The last part of the code prints out the prices of properties for the top 10 rows. We use list comprehension to create a list of the prices. You can find the first occurrence of a certain item in a list by calling `.index(...)` on a list object, as we did in this example.

See also

Check the `pandas` documentation for `read_excel` at `http://pandas.pydata.org/pandas-docs/stable/io.html#io-excel`. Also, you can visit `http://www.python-excel.org` for a list of modules that allow you to work with data stored in different Excel formats, both older `.xls` and newer `.xlsx` files.

Reading and writing XML files with Python

XML stands for eXtensible Markup Language. Although not as popular to store data as the formats described previously, certain web APIs return XML-encoded information on request.

An XML-encoded document has a tree-like structure. To read the contents, we start at the root of the tree (normally, the name of the element that follows the XML declaration `<?xml version="1.0" encoding="UTF-8"?>`; every XML-encoded document needs to begin with such declaration). In our case, the root of our XML-encoded document is `<records>`. A single `<record>...</record>` contains a list of `<var var_name=...>...</var>`.

> Warning: The `xml` module is not secure. Caution is required when dealing with XML-encoded messages from untrusted sources. An attacker might access local files, carry out DoS attacks, and more. Refer to the documentation for the `xml` module at `https://docs.python.org/3/library/xml.html`.

Getting ready

In order to execute the following recipe, you need the `pandas` and `xml` modules available. No other prerequisites are required.

How to do it...

Reading the data from an XML file directly to a `pandas` DataFrame requires some supplementary code; this is because each XML file has a different structure and requires a made-to-fit parsing. We will define the innards of the methods defined in the following section of this recipe. The source code for this section can be found in the `read_xml.py` file:

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

```
def read_xml(xml_tree):
    '''
        Read an XML encoded data and return pd.DataFrame
    '''

def iter_records(records):
    '''
        Generator to iterate through all the records
    '''

def write_xml(xmlFileName, data):
    '''
        Save the data in an XML format
    '''

def xml_encode(row):
    '''
        Encode the row as an XML with a specific hierarchy
    '''

# names of files to read from and write to
r_filenameXML = '../../Data/Chapter01/realEstate_trans.xml'
w_filenameXML = '../../Data/Chapter01/realEstate_trans.xml'

# read the data
xml_read = read_xml(r_filenameXML)

# print the first 10 records
print(xml_read.head(10))

# write back to the file in an XML format
write_xml(w_filenameXML, xml_read)
```

How it works...

Let's analyze the preceding code step by step. First, we import all the modules that we need. The `xml.etree.ElementTree` module is a lightweight XML parser of the XML tree and we will use it to parse the XML structure of our file. As before, we define names of the files to read and write in separate variables (`r_filenameXML`, `w_filenameXML`).

To read the data from the XML-encoded file, we use the `read_xml(...)` method:

```
def read_xml(xmlFileName):
    with open(xmlFileName, 'r') as xml_file:
        # read the data and store it as a tree
```

```
            tree = ET.parse(xml_file)

            # get the root of the tree
            root = tree.getroot()

            # return the DataFrame
            return pd.DataFrame(list(iter_records(root)))
```

The method takes the name of the file as its only parameter. First, the file is opened. Using the `.parse(...)` method, we create a tree-like structure from our XML-encoded file and store it in the `tree` object. We then extract the root using the `.getroot()` method on the `tree` object: this is the starting point to process the data further. The return statement calls the `iter_records` method passing the reference to the root of the tree and then converts the returned information to a DataFrame:

```
    def iter_records(records):
        for record in records:
            # temporary dictionary to hold values
            temp_dict = {}

            # iterate through all the fields
            for var in record:
                temp_dict[
                    var.attrib['var_name']
                ] = var.text

            # generate the value
            yield temp_dict
```

The `iter_records` method is a *generator*: a method that, as the name suggests, generates the values. Unlike regular methods that have to return all the values when the function finishes (a `return` statement), generators hand over the data back to the calling method one at a time (hence the `yield` keyword) until done.

