web-poet Documentation

Release 0.17.0

Zyte Group Ltd

Mar 04, 2024

GETTING STARTED

`web-poet` is a Python 3.8+ implementation of the page object pattern for web scraping. It enables writing portable, reusable web parsing code.

> **Warning:** web-poet is in early stages of development; backward-incompatible changes are possible.

CHAPTER

ONE

OVERVIEW

A good web scraping framework helps to keep your code maintainable by, among other things, enabling and encouraging separation of concerns.

For example, Scrapy lets you implement different aspects of web scraping, like ban avoidance or data delivery, into separate components.

However, there are 2 core aspects of web scraping that can be hard to decouple: *crawling*, i.e. visiting URLs, and *parsing*, i.e. extracting data.

web-poet lets you *write data extraction code* that:

- Makes your web scraping code easier to maintain, since your data extraction and crawling code are no longer intertwined and can be maintained separately.
- Can be reused with different versions of your crawling code, i.e. with different crawling strategies.
- Can be executed independently of your crawling code, enabling easier debugging and easier automated testing.
- Can be used with any Python web scraping framework or library that implements the *web-poet specification*, either directly or through a third-party plugin. See *Frameworks*.

To learn more about why and how web-poet came to be, see *From the ground up*.

CHAPTER

TWO

INSTALLATION

To be able to write *page objects* and *test them*, install web-poet from PyPI:

```
pip install web-poet
```

To use page objects in production, however, you will need a *web-poet framework*.

CHAPTER

THREE

TUTORIAL

In this tutorial you will learn to use web-poet as you write web scraping code for book detail pages from books.toscrape.com.

To follow this tutorial you must first be familiar with Python and have *installed web-poet*.

3.1 Create a project directory

web-poet does not limit how you structure your web-poet web scraping code, beyond the limitations of Python itself.

However, in this tutorial you will use a specific project directory structure designed with web-poet best practices in mind. Consider using a similar project directory structure in all your web-poet projects.

First create your project directory: `tutorial-project/`.

Within the `tutorial-project` directory, create:

- A `run.py` file, a file specific to this tutorial where you will put code to test the execution of your web scraping code.
- A `tutorial` directory, where you will place your web scraping code.

Within the `tutorial-project/tutorial` directory, create:

- An `__init__.py` file, so that the `tutorial` directory becomes an importable Python module.
- An `items.py` file, where you will define item classes to store extracted data.
- A `pages` directory, where you will define your page object classes.

Within the `tutorial-project/tutorial/pages` directory, create:

- An `__init__.py` file.
- A `books_toscrape_com.py` file, for page object class code targeting books.toscrape.com.

Your project directory should look as follows:

```
tutorial-project
├── run.py
└── tutorial
    ├── __init__.py
    ├── items.py
    └── pages
        ├── __init__.py
        └── books_toscrape_com.py
```

3.2 Create an item class

While it is possible to store the extracted data in a Python dictionary, it is a good practice to create an item class that:

- Defines the specific attributes that you aim to extract, triggering an exception if you extract unintended attributes or fail to extract expected attributes.
- Allows defining default values for some attributes.

web-poet uses itemadapter for item class support, which means that any kind of item class can be used. In this tutorial, you will use attrs to define your item class.

Copy the following code into `tutorial-project/tutorial/items.py`:

```
from attrs import define


@define
class Book:
    title: str
```

This code defines a `Book` item class, with a single required `title` string attribute to store the book title.

`Book` is a minimal class designed specifically for this tutorial. In real web-poet projects, you will usually define item classes with many more attributes.

Tip: For an example of real item classes, see the zyte-common-items library.

Also mind that, while in this tutorial you use `Book` only for data from 1 website, books.toscrape.com, item classes are usually meant to be usable for many different websites that provide data with a similar data schema.

3.3 Create a page object class

To write web parsing code with web-poet, you write *page object classes*, Python classes that define how to extract data from a given type of input, usually some type of webpage from a specific website.

In this tutorial you will write a page object class for webpages of books.toscrape.com that show details about a book, such as these:

- http://books.toscrape.com/catalogue/the-exiled_247/index.html
- http://books.toscrape.com/catalogue/when-we-collided_955/index.html
- http://books.toscrape.com/catalogue/set-me-free_988/index.html

Copy the following code into `tutorial-project/tutorial/pages/books_toscrape_com.py`:

```
from web_poet import field, handle_urls, WebPage

from ..items import Book


@handle_urls("books.toscrape.com")
class BookPage(WebPage[Book]):

```

(continues on next page)

(continued from previous page)

```
    @field
    async def title(self):
        return self.css("h1::text").get()
```

In the code above:

- You define a page object class named `BookPage` by subclassing *`WebPage`*.

  It is possible to create a page object class subclassing instead the simpler *`ItemPage`* class. However, *`WebPage`*:

  - Indicates that your page object class requires an HTTP response as input, which gets stored in the *`response`* attribute of your page object class as an *`HttpResponse`* object.
  - Provides attributes like *`html`* and *`url`*, and methods like *`css()`*, *`urljoin()`*, and *`xpath()`*, that make it easier to write parsing code.

- `BookPage` declares `Book` as its return type.

  `WebPage`, like its parent class *`ItemPage`*, is a generic class that accepts a type parameter. Unlike most generic classes, however, the specified type parameter is used for more than type hinting: it determines the item class that is used to store the data that fields return.

- `BookPage` is decorated with *`handle_urls()`*, which indicates for which domain `BookPage` is intended to work.

  It is possible to specify more specific URL patterns, instead of only the target URL domain. However, the URL domain and the output type (`Book`) are usually all the data needed to determine which page object class to use, which is the goal of the *`handle_urls()`* decorator.

- `BookPage` defines a field named `title`.

  Fields are methods of page object classes, preferably async methods, decorated with *`field()`*. Fields define the logic to extract a specific piece of information from the input of your page object class.

  `BookPage.title` extracts the title of a book from a book details webpage. Specifically, it extracts the text from the first `h1` element on the input HTTP response.

  Here, `title` is not an arbitrary name. It was chosen specifically to match `Book.title`, so that during parsing the value that `BookPage.title` returns gets mapped to `Book.title`.

3.4 Use your page object class

Now that you have a page object class defined, it is time to use it.

First, install requests, which is required by `web_poet.example`.

Then copy the following code into `tutorial-project/run.py`:

```
from web_poet import consume_modules
from web_poet.example import get_item

from tutorial.items import Book

consume_modules("tutorial.pages")

item = get_item(
    "http://books.toscrape.com/catalogue/the-exiled_247/index.html",
    Book,
```

(continues on next page)

(continued from previous page)

```
)
print(item)
```

Execute that code:

```
python tutorial-project/run.py
```

And the `print(item)` statement should output the following:

```
Book(title='The Exiled')
```

In this tutorial you use `web_poet.example.get_item`, which is a simple, incomplete implementation of the web-poet specification, built specifically for this tutorial, for demonstration purposes. In real projects, use instead an actual *web-poet framework*.

`web_poet.example.get_item` serves to illustrate the power of web-poet: once you have defined your page object class, a web-poet framework only needs 2 inputs from you:

- the URL from which you want to extract data, and
- the desired output, either a *page object class* or, in this case, an *item class*.

Notice that you must also call *`consume_modules()`* once before your first call to `get_item`. `consume_modules` ensures that the specified Python modules are loaded. You pass `consume_modules` the import paths of the modules where your page object classes are defined. After loading those modules, *`handle_urls()`* decorators register the page object classes that they decorate into `web_poet.default_registry`, which `get_item` uses to determine which page object class to use based on its input parameters (URL and item class).

Your web-poet framework can take care of everything else:

1. It matches the input URL and item class to `BookPage`, based on the URL pattern that you defined with the *`handle_urls()`* decorator, and the return type that you declared in the page object class (`Book`).
2. It inspects the inputs declared by `BookPage`, and builds an instance of `BookPage` with the required inputs.

   `BookPage` is a *`WebPage`* subclass, and *`WebPage`* declares an attribute named `response` of type *`HttpResponse`*. Your web-poet framework sees this, and creates an *`HttpResponse`* object from the input URL as a result, by downloading the URL response, and assigns that object to the `response` attribute of a new `BookPage` object.
3. It builds the output item, `Book(title='The Exiled')`, using the *`to_item()`* method of `BookPage`, inherited from *`ItemPage`*, which in turn uses all fields of `BookPage` to create an instance of `Book`, which you declared as the return type of `BookPage`.

3.5 Extend and override your code

To continue this tutorial, you will need extended versions of `Book` and `BookPage`, with additional fields. However, rather than editing the existing `Book` and `BookPage` classes, you will see how you can instead create new classes that inherit them.

Append the following code to `tutorial-project/tutorial/items.py`:

```
from typing import Optional


@define
class CategorizedBook(Book):
```

(continues on next page)

(continued from previous page)

```
    category: str
    category_rank: Optional[int] = None
```

The code above defines a new item class, `CategorizedBook`, that inherits the `title` attribute from `Book` and defines 2 more attributes: `category` and `category_rank`.

Append the following code to `tutorial-project/tutorial/pages/books_toscrape_com.py`:

```
from web_poet import Returns

from ..items import CategorizedBook


@handle_urls("books.toscrape.com")
class CategorizedBookPage(BookPage, Returns[CategorizedBook]):

    @field
    async def category(self):
        return self.css(".breadcrumb a::text").getall()[-1]
```

In the code above:

- You define a new page object class: `CategorizedBookPage`.
- `CategorizedBookPage` subclasses `BookPage`, inheriting its `title` field, and defining a new one: `category`.

  `CategorizedBookPage` does *not* define a `category_rank` field yet, you will add it later on. For now, the default value defined in `CategorizedBook` for `category_rank` will be `None`.
- `CategorizedBookPage` indicates that it returns a `CategorizedBook` object.

  WebPage is a generic class, which is why we could use `WebPage[Book]` in the definition of `BookPage` to indicate `Book` as the output type of `BookPage`. However, `BookPage` is not a generic class, so something like `BookPage[CategorizedBook]` would not work.

  So instead you use *Returns*, a special, generic class that you can inherit to re-define the output type of your page object subclasses.

After you update your `tutorial-project/run.py` script to request a `CategorizedBook` item:

```
from web_poet import consume_modules
from web_poet.example import get_item

from tutorial.items import CategorizedBook

consume_modules("tutorial.pages")

item = get_item(
    "http://books.toscrape.com/catalogue/the-exiled_247/index.html",
    CategorizedBook,
)
print(item)
```

And you execute it again:

```
python tutorial-project/run.py
```

You can see in the new output that your new classes have been used:

```
CategorizedBook(title='The Exiled', category='Mystery', category_rank=None)
```

3.6 Use additional requests

To extract data about an item, sometimes the HTTP response to a single URL is not enough. Sometimes, you need additional HTTP responses to get all the data that you want. That is the case with the `category_rank` attribute.

The `category_rank` attribute indicates the position in which a book appears in the list of books of the category of that book. For example, The Exiled is 24th in the Mystery category, so the value of `category_rank` should be `24` for that book.

However, there is no indication of this value in the book details page. To get this value, you need to visit the URL of the category of the book whose data you are extracting, find the entry of that book within the grid of books of the category, and record in which position you found it. And categories with more than 20 books are split into multiple pages, so you may need more than 1 additional request for some books.

Extend `CategorizedBookPage` in `tutorial-project/tutorial/pages/books_toscrape_com.py` as follows:

```
from attrs import define

from web_poet import HttpClient, Returns

from ..items import CategorizedBook


@handle_urls("books.toscrape.com")
@define
class CategorizedBookPage(BookPage, Returns[CategorizedBook]):
    http: HttpClient
    _books_per_page = 20

    @field
    async def category(self):
        return self.css(".breadcrumb a::text").getall()[-1]

    @field
    async def category_rank(self):
        response, book_url, page = self.response, self.url, 0
        category_page_url = self.css(".breadcrumb a::attr(href)").getall()[-1]
        while category_page_url:
            category_page_url = response.urljoin(category_page_url)
            response = await self.http.get(category_page_url)
            urls = response.css("h3 a::attr(href)").getall()
            for position, url in enumerate(urls, start=1):
                url = str(response.urljoin(url))
                if url == book_url:
                    return page * self._books_per_page + position
            category_page_url = response.css(".next a::attr(href)").get()
            if not category_page_url:
                return None
            page += 1
```

In the code above:

- You declare a new input in `CategorizedBookPage`, `http`, of type *`HttpClient`*.

  You also add the `@attrs.define` decorator to `CategorizedBookPage`, as it is required when adding new required attributes to subclasses of attrs classes.

- You define the `category_rank` field so that it uses the `http` input object to send additional requests to find the position of the current book within its category.

  Specifically:

  1. You extract the category URL from the book details page.
  2. You visit that category URL, and you iterate over the listed books until you find one with the same URL as the current book.

     If you find a match, you return the position at which you found the book.
  3. If there is no match, and there is a next page, you repeat the previous step with the URL of that next page as the category URL.
  4. If at some point there are no more "next" pages and you have not yet found the book, you return `None`.

When you execute `tutorial-project/run.py` now, `category_rank` has the expected value:

```
CategorizedBook(title='The Exiled', category='Mystery', category_rank=24)
```

3.7 Use parameters

You may notice that the execution takes longer now. That is because `CategorizedBookPage` now requires 2 or more requests, to find the value of the `category_rank` attribute.

If you use `CategorizedBookPage` as part of a web scraping project that targets a single book URL, it cannot be helped. If you want to extract the `category_rank` attribute, you need those additional requests. Your only option to avoid additional requests is to stop extracting the `category_rank` attribute.

However, if your web scraping project is targeting all book URLs from one or more categories by visiting those category URLs, extracting book URLs from them, and then using `CategorizedBookPage` with those book URLs as input, there is something you can change to save many requests: keep track of the positions where you find books as you visit their categories, and pass that position to `CategorizedBookPage` as additional input.

Extend `CategorizedBookPage` in `tutorial-project/tutorial/pages/books_toscrape_com.py` as follows:

```
from attrs import define

from web_poet import HttpClient, PageParams, Returns

from ..items import CategorizedBook


@handle_urls("books.toscrape.com")
@define
class CategorizedBookPage(BookPage, Returns[CategorizedBook]):
    http: HttpClient
    page_params: PageParams
    _books_per_page = 20
```

(continues on next page)

(continued from previous page)

```
    @field
    async def category(self):
        return self.css(".breadcrumb a::text").getall()[-1]

    @field
    async def category_rank(self):
        category_rank = self.page_params.get("category_rank")
        if category_rank is not None:
            return category_rank
        response, book_url, page = self.response, self.url, 0
        category_page_url = self.css(".breadcrumb a::attr(href)").getall()[-1]
        while category_page_url:
            category_page_url = response.urljoin(category_page_url)
            response = await self.http.get(category_page_url)
            urls = response.css("h3 a::attr(href)").getall()
            for position, url in enumerate(urls, start=1):
                url = str(response.urljoin(url))
                if url == book_url:
                    return page * self._books_per_page + position
            category_page_url = response.css(".next a::attr(href)").get()
            if not category_page_url:
                return None
            page += 1
```

In the code above, you declare a new input in `CategorizedBookPage`, `page_params`, of type *PageParams*. It is a dictionary of parameters that you may receive from the code using your page object class.

In the `category_rank` field, you check if you have received a parameter also called `category_rank`, and if so, you return that value instead of using additional requests to find the value.

You can now update your `tutorial-project/run.py` script to pass that parameter to `get_item`:

```
item = get_item(
    "http://books.toscrape.com/catalogue/the-exiled_247/index.html",
    CategorizedBook,
    page_params={"category_rank": 24},
)
```

When you execute `tutorial-project/run.py` now, execution should take less time, but the result should be the same as before:

```
CategorizedBook(title='The Exiled', category='Mystery', category_rank=24)
```

Only that now the value of `category_rank` comes from `tutorial-project/run.py`, and not from additional requests sent by `CategorizedBookPage`.

CHAPTER

FOUR

FROM THE GROUND UP

Learn why and how web-poet came to be as you transform a simple, rigid starting web scraping code snippet into maintainable, reusable web-poet code.

