC), the *trang* utility can translate it. Typically, RNC files have names ending in *.rnc*, while Relax NG files in the XML format have names ending in *.rng*. To translate, use this command:

```
trang F.rnc F.rng
```

where *F* is the unqualified name of the file. For example, if your RNC schema is named *roster.rnc*, translate it using:

```
trang roster.rnc roster.rng
```

Once you have the schema in *.rng* format, here is the general form of the command that executes *pyrang*:

---

<sup>4</sup> <http://www.nmt.edu/tcc/help/pubs/rnc/>

<sup>5</sup> <http://www.thaiopensource.com/relaxng/trang.html>

<sup>6</sup> <http://www.nmt.edu/tcc/help/pubs/pyxml4/>

```
pyrang [-p prefix] [-o outFile] F.rng
```

where:

**-p *prefix***

If you supply this optional argument, the given *prefix* will be appended to the front of every generated name.

For example, suppose you use this command:

```
pyrang -p RNC_ foo.rng
```

The manifest constant generated for an element named `player` would be `RNC_PLAYER_N` instead of the default `PLAYER_N`.

This option is helpful if your application works with multiple schemas that may have some overlap in their element or attribute names.

***F.rng***

The name of the input schema file is required; it will generally end in `.rng`.

**-o *outFile***

If this argument is provided, output is written to that file. Any existing file is deleted (assuming you have write permission for that file).

If omitted, output is written to the standard output stream.

Here is an example. This command would read schema file `roster.rng` and write the corresponding Python definitions to file `rnc_roster.py`.

```
pyrang -o rnc_roster.py roster.rng
```

You can then include the generated manifest constants in your program using this Python `import`:

```
from rnc_roster import *
```

### 3. Setting up Makefile rules for *pyrang*

The Unix *make* utility is very handy if your schema is written in RNC format. This rule in your *Makefile* will rebuild the `.rng` file from the `.rnc`, if needed, whenever you type the `make` command:

```
F.rng: F.rnc  
    trang F.rnc F.rng
```

Here's a complete example. Suppose your reference schema is `roster.rnc`, and further suppose your Python script expects the manifest constants to live in a file named `rnc_roster.py`, and each constant starts with the string "RNC\_". These *Makefile* rules would rebuild the manifest constants:

```
roster.rng: roster.rnc  
    trang roster.rnc roster.rng  
  
rnc_roster.py: roster.rng  
    pyrang -p RNC_ -o $@ $<
```

## 4. *pyrang* internals

---

Here, in lightweight literate programming (LLP) form, is the actual code for *pyrang*. For more on LLP, see the author's LLP page<sup>7</sup>.

### 4.1. Code prologue

The actual script starts with the traditional “pound-bang line” to make the script self-executing under Unix. This is followed by a comment that points back to this document.

```
pyrang
#!/usr/bin/env python
#=====
# pyrang:  Writes Python declarations for RNC schema names.
#   For documentation and code, see:
#     http://www.nmt.edu/tcc/help/lang/python/examples/pyrang/
#-----
```

### 4.2. Module imports

First we'll need to import the standard Python `sys` module, which will give us access to the command line arguments and the standard I/O streams.

```
pyrang
#=====
# Imports
#-----

import sys
```

Command line argument processing is done with the `sysargs.py` module; see the author's Python library page<sup>8</sup>.

```
pyrang
from sysargs import *
```

To read the RNG schema file, we need the `Parse` function from the `4Suite` package.

```
pyrang
from Ft.Xml import Parse
```

### 4.3. Manifest constants

Here we define symbolic constants for two element names and an attribute name from the Relax NG schema document type.

```
pyrang
ELEMENT_N = 'element'
ATTRIBUTE_N = 'attribute'
NAME_A = 'name'
```

