soupy Documentation

Release 0.3

Chris Beaumont

April 13, 2015

Contents

Soupy is a wrapper around BeautifulSoup that makes it easier to search through HTML and XML documents.

```
from soupy import Soupy, Q

html = """
<div id="main">
  <div>The web is messy</div>
  and full of traps
  <div>but Soupy loves you</div>
</div>"""

print(Soupy(html).find(id='main').children
      .each(Q.text.strip()) # extract text from each node, trim whitespace
      .filter(len)          # remove empty strings
      .val())               # dump out of Soupy

[u'The web is messy', u'and full of traps', u'but Soupy loves you']
```

Compare to the same task in BeautifulSoup:

```
from bs4 import BeautifulSoup, NavigableString

html = """
<div id="main">
  <div>The web is messy</div>
  and full of traps
  <div>but Soupy loves you</div>
</div>"""

result = []
for node in BeautifulSoup(html).find(id='main').children:
    if isinstance(node, NavigableString):
        text = node.strip()
    else:
        text = node.text.strip()
    if len(text):
        result.append(text)

print(result)

[u'The web is messy', u'and full of traps', u'but Soupy loves you']
```

Soupy uses BeautifulSoup under the hood and provides a very similar API, while smoothing over some of the warts in BeautifulSoup. Soupy also adds a functional interface for chaining together operations, gracefully dealing with failed searches, and extracting data into simpler formats.

CHAPTER 1

Installation

```
pip install soupy
```

or download the GitHub source.

Contents:

1.1 Getting Started

1.1.1 The Problem Soupy Aims to Solve

BeautifulSoup is a great library for searching through HTML and XML documents. However, the datatypes returned by BeautifulSoup methods can be inconsistent, especially with messily-structured files. For example, consider the following sensible query to find the first p tag inside the content div under an h2 tag:

```
dom.find('h2').find('div', 'content').find('p')
```

Depending on the document, each of these find calls may return a Tag, a unicode string, an integer (if the previous `find` returned a string instead of a Tag), None, or raise an AttributeError (if the previous `find` returned an integer or None instead of a Tag). Of these, only Tags can be safely chained together. In general, code like the line above risks generating exceptions. There are lots of other examples like this in BeautifulSoup.

In a nutshell, Soupy lets you safely chain queries together more gracefully, even when searches fail.

```
dom.find('h2').find('div', 'content').find('p').orelse('not found').val()
```

Let's see how that works.

1.1.2 Comparison to BeautifulSoup

Soupy wraps BeautifulSoup objects inside special wrappers:

- A `Node` wraps a BeautifulSoup DOM object like a Tag or a NavigableString.
- A `Collection` wraps a list of other wrappers together.
- A `Scalar` wraps other objects (numbers, strings, dicts, etc).

The most important object is `Node`. It behaves very similarly to a BeautifulSoup `Tag`, but its behavior is more predictable.

When BeautifulSoup is well-behaved, Soupy code is basically identical:

```
html = "<html>Hello<b>world</b></html>"
bs = BeautifulSoup(html)
soup = Soupy(html)

bs_val = bs.find('b').text
soup_val = soup.find('b').text.val()
assert bs_val == soup_val

bs_val = bs.find('b').parent
soup_val = soup.find('b').parent.val()
assert bs_val == soup_val
```

Notice that in these examples, the only difference is that you always call `val()` to pull data out of a Soupy wrapper when you are ready. **This is the essential concept to learn when transitioning from BeautifulSoup to Soupy**.

Things get more interesting when we look at corner cases (and the web is *full* of corner cases). For example, consider what happens when a search doesn't match anything:

```
>>> bs.find('b').find('a')    # AttributeError
>>> soup.find('b').find('a')
NullNode()
```

BeautifulSoup returns `None` when a match fails, which makes it impossible to chain expressions together. Soupy returns a `NullNode`, which can be further chained without raising exceptions. However, since `NullNode` represents a failed match, trying to extract any data raises an error:

```
>>> soup.find('b').find('a').val()
Traceback (most recent call last):
...
NullValueError:
```

Fortunately the `Node.orelse()` method can be used to specify a fallback value when a query doesn't match:

```
>>> soup.find('b').find('a').orelse('Not Found').val()
'Not Found'
```

There are lots of little corner cases like this in BeautifulSoup – sometimes functions return strings instead of Tags, sometimes they return None, sometimes certain methods or attributes aren't defined, etc.