4.1 Writing reusable parsing code

Imagine you are writing code to scrape a book web page from books.toscrape.com, and you implement a `scrape` function like this:

```
import requests
from parsel import Selector


def scrape(url: str) -> dict:
    response = requests.get(url)
    selector = Selector(response.text)
    return {
        "url": response.url,
        "title": selector.css("h1").get(),
    }

item = scrape("http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html")
```

This `scrape` function is simple, but it has a big issue: it only supports downloading the specified URL using the requests library. What if you want to use aiohttp, for concurrency support? What if you want to run `scrape` with a local snapshot of a URL response, to write an automated test for `scrape` that does not rely on a network connection?

The first step towards addressing this issue is to split your `scrape` function into 2 separate functions, `download` and `parse`:

```
import requests
from parsel import Selector


def parse(response: requests.Response) -> dict:
    selector = Selector(response.text)
    return {
        "url": response.url,
        "title": selector.css("h1").get(),
    }
```

(continues on next page)

(continued from previous page)

```
def download(url: str) -> requests.Response:
    return requests.get(url)

url = "http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html"
response = download(url)
item = parse(response)
```

Now that `download` and `parse` are separate functions, you can replace `download` with an alternative implementation that uses aiohttp, or that reads from local files.

There is still an issue, though: `parse` expects an instance of requests.Response. Any alternative implementation of `download` would need to create a response object of the same type, forcing a dependency on requests even if downloads are handled with a different library.

So you need to change the input of the `parse` function into something that will not tie you to a specific download library. One option is to create your own, download-independent `Response` class, to store the response data that any download function should be able to provide:

```
import requests
from dataclasses import dataclass
from parsel import Selector


@dataclass
class Response:
    url: str
    text: str


def parse(response: Response) -> dict:
    selector = Selector(response.text)
    return {
        "url": response.url,
        "title": selector.css("h1").get(),
    }


def download(url: str) -> Response:
    response = requests.get(url)
    return Response(url=response.url, text=response.text)


url = "http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html"
response = download(url)
item = parse(response)
```

The `parse` function is no longer tied to any specific download library, and alternative versions of the `download` function can be implemented with other libraries.

4.2 Parsing with web-poet

web-poet asks you to organize your code in a very similar way. Let's convert the `parse` function into a *web-poet page object class*:

```
import requests
from web_poet import Injectable, HttpResponse


class BookPage(Injectable):
    def __init__(self, response: HttpResponse):
        self.response = response

    def to_item(self) -> dict:
        return {
            "url": self.response.url,
            "title": self.response.css("h1").get(),
        }


def download(url: str) -> Response:
    response = requests.get(url)
    return HttpResponse(
        url=response.url,
        body=response.content,
        headers=response.headers,
    )


url = "http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html"
response = download(url)
book_page = BookPage(response=response)
item = book_page.to_item()
```

Differences from a previous example:

- web-poet provides a standard *HttpResponse* class, with helper methods like *css()*.

  Note how headers are passed when creating an *HttpResponse* instance. This is needed to properly decode the body (which is `bytes`) as text using web browser rules. It involves checking the `Content-Encoding` header, HTML meta tags, BOM markers in the body, etc.

- Instead of the `parse` function we've got a `BookPage` class, which inherits from the *Injectable* base class, receives response data in its `__init__` method, and returns the extracted item in the `to_item()` method. `to_item` is a standard method name used by `web-poet`.

Receiving a `response` argument in `__init__` is very common for page objects, so `web-poet` provides a shortcut for it: inherit from *WebPage*, which provides this `__init__` method implementation. You can then refactor your `BookPage` class as follows:

```
from web_poet import WebPage

class BookPage(WebPage):
    def to_item(self) -> dict:
        return {
```

(continues on next page)

(continued from previous page)

```
            "url": self.response.url,
            "title": self.response.css("h1").get(),
        }
```

WebPage even provides shortcuts for some response attributes and methods:

```
from web_poet import WebPage

class BookPage(WebPage):
    def to_item(self) -> dict:
        return {
            "url": self.url,
            "title": self.css("h1").get(),
        }
```

At this point you may be wondering why web-poet requires you to write a class with a `to_item` method rather than a function. The answer is flexibility.

For example, the use of a class instead of a function makes *fields* possible, which make parsing code easier to read:

```
from web_poet import WebPage, field


class BookPage(WebPage):
    @field
    def url(self):
        return self.url

    @field
    def title(self):
        return self.css("h1").get()
```

Using fields also makes it unnecessary to define `to_item()` manually, and allows reading individual fields when you don't need the complete `to_item()` output.

Note: The `BookPage.to_item()` method is `async` in the example above. See *Fields* for more information.

Using classes also makes it easy, for example, to implement dependency injection, which is how web-poet builds *inputs*.

4.3 Downloading with web-poet

What about the implementation of the `download` function? How would you implement that in web-poet? Well, ideally, you wouldn't.

To parse data from a web page using web-poet, you would only need to write the parsing part, e.g. the `BookPage` *page object class* above.

Then, you let a *web-poet framework* handle the download part for you. You pass that framework the URL of a web page to parse, and either a page object class (the `BookPage` class here) or an *item class*, and that's it:

```
item = some_framework.get(url, BookPage)
```

web-poet does *not* provide any framework, beyond *an example one featured in the tutorial* and not intended for production. The role of web-poet is to define a specification on how to write parsing logic so that it can be reused with different frameworks.

Page object classes should be flexible enough to be used with very different frameworks, including:

- synchronous or asynchronous frameworks
- asynchronous frameworks based on callbacks or based on coroutines (`async def / await` syntax)
- single-node and distributed systems
- different underlying HTTP implementations, or even implementations with no HTTP support at all

CHAPTER

FIVE

PAGE OBJECTS

A page object is a code wrapper for a webpage, or for a part of a webpage, that implements the logic to parse the raw webpage data into structured data.

To use web-poet, *define page object classes* for your target websites, and *get the output item* using a *web-poet framework*.

5.1 Defining a page object class

A page object class is a Python class that:

- Subclasses *ItemPage*.
- Declares *typed input parameters* in its `__init__` method.
- Uses *fields*.

  Alternatively, you can implement a `to_item` method, which can be synchronous or asynchronous, and returns the webpage content as an *item*.

For example:

```
from web_poet import HttpResponse, ItemPage, field


class FooPage(ItemPage[MyItem]):
    def __init__(self, response: HttpResponse):
        self.response = response

    @field
    def foo(self) -> str:
        return self.response.css(".foo").get()
```

Note: `MyItem` in the code examples of this page is a placeholder for an *item class*.

5.1.1 Minimizing boilerplate

There are a few ways for you to minimize boilerplate when defining a page object class.

For example, you can use attrs to remove the need for a custom `__init__` method:

```
from attrs import define

from web_poet import HttpResponse, ItemPage, field


@define
class FooPage(ItemPage[MyItem]):
    response: HttpResponse

    @field
    def foo(self) -> str:
        return self.response.css(".foo").get()
```

If your page object class needs *HttpResponse* as input, there is also *WebPage*, an *ItemPage* subclass that declares an *HttpResponse* input and provides helper methods to use it:

```
from web_poet import WebPage, field


class FooPage(WebPage[MyItem]):
    @field
    def foo(self) -> str:
        return self.css(".foo").get()
```

5.2 Getting the output item

You should *include your page object classes into a page object registry*, e.g. decorate them with *handle_urls()*:

```
from web_poet import WebPage, field, handle_urls


@handle_urls("example.com")
class FooPage(WebPage[MyItem]):
    @field
    def foo(self) -> str:
        return self.css(".foo").get()
```

Then, provided your page object class code is imported (see *consume_modules()*), your *framework* can build the output item after you provide the target URL and the desired *output item class*, as *shown in the tutorial*.

Your framework chooses the right page object class based on your input parameters, downloads the required data, builds a page object, and calls the `to_item` method of that page object.

Note that, while the examples above use `dict` as an output item for simplicity, using less generic *item classes* is recommended. That way, you can use different page object classes, with different output items, for the same website.

5.2.1 Getting a page object

Alternatively, frameworks can return a page object instead of an item, and you can call `to_item` yourself.

However, there are drawbacks to this approach:

- `to_item` can be synchronous or asynchronous, so you need to use *ensure_awaitable()*:

```
from web_poet.utils import ensure_awaitable

item = await ensure_awaitable(foo_page.to_item())
```

- `to_item` may raise certain exceptions, like *Retry* or *UseFallback*, which, depending on your *framework*, may not be handled automatically when getting a page object instead of an item.

5.2.2 Building a page object manually

It is possible to create a page object from a page object class passing its inputs as parameters. For example, to manually create an instance of the `FooPage` page object class defined above:

```
foo_page = FooPage(
    response=HttpResponse(
        "https://example.com",
        b"<!DOCTYPE html>\n<title>Foo</title>",
    ),
)
```

However, your code will break if the page object class changes its *inputs*. Building page objects using *frameworks* prevents that.

CHAPTER

SIX

INPUTS

Page object classes, in their `__init__` method, must define input parameters with type hints pointing to input classes.

Those input classes may be:

- *Built-in web-poet input classes*.
- *Custom input classes*.
- Other *page object classes*.
- *Item classes*, when using a *framework* that can provide item classes.
- Any other class that subclasses *`Injectable`* or is registered or decorated with `Injectable.register`.

Based on the target URL and parameter type hints, *frameworks* automatically build the required objects at run time, and pass them to the `__init__` method of the corresponding page object class.

For example, if a page object class has an `__init__` parameter of type *`HttpResponse`*, and the target URL is https://example.com, your framework would send an HTTP request to https://example.com, download the response, build an *`HttpResponse`* object with the response data, and pass it to the `__init__` method of the page object class being used.

6.1 Built-in input classes

Warning: Not all *frameworks* support all web-poet built-in input classes.

The *`web_poet.page_inputs`* module defines multiple classes that you can define as inputs for a page object class, including:

- *`HttpResponse`*, a complete HTTP response, including URL, headers, and body. This is the most common input for a page object class. See *Working with HttpResponse*.
- *`HttpClient`*, to send *additional requests*.
- *`RequestUrl`*, the target URL before following redirects. Useful, for example, to skip the target URL download, and instead use *`HttpClient`* to send a custom request based on parts of the target URL.
- *`PageParams`*, to receive data from the crawling code.
- *`Stats`*, to write key-value data pairs during parsing that you can inspect later, e.g. for debugging purposes.
- *`BrowserResponse`*, which includes URL, status code and *`BrowserHtml`* of a rendered web page.
- *`AnyResponse`*, which either holds *`BrowserResponse`* or *`HttpResponse`* as the `.response` instance, depending on which one is available or is more appropriate.

6.2 Working with HttpResponse

HttpResponse has many attributes and methods.

To get the entire response body, you can use *body* for the raw `bytes`, *text* for the `str` (decoded with the detected *encoding*), or *json()* to load a JSON response as a Python data structure:

```
>>> response.body
b'{"foo": "bar"}'
>>> response.text
'{"foo": "bar"}'
>>> response.json()
{'foo': 'bar'}
```

There are also methods to select content from responses: *jmespath()* for JSON and *css()* and *xpath()* for HTML and XML:

```
>>> response.jmespath("foo")
[<Selector query='foo' data='bar'>]
>>> response.css("h1::text")
[<Selector query='descendant-or-self::h1/text()' data='Title'>]
>>> response.xpath("//h1/text()")
[<Selector query='//h1/text()' data='Title'>]
```

6.3 Custom input classes

You may define your own input classes if you are using a *framework* that supports it.

However, note that custom input classes may make your *page object classes* less portable across frameworks.

CHAPTER

SEVEN

ITEMS

The `to_item` method of a *page object class* must return an item.

An item is a data container object supported by the itemadapter library, such as a `dict`, an attrs class, or a `dataclass()` class. For example:

```
@attrs.define
class MyItem:
    foo: int
    bar: str
```

Because itemadapter allows implementing support for arbitrary classes, any kind of Python object can potentially work as an item.

7.1 Defining the item class of a page object class

When inheriting from *ItemPage*, indicate the item class to return between brackets:

```
@attrs.define
class MyPage(ItemPage[MyItem]):
    ...
```

`to_item` builds an instance of the specified item class based on the page object class *fields*.

```
page = MyPage(...)
item = await page.to_item()
assert isinstance(item, MyItem)
```

You can also define *ItemPage* subclasses that are not meant to be used, only subclassed, and not annotate *ItemPage* in them. You can then annotate those classes when subclassing them:

```
@attrs.define
class MyBasePage(ItemPage):
    ...

@attrs.define
class MyPage(MyBasePage[MyItem]):
    ...
```

To change the item class of a subclass that has already defined its item class, use *Returns*:

```
@attrs.define
class MyOtherPage(MyPage, Returns[MyOtherItem]):
    ...
```

7.2 Best practices for item classes

To keep your code maintainable, we recommend you to:

- Instead of `dict`, use proper item classes based on `dataclasses` or attrs, to make it easier to detect issues like field name typos or missing required fields.
- Reuse item classes.

  For example, if you want to extract product details data from 2 e-commerce websites, try to use the same item class for both of them. Or at least try to define a base item class with shared fields, and only keep website-specific fields in website-specific items.
- Keep item classes as logic-free as possible.

  For example, any parsing and field cleanup logic is better handled through *page object classes*, e.g. using *field processors*.

  Having code that makes item field values different from their counterpart page object field values can subvert the expectations of users of your code, which might need to access page object fields directly, for example for field subset selection.

If you are looking for ready-made item classes, check out zyte-common-items.

CHAPTER

EIGHT

RULES

Rules are *ApplyRule* objects that tell web-poet which *page object class* to use based on user input, i.e. the target URL and the requested output class (a *page object class* or an *item class*).

Rules are necessary if you want to request an item instance, because rules tell web-poet which page object class to use to generate your item instance. Rules can also be useful as documentation or to get information about page object classes programmatically.

Rule precedence can also be useful. For example, to implement generic page object classes that you can override for specific websites.

8.1 Defining rules

The *handle_urls()* decorator is the simplest way to define a rule for a page object. For example:

```
from web_poet import ItemPage, handle_urls

from my_items import MyItem

@handle_urls("example.com")
class MyPage(ItemPage[MyItem]):
    ...
```

The code above tells web-poet to use the `MyPage` *page object class* when given a URL pointing to the `example.com` domain name and being asked for `MyPage` or `MyItem` as output class.

Alternatively, you can manually create and register *ApplyRule* objects:

```
from url_matcher import Patterns
from web_poet import ApplyRule, ItemPage, default_registry

from my_items import MyItem

class MyPage(ItemPage[MyItem]):
    ...

rule = ApplyRule(
    for_patterns=Patterns(include=['example.com']),
    use=MyPage,
    to_return=MyItem,
)
default_registry.add_rule(rule)
```

8.1.1 URL patterns

Every rule defines a `url_matcher.Patterns` object that determines if any given URL is a match for the rule.

`Patterns` objects offer a simple but powerful syntax for URL matching. For example:

Pattern	Behavior
(empty string)	Matches any URL
example.com	Matches any URL on the example.com domain and subdomains
example.com/products/	Matches example.com URLs under the /products/ path
example.com?productId=*	Matches example.com URLs with productId=… in their query string

For details and more examples, see the url-matcher documentation.

When using the `handle_urls()` decorator, its `include`, `exclude`, and `priority` parameters are used to create a `Patterns` object. When creating an *`ApplyRule`* object manually, you must create a `Patterns` object yourself and pass it to the `for_patterns` parameter of *`ApplyRule`*.

8.1.2 Rule precedence

Often you define rules so that a given user input, i.e. a combination of a target URL and an output class, can only match 1 rule. However, there are scenarios where it can be useful to define 2 or more rules that can all match a given user input.

For example, you might want to define a "generic" page object class with some default implementation of field extraction, e.g. based on semantic markup or machine learning, and be able to override it based on the input URL, e.g. for specific websites or URL patterns, with a more specific page object class.

For a given user input, when 2 or more rules are a match, web-poet breaks the tie as follows:

- One rule can indicate that its *page object class* **overrides** another page object class.

  This is specified by `ApplyRule.instead_of`. When using the `handle_urls()` decorator, the value comes from the `instead_of` parameter of the decorator.

