web-poet Documentation

Release 0.6.0

Zyte Group Ltd

Nov 30, 2022

GETTING STARTED

`web-poet` is a Python 3.7+ implementation of the page object pattern for web scraping. It enables writing portable, reusable web data extraction 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.

CHAPTER

TWO

INSTALLATION

Install web-poet from PyPI:

```
pip install web-poet
```

CHAPTER

THREE

WEB-POET ON A SURFACE

In this tutorial, we'll assume that web-poet is already installed on your system.

We are going to scrape books.toscrape.com, a website that lists books from famous authors.

3.1 Creating a Page Object

Let's create a Python file where we'll put our first Page Object implementation. This Page Object will be responsible for extracting book links from the book list page on books.toscrape.com.

```
from web_poet.pages import WebPage


class BookLinksPage(WebPage):

    @property
    def links(self):
        return self.css(".image_container a::attr(href)").getall()

    def to_item(self) -> dict:
        return {
            "links": self.links,
        }
```

3.2 Downloading Response

The `BookLinksPage` Page Object requires a *`HttpResponse`* with the book list page content in order to extract the information we need. First, let's download the page using `requests` library.

```
import requests


response = requests.get("http://books.toscrape.com")
```

3.3 Creating Page Input

Now we need to create and populate a *HttpResponse* instance.

```
from web_poet.page_inputs import HttpResponse


response_data = HttpResponse(response.url,
                             body=response.content,
                             headers=response.headers)
page = BookLinksPage(response_data)

print(page.to_item())
```

3.4 Final Result

Our simple Python script might look like this:

```
import requests

from web_poet.pages import WebPage
from web_poet.page_inputs import HttpResponse


class BookLinksPage(WebPage):

    @property
    def links(self):
        return self.css(".image_container a::attr(href)").getall()

    def to_item(self) -> dict:
        return {
            "links": self.links,
        }


response = requests.get("http://books.toscrape.com")
response_data = HttpResponse(response.url,
                             body=response.content,
                             headers=response.headers)

page = BookLinksPage(response_data)

print(page.to_item())
```

And it should output data similar to this:

```
{
    "links": [
        "catalogue/a-light-in-the-attic_1000/index.html",
        "catalogue/tipping-the-velvet_999/index.html",
```

(continues on next page)

(continued from previous page)

```
        "catalogue/soumission_998/index.html",
        "catalogue/sharp-objects_997/index.html",
        "catalogue/sapiens-a-brief-history-of-humankind_996/index.html",
        "catalogue/the-requiem-red_995/index.html",
        "catalogue/the-dirty-little-secrets-of-getting-your-dream-job_994/index.html",
        "catalogue/the-coming-woman-a-novel-based-on-the-life-of-the-infamous-feminist-
↪victoria-woodhull_993/index.html",
        "catalogue/the-boys-in-the-boat-nine-americans-and-their-epic-quest-for-gold-at-
↪the-1936-berlin-olympics_992/index.html",
        "catalogue/the-black-maria_991/index.html",
        "catalogue/starving-hearts-triangular-trade-trilogy-1_990/index.html",
        "catalogue/shakespeares-sonnets_989/index.html",
        "catalogue/set-me-free_988/index.html",
        "catalogue/scott-pilgrims-precious-little-life-scott-pilgrim-1_987/index.html",
        "catalogue/rip-it-up-and-start-again_986/index.html",
        "catalogue/our-band-could-be-your-life-scenes-from-the-american-indie-
↪underground-1981-1991_985/index.html",
        "catalogue/olio_984/index.html",
        "catalogue/mesaerion-the-best-science-fiction-stories-1800-1849_983/index.html",
        "catalogue/libertarianism-for-beginners_982/index.html",
        "catalogue/its-only-the-himalayas_981/index.html",
    ]
}
```

3.5 Next Steps

As you can see, it's possible to use web-poet with built-in libraries such as `requests`, but it's also possible to use Scrapy with the help of scrapy-poet.

If you want to understand the idea behind web-poet better, check the *web-poet from the ground up* tutorial.

CHAPTER

FOUR

WEB-POET FROM THE GROUND UP

This tutorial explains the motivation behind web-poet, and its main concepts. You would learn

- what are the issues which `web-poet` addresses, why does the library exist;
- how `web-poet` addresses these issues;
- what is a Page Object, and what is a page object input;
- how are libraries like scrapy-poet implemented, and what are they for.