Soupy's API is more predictable, and better suited for searching through messily-formated documents. Here are the main properties and methods copied over from BeautifulSoup. All of these features perform the same conceptual task as their BeautifulSoup counterparts, but they *always* return the same wrapper class. The primary goal of Soupy's design is to allow you to string together complex queries, without worrying about query failures at each step of the search.

- Properties and Methods that return `Node` (or `NullNode`)
- `Node.parent`
- `Node.next_sibling`
- `Node.previous_sibling`
- `Node.find()`
- `Node.find_parent()`
- `Node.find_next_sibling()`
- `Node.find_previous_sibling()`
- Properties that return `Scalar` (or `Null`)

- `Node.text`
- `Node.attrs`
- `Node.name`
- Properties and Methods that return a `Collection` of Nodes
- `Node.children`
- `Node.contents`
- `Node.descendants`
- `Node.parents`
- `Node.next_siblings`
- `Node.previous_siblings`
- `Node.find_all()`
- `Node.select()`
- `Node.find_parents()`
- `Node.find_next_siblings()`
- `Node.find_previous_siblings()`

1.1.3 Functional API

The main benefit of Soupy's wrappers is the ability to reliably chain them together. This also allows you to use general purpose libraries like itertools, functools, toolz, more_itertools, etc., to compose more complex data processing pipelines. For convenience, Soupy also priovides several such utilities to support more extensive method chaining.

Iterating over results with each, dump, dictzip

A common pattern in BeautifulSoup is to iterate over results from a call like `find_all()` using a list comprehension. For example, consider the query to extract all the movie titles on this IMDB page

```
import requests
url = 'http://chrisbeaumont.github.io/soupy/imdb_demo.html'
html = requests.get(url).text
bs = BeautifulSoup(html, 'html.parser')
soup = Soupy(html, 'html.parser')

print([node.find('a').text
       for node in bs.find_all('td', 'title')])

[u'The Shawshank Redemption', u'The Dark Knight', u'Inception',...]
```

Soupy provides an additional syntax for this with the `each()` method:

```
>>> print(soup.find_all('td', 'title').each(
...       lambda node: node.find('a').text).val())
[u'The Shawshank Redemption',...]
```

`Collection.each()` applies a function to every node in a collection, and wraps the result into a new collection.

Because typing `lambda` all the time is cumbersome, Soupy also has a shorthand `Q` object to make this task easier. This same query can be written as

```
>>> print(soup.find_all('td', 'title').each(Q.find('a').text).val())
[u'The Shawshank Redemption',...]
```

Think of `Q[stuff]` as shorthand for `lambda x: x[stuff]`.

The `Collection.dump()` method works similarly to `each()`, except that it extracts multiple values from each node, and packs them into a list of dictionaries. It's a convenient way to extract a JSON blob out of a document.

For example,

```
print(soup.find_all('td', 'title').dump(
      name=Q.find('a').text,
      year=Q.find('span', 'year_type').text[1:-1]
).val())

[{'name': u'The Shawshank Redemption', 'year': u'1994'}, ...]
```

Note: You can also run dump on a node to extract a single dictionary.

If you want to set keys based on a list of values (instead of hardcoding them), you can use the `Collection.dictzip()` method.

```
keys = Soupy('<b>a</b><b>b</b><b>c</b>')
vals = Soupy('<b>1</b><b>2</b><b>3</b>')

keys = keys.find_all('b').each(Q.text)
vals = vals.find_all('b').each(Q.text)
print(vals.dictzip(keys).val() == {'a': '1', 'b': '2', 'c': '3'})

True
```

`dictzip()` is so-named because the output is equivalent to `dict(zip((keys.val(), vals.val())))`

Transforming values with map and apply

Notice in the IMDB example above that we extracted each "year" value as a string.

```
>>> y = soup.find('td', 'title').find('span', 'year_type').text[1:-1]
>>> y
Scalar(u'1994')
```

We'd like to use integers instead. The `map()` and `apply()` methods let us transform the data inside a Soupy wrapper, and build a new wrapper out of the transformed value.

