Practical 3: Reading XML using XPath

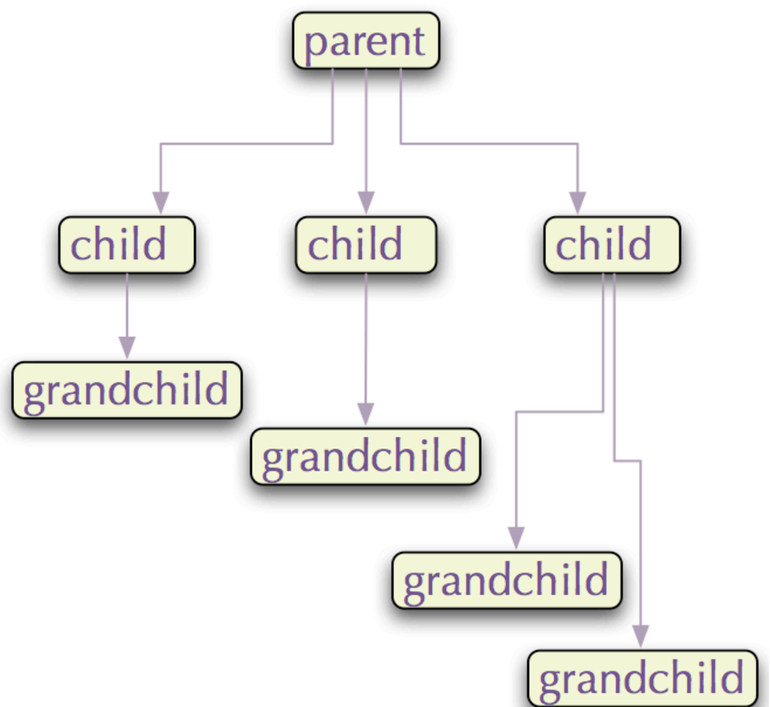
In Practical 2 you looked at ways of producing XML output from Fortran codes. In this afternoon's exercises you will look using the XML in ways beyond reading it into a web browser or Google Earth. The aim of the session is to give you an understanding of XPath, a rather high level interface to data from an XML document used by many XML technologies. It's worth designing your documents so that they can be easily used with XPath expressions. However, there is no Fortran XPath interface, so this exercise will be done using other tools including Python or Perl.

Don't worry though, you don't need to know either scripting language, and XPath is a language-agnostic interface; for this exercise you will only be doing XPath, not writing code. But this also reflects practicalities, in that even for data produced by Fortran codes, much analysis and post-processing is done using scripting languages such as Perl or Python.

The files needed for this practical can be found in the directory `~/Practical_3`. Some of the exercises in this practical can be performed in more then one language (Perl, Python and sometimes Shell examples are given), unless you have a particular reason to choose one of these we recommend you work on the Python examples as Python has a cleaner interface to the XPath library you will be using.

Exercise 3.1: xmllint

Before you get started with XPath proper, you should spend a bit of time exploring a simple document using a program called 'xmllint'. This is a command line interface to the C library libXML2. You'll be using bindings to this library from scripting languages in the later parts of this practical. xmllint and the underlying library are installed by default on all modern linux and Mac OS X computers.



In the exercise_1 directory you will find a simple XML document (document.xml) which represents data similar to the structure on the right. The first use of xmllint is to check if a document is well formed. This can be done by running the command:

```
xmllint document.xml
```

if the document is well formed then the XML file will be printed to the screen (useful for piping into a second process), but if it is not an error message will be reported including a line number of where the error occurred. This can be an incredibly useful tool when debugging! Is the provided document well formed? If not, how can you fix it?

For the record, there is an error (a missing ‘>’) at the end of the last line of the provided document. Once you have fixed this you can use xmllint to interactively explore the XML file, do the following:

```
xmllint document.xml --shell
```

This will give you a prompt, looking like:

```
/ >
```

You can use this prompt in the same way as a normal shell prompt - type commands into it, and then press return for them to be executed.

By starting up xmllint in this way, it lets you walk around the XML document, as if it were a filesystem tree, as described in the lecture. At the start of a session, you are placed at the top of the document tree.

So you are now in the XML document at the top level. You can navigate it as if were a directory tree, using 'cd' to change directory, and 'ls' to look at directory contents. So, as a first step, do:

```
/ > cd parent
```

Note that as you do this, the prompt changes to show you where you are in the tree.

```
parent > ls
```

You will see a list of results. These are all the XML elements which are the top-most children of the XML file. That is, the top-level 'directories' if this were a filesystem tree. There should be three 'child' elements. (You can ignore the 1st and 2nd columns of output.)