4.1 Reusable web scraping code

Forget about web-poet for a minute. Let's say you're writing code to scrape a book web page from books.toscrape.com:

```
import requests
import parsel


def extract_book(url):
    """ Extract book information from a book page on
    http://books.toscrape.com website, e.g. from
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    resp = requests.get(url)
    sel = parsel.Selector(resp)
    return {
        "url": resp.url,
        "title": sel.css("h1").get(),
        "description": sel.css("#product_description+ p").get().strip(),
        # ...
    }

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

This is sweet & easy, but you realize this code is tightly coupled, and hardly reusable or testable. So you refactor it, to separate downloading from the extraction:

```
import requests
import parsel

```

(continues on next page)

(continued from previous page)

```
def extract_book(response):
    """ Extract book information from a requests.Response obtained
    from a book page on http://books.toscrape.com website, e.g. from
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    sel = parsel.Selector(response.text)
    return {
        "url": response.url,
        "title": sel.css("h1").get(),
        "description": sel.css("#product_description+ p").get().strip(),
        # ...
    }

def download_and_extract_book(url):
    resp = requests.get(url)
    return extract_book(resp)

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

Much cleaner! It is slightly more code, but extract_book is more reusable & testable now.

Then your project evolves, and you realize that you'd like to download web pages using aiohttp. It means that `extract_book` now can't receive `requests.Response`, it needs to work with `aiohttp.Response`. To complicate things, you want to keep `requests` support. There are few options, such as:

- make `extract_book` receive Selector instance
- make `extract_book` receive url and unicode response body
- make `extract_book` receive a Response object which is compatible for both libraries.

No problem, let's refactor it further. You may end up with something like that:

```
import aiohttp
from dataclasses import dataclass
import requests
import parsel

@dataclass
class Response:
    url: str
    text: str

# === Extraction code
def extract_book(response: Response) -> dict:
    """ Extract book information from a book page
    on http://books.toscrape.com website, e.g. from
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    sel = parsel.Selector(response.text)
    return {
        "url": response.url,
        "title": sel.css("h1").get(),
```

(continues on next page)

(continued from previous page)

```
        "description": sel.css("#product_description+ p").get().strip(),
        # ...
    }

# === Framework-specific I/O code
def download_sync(url) -> Response:
    resp = requests.get(url)
    return Response(url=resp.url, text=resp.text)

async def download_async(url) -> Response:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            text = await response.text()
    return Response(url=url, text=text)

# === Usage example
# the way to get the Response instance depends on an HTTP client
resp = download_sync("http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/
↪index.html")

# but after we got the response, the usage is the same
item = extract_book(resp)
```

`extract_book` function now has all the desired properties: it is easily testable and reusable, and it works with any method of downloading data.

4.2 The same, but using web-poet

`web-poet` asks you to organize code in a very similar way. Let's convert `extract_book` function to a Page Object, by defining the BookPage class:

```
import aiohttp
import requests
from web_poet import ItemPage, HttpResponse


# === Extraction code
class BookPage(ItemPage):
    """
    A book page on http://books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    def __init__(self, response: HttpResponse):
        self.response = response

    def to_item(self) -> dict:
        return {
            "url": self.response.url,
            "title": self.response.css("h1").get(),
            "description": self.response.css("#product_description+ p").get().strip(),
```

(continues on next page)

(continued from previous page)

```
            # ...
        }

# === Framework-specific I/O code
def download_sync(url) -> HttpResponse:
    resp = requests.get(url)
    return HttpResponse(url=resp.url, body=resp.content, headers=resp.headers)

async def download_async(url) -> HttpResponse:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            body = await response.content.read()
            headers = response.headers

    return HttpResponse(url=resp.url, body=body, headers=headers)

# === Usage example

# the way to get response depends on an HTTP client
response = download_sync("http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/
↪index.html")

# but after we got the response, the usage is the same
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 *HttpResponse* instance. This is needed to decode body (which is `bytes`) to unicode properly, using the web browser rules. It involves checking `Content-Encoding` header, meta tags in HTML, BOM markers in the body, etc.

- instead of the `extract_book` function we've got a `BookPage` class, which inherits from the *ItemPage* 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:

```
from web_poet import WebPage

class BookPage(WebPage):
    def to_item(self) -> dict:
        return {
            "url": self.response.url,
            "title": self.response.css("h1").get(),
            "description": self.response.css("#product_description+ p").get().strip(),
            # ...
        }
```

There are pros and cons for using classes vs functions for writing such extraction code, but the distinction is not that important; web-poet uses classes at the moment.

4.3 to_item() method

It is common to have Page Objects for a web page where a single main data record needs to be extracted (e.g. book information in our example). `web-poet` standardizes this, by asking to name a method implementing the extraction `to_item`.

As the method name is now standardized, the code which creates a Page Object instance can now work for other Page Objects like that. For example, you can have `ToscrapeBookPage` and `BamazonBookPage` classes, and

```
def get_item(page_cls: WebPage, response: HttpResponse) -> dict:
    page = page_cls(response=response)
    return page.to_item()
```

would work for both.

But wait. Before the example was converted to `web-poet`, we were getting it for free:

```
def get_item(extract_func, response: HttpResponse) -> dict:
    return extract_func(url=response.url, text=response.text)
```

No need to agree on `to_item` name or have a base class. Why bother with classes then?

4.4 Classes for web scraping code

A matter of preference. Functions are great, too. Classes sometimes can make it a easier to organize web scraping code. For example, we can extract logic for different attributes into properties:

```
class BookPage(WebPage):
    """
    A book page on http://books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """

    @property
    def title(self):
        return self.response.css("h1").get()

    @property
    def description(self):
        return self.response.css("#product_description+ p").get().strip()

    def to_item(self):
        return {
            "url": self.response.url,
            "title": self.title,
            "description": self.description,
            # ...
        }
```

It might be easier to read the code written this way. Also, this style allows to extract only some of the attributes - if you don't need the complete to_item() output, you still can access individual properties.

web-poet provides a small framework to simplify writing Page Objects in this style; see *Fields*. The example above can be simplified using web-poet fields - there is no need to write `to_item` boilerplate:

```
from web_poet import WebPage, field

class BookPage(WebPage):
    """
    A book page on http://books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """

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

    @field
    def description(self):
        return self.response.css("#product_description+ p").get().strip()

    @field
    def url(self):
        return self.response.url
```

Note: The `BookPage.to_item()` method is `async` in the example above. Make sure to check *Fields* if you want to use web-poet fields.

Another reason to consider classes for the extraction code is that sometimes there is no a single "main" method, but you still want to group the related code. For example, you may define a "Pagination" page object:

```
class Pagination(WebPage):

    def page_urls(self):
        # ...

    def prev_url(self):
        # ...

    def next_url(self):
        # ...
```

or a Listing page on a web site, where you need to get URLs to individual pages and pagination URLs:

```
class BookListPage(ProductListingPage):
    def item_urls(self):
        return self.response.css(".product a::attr(href)").getall()

    def page_urls(self):
        return self.response.css(".paginator a::attr(href)").getall()
```

4.5 Web Scraping Frameworks

Let's recall the example we started with:

```
import requests
import parsel


def extract_book(url):
    """ Extract book information from a book page on
    http://books.toscrape.com website, e.g. from
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    resp = requests.get(url)
    sel = parsel.Selector(resp)
    return {
        "url": resp.url,
        "title": sel.response.css("h1").get(),
        "description": sel.response.css("#product_description+ p").get().strip(),
        # ...
    }

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

And this is what we ended up with:

```
import requests
from web_poet import WebPage, HttpResponse


# === Extraction code
class BookPage(WebPage):
    """ A book page on http://books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    def to_item(self):
        return {
            "url": self.response.url,
            "title": self.response.css("h1").get(),
            "description": self.response.css("#product_description+ p").get().strip(),
            # ...
        }

# === Framework-specific I/O code
def download_sync(url):
    resp = requests.get(url)
    return HttpResponse(url=resp.url, body=resp.content, headers=resp.headers)

async def download_async(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            body = await response.content.read()
```

(continues on next page)

(continued from previous page)

```
            headers = response.headers

    return HttpResponse(url=resp.url, body=body, headers=headers)

# === Usage example
def get_item(page_cls: WebPage, resp_data: HttpResponse) -> dict:
    page = page_cls(response=resp_data)
    return page.to_item()

# the way to get resp_data depends on an HTTP client
response = download_sync("http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/
↪index.html")
item = get_item(BookPage, resp_data=response)
```

We created a monster!!! The examples in this tutorial are becoming longer and longer, harder and harder to understand. What's going on?