`map()` takes a function as input, applies that function to the wrapped data, and returns a new wrapper. So we can extract integer years via

```
>>> y.map(int)
Scalar(1994)
```

`map()` can be applied to any wrapper:

- Scalar.map applies the transformation to the data in the scalar
- Node.map applies the transformation to the BeautifulSoup element
- Collection.map applies the transformation to the list of nodes (rarely used)

The `apply()` function is similar to `map()`, except that the input function is called on the wrapper itself, and not the data inside the wrapper (the output will be re-wrapped automatically if needed).

Note also that Q-expressions are not restricted to working with Soupy nodes – they can be used on any object. For example, to uppercase all movie titles:

```
>>> soup.find('td', 'title').find('a').text.map(Q.upper())
Scalar(u'THE SHAWSHANK REDEMPTION')
```

Filtering Collections with filter, takewhile, dropwhile

The `filter()`, `takewhile()`, and `dropwhile()` methods remove unwanted nodes from collections. They accept a function which is applied to each element in the collection, and converted to a boolean value. `filter(func)` removes items where `func(item)` is False. `takewhile(func)` removes items on and after the first False, and `dropwhile(func)` drops items until the first True return value.

```
>>> soup.find_all('td', 'title').each(Q.find('a').text).filter(Q.startswith('B')).val()
[u'Batman Begins', u'Braveheart', u'Back to the Future']
```

This query selects only movies whose titles begin with "B".

You can also filter lists using slice syntax `nodes[::3]`.

Combining most of these ideas, here's a succinct JSON-summary of the IMDB movie list:

```
cast_split = Q.text != '\n    With: '

print(soup.find_all('td', 'title').dump(
      name=Q.find('a').text,
      year=Q.find('span', 'year_type').text,
      genres=Q.find('span', 'genre').find_all('a').each(Q.text),
      cast=(Q.find('span', 'credit').contents.dropwhile(cast_split)[1::2].each(Q.text)),
      directors=(Q.find('span', 'credit').contents.takewhile(cast_split)[1::2].each(Q.text)),
      rating=(Q.select('div.user_rating span.rating-rating span.value')[0].text.map(float)),
).val())

[{'rating': 9.3,
  'genres': [u'Crime', u'Drama'],
  'name': u'The Shawshank Redemption',
  'cast': [u'Tim Robbins', u'Morgan Freeman', u'Bob Gunton']...
```

Enforcing Assertions with nonnull and require

Soupy prevents unmatched queries from raising errors until `val` is called. Usually that's convenient, but sometimes you want to "fail loudly" in the event of unexpected input. There are a few methods to help with this.

`nonnull()` raises a `NullValueError` if called on a Null wrapper, and returns the unmodified wrapper otherwise. Thus, it can be used to require that part of a query has matched.

```
>>> s = Soupy('<p> No links here </p>')
>>> s.find('p').nonnull().find('a')['href'].orelse(None)
Scalar(None)
>>> s.find('b').nonnull().find('a')['href'].orelse(None)
Traceback (most recent call last):
...
NullValueError:
```

Here we require that the first `find` matches against something, while providing a fallback in case the second `find` fails.

`require()` behaves like `assert`: it takes a function which is `apply`-ed to the wrapper, and raises an exception if the result isn't Truthy.

```
>>> s = Scalar(3)
>>> s.require(Q > 2, 'Too small!')
Scalar(3)
>>> s.require(Q > 5, 'Too small!')
Traceback (most recent call last):
...
NullValueError: Too small!
```

1.1.4 Working with Q Expressions

Many of the previous examples have used the *Q* function-builder as a shorthand for `lambda` or manually defined functions. As mentioned above, `Q[stuff]` is rougly equivalent to `lambda x: x[stuff]`, so it should feel natural to pick up. Here are some example Q expressions, and their lambda equivalents:

Q Expression	lambda expression
`Q + 3`	`lambda x: x + 3`
`Q.a`	`lambda x: x.a`
`Q(5)`	`lambda x: x(5)`
`Q.func(3)`	`lambda x: x.func(3)`
`Q[key]`	`lambda x: x['key']`
`Q.map(Q > 3)`	`lambda x: x.map(lambda y: y > 3)`