---

<sup>7</sup> <http://www.nmt.edu/~shipman/soft/litprog/>

<sup>8</sup> <http://infohost.nmt.edu/~shipman/soft/clean/lib.html>

## 4.4. main ( ) : The main program

The purpose of the script is to extract all the element and attribute names from the input schema, and write a Python module containing assignment statements that define a symbolic name for each unique name. For example, if the schema has an element named “player” that has an attribute “first-name”, we’ll want to generate two Python assignments from those names:

```
PLAYER_N = 'player'
FIRST_NAME_A = 'first-name'
```

The obvious data structure for accumulating these names is a Python dictionary; we’ll call it `nameTable`. We’ll use the Python name (e.g., `FIRST_NAME_A`) as the key, and the XML name (e.g., `first-name`) as the value, in each dictionary entry.

The same attribute name may occur in more than one element. In that case, we’ll want to avoid writing duplicate declarations. Because dictionary key values must be unique, we can just throw each name we find into the dictionary, and at the end, there will be no duplications.

Here is the overall program flow:

1. Process the command line arguments, and figure out whether the output is going to a file or to standard output.

We’ll define a small class named `Args` to encapsulate the processing and representation of the command line arguments. See Section 4.11, “class `Args`: Command line argument object” (p. 12).

2. The `4Suite Parse ( )` function reads the RNG schema document and gives it to us as a DOM tree.
3. We create `nameTable` as an empty dictionary, then walk the DOM tree recursively looking for element and attribute names. Each name we find is stored into `nameTable`.
4. Finally, each key-value pair in `nameTable` is written to the output as a Python assignment statement.

The code follows. First, we process the command line arguments; see Section 4.11, “class `Args`: Command line argument object” (p. 12).

pyrang

```
def main():
    """Main program
    """

    #-- 1 --
    # [ if sys.argv contains valid command line arguments ->
    #     args := an Args instance representing those arguments
    #     else ->
    #     sys.stderr += error message(s)
    #     stop execution ]
    args = Args()
```

This step takes care of reading the input file and building the `nameTable` dictionary. See Section 4.6, “processInput ( ) : Read the schema” (p. 8).

pyrang

```
#-- 2 --
# [ args is an Args object ->
#     if args.inFileName names a readable file containing a
#     valid RNG schema ->
#     nameTable := a dictionary whose keys are the Python
```

```

#         manifest constant names for the element and
#         attribute names in that schema (using args.prefix
#         as a prefix), and each corresponding value is the
#         XML name
#     else ->
#         sys.stderr += error message(s)
#         stop execution ]
nameTable = processInput ( args )

```

All that remains is to write the output file; see Section 4.10, “writeOutput(): Generate the Python file” (p. 12).

pyrang

```

#-- 3 --
# [ args is an Args object ->
#     if args.outFileName is None ->
#         outFile := sys.stdout
#     else if args.outFileName names a writeable file ->
#         outFile := that file opened new for writing
#     else ->
#         sys.stderr += error message(s)
#         stop execution ]
if args.outFileName is None:
    outFile = sys.stdout
else:
    try:
        outFile = open ( args.outFileName, "w" )
    except IOError, detail:
        fatal ( "Can't open output file '%s' for writing." %
              args.outFileName )

#-- 4 --
# [ (outFile is a writeable file) and
#     (nameTable is a dictionary whose keys are Python names
#     and whose values are XML names ->
#     outFile := Python statements of the form 'n = v'
#             for n in the set of keys of nameTable and each v
#             is the corresponding nameTable value ]
writeOutput ( args, outFile, nameTable )

```

## 4.5. fatal(): Write a message and stop

This function writes a message to the standard error stream and stops execution.

pyrang

```

def fatal ( *L ):
    """Write a message and terminate.

    [ L is a list of strings ->
      sys.stderr += (concatenated elements of L)
      stop execution ]
    """
    print >>sys.stderr, "*** Error: %s" % "".join(L)
    sys.exit(1)

```

## 4.6. processInput ( ) : Read the schema

This function reads the schema, extracts the element and attribute names, and returns the nameTable dictionary with the Python/XML name pairs.