  For example, the following page object class would override `MyPage` from *above*:

  ```
  @handle_urls("example.com", instead_of=MyPage)
  class OverridingPage(ItemPage[MyItem]):
      ...
  ```

  That is:

  - If the requested output class is `MyPage`, an instance of `OverridingPage` is returned instead.
  - If the requested output class is `MyItem`, an instance of `OverridingPage` is created, and used to build an instance of `MyItem`, which is returned.

- One rule can declare a higher **priority** than another rule, taking precedence.

  Rule priority is determined by the value of `ApplyRule.for_patterns.priority`. When using the `handle_urls()` decorator, the value comes from the `priority` parameter of the decorator. Rule priority is 500 by default.

  For example, given the following page object class:

```
@handle_urls("example.com", priority=510)
class PriorityPage(ItemPage[MyItem]):
    ...
```

The following would happen:

- If the requested output class is `MyItem`, an instance of `PriorityPage` is created, and used to build an instance of `MyItem`, which is returned.
- If the requested output class is `MyPage`, an instance of `MyPage` is returned, since `PriorityPage` is not defined as an override for `MyPage`.

`instead_of` triumphs `priority`: If a rule overrides another rule using `instead_of`, it does not matter if the overridden rule had a higher priority.

When multiple rules override the same page object class, through, `priority` can break the tie.

If none of those tie breakers are in place, the first rule added to the registry takes precedence. However, relying on registration order is discouraged, and you will get a warning if you register 2 or more rules with the same URL patterns, same output item class, same priority, and no `instead_of` value. See also *Rule conflicts*.

8.2 Rule registries

Rules should be stored in a *RulesRegistry* object.

web-poet defines a default, global *RulesRegistry* object at `web_poet.default_registry`. Rules defined with the *handle_urls()* decorator are added to this registry.

8.2.1 Loading rules

For a *framework* to apply your rules, you need to make sure that your code that adds those rules to `web_poet.default_registry` is executed.

When using the *handle_urls()* decorator, that usually means that you need to make sure that Python imports the files where the decorator is used.

You can use the *consume_modules()* function in some entry point of your code for that:

```
from web_poet import consume_modules

consume_modules("my_package.pages", "external_package.pages")
```

The ideal location for this function depends on your framework. Check the documentation of your framework for more information.

8.3 Rule conflicts

A rule conflict occurs when multiple rules have the same `instead_of` and `priority` values and can match the same URL.

When it affects rules defined in your code base, solve the conflict adjusting those `instead_of` and `priority` values as needed.

When it affects rules from a external package, you have the following options to solve the conflict:

- **Subclass** one of the conflicting page object classes in your code base, using a similar rule except for a tie-breaking change to its `instead_of` or `priority` value.

  For example, if `package1.A` and `package2.B` are page object classes with conflicting rules, with a default priority (500), and you want `package1.A` to take precedence, declare a new page object class as follows:

```
from package1 import A
from web_poet import handle_urls

@handle_urls(..., priority=510)
class NewA(A):
    pass
```

- If your *framework* allows defining a **custom list of rules**, you could use `web_poet.default_registry` methods like *`get_rules()`* or *`search()`* to build such a list, including only rules that have no conflicts.

CHAPTER

NINE

FIELDS

A field is a read-only property in a *page object class* decorated with `@field` instead of `@property`.

Each field is named after a key of the *item* that the page object class returns. A field uses the *inputs* of its page object class to return the right value for the matching item key.

For example:

```
from typing import Optional

import attrs
from web_poet import ItemPage, HttpResponse, field


@attrs.define
class MyPage(ItemPage):
    response: HttpResponse

    @field
    def foo(self) -> Optional[str]:
        return self.response.css(".foo").get()
```

9.1 Synchronous and asynchronous fields

Fields can be either synchronous (`def`) or asynchronous (`async def`).

Asynchronous fields make sense, for example, when sending *additional requests*:

```
from typing import Optional

import attrs
from web_poet import ItemPage, HttpClient, HttpResponse, field


@attrs.define
class MyPage(ItemPage):
    response: HttpResponse
    http: HttpClient

    @field
    def name(self) -> Optional[str]:
```

(continues on next page)

(continued from previous page)

```
        return self.response.css(".name").get()

    @field
    async def price(self) -> Optional[str]:
        resp = await self.http.get("...")
        return resp.json().get("price")
```

Unlike the values of synchronous fields, the values of asynchronous fields need to be awaited:

```
page = MyPage(...)
name = page.name
price = await page.price
```

Mixing synchronous and asynchronous fields can be messy:

- You need to know whether a field is synchronous or asynchronous to write the right code to read its value.
- If a field changes from synchronous to asynchronous or vice versa, calls that read the field need to be updated.

  Changing from synchronous to asynchronous might be sometimes necessary due to website changes (e.g. needing *additional requests*).

To address these issues, use *`ensure_awaitable()`* to read both synchronous and asynchronous fields with the same code:

```
from web_poet.utils import ensure_awaitable

page = MyPage(...)
name = await ensure_awaitable(page.name)
price = await ensure_awaitable(page.price)
```

Note: Using asynchronous fields only also works, but prevents accessing other fields from *field processors*.

9.2 Inheritance

To create a page object class that is very similar to another, subclassing the former page object class is often a good approach to maximize code reuse.

In a subclass of a *page object class* you can *reimplement fields*, *add fields*, *remove fields*, or *rename fields*.

9.2.1 Reimplementing a field

Reimplementing a field when subclassing a *page object class* should be straightforward:

```
import attrs
from web_poet import field, ensure_awaitable

from my_library import BasePage

@attrs.define
```

(continues on next page)

(continued from previous page)

```
class CustomPage(BasePage):

    @field
    async def foo(self) -> str:
        base_foo = await ensure_awaitable(super().foo)
        return f"{base_foo} (modified)"
```

9.2.2 Adding a field

To add a new field to a *page object class* when subclassing:

1. Define a new *item class* that includes the new field, for example a subclass of the item class returned by the original page object class.
2. In your new page object class, subclass both the original page object class and `Returns`, the latter including the new item class between brackets.
3. Implement the extraction code for the new *field* in the new page object class.

For example:

```
import attrs
from web_poet import field, Returns

from my_library import BasePage, BaseItem

@attrs.define
class CustomItem(BaseItem):
    new_field: str

@attrs.define
class CustomPage(BasePage, Returns[CustomItem]):

    @field
    def new_field(self) -> str:
        ...
```

9.2.3 Removing a field

To remove a field from a *page object class* when subclassing:

1. Define a new *item class* that defines all fields but the one being removed.
2. In your new page object class, subclass the original page object class, `Returns` with the new item class between brackets, and set `skip_nonitem_fields=True`.

   When building an item, page object class fields without a matching item class field will now be ignored, rather than raising an exception.

Your new page object class will still define the field, but the resulting item will not.

For example:

```
import attrs
from web_poet import Returns

from my_library import BasePage

@attrs.define
class CustomItem:
    kept_field: str

@attrs.define
class CustomPage(BasePage, Returns[CustomItem], skip_nonitem_fields=True):
    pass
```

Alternatively, you can consider *using a page object as input* for removing fields. It is more verbose than subclassing, because you need to define every field in your page object class, but it can catch some mismatches between page object class fields and item class fields that would otherwise be hidden by `skip_nonitem_fields`.

9.2.4 Renaming a field

To rename a field from a *page object class* when subclassing:

1. Define a new *item class* that defines all fields, including the renamed field.
2. In your new page object class, subclass the original page object class, *`Returns`* with the new item class between brackets, and set `skip_nonitem_fields=True`.

   When building an item, page object class fields without a matching item class field will now be ignored, rather than raising an exception.
3. Define a field for the new field name that returns the value from the old field name.

Your new page object class will still define the old field name, but the resulting item will not.

For example:

```
import attrs
from web_poet import Returns

from my_library import BasePage

@attrs.define
class CustomItem:
    new_field: str

@attrs.define
class CustomPage(BasePage, Returns[CustomItem], skip_nonitem_fields=True):

    @field
    async def new_field(self) -> str:
        return ensure_awaitable(self.old_field)
```

Alternatively, you can consider *using a page object as input* for renaming fields. It is more verbose than subclassing, because you need to define every field in your page object class, but it can catch some mismatches between page object class fields and item class fields that would otherwise be hidden by `skip_nonitem_fields`.

9.3 Composition

There are 2 forms of composition that you can use when writing a page object: *using a page object as input*, and *using a field mixing*.

9.3.1 Using a page object as input

You can reuse a page object class from another page object class using composition instead of *inheritance* by using the original page object class as a dependency in a brand new page object class returning a brand new item class.

This is a good approach when you want to reuse code but the page object classes are very different, or when you want to remove or rename fields without relying on `skip_nonitem_fields`.

For example:

```
import attrs
from web_poet import ItemPage, field, ensure_awaitable

from my_library import BasePage

@attrs.define
class CustomItem:
    name: str

@attrs.define
class CustomPage(ItemPage[CustomItem]):
    base: BasePage

    @field
    async def name(self) -> str:
        name = await ensure_awaitable(self.base.name)
        brand = await ensure_awaitable(self.base.brand)
        return f"{brand}: {name}"
```

Instead of a page object, it is possible to declare the *item* it returns as a dependency in your new page object class. For example:

```
import attrs
from web_poet import ItemPage, field

from my_library import BaseItem

@attrs.define
class CustomItem:
    name: str

@attrs.define
class CustomPage(ItemPage[CustomItem]):
    base: BaseItem

    @field
    def name(self) -> str:
        return f"{self.base.brand}: {self.base.name}"
```

This gives you the flexibility to use *rules* to set the page object class to use when building the item. Also, item fields can be read from synchronous methods even if the source page object fields were *asynchronous*.

On the other hand, all fields of the source page object class will always be called to build the entire item, which may be a waste of resources if you only need to access some of the item fields.

9.3.2 Field mixins

You can subclass *web_poet.fields.FieldsMixin* to create a mixin to reuse field definitions across multiple, otherwise-unrelated classes. For example:

```
import attrs
from web_poet import ItemPage, field
from web_poet.fields import FieldsMixin

from my_library import BaseItem1, BaseItem2

@attrs.define
class CustomItem:
    name: str


class NameMixin(FieldsMixin):
    @field
    def name(self) -> str:
        return f"{self.base.brand}: {self.base.name}"


@attrs.define
class CustomPage1(NameMixin, ItemPage[CustomItem]):
    base: BaseItem1


@attrs.define
class CustomPage2(NameMixin, ItemPage[CustomItem]):
    base: BaseItem2
```

9.4 Field processors

It's often needed to clean or process field values using reusable functions. *@field* takes an optional `out` argument with a list of such functions. They will be applied to the field value before returning it:

```
from web_poet import ItemPage, HttpResponse, field

def clean_tabs(s: str) -> str:
    return s.replace('\t', ' ')

def add_brand(s: str, page: ItemPage) -> str:
    return f"{page.brand} - {s}"

class MyPage(ItemPage):
```

(continues on next page)

(continued from previous page)

```
    response: HttpResponse

    @field(out=[clean_tabs, str.strip, add_brand])
    def name(self) -> str:
        return self.response.css(".name ::text").get() or ""

    @field(cached=True)
    def brand(self) -> str:
        return self.response.css(".brand ::text").get() or ""
```

9.4.1 Accessing other fields from field processors

If a processor takes an argument named `page`, that argument will contain the page object instance. This allows processing a field differently based on the values of other fields.

Be careful of circular references. Accessing a field runs its processors; if two fields reference each other, `RecursionError` will be raised.

You should enable *caching* for fields accessed in processors, to avoid unnecessary recomputation.

Processors can be applied to asynchronous fields, but processor functions must be synchronous. As a result, only values of synchronous fields can be accessed from processors through the `page` argument.

9.4.2 Default processors

In addition to the `out` argument of *`@field`*, you can define processors at the page object class level by defining a nested class named `Processors`:

```
import attrs
from web_poet import ItemPage, HttpResponse, field

def clean_tabs(s: str) -> str:
    return s.replace('\t', ' ')

@attrs.define
class MyPage(ItemPage):
    response: HttpResponse

    class Processors:
        name = [clean_tabs, str.strip]

    @field
    def name(self) -> str:
        return self.response.css(".name ::text").get() or ""
```

If `Processors` contains an attribute with the same name as a field, the value of that attribute is used as a list of default processors for the field, to be used if the `out` argument of *`@field`* is not defined.

You can also reuse and extend the processors defined in a base class by explicitly accessing or subclassing the `Processors` class:

```
import attrs
from web_poet import ItemPage, HttpResponse, field

def clean_tabs(s: str) -> str:
    return s.replace('\t', ' ')

@attrs.define
class MyPage(ItemPage):
    response: HttpResponse

    class Processors:
        name = [str.strip]

    @field
    def name(self) -> str:
        return self.response.css(".name ::text").get() or ""

class MyPage2(MyPage):
    class Processors(MyPage.Processors):
        # name uses the processors in MyPage.Processors.name
        # description now also uses them and also clean_tabs
        description = MyPage.Processors.name + [clean_tabs]

    @field
    def description(self) -> str:
        return self.response.css(".description ::text").get() or ""

    # brand uses the same processors as name
    @field(out=MyPage.Processors.name)
    def brand(self) -> str:
        return self.response.css(".brand ::text").get() or ""
```

9.4.3 Processors for nested fields

Some item fields contain nested items (e.g. a product can contain a list of variants) and it's useful to have processors for fields of these nested items.

You can use the same logic for them as for normal fields if you define an extractor class that produces these nested items. Such classes should inherit from *Extractor*.

In the simplest cases you need to pass a selector to them:

```
from typing import Any, Dict, List

import attrs
from parsel import Selector
from web_poet import Extractor, ItemPage, HttpResponse, field

@attrs.define
class MyPage(ItemPage):
    response: HttpResponse
```

(continues on next page)

(continued from previous page)

```
    @field
    async def variants(self) -> List[Dict[str, Any]]:
        variants = []
        for color_sel in self.response.css(".color"):
            variant = await VariantExtractor(color_sel).to_item()
            variants.append(variant)
        return variants

@attrs.define
class VariantExtractor(Extractor):
    sel: Selector

    @field(out=[str.strip])
    def color(self) -> str:
        return self.sel.css(".name::text").get() or ""
```

In such cases you can also use *SelectorExtractor* as a shortcut that provides `css()` and `xpath()`:

```
class VariantExtractor(SelectorExtractor):
    @field(out=[str.strip])
    def color(self) -> str:
        return self.css(".name::text").get() or ""
```

You can also pass other data in addition to, or instead of, selectors, such as dictionaries with some data:

```
@attrs.define
class VariantExtractor(Extractor):
    variant_data: dict

    @field(out=[str.strip])
    def color(self) -> str:
        return self.variant_data.get("color") or ""
```

9.5 Field caching

When writing extraction code for Page Objects, it's common that several attributes reuse some computation. For example, you might need to do an additional request to get an API response, and then fill several attributes from this response:

```
from typing import Dict, Optional

from web_poet import ItemPage, HttpResponse, HttpClient, validates_input

class MyPage(ItemPage):
    response: HttpResponse
    http: HttpClient

    @validates_input
    async def to_item(self) -> Dict[str, Optional[str]]:
        api_url = self.response.css("...").get()
```

(continues on next page)

(continued from previous page)

```
        api_response = await self.http.get(api_url).json()
        return {
            'name': self.response.css(".name ::text").get(),
            'price': api_response.get("price"),
            'sku': api_response.get("sku"),
        }
```

When converting such Page Objects to use fields, be careful not to make an API call (or some other heavy computation) multiple times. You can do it by extracting the heavy operation to a method, and caching the results:

```
from typing import Dict

from web_poet import ItemPage, HttpResponse, HttpClient, field, cached_method

class MyPage(ItemPage):
    response: HttpResponse
    http: HttpClient

    @cached_method
    async def api_response(self) -> Dict[str, str]:
        api_url = self.response.css("...").get()
        return await self.http.get(api_url).json()

    @field
    def name(self) -> str:
        return self.response.css(".name ::text").get() or ""

    @field
    async def price(self) -> str:
        api_response = await self.api_response()
        return api_response.get("price") or ""

    @field
    async def sku(self) -> str:
        api_response = await self.api_response()
        return api_response.get("sku") or ""
```

As you can see, `web-poet` provides *`cached_method()`* decorator, which allows to memoize the function results. It supports both sync and async methods, i.e. you can use it on regular methods (`def foo(self)`), as well as on async methods (`async def foo(self)`).