The third example introduces a slight twist with Q expressions. Because `Q(5)` builds a function like `lambda x: x(5)`, we can't directly call this function using the normal `(arg)` syntax – doing so would actually build a *new* function behaving like `lambda x: x(5)(arg)`. You normally don't need to manually evaulate Q expressions, but if you do you can use the `eval_()` method.

```
>>> x = Q.upper()[0:2]
>>> x('testing')  # No! Builds a new function
Q.upper()[slice(0, 2, None)]('testing')
>>> x.eval_('testing')  # Yes!
'TE'
```

Debugging Q expressions

Despite your best efforts, you will *still* encounter messy documents that trigger errors in your code. Here's a simplified example:

```
>>> html = ['<a href="/index"></a>'] * 100
>>> html[30] = '<a href="#"></a>'
>>> dom = Soupy(''.join(html))
>>> dom.find_all('a').each(Q['href'].split('/')[1])
Traceback (most recent call last):
...
IndexError: list index out of range

    Encountered when evaluating Scalar(['#'])[1]
```

This code tries to extract the links in all `a` tags, but fails on links that don't have a slash. Debugging issues like this can be frustrating, because these errors are often triggered by rare edge cases in the document that can be hard to track down.

If your errors are generated inside a Q expression (as is the case here), the `Q.debug_` method will return data to isolate the failure.

```
>>> dbg = Q.debug_()
>>> dbg
QDebug(expr=Q['href'].split('/')[1], inner_expr=[1], val=Node(<a href="#"></a>), inner_val=Scalar(['#
>>> dbg.expr
Q['href'].split('/')[1]
>>> dbg.inner_expr
[1]
>>> dbg.val
Node(<a href="#"></a>)
>>> dbg.inner_val
Scalar(['#'])
```

The attributes returned by `debug_` are the full Q expression that triggered the error, the specific subexpression that triggered the error (in this case, the `['href']` part), the value that was passed to `full_expr`, and the value passsed to `expr`. So for example we can re-trigger the error via

```
>>> dbg.expr.eval_(dbg.val)
Traceback (most recent call last):
...
IndexError: list index out of range

    Encountered when evaluating Scalar(['#'])[1]
```

1.2 Soupy API Documentation

1.2.1 Main Wrapper Classes

class soupy.**Node**(*value*)

The Node class is the main wrapper around BeautifulSoup elements like Tag. It implements many of the same properties and methods as BeautifulSoup for navigating through documents, like find, select, parents, etc.

dump(***kwargs*)

Extract derived values into a Scalar(dict)

The keyword names passed to this function become keys in the resulting dictionary.

The keyword values are functions that are called on this Node.

Notes

- The input functions are called on the Node, **not** the underlying BeautifulSoup element
- If the function returns a wrapper, it will be unwrapped

Example

```
>>> soup = Soupy("<b>hi</b>").find('b')
>>> data = soup.dump(name=Q.name, text=Q.text).val()
>>> data == {'text': 'hi', 'name': 'b'}
True
```

val()

Return the value inside a wrapper.

Raises `NullValueError` if called on a Null object

orelse(*value*)
Provide a fallback value for failed matches.

Examples

```
>>> Scalar(5).orelse(10).val()
5
>>> Null().orelse(10).val()
10
```

nonnull()
Require that a node is not null

Null values will raise NullValueError, whereas nonnull values return self.

useful for being strict about portions of queries.

Examples

node.find('a').nonnull().find('b').orelse(3)

This will raise an error if find('a') doesn't match, but provides a fallback if find('b') doesn't match.

require(*func*, *msg=u'Requirement violated'*)
Assert that self.apply(func) is True.

Parameters

- **func** – func(wrapper)
- **msg** – str The error message to display on failure

Returns If self.apply(func) is True, returns self. Otherwise, raises NullValueError.

attrs
A `Scalar` of this Node's attribute dictionary

Example

```
>>> Soupy("<a val=3></a>").find('a').attrs
Scalar({'val': '3'})
```

children
A `Collection` of the child elements.

contents
A `Collection` of the child elements.

descendants
A `Collection` of all elements nested inside this Node.

find(**args*, ***kwargs*)
Find a single Node among this Node's descendants.