pyrang

```
def processInput ( args ) :
    """Process the input schema.

    [ args is an Args object ->
      if args.inFileName names a readable file containing a
      valid RNG schema ->
        nameTable := a dictionary whose keys are the Python
                    manifest constant names for the element and
                    attribute names in that schema (using args.prefix
                    as a prefix), and each corresponding value is the
                    XML name
      else ->
        sys.stderr += error message(s)
        stop execution ]
    """
```

First we create the name table as an empty dictionary.

pyrang

```
#-- 1 --
# [ nameTable := a new, empty dictionary ]
nameTable = {}
```

Next we attempt to open the input file and hand it to the 4Suite Parse ( ) function to build a DOM tree.

pyrang

```
#-- 2 --
# [ if args.inFileName can be opened for reading ->
#   inFile := that file, so opened
#   else ->
#     sys.stderr += error message
#     stop execution ]
try:
    inFile = open ( args.inFileName )
except IOError, detail:
    fatal ( "Can't open '%s' for reading: %s" %
           (args.inFileName, detail) )

#-- 3 --
# [ if inFile contains a valid XML document ->
#   doc := that document as a DOM tree
#   else ->
#     sys.stderr += error message(s)
#     stop execution ]
try:
    doc = Parse ( inFile )
except Exception, detail:
    fatal ( "Can't parse the schema file: %s" % detail )
```

Walking the tree to find all the `element` and `attribute` elements is quite easy with recursion. The function in Section 4.7, “`findNames()`: Recursive tree walker” (p. 9) is recursive, and finds all the names in a subtree rooted in the node you pass it as its first argument. In a DOM tree, the `doc.documentElement` attribute is the root `Element` node of the schema.

pyrang

```
#-- 4 --
# [ (nameTable is a dictionary) and
#   (args is an Args object) ->
#     nameTable := nameTable with Python/XML name pairs added
#                 from the subtree rooted at doc.documentElement,
#                 with the names prefixed by args.prefix ]
findNames ( doc.documentElement, args, nameTable )
```

Finally, we return the `nameTable` dictionary we've built.

pyrang

```
#-- 5 --
return nameTable
```

## 4.7. `findNames()`: Recursive tree walker

This function finds all element and attribute names in a given subtree of the schema's DOM tree, and adds entries for each name to the `nameTable` dictionary.

pyrang

```
def findNames ( node, args, nameTable ):
    """Find element and attribute names in a subtree.

    [ (node is a DOM Element node) and
      (args is an Args object) and
      (nameTable is a dictionary whose keys are the Python
       names for elements and attributes, and each corresponding
       value is the XML name) ->
        nameTable := nameTable with Python/XML name pairs added
                     from the subtree rooted at node, with the names
                     prefixed by args.prefix ]
    """
```

In an RNG schema, we are looking for elements of these two forms:

```
<element name="N">...
<attribute name="N">...
```

Therefore, all we have to do is check the node's name to see if it is either `element` or `attribute`, and in those cases add the name to `nameTable`. The logic that builds the Python equivalent of the XML name, and prepends `args.prefix`, is in Section 4.8, “`addName()`: Add one name to the name table” (p. 10).

pyrang

```
#-- 1 --
# [ if node.nodeName is ELEMENT_N or ATTRIBUTE_N ->
#   nameTable := nameTable with an entry added with a
#               the Python equivalent of node's "name" attribute as
#               the key, and node's "name" attribute as the value
```

```

# else -> I ]
if node.nodeName == ELEMENT_N:
    eltName = node.getAttributeNS ( None, NAME_A )
    addName ( nameTable, args, eltName, "N" )
elif node.nodeName == ATTRIBUTE_N:
    attrname = node.getAttributeNS ( None, NAME_A )
    addName ( nameTable, args, attrname, "A" )

```

That takes care of extracting names from `node` itself. To recursively add names from its subtree, we iterate over `node`'s children.

