itemloaders Documentation

Zyte

Apr 18, 2024

CONTENTS

`itemloaders` provide a convenient mechanism for populating data records. Its design provides a flexible, efficient and easy mechanism for extending and overriding different field parsing rules, either by raw data, or by source format (HTML, XML, etc) without becoming a nightmare to maintain.

To install `itemloaders`, run:

```
pip install itemloaders
```

Note: Under the hood, `itemloaders` uses itemadapter as a common interface. This means you can use any of the types supported by `itemadapter` here.

Warning: `dataclasses` and `attrs` support is still experimental. Please, refer to *`default_item_class`* in the *API Reference* for more information.

CHAPTER

ONE

GETTING STARTED WITH ITEMLOADERS

To use an Item Loader, you must first instantiate it. You can either instantiate it with a dict-like object (*item*) or without one, in which case an *item* is automatically instantiated in the Item Loader `__init__` method using the *item* class specified in the *ItemLoader.default_item_class* attribute.

Then, you start collecting values into the Item Loader, typically using CSS or XPath Selectors. You can add more than one value to the same item field; the Item Loader will know how to "join" those values later using a proper processing function.

Note: Collected data is stored internally as lists, allowing to add several values to the same field. If an `item` argument is passed when creating a loader, each of the item's values will be stored as-is if it's already an iterable, or wrapped with a list if it's a single value.

Here is a typical Item Loader usage:

```
from itemloaders import ItemLoader
from parsel import Selector

html_data = '''
<!DOCTYPE html>
<html>
    <head>
        <title>Some random product page</title>
    </head>
    <body>
        <div class="product_name">Some random product page</div>
        <p id="price">$ 100.12</p>
    </body>
</html>
'''
l = ItemLoader(selector=Selector(html_data))
l.add_xpath('name', '//div[@class="product_name"]/text()')
l.add_xpath('name', '//div[@class="product_title"]/text()')
l.add_css('price', '#price::text')
l.add_value('last_updated', 'today') # you can also use literal values
item = l.load_item()
item
# {'name': ['Some random product page'], 'price': ['$ 100.12'], 'last_updated': ['today']}
```

By quickly looking at that code, we can see the `name` field is being extracted from two different XPath locations in the page:

1. `//div[@class="product_name"]`
2. `//div[@class="product_title"]`

In other words, data is being collected by extracting it from two XPath locations, using the *add_xpath()* method. This is the data that will be assigned to the `name` field later.

Afterwards, similar calls are used for `price` field using a CSS selector with the *add_css()* method, and finally the `last_update` field is populated directly with a literal value (`today`) using a different method: *add_value()*.

Finally, when all data is collected, the *ItemLoader.load_item()* method is called which actually returns the item populated with the data previously extracted and collected with the *add_xpath()*, *add_css()*, and *add_value()* calls.

1.1 Contents

1.1.1 Declaring Item Loaders

Item Loaders are declared by using a class definition syntax. Here is an example:

```
from itemloaders import ItemLoader
from itemloaders.processors import TakeFirst, MapCompose, Join

class ProductLoader(ItemLoader):

    default_output_processor = TakeFirst()

    name_in = MapCompose(str.title)
    name_out = Join()

    # using a built-in processor
    price_in = MapCompose(str.strip)

    # using a function
    def price_out(self, values):
        return float(values[0])

loader = ProductLoader()
loader.add_value('name', 'plasma TV')
loader.add_value('price', '999.98')
loader.load_item()
# {'name': 'Plasma Tv', 'price': 999.98}
```

As you can see, input processors are declared using the `_in` suffix while output processors are declared using the `_out` suffix. And you can also declare a default input/output processors using the *ItemLoader.default_input_processor* and *ItemLoader.default_output_processor* attributes.

The precedence order, for both input and output processors, is as follows:

1. Item Loader field-specific attributes: `field_in` and `field_out` (most precedence)
2. Field metadata (`input_processor` and `output_processor` keys).

   Check out itemadapter field metadata for more information.

   Added in version 1.0.1.

3. Item Loader defaults: *ItemLoader.default_input_processor()* and *ItemLoader.default_output_processor()* (least precedence)

See also: *Reusing and extending Item Loaders*.

