Piculet Documentation

Release 1.0

H. Turgut Uyar

May 30, 2018

Contents

1 Contents **3**
- 1.1 Overview . . . 3
- 1.2 Data extraction . . . 5
- 1.3 Preprocessing . . . 14
- 1.4 Lower-level functions . . . 14
- 1.5 API . . . 16
- 1.6 History . . . 22

2 Indices and Tables **25**

Python Module Index **27**

Piculet is a module for extracting data from XML or HTML documents using XPath queries. It consists of a single source file with no dependencies other than the standard library, which makes it very easy to integrate into applications. It also provides a command line interface.

PyPI https://pypi.org/project/piculet/

Repository https://bitbucket.org/uyar/piculet

Documentation https://piculet.readthedocs.io/

Piculet has been tested with Python 2.7, Python 3.4+, PyPy2 5.7+, and PyPy3 5.7+. You can install the latest version using `pip`:

```
pip install piculet
```

CHAPTER 1

Contents

1.1 Overview

Scraping a document consists of three stages:

1. Building a DOM tree out of the document. This is a straightforward operation for an XML document. For an HTML document, Piculet will first try to convert it into XHTML and then build the tree from that.
2. Preprocessing the tree. This is an optional stage. In some cases it might be helpful to do some changes on the tree to simplify the extraction process.
3. Extracting data out of the tree.

The preprocessing and extraction stages are expressed as part of a scraping specification. The specification is a mapping which can be stored in a file format that can represent a mapping, such as JSON or YAML. Details about the specification are given in later chapters.

1.1.1 Command Line Interface

Installing Piculet creates a script named `piculet` which can be used to invoke the command line interface:

```
$ piculet -h
usage: piculet [-h] [--debug] command ...
```

The `scrape` command extracts data out of a document as described by a specification file:

```
$ piculet scrape -h
usage: piculet scrape [-h] -s SPEC [--html] document
```

The location of the document can be given as a file path or a URL. For example, say you want to extract some data from the file shining.html. An example specification is given in movie.json. Download both of these files and run the command:

```
$ piculet scrape -s movie.json shining.html
```

This should print the following output:

```
{
  "cast": [
    {
      "character": "Jack Torrance",
      "link": "/people/2",
      "name": "Jack Nicholson"
    },
    {
      "character": "Wendy Torrance",
      "link": "/people/3",
      "name": "Shelley Duvall"
    }
  ],
  "director": {
    "link": "/people/1",
    "name": "Stanley Kubrick"
  },
  "genres": [
    "Horror",
    "Drama"
  ],
  "language": "English",
  "review": "Fantastic movie. Definitely recommended.",
  "runtime": "144 minutes",
  "title": "The Shining",
  "year": 1980
}
```

For HTML documents, the `--html` option has to be used. If the document address starts with `http://` or `https://`, the content will be taken from the given URL. For example, to extract some data from the Wikipedia page for David Bowie, download the wikipedia.json file and run the command:

```
piculet scrape -s wikipedia.json --html "https://en.wikipedia.org/wiki/David_Bowie"
```

This should print the following output:

```
{
  "birthplace": "Brixton, London, England",
  "born": "1947-01-08",
  "died": "2016-01-10",
  "name": "David Bowie",
  "occupation": [
    "Singer",
    "songwriter",
    "actor"
  ]
}
```

In the same command, change the name part of the URL to `Merlene_Ottey` and you will get similar data for Merlene Ottey. Note that since the markup used in Wikipedia pages for persons varies, the kinds of data you get with this specification will also vary.

Piculet can be used as an HTML to XHTML convertor by invoking it with the `h2x` command. This command takes the file name as input and prints the converted content, as in `piculet h2x foo.html`. If the input file name is given as `-` it will read the content from the standard input and therefore can be used as part of a pipe: `cat foo.html | piculet h2x -`

1.1.2 Using in programs

The scraping operation can also be invoked programmatically using the *`scrape_document`* function:

```
from piculet import scrape_document

url = 'https://en.wikipedia.org/wiki/David_Bowie'
spec = 'wikipedia.json'
data = scrape_document(url, spec, content_format='html')
```

1.1.3 YAML support

To use YAML for specification, Piculet has to be installed with YAML support:

```
pip install piculet[yaml]
```

Note that this will install an external module for parsing YAML files, and therefore will not be contained to the standard library anymore.

The YAML version of the configuration example above can be found in movie.yaml.