Returns `NullNode` if nothing matches.

This inputs to this function follow the same semantics as BeautifulSoup. See http://bit.ly/bs4doc for more info.

Examples

•node.find('a') # look for *a* tags

•node.find('a', 'foo') # look for *a* tags with class='foo'

•node.find(func) # find tag where func(tag) is True

•node.find(val=3) # look for tag like <a, val=3>

find_all(**args*, ***kwargs*)
Like `find()`, but selects all matches (not just the first one).

Returns a `Collection`.

If no elements match, this returns a Collection with no items.

find_next_sibling(**args*, ***kwargs*)
Like `find()`, but searches through `next_siblings`

find_next_siblings(**args*, ***kwargs*)
Like `find_all()`, but searches through `next_siblings`

find_parent(**args*, ***kwargs*)
Like `find()`, but searches through `parents`

find_parents(**args*, ***kwargs*)
Like `find_all()`, but searches through `parents`

find_previous_sibling(**args*, ***kwargs*)
Like `find()`, but searches through `previous_siblings`

find_previous_siblings(**args*, ***kwargs*)
Like `find_all()`, but searches through `previous_siblings`

name
A `Scalar` of this Node's tag name.

Example

```
>>> node = Soupy('<p>hi there</p>').find('p')
>>> node
Node(<p>hi there</p>)
>>> node.name
Scalar('p')
```

next_sibling
The `Node` sibling after this, or `NullNode`

next_siblings
A `Collection` of all siblings after this node

parent
The parent `Node`, or `NullNode`

parents
A `Collection` of the parents elements.

previous_sibling
The `Node` sibling prior to this, or `NullNode`

previous_siblings
A Collection of all siblings before this node

select (*selector*)
Like find_all(), but takes a CSS selector string as input.

text
A Scalar of this Node's text.

Example

```
>>> node = Soupy('<p>hi there</p>').find('p')
>>> node
Node(<p>hi there</p>)
>>> node.text
Scalar(u'hi there')
```

class soupy.**Collection** (*items*)
Collection's store lists of other wrappers.

They support most of the list methods (len, iter, getitem, etc).

apply (*func*)
Call a function on a wrapper, and wrap the result if necessary.

Parameters **func** – function(wrapper) -> val

Examples

```
>>> s = Scalar(5)
>>> s.apply(lambda val: isinstance(val, Scalar))
Scalar(True)
```

map (*func*)
Call a function on a wrapper's value, and wrap the result if necessary.

Parameters **func** – function(val) -> val

Examples

```
>>> s = Scalar(3)
>>> s.map(Q * 2)
Scalar(6)
```

all()
Scalar(True) if all items are truthy, or collection is empty.

any()
Scalar(True) if any items are truthy. False if empty.

count()
Return the number of items in the collection, as a Scalar

dictzip (*keys*)
Turn this collection into a Scalar(dict), by zipping keys and items.

Parameters **keys** – list or Collection of NavigableStrings The keys of the dictionary

Examples

```
>>> c = Collection([Scalar(1), Scalar(2)])
>>> c.dictzip(['a', 'b']).val() == {'a': 1, 'b': 2}
True
```

dropwhile (*func*)
Return a new Collection with the first few items removed.

Parameters **func** – function(Node) -> Node

Returns A new Collection, discarding all items before the first item where bool(func(item)) == True

dump (**args*, ***kwargs*)
Build a list of dicts, by calling `Node.dump()` on each item.

Each keyword provides a function that extracts a value from a Node.

Examples

```
>>> c = Collection([Scalar(1), Scalar(2)])
>>> c.dump(x2=Q*2, m1=Q-1).val()
[{'x2': 2, 'm1': 0}, {'x2': 4, 'm1': 1}]
```

each (**funcs*)
Call *func* on each element in the collection.

If multiple functions are provided, each item in the output will be a tuple of each func(item) in self.

Returns a new Collection.

Example

```
>>> col = Collection([Scalar(1), Scalar(2)])
>>> col.each(Q * 10)
Collection([Scalar(10), Scalar(20)])
>>> col.each(Q * 10, Q - 1)
Collection([Scalar((10, 0)), Scalar((20, 1))])
```

filter (*func*)
Return a new Collection with some items removed.

