Project Description

A high-level abstraction layer for creating and editing XML documents.

This high-level abstraction layer is implemented by using wrapper classes for the standard **xml.dom.minidom** library and the **defusedxml** package. There are two wrapper classes **:**

- **XDocument -** A defined class for creating, loading, and saving an XML document.
- **XElement -** A defined class which represents an XML document element.

The **XDocument** and **XElement** classes provide a concise, tree structure hierarchy for the XML document. Every XML document has a single root element. The root element can have multiple child elements. Each child element can have its own children, and so on and so forth. With the exception of comments, the XML document is composed entirely of elements. The following is a depiction of the XML document structure:

```
<root>
  <child>
    <grand_child>
      <great_grand_child/>
    </grand_child>
  </child>
</root>
```

The **XDocument** class provides all the functionality needed for creating a new XML document, loading the XML document from a file, and saving the XML document to a file. When loading from an existing file, this class uses the **defusedxml.minidom.parse()** function to generate a 'safe' XML document from the file. When saving the XML document to a file, the information is written in a human-readable format which depicts the tree structure hierarchy of the XML document.

Every element of the XML document is an instance of the **XElement** class. In addition to child elements, every element can have multiple attributes as well as a text value. This class provides a wide range of methods for adding, modifying, and removing elements in the XML document.

Installation

```
pip install xdocument
```

Quick Examples

Example #1

A bare minimum implementation the **xdocument** package would be:

```
from xdocument import XDocument

XDocument('Blank.xml')
```

Running this script produces a file named "Blank.xml", and the text content of this file is:

```
<?xml version="1.0" encoding="utf-8"?>
<XDocument/>
```

Initially, the "Blank.xml" file does not exist, so the **XDocument** constructor creates a new blank XML document (with only the root element) and saves it to a file named "Blank.xml". The default root name is "XDocument".

When the script is run again, the **XDocument** constructor will load the existing "Blank.xml" file as the XML document. It does not save the XML document back to the "Blank.xml" file.

Example #2

This example demonstrates how to populate an XML document with child elements.

```
from xdocument import XDocument

document = XDocument('Simple.xml', 'Root', 'A Simple XML Document')
if document.root.first_child is None:
    child = document.root.add('Child', 'Attrib', 'Value')
    child.add('GrandChild').value = 'Text'
    document.save()
```

As before, the **XDocument** constructor will create a new XML document and save it to a file named "Simple.xml". The two additional parameter values provide an optional name for the root element and the text field for an optional XML document comment.

The **if** statement checks for a newly created XML document. The **root** property of the **XDocument** class is the root element of the XML document. When the XML document is newly created, the root element has no children, and its **first_child** property will evaluate to **None.**

Moving inside the **if** statement code block, the first statement calls the root element's **add()** method. This method creates a new element named "Child", assigns it an attribute (optional), and makes it a child of the root element. This method also returns a reference to the newly created child element. In the second statement, a grandchild element is generated by calling the child element's **add()** method, and the **value** property of the grandchild is assigned a string value. The third statement saves the modified XML document to the "Simple.xml" file.

Running this script produces a file named "Simple.xml", and the text content of this file is:

```
<?xml version="1.0" encoding="utf-8"?>
<!--A Simple XML Document-->
<Root>
  <Child Attrib="Value">
    <GrandChild>Text</GrandChild>
  </Child>
</Root>
```

When the script is run again, the **XDocument** constructor will load the existing "Simple.xml" file as an XML document. The XML document's root element will now have a child element, and the **if** statement's code block will not be executed. An easy way to demonstrate this, is to append an **else** statement and its associated code block to the end of the previous script. The statements in this code block will be executed when there is an existing "Simple.xml" file. The following addition to the previous script will allow it to read and print both the attribute value of the Child element and the text value of the GrandChild element:

```
from xdocument import XDocument

document = XDocument('Simple.xml', 'Root', 'A Simple XML Document')
if document.root.first_child is None:
    child = document.root.add('Child', 'Attrib', 'Value')
    child.add('GrandChild').value = 'Text'
    document.save()
# -------------------------------------------------------------------
else:  # Read and print the contents of the XML document
    child = document.root.first_child
    print(f"Attribute Value = {child.read_attribute('Attrib')}")
    print(f"GrandChild Value = {child.read_child('GrandChild')}")
```

The reader is encouraged to make changes to the "Simple.xml" file by carefully editing the attribute value of the Child element (the quotation marks are required) and/or the text value of the GrandChild element. The reader should save these changes to the "Simple.xml" file and then run the script again to observe the results. In the event that these changes result in a file that cannot be successfully loaded; simply delete the "Simple.xml" file, and run the script to recreate the original "Simple.xml" file.