1.2 Data extraction

This section explains how to write the specification for extracting data from a document. We'll scrape the following HTML content for the movie "The Shining" in our examples:

```
<html>
    <head>
        <meta charset="utf-8" />
        <title>The Shining</title>
    </head>
    <body>
        <h1>The Shining (<span class="year">1980</span>)</h1>
        <ul class="genres">
            <li>Horror</li>
            <li>Drama</li>
        </ul>
        <div class="director">
            <h3>Director:</h3>
            <p><a href="/people/1">Stanley Kubrick</a></p>
        </div>
        <table class="cast">
            <tr>
                <td><a href="/people/2">Jack Nicholson</a></td>
                <td>Jack Torrance</td>
            </tr>
            <tr>
                <td><a href="/people/3">Shelley Duvall</a></td>
                <td>Wendy Torrance</td>
            </tr>
        </table>
        <div class="info">
            <h3>Runtime:</h3>
            <p>144 minutes</p>
        </div>
        <div class="info">
            <h3>Language:</h3>
            <p>English</p>
        </div>
        <div class="review">
            <em>Fantastic</em> movie.
            Definitely recommended.
```

(continues on next page)

(continued from previous page)

```
        </div>
    </body>
</html>
```

Instead of the *scrape_document* function that reads the content and the specification from files, we'll use the *scrape* function that works directly on the content and the specification map:

```
>>> from piculet import scrape
```

Assuming the HTML document above is saved as `shining.html`, let's get its content:

```
>>> with open('shining.html') as f:
...     document = f.read()
```

The *scrape* function assumes that the document is in XML format. So if any conversion is needed, it has to be done before calling this function.[1] After building the DOM tree, the function will apply the extraction rules to the root element of the tree, and return a mapping where each item is generated by one of the rules.

The specification mapping contains two keys: the `pre` key is for specifying the preprocessing operations (these will be covered in the next section), and the `items` key is for specifying the rules that describe how to extract the data:

```
spec = {'pre': [...], 'items': [...]}
```

The items list contains item mappings, where each item has a `key` and a `value` description. The key specifies the key for the item in the output mapping and the value specifies how to extract the data to set as the value for that item. Typically, a value specifier consists of a path query and a reducing function. The query is applied to the root and a list of strings is obtained. Then, the reducing function converts this list into a single string.[2]

For example, to get the title of the movie from the example document, we can write:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'title',
...             'value': {
...                 'path': '//title/text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'title': 'The Shining'}
```

The `.//title/text()` path generates the list `['The Shining']` and the reducing function `first` selects the first element from that list.

Note: Piculet uses the ElementTree module for building and querying XML trees. Therefore, the XPath queries are limited by what ElementTree supports (plus the `text()` and `@attr` clauses which are added by Piculet). However, Piculet will make use of the lxml package if it's installed, and in that case, a much wider range of XPath constructs can be used.

Multiple items can be collected in a single invocation:

[1] Note that the example document is already in XML format.

[2] This means that the query has to end with either `text()` or some attribute value as in `@attr`. And the reducing function should be implemented so that it takes a list of strings and returns a string.

```
>>> spec = {
...     'items': [
...         {
...             'key': 'title',
...             'value': {
...                 'path': '//title/text()',
...                 'reduce': 'first'
...             }
...         },
...         {
...             'key': 'year',
...             'value': {
...                 'path': '//span[@class="year"]/text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'title': 'The Shining', 'year': '1980'}
```

If a path doesn't match any element in the tree, the item will be excluded from the output. Note that in the following example, the "foo" key doesn't get included:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'title',
...             'value': {
...                 'path': '//title/text()',
...                 'reduce': 'first'
...              }
...         },
...         {
...             'key': "foo",
...             'value': {
...                 'path': '//foo/text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'title': 'The Shining'}
```

1.2.1 Reducing

Piculet contains a few predefined reducing functions. Other than the `first` reducer used in the examples above, a very common reducer is `concat` which will concatenate the selected strings:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'full_title',
...             'value': {
...                 'path': '//h1//text()',
...                 'reduce': 'concat'
...             }
...         }
...     ]
```

(continues on next page)

(continued from previous page)

```
... }
>>> scrape(document, spec)
{'full_title': 'The Shining (1980)'}
```

`concat` is the default reducer, i.e. if no reducer is given, the strings will be concatenated:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'full_title',
...             'value': {
...                 'path': '//h1//text()'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'full_title': 'The Shining (1980)'}
```