The refactored example, with per-attribute fields, is more verbose than the original one, where a single `to_item` method is used. However, it provides some advantages — if only a subset of attributes is needed, then it's possible to use the Page Object without doing unnecessary work. For example, if user only needs `name` field in the example above, no additional requests (API calls) will be made.

Sometimes you might want to cache a `@field`, i.e. a property which computes an attribute of the final item. In such cases, use `@field(cached=True)` decorator instead of `@field`.

9.5.1 cached_method vs lru_cache vs cached_property

If you're an experienced Python developer, you might wonder why is *cached_method()* decorator needed, if Python already provides functools.lru_cache(). For example, one can write this:

```
from functools import lru_cache
from web_poet import ItemPage

class MyPage(ItemPage):
    # ...
    @lru_cache
    def heavy_method(self):
        # ...
```

Don't do it! There are two issues with functools.lru_cache(), which make it unsuitable here:

1. It doesn't work properly on methods, because `self` is used as a part of the cache key. It means a reference to an instance is kept in the cache, and so created page objects are never deallocated, causing a memory leak.
2. functools.lru_cache() doesn't work on `async def` methods, so you can't cache e.g. results of API calls using functools.lru_cache().

cached_method() solves both of these issues. You may also use functools.cached_property(), or an external package like async_property with async versions of `@property` and `@cached_property` decorators; unlike functools.lru_cache(), they all work fine for this use case.

9.5.2 Exception caching

Note that exceptions are not cached - neither by *cached_method()*, nor by *@field(cached=True)*, nor by functools.lru_cache(), nor by functools.cached_property().

Usually it's not an issue, because an exception is usually propagated, and so there are no duplicate calls anyways. But, just in case, keep this in mind.

9.6 Field metadata

`web-poet` allows to store arbitrary information for each field using the `meta` keyword argument:

```
from web_poet import ItemPage, field

class MyPage(ItemPage):

    @field(meta={"expensive": True})
    async def my_field(self):
        ...
```

To retrieve this information, use *web_poet.fields.get_fields_dict()*; it returns a dictionary, where keys are field names, and values are *web_poet.fields.FieldInfo* instances.

```
from web_poet.fields import get_fields_dict

fields_dict = get_fields_dict(MyPage)
field_names = fields_dict.keys()
```

(continues on next page)

(continued from previous page)

```
my_field_meta = fields_dict["my_field"].meta

print(field_names)  # dict_keys(['my_field'])
print(my_field_meta)  # {'expensive': True}
```

9.7 Input validation

Input validation, if used, happens before field evaluation, and it may override the values of fields, preventing field evaluation from ever happening. For example:

```
class Page(ItemPage[Item]):
    def validate_input(self) -> Item:
        return Item(foo="bar")

    @field
    def foo(self):
        raise RuntimeError("This exception is never raised")

assert Page().foo == "bar"
```

Field evaluation may still happen for a field if the field is used in the implementation of the `validate_input` method. Note, however, that only synchronous fields can be used from the `validate_input` method.

CHAPTER

TEN

ADDITIONAL REQUESTS

Some websites require page interactions to load some information, such as clicking a button, scrolling down or hovering on some element. These interactions usually trigger background requests that are then loaded using JavaScript.

To extract such data, reproduce those requests using *HttpClient*. Include *HttpClient* among the *inputs* of your *page object*, and use an asynchronous *field* or method to call one of its methods.

For example, simulating a click on a button that loads product images could look like:

```
import attrs
from web_poet import HttpClient, HttpError, field
from zyte_common_items import Image, ProductPage


@attrs.define
class MyProductPage(ProductPage]):
    http: HttpClient

    @field
    def productId(self):
        return self.css("::attr(product-id)").get()

    @field
    async def images(self):
        url = f"https://api.example.com/v2/images?id={self.productId}"
        try:
            response = await self.http.get(url)
        except HttpError:
            return []
        else:
            urls = response.css(".product-images img::attr(src)").getall()
            return [Image(url=url) for url in urls]
```

Warning: *HttpClient* should only be used to handle the type of scenarios mentioned above. Using *HttpClient* for crawling logic would defeat *the purpose of web-poet*.

10.1 Making a request

HttpClient provides multiple asynchronous request methods, such as:

```
http = HtpClient()
response = await http.get(url)
response = await http.post(url, body=b"...")
response = await http.request(url, method="...")
response = await http.execute(HttpRequest(url, method="..."))
```

Request methods also accept custom headers and body, for example:

```
http.post(
    url,
    headers={"Content-Type": "application/json;charset=UTF-8"},
    body=json.dumps({"foo": "bar"}).encode("utf-8"),
)
```

Request methods may either raise an *HttpError* or return an *HttpResponse*. See *Working with HttpResponse*.

Note: *HttpClient* methods are expected to follow any redirection except when the request method is `HEAD`. This means that the *HttpResponse* that you get is already the end of any redirection trail.

10.2 Concurrent requests

To send multiple requests concurrently, use *HttpClient.batch_execute*, which accepts any number of *HttpRequest* instances as input, and returns *HttpResponse* instances (and *HttpError* instances when using `return_exceptions=True`) in the input order. For example:

```
import attrs
from web_poet import HttpClient, HttpError, HttpRequest, field
from zyte_common_items import Image, ProductPage, ProductVariant


@attrs.define
class MyProductPage(ProductPage):
    http: HttpClient

    max_variants = 10

    @field
    def productId(self):
        return self.css("::attr(product-id)").get()

    @field
    async def variants(self):
        requests = [
            HttpRequest(f"https://example.com/api/variant/{self.productId}/{index}")
            for index in range(self.max_variants)
        ]
```

(continues on next page)

(continued from previous page)

```
        responses = await self.http.batch_execute(*requests, return_exceptions=True)
        return [
            ProductVariant(color=response.css("::attr(color)").get())
            for response in responses
            if not isinstance(response, HttpError)
        ]
```

You can alternatively use `asyncio` together with *`HttpClient`* to handle multiple requests. For example, you can use `asyncio.as_completed()` to process the first response from a group of requests as early as possible.

10.3 Error handling

`HttpClient` methods may raise an exception of type *`HttpError`* or a subclass.

If the response HTTP status code (*`response.status`*) is 400 or higher, *`HttpResponseError`* is raised. In case of connection errors, TLS errors and similar, *`HttpRequestError`* is raised.

`HttpError` provides access to the offending *`request`*, and *`HttpResponseError`* also provides access to the offending *`response`*.

10.4 Retrying additional requests

Input validation allows retrying all inputs from a page object. To retry only additional requests, you must handle retries on your own.

Your code is responsible for retrying additional requests until good response data is received, or until some maximum number of retries is exceeded.

It is up to you to decide what the maximum number of retries should be for a given additional request, based on your experience with the target website.

It is also up to you to decide how to implement retries of additional requests.

One option would be tenacity. For example, to try an additional request 3 times before giving up:

```
import attrs
from tenacity import retry, stop_after_attempt
from web_poet import HttpClient, HttpError, field
from zyte_common_items import ProductPage


@attrs.define
class MyProductPage(ProductPage):
    http: HttpClient

    @field
    def productId(self):
        return self.css("::attr(product-id)").get()

    @retry(stop=stop_after_attempt(3))
    async def get_images(self):
        return self.http.get(f"https://api.example.com/v2/images?id={self.productId}")
```

(continues on next page)

(continued from previous page)

```
    @field
    async def images(self):
        try:
            response = await self.get_images()
        except HttpError:
            return []
        else:
            urls = response.css(".product-images img::attr(src)").getall()
            return [Image(url=url) for url in urls]
```

If the reason your additional request fails is outdated or missing data from page object input, do not try to reproduce the request for that input as an additional request. *Request fresh input instead*.

CHAPTER

ELEVEN

INPUT VALIDATION

Sometimes the data that your page object receives as input may be invalid.

You can define a `validate_input` method in a page object class to check its input data and determine how to handle invalid input.

`validate_input` is called on the first execution of `ItemPage.to_item()` or the first access to a *field*. In both cases validation happens early; in the case of fields, it happens before field evaluation.

`validate_input` is a synchronous method that expects no parameters, and its outcome may be any of the following:

- Return `None`, indicating that the input is valid.
- Raise *`Retry`*, indicating that the input looks like the result of a temporary issue, and that trying to fetch similar input again may result in valid input.

  See also *Retrying additional requests*.
- Raise *`UseFallback`*, indicating that the page object does not support the input, and that an alternative parsing implementation should be tried instead.

  For example, imagine you have a page object for website commerce.example, and that commerce.example is built with a popular e-commerce web framework. You could have a generic page object for products of websites using that framework, `FrameworkProductPage`, and a more specific page object for commerce.example, `EcommerceExampleProductPage`. If `EcommerceExampleProductPage` cannot parse a product page, but it looks like it might be a valid product page, you would raise *`UseFallback`* to try to parse the same product page with `FrameworkProductPage`, in case it works.

  Note: web-poet does not dictate how to define or use an alternative parsing implementation as fallback. It is up to web-poet frameworks to choose how they implement fallback handling.

- Return an item to override the output of the `to_item` method and of fields.

  For input not matching the expected type of data, returning an item that indicates so is recommended.

  For example, if your page object parses an e-commerce product, and the input data corresponds to a list of products rather than a single product, you could return a product item that somehow indicates that it is not a valid product item, such as `Product(is_valid=False)`.

For example:

```
def validate_input(self):
    if self.css('.product-id::text') is not None:
        return
    if self.css('.http-503-error'):
        raise Retry()
```

(continues on next page)

(continued from previous page)

```
    if self.css('.product'):
        raise UseFallback()
    if self.css('.product-list'):
        return Product(is_valid=False)
```

You may use fields in your implementation of the `validate_input` method, but only synchronous fields are supported. For example:

```
class Page(WebPage[Item]):
    def validate_input(self):
        if not self.name:
            raise UseFallback()

    @field(cached=True)
    def name(self):
        return self.css(".product-name ::text")
```

Tip: *Cache fields* used in the `validate_input` method, so that when they are used from `to_item` they are not evaluated again.

If you implement a custom `to_item` method, as long as you are inheriting from *ItemPage*, you can enable input validation decorating your custom `to_item` method with `validates_input()`:

```
from web_poet import validates_input

class Page(ItemPage[Item]):
    @validates_input
    async def to_item(self):
        ...
```

Retry and *UseFallback* may also be raised from the `to_item` method. This could come in handy, for example, if after you execute some asynchronous code, such as an *additional request*, you find out that you need to retry the original request or use a fallback.

11.1 Input Validation Exceptions

exception web_poet.exceptions.**PageObjectAction**

Base class for exceptions that can be raised from a page object to indicate something to be done about that page object.

exception web_poet.exceptions.**Retry**

The page object found that the input data is partial or empty, and a request retry may provide better input.

exception web_poet.exceptions.**UseFallback**

The page object cannot extract data from the input, but the input seems valid, so an alternative data extraction implementation for the same item type may succeed.

CHAPTER

TWELVE

USING PAGE PARAMS

In some cases, *page object classes* might require or allow parameters from the calling code, e.g. to change their behavior or make optimizations.

To support parameters, add *PageParams* to your *inputs*:

```
import attrs
from web_poet import PageParams, WebPage

@attrs.define
class MyPage(WebPage):
    page_params: PageParams
```

In your page object class, you can read parameters from a *PageParams* object as you would from a `dict`:

```
foo = self.page_params["foo"]
bar = self.page_params.get("bar", "default")
```

The way the calling code sets those parameters depends on your *web-poet framework*.

12.1 Example: Controlling item values

```
import attrs
import web_poet
from web_poet import validates_input


@attrs.define
class ProductPage(web_poet.WebPage):
    page_params: web_poet.PageParams

    default_tax_rate = 0.10

    @validates_input
    def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "price": self.css("#main .price ::text").get(),
        }
```

(continues on next page)

(continued from previous page)

```
        self.calculate_price_with_tax(item)
        return item

    @staticmethod
    def calculate_price_with_tax(item):
        tax_rate = self.page_params.get("tax_rate", self.default_tax_rate)
        item["price_with_tax"] = item["price"] * (1 + tax_rate)
```

From the example above, we were able to provide an optional information regarding the **tax rate** of the product. This could be useful when trying to support the different tax rates for each state or territory. However, since we're treating the **tax_rate** as optional information, notice that we also have a the `default_tax_rate` as a backup value just in case it's not available.

12.2 Example: Controlling page object behavior

Let's try an example wherein *PageParams* is able to control how *additional requests* are being used. Specifically, we are going to use *PageParams* to control the number of pages visited.

```
from typing import List

import attrs
import web_poet
from web_poet import validates_input


@attrs.define
class ProductPage(web_poet.WebPage):
    http: web_poet.HttpClient
    page_params: web_poet.PageParams

    default_max_pages = 5

    @validates_input
    async def to_item(self):
        return {"product_urls": await self.get_product_urls()}

    async def get_product_urls(self) -> List[str]:
        # Simulates scrolling to the bottom of the page to load the next
        # set of items in an "Infinite Scrolling" category list page.
        max_pages = self.page_params.get("max_pages", self.default_max_pages)
        requests = [
            self.create_next_page_request(page_num)
            for page_num in range(2, max_pages + 1)
        ]
        responses = await http.batch_execute(*requests)
        return [
            url
            for response in responses
            for product_urls in self.parse_product_urls(response)
            for url in product_urls
```

(continues on next page)

(continued from previous page)

```
        ]

    @staticmethod
    def create_next_page_request(page_num):
        next_page_url = f"https://example.com/category/products?page={page_num}"
        return web_poet.Request(url=next_page_url)

    @staticmethod
    def parse_product_urls(response: web_poet.HttpResponse):
        return response.css("#main .products a.link ::attr(href)").getall()
```

From the example above, we can see how *PageParams* is able to arbitrarily limit the pagination behavior by passing an optional **max_pages** info. Take note that a `default_max_pages` value is also present in the page object class in case the *PageParams* instance did not provide it.

CHAPTER

THIRTEEN

STATS

During parsing, storing some data about the parsing itself can be useful for debugging, monitoring, and reporting. The *Stats* page input allows storing such data.

For example, you can use stats to track which parsing code is actually used, so that you can remove code once it is no longer necessary due to upstream changes:

```
from attrs import define
from web_poet import field, Stats, WebPage

@attrs.define
class MyPage(WebPage):
    stats: Stats

    @field
    def title(self):
        if title := self.css("h1::text").get():
            self.stats.inc("MyPage/field-src/title/h1")
        elif title := self.css("h2::text").get():
            self.stats.inc("MyPage/field-src/title/h2")
        return title
```

CHAPTER

FOURTEEN

TESTS FOR PAGE OBJECTS

Page Objects that inherit from *ItemPage* can be tested by saving the dependencies needed to create one and the result of *to_item()*, recreating the Page Object from the dependencies, running its *to_item()* and comparing the result to the saved one. `web-poet` provides the following tools for this:

- dependency serialization into a Python object and into a set of files;
- recreating Page Objects from the serialized dependencies;
- a high-level function to save a test fixture;
- a `pytest` plugin that discovers fixtures and runs tests for them.

14.1 Serialization

`web_poet.serialization.serialize()` can be used to serialize an iterable of Page Object dependencies to a Python object. `web_poet.serialization.deserialize()` can be used to recreate a Page Object from this serialized data.

An instance of `web_poet.serialization.SerializedDataFileStorage` can be used to write the serialized data to a set of files in a given directory and to read it back.

Note: We only support serializing dependencies, not Page Object instances, because the only universal way to recreate a Page Object is from its dependencies, not from some saved internal state.

Each dependency is serialized to one or several `bytes` objects, each of which is saved as a single file. `web_poet.serialization.serialize_leaf()` and `web_poet.serialization.deserialize_leaf()` are used to convert between a dependency and this set of `bytes` objects. They are implemented using `functools.singledispatch()` and while the types provided by `web-poet` are supported out of the box, user-defined types need a pair of implementation functions that need to be registered using `web_poet.serialization.register_serialization()`.