Documentation

XDocument

XDocument(*filename, root_name=' ', comment=' ', details=None*)

Create an XML document from the specified XML document file. If the specified XML document file does not exist, a new XML Document is created, initialized, and saved as a new XML document file.

- ***filename* -** The full filename of the specified XML document file.
- ***root_name* -** The optional XML document root name (the default is the class name).
- ***comment* -** The optional XML document comment text string.
- ***details* -** The optional initialization details for the derived subclass.

_subclass_details( ) - This method can be used to provide the initialization details for a derived subclass.

```
    if self._details is not None:
        pass  # Add the default implementation details here
```

This method is only called during the class initialization when the specified XML document file does not exist. Overriding this method allows the user to provide the default implementation details in the user's derived subclass. An example of overriding this method in a subclass is shown in the **XDocument Usage Example.**

Properties

- **root -** The root element of the XML Document (readonly).

Methods

- **create_element(*name*) -** Create a new element for the XML document. This method returns the newly created element.
    - ***name*** - The name of the element.
- **load(*filename*) -** Load a new XML document from the specified XML document file. If the file does not exist, this method creates a blank XML document which contains only the root element.
    - ***filename*** - The full filename of the specified XML document file.
- **save(*filename*=' ') -** Save the XML document to the specified XML document file.
    - ***filename*** - The optional filename (the default is the current filename).

XElement

Every element of the XML document is an instance of the **XElement** class, and every element has the following properties and methods.

Properties

- **attributes -** A list of all the attributes of this element (readonly).
- **children -** A list of all the child elements of this element (readonly).
- **first_attribute -** The first attribute of this element (readonly).
- **first_child -** The first child of this element (readonly).
- **has_attributes -** True if this element has at least one attribute (readonly).
- **has_children -** True if this element has at least one child (readonly).
- **last_attribute -** The last attribute of this element (readonly).
- **last_child -** The last child of this element (readonly).
- **level -** The zero-based depth of this element in the hierarchy (readonly).
- **name -** The name of this element (read / write).
- **parent -** The parent element of this element (readonly).
- **value -** The text content of this element (read / write).

Notes:

1. The **first_child** and **last_child** properties evaluate to **None** if that child does not exist.
2. The **name** and **value** properties are readonly for the root element.

Methods

- **add(*name, attr=* ' ', *value=None*) -** Create and add a new child element. This method returns the newly created child element.
    - ***name*** **-** The name of the element.
    - ***attr*** **-** The name of the optional attribute.
    - ***value*** **-** The value of the optional attribute.
- **add_comment(*text_field*) -** Create and add a comment to the element.
    - ***text_field*** **-** The text field of the comment.
- **add_element(*element*) -** Add an existing element as a child element. This method returns the added child element
    - ***element*** **-** The existing element.
- **clone(*deep=True*) -** Create and return a copy of this element.
    - ***deep*** **-** if True, include all the descendant elements (default = True)

- **find_child(*name*) -** Find the first child of this element with the specified name. This method returns the first matching child element if found, otherwise **None.**
    - ***name* -** The specified name of the child element.
- **find_descendants(*name*=' * ') -** Find all the descendants of this element with the specified name. This method returns a list of all descendant elements with the specified name.
    - ***name* -** The specified name (include all descendants if the name is **' * '**).
- **insert_element(*ref_element, new_element*) -** Insert a new child element before the referenced child element. This method returns the newly inserted child element.
    - ***ref_element* -** The existing referenced child.
    - ***new_element* -** The element to be inserted.
- **read_attribute(*name*) -** Read the value of the named attribute. This method returns the value of the attribute if found, otherwise an empty string.
    - ***name* -** The name of the attribute.
- **read_child(*name, default=None*) -** Read the value of the named child element. The default parameter value type determines the return value type. When the default parameter is **None,** the return value type is a string. This method returns the value of the child if found, otherwise the default value.
    - ***name* -** The name of the child element.
    - ***default* -** The default return value (**None** equates to an empty string).
- **remove(*name*) -** Remove the named child element. This method returns the removed child element if successful, otherwise **None.**
    - ***name* -** The name of the child element.
- **remove_all() -** Remove all children and comments from this element.
- **remove_all_attributes() -** Remove all attributes from this element.
- **remove_attribute(*name*) -** Remove the named attribute from this element.
    - ***name* -** The name of the attribute.
- **remove_element(*element*) -** Remove the specified child element. This method returns the removed child element if successful, otherwise **None.**
    - ***element* -** The specified child element.