If you want to get rid of extra whitespace, you can use the `clean` reducer. After concatenating the strings, this will remove leading and trailing whitespace and replace multiple whitespace with a single space:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'review',
...             'value': {
...                 'path': '//div[@class="review"]//text()',
...                 'reduce': 'clean'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'review': 'Fantastic movie. Definitely recommended.'}
```

In this example, the `concat` reducer would have produced the value `'\n Fantastic movie.\n Definitely recommended.\n '`

As explained above, if a path query doesn't match any element, the item gets automatically excluded. That means, Piculet doesn't try to apply the reducing function on the result of the path query if it's an empty list. Therefore, reducing functions can safely assume that the path result is a non-empty list.

If you want to use a custom reducer, you have to register it first. The name for the specifier (the first parameter) has to be a valid Python identifier.

```
>>> from piculet import reducers
>>> reducers.register('second', lambda x: x[1])
>>> spec = {
...     'items': [
...         {
...             'key': 'year',
...             'value': {
...                 'path': '//h1//text()',
...                 'reduce': 'second'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'year': '1980'}
```

1.2.2 Transforming

After the reduction operation, you can apply a transformation to the resulting string. A transformation function must take a string as parameter and can return any value of any type. Piculet contains several predefined transformers: `int`, `float`, `bool`, `len`, `lower`, `upper`, `capitalize`. For example, to get the year of the movie as an integer:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'year',
...             'value': {
...                 'path': '//span[@class="year"]/text()',
...                 'reduce': 'first',
...                 'transform': 'int'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'year': 1980}
```

If you want to use a custom transformer, you have to register it first:

```
>>> from piculet import transformers
>>> transformers.register('year25', lambda x: int(x) + 25)
>>> spec = {
...     'items': [
...         {
...             'key': 'year',
...             'value': {
...                 'path': '//span[@class="year"]/text()',
...                 'reduce': 'first',
...                 'transform': 'year25'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'25th_year': 2005}
```

1.2.3 Multi-valued items

Data with multiple values can be created by using a `foreach` key in the value specifier. This is a path expression to select elements from the tree.[3] The path and reducing function will be applied *to each selected element* and the obtained values will be the members of the resulting list. For example, to get the genres of the movie, we can write:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'genres',
...             'value': {
...                 'foreach': '//ul[@class="genres"]/li',
...                 'path': './text()',
...                 'reduce': 'first'
...             }
...         }
```

(continues on next page)

[3] This implies that the `foreach` query should **not** end in `text()` or `@attr`.

(continued from previous page)

```
...     ]
... }
>>> scrape(document, spec)
{'genres': ['Horror', 'Drama']}
```

If the `foreach` key doesn't match any element the item will be excluded from the result:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'foos',
...             'value': {
...                 'foreach': '//ul[@class="foos"]/li',
...                 'path': './text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{}
```

If a transformation is specified, it will be applied to every element in the resulting list:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'genres',
...             'value': {
...                 'foreach': '//ul[@class="genres"]/li',
...                 'path': './text()',
...                 'reduce': 'first',
...                 'transform': 'lower'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'genres': ['horror', 'drama']}
```

1.2.4 Subrules

Nested structures can be created by writing subrules as value specifiers. If the value specifier is a mapping that contains an `items` key, then this will be interpreted as a subrule and the generated mapping will be the value for the key.

```
>>> spec = {
...     'items': [
...         {
...             'key': 'director',
...             'value': {
...                 'items': [
...                     {
...                         'key': 'name',
...                         'value': {
...                             'path': '//div[@class="director"]//a/text()',
...                             'reduce': 'first'
...                         }
...                     },
```

(continues on next page)

(continued from previous page)

```
...                         {
...                             'key': 'link',
...                             'value': {
...                                 'path': '//div[@class="director"]//a/@href',
...                                 'reduce': 'first'
...                             }
...                         }
...                     ]
...                 }
...             }
...         ]
... }
>>> scrape(document, spec)
{'director': {'link': '/people/1', 'name': 'Stanley Kubrick'}}
```

Subrules can be combined with lists:

```
>>> spec = {
...     'items': [
...         {
...             'key': 'cast',
...             'value': {
...                 'foreach': '//table[@class="cast"]/tr',
...                 'items': [
...                     {
...                         'key': 'name',
...                         'value': {
...                             'path': './td[1]/a/text()',
...                             'reduce': 'first'
...                         }
...                     },
...                     {
...                         'key': 'link',
...                         'value': {
...                             'path': './td[1]/a/@href',
...                             'reduce': 'first'
...                         }
...                     },
...                     {
...                         'key': 'character',
...                         'value': {
...                             'path': './td[2]/text()',
...                             'reduce': 'first'
...                         }
...                     }
...                 ]
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'cast': [{'character': 'Jack Torrance',
   'link': '/people/2',
   'name': 'Jack Nicholson'},
  {'character': 'Wendy Torrance',
   'link': '/people/3',
   'name': 'Shelley Duvall'}]}
```

Items generated by subrules can also be transformed. The transformation function is always applied as the last step in a "value" definition. But transformers for subitems take mappings (as opposed to strings) as parameter.

```
>>> transformers.register('stars', lambda x: '%(name)s as %(character)s' % x)
>>> spec = {
...     'items': [
...         {
...             'key': 'cast',
...             'value': {
...                 'foreach': '//table[@class="cast"]/tr',
...                 'items': [
...                     {
...                         'key': 'name',
...                         'value': {
...                             'path': './td[1]/a/text()',
...                             'reduce': 'first'
...                         }
...                     },
...                     {
...                         'key': 'character',
...                         'value': {
...                             'path': './td[2]/text()',
...                             'reduce': 'first'
...                         }
...                     }
...                 ],
...                 'transform': 'stars'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'cast': ['Jack Nicholson as Jack Torrance',
  'Shelley Duvall as Wendy Torrance']}
```

1.2.5 Generating keys from content

You can generate items where the key value also comes from the content. For example, consider how you would get the runtime and the language of the movie. Instead of writing multiple items for each `h3` element under an "info" class `div`, we can write only one item that will select these divs and use the h3 text as the key. These elements can be selected using `foreach` specifications in the items. This will cause a new item to be generated for each selected element. To get the key value, we can use paths, reducers -and also transformers- that will be applied to the selected element:

```
>>> spec = {
...     'items': [
...         {
...             'foreach': '//div[@class="info"]',
...             'key': {
...                 'path': './h3/text()',
...                 'reduce': 'first'
...             },
...             'value': {
...                 'path': './p/text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'Language:': 'English', 'Runtime:': '144 minutes'}
```

The `normalize` reducer concatenates the strings, converts it to lowercase, replaces spaces with underscores and

strips other non-alphanumeric characters:

```
>>> spec = {
...     'items': [
...         {
...             'foreach': '//div[@class="info"]',
...             'key': {
...                 'path': './h3/text()',
...                 'reduce': 'normalize'
...             },
...             'value': {
...                 'path': './p/text()',
...                 'reduce': 'first'
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'language': 'English', 'runtime': '144 minutes'}
```

You could also give a string instead of a path and reducer for the key. In this case, the elements would still be traversed; only the last one would set the final value for the item. This could be OK if you are sure that there is only one element that matches the `foreach` path of the key.

1.2.6 Sections

The specification also provides the ability to define sections within the document. An element can be selected as the root of a section such that the XPath queries in that section will be relative to that root. This can be used to make XPath expressions shorter and also constrain the search in the tree. For example, the "director" example above can also be written using sections:

```
>>> spec = {
...     'section': '//div[@class="director"]//a',
...     'items': [
...         {
...             'key': 'director',
...             'value': {
...                 'items': [
...                     {
...                         'key': 'name',
...                         'value': {
...                             'path': './text()',
...                             'reduce': 'first'
...                         }
...                     },
...                     {
...                         'key': 'link',
...                         'value': {
...                             'path': './@href',
...                             'reduce': 'first'
...                         }
...                     }
...                 ]
...             }
...         }
...     ]
... }
>>> scrape(document, spec)
{'director': {'link': '/people/1', 'name': 'Stanley Kubrick'}}
```

1.3 Preprocessing

Other than extraction rules, specifications can also contain preprocessing operations which allow modifications on the tree before starting data extraction. Such operations can be needed to make data extraction simpler or to remove the need for some postprocessing operations on the collected data.

The syntax for writing preprocessing operations is as follows:

```
rules = {
    'pre': [
        {
            'op': '...',
            ...
        },
        {
            'op': '...',
            ...
        }
    ],
    'items': [ ... ]
}
```

Every preprocessing operation item has a name which is given as the value of the "op" key. The other items in the mapping are specific to the operation. The operations are applied in the order as they are written in the operations list.

The predefined preprocessing operations are explained below.

1.3.1 Removing elements

This operation removes from the tree all the elements (and its subtree) that are selected by a given XPath query:

```
{'op': 'remove', 'path': '...'}
```

1.3.2 Setting element attributes

This operation selects all elements by a given XPath query and sets an attribute for these elements to a given value:

```
{'op': 'remove', 'path': '...', 'name': '...', 'value': '...'}
```

The attribute "name" can be a literal string or an extractor as described in the data extraction chapter. Similarly, the attribute "value" can be given as a literal string or an extractor.

1.3.3 Setting element text

This operation selects all elements by a given XPath query and sets their texts to a given value:

```
{'op': 'remove', 'path': '...', 'text': '...'}
```

The "text" can be a literal string or an extractor.

1.4 Lower-level functions

Piculet also provides a lower-level API where you can run the stages separately. For example, if the same document will be scraped multiple times with different rules, calling the `scrape` function repeatedly will cause the

document to be parsed into a DOM tree repeatedly. Instead, you can create the DOM tree once and run extraction rules against this tree multiple times.

Also, this API uses classes to express the specification and therefore development tools can help better in writing the rules by showing error indicators and suggesting autocompletions.

1.4.1 Building the tree

The DOM tree can be created from the document using the *build_tree* function:

```
>>> from piculet import build_tree
>>> root = build_tree(document)
```

If the document needs to be converted from HTML to XML, you can use the *html_to_xhtml* function:

```
>>> from piculet import html_to_xhtml
>>> converted = html_to_xhtml(document)
>>> root = build_tree(converted)
```

1.4.2 Preprocessing

The tree can be modified using the *preprocess* function:

```
>>> from piculet import preprocess
>>> ops = [{'op': 'remove', 'path': '//div'}]
>>> preprocess(root, ops)
```

1.4.3 Data extraction

The class-based API to data extraction has a one-to-one correspondance with the specification mapping. A *Rule* object corresponds to a key-value pair in the items list. Its value is produced by an `extractor`. In the simple case, an extractor is a *Path* object which is a combination of a path, a reducer, and a transformer.

```
>>> from piculet import Path, Rule
>>> extractor = Path('//span[@class="year"]/text()',
...                  reduce=reducers.first, transform=int)
>>> rule = Rule(key='year', extractor=extractor)
>>> rule.extract(root)
{'year': 1980}
```

An extractor can have a `foreach` attribute if it will be multi-valued:

```
>>> extractor = Path(foreach='//ul[@class="genres"]/li',
...                  path='./text()', reduce=reducers.first,
...                  transform=str.lower)
>>> rule = Rule(key='genres', extractor=extractor)
>>> rule.extract(root)
{'genres': ['horror', 'drama']}
```

The `key` attribute of a rule can be an extractor in which case it can be used to extract the key value from content. A rule can also have a `foreach` attribute for generating multiple items in one rule. These features will work as they are described in the data extraction section.

A *Rules* object contains a collection of rule objects and it corresponds to the "items" part in the specification mapping. It acts both as the top level extractor that gets applied to the root of the tree, and also as an extractor for any rule with subrules.

```
>>> from piculet import Rules
>>> rules = [Rule(key='title',
...               extractor=Path('//title/text()')),
...          Rule('year',
...               extractor=Path('//span[@class="year"]/text()', transform=int))]
>>> Rules(rules).extract(root)
{'title': 'The Shining', 'year': 1980}
```

A more complete example with transformations is below. Again note that, the specification is exactly the same as given in the corresponding mapping example in the data extraction chapter.

```
>>> rules = [
...     Rule(key='cast',
...          extractor=Rules(
...              foreach='//table[@class="cast"]/tr',
...              rules=[
...                  Rule(key='name',
...                       extractor=Path('./td[1]/a/text()')),
...                  Rule(key='character',
...                       extractor=Path('./td[2]/text()'))
...              ],
...              transform=lambda x: '%(name)s as %(character)s' % x
...          ))
... ]
>>> Rules(rules).extract(root)
{'cast': ['Jack Nicholson as Jack Torrance',
  'Shelley Duvall as Wendy Torrance']}
```

A rules object can have a `section` attribute as described in the data extraction chapter:

