

Reading and Writing XML Files in Python

By  Scott Robinson (<https://twitter.com/ScottWRobinson>) • November 30, 2017 •

[1 Comment \(/reading-and-writing-xml-files-in-python/#disqus_thread\)](#)

XML, or Extensible Markup Language, is a markup-language that is commonly used to structure, store, and transfer data between systems. While not as common as it used to be, it is still used in services like RSS and SOAP, as well as for structuring files like Microsoft Office documents.

With Python being a popular language for the web and data analysis, it's likely you'll need to read or write XML data at some point, in which case you're in luck.

Throughout this article we'll primarily take a look at the `ElementTree` (<https://docs.python.org/3/library/xml.etree.elementtree.html>) module for reading, writing, and modifying XML data. We'll also compare it with the older `minidom` (<https://docs.python.org/3/library/xml.dom.minidom.html>) module in the first few sections so you can get a good comparison of the two.

The XML Modules

The `minidom`, or Minimal DOM Implementation, is a simplified implementation of the Document Object Model (DOM). The DOM (https://en.wikipedia.org/wiki/Document_Object_Model) is an application programming interface that treats XML as a tree structure, where each node in the tree is an object. Thus, the use of this module requires that we are familiar with its functionality.

The `ElementTree` module provides a more "Pythonic" interface to handling XML and is a good option for those not familiar with the DOM. It is also likely a better candidate to be used by more novice programmers due to its simple interface, which you'll see throughout this article.

In this article, the `ElementTree` module will be used in all examples, whereas `minidom` will also be demonstrated, but only for counting and reading XML documents.

XML File Example

In the examples below, we will be using the following XML file, which we will save as "items.xml":

```
<data>
  <items>
    <item name="item1">item1abc</item>
    <item name="item2">item2abc</item>
  </items>
</data>
```

As you can see, it's a fairly simple XML example, only containing a few nested objects and one attribute. However, it should be enough to demonstrate all of the XML operations in this article.

Reading XML Documents

Using minidom

In order to parse an XML document using `minidom`, we must first import it from the `xml.dom` module. This module uses the `parse` function to create a DOM object from our XML file. The `parse` function has the following syntax:

```
xml.dom.minidom.parse(filename_or_file[, parser[, bufsize]])
```

Here the file name can be a string containing the file path or a file-type object. The function returns a document, which can be handled as an XML type. Thus, we can use the function `getElementByTagName()` to find a specific tag.

Since each node can be treated as an object, we can access the attributes and text of an element using the properties of the object. In the example below, we have accessed the attributes and text of a specific node, and of all nodes together.

```
from xml.dom import minidom

# parse an xml file by name
mydoc = minidom.parse('items.xml')

items = mydoc.getElementsByTagName('item')

# one specific item attribute
print('Item #2 attribute:')
print(items[1].attributes['name'].value)

# all item attributes
print('\nAll attributes:')
for elem in items:
    print(elem.attributes['name'].value)

# one specific item's data
print('\nItem #2 data:')
print(items[1].firstChild.data)
print(items[1].childNodes[0].data)

# all items data
print('\nAll item data:')
for elem in items:
    print(elem.firstChild.data)
```

The result is as follows:

```
$ python minidomparser.py
Item #2 attribute:
item2

All attributes:
item1
item2

Item #2 data:
item2abc
item2abc

All item data:
item1abc
item2abc
```

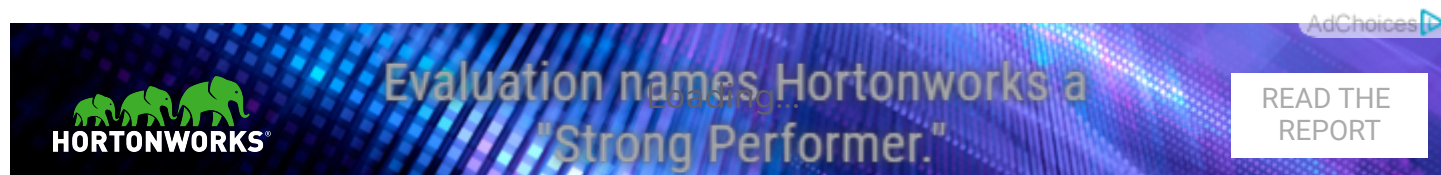


Figure 1

If we wanted to use an already-opened file, can just pass our file object to `parse` like so:

```
datasource = open('items.xml')

# parse an open file
mydoc = parse(datasource)
```

Also, if the XML data was already loaded as a string then we could have used the `parseString()` function instead.