Parameters **func** – function(Node) -> Node

Returns A new Collection consisting of the items where bool(func(item)) == True

Examples

node.find_all('a').filter(Q['href'].startswith('http'))

first ()
Return the first element of the collection, or `Null`

iter_val ()
An iterator version of `val()`

none()
Scalar(True) if no items are truthy, or collection is empty.

takewhile(*func*)
Return a new Collection with the last few items removed.

Parameters **func** – function(Node) -> Node

Returns A new Collection, discarding all items at and after the first item where bool(func(item)) == False

Examples

node.find_all('tr').takewhile(Q.find_all('td').count() > 3)

val()
Unwraps each item in the collection, and returns as a list

zip(**others*)
Zip the items of this collection with one or more other sequences, and wrap the result.

Unlike Python's zip, all sequences must be the same length.

Parameters **others** – One or more iterables or Collections

Returns A new collection.

Examples

```
>>> c1 = Collection([Scalar(1), Scalar(2)])
>>> c2 = Collection([Scalar(3), Scalar(4)])
>>> c1.zip(c2).val()
[(1, 3), (2, 4)]
```

class soupy.**Scalar**(*value*)
A wrapper around single values.

Scalars support boolean testing (<, ==, etc), and use the wrapped value in the comparison. They return the result as a Scalar(bool).

Calling a Scalar calls the wrapped value, and wraps the result.

Examples

```
>>> s = Scalar(3)
>>> s > 2
Scalar(True)
>>> s.val()
3
>>> s + 5
Scalar(8)
>>> s + s
Scalar(6)
>>> bool(Scalar(3))
True
>>> Scalar(lambda x: x+2)(5)
Scalar(7)
```

1.2.2 Null Wrappers

class soupy.**NullValueError**
The NullValueError exception is raised when attempting to extract values from Null objects

class soupy.**Null**
The class for ill-defined Scalars.

class soupy.**NullNode**
NullNode is returned when a query doesn't match any node in the document.

children
Returns the NullCollection

contents
Returns the NullCollection

descendants
Returns the NullCollection

dump(***kwargs*)
Returns Null

find(**args*, ***kwargs*)
Returns NullNode

find_all(**args*, ***kwargs*)
Returns NullCollection

find_next_sibling(**args*, ***kwargs*)
Returns NullNode

find_next_siblings(**args*, ***kwargs*)
Returns NullCollection

find_parent(**args*, ***kwargs*)
Returns NullNode

find_parents(**args*, ***kwargs*)
Returns NullCollection

find_previous_sibling(**args*, ***kwargs*)
Returns NullNode

find_previous_siblings(**args*, ***kwargs*)
Returns NullCollection

next_sibling
Returns the NullNode

next_siblings
Returns the NullCollection

parent
Returns the NullNode

parents
Returns the NullCollection

previous_sibling
Returns the NullNode

previous_siblings
Returns the NullCollection

select(*selector*)
Returns NullCollection

class soupy.**NullCollection**
Represents in invalid Collection.

Returned by some methods on other Null objects.

1.2.3 Expressions

class soupy.**Expression**
Soupy expressions are a shorthand for building single-argument functions.

Users should use the Q object, which is just an instance of Expression.

debug_()
Returns debugging information for the previous error raised during expression evaluation.

Returns a QDebug namedtuple with four fields:

- expr is the last full expression to have raised an exception
- inner_expr is the specific sub-expression that raised the exception
- val is the value that expr tried to evaluate.
- inner_val is the value that inner_expr tried to evaluate

If no exceptions have been triggered from expression evaluation, then each field is None.

Examples

```
>>> Scalar('test').map(Q.upper().foo)
Traceback (most recent call last):
...
AttributeError: 'str' object has no attribute 'foo'
...
>>> dbg = Q.debug_()
>>> dbg.expr
Q.upper().foo
>>> dbg.inner_expr
.foo
>>> dbg.val
'test'
>>> dbg.inner_val
'TEST'
```

eval_(*val*)
Pass the argument val to the function, and return the result.

This special method is necessary because the __call__ method builds a new function stead of evaluating the current one.

class soupy.**QDebug**
Namedtuple that holds information about a failed expression evaluation.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*

Q

R

S

T

V

Z