1.1.2 Input and Output processors

An Item Loader contains one input processor and one output processor for each (item) field. The input processor processes the extracted data as soon as it's received (through the *add_xpath()*, *add_css()* or *add_value()* methods) and the result of the input processor is collected and kept inside the ItemLoader. After collecting all data, the *ItemLoader.load_item()* method is called to populate and get the populated item object. That's when the output processor is called with the data previously collected (and processed using the input processor). The result of the output processor is the final value that gets assigned to the item.

Let's see an example to illustrate how the input and output processors are called for a particular field (the same applies for any other field):

```
l = ItemLoader(selector=some_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_css('name', css) # (3)
l.add_value('name', 'test') # (4)
return l.load_item() # (5)
```

So what happens is:

1. Data from `xpath1` is extracted, and passed through the *input processor* of the `name` field. The result of the input processor is collected and kept in the Item Loader (but not yet assigned to the item).
2. Data from `xpath2` is extracted, and passed through the same *input processor* used in (1). The result of the input processor is appended to the data collected in (1) (if any).
3. This case is similar to the previous ones, except that the data is extracted from the `css` CSS selector, and passed through the same *input processor* used in (1) and (2). The result of the input processor is appended to the data collected in (1) and (2) (if any).
4. This case is also similar to the previous ones, except that the value to be collected is assigned directly, instead of being extracted from a XPath expression or a CSS selector. However, the value is still passed through the input processors. In this case, since the value is not iterable it is converted to an iterable of a single element before passing it to the input processor, because input processor always receive iterables.
5. The data collected in steps (1), (2), (3) and (4) is passed through the *output processor* of the `name` field. The result of the output processor is the value assigned to the `name` field in the item.

It's worth noticing that processors are just callable objects, which are called with the data to be parsed, and return a parsed value. So you can use any function as input or output processor. The only requirement is that they must accept one (and only one) positional argument, which will be an iterable.

Note: Both input and output processors must receive an iterable as their first argument. The output of those functions can be anything. The result of input processors will be appended to an internal list (in the Loader) containing the collected values (for that field). The result of the output processors is the value that will be finally assigned to the item.

The other thing you need to keep in mind is that the values returned by input processors are collected internally (in lists) and then passed to output processors to populate the fields.

Last, but not least, `itemloaders` comes with some *commonly used processors* built-in for convenience.

1.1.3 Item Loader Context

The Item Loader Context is a mechanism that allows to change the input/ouput processors behavior. It's just a `dict` of arbitrary key/values which is shared among all processors. By default, the context contains the `selector` and any other *keyword arguments* sent to the Loaders's `__init__`. The context can be passed when declaring, instantiating or using Item Loader.

For example, suppose you have a function `parse_length` which receives a text value and extracts a length from it:

```
def parse_length(text, loader_context):
    unit = loader_context.get('unit', 'm')
    # ... length parsing code goes here ...
    return parsed_length
```

By accepting a `loader_context` argument the function is explicitly telling the Item Loader that it's able to receive an Item Loader context, so the Item Loader passes the currently active context when calling it, and the processor function (`parse_length` in this case) can thus use them.

There are several ways to modify Item Loader context values:

1. By modifying the currently active Item Loader context (*context* attribute):

   ```
   loader = ItemLoader(product)
   loader.context['unit'] = 'cm'
   ```

2. On Item Loader instantiation (the keyword arguments of Item Loader `__init__` method are stored in the Item Loader context):

   ```
   loader = ItemLoader(product, unit='cm')
   ```

3. On Item Loader declaration, for those input/output processors that support instantiating them with an Item Loader context. *MapCompose* is one of them:

   ```
   class ProductLoader(ItemLoader):
       length_out = MapCompose(parse_length, unit='cm')
   ```

1.1.4 Nested Loaders

When parsing related values from a subsection of a document, it can be useful to create nested loaders. Imagine you're extracting details from a footer of a page that looks something like:

Example:

```
<footer>
    <a class="social" href="https://facebook.com/whatever">Like Us</a>
    <a class="social" href="https://twitter.com/whatever">Follow Us</a>
    <a class="email" href="mailto:whatever@example.com">Email Us</a>
</footer>
```

Without nested loaders, you need to specify the full xpath (or css) for each value that you wish to extract.

Example:

```
loader = ItemLoader()
# load stuff not in the footer
```

(continues on next page)

(continued from previous page)

```
loader.add_xpath('social', '//footer/a[@class = "social"]/@href')
loader.add_xpath('email', '//footer/a[@class = "email"]/@href')
loader.load_item()
```

Instead, you can create a nested loader with the footer selector and add values relative to the footer. The functionality is the same but you avoid repeating the footer selector.