- **replace_element(*old_element, new_element*) -** Replace an existing child element with a new child element. This method returns the new child element.
    - ***old_element* -** The existing child element.
    - ***new_element* -** The replacement child element.
- **write_attribute(*name, value*) -** Write a new value to the named attribute. If the named attribute does not exist, a new attribute is created and added to the element.
    - ***name* -** The name of the attribute.
    - ***value* -** The new value of the attribute.
- **write_child(*name, value*) -** Write a new value to the named child element. If the named child element does not exist, a new child element is created and added to the element.
    - ***name* -** The name of the child element.
    - ***value* -** The new value of the named child element.

XDocument Usage Example

XML document files are often used by application programs to record their configuration information, and application programs generally use dataclasses to manage their configuration information internally. These programs need the ability to:

1. Record the configuration information contained in a dataclass to an XML document file.
2. Retrieve the configuration information contained in the XML document file back to the dataclass.

This example demonstrates how to create a python module which can provide that capability to a program. The following descriptions refer to the python module, **config_data.py**, which is listed on the next page.

The **ConfigData** class is derived from the **XDocument** class, and it is a custom XML document class for working with dataclasses. Specifically, it provides the functionality needed to record and retrieve the attribute values of a dataclass.

The **ConfigData(*filename, dataclass*)** class constructor has two arguments. The first argument of the constructor is the name of the XML document file, and the second argument is a reference to a dataclass. The dataclass reference provides the initialization details for the **ConfigData** class. Those details are used by the **_subclass_details( )** method to record the default dataclass attribute values in the newly created the XML document file.

The **read(*dataclass*)** method retrieves the configuration information from the XML document and updates the attribute values of the referenced dataclass.

The **write(*dataclass*)** method records the configuration information by writing the attribute values of the referenced dataclass to the XML document. This method does not save the updated XML document to the associated XML document file.

```python
from dataclasses import asdict
from xdocument import XDocument


class ConfigData(XDocument):
    """An XML document class for dataclass attribute values."""

    def __init__(self, filename, dataclass):
        """Load an existing file or create a new XML document file."""
        super().__init__(filename, details=dataclass)

    def _subclass_details(self) -> None:
        """Provide the initialization details for the derived subclass."""
        if self._details is not None:
            self.write(self._details)  # The default dataclass attribute values

    def read(self, dataclass):
        """Read the dataclass attribute values from the XML document."""
        element = self.root.find_child(class_name(dataclass))
        if element is not None:  # The XML document may be blank or corrupted
            for name, value in asdict(dataclass).items():
                attribute_value = element.read_child(pascal_case(name), value)
                setattr(dataclass, name, attribute_value)
        return dataclass

    def write(self, dataclass):
        """Write the dataclass attribute values to the XML document."""
        element = self.root.find_child(class_name(dataclass))
        if element is None:  # The XML document may be blank or corrupted
            element = self.root.add(class_name(dataclass))
        for name, value in asdict(dataclass).items():
            element.write_child(pascal_case(name), value)


def class_name(dataclass):
    """Return the name of the dataclass."""
    return type(dataclass).__name__


def pascal_case(attribute_name):
    """Convert the attribute name to a Pascal Case name."""
    return attribute_name.replace('_', ' ').title().replace(' ', '')
```

The **class_name(*dataclass*)** function provides the name of the referenced dataclass.

Use of the **pascal_case(*attribute_name*)** function allows for a consistent naming style throughout the XML document.

The following script uses the **ConfigData** class for exchanging configuration information between a dataclass and an XML document file:

```
from dataclasses import dataclass, asdict
from config_data import ConfigData, class_name


@dataclass
class Parameters:
    """An example of configuration parameters."""

    min_threshold: int = 25
    calibration: float = 0.035
    display_results: bool = True
    units: str = 'millimeters'


if __name__ == '__main__':
    document = ConfigData('Config.xml', Parameters())
    parameters = document.read(Parameters())
    print('\n' + class_name(parameters))
    for name, value in asdict(parameters).items():
        print(f'   {name} = {value}')
```

When the script is executed for the first time, the default attribute values in the **Parameters** dataclass are recorded as the configuration information in the "Config.xml" file. The text content of this file is:

```
<?xml version="1.0" encoding="utf-8"?>
<ConfigData>
  <Parameters>
    <MinThreshold>25</MinThreshold>
    <Calibration>0.035</Calibration>
    <DisplayResults>True</DisplayResults>
    <Units>millimeters</Units>
  </Parameters>
</ConfigData>
```

When the script is run again, the script will retrieve the configuration information from the "Config.xml" file and update the attribute values of the dataclass. The attribute names and values are always printed by the script.

The reader is again encouraged to make changes to the "Config.xml" file by editing the text values of the configuration attribute elements. Care should be taken to ensure that the data type represented by an edited text value matches the defined data type of that attribute. The reader should save these changes to the "Config.xml" file and then run the script again to observe the results.