Scrapy



# Item Loaders

Item Loaders provide a convenient mechanism for populating scraped Items. Even though Items can be populated using their own dictionary-like API, Item Loaders provide a much more convenient API for populating them from a scraping process, by automating some common tasks like parsing the raw extracted data before assigning it.

In other words, Items provide the container of scraped data, while Item Loaders provide the mechanism for populating that container.

Item Loaders are designed to provide a flexible, efficient and easy mechanism for extending and overriding different field parsing rules, either by spider, or by source format (HTML, XML, etc) without becoming a nightmare to maintain.

## Using Item Loaders to populate items

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

Then, you start collecting values into the Item Loader, typically using 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.

Here is a typical Item Loader usage in a Spider, using the Product item declared in the Items chapter:

```
from scrapy.loader import ItemLoader
from myproject.items import Product

def parse(self, response):
    l = ItemLoader(item=Product(), response=response)
    l.add_xpath('name', '//div[@class="product_name"]')
```

```
    l.add_xpath('name', '//div[@class="product_title"]')
    l.add_xpath('price', '//p[@id="price"]')
    l.add_css('stock', 'p#stock]')
    l.add_value('last_updated', 'today') # you can also use literal values
    return l.load_item()
```

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` and `stock` fields (the latter 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.

## 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(Product(), 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 iterator.

**Note**

Both input and output processors must receive an iterator 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, Scrapy comes with some commonly used processors built-in for convenience.

## Declaring Item Loaders

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

```
from scrapy.loader import ItemLoader
from scrapy.loader.processors import TakeFirst, MapCompose, Join

class ProductLoader(ItemLoader):

    default_output_processor = TakeFirst()

    name_in = MapCompose(unicode.title)
```

```
    name_out = Join()

    price_in = MapCompose(unicode.strip)

    # ...
```

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.

## Declaring Input and Output Processors

As seen in the previous section, input and output processors can be declared in the Item Loader definition, and it's very common to declare input processors this way. However, there is one more place where you can specify the input and output processors to use: in the Item Field metadata. Here is an example:

```
import scrapy
from scrapy.loader.processors import Join, MapCompose, TakeFirst
from w3lib.html import remove_tags

def filter_price(value):
    if value.isdigit():
        return value

class Product(scrapy.Item):
    name = scrapy.Field(
        input_processor=MapCompose(remove_tags),
        output_processor=Join(),
    )
    price = scrapy.Field(
        input_processor=MapCompose(remove_tags, filter_price),
        output_processor=TakeFirst(),
    )
```

```
>>> from scrapy.loader import ItemLoader
>>> il = ItemLoader(item=Product())
>>> il.add_value('name', [u'Welcome to my', u'<strong>website</strong>'])
>>> il.add_value('price', [u'&euro;', u'<span>1000</span>'])
>>> il.load_item()
{'name': u'Welcome to my website', 'price': u'1000'}
```

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

See also: Reusing and extending Item Loaders.

## Item Loader Context

The Item Loader Context is a dict of arbitrary key/values which is shared among all input and output processors in the Item Loader. It can be passed when declaring, instantiating or using Item Loader. They are used to modify the behaviour of the input/output processors.

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 constructor 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')
   ```

## ItemLoader objects

class **scrapy.loader.ItemLoader**( [ item, selector, response, ] **kwargs)

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 selector or a response parameters the `ItemLoader` class provides convenient mechanisms for extracting data from web pages using selectors.

**Parameters:**
- **item** (`Item` object) – The item instance to populate using subsequent calls to `add_xpath()`, `add_css()`, or `add_value()`.
- **selector** (`Selector` object) – The selector to extract data from, when using the `add_xpath()` (resp. `add_css()`) or `replace_xpath()` (resp. `replace_css()`) method.
- **response** (`Response` object) – The response used to construct the selector using the `default_selector_class`, unless the selector argument is given, in which case this argument is ignored.

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

`ItemLoader` instances have the following methods:

**get_value**(value, *processors, **kwargs)

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

Available keyword arguments:

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

Examples:

```
>>> from scrapy.loader.processors import TakeFirst
>>> loader.get_value(u'name: foo', TakeFirst(), unicode.upper, re='name: (.+)')
'FOO`
```

**add_value**(field_name, value, *processors, **kwargs)

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', u'Color TV')
```

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

**replace_value**(field_name, value, *processors, **kwargs)

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

**get_xpath**(xpath, *processors, **kwargs)

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 compiled regex) – 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 (.*)')
```

**add_xpath**(field_name, xpath, *processors, **kwargs)

Similar to `ItemLoader.add_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`.

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 (.*)')
```

**replace_xpath**(field_name, xpath, *processors, **kwargs)

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

**get_css**(css, *processors, **kwargs)

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 compiled regex) – 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 (.*)')
```

### add_css(field_name, css, *processors, **kwargs)

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 (.*)')
```

### replace_css(field_name, css, *processors, **kwargs)

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

### 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_xpath(xpath)

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.

### nested_css(css)

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.

**`get_collected_values`**(field_name)

Return the collected values for the given field.

**`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_input_processor`**(field_name)

Return the input processor for the given field.

**`get_output_processor`**(field_name)

Return the output processor for the given field.

`ItemLoader` instances have the following attributes:

**`item`**

The `Item` object being parsed by this Item Loader.

**`context`**

The currently active Context of this Item Loader.

**`default_item_class`**

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

**`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.

**`default_selector_class`**

The class used to construct the `selector` of this `ItemLoader`, if only a response is given in the constructor. If a selector is given in the constructor this attribute is ignored. This attribute is sometimes overridden in subclasses.

`selector`

The `Selector` object to extract data from. It's either the selector given in the constructor or one created from the response given in the constructor using the `default_selector_class`. This attribute is meant to be read-only.

## 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(item=Item())
# load stuff not in the footer
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(item=Item())
# 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.

## Reusing and extending Item Loaders

As your project grows bigger and acquires more and more spiders, maintenance becomes a fundamental problem, especially when you have to deal with many different parsing rules for each spider, having a lot of exceptions, but also wanting to reuse the common processors.

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 of specific spiders (or groups of spiders).

Suppose, for example, that some particular site encloses their product names in three dashes (e.g. `---Plasma TV---`) and you don't want to end up scraping 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 scrapy.loader.processors import MapCompose
from myproject.ItemLoaders 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 scrapy.loader.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 processors.

As for output processors, it is more common to declare them in the field metadata, as they usually depend only on the field and not on each specific site parsing rule (as input processors do). See also: Declaring Input and 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. Scrapy 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.

## Available built-in processors

Even though you can use any callable function as input and output processors, Scrapy 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:

### class scrapy.loader.processors.Identity

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

Example:

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

### class scrapy.loader.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 constructor arguments, nor does it accept Loader contexts.

Example:

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

### class scrapy.loader.processors.Join(separator=u' ')

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

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

Examples:

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

### class scrapy.loader.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 scrapy.loader.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 constructor 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 scrapy.loader.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 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 scrapy.loader.processors import MapCompose
>>> proc = MapCompose(filter_world, unicode.upper)
>>> proc([u'hello', u'world', u'this', u'is', u'scrapy'])
[u'HELLO, u'THIS', u'IS', u'SCRAPY']
```

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

### class scrapy.loader.processors.SelectJmes(json_path)

Queries the value using the json path provided to the constructor and returns the output. Requires jmespath (https://github.com/jmespath/jmespath.py) to run. This processor takes only one input at a time.

Example:

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