Using ElementTree

`ElementTree` presents us with an very simple way to process XML files. As always, in order to use it we must first import the module. In our code we use the `import` command with the `as` keyword, which allows us to use a simplified name (`ET` in this case) for the module in the code.

Following the import, we create a tree structure with the `parse` function, and we obtain its root element. Once we have access to the root node we can easily traverse around the tree, because a tree is a connected graph.

Using `ElementTree` , and like the previous code example, we obtain the node attributes and text using the objects related to each node.

The code is as follows:

```
import xml.etree.ElementTree as ET
tree = ET.parse('items.xml')
root = tree.getroot()

# one specific item attribute
print('Item #2 attribute:')
print(root[0][1].attrib)

# all item attributes
print('\nAll attributes:')
for elem in root:
    for subelem in elem:
        print(subelem.attrib)

# one specific item's data
print('\nItem #2 data:')
print(root[0][1].text)

# all items data
print('\nAll item data:')
for elem in root:
    for subelem in elem:
        print(subelem.text)
```

The result will be as follows:

```
$ python treeparser.py
Item #2 attribute:
item2

All attributes:
item1
item2

Item #2 data:
item2abc

All item data:
item1abc
item2abc
```

Figure 2

As you can see, this is very similar to the `minidom` example. One of the main differences is that the `attrib` object is simply a dictionary object, which makes it a bit more compatible with other Python code. We also don't need to use `value` to access the item's attribute value like we did before.

You may have noticed how accessing objects and attributes with `ElementTree` is a bit more Pythonic, as we mentioned before. This is because the XML data is parsed as simple lists and dictionaries, unlike with `minidom` where the items are parsed as custom `xml.dom.minidom.Attr` and "DOM Text nodes".

Counting the Elements of an XML Document

Using minidom

As in the previous case, the `minidom` must be imported from the `dom` module. This module provides the function `getElementsByTagName`, which we'll use to find the tag item. Once obtained, we use the `len()` built-in method to obtain the number of sub-items connected to a node. The result obtained from the code below is shown in *Figure 3*.

```
from xml.dom import minidom

# parse an xml file by name
mydoc = minidom.parse('items.xml')

items = mydoc.getElementsByTagName('item')

# total amount of items
print(len(items))

$ python counterxmldom.py
2
```

Figure 3

Keep in mind that this will only count the number of children items under the node you execute `len()` on, which in this case is the root node. If you want to find all sub-elements in a much larger tree, you'd need to traverse all elements and count each of their children.

Using ElementTree

Similarly, the `ElementTree` module allows us to calculate the amount of nodes connected to a node.

Example code:

```
import xml.etree.ElementTree as ET
tree = ET.parse('items.xml')
root = tree.getroot()

# total amount of items
print(len(root[0]))
```

The result is as follows:

```
$ python counterxml.py
2
```

Figure 4

Writing XML Documents

Using ElementTree

`ElementTree` is also great for writing data to XML files. The code below shows how to create an XML file with the same structure as the file we used in the previous examples.

The steps are:

1. Create an element, which will act as our root element. In our case the tag for this element is "data".
2. Once we have our root element, we can create sub-elements by using the `SubElement` function. This function has the syntax:

```
SubElement(parent, tag, attrib={}, **extra)
```

Here `parent` is the parent node to connect to, `attrib` is a dictionary containing the element attributes, and `extra` are additional keyword arguments. This function returns an element to us, which can be used to attach other sub-elements, as we do in the following lines by passing items to the `SubElement` constructor.