14.2 Fixtures

The provided `pytest` plugin expects fixtures in a certain layout. A set of fixtures for a single Page Object should be contained in a directory named as that Page Object fully qualified class name. Each fixture is a directory inside it, that contains data for Page Object inputs and output:

```
fixtures
└── my_project.pages.MyItemPage
    ├── test-1
```

(continues on next page)

(continued from previous page)

```
    │   ├── inputs
    │   │   ├── HttpClient.exists
    │   │   ├── HttpResponse-body.html
    │   │   ├── HttpResponse-info.json
    │   │   └── ResponseUrl.txt
    │   ├── meta.json
    │   └── output.json
    └── test-2
        ├── inputs
        │   ├── HttpClient.exists
        │   ├── HttpClient-0-HttpRequest.info.json
        │   ├── HttpClient-0-HttpResponse.body.html
        │   ├── HttpClient-0-HttpResponse.info.json
        │   ├── HttpClient-1-HttpRequest.body.txt
        │   ├── HttpClient-1-HttpRequest.info.json
        │   ├── HttpClient-1-HttpResponse.body.html
        │   ├── HttpClient-1-HttpResponse.info.json
        │   ├── HttpResponse-body.html
        │   ├── HttpResponse-info.json
        │   └── ResponseUrl.txt
        ├── meta.json
        └── output.json
```

`web_poet.testing.Fixture.save()` can be used to create a fixture inside a Page Object directory from an iterable of dependencies, an output item and an optional metadata dictionary. It can optionally take a name for the fixture directory. By default it uses incrementing names "test-1", "test-2" etc.

Note: `output.json` contains a result of `page_object.to_item()` converted to a dict using the itemadapter library and saved as JSON.

After generating a fixture you can edit `output.json` to modify expected field values and add new fields, which is useful when creating tests for code that isn't written yet or before modifying its behavior.

14.3 scrapy-poet integration

Projects that use the scrapy-poet library can use the Scrapy command provided by it to generate fixtures in a convenient way. It's available starting with scrapy-poet 0.8.0.

14.4 Running tests

The provided `pytest` plugin is automatically registered when `web-poet` is installed, and running `python -m pytest` in a directory containing fixtures will discover them and run tests for them.

By default, the plugin generates:

- a test which checks that `to_item()` doesn't raise an exception (i.e. it can be executed),
- a test per each output attribute of the item,
- an additional test to check that there are no extra attributes in the output.

For example, if your item has 5 attributes, and you created 2 fixtures, pytest will run (5+1+1)*2 = 14 tests. This allows to report failures for individual fields separately.

If `to_item` raises an error, there is no point in running other tests, so they're skipped in this case.

If you prefer less granular test failure reporting, you can use pytest with the `--web-poet-test-per-item` option:

```
python -m pytest --web-poet-test-per-item
```

In this case there is going to be a single test per fixture: if the result is not fully correct, the test fails. So, following the previous example, it'd be 2 tests instead of 14.

14.5 Test-Driven Development

You can follow TDD (Test-Driven Development) approach to develop your page objects. To do so,

1. Generate a fixture (see *scrapy-poet integration*).
2. Populate `output.json` with the correct expected output.
3. Run the tests (see *Running tests*) and update the code until all tests pass. It's convenient to use web-poet *Fields*, and implement extraction field-by-field, because you'll be getting an additional test passing after each field is implemented.

This approach allows a fast feedback loop: there is no need to download page multiple times, and you have a clear progress indication for your work (number of failing tests remaining). Also, in the end you get a regression test, which can be helpful later.

Sometimes it may be awkward to set the correct value in JSON before starting the development, especially if a value is large or has a complex structure. For example, this could be the case for e-commerce product description field, which can be hard to copy-paste from the website, and which may have various whitespace normalization rules which you need to apply.

In this case, it may be more convenient to implement the extraction first, and only then populate the `output.json` file with the correct value.

You can use `python -m web_poet.testing rerun <fixture_path>` command in this case, to re-run the page object using the inputs saved in a fixture. This command prints output of the page object, as JSON; you can then copy-paste relevant parts to the `output.json` file. It's also possible to make the command print only some of the fields. For example, you might run the following command after implementing extraction for "description" and "descriptionHtml" fields in `my_project.pages.MyItemPage`:

```
python -m web_poet.testing rerun \
    fixtures/my_project.pages.MyItemPage/test-1 \
    --fields description,descriptionHtml
```

It may output something like this:

```
{
    "description": "..description of the product..",
    "descriptionHtml": "<p>...</p>"
}
```

If these values look good, you can update `fixtures/my_project.pages.MyItemPage/test-1/output.json` file with these values.

14.6 Handling time fields

Sometimes output of a page object might depend on the current time. For example, the item may contain the scraping datetime, or a current timestamp may be used to build some URLs. When a test runs at a different time it will break. To avoid this *the metadata dictionary* can contain a `frozen_time` field set to the time value used when generating the test. This will instruct the test runner to use the same time value so that field comparisons are still correct.

The value can be any string understood by dateutil. If it doesn't include timezone information, the local time of the machine will be assumed. If it includes timezone information, on non-Windows systems the test process will be executed in that timezone, so that output fields that contain local time are correct. On Windows systems (where changing the process timezone is not possible) the time value will be converted to the local time of the machine, and such fields will containt wrong data if these timezones don't match. Consider an example item:

```
import datetime
from web_poet import WebPage, validates_input

class DateItemPage(WebPage):
    @validates_input
    async def to_item(self) -> dict:
        # e.g. 2001-01-01 11:00:00 +00
        now = datetime.datetime.now(datetime.timezone.utc)
        return {
            # '2001-01-01T11:00:00Z'
            "time_utc": now.strftime("%Y-%M-%dT%H:%M:%SZ"),
            # if the current timezone is CET, then '2001-01-01T12:00:00+01:00'
            "time_local": now.astimezone().strftime("%Y-%M-%dT%H:%M:%S%z"),
        }
```

We will assume that the fixture was generated in CET (UTC+1).

- If the fixture doesn't have the `frozen_time` metadata field, the item will simply contain the current time and the test will always fail.
- If `frozen_time` doesn't contain the timezone data (e.g. it is `2001-01-01T11:00:00`), the item will depend on the machine timezone: in CET it will contain the expected values, in timezones with a different offset `time_local` will be different.
- If `frozen_time` contains the timezone data and the system is not Windows, the `time_local` field will contain the date in that timezone, so if the timezone in `frozen_time` is not UTC+1, the test will fail.
- If the system is Windows, the `frozen_time` value will be converted to the machine timezone, so the item will depend on that timezone, just like when `frozen_time` doesn't contain the timezone data, and `time_local` will similarly be only correct if the machine timezone has the same offset as CET.

This means that most combinations of setups will work if `frozen_time` contains the timezone data, except for running tests on Windows, in which case the machine timezone should match the timezone in `frozen_time`. Also, if items do not depend on the machine timezone (e.g. if all datetime-derived data they contain is in UTC), the tests for them should work everywhere.

There is an additional limitation which we plan to fix in future versions. The time is set to the `frozen_time` value when the test generation (if using the `scrapy-poet` command) or the test run starts, but it ticks during the generation/run itself, so if it takes more than 1 second (which is quite possible even in simple cases) the time fields will have values several seconds later than `frozen_time`. For now we recommend to work around this problem by manually editing the `output.json` file to put the value equal to `frozen_time` in these fields, as running the test shoudn't take more than 1 second.

14.7 Storing fixtures in Git

Fixtures can take a lot of disk space, as they usually include page responses and may include other large files, so we recommend using Git LFS when storing them in Git repos to reduce the repo space and get other performance benefits. Even if your fixtures are currently small, it may be useful to do this from the beginning, as migrating files to LFS is not easy and requires rewriting the repo history.

To use Git LFS you need a Git hosting provider that supports it, and major providers and software (e.g. GitHub, Bitbucket, GitLab) support it. There are also implementations for standalone Git servers.

Assuming you store the fixtures in the directory named "fixtures" in the repo root, the workflow should be as following. Enable normal diffs for LFS files in this repo:

```
git config diff.lfs.textconv cat
```

Enable LFS for the fixtures directory before committing anything in it:

```
git lfs track "fixtures/**"
```

Commit the `.gitattributes` file (which stores the tracking information):

```
git add .gitattributes
git commit
```

After generating the fixtures just commit them as usual:

```
git add fixtures/test-1
git commit
```

After this all usual commands including `push`, `pull` or `checkout` should work as expected on these files.

Please also check the official Git LFS documentation for more information.

14.8 Additional requests support

If the page object uses the *`HttpClient`* dependency to make *additional requests*, the generated fixtures will contain these requests and their responses (or exceptions raised when the response is not received). When the test runs, *`HttpClient`* will return the saved responses without doing actual requests.

Currently requests are compared by their URL, method, headers and body, so if a page object makes requests that differ between runs, the test won't be able to find a saved response and will fail.

14.9 Test coverage

The coverage for page object code is reported correctly if tools such as coverage are used when running web-poet tests.

14.10 Item adapters

The testing framework uses the itemadapter library to convert items to dicts when storing them in fixtures and when comparing the expected and the actual output. As adapters may influence the resulting dicts, it's important to use the same adapter when generating and running the tests.

It may also be useful to use different adapters in tests and in production. For example, you may want to omit empty fields in production, but be able to distinguish between empty and absent fields in tests.

For this you can set the `adapter` field in *the metadata dictionary* to the class that inherits from `itemadapter.ItemAdapter` and has the adapter(s) you want to use in tests in its `ADAPTER_CLASSES` attribute (see the relevant itemadapter docs for more information). An example:

```
from collections import deque

from itemadapter import ItemAdapter
from itemadapter.adapter import DictAdapter


class MyAdapter(DictAdapter):
    # any needed customization
    ...

class MyItemAdapter(ItemAdapter):
    ADAPTER_CLASSES = deque([MyAdapter])
```

You can then put the `MyItemAdapter` class object into `adapter` and it will be used by the testing framework.

If `adapter` is not set, `WebPoetTestItemAdapter` will be used. It works like `itemadapter.ItemAdapter` but doesn't change behavior when `itemadapter.ItemAdapter.ADAPTER_CLASSES` is modified.

CHAPTER

FIFTEEN

FRAMEWORKS

Page objects are not meant to be used in isolation with web-poet. They are meant to be used with a web-poet framework.

A web-poet framework is a Python web scraping framework, library, or plugin that implements the *web-poet specification*.

At the moment, the only production-ready web-poet framework that exists is scrapy-poet, which brings web-poet support to Scrapy.

As web-poet matures and sees wider adoption, we hope to see more frameworks add support for it.

CHAPTER

SIXTEEN

FRAMEWORK SPECIFICATION

Learn how to build a *web-poet framework*.

16.1 Design principles

Page objects should be flexible enough to be used with:

- synchronous or asynchronous code, callback-based and `async def / await` based,
- single-node and distributed systems,
- different underlying HTTP implementations - or without HTTP support at all, etc.

16.2 Minimum requirements

A web-poet framework must support building a *page object* given a page object class.

It must be able to build *input objects* for a page object based on type hints on the page object class, i.e. dependency injection, and additional input data required by those input objects, such as a target URL or a dictionary of *page parameters*.

You can implement dependency injection with the andi library, which handles signature inspection, `Optional` and `Union` annotations, as well as indirect dependencies. For practical examples, see the source code of scrapy-poet and of the *`web_poet.example`* module.

16.3 Additional features

To provide a better experience to your users, consider extending your web-poet framework further to:

- Support as many input classes from the *`web_poet.page_inputs`* module as possible.
- Support returning a *page object* given a target URL and a desired *output item class*, determining the right *page object class* to use based on *rules*.
- Allow users to request an *output item* directly, instead of requesting a page object just to call its `to_item` method.

  If you do, consider supporting both synchronous and asynchronous definitions of the `to_item` method, e.g. using *`ensure_awaitable()`*.
- Support *additional requests*.
- Support *retries*.

- Let users set their own *rules*, e.g. to *solve conflicts*.

CHAPTER

SEVENTEEN

SUPPORTING RULES

Ideally, a framework should support returning the right *page object* or *output item* given a target URL and a desired *output item class* when *rules* are used.

To provide basic support for rules in your framework, use the *RulesRegistry* object at `web_poet.default_registry` to choose a page object based on rules:

```
from web_poet import default_registry

page_cls = default_registry.page_cls_for_item("https://example.com", MyItem)
```

You should also let your users know what is the best approach to *load rules* when using your framework. For example, let them know the best location for their calls to the *consume_modules()* function.

CHAPTER

EIGHTEEN

SUPPORTING ADDITIONAL REQUESTS

To support *additional requests*, your framework must provide the request download implementation of *HttpClient*.

18.1 Providing the Downloader

On its own, *HttpClient* doesn't do anything. It doesn't know how to execute the request on its own. Thus, for frameworks or projects wanting to use additional requests in Page Objects, they need to set the implementation on how to execute an *HttpRequest*.

For more info on this, kindly read the API Specifications for *HttpClient*.

In any case, frameworks that wish to support **web-poet** could provide the HTTP downloader implementation in two ways:

18.1.1 1. Context Variable

`contextvars` is natively supported in `asyncio` in order to set and access context-aware values. This means that the framework using **web-poet** can assign the request downloader implementation using the `contextvars` instance named `web_poet.request_downloader_var`.

This can be set using:

```
import attrs
import web_poet
from web_poet import validates_input

async def request_implementation(req: web_poet.HttpRequest) -> web_poet.HttpResponse:
    ...


def create_http_client():
    return web_poet.HttpClient()


@attrs.define
class SomePage(web_poet.WebPage):
    http: web_poet.HttpClient

    @validates_input
    async def to_item(self):
```

(continues on next page)

(continued from previous page)

```
        ...

# Once this is set, the ``request_implementation`` becomes available to
# all instances of HttpClient, unless HttpClient is created with
# the ``request_downloader`` argument (see the #2 Dependency Injection
# example below).
web_poet.request_downloader_var.set(request_implementation)

# Assume that it's constructed with the necessary arguments taken somewhere.
response = web_poet.HttpResponse(...)

page = SomePage(response=response, http=create_http_client())
item = await page.to_item()
```

When the `web_poet.request_downloader_var` contextvar is set, *`HttpClient`* instances use it by default.

Warning: If no value for `web_poet.request_downloader_var` is set, then *`RequestDownloaderVarError`* is raised. However, no exception is raised if **option 2** below is used.

18.1.2 2. Dependency Injection

The framework using **web-poet** may be using libraries that don't have a full support to `contextvars` *(e.g. Twisted)*. With that, an alternative approach would be to supply the request downloader implementation when creating an *`HttpClient`* instance:

```
import attrs
import web_poet
from web_poet import validates_input

async def request_implementation(req: web_poet.HttpRequest) -> web_poet.HttpResponse:
    ...

def create_http_client():
    return web_poet.HttpClient(request_downloader=request_implementation)


@attrs.define
class SomePage(web_poet.WebPage):
    http: web_poet.HttpClient

    @validates_input
    async def to_item(self):
        ...

# Assume that it's constructed with the necessary arguments taken somewhere.
response = web_poet.HttpResponse(...)

page = SomePage(response=response, http=create_http_client())
item = await page.to_item()
```

From the code sample above, we can see that every time an *HttpClient* instance is created for Page Objects needing it, the framework must create *HttpClient* with a framework-specific **request downloader implementation**, using the `request_downloader` argument.

18.2 Downloader Behavior

The request downloader MUST accept an instance of *HttpRequest* as the input and return an instance of *HttpResponse*. This is important in order to handle and represent generic HTTP operations. The only time that it won't be returning *HttpResponse* would be when it's raising exceptions (see *Exception Handling*).

The request downloader MUST resolve Location-based **redirections** when the HTTP method is not `HEAD`. In other words, for non-`HEAD` requests the returned *HttpResponse* must be the final response, after all redirects. For `HEAD` requests redirects MUST NOT be resolved.

Lastly, the request downloader function MUST support the `async/await` syntax.

18.3 Exception Handling

Page Object developers could use the exception classes built inside **web-poet** to handle various ways additional requests MAY fail. In this section, we'll see the rationale and ways the framework MUST be able to do that.

18.3.1 Rationale

Frameworks that handle **web-poet** MUST be able to ensure that Page Objects having additional requests using *HttpClient* are able to work with any type of HTTP downloader implementation.