Let's look at the first element. There are two ways to do this:

```
parent > cat child[1]
```

shows you the raw XML for that element and:

```
parent > dir child[1]
```

shows you what information the XML actually contains.

The arguments you have been providing are actually XPath expressions. So you can get all of the grandchild names by doing:

```
parent > ls child/grandchild/@name
```

or:

```
parent > cat child/grandchild/@name
```

Here the ‘@’ is asking for an attribute of with the name ‘name’. But what if you want to select the date of birth (born attribute) of, say, Elizabeth I? or the names of the children of Margaret Tudor? To do this you will need to add a ‘predicate’, a condition that must be met for the XPath expression to match. These are included in square brackets. The date of birth of Elizabeth I can be found by doing:

```
cat child/grandchild[@name='Elizabeth I']/@born
```

and the children of Margaret Tudor can be found by doing:

```
cat child[@name='Margaret Tudor']/grandchild/@name
```

In this exercise we have only scratched the surface of xmllint’s capabilities. It can be used for a wide range of XML related tasks including validation, changing the text encoding, canonicalisation and processing Xinclude statements but rather then looking at this in any depth we’ll now move on to look at building XPath expressions in scripting environments. One thing we have not touched on is namespaces in xmllint. This is not because they are not supported, but because the commands needed rapidly get quite involved and (as we’ll see below) namespaces are easily handled from Python.

Exercise 3.2: Exploring XPath

Files for this exercise can be found in the exercise_2 directory. The file `xpather.py` is a simple python script designed to read an XML document, run an XPath query and print the result. The XML document `monty.xml` contains some quotations from the film “Monty Python and the Holy Grail”. Take a look at the XML file and the `xpather.py` script and try to deduce what will be output when the script is run. Now run the script by typing:

```
python xpather.py
```

You should see “`[‘Monty Python and the Holy Grail’]`” - which is python’s way of printing a one element list (array) with a single string element.

For the rest of this part of the exercise you should change the XPath query to extract other data. To change the `xpather.py` script you just need to edit and save it- there is no need to compile a python script. The only line you will need to change is:

```
answer = docRoot.xpath("/film/@name")
```

Modify the XPath query to extract the following information:

1. The date of the film encoded in the “<data date='1975'/>” element. *Hint: you will need to add an additional location step in the XPath query, and modify the attribute name.*

   XPath query: ____

2. The names of all five of the listed Pythons. *Hint: The fact that this requires a list of names to be returned does not add to the complexity of the XPath query needed, again you just need to change a location step and modify the attribute name.*

   XPath query: ____

3. The quotations of all the characters. *Hint: you won t need any attributes, just three location steps, and you will need /text() to recover the text.*

   XPath query: ____

4. You can also complete part 3 with a single location step. Can you construct such an XPath expression?

   XPath query: ____

5. Modify your solution to part 2 so that only the name of the character played by Terry Gilliam is reported. *Hint: You will need to use a predicate (in square brackets).*

   XPath query: ____

6. Write a query to return the quotation of Eric Idle.

   XPath query: ____

7. Write a query to return the quotation of the character Sir Bedevere.

   XPath query: ____

Exercise 3.3: XPath with Namespaces.
In the exercise_3 directory you should also find a file called `monty_ns.xml`. As the name suggests, this is a version of `monty.xml` with XML namespaces added. We saw in Practical 2 that documents with multiple namespaces are a useful way to combine different XML vocabularies in the same file. In this exercise we you will examine ways to use XPath on such a document. In `monty_ns.xml` I have imagined that two XML vocabularies exist, one to describe films with the namespace `http://www.example.com/films`, and one to describe quotations with the namespace `http://www.example.com/quotes`. We will use this mixed vocabulary for this exercise.

First try some of the solutions to exercise 3.2 with the file `monty_ns.xml`. A copy of `xpather.py` is included in the directory. Note that the file name `monty.xml` has been changed to `monty_ns.xml` on the line to load the XML:

```
docRoot = lxml.etree.parse(source="monty_ns.xml")
```

Do any of the previous expressions work?

You should find that all the expressions return an empty list (‘`[]`’). The XPath expressions have not matched any nodes. This is because we have not specified the namespace and so the XPath library is only looking for nodes with no defined namespace. All nodes in `monty_ns.xml` have a namespace defined, either directly with an ‘`xmlns=`’ declaration, or by inheritance.

The file `xpather_ns.py` is a python script set up to do the same work as `xpather.py`, but for namespaced documents. Take a look at the `xpather_ns.py file`. You should note two changes. First the line:

```
namespaces = {'q':'http://www.example.com/quotes',
              'f':'http://www.example.com/films'}
```

has been added. This declares a python dictionary of namespaces and related local short names (f and q) to enable their use in XPath expressions with much less typing. Secondly, the call to the XPath library has been modified:

```
answer = docRoot.xpath("/f:film/@name", namespaces)
```

to tell the library to use the dictionary of namespaces. One way to think of this is that the dictionary lists all namespaces this script is designed to know about. XPath expressions will simply ignore namespaces that the script does not understand.