Example:

```
loader = ItemLoader()
# load stuff not in the footer
footer_loader = loader.nested_xpath('//footer')
footer_loader.add_xpath('social', 'a[@class = "social"]/@href')
footer_loader.add_xpath('email', 'a[@class = "email"]/@href')
# no need to call footer_loader.load_item()
loader.load_item()
```

You can nest loaders arbitrarily and they work with either xpath or css selectors. As a general guideline, use nested loaders when they make your code simpler but do not go overboard with nesting or your parser can become difficult to read.

1.1.5 Reusing and extending Item Loaders

Item Loaders are designed to ease the maintenance burden of parsing rules, without losing flexibility and, at the same time, providing a convenient mechanism for extending and overriding them. For this reason Item Loaders support traditional Python class inheritance for dealing with differences in data schemas.

Suppose, for example, that you get some particular product names enclosed in three dashes (e.g. `---Plasma TV---`) and you don't want to end up with those dashes in the final product names.

Here's how you can remove those dashes by reusing and extending the default Product Item Loader (`ProductLoader`):

```
from itemloaders.processors import MapCompose
from myproject.loaders import ProductLoader

def strip_dashes(x):
    return x.strip('-')

class SiteSpecificLoader(ProductLoader):
    name_in = MapCompose(strip_dashes, ProductLoader.name_in)
```

Another case where extending Item Loaders can be very helpful is when you have multiple source formats, for example XML and HTML. In the XML version you may want to remove `CDATA` occurrences. Here's an example of how to do it:

```
from itemloaders.processors import MapCompose
from myproject.ItemLoaders import ProductLoader
from myproject.utils.xml import remove_cdata

class XmlProductLoader(ProductLoader):
    name_in = MapCompose(remove_cdata, ProductLoader.name_in)
```

And that's how you typically extend input/output processors.

There are many other possible ways to extend, inherit and override your Item Loaders, and different Item Loaders hierarchies may fit better for different projects. `itemloaders` only provides the mechanism; it doesn't impose any specific organization of your Loaders collection - that's up to you and your project's needs.

1.1.6 Available built-in processors

Even though you can use any callable function as input and output processors, `itemloaders` provides some commonly used processors, which are described below.

Some of them, like the *MapCompose* (which is typically used as input processor) compose the output of several functions executed in order, to produce the final parsed value.

Here is a list of all built-in processors: This module provides some commonly used processors for Item Loaders.

See documentation in docs/topics/loaders.rst

class itemloaders.processors.**Compose**(**functions*, ***default_loader_context*)

A processor which is constructed from the composition of the given functions. This means that each input value of this processor is passed to the first function, and the result of that function is passed to the second function, and so on, until the last function returns the output value of this processor.

By default, stop process on `None` value. This behaviour can be changed by passing keyword argument `stop_on_none=False`.

Example:

```
>>> from itemloaders.processors import Compose
>>> proc = Compose(lambda v: v[0], str.upper)
>>> proc(['hello', 'world'])
'HELLO'
```

Each function can optionally receive a `loader_context` parameter. For those which do, this processor will pass the currently active *Loader context* through that parameter.

The keyword arguments passed in the `__init__` method are used as the default Loader context values passed to each function call. However, the final Loader context values passed to functions are overridden with the currently active Loader context accessible through the *ItemLoader.context* attribute.

class itemloaders.processors.**Identity**

The simplest processor, which doesn't do anything. It returns the original values unchanged. It doesn't receive any `__init__` method arguments, nor does it accept Loader contexts.

Example:

```
>>> from itemloaders.processors import Identity
>>> proc = Identity()
>>> proc(['one', 'two', 'three'])
['one', 'two', 'three']
```

class itemloaders.processors.**Join**(*separator=' '*)

Returns the values joined with the separator given in the `__init__` method, which defaults to `' '`. It doesn't accept Loader contexts.

When using the default separator, this processor is equivalent to the function: `' '.join`

Examples:

```
>>> from itemloaders.processors import Join
>>> proc = Join()
>>> proc(['one', 'two', 'three'])
'one two three'
>>> proc = Join('<br>')
>>> proc(['one', 'two', 'three'])
'one<br>two<br>three'
```

class itemloaders.processors.**MapCompose**(**functions*, ***default_loader_context*)

A processor which is constructed from the composition of the given functions, similar to the *Compose* processor. The difference with this processor is the way internal results are passed among functions, which is as follows:

The input value of this processor is *iterated* and the first function is applied to each element. The results of these function calls (one for each element) are concatenated to construct a new iterable, which is then used to apply the second function, and so on, until the last function is applied to each value of the list of values collected so far. The output values of the last function are concatenated together to produce the output of this processor.

