# 2.1 File processing - XML files

## XML processing in Python

Python is commonly used to process various types of data. Perhaps, while working as a programmer, you'll have to read or create a data file in the XML format. Soon, doing that will be a piece of cake.

The standard Python library offers some interesting modules for working with XML:

- **xml.etree.ElementTree** – has a very simple API for analyzing and creating XML data. It's an excellent choice for people who have never worked with the Document Object Model (DOM) before.
- **xml.dom.minidom** – is the minimum implementation of the Document Object Model (DOM). Using the DOM, the approach to an XML document is slightly different, because it's parsed into a tree structure in which each node is an object.
- **xml.sax** – SAX is an acronym for “Simple API for XML”. SAX is an interface in Python for event-driven XML document analysis. Unlike the above modules, analyzing a simple XML document using this module requires more work.

In this course, you'll learn how to create and process XML documents using the `xml.etree.ElementTree` module. Let's not waste any time. Let's go!

## What is XML?

Extensible Markup Language (XML) is a markup language intended for storing and transporting data, e.g., by systems using the SOAP communication protocol. One of its main advantages is the ability to define your own tags that make the document more readable to humans. XML is a standard recommended by the W3C organization. Let's look at what elements XML documents contain:

- **prolog** – the first (optional) line of the document. In the prolog, you can specify character encoding, e.g., `<?xml version=«1.0» encoding=«ISO-8859-2»?> ''changes the default character encoding (UTF-8) to ISO-8859-2. * **root element** – the XML document must have one root element that contains all other elements. In the example below, the main element is the ''data` tag.
- **elements** – these consist of opening and closing tags. The elements include text, attributes, and other child elements. In the example below, we can find the `book` element with the title attribute and two child elements (`author` and `year`).
- **attributes** – these are placed in the opening tags. They consist of key-value pairs, e.g., `title = «The Little Prince»`.
- NOTE**: Each open XML tag must have a corresponding closing tag.**

```
<?xml version="1.0"?>
<data>
    <book title="The Little Prince">
        <author>Antoine de Saint-Exupéry</author>
        <year>1943</year>
    </book>
    <book title="Hamlet">
        <author>William Shakespeare</author>
        <year>1603</year>
    </book>
</data>
```

# XML parsing (part 1)

Processing XML files in Python is very easy thanks to the `ElementTree` class provided by the `xml.etree.ElementTree` module. The `ElementTree` object is responsible for presenting the XML document in the form of a tree on which we can move up or down.

First, we need to import the appropriate module and define an alias for it. It's common to use the alias ET, but of course you can give it any name you like. To create a tree (an `ElementTree` object) from an existing XML document, pass it to the parse method as follows:

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
```

The `getroot` method returns the root element. With access to the root element, we can reach any elements in the document. Each of these elements is represented by a class called `Element`.

In addition to the parse method, we can use the method called `fromstring`, which, as an argument, takes XML as a string:

```
import xml.etree.ElementTree as ET

root = ET.fromstring(your_xml_as_string)
```

**NOTE**: The `fromstring` method doesn't return the `ElementTree` object, but instead returns the root element represented by the Element class.

# XML parsing (part 2)

You already know how to access the root element. Let's use it to visit the elements of our tree - look at the code in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
print('The root tag is:', root.tag)
print('The root has the following children:')
for child in root:
    print(child.tag, child.attrib)
```

Result:

```
The root tag is: data
The root has the following children:
book {'title': 'The Little Prince'}
book {'title': 'Hamlet'}
```

The root element and all its children are `Element` objects. In the example above, we use the following properties:

**tag** – this returns the tag name as a string

**attrib** – this returns all attributes in the tag as a dictionary. To retrieve the value of a single attribute, use its key, e.g., `child.attrib ['title']`.

## XML parsing (part 3)

In addition to iterating over tree elements, we can access them directly using indexes. Take a look at the example below in which we use the current book element to retrieve text from its child elements. Look at the code in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
print("My books:\n")
for book in root:
    print('Title: ', book.attrib['title'])
    print('Author:', book[0].text)
    print('Year: ', book[1].text, '\n')
```

Result:

```
My books:

Title:  The Little Prince
Author: Antoine de Saint-Exupéry
Year:  1943

Title:  Hamlet
Author: William Shakespeare
Year:  1603
```

During each iteration, we refer to the children of the book element by using indexes. Index `0` refers to the first child of the book element, while index`1` refers to its second child. Displaying text is possible thanks to the `text` property, available in the `Element` object. **NOTE**: Indexes are also used in deeper nesting, e.g., `root [0] [0] .text` returns the first book element, and then displays the text placed in its first child.

## XML parsing (part 4)

The `xml.etree.ElementTree` module, or more precisely, the `Element` class, provides several useful methods for finding elements in an XML document. Let's start with the method called `iter`.

The `iter` method returns all elements by having the tag passed as an argument. The element that calls it is treated as the main element from which the search starts. In order to find all matches, the iterator iterates recursively through all child elements and their nested elements.

Look at the code in the editor to see an example of a search for all items with the `author` tag.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for author in root.iter('author'):
```

```
    print(author.text)
```

Result:

```
Antoine de Saint-Exupéry
William Shakespeare
```

## XML parsing (part 5)

The Element object has a method called `findall` to search for direct child elements. Unlike the `iter` method, the `findall` method only searches the children at the first nesting level. Take a look at the example in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for author in root.findall('author'):
    print(author.text)
```

The example doesn't return any results, because the `findall` method can only iterate over the book elements that are the closest children of the root element. The correct code looks like this:

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for book in root.findall('book'):
    print(book.get('title'))
```

Result:

```
The Little Prince
Hamlet
```

To display the value of the title attributes, we use the `get` method, which looks much better than a `book.attrib ['title']` call. Its use is very simple: just enter the name of the attribute and optionally (as a second argument) the value that will be returned if the attribute is not found (the default is `None`).