For a more in-depth discussion on generators, I suggest reading `https://www.jeffknupp.com/blog/2013/04/07/improve-your-python-yield-and-generators-explained/`.

Our `iter_records` method, for each record read, emits a `temp_dict` dictionary object back to the `read_xml` method. Each element of the dictionary has a key equal to the `var_name` attribute of the `<var>` XML element. (Our `<var>` has the following format: `<var var_name=...>`.)

[The `<var>` tag could have more attributes with other names—these would be stored in the `.attrib` dictionary (a property of the XML tree node) and would be accessible by their names—see the highlighted line in the previous source code.]

The value of `<var>` (contained within `<var>...</var>`) is accessible through the `.text` property of the XML node, while the `.tag` property stores its name (in our case, `var`).

The `return` statement of the `read_xml` method creates a list from all the dictionaries passed, which is then turned into a DataFrame.

To write the data in an XML format, we use the `write_xml(...)` method:

```
def write_xml(xmlFileName, data):
    with open(xmlFileName, 'w') as xmlFile:

        # write the headers
        xmlFile.write(
            '<?xml version="1.0" encoding="UTF-8"?>\n'
        )
        xmlFile.write('<records>\n')

        # write the data
        xmlFile.write(
            '\n'.join(data.apply(xml_encode, axis=1))
        )

        # write the footer
        xmlFile.write('\n</records>')
```

The method opens the file specified by the `xmlFileName` parameter. Every XML file needs to start with the XML declaration (see the introduction to this recipe) in the first line. Then, we write out the root of our XML schema, `<records>`.

Next, it is time to write out the data. We use the `.apply(...)` method of the DataFrame object to iterate through the records contained within. Its first parameter specifies the method to be applied to each record. By default, the `axis` parameter is set to `0`. This means that the method specified in the first parameter would be applied to each `column` of the DataFrame. By setting the parameter to `1`, we instruct the `.apply(...)` method that we want to apply the `xml_encode(...)` method specified in the first parameter to each `row`. We use the `xml_encode(...)` method to process each record from the `data` DataFrame:

```
def xml_encode(row):
    # first -- we output a record
    xmlItem = ['  <record>']
```

```
        # next -- for each field in the row we create a XML markup
        #         in a <field name=...>...</field> format
        for field in row.index:
            xmlItem \
                .append(
                    '    <var var_name="{0}">{1}</var>' \
                    .format(field, row[field])
                )

        # last -- this marks the end of the record
        xmlItem.append('  </record>')

        # return a string back to the calling method
        return '\n'.join(xmlItem)
```

The code creates a list of strings, `xmlItem`. The first element of the list is the `<record>` indicator and the last one will be `</record>`. Then, for each field in the row, we append values of each column for that record encapsulated within `<var var_name=`**<column_name>**`>`**<value>**`</var>`. The variables in bold indicate specific column names from the record (`<column_name>`) and corresponding value (`<value>`). Once all the fields of the record have been parsed, we create a long string by concatenating all the items of the `xmlItem` list using the `'\n'.join(...)` method. Each `<var>...</var>` tag is then separated by `\n`. The string is returned to the caller (`write_xml`). Each record is further concatenated in the `write_xml(...)` method and then output to the file. We finish with the closing tag, `</records>`.

Retrieving HTML pages with pandas

Although not as popular to store large datasets as previous formats, sometimes we find data in a table on a web page. These structures are normally enclosed within the `<table> </table>` HTML tags. This recipe will show you how to retrieve data from a web page.

Getting ready

In order to execute the following recipe, you need `pandas` and `re` modules available. The `re` module is a regular expressions module for Python and we will use it to clean up the column names. Also, the `read_html(...)` method of `pandas` requires `html5lib` to be present on your computer. If you use the Anaconda distribution of Python, you can do it by issuing the following command from your command line:

```
conda install html5lib
```