```
>>> rules = [
...     Rule(key='director',
...          extractor=Rules(
...              section='//div[@class="director"]//a',
...              rules=[
...                  Rule(key='name',
...                       extractor=Path('./text()')),
...                  Rule(key='link',
...                       extractor=Path('./@href'))
...              ]))
... ]
>>> Rules(rules).extract(root)
{'director': {'link': '/people/1', 'name': 'Stanley Kubrick'}}
```

1.5 API

Piculet is a module for scraping XML and HTML documents using XPath queries.

It consists of this single source file with no dependencies other than the standard library, which makes it very easy to integrate into applications. It has been tested with Python 2.7, Python 3.4+, PyPy2 5.7+, and PyPy3 5.7+.

For more information, please refer to the documentation: https://piculet.readthedocs.io/

class piculet.**Extractor**(*transform=None*, *foreach=None*)

Bases: object

Abstract base extractor for getting data out of an XML element.

Parameters

- **transform** (*Optional[Transformer]*) – Function to transform the extracted value.

- **foreach** (*Optional[str]*) – Path to apply for generating a collection of values.

apply(*element*)
Get the raw data from an element using this extractor.

Parameters **element** – Element to apply this extractor to.

Return type ExtractedItem

Returns Extracted raw data.

extract(*element*, *transform=True*)
Get the processed data from an element using this extractor.

Parameters

- **element** – Element to extract the data from.
- **transform** – Whether the transformation will be applied or not.

Return type Any

Returns Extracted and optionally transformed data.

foreach = None
Path to apply for generating a collection of values.

static from_map(*item*)
Generate an extractor from a description map.

Parameters **item** (*Mapping[str, Any]*) – Extractor description.

Return type *Extractor*

Returns Extractor object.

Raises **ValueError** – When reducer or transformer names are unknown.

transform = None
Function to transform the extracted value.

class piculet.**HTMLNormalizer**(*omit_tags=None*, *omit_attrs=None*)
Bases: html.parser.HTMLParser

HTML cleaner and XHTML convertor.

DOCTYPE declarations and comments are removed.

Parameters

- **omit_tags** (*Optional[Iterable[str]]*) – Tags to remove, along with all their content.
- **omit_attrs** (*Optional[Iterable[str]]*) – Attributes to remove.

SELF_CLOSING_TAGS = {'input', 'link', 'img', 'hr', 'meta', 'br'}
Tags to handle as self-closing.

handle_charref(*name*)
Process a character reference.

handle_data(*data*)
Process collected character data.

handle_endtag(*tag*)
Process the ending of an element.

handle_entityref(*name*)
Process an entity reference.

handle_starttag(*tag*, *attrs*)
Process the starting of a new element.

class piculet.**Path**(*path*, *reduce=None*, *transform=None*, *foreach=None*)
Bases: *piculet.Extractor*

An extractor for getting text out of an XML element.

Parameters

- **path** (*str*) – Path to apply to get the data.
- **reduce** (*Optional[Reducer]*) – Function to reduce selected texts into a single string.
- **transform** (*Optional[PathTransformer]*) – Function to transform extracted value.
- **foreach** (*Optional[str]*) – Path to apply for generating a collection of data.

apply(*element*)
Apply this extractor to an element.

Parameters **element** – Element to apply this extractor to.

Return type str

Returns Extracted text.

path = None
XPath evaluator to apply to get the data.

reduce = None
Function to reduce selected texts into a single string.

class piculet.**Registry**(*entries*)
Bases: object

A simple, attribute-based namespace.

Parameters **entries** (*Mapping[str, Any]*) – Entries to add to this registry.

get(*item*)
Get the value of an entry from this registry.

Parameters **item** – Entry to get the value for.

Return type Any

Returns Value of entry.

register(*key*, *value*)
Register a new entry in this registry.

Parameters

- **key** – Key to search the entry in this registry.
- **value** – Value to store for the entry.

class piculet.**Rule**(*key*, *extractor*, *foreach=None*)
Bases: object

A rule describing how to get a data item out of an XML element.

Parameters

- **key** (*Union[str, Extractor]*) – Name to distinguish this data item.
- **extractor** (*Extractor*) – Extractor that will generate this data item.
- **foreach** (*Optional[str]*) – Path for generating multiple items.

extract(*element*)
Extract data out of an element using this rule.

Parameters **element** – Element to extract the data from.