pyrang

```

#-- 2 --
# [ nameTable := nameTable with new element and attribute
#     names added from children of node ]
for child in node.childNodes:
    findNames ( child, args, nameTable )

```

## 4.8. addName ( ) : Add one name to the name table

This function takes care of building the Python equivalent of each XML name and making sure there is an entry for that pair in `nameTable`.

pyrang

```

def addName ( nameTable, args, name, suffix ) :
    """Add one element or attribute name to nameTable.

    [ (nameTable is a dictionary) and
      (args is an Args object) and
      (name is an XML element or attribute name as a string) and
      (suffix is a string) ->
        nameTable := nameTable with an entry added with
                    (args.prefix + (name, uppercased and with each "-"
                    replaced by "_") + "_" + suffix) as the key, and
                    name as the corresponding value ]
    """

```

First, we form the Python equivalent of `name`. Because `name` comes out of the DOM, it will be a Unicode string, so we use the `str()` function to convert it to a regular string. The `.upper()` method uppercases it, and the `.translate()` method converts hyphens to underbars.

pyrang

```

#-- 1 --
# [ pyName := name, convert to str, uppercased, and with
#     hyphens converted to underbars, and "_" inserted
#     at each lowercase->uppercase transition ]
pyName = pythonizeName ( name )

```

The key consists of `pyName`, prefixed with `args.prefix`, with an underbar and the `suffix` argument appended.

pyrang

```

#-- 2 --
# [ key := args.prefix + pyName + "_" + suffix ]
key = "%s%s_%s" % (args.prefix, pyName, suffix)

```

Now we are ready to add the new entry to the table.

pyrang

```
#-- 3 --
# [ nameTable := nameTable with an entry whose key=key and
#     whose value=name ]
nameTable[key] = name
```

## 4.9. pythonizeName(): Sanitize an XML name for Python use

This function implements the various rules for converting an XML name.

pyrang

```
def pythonizeName ( s ):
    """Convert an XML name to its Python equivalent.
    """
```

The general approach will be to add the translated characters to a list named `xlated`. We start by creating this list empty.

pyrang

```
#-- 1 --
xlated = []
```

Next we work through the input string, adding each character's worth of content to the `xlated` list.

pyrang

```
#-- 2 --
# [ xlated += characters from s, convert to string type,
#     uppercasing lowercase characters, adding "_" at each
#     lowercase->uppercase transition, and converting "-"
#     to "-" ]
for i in range(len(s)):
    #-- 2 body --
    # [ if s[i] == '-' ->
    #     xlated += ['_']
    #     else if s[i] is lowercase ->
    #     xlated += [str(s[i]), uppercased]
    #     else if (s[i] is uppercase) and (i>0) and
    #     (s[i-1] is lowercase) ->
    #     xlated += ["-", str(s[i])]
    #     else ->
    #     xlated += [str(s[i])] ]

    #-- 2.1 --
    c = str(s[i])

    #-- 2.2 --
    if c == '-':
        xlated.append ( '_' )
    elif c.islower():
        xlated.append ( c.upper() )
    elif c.isupper():
        if ( ( i > 0 ) and
            ( s[i-1].islower() ) ):
```

```

        xlated.append ( '_' )
    xlated.append ( c )
else:
    xlated.append ( c )

```

The result is the concatenation of the elements of xlated.

pyrang

```

#-- 3 --
return "".join ( xlated )

```

## 4.10. writeOutput(): Generate the Python file

This function writes the actual Python declarations to the output file. This output is prefaced by a brief Python comment warning users not to edit the file, that it is produced by this script.

pyrang

```

def writeOutput ( args, outFile, nameTable ):
    """Output the Python assignment statements.

    [ (args is an Args object) and
      (outFile is a writeable file) and
      (nameTable is a dictionary whose keys are Python names
       and each value is a string) ->
      outFile +=: (opening comment) + (lines of the
                  form 'name = value', one for each entry in
                  nameTable) ]

    """

    #-- 1 --
    print >>outFile, (
        '''Do not edit this file.  It was produced automatically from\n"
        "  the %s schema by the %s script.\n"
        ''' % (args.inFileName, sys.argv[0]) )