Run `xpather_ns.py` - does it work? You should now repeat Exercise 3.2 with this namespace aware version.

1. The date of the film encoded in the "`<data date='1975'/>`" element. Do you need to give a namespace to each data element in the search path, or is the namespace inherited through the query?

   XPath query: ____

2. The names of all five of the listed Pythons.

   XPath query: ____

3. The quotations of all the characters. *Hint: remember that <quote> elements are in a different namespace to <film>, <data> and <comic>.*

   XPath query: ____

4. You can also complete part 3 with a single location step. Can you construct such an XPath expression?

   XPath query: ____

5. Modify your solution to part 2 so that only the name of the character played by Terry Gilliam.

   XPath query: ____

6. Write a query to return the quotation of Eric Idle.

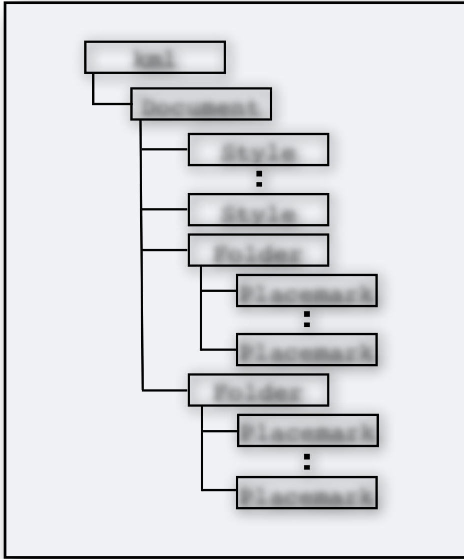
   XPath query: ____

7. Write a query to return the quotation of the character Sir Bedevere.

   XPath query: ____

Exercise 3.4: Matching Nodes

The remainder of this morning's practical involves using XPath to extract data from the mixed namespace KML document produced by yesterday's final exercise. In case you did not finish the final part of that exercise, a suitable XML document named `hypoDD.kml` is provided in the exercise_4 directory along with all the python scripts needed for the the remainder of this practical.



The basic structure a typical KML document is given in the figure on the left in terms of a tree-like representation (not all elements are shown). The document root is represented by a `<kml>` element, this (may) have one or more `<Document>` child elements called, each of which contains some number of `<Style>`s and `<Folder>`s. The python script "`document_info.py`" is designed to print some information about the KML file, but the XPath expressions are missing. Fill them in to make the script work.

The first XPath expression is intended to return the name(s) of any `<Document>`s which are children of the root element. The second two expressions are supposed to return numbers, specifically the number of `<Style>`s and `<Folder>`s belonging to the `<Document>`s.

First XPath query: ____

Second XPath query: ____

Third XPath query: ____

Hint: Absolute XPath expressions (starting with a / including all the elements needed to find the information) can be used for this exercise. The XPath function count() can be used to find the number of nodes returned by an XPath query.

Exercise 3.5: Nodesets and loops

The python script `folder_info.py` is designed to extract some simple information regarding all the folders within a KML document. This script is intended to perform data extraction in two stages. The first query should extract a list of all folders and stores the result in the variable 'folders' (which will be a list). This XPath query should match all `<Folder>` elements in the document wherever they are located. The second XPath query is located within a loop which extracts each node in series from the list in the 'folders' variable and places it in the object 'node' which is passed to the subsequent XPath expressions. These two expressions need to contain relative XPath queries (not starting with a '/'). The first should return the name of the folder represented by 'node' and the second should count the number of Placemarks within the folder.

First XPath query: ____

Second XPath query: ____

Third XPath query: ____

Exercise 3.6: Analysis
The python script `calculate_moves.py` performs some more involved analysis of the output data using the quakeML data embedded within the KML document. Specifically, the script is designed to work out how far each of the earthquake locations have been moved during the *hypoDD* run represented by the document. Six XPath expressions are needed. In order, these need to:

1. Extract a list of placemarks within the folder with the name 'Initial positions'.

   XPath query: ____

2. Extract the `unique_id` from the quakeML location element embedded in each placemark in turn. This expression involves a change of namespace.

   XPath query: ____

3. Extract the latitude and longitude of each event.

   XPath query: ____

   XPath query: ____

4. Find the latitude and longitude of the event with a matching unique_id from the 'Final positions' folder.

   XPath query: ____

The script should then print the distance that *hypoDD* has moved each earthquake during its refinement process.