For example, in Python, the common HTTP libraries have different types of base exceptions when something has occurred:

- aiohttp.ClientError
- requests.RequestException
- urllib.error.HTTPError

Imagine if Page Objects are **expected** to work in *different* backend implementations like the ones above, then it would cause the code to look like:

```
import urllib

import aiohttp
import attrs
import requests
import web_poet
from web_poet import validates_input


@attrs.define
class SomePage(web_poet.WebPage):
    http: web_poet.HttpClient

    @validates_input
    async def to_item(self):
```

(continues on next page)

(continued from previous page)

```
        try:
            response = await self.http.get("...")
        except (aiohttp.ClientError, requests.RequestException, urllib.error.HTTPError):
            # handle the error here
```

Such code could turn messy in no time especially when the number of HTTP backends that Page Objects have to support are steadily increasing. Not to mention the plethora of exception types that HTTP libraries have. This means that Page Objects aren't truly portable in different types of frameworks or environments. Rather, they're only limited to work in the specific framework they're supported.

In order for Page Objects to work in different Downloader Implementations, the framework that implements the HTTP Downloader backend MUST raise exceptions from the *web_poet.exceptions.http* module in lieu of the backend specific ones *(e.g. aiohttp, requests, urllib, etc.)*.

This makes the code simpler:

```
import attrs
import web_poet
from web_poet import validates_input


@attrs.define
class SomePage(web_poet.WebPage):
    http: web_poet.HttpClient

    @validates_input
    async def to_item(self):
        try:
            response = await self.http.get("...")
        except web_poet.exceptions.HttpError:
            # handle the error here
```

18.3.2 Expected behavior for Exceptions

All exceptions that the HTTP Downloader Implementation (see *Providing the Downloader* doc section) explicitly raises when implementing it for **web-poet** MUST be *web_poet.exceptions.http.HttpError (or a subclass from it)*.

For frameworks that implement and use **web-poet**, exceptions that occurred when handling the additional requests like *connection errors*, *TLS errors*, etc MUST be replaced by *web_poet.exceptions.http.HttpRequestError* by raising it explicitly.

For responses that are not really errors like in the `100-3xx` status code range, exception MUST NOT be raised at all. For responses with status codes in the `400-5xx` range, **web-poet** raises the *web_poet.exceptions.http.HttpResponseError* exception.

From this distinction, the framework MUST NOT raise *web_poet.exceptions.http.HttpResponseError* on its own at all, since the *HttpClient* already handles that.

CHAPTER

NINETEEN

SUPPORTING RETRIES

Web-poet frameworks must catch *Retry* exceptions raised from the *to_item()* method of a page object.

When *Retry* is caught:

1. The original request whose response was fed into the page object must be retried.
2. A new page object must be created, of the same type as the original page object, and with the same input, except for the response data, which must be the new response.

The *to_item()* method of the new page object may raise *Retry* again. Web-poet frameworks must allow multiple retries of page objects, repeating the *Retry*-capturing logic.

However, web-poet frameworks are also encouraged to limit the amount of retries per page object. When retries are exceeded for a given page object, the page object output is ignored. At the moment, web-poet does not enforce any specific maximum number of retries on web-poet frameworks.

CHAPTER

TWENTY

SUPPORTING STATS

To support *stats*, your framework must provide the `StatCollector` implementation of `Stats`.

It is up to you to decide how to store the stats, and how your users can access them at run time (outside page objects) or afterwards.

CHAPTER

TWENTYONE

API REFERENCE

21.1 Page Inputs

class web_poet.page_inputs.browser.**BrowserHtml**

Bases: SelectableMixin, str

HTML returned by a web browser, i.e. snapshot of the DOM tree in HTML format.

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of `parsel.selector.Selector`.

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

class web_poet.page_inputs.browser.**BrowserResponse**(*url: Union[str, _Url]*, *html*, ***, *status: Optional[int] = None*)

Bases: SelectableMixin, UrlShortcutsMixin

Browser response: url, HTML and status code.

`url` should be browser's window.location, not a URL of the request, if possible.

`html` contains the HTML returned by the browser, i.e. a snapshot of DOM tree in HTML format.

The following are optional since it would depend on the source of the `BrowserResponse` if these are available or not:

`status` should represent the int status code of the HTTP response.

url: ResponseUrl

html: *BrowserHtml*

status: Optional[int]

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of parsel.selector.Selector.

urljoin(*url: Union[str,* RequestUrl*,* ResponseUrl*]*) → *RequestUrl*

Return *url* as an absolute URL.

If *url* is relative, it is made absolute relative to the base URL of *self*.

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

class web_poet.page_inputs.client.**HttpClient**(*request_downloader: Callable = None, *, save_responses: bool = False, return_only_saved_responses: bool = False, responses: Optional[Iterable[_SavedResponseData]] = None*)

Async HTTP client to be used in Page Objects.

See *Additional requests* for the usage information.

HttpClient doesn't make HTTP requests on itself. It uses either the request function assigned to the `web_poet.request_downloader_var` `contextvar`, or a function passed via `request_downloader` argument of the `__init__()` method.

Either way, this function should be an `async def` function which receives an *HttpRequest* instance, and either returns a *HttpResponse* instance, or raises a subclass of *HttpError*. You can read more in the *Providing the Downloader* documentation.

async request(*url: Union[str, _Url], *, method: str = 'GET', headers: Optional[Union[Dict[str, str],* HttpRequestHeaders*]] = None, body: Optional[Union[bytes,* HttpRequestBody*]] = None, allow_status: Union[str, int, List[Union[str, int]]] = None*) → *HttpResponse*

This is a shortcut for creating an *HttpRequest* instance and executing that request.

HttpRequestError is raised for *connection errors*, *connection and read timeouts*, etc.

An *HttpResponse* instance is returned for successful responses in the `100-3xx` status code range.

Otherwise, an exception of type *HttpResponseError* is raised.

Rasing *HttpResponseError* can be suppressed for certain status codes using the `allow_status` param - it is a list of status code values for which *HttpResponse* should be returned instead of raising *HttpResponseError*.

There is a special "*" `allow_status` value which allows any status code.

There is no need to include `100-3xx` status codes in `allow_status`, because *HttpResponseError* is not raised for them.

async get(*url: Union[str, _Url], *, headers: Optional[Union[Dict[str, str],* HttpRequestHeaders*]] = None, allow_status: Union[str, int, List[Union[str, int]]] = None*) → *HttpResponse*

Similar to *request()* but peforming a `GET` request.

async post(*url: Union[str, _Url], *, headers: Optional[Union[Dict[str, str],* HttpRequestHeaders*]] = None, body: Optional[Union[bytes,* HttpRequestBody*]] = None, allow_status: Union[str, int, List[Union[str, int]]] = None*) → *HttpResponse*

Similar to *request()* but performing a `POST` request.

async execute(*request:* HttpRequest*, *, allow_status: Union[str, int, List[Union[str, int]]] = None*) → *HttpResponse*

Execute the specified *HttpRequest* instance using the request implementation configured in the *HttpClient* instance.

HttpRequestError is raised for *connection errors*, *connection and read timeouts*, etc.

HttpResponse instance is returned for successful responses in the `100-3xx` status code range.

Otherwise, an exception of type *HttpResponseError* is raised.

Rasing *HttpResponseError* can be suppressed for certain status codes using the `allow_status` param - it is a list of status code values for which *HttpResponse* should be returned instead of raising *HttpResponseError*.

There is a special "*" `allow_status` value which allows any status code.

There is no need to include `100-3xx` status codes in `allow_status`, because *HttpResponseError* is not raised for them.

async batch_execute(**requests:* HttpRequest, *return_exceptions: bool = False*, *allow_status: Union[str, int, List[Union[str, int]]] = None*) → List[Union[*HttpResponse*, *HttpResponseError*]]

Similar to *execute()* but accepts a collection of *HttpRequest* instances that would be batch executed.

The order of the `HttpResponses` would correspond to the order of *HttpRequest* passed.

If any of the *HttpRequest* raises an exception upon execution, the exception is raised.

To prevent this, the actual exception can be returned alongside any successful *HttpResponse*. This enables salvaging any usable responses despite any possible failures. This can be done by setting `True` to the `return_exceptions` parameter.

Like *execute()*, *HttpResponseError* will be raised for responses with status codes in the `400-5xx` range. The `allow_status` parameter could be used the same way here to prevent these exceptions from being raised.

You can omit `allow_status="*"` if you're passing `return_exceptions=True`. However, it would be returning *HttpResponseError* instead of *HttpResponse*.

Lastly, a *HttpRequestError* may be raised on cases like *connection errors*, *connection and read timeouts*, etc.

get_saved_responses() → Iterable[_SavedResponseData]

Return saved requests and responses.

class web_poet.page_inputs.http.**RequestUrl**(**args*, ***kwargs*)

Bases: `RequestUrl`

class web_poet.page_inputs.http.**ResponseUrl**(**args*, ***kwargs*)

Bases: `ResponseUrl`

class web_poet.page_inputs.http.**HttpRequestBody**

Bases: `bytes`

A container for holding the raw HTTP request body in bytes format.

class web_poet.page_inputs.http.**HttpResponseBody**

Bases: `bytes`

A container for holding the raw HTTP response body in bytes format.

bom_encoding() → Optional[str]

Returns the encoding from the byte order mark if present.

declared_encoding() → Optional[str]

Return the encoding specified in meta tags in the html body, or `None` if no suitable encoding was found

json() → Any

Deserialize a JSON document to a Python object.

class web_poet.page_inputs.http.**HttpRequestHeaders**

Bases: _HttpHeaders

A container for holding the HTTP request headers.

It's able to accept instantiation via an Iterable of Tuples:

```
>>> pairs = [("Content-Encoding", "gzip"), ("content-length", "648")]
>>> HttpRequestHeaders(pairs)
<HttpRequestHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

It's also accepts a mapping of key-value pairs as well:

```
>>> pairs = {"Content-Encoding": "gzip", "content-length": "648"}
>>> headers = HttpRequestHeaders(pairs)
>>> headers
<HttpRequestHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

Note that this also supports case insensitive header-key lookups:

```
>>> headers.get("content-encoding")
'gzip'
>>> headers.get("Content-Length")
'648'
```

These are just a few of the functionalities it inherits from `multidict.CIMultiDict`. For more info on its other features, read the API spec of `multidict.CIMultiDict`.

copy()

Return a copy of itself.

classmethod from_bytes_dict(*arg: Dict[AnyStr, Union[AnyStr, List, Tuple[AnyStr, ...]]]*, *encoding: str = 'utf-8'*) → T_headers

An alternative constructor for instantiation where the header-value pairs could be in raw bytes form.

This supports multiple header values in the form of `List[bytes]` and `Tuple[bytes]]` alongside a plain `bytes` value. A value in `str` also works and wouldn't break the decoding process at all.

By default, it converts the `bytes` value using "utf-8". However, this can easily be overridden using the `encoding` parameter.

```
>>> raw_values = {
...     b"Content-Encoding": [b"gzip", b"br"],
...     b"Content-Type": [b"text/html"],
...     b"content-length": b"648",
... }
>>> headers = _HttpHeaders.from_bytes_dict(raw_values)
>>> headers
<_HttpHeaders('Content-Encoding': 'gzip', 'Content-Encoding': 'br', 'Content-
↪Type': 'text/html', 'content-length': '648')>
```

classmethod from_name_value_pairs(*arg: List[Dict]*) → T_headers

An alternative constructor for instantiation using a `List[Dict]` where the 'key' is the header name while the 'value' is the header value.

```
>>> pairs = [
...     {"name": "Content-Encoding", "value": "gzip"},
...     {"name": "content-length", "value": "648"}
... ]
>>> headers = _HttpHeaders.from_name_value_pairs(pairs)
>>> headers
<_HttpHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

class web_poet.page_inputs.http.**HttpResponseHeaders**

Bases: `_HttpHeaders`

A container for holding the HTTP response headers.

It's able to accept instantiation via an Iterable of Tuples:

```
>>> pairs = [("Content-Encoding", "gzip"), ("content-length", "648")]
>>> HttpResponseHeaders(pairs)
<HttpResponseHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

It's also accepts a mapping of key-value pairs as well:

```
>>> pairs = {"Content-Encoding": "gzip", "content-length": "648"}
>>> headers = HttpResponseHeaders(pairs)
>>> headers
<HttpResponseHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

Note that this also supports case insensitive header-key lookups:

```
>>> headers.get("content-encoding")
'gzip'
>>> headers.get("Content-Length")
'648'
```

These are just a few of the functionalities it inherits from `multidict.CIMultiDict`. For more info on its other features, read the API spec of `multidict.CIMultiDict`.

declared_encoding() → Optional[str]

Return encoding detected from the Content-Type header, or None if encoding is not found

copy()

Return a copy of itself.

classmethod from_bytes_dict(*arg: Dict[AnyStr, Union[AnyStr, List, Tuple[AnyStr, ...]]]*, *encoding: str = 'utf-8'*) → T_headers

An alternative constructor for instantiation where the header-value pairs could be in raw bytes form.

This supports multiple header values in the form of `List[bytes]` and `Tuple[bytes]` alongside a plain `bytes` value. A value in `str` also works and wouldn't break the decoding process at all.

By default, it converts the `bytes` value using "utf-8". However, this can easily be overridden using the `encoding` parameter.

```
>>> raw_values = {
...     b"Content-Encoding": [b"gzip", b"br"],
...     b"Content-Type": [b"text/html"],
...     b"content-length": b"648",
```

(continues on next page)

(continued from previous page)

```
... }
>>> headers = _HttpHeaders.from_bytes_dict(raw_values)
>>> headers
<_HttpHeaders('Content-Encoding': 'gzip', 'Content-Encoding': 'br', 'Content-
↪Type': 'text/html', 'content-length': '648')>
```

classmethod from_name_value_pairs(*arg: List[Dict]*) → T_headers

An alternative constructor for instantiation using a `List[Dict]` where the 'key' is the header name while the 'value' is the header value.