```

Since the ordering of entries in a dictionary is arbitrary, we'll extract the keys and sort them so the output won't be completely random.

pyrang

```

#-- 2 --
# [ outFile +=: (lines of the form 'name = value', one for
#   each entry in nameTable) ]
keyList = nameTable.keys()
keyList.sort()
for key in keyList:
    print >>outFile, "%s = '%s'" % (key, nameTable[key] )

outFile.close()

```

## 4.11. class Args: Command line argument object

This class encapsulates the processing of the command line arguments.

```

class Args:
    """Represents the command line arguments.

    Exports:
    Args():
        [ if sys.argv contains valid command line arguments ->
          return a new Args object representing those arguments
          else ->
            sys.stderr += (usage message) + (error message)
            stop execution ]
    .inFileName:
        [ the input file name argument from sys.argv ]
    .prefix:
        [ if sys.argv specifies a prefix argument ->
          that argument as a string
          else -> "" ]
    .outFileName:
        [ if sys.argv specifies an output file name ->
          that file name as a string
          else -> None ]
    """

```

The `SysArgs` class divides command line arguments into switches (such as `-p`) and positional arguments. As class variables, we now define symbolic names for the command line switches and positional arguments.

```

PREFIX_SWITCH = 'p'
OUTFILE_SWITCH = 'o'
INFILE_ARG = 'inFile'

```

The `SysArgs` constructor expects a list of `SwitchArg` objects defining the switch-type arguments, and a list of `PosArg` objects defining the positional arguments. We define these also as class variables.

```

SWITCH_SPECS = [
    SwitchArg ( PREFIX_SWITCH,
                [ "Prefix for each generated name" ], takesValue=1 ),
    SwitchArg ( OUTFILE_SWITCH,
                [ "Optional output file name" ], takesValue=1 ) ]
POS_SPECS = [
    PosArg ( INFILE_ARG,
             [ "Name of input .rng file" ] ) ]

```

## 4.12. `Args.__init__()`: Constructor

This class checks the command line arguments and makes their values available.

```

def __init__ ( self ):
    """Check and process command line arguments.
    """

```

Much of the work of checking and collecting command line arguments is done by the `SysArgs` class from the author's library module `sysargs.py` to do preliminary argument processing; for documentation on that class, see the author's Python library page<sup>9</sup>.

pyrang

```
#-- 1 --
# [ if sys.argv contains valid command line arguments ->
#     sysArgs := a SysArgs object representing those
#             arguments
# else ->
#     sys.stderr += (usage message) + (error message)
#     stop execution ]
sysArgs = SysArgs ( self.SWITCH_SPECS, self.POS_SPECS )
```

At this point we know that all switches and positional arguments were valid, so we can copy them over to our exported attributes.

pyrang

```
#-- 2 --
# [ if sysArgs.switchMap has a key self.PREFIX_SWITCH ->
#     self.prefix := the corresponding value
# else ->
#     self.prefix := "" ]
self.prefix = sysArgs.switchMap[self.PREFIX_SWITCH]
if self.prefix is None:
    self.prefix = ""

#-- 3 --
# [ if sysArgs.switchMap has a key self.OUTFILE_SWITCH ->
#     self.outFileName := the corresponding value
# else ->
#     self.outFileName := None ]
try:
    self.outFileName = sysArgs.switchMap[self.OUTFILE_SWITCH]
except KeyError:
    self.outFileName = None

#-- 4 --
self.inFileName = sysArgs.posMap[self.INFILE_ARG]
```

## 4.13. Code epilogue

This is the last bit of code in the script; it calls the `main()` function to start execution.

pyrang

```
#####
# Epilogue
#-----

if __name__ == '__main__':
    main()
```

<sup>9</sup> <http://infohost.nmt.edu/~shipman/soft/clean/lib.html>