**NOTE**: The `findall` method also accepts an XPath expression. We encourage you to deepen your knowledge of XPath expressions and apply it to the example shown.

## XML parsing (part 6)

Another useful method used to parse an XML document is a method called `find`. The `find` method returns the first child element containing the specified tag or matching XPath expression. Look at the code in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()

```

```
print(root.find('book').get('title'))
```

Result:

```
The Little Prince
```

In the example, we use the find method to `find` the first child element containing the `book` tag, and then display the value of its title attribute. Note that the element from which we start the search is the root element.

## Modifying an XML document (part 1)

You've already learned how to parse an XML document. Time for the next step. Let's modify the element tree and create a new XML file based on it with the following movie data:

```
<?xml version="1.0"?>
<data>
    <movie title="The Little Prince" rate="5"></movie>
    <movie title="Hamlet" rate="5"></movie>
</data>
```

Are you wondering if it will be difficult to convert book data to movie data? Thanks to the `xml.etree.ElementTree` module, it's a piece of cake. To change the tag of the `Element` object, we must assign a new value to its tag property. Look at the code in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

Result:

```
movie {'title': 'The Little Prince'}
author : Antoine de Saint-Exupéry
year : 1943
movie {'title': 'Hamlet'}
author : William Shakespeare
year : 1603
```

In the example, during each iteration through the `book` elements, we replace them with the `movie` tag, and then make sure that all changes have gone through correctly.

## Modifying an XML document (part 2)

Our XML has a few unnecessary elements, such as `author` and `year`. In order to remove them, we need to use the method called `remove`, provided by the `Element` class. Look at the code in the editor.

```
import xml.etree.ElementTree as ET
```

```
tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

Result:

```
movie {'title': 'The Little Prince'}
movie {'title': 'Hamlet'}
```

The `remove` method removes the child element passed as its argument, which must be an `Element` object. Note that for this purpose we use the method called `find`, which you're already familiar with.

## Modifying an XML document (part 3)

Do you remember the `get` method that gets the value of the attribute? The `Element` object also has a method called `set`, which allows you to set any attribute. Look at the code in the editor.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    child.set('rate', '5')
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)
```

Result:

```
movie {'title': 'The Little Prince', 'rate': '5'}
movie {'title': 'Hamlet', 'rate': '5'}
```

The `set` method takes the attribute name and its value as arguments. In our case, we use it to set the highest rating for each of the movies.

## Modifying an XML document (part 4)

You must have noticed that the modified XML document is not saved anywhere. To save all the changes we've made to the tree, we have to use the method called `write`.

The `write` method has only one required argument, which is a file name of the output XML file, or a file object opened for writing. In addition, we can define character encoding by using the second argument (the default is

US-ASCII). To add a prolog to our document, we must pass True in the third argument.

Take a look at the example in the editor, in which we save the modified tree in a file called movies.xml.

```
import xml.etree.ElementTree as ET

tree = ET.parse('books.xml')
root = tree.getroot()
for child in root:
    child.tag = 'movie'
    child.remove(child.find('author'))
    child.remove(child.find('year'))
    child.set('rate', '5')
    print(child.tag, child.attrib)
    for sub_child in child:
        print(sub_child.tag, ':', sub_child.text)

tree.write('movies.xml', 'UTF-8', True)
```

The created file looks like this:

```
<?xml version='1.0' encoding='UTF-8'?>
<data><movie rate="5" title="The Little Prince" /><movie rate="5" title="Hamlet"
/></data>
```

## Building an XML document (part 1)

During this course, you haven't had the opportunity to create an Element object yourself. Let's see how to build an XML document in Python.

The Element class constructor takes two arguments. The first is the name of the tag to be created, while the second (optional) is the attribute dictionary. In the example in the editor, we've created the root element represented by a data tag with no attributes - look at the code.

```
import xml.etree.ElementTree as ET

root = ET.Element('data')
ET.dump(root)
```

Result:

```
<data />
```

In the example above, we use the dump method, which allows us to debug either the whole tree or a single element.

## Building an XML document (part 2)

In addition to the Element class, the xml.etree.ElementTree module offers a function for creating child elements called SubElement. The SubElement function takes three arguments.

The first one defines the parent element, the second one defines the tag name, and the third (optional) defines

the attributes of the element. Let's see how it looks in action and analyze the code in the editor.

```
import xml.etree.ElementTree as ET

root = ET.Element('data')
movie_1 = ET.SubElement(root, 'movie', {'title': 'The Little Prince', 'rate': '5'})
movie_2 = ET.SubElement(root, 'movie', {'title': 'Hamlet', 'rate': '5'})
ET.dump(root)
```

Result:

```
<data><movie rate="5" title="The Little Prince" /><movie rate="5" title="Hamlet"
/></data>
```

In the example, we've added two child elements called `movie` to the root element. The created elements are objects of the `Element` class, so we can use all of the methods that we learned about during this course.

**NOTE**: To save a document using the write method, we need to have an `ElementTree` object. To do this, pass our root element to its constructor:

```
tree = ET.ElementTree(root)
```

From:
https://matebcn.eu/ - **miguel angel torres egea**

Permanent link:
**https://matebcn.eu/info:cursos:pue:python-pcpp1:m5:2.1**

Last update: **04/03/2024 11:37**