Each particular function can return a value or a list of values, which is flattened with the list of values returned by the same function applied to the other input values. The functions can also return `None` in which case the output of that function is ignored for further processing over the chain.

This processor provides a convenient way to compose functions that only work with single values (instead of iterables). For this reason the *MapCompose* processor is typically used as input processor, since data is often extracted using the `extract()` method of parsel selectors, which returns a list of unicode strings.

The example below should clarify how it works:

```
>>> def filter_world(x):
...     return None if x == 'world' else x
...
>>> from itemloaders.processors import MapCompose
>>> proc = MapCompose(filter_world, str.upper)
>>> proc(['hello', 'world', 'this', 'is', 'something'])
['HELLO', 'THIS', 'IS', 'SOMETHING']
```

As with the Compose processor, functions can receive Loader contexts, and `__init__` method keyword arguments are used as default context values. See *Compose* processor for more info.

class itemloaders.processors.**SelectJmes**(*json_path*)

Query the input string for the jmespath (given at instantiation), and return the answer Requires : jmespath(https://github.com/jmespath/jmespath) Note: SelectJmes accepts only one input element at a time.

Example:

```
>>> from itemloaders.processors import SelectJmes, Compose, MapCompose
>>> proc = SelectJmes("foo") #for direct use on lists and dictionaries
>>> proc({'foo': 'bar'})
'bar'
>>> proc({'foo': {'bar': 'baz'}})
{'bar': 'baz'}
```

Working with Json:

```
>>> import json
>>> proc_single_json_str = Compose(json.loads, SelectJmes("foo"))
>>> proc_single_json_str('{"foo": "bar"}')
```

(continues on next page)

(continued from previous page)

```
'bar'
>>> proc_json_list = Compose(json.loads, MapCompose(SelectJmes('foo')))
>>> proc_json_list('[{"foo":"bar"}, {"baz":"tar"}]')
['bar']
```

class itemloaders.processors.**TakeFirst**

Returns the first non-null/non-empty value from the values received, so it's typically used as an output processor to single-valued fields. It doesn't receive any `__init__` method arguments, nor does it accept Loader contexts.

Example:

```
>>> from itemloaders.processors import TakeFirst
>>> proc = TakeFirst()
>>> proc(['', 'one', 'two', 'three'])
'one'
```

1.1.7 API Reference

class itemloaders.**ItemLoader**(*item=None*, *selector=None*, *parent=None*, ***context*)

Return a new Item Loader for populating the given item. If no item is given, one is instantiated automatically using the class in *default_item_class*.

When instantiated with a :param `selector` parameter the *ItemLoader* class provides convenient mechanisms for extracting data from web pages using parsel selectors.

Parameters

- **item** (`dict` object) – The item instance to populate using subsequent calls to *add_xpath()*, *add_css()*, *add_jmes()* or *add_value()*.
- **selector** (`Selector` object) – The selector to extract data from, when using the *add_xpath()* (resp. *add_css()*, *add_jmes()*) or *replace_xpath()* (resp. *replace_css()*, *replace_jmes()*) method.

The item, selector and the remaining keyword arguments are assigned to the Loader context (accessible through the *context* attribute).

item

The item object being parsed by this Item Loader. This is mostly used as a property so when attempting to override this value, you may want to check out *default_item_class* first.

context

The currently active *Context* of this Item Loader. Refer to <loaders-context> for more information about the Loader Context.

default_item_class

An Item class (or factory), used to instantiate items when not given in the `__init__` method.

> **Warning:** Currently, this factory/class needs to be callable/instantiated without any arguments. If you are using `dataclasses`, please consider the following alternative:
>
> ```
> from dataclasses import dataclass, field
> from typing import Optional
> ```

```
@dataclass
class Product:
    name: Optional[str] = field(default=None)
    price: Optional[float] = field(default=None)
```

default_input_processor

The default input processor to use for those fields which don't specify one.

default_output_processor

The default output processor to use for those fields which don't specify one.

selector

The `Selector` object to extract data from. It's the selector given in the `__init__` method. This attribute is meant to be read-only.

add_css(*field_name*, *css*, **processors*, *re=None*, ***kw*)

Similar to *`ItemLoader.add_value()`* but receives a CSS selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *`ItemLoader`*.