3. Although we can add our attributes with the `SubElement` function, we can also use the `set()` function, as we do in the following code. The element text is created with the `text` property of the `Element` object.
4. In the last 3 lines of the code below we create a string out of the XML tree, and we write that data to a file we open.

Example code:

```
import xml.etree.ElementTree as ET

# create the file structure
data = ET.Element('data')
items = ET.SubElement(data, 'items')
item1 = ET.SubElement(items, 'item')
item2 = ET.SubElement(items, 'item')
item1.set('name', 'item1')
item2.set('name', 'item2')
item1.text = 'item1abc'
item2.text = 'item2abc'

# create a new XML file with the results
mydata = ET.tostring(data)
myfile = open("items2.xml", "w")
myfile.write(mydata)
```

Executing this code will result in a new file, "items2.xml", which should be equivalent to the original "items.xml" file, at least in terms of the XML data structure. You'll probably notice that the resulting string is only one line and contains no indentation, however.

Finding XML Elements

Using ElementTree

The `ElementTree` module offers the `findall()` function, which helps us in finding specific items in the tree. It returns all items with the specified condition. In addition, the module has the function `find()`, which returns only the *first* sub-element that matches the specified criteria. The syntax for both of these functions are as follows:

```
findall(match, namespaces=None)
```

```
find(match, namespaces=None)
```

For both of these functions the `match` parameter can be an XML tag name or a path. The function `findall()` returns a list of elements, and `find` returns a single object of type `Element`.

In addition, there is another helper function that returns the text of the first node that matches the given criterion:

```
findtext(match, default=None, namespaces=None)
```

Here is some example code to show you exactly how these functions operate:

```
import xml.etree.ElementTree as ET
tree = ET.parse('items.xml')
root = tree.getroot()

# find the first 'item' object
for elem in root:
    print(elem.find('item').get('name'))

# find all "item" objects and print their "name" attribute
for elem in root:
    for subelem in elem.findall('item'):

        # if we don't need to know the name of the attribute(s), get the dict
        print(subelem.attrib)

        # if we know the name of the attribute, access it directly
        print(subelem.get('name'))
```

And here is the result of running this code:

```
$ python findtree.py
item1
{'name': 'item1'}
item1
{'name': 'item2'}
item2
```

Figure 5

Modifying XML Elements

Using ElementTree

The `ElementTree` module presents several tools for modifying existing XML documents. The example below shows how to change the name of a node, change the name of an attribute and modify its value, and how to add an extra attribute to an element.

A node text can be changed by specifying the new value in the text field of the node object. The attribute's name can be redefined by using the `set(name, value)` function. The `set` function doesn't have to just work on an existing attribute, it can also be used to define a new attribute.

The code below shows how to perform these operations:

```
import xml.etree.ElementTree as ET

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

# changing a field text
for elem in root.iter('item'):
    elem.text = 'new text'

# modifying an attribute
for elem in root.iter('item'):
    elem.set('name', 'newitem')

# adding an attribute
for elem in root.iter('item'):
    elem.set('name2', 'newitem2')

tree.write('newitems.xml')
```

After running the code, the resulting XML file "newitems.xml" will have an XML tree with the following data:

```
<data>
  <items>
    <item name="newitem" name2="newitem2">new text</item>
    <item name="newitem" name2="newitem2">new text</item>
  </items>
</data>
```

As we can see when comparing with the original XML file, the names of the item elements have changed to "newitem", the text to "new text", and the attribute "name2" has been added to both nodes.

You may also notice that writing XML data in this way (calling `tree.write` with a file name) adds some more formatting to the XML tree so it contains newlines and indentation.

Creating XML Sub-Elements

Using ElementTree

The `ElementTree` module has more than one way to add a new element. The first way we'll look at is by using the `makeelement()` function, which has the node name and a dictionary with its attributes as parameters.

The second way is through the `SubElement()` class, which takes in the parent element and a dictionary of attributes as inputs.