```
>>> pairs = [
...     {"name": "Content-Encoding", "value": "gzip"},
...     {"name": "content-length", "value": "648"}
... ]
>>> headers = _HttpHeaders.from_name_value_pairs(pairs)
>>> headers
<_HttpHeaders('Content-Encoding': 'gzip', 'content-length': '648')>
```

class web_poet.page_inputs.http.**HttpRequest**(*url: Union[str, _Url]*, ***, *method: str = 'GET'*, *headers=_Nothing.NOTHING*, *body=_Nothing.NOTHING*)

Bases: `object`

Represents a generic HTTP request used by other functionalities in **web-poet** like *`HttpClient`*.

url: **RequestUrl**

method: `str`

headers: *`HttpRequestHeaders`*

body: *`HttpRequestBody`*

urljoin(*url: Union[str, RequestUrl, ResponseUrl]*) → RequestUrl

Return *url* as an absolute URL.

If *url* is relative, it is made absolute relative to *`url`*.

class web_poet.page_inputs.http.**HttpResponse**(*url: Union[str, _Url]*, *body*, ***, *status: Optional[int] = None*, *headers=_Nothing.NOTHING*, *encoding: Optional[str] = None*)

Bases: `SelectableMixin`, `UrlShortcutsMixin`

A container for the contents of a response, downloaded directly using an HTTP client.

`url` should be a URL of the response (after all redirects), not a URL of the request, if possible.

`body` contains the raw HTTP response body.

The following are optional since it would depend on the source of the `HttpResponse` if these are available or not. For example, the responses could simply come off from a local HTML file which doesn't contain `headers` and `status`.

`status` should represent the int status code of the HTTP response.

`headers` should contain the HTTP response headers.

`encoding` encoding of the response. If None (default), encoding is auto-detected from headers and body content.

url: ResponseUrl

body: *HttpResponseBody*

status: Optional[int]

headers: *HttpResponseHeaders*

property text: str

Content of the HTTP body, converted to unicode using the detected encoding of the response, according to the web browser rules (respecting Content-Type header, etc.)

property encoding: Optional[str]

Encoding of the response

json() → Any

Deserialize a JSON document to a Python object.

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of `parsel.selector.Selector`.

urljoin(*url: Union[str,* RequestUrl, ResponseUrl*]*) → *RequestUrl*

Return *url* as an absolute URL.

If *url* is relative, it is made absolute relative to the base URL of *self*.

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

web_poet.page_inputs.http.**request_fingerprint**(*req:* HttpRequest) → str

Return the fingerprint of the request.

class web_poet.page_inputs.response.**AnyResponse**(*response: Union[*BrowserResponse, HttpResponse*]*)

Bases: SelectableMixin, UrlShortcutsMixin

A container that holds either *BrowserResponse* or *HttpResponse*.

response: Union[*BrowserResponse*, *HttpResponse*]

property url: ResponseUrl

URL of the response.

property text: str

Text or HTML contents of the response.

property status: Optional[int]

The int status code of the HTTP response, if available.

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of parsel.selector.Selector.

urljoin(*url: Union[str,* RequestUrl*,* ResponseUrl*]*) → *RequestUrl*

Return *url* as an absolute URL.

If *url* is relative, it is made absolute relative to the base URL of *self*.

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

class web_poet.page_inputs.page_params.**PageParams**

Bases: dict

Container class that could contain any arbitrary data to be passed into a Page Object.

Note that this is simply a subclass of Python's `dict`.

class web_poet.page_inputs.stats.**StatCollector**

Bases: ABC

Base class for web-poet to implement the storing of data written through *Stats*.

abstract set(*key: str*, *value: Any*) → None

Set the value of stat *key* to *value*.

abstract inc(*key: str*, *value: Union[int, float] = 1*) → None

Increment the value of stat *key* by *value*, or set it to *value* if *key* has no value.

class web_poet.page_inputs.stats.**DummyStatCollector**

Bases: *StatCollector*

StatCollector implementation that does not persist stats. It is used when running automatic tests, where stat storage is not necessary.

set(*key: str*, *value: Any*) → None

Set the value of stat *key* to *value*.

inc(*key: str*, *value: Union[int, float] = 1*) → None

Increment the value of stat *key* by *value*, or set it to *value* if *key* has no value.

class web_poet.page_inputs.stats.**Stats**(*stat_collector=None*)

Bases: object

Page input class to write key-value data pairs during parsing that you can inspect later. See *Stats*.

Stats can be set to a fixed value or, if numeric, incremented.

Stats are write-only.

Storage and read access of stats depends on the web-poet framework that you are using. Check the documentation of your web-poet framework to find out if it supports stats, and if so, how to read stored stats.

set(*key: str*, *value: Any*) → None

Set the value of stat *key* to *value*.

inc(*key: str*, *value: Union[int, float] = 1*) → None

Increment the value of stat *key* by *value*, or set it to *value* if *key* has no value.

21.2 Pages

class web_poet.pages.**Injectable**

Bases: ABC, *FieldsMixin*

Base Page Object class, which all Page Objects should inherit from (probably through Injectable subclasses).

Frameworks which are using `web-poet` Page Objects should use *is_injectable()* function to detect if an object is an Injectable, and if an object is injectable, allow building it automatically through dependency injection, using https://github.com/scrapinghub/andi library.

Instead of inheriting you can also use `Injectable.register(MyWebPage)`. `Injectable.register` can also be used as a decorator.

web_poet.pages.**is_injectable**(*cls: Any*) → bool

Return True if `cls` is a class which inherits from *Injectable*.

class web_poet.pages.**ItemPage**

Bases: *Extractor*[ItemT], *Injectable*

Base class for page objects.

async to_item() → ItemT

Extract an item from a web page

class web_poet.pages.**WebPage**(*response: HttpResponse*)

Bases: *ItemPage*[ItemT], *ResponseShortcutsMixin*

Base Page Object which requires *HttpResponse* and provides XPath / CSS shortcuts.

response: *HttpResponse*

property base_url: str

Return the base url of the given response

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

property html: str

Shortcut to HTML Response's content.

property item_cls: type

Item class

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of `parsel.selector.Selector`.

async to_item() → ItemT

Extract an item from a web page

property url: str

Shortcut to HTML Response's URL, as a string.

urljoin(*url: str*) → str

Convert url to absolute, taking in account url and baseurl of the response

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

class web_poet.pages.**Returns**

Bases: Generic[ItemT]

Inherit from this generic mixin to change the item class used by *ItemPage*

property item_cls: type

Item class

class web_poet.pages.**Extractor**

Bases: *Returns*[ItemT], *FieldsMixin*

Base class for field support.

async to_item() → ItemT

Extract an item

class web_poet.pages.**SelectorExtractor**(*selector: Selector*)

Bases: *Extractor*[ItemT], SelectorShortcutsMixin

Extractor that takes a `parsel.Selector` and provides shortcuts for its methods.

21.3 Mixins

class web_poet.mixins.**ResponseShortcutsMixin**(**args*, ***kwargs*)

Common shortcut methods for working with HTML responses. This mixin could be used with Page Object base classes.

It requires "response" attribute to be present.

property url: str

Shortcut to HTML Response's URL, as a string.

property html: str

Shortcut to HTML Response's content.

property base_url: str

Return the base url of the given response

urljoin(*url: str*) → str

Convert url to absolute, taking in account url and baseurl of the response

css(*query*) → SelectorList

A shortcut to `.selector.css()`.

jmespath(*query: str*, ***kwargs*) → SelectorList

A shortcut to `.selector.jmespath()`.

property selector: Selector

Cached instance of `parsel.selector.Selector`.

xpath(*query*, ***kwargs*) → SelectorList

A shortcut to `.selector.xpath()`.

21.4 Requests

web_poet.requests.**request_downloader_var: ContextVar = <ContextVar name='request_downloader'>**

Frameworks that wants to support additional requests in `web-poet` should set the appropriate implementation of `request_downloader_var` for requesting data.

21.5 Exceptions

21.5.1 Core Exceptions

These exceptions are tied to how **web-poet** operates.

exception web_poet.exceptions.core.**RequestDownloaderVarError**

The `web_poet.request_downloader_var` had its contents accessed but there wasn't any value set during the time requests are executed.

See the documentation section about *setting up the contextvars* to learn more about this.

exception web_poet.exceptions.core.**PageObjectAction**

Base class for exceptions that can be raised from a page object to indicate something to be done about that page object.

exception web_poet.exceptions.core.**Retry**

The page object found that the input data is partial or empty, and a request retry may provide better input.

exception web_poet.exceptions.core.**UseFallback**

The page object cannot extract data from the input, but the input seems valid, so an alternative data extraction implementation for the same item type may succeed.

exception web_poet.exceptions.core.**NoSavedHttpResponse**(*msg: str = None*, *request: HttpRequest = None*)

Indicates that there is no saved response for this request.

Can only be raised when a *HttpClient* instance is used to get saved responses.

Parameters

request (HttpRequest) – The *HttpRequest* instance that was used.

21.5.2 HTTP Exceptions

These are exceptions pertaining to common issues faced when executing HTTP operations.

exception web_poet.exceptions.http.**HttpError**(*msg: str = None*, *request: HttpRequest = None*)

Bases: OSError

Indicates that an exception has occurred when handling an HTTP operation.

This is used as a **base class** for more specific errors and could be vague since it could denote problems either in the HTTP Request or Response.

For more specific errors, it would be better to use *HttpRequestError* and *HttpResponseError*.

Parameters

request (HttpRequest) – Request that triggered the exception.

request: Optional[*HttpRequest*]

Request that triggered the exception.

exception web_poet.exceptions.http.**HttpRequestError**(*msg: str = None*, *request:* HttpRequest = *None*)

Bases: *HttpError*

Indicates that an exception has occurred when the **HTTP Request** was being handled.

Parameters

request (HttpRequest) – The *HttpRequest* instance that was used.

exception web_poet.exceptions.http.**HttpResponseError**(*msg: str = None*, *response:* HttpResponse = *None*, *request:* HttpRequest = *None*)

Bases: *HttpError*

Indicates that an exception has occurred when the **HTTP Response** was received.

For responses that are in the status code `100-3xx range`, this exception shouldn't be raised at all. However, for responses in the `400-5xx`, this will be raised by **web-poet**.

Note: Frameworks implementing **web-poet** should **NOT** raise this exception.

This exception is raised by web-poet itself, based on `allow_status` parameter found in the methods of *HttpClient*.

Parameters

- **request** (HttpRequest) – Request that got the response that triggered the exception.
- **response** (HttpResponse) – Response that triggered the exception.

response: Optional[*HttpResponse*]

Response that triggered the exception.

21.6 Apply Rules

See *Rules* for more context about its use cases and some examples.

web_poet.**handle_urls**(*include: Union[str, Iterable[str]]*, ***, *overrides: Optional[Type[*ItemPage*]] = None*, *instead_of: Optional[Type[*ItemPage*]] = None*, *to_return: Optional[Type] = None*, *exclude: Optional[Union[str, Iterable[str]]] = None*, *priority: int = 500*, ***kwargs*)

Class decorator that indicates that the decorated Page Object should work for the given URL patterns.

The URL patterns are matched using the `include` and `exclude` parameters while `priority` breaks any ties. See the documentation of the url-matcher package for more information about them.

This decorator is able to derive the item class returned by the Page Object. This is important since it marks what type of item the Page Object is capable of returning for the given URL patterns. For certain advanced cases, you can pass a `to_return` parameter which replaces any derived values (though this isn't generally recommended).

Passing another Page Object into the `instead_of` parameter indicates that the decorated Page Object will be used instead of that for the given set of URL patterns. See *Rule precedence*.

Any extra parameters are stored as meta information that can be later used.

Parameters

- **include** – The URLs that should be handled by the decorated Page Object.
- **instead_of** – The Page Object that should be *replaced*.
- **to_return** – The item class holding the data returned by the Page Object. This could be omitted as it could be derived from the `Returns[ItemClass]` or `ItemPage[ItemClass]` declaration of the Page Object. See *Items* section.
- **exclude** – The URLs for which the Page Object should **not** be applied.
- **priority** – The resolution priority in case of *conflicting* rules. A conflict happens when the `include`, `override`, and `exclude` parameters are the same. If so, the *highest priority* will be chosen.

class web_poet.rules.**ApplyRule**(*for_patterns: Union[str, Patterns]*, ***, *use: Type[*ItemPage*]*, *instead_of: Optional[Type[*ItemPage*]] = None*, *to_return: Optional[Type[Any]] = None*, *meta: Dict[str, Any] = _Nothing.NOTHING*)

A rule that primarily applies Page Object and Item overrides for a given URL pattern.

This is instantiated when using the *web_poet.handle_urls()* decorator. It's also being returned as a `List[ApplyRule]` when calling the `web_poet.default_registry`'s *get_rules()* method.

You can access any of its attributes:

- `for_patterns` - contains the list of URL patterns associated with this rule. You can read the API documentation of the url-matcher package for more information about the patterns.
- `use` - The Page Object that will be **used** in cases where the URL pattern represented by the `for_patterns` attribute is matched.
- `instead_of` - *(optional)* The Page Object that will be **replaced** with the Page Object specified via the `use` parameter.
- `to_return` - *(optional)* The item class that the Page Object specified in `use` is capable of returning.
- `meta` - *(optional)* Any other information you may want to store. This doesn't do anything for now but may be useful for future API updates.

The main functionality of this class lies in the `instead_of` and `to_return` parameters. Should both of these be omitted, then *ApplyRule* simply tags which URL patterns the given Page Object defined in `use` is expected to be used on.

When `to_return` is not None (e.g. `to_return=MyItem`), the Page Object in `use` is declared as capable of returning a certain item class (i.e. `MyItem`).

When `instead_of` is not None (e.g. `instead_of=ReplacedPageObject`), the rule adds an expectation that the `ReplacedPageObject` wouldn't be used for the URLs matching `for_patterns`, since the Page Object in `use` will replace it.

If there are multiple rules which match a certain URL, the rule to apply is picked based on the priorities set in `for_patterns`.

More information regarding its usage in *Rules*.

Tip: The *ApplyRule* is also hashable. This makes it easy to store unique rules and identify any duplicates.

---

class web_poet.rules.**RulesRegistry**(***, *rules: Optional[Iterable[*ApplyRule*]] = None*)

RulesRegistry provides features for storing, retrieving, and searching for the *ApplyRule* instances.

`web-poet` provides a default Registry named `default_registry` for convenience. It can be accessed this way:

```
from web_poet import handle_urls, default_registry, WebPage
from my_items import Product

@handle_urls("example.com")
class ExampleComProductPage(WebPage[Product]):
    ...

rules = default_registry.get_rules()
```

The `@handle_urls` decorator exposed as `web_poet.handle_urls` is a shortcut for `default_registry.handle_urls`.

Note: It is encouraged to use the `web_poet.default_registry` instead of creating your own *RulesRegistry* instance. Using multiple registries would be unwieldy in most cases.

However, it might be applicable in certain scenarios like storing custom rules to separate it from the `default_registry`.

---

add_rule(*rule:* ApplyRule) → None

Registers an *web_poet.rules.ApplyRule* instance.

classmethod from_override_rules(*rules: List[*ApplyRule*]*) → RulesRegistryTV

Deprecated. Use `RulesRegistry(rules=...)` instead.

get_rules() → List[*ApplyRule*]

Return all the *ApplyRule* that were declared using the `@handle_urls` decorator.

Note: Remember to consider calling *consume_modules()* beforehand to recursively import all submodules which contains the `@handle_urls` decorators from external Page Objects.

---

get_overrides() → List[*ApplyRule*]

Deprecated, use *get_rules()* instead.

search(***kwargs*) → List[*ApplyRule*]

Return any *ApplyRule* from the registry that matches with all the provided attributes.

Sample usage:

```
rules = registry.search(use=ProductPO, instead_of=GenericPO)
print(len(rules))           # 1
print(rules[0].use)         # ProductPO
print(rules[0].instead_of)  # GenericPO
```

search_overrides(***kwargs*) → List[*ApplyRule*]

Deprecated, use *search()* instead.

overrides_for(*url: Union[_Url, str]*) → Mapping[Type[*ItemPage*], Type[*ItemPage*]]

Finds all of the page objects associated with the given URL and returns a Mapping where the 'key' represents the page object that is **overridden** by the page object in 'value'.

page_cls_for_item(*url: Union[_Url, str]*, *item_cls: Type*) → Optional[Type]

Return the page object class associated with the given URL that's able to produce the given `item_cls`.

web_poet.rules.**consume_modules**(**modules: str*) → None

This recursively imports all packages/modules so that the `@handle_urls` decorators are properly discovered and imported.

Let's take a look at an example:

```
# FILE: my_page_obj_project/load_rules.py

from web_poet import default_registry, consume_modules

consume_modules("other_external_pkg.po", "another_pkg.lib")
rules = default_registry.get_rules()
```

For this case, the *ApplyRule* are coming from:

- `my_page_obj_project` *(since it's the same module as the file above)*
- `other_external_pkg.po`
- `another_pkg.lib`
- any other modules that was imported in the same process inside the packages/modules above.

If the `default_registry` had other `@handle_urls` decorators outside of the packages/modules listed above, then the corresponding *ApplyRule* won't be returned. Unless, they were recursively imported in some way similar to *consume_modules()*.

class web_poet.rules.**OverrideRule**(**args*, ***kwargs*)

class web_poet.rules.**PageObjectRegistry**(**args*, ***kwargs*)

21.7 Fields

`web_poet.fields` is a module with helpers for putting extraction logic into separate Page Object methods / properties.

class web_poet.fields.**FieldInfo**(*name: str*, *meta: Optional[dict] = None*, *out: Optional[List[Callable]] = None*)

Information about a field

name: `str`
: name of the field

meta: `Optional[dict]`
: field metadata

out: `Optional[List[Callable]]`
: field processors

class web_poet.fields.**FieldsMixin**

A mixin which is required for a class to support fields

web_poet.fields.**field**(*method=None*, ***, *cached: bool = False*, *meta: Optional[dict] = None*, *out: Optional[List[Callable]] = None*)

Page Object method decorated with `@field` decorator becomes a property, which is then used by *ItemPage*'s to_item() method to populate a corresponding item attribute.

By default, the value is computed on each property access. Use `@field(cached=True)` to cache the property value.

The `meta` parameter allows to store arbitrary information for the field, e.g. `@field(meta={"expensive": True})`. This information can be later retrieved for all fields using the *get_fields_dict()* function.

The `out` parameter is an optional list of field processors, which are functions applied to the value of the field before returning it.

web_poet.fields.**get_fields_dict**(*cls_or_instance*) → Dict[str, *FieldInfo*]

Return a dictionary with information about the fields defined for the class: keys are field names, and values are *web_poet.fields.FieldInfo* instances.

async web_poet.fields.**item_from_fields**(*obj*, *item_cls: ~typing.Type[~web_poet.fields.T] = <class 'dict'>*, **, skip_nonitem_fields: bool = False*) → T

Return an item of `item_cls` type, with its attributes populated from the `obj` methods decorated with *field* decorator.

If `skip_nonitem_fields` is True, `@fields` whose names are not among `item_cls` field names are not passed to `item_cls.__init__`.

When `skip_nonitem_fields` is False (default), all `@fields` are passed to `item_cls.__init__`, possibly causing exceptions if `item_cls.__init__` doesn't support them.

web_poet.fields.**item_from_fields_sync**(*obj*, *item_cls: ~typing.Type[~web_poet.fields.T] = <class 'dict'>*, **, skip_nonitem_fields: bool = False*) → T

Synchronous version of *item_from_fields()*.

21.8 typing.Annotated support

class web_poet.annotated.**AnnotatedInstance**(*result: Any*, *metadata: Tuple[Any, ...]*)

Wrapper for instances of annotated dependencies.

It is used when both the dependency value and the dependency annotation are needed.

Parameters

- **result** (*Any*) – The wrapped dependency instance.
- **metadata** (*Tuple[Any, ...]*) – The copy of the annotation.

get_annotated_cls()

Returns a re-created `typing.Annotated` type.

21.9 Utils

web_poet.utils.**get_fq_class_name**(*cls: type*) → str

Return the fully qualified name for a type.