Return type Mapping[str, Any]

Returns Extracted data.

extractor = None
Extractor that will generate this data item.

foreach = None
XPath evaluator for generating multiple items.

static from_map(*item*)
Generate a rule from a description map.

Parameters **item** (*Mapping[str, Any]*) – Item description.

Return type *Rule*

Returns Rule object.

key = None
Name to distinguish this data item.

class piculet.**Rules**(*rules*, *section=None*, *transform=None*, *foreach=None*)
Bases: *piculet.Extractor*

An extractor for getting data items out of an XML element.

Parameters

- **rules** (*Sequence[Rule]*) – Rules for generating the data items.
- **section** (*str*) – Path for setting the root of this section.
- **transform** (*Optional[MapTransformer]*) – Function to transform extracted value.
- **foreach** (*Optional[str]*) – Path for generating multiple items.

apply(*element*)
Apply this extractor to an element.

Parameters **element** – Element to apply the extractor to.

Return type Mapping[str, Any]

Returns Extracted mapping.

rules = None
Rules for generating the data items.

section = None
XPath expression for selecting a subroot for this section.

class piculet.**XPath**(*path*)
Bases: object

An XPath expression evaluator.

This class is mainly needed to compensate for the lack of `text()` and `@attr` axis queries in ElementTree XPath support.

Parameters **path** (*str*) – XPath expression to evaluate.

piculet.**build_tree**(*document*, *force_html=False*)
Build a tree from an XML document.

Parameters

- **document** (*str*) – XML document to build the tree from.
- **force_html** (*Optional[bool]*) – Force to parse from HTML without converting.

Return type Element

Returns Root element of the XML tree.

piculet.**decode_html**(*content*, *charset=None*, *fallback_charset='utf-8'*)
Decode an HTML document according to a character set.

If no character set is given, this will try to figure it out from the corresponding `meta` tags.

Parameters

- **content** (*bytes*) – Content of HTML document to decode.
- **charset** (*Optional[str]*) – Character set of the page.
- **fallback_charset** (*Optional[str]*) – Character set to use if it can't be figured out.

Return type str

Returns Decoded content of the document.

piculet.**extract**(*element*, *items*, *section=None*)
Extract data from an XML element.

Parameters

- **element** (*Element*) – Element to extract the data from.
- **items** (*Sequence[Mapping[str, Any]]*) – Descriptions for extracting items.
- **section** (*Optional[str]*) – Path to select the root element for these items.

Return type Mapping[str, Any]

Returns Extracted data.

piculet.**h2x**(*source*)
Convert an HTML file into XHTML and print.

Parameters **source** (*str*) – Path of HTML file to convert.

piculet.**html_to_xhtml**(*document*, *omit_tags=None*, *omit_attrs=None*)
Clean HTML and convert to XHTML.

Parameters

- **document** (*str*) – HTML document to clean and convert.
- **omit_tags** (*Optional[Iterable[str]]*) – Tags to exclude from the output.
- **omit_attrs** (*Optional[Iterable[str]]*) – Attributes to exclude from the output.

Return type str

Returns Normalized XHTML content.

piculet.**main**(*argv=None*)
Entry point of the command line utility.

Parameters **argv** (*Optional[List[str]]*) – Command line arguments.

piculet.**make_parser**(*prog*)
Build a parser for command line arguments.

Parameters **prog** (*str*) – Name of program.

Return type ArgumentParser

Returns Parser for arguments.

piculet.**preprocess**(*root*, *pre*)
Process a tree before starting extraction.

Parameters

- **root** (*Element*) – Root of tree to process.
- **pre** (*Sequence[Mapping[str, Any]]*) – Descriptions for processing operations.

piculet.**preprocessors = <piculet.Registry object>**
Predefined preprocessors.

piculet.**reducers = <piculet.Registry object>**
Predefined reducers.

piculet.**remove_elements**(*root*, *path*)
Remove selected elements from the tree.

Parameters

- **root** (*Element*) – Root element of the tree.
- **path** (*str*) – XPath to select the elements to remove.

piculet.**scrape**(*document*, *spec*)
Extract data from a document after optionally preprocessing it.

Parameters

- **document** (*str*) – Document to scrape.
- **spec** (*Mapping[str, Any]*) – Extraction specification.

Return type Mapping[str, Any]

Returns Extracted data.

piculet.**scrape_document**(*address*, *spec*, *content_format='xml'*)
Scrape data from a file path or a URL and print.

Parameters

- **address** (*str*) – File path or URL of document to scrape.