See *`get_css()`* for `kwargs`.

Parameters

css (*str*) – the CSS selector to extract data from

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.add_css('name', 'p.product-name')
# HTML snippet: <p id="price">the price is $1200</p>
loader.add_css('price', 'p#price', re='the price is (.*)')
```

add_jmes(*field_name*, *jmes*, **processors*, *re=None*, ***kw*)

Similar to *`ItemLoader.add_value()`* but receives a JMESPath selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *`ItemLoader`*.

See *`get_jmes()`* for `kwargs`.

Parameters

jmes (*str*) – the JMESPath selector to extract data from

Examples:

```
# HTML snippet: {"name": "Color TV"}
loader.add_jmes('name')
# HTML snippet: {"price": the price is $1200"}
loader.add_jmes('price', TakeFirst(), re='the price is (.*)')
```

add_value(*field_name*, *value*, **processors*, *re=None*, ***kw*)

Process and then add the given `value` for the given field.

The value is first passed through *`get_value()`* by giving the `processors` and `kwargs`, and then passed through the *field input processor* and its result appended to the data collected for that field. If the field already contains collected data, the new data is added.

The given `field_name` can be `None`, in which case values for multiple fields may be added. And the processed value should be a dict with field_name mapped to values.

Examples:

```
loader.add_value('name', 'Color TV')
loader.add_value('colours', ['white', 'blue'])
loader.add_value('length', '100')
loader.add_value('name', 'name: foo', TakeFirst(), re='name: (.+)')
loader.add_value(None, {'name': 'foo', 'sex': 'male'})
```

add_xpath(*field_name*, *xpath*, **processors*, *re=None*, ***kw*)

Similar to *ItemLoader.add_value()* but receives an XPath instead of a value, which is used to extract a list of strings from the selector associated with this *ItemLoader*.

See *get_xpath()* for `kwargs`.

Parameters

xpath (*str*) – the XPath to extract data from

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.add_xpath('name', '//p[@class="product-name"]')
# HTML snippet: <p id="price">the price is $1200</p>
loader.add_xpath('price', '//p[@id="price"]', re='the price is (.*)')
```

get_collected_values(*field_name*)

Return the collected values for the given field.

get_css(*css*, **processors*, *re=None*, ***kw*)

Similar to *ItemLoader.get_value()* but receives a CSS selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

Parameters

- **css** (*str*) – the CSS selector to extract data from
- **re** (*str or Pattern*) – a regular expression to use for extracting data from the selected CSS region

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.get_css('p.product-name')
# HTML snippet: <p id="price">the price is $1200</p>
loader.get_css('p#price', TakeFirst(), re='the price is (.*)')
```

get_jmes(*jmes*, **processors*, *re=None*, ***kw*)

Similar to *ItemLoader.get_value()* but receives a JMESPath selector instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

Parameters

- **jmes** (*str*) – the JMESPath selector to extract data from
- **re** (*str or Pattern*) – a regular expression to use for extracting data from the selected JMESPath

Examples:

```
# HTML snippet: {"name": "Color TV"}
loader.get_jmes('name')
```

(continues on next page)

(continued from previous page)

```
# HTML snippet: {"price": the price is $1200"}
loader.get_jmes('price', TakeFirst(), re='the price is (.*)')
```

get_output_value(*field_name*)

Return the collected values parsed using the output processor, for the given field. This method doesn't populate or modify the item at all.

get_value(*value*, **processors*, *re=None*, ***kw*)

Process the given `value` by the given `processors` and keyword arguments.

Available keyword arguments:

Parameters
re (*str or Pattern*) – a regular expression to use for extracting data from the given value using `extract_regex()` method, applied before processors

Examples:

```
>>> from itemloaders import ItemLoader
>>> from itemloaders.processors import TakeFirst
>>> loader = ItemLoader()
>>> loader.get_value('name: foo', TakeFirst(), str.upper, re='name: (.+)')
'FOO'
```

get_xpath(*xpath*, **processors*, *re=None*, ***kw*)

Similar to *ItemLoader.get_value()* but receives an XPath instead of a value, which is used to extract a list of unicode strings from the selector associated with this *ItemLoader*.

Parameters

- **xpath** (*str*) – the XPath to extract data from
- **re** (*str or Pattern*) – a regular expression to use for extracting data from the selected XPath region

Examples:

```
# HTML snippet: <p class="product-name">Color TV</p>
loader.get_xpath('//p[@class="product-name"]')
# HTML snippet: <p id="price">the price is $1200</p>
loader.get_xpath('//p[@id="price"]', TakeFirst(), re='the price is (.*)')
```

load_item()

Populate the item with the data collected so far, and return it. The data collected is first passed through the *output processors* to get the final value to assign to each item field.

nested_css(*css*, ***context*)

Create a nested loader with a css selector. The supplied selector is applied relative to selector associated with this *ItemLoader*. The nested loader shares the item with the parent *ItemLoader* so calls to *add_xpath()*, *add_value()*, *replace_value()*, etc. will behave as expected.

nested_xpath(*xpath*, ***context*)

Create a nested loader with an xpath selector. The supplied selector is applied relative to selector associated with this *ItemLoader*. The nested loader shares the item with the parent *ItemLoader* so calls to *add_xpath()*, *add_value()*, *replace_value()*, etc. will behave as expected.

replace_css(*field_name*, *css*, **processors*, *re=None*, ***kw*)

Similar to *add_css()* but replaces collected data instead of adding it.

replace_jmes(*field_name*, *jmes*, **processors*, *re=None*, ***kw*)

Similar to *add_jmes()* but replaces collected data instead of adding it.

replace_value(*field_name*, *value*, **processors*, *re=None*, ***kw*)

Similar to *add_value()* but replaces the collected data with the new value instead of adding it.

replace_xpath(*field_name*, *xpath*, **processors*, *re=None*, ***kw*)

Similar to *add_xpath()* but replaces collected data instead of adding it.

1.1.8 Release notes

itemloaders 1.2.0 (2024-04-18)

- Added official support for Python 3.12 and PyPy 3.10 (#75)
- Removed official support for Python 3.7 (#72)
- Improved performance of `itemloaders.utils.arg_to_iter` (#51)
- Fixed test expectations on recent Python versions (#77)
- Improved CI (#78)

itemloaders 1.1.0 (2023-04-21)

- Added JMESPath support (*ItemLoader.add_jmes()* etc.), requiring Parsel 1.8.1+ (#68)
- Added official support for Python 3.11 (#59)
- Removed official support for Python 3.6 (#61)
- Internal code cleanup (#65, #66)
- Added `pre-commit` support and applied changes from `black` and `flake8` (#70).
- Improved CI (#60)

itemloaders 1.0.6 (2022-08-29)

- Fixes a regression introduced in 1.0.5 that would cause the `re` parameter of *ItemLoader.add_xpath()* and similar methods to be passed to lxml, which would trigger an exception when the value of `re` was a compiled pattern and not a string (#56)

itemloaders 1.0.5 (2022-08-25)

- Allow additional args to be passed when calling *ItemLoader.add_xpath()* (#48)
- Fixed missing space in an exception message (#47)
- Updated company name in author and copyright sections (#42)
- Added official support for Python 3.9 and improved PyPy compatibility (#44)
- Added official support for Python 3.10 (#53)

itemloaders 1.0.4 (2020-11-12)

- When adding a `scrapy.item.scrapy.Item` object as a value into an *`ItemLoader`* object, that item is now added *as is*, instead of becoming a `list` of keys from its `scrapy.item.scrapy.Item.fields` (#28, #29)
- Increased test coverage (#27)

itemloaders 1.0.3 (2020-09-09)

- Calls to *`ItemLoader.get_output_value()`* no longer affect the output of *`ItemLoader.load_item()`* (#21, #22)
- Fixed some documentation links (#19, #23)
- Fixed some test warnings (#24)

itemloaders 1.0.2 (2020-08-05)

- Included the license file in the source releases (#13)
- Cleaned up some remnants of Python 2 (#16, #17)

itemloaders 1.0.1 (2020-07-02)

- Extended item type support to all item types supported by itemadapter (#13)
- *Input and output processors* defined in item field metadata are now taken into account (#13)
- Lowered some minimum dependency versions (#10):
  - parsel: 1.5.2 → 1.5.0
  - w3lib: 1.21.0 → 1.17.0
- Improved the README file (#9)
- Improved continuous integration (e62d95b)

itemloaders 1.0.0 (2020-05-18)

- Initial release, based on a part of the Scrapy code base.

PYTHON MODULE INDEX

i

INDEX