```
>>> from web_poet import Injectable
>>> get_fq_class_name(Injectable)
'web_poet.pages.Injectable'
>>> from decimal import Decimal
>>> get_fq_class_name(Decimal)
'decimal.Decimal'
```

web_poet.utils.**memoizemethod_noargs**(*method: CallableT*) → CallableT

Decorator to cache the result of a method (without arguments) using a weak reference to its object.

It is faster than *cached_method()*, and doesn't add new attributes to the instance, but it doesn't work if objects are unhashable.

web_poet.utils.**cached_method**(*method: CallableT*) → CallableT

A decorator to cache method or coroutine method results, so that if it's called multiple times for the same instance, computation is only done once.

The cache is unbound, but it's tied to the instance lifetime.

Note: *cached_method()* is needed because `functools.lru_cache()` doesn't work well on methods: self is used as a cache key, so a reference to an instance is kept in the cache, and this prevents deallocation of instances.

This decorator adds a new private attribute to the instance named `_cached_method_{decorated_method_name}`; make sure the class doesn't define an attribute of the same name.

web_poet.utils.**as_list**(*value: Optional[Any]*) → List[Any]

Normalizes the value input as a list.

```
>>> as_list(None)
[]
>>> as_list("foo")
['foo']
>>> as_list(123)
[123]
>>> as_list(["foo", "bar", 123])
['foo', 'bar', 123]
>>> as_list(("foo", "bar", 123))
['foo', 'bar', 123]
>>> as_list(range(5))
[0, 1, 2, 3, 4]
>>> def gen():
...     yield 1
...     yield 2
>>> as_list(gen())
[1, 2]
```

async web_poet.utils.**ensure_awaitable**(*obj*)

Return the value of obj, awaiting it if needed

web_poet.utils.**get_generic_param**(*cls: type*, *expected: Union[type, Tuple[type, ...]]*) → Optional[type]

Search the base classes recursively breadth-first for a generic class and return its param.

Returns the param of the first found class that is a subclass of `expected`.

21.10 Example framework

The *web_poet.example* module is a simplified, incomplete example of a web-poet framework, written as support material for the *tutorial*.

No part of the *web_poet.example* module is intended for production use, and it may change in a backward-incompatible way at any point in the future.

web_poet.example.**get_item**(*url: str*, *item_cls: Type*, ***, *page_params: Optional[Dict[Any, Any]] = None*) → Any

Returns an item built from the specified URL using a page object class from the default registry.

This function is an example of a minimal, incomplete web-poet framework implementation, intended for use in the web-poet tutorial.

CHAPTER

TWENTYTWO

CONTRIBUTING

web-poet is an open-source project. Your contribution is very welcome!

22.1 Issue Tracker

If you have a bug report, a new feature proposal or simply would like to make a question, please check our issue tracker on Github: https://github.com/scrapinghub/web-poet/issues

22.2 Source code

Our source code is hosted on Github: https://github.com/scrapinghub/web-poet

Before opening a pull request, it might be worth checking current and previous issues. Some code changes might also require some discussion before being accepted so it might be worth opening a new issue before implementing huge or breaking changes.

22.3 Testing

We use tox to run tests with different Python versions:

```
tox
```

The command above also runs type checks; we use mypy.

CHAPTER

TWENTYTHREE

CHANGELOG

23.1 0.17.0 (2024-03-04)

- Now requires `andi >= 0.5.0`.
- Package requirements that were unversioned now have minimum versions specified.
- Added support for Python 3.12.
- Added support for `typing.Annotated` dependencies to the serialization and testing code.
- Documentation improvements.
- CI improvements.

23.2 0.16.0 (2024-01-23)

- Added new *`AnyResponse`* which holds either *`BrowserResponse`*, or *`HttpResponse`*.
- Documentation improvements.

23.3 0.15.1 (2023-11-21)

- `HttpRequestHeaders` now has a `from_bytes_dict` class method, like `HttpResponseHeaders`.

23.4 0.15.0 (2023-09-11)

- A new dependency, *`Stats`*, has been added. It allows storing key-value data pairs for different purposes. See *Stats*.

23.5 0.14.0 (2023-08-03)

- Dropped Python 3.7 support.
- Now requires `packaging >= 20.0`.
- Fixed detection of the *`Returns`* base class.
- Improved docs.
- Updated type hints.
- Updated CI tools.

23.6 0.13.1 (2023-05-30)

- Fixed an issue with *`HttpClient`* which happens when a response with a non-standard status code is received.

23.7 0.13.0 (2023-05-30)

- A new dependency *`BrowserResponse`* has been added. It contains a browser-rendered page URL, status code and HTML.
- The *Rules* documentation section has been rewritten.

23.8 0.12.0 (2023-05-05)

- The *testing framework* now allows defining a *custom item adapter*.
- We have made a backward-incompatible change on test fixture serialization: the `type_name` field of exceptions has been renamed to `import_path`.
- Fixed built-in Python types, e.g. `int`, not working as *field processors*.

23.9 0.11.0 (2023-04-24)

- JMESPath support is now available: you can use *`WebPage.jmespath()`* and *`HttpResponse.jmespath()`* to run queries on JSON responses.
- The testing framework now supports page objects that raise exceptions from the `to_item` method.

23.10 0.10.0 (2023-04-19)

- New class *Extractor* can be used for easier extraction of nested fields (see *Processors for nested fields*).
- Exceptions raised while getting a response for an additional request are now saved in *test fixtures*.
- Multiple documentation improvements and fixes.
- Add a `twine check` CI check.

23.11 0.9.0 (2023-03-30)

- Standardized *input validation*.
- *Field processors* can now also be defined through a nested `Processors` class, so that field redefinitions in subclasses can inherit them. See *Default processors*.
- *Field processors* can now opt in to receive the page object whose field is being read.
- *web_poet.fields.FieldsMixin* now keeps fields from all base classes when using multiple inheritance.
- Fixed the documentation build.

23.12 0.8.1 (2023-03-03)

- Fix the error when calling *.to_item()*, *item_from_fields_sync()*, or *item_from_fields()* on page objects defined as slotted attrs classes, while setting `skip_nonitem_fields=True`.

23.13 0.8.0 (2023-02-23)

This release contains many improvements to the web-poet testing framework, as well as some other improvements and bug fixes.

Backward-incompatible changes:

- *cached_method()* no longer caches exceptions for `async def` methods. This makes the behavior the same for sync and async methods, and also makes it consistent with Python's stdlib caching (i.e. `functools.lru_cache()`, `functools.cached_property()`).
- The testing framework now uses the `HttpResponse-info.json` file name instead of `HttpResponse-other.json` to store information about HttpResponse instances. To make tests generated with older web-poet work, rename these files on disk.

Testing framework improvements:

- Improved test reporting: better diffs and error messages.
- By default, the pytest plugin now generates a test per item attribute (see *Running tests*). There is also an option (`--web-poet-test-per-item`) to run a test per item instead.
- Page objects with the *HttpClient* dependency are now supported (see *Additional requests support*).
- Page objects with the *PageParams* dependency are now supported.
- Added a new `python -m web_poet.testing rerun` command (see *Test-Driven Development*).

- Fixed support for nested (indirect) dependencies in page objects. Previously they were not handled properly by the testing framework.
- Non-ASCII output is now stored without escaping in the test fixtures, for better readability.

Other changes:

- Testing and CI fixes.
- Fixed a packaging issue: `tests` and `tests_extra` packages were installed, not just `web_poet`.

23.14 0.7.2 (2023-02-01)

- Restore the minimum version of `itemadapter` from 0.7.1 to 0.7.0, and prevent a similar issue from happening again in the future.

23.15 0.7.1 (2023-02-01)

- Updated the *tutorial* to cover recent features and focus on best practices. Also, a new module was added, *web_poet.example*, that allows using page objects while following the tutorial.
- *Tests for page objects* now covers *Git LFS* and *scrapy-poet*, and recommends `python -m pytest` instead of `pytest`.
- Improved the warning message when duplicate `ApplyRule` objects are found.
- `HttpResponse-other.json` content is now indented for better readability.
- Improved test coverage for *fields*.

23.16 0.7.0 (2023-01-18)

- Add *a framework for creating tests and running them with pytest*.
- Support implementing fields in mixin classes.
- Introduce new methods for *web_poet.rules.RulesRegistry*:
  - *web_poet.rules.RulesRegistry.add_rule()*
  - *web_poet.rules.RulesRegistry.overrides_for()*
  - *web_poet.rules.RulesRegistry.page_cls_for_item()*
- Improved the performance of *web_poet.rules.RulesRegistry.search()* where passing a single parameter of either `instead_of` or `to_return` results in *O(1)* look-up time instead of *O(N)*. Additionally, having either `instead_of` or `to_return` present in multi-parameter search calls would filter the initial candidate results resulting in a faster search.
- Support *page object dependency serialization*.
- Add new dependencies used in testing and serialization code: `andi`, `python-dateutil`, and `time-machine`. Also `backports.zoneinfo` on non-Windows platforms when the Python version is older than 3.9.

23.17 0.6.0 (2022-11-08)

In this release, the `@handle_urls` decorator gets an overhaul; it's not required anymore to pass another Page Object class to `@handle_urls("...", overrides=...)`.

Also, the `@web_poet.field` decorator gets support for output processing functions, via the `out` argument.

Full list of changes:

- **Backwards incompatible** `PageObjectRegistry` is no longer supporting dict-like access.
- Official support for Python 3.11.
- New `@web_poet.field(out=[...])` argument which allows to set output processing functions for web-poet fields.
- The `web_poet.overrides` module is deprecated and replaced with `web_poet.rules`.
- The `@handle_urls` decorator is now creating `ApplyRule` instances instead of `OverrideRule` instances; `OverrideRule` is deprecated. `ApplyRule` is similar to `OverrideRule`, but has the following differences:
  - `ApplyRule` accepts a `to_return` parameter, which should be the data container (item) class that the Page Object returns.
  - Passing a string to `for_patterns` would auto-convert it into `url_matcher.Patterns`.
  - All arguments are now keyword-only except for `for_patterns`.
- New signature and behavior of `handle_urls`:
  - The `overrides` parameter is made optional and renamed to `instead_of`.
  - If defined, the item class declared in a subclass of `web_poet.ItemPage` is used as the `to_return` parameter of `ApplyRule`.
  - Multiple `handle_urls` annotations are allowed.
- `PageObjectRegistry` is replaced with `RulesRegistry`; its API is changed:
  - **backwards incompatible** dict-like API is removed;
  - **backwards incompatible** *O(1)* lookups using `.search(use=PagObject)` has become *O(N)*;
  - `search_overrides` method is renamed to `search`;
  - `get_overrides` method is renamed to `get_rules`;
  - `from_override_rules` method is deprecated; use `RulesRegistry(rules=...)` instead.
- Typing improvements.
- Documentation, test, and warning message improvements.

Deprecations:

- The `web_poet.overrides` module is deprecated. Use `web_poet.rules` instead.
- The `overrides` parameter from `@handle_urls` is now deprecated. Use the `instead_of` parameter instead.
- The `OverrideRule` class is now deprecated. Use `ApplyRule` instead.
- `PageObjectRegistry` is now deprecated. Use `RulesRegistry` instead.
- The `from_override_rules` method of `PageObjectRegistry` is now deprecated. Use `RulesRegistry(rules=...)` instead.
- The `PageObjectRegistry.get_overrides` method is deprecated. Use `PageObjectRegistry.get_rules` instead.

- The `PageObjectRegistry.search_overrides` method is deprecated. Use `PageObjectRegistry.search` instead.

23.18 0.5.1 (2022-09-23)

- The BOM encoding from the response body is now read before the response headers when deriving the response encoding.
- Minor typing improvements.

23.19 0.5.0 (2022-09-21)

Web-poet now includes a mini-framework for organizing extraction code as Page Object properties:

```
import attrs
from web_poet import field, ItemPage

@attrs.define
class MyItem:
    foo: str
    bar: list[str]


class MyPage(ItemPage[MyItem]):
    @field
    def foo(self):
        return "..."

    @field
    def bar(self):
        return ["...", "..."]
```

Backwards incompatible changes:

- `web_poet.ItemPage` is no longer an abstract base class which requires `to_item` method to be implemented. Instead, it provides a default `async def to_item` method implementation which uses fields marked as `web_poet.field` to create an item. This change shouldn't affect the user code in a backwards incompatible way, but it might affect typing.

Deprecations:

- `web_poet.ItemWebPage` is deprecated. Use `web_poet.WebPage` instead.

Other changes:

- web-poet is declared as PEP 561 package which provides typing information; mypy is going to use it by default.
- Documentation, test, typing and CI improvements.

23.20 0.4.0 (2022-07-26)

- New `HttpResponse.urljoin` method, which take page's base url in account.
- New `HttpRequest.urljoin` method.
- standardized `web_poet.exceptions.Retry` exception, which allows to initiate a retry from the Page Object, e.g. based on page content.
- Documentation improvements.

23.21 0.3.0 (2022-06-14)

- Backwards Incompatible Change:
  - `web_poet.requests.request_backend_var` is renamed to `web_poet.requests.request_downloader_var`.
- Documentation and CI improvements.

23.22 0.2.0 (2022-06-10)

- Backward Incompatible Change:
  - `ResponseData` is replaced with `HttpResponse`.

    `HttpResponse` exposes methods useful for web scraping (such as xpath and css selectors, json loading), and handles web page encoding detection. There are also new types like `HttpResponseBody` and `HttpResponseHeaders`.
- Added support for performing additional requests using `web_poet.HttpClient`.
- Introduced `web_poet.BrowserHtml` dependency
- Introduced `web_poet.PageParams` to pass arbitrary information inside a Page Object.
- Added `web_poet.handle_urls` decorator, which allows to declare which websites should be handled by the page objects. Lower-level `PageObjectRegistry` class is also available.
- removed support for Python 3.6
- added support for Python 3.10

23.23 0.1.1 (2021-06-02)

- `base_url` and `urljoin` shortcuts

23.24 0.1.0 (2020-07-18)

- Documentation
- WebPage, ItemPage, ItemWebPage, Injectable and ResponseData are available as top-level imports (e.g. `web_poet.ItemPage`)

23.25 0.0.1 (2020-04-27)

Initial release.

CHAPTER

TWENTYFOUR

LICENSE

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of Zyte nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

PYTHON MODULE INDEX

W

INDEX

H

I

J

M

N

O

P

R

S

T

U

W

X