- **spec** (*str*) – Path of spec file.
- **content_format** (*Optional[str]*) – Whether the content is XML or HTML.

piculet.**set_element_attr**(*root*, *path*, *name*, *value*)
Set an attribute for selected elements.

Parameters

- **root** (*Element*) – Root element of the tree.
- **path** (*str*) – XPath to select the elements to set attributes for.
- **name** (*Union[str, Mapping[str, Any]]*) – Description for name generation.
- **value** (*Union[str, Mapping[str, Any]]*) – Description for value generation.

piculet.**set_element_text**(*root*, *path*, *text*)
Set the text for selected elements.

Parameters

- **root** (*Element*) – Root element of the tree.
- **path** (*str*) – XPath to select the elements to set attributes for.
- **text** (*Union[str, Mapping[str, Any]]*) – Description for text generation.

piculet.**transformers = <piculet.Registry object>**
Predefined transformers.

1.6 History

1.6.1 1.0b7 (2018-03-21)

- Dropped support for Python 3.3.
- Fixes for handling Unicode data in HTML for Python 2.
- Added registry for preprocessors.

1.6.2 1.0b6 (2018-01-17)

- Support for writing specifications in YAML.

1.6.3 1.0b5 (2018-01-16)

- Added a class-based API for writing specifications.
- Added predefined transformation functions.
- Removed callables from specification maps. Use the new API instead.
- Added support for registering new reducers and transformers.
- Added support for defining sections in document.
- Refactored XPath evaluation method in order to parse path expressions once.
- Preprocessing will be done only once when the tree is built.
- Concatenation is now the default reducing operation.

1.6.4 1.0b4 (2018-01-02)

- Added "–version" option to command line arguments.
- Added option to force the use of lxml's HTML builder.
- Fixed the error where non-truthy values would be excluded from the result.
- Added support for transforming node text during preprocess.
- Added separate preprocessing function to API.
- Renamed the "join" reducer as "concat".
- Renamed the "foreach" keyword for keys as "section".
- Removed some low level debug messages to substantially increase speed.

1.6.5 1.0b3 (2017-07-25)

- Removed the caching feature.

1.6.6 1.0b2 (2017-06-16)

- Added helper function for getting cache hash keys of URLs.

1.6.7 1.0b1 (2017-04-26)

- Added optional value transformations.
- Added support for custom reducer callables.
- Added command-line option for scraping documents from local files.

1.6.8 1.0a2 (2017-04-04)

- Added support for Python 2.7.
- Fixed lxml support.

1.6.9 1.0a1 (2016-08-24)

- First release on PyPI.

CHAPTER 2

Indices and Tables

- genindex
- search

Python Module Index

p

piculet, 16

Index

A

apply() (piculet.Extractor method), 17
apply() (piculet.Path method), 18
apply() (piculet.Rules method), 19

B

build_tree() (in module piculet), 19

D

decode_html() (in module piculet), 20

E

extract() (in module piculet), 20
extract() (piculet.Extractor method), 17
extract() (piculet.Rule method), 18
Extractor (class in piculet), 16
extractor (piculet.Rule attribute), 19

F

foreach (piculet.Extractor attribute), 17
foreach (piculet.Rule attribute), 19
from_map() (piculet.Extractor static method), 17
from_map() (piculet.Rule static method), 19

G

get() (piculet.Registry method), 18

H

h2x() (in module piculet), 20
handle_charref() (piculet.HTMLNormalizer method), 17
handle_data() (piculet.HTMLNormalizer method), 17
handle_endtag() (piculet.HTMLNormalizer method), 17
handle_entityref() (piculet.HTMLNormalizer method), 17
handle_starttag() (piculet.HTMLNormalizer method), 17
html_to_xhtml() (in module piculet), 20
HTMLNormalizer (class in piculet), 17

K

key (piculet.Rule attribute), 19

M

main() (in module piculet), 20
make_parser() (in module piculet), 20

P

Path (class in piculet), 17
path (piculet.Path attribute), 18
piculet (module), 16
preprocess() (in module piculet), 20
preprocessors (in module piculet), 21

R

reduce (piculet.Path attribute), 18
reducers (in module piculet), 21
register() (piculet.Registry method), 18
Registry (class in piculet), 18
remove_elements() (in module piculet), 21
Rule (class in piculet), 18
Rules (class in piculet), 19
rules (piculet.Rules attribute), 19

S

scrape() (in module piculet), 21
scrape_document() (in module piculet), 21
section (piculet.Rules attribute), 19
SELF_CLOSING_TAGS (piculet.HTMLNormalizer attribute), 17
set_element_attr() (in module piculet), 21
set_element_text() (in module piculet), 21

T

transform (piculet.Extractor attribute), 17
transformers (in module piculet), 21

X

XPath (class in piculet), 19