In our example below we show both methods. In the first case the node has no attributes, so we created an empty dictionary (`attrib = {}`). In the second case, we use a populated dictionary to create the attributes.


```
import xml.etree.ElementTree as ET

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

# adding an element to the root node
attrib = {}
element = root.makeelement('seconditems', attrib)
root.append(element)

# adding an element to the seconditem node
attrib = {'name2': 'secondname2'}
subelement = root[0][1].makeelement('seconditem', attrib)
ET.SubElement(root[1], 'seconditem', attrib)
root[1][0].text = 'seconditemabc'

# create a new XML file with the new element
tree.write('newitems2.xml')
```

After running this code the resulting XML file will look like this:

```
<data>
  <items>
    <item name="item1">item1abc</item>
    <item name="item2">item2abc</item>
  </items>
  <seconditems>
    <seconditem name2="secondname2">seconditemabc</seconditem>
  </seconditems>
</data>
```

As we can see when comparing with the original file, the "seconditems" element and its sub-element "seconditem" have been added. In addition, the "seconditem" node has "name2" as an attribute, and its text is "seconditemabc", as expected.

Deleting XML Elements

Using ElementTree

As you'd probably expect, the `ElementTree` module has the necessary functionality to delete node's attributes and sub-elements.

Deleting an attribute

The code bellow shows how to remove a node's attribute by using the `pop()` function. The function applies to the `attrib` object parameter. It specifies the name of the attribute and sets it to `None`.

```
import xml.etree.ElementTree as ET

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

# removing an attribute
root[0][0].attrib.pop('name', None)

# create a new XML file with the results
tree.write('newitems3.xml')
```

The result will be the following XML file:

```
<data>
  <items>
    <item>item1abc</item>
    <item name="item2">item2abc</item>
  </items>
</data>
```


As we can see in the XML code above, the first item has no attribute "name".

Deleting one sub-element

One specific sub-element can be deleted using the `remove` function. This function must specify the node that we want to remove.

The following example shows us how to use it:

```
import xml.etree.ElementTree as ET

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

# removing one sub-element
root[0].remove(root[0][0])

# create a new XML file with the results
tree.write('newitems4.xml')
```

The result will be the following XML file:

```
<data>
  <items>
    <item name="item2">item2abc</item>
  </items>
</data>
```

As we can see from the XML code above, there is now only one "item" node. The second one has been removed from the original tree.

Deleting all sub-elements

The `ElementTree` module presents us with the `clear()` function, which can be used to remove *all* sub-elements of a given element.

The example below shows us how to use `clear()` :

```
import xml.etree.ElementTree as ET

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

# removing all sub-elements of an element
root[0].clear()

# create a new XML file with the results
tree.write('newitems5.xml')
```

The result will be the following XML file:


```
<data>
  <items />
</data>
```

As we can see in the XML code above, all sub-elements of the "items" element have been removed from the tree.

Wrapping Up


Python offers several options to handle XML files. In this article we have reviewed the `ElementTree` module, and used it to parse, create, modify and delete XML files. We have also used the `minidom` model to parse XML files. Personally, I'd recommend using the `ElementTree` module as it is much easier to work with and is the more modern module of the two.

 `python (/tag/python/)`, `xml (/tag/xml/)`

 (<https://twitter.com/share?text=Reading%20and%20Writing%20XML%20Files%20in%20Python&url=https://stackabuse.com/reading-and-writing-xml-files-in-python/>)

 (<https://www.facebook.com/sharer/sharer.php?u=https://stackabuse.com/reading-and-writing-xml-files-in-python/>)

 (<https://plus.google.com/share?url=https://stackabuse.com/reading-and-writing-xml-files-in-python/>)

 (<https://www.linkedin.com/shareArticle?mini=true%26url=https://stackabuse.com/reading-and-writing-xml-files-in-python/%26source=https://stackabuse.com>)



([/author/scott/](#))

About Scott Robinson ([/author/scott/](#))

 [Twitter \(https://twitter.com/ScottWRobinson\)](https://twitter.com/ScottWRobinson)  [Github \(https://github.com/ScottWRobinson\)](https://github.com/ScottWRobinson)