To understand better why this is happening, let's check the `web-poet` example in more detail. There are 3 main sections:

1. Extraction code
2. Lower-level I/O code
3. "Usage example" - it connects (1) and (2) to get the extracted data.

Extraction code needs to be written for every new web site. But the I/O code and the "Usage example" can, and should be written only once!

In other words, we've been creating a web scraping framework here, and that's where most of the complexity is coming from.

In a real world, a developer who needs to extract data from a web page would only need to write the "extraction" part:

```
from web_poet import WebPage

class BookPage(WebPage):
    """ A book page on http://books.toscrape.com website, e.g.
    http://books.toscrape.com/catalogue/a-light-in-the-attic_1000/index.html
    """
    def to_item(self):
        return {
            "url": self.response.url,
            "title": self.response.css("h1").get(),
            "description": self.response.css("#product_description+ p").get().strip(),
            # ...
        }
```

Then point the framework to the `BookPage` class, tell which web page to process, and that's it:

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

`web-poet` **does not** provide such a framework. The role of `web-poet` is to define a standard on how to write the extraction logic, and allow it to be reused in different frameworks. `web-poet` Page Objects should be flexible enough to be used with:

- synchronous or async frameworks, callback-based and `async def / await` based,
- single node and distributed systems,

- different underlying HTTP implementations - or without HTTP support at all, etc.

4.6 Page Objects

In this document "Page Objects" were casually mentioned a few times, but what are they?

Note: This term comes from a Page Object design pattern; see a description on Martin Fowler's website: https://martinfowler.com/bliki/PageObject.html. web-poet page objects are inspired by Martin Fowler's page object, but they are not the same.

Essentially, the idea is to create an object which represents a web page (or a part of web page - recall the `Pagination` example), and allows to extract data from there. Page Object must:

1. Define all the inputs needed in its `__init__` method. Usually these inputs are then stored as attributes.
2. Provide methods or properties to extract structured information, using the data saved in `__init__`.
3. Inherit from *`Injectable`*; this inheritance is used as a marker.

For example, a very basic Page Object could look like this:

```
from parsel import Selector
from web_poet.pages import Injectable
from web_poet.page_inputs import HttpResponse


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

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

There is no *need* to use other base classes and mixins defined by `web-poet` (*`WebPage`*, *`ItemPage`*, etc.), but it can be a good idea to familiarize yourself with them, as they are taking some of the boilerplate out.

4.7 Page Object Inputs

Here we got to the last, and probably the most complicated and important part of `web-poet`. So far we've been passing *`HttpResponse`* to the page objects. But is it enough?

If that'd be enough, there wouldn't be `web-poet`. We would say "please write `def extract(url, html): ...` functions", and call it a day.

In practice you may need to use other information to extract data from a web page, not only *`HttpResponse`*. For example, you may want to

- render a web page in a headless browser like Splash, and use HTML after the rendering (snapshot of a DOM tree);

- query third-party API like AutoExtract, to extract most of the data automatically - a Page Object may just return the result as-is, or enrich / post-process it;
- take some state in account, passed e.g. from the crawling code.
- use a combination of inputs: e.g. you may need HTML after Headless Chrome rendering + crawling state.

The information you need can depend on a web site. For example, Splash can be required for extracting book information from Bamazon, while for http://books.toscrape.com you may need HTTP response body and some crawl state (not really, but let's imagine it is needed). You may define page objects for this task:

```
class BamazonBookPage(Injectable):
    def __init__(self, response: SplashResponse):
        self.response = response

    def to_item(self):
        # ...

class ToScrapeBookPage(Injectable):
    def __init__(self, response: HttpResponse, crawl_state: dict):
        self.response = response
        self.crawl_state = crawl_state

    def to_item(self):
        # ...
```

Then, we would like to use these page objects in some web scraping framework, like we did before:

```
item1 = some_framework.extract(bamazon_url, BamazonBookPage)
item2 = some_framework.extract(toscrape_url, ToScrapeBookPage)
```

To be able to implement the imaginary `some_framework.extract` method, some_framework must

1. Figure out somehow which inputs the Page Objects need.
2. Create these inputs. If needed, make a Splash request, make a direct HTTP request, get a dictionary with the crawl state from somewhere. These actions can be costly; framework should avoid doing unnecessary work here.
3. Pass the obtained data as keyword arguments to `__init__` method.

(2) and *(3)* are straightforward, once the framework knows that "To create BamazonBookPage, I need to pass output of Splash as a `response` keyword argument", i.e. once *(1)* is done.

`web-poet` uses **type annotations** of `__init__` arguments to declare Page Object dependencies. So, type annotations in the examples like the following were not just a nice-thing-to-have:

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

By annotating `__init__` arguments we were actually telling `web-poet` (or, more precisely, a framework which uses `web-poet`):

> To create a `BookPage` instance, please obtain *`HttpResponse`* instance somehow, and pass it as a `response` keyword argument. That's all you need to create a `BookPage` instance.

Note: If it sounds like Dependency Injection, you're right.

If something other than *HttpResponse* needs to be passed, a different type annotation should be used:

```
class BamazonBookPage(Injectable):
    def __init__(self, response: SplashResponse):
        self.response = response

class ToScrapeBookPage(Injectable):
    def __init__(self, response: HttpResponse, crawl_state: CrawlState):
        self.response = response
        self.crawl_state = crawl_state
```

For each possible input a separate class needs to be defined, even if the data has the same format. For example, both *HttpResponse* and `SplashResponse` may have the same `url` and `text` properties, but they can't be the same class, because they need to work as "markers" - tell frameworks if the html should be taken from HTTP response body or from Splash DOM snapshot.

`CrawlState` in the example above can be defined as a class with some specific properties, or maybe even as a `class CrawlState(dict): pass` - an important thing is that it is an unique type, and that we agree on what should be put into arguments annotated as `CrawlState`.

Pro tip: defining classes like

```
class ToScrapeBookPage(Injectable):
    def __init__(self, response: HttpResponse, crawl_state: CrawlState):
        self.response = response
        self.crawl_state = crawl_state
```

can get tedious; Python's `dataclasses` (or attrs, if that's your preference) make it nicer:

```
from dataclasses import dataclass

@dataclass
class ToScrapeBookPage(Injectable):
    response: HttpResponse
    crawl_state: CrawlState
```

4.8 web-poet role

How do you actually inspect the `__init__` method signature - e.g. if you're working on supporting `web-poet` page objects in some framework? `web-poet` itself doesn't provide any helpers for doing this.

Use andi library. For example, scrapy-poet uses andi. In addition to signature inspection, it also handles `typing.Optional` and `typing.Union`, and allows to create a build plan for dependency trees, indirect dependencies: that's allowed to annotate an argument as another *Injectable* subclass.

`web-poet` is not using andi on its own; `web-poet`'s role is to standardize things + provide some helpers to write the extraction code easier:

1. Standardize a list of possible inputs for the page objects. This helps with reusability of extraction code across different environments. For example, if you want to support extraction from raw HTTP response bodies, you need to figure out how to populate *HttpResponse* in the given environment, and that's all.

   Users are free to define their own inputs (input types), but they may be less portable across environments - which can be fine.

2. Define an interface for the Page Object itself. This allows to have a code which can instantiate and use a Page Object without knowing about its implementation upfront. `web-poet` requires you to use a base class (*`Injectable`*), and defines the semantics of `to_item()` method.

Then, framework's role is to:

1. Figure out which inputs a Page Object needs, likely using andi library.
2. Create all the necessary inputs. For example, creating *`HttpResponse`* instance may involve making an HTTP request; creating `CrawlState` (from the previous examples) may involve getting some data from the shared storage, or from an in-memory data structure.
3. Create a Page Object instance, passing it the inputs it needs.
4. Depending on a task, either return a newly created Page Object instance to the user, or call some predefined method (a common case is `to_item`).

For example, `web-poet` + Scrapy integration package (scrapy-poet) inspects a WebPage subclass you defined, figures out it needs *`HttpResponse`* and nothing else, fetches Scrapy's `TextResponse`, creates an *`HttpResponse`* instance from it, creates your Page Object instance, and passes it to a spider callback.

Finally, the Developer's role is to:

1. Write a Page Object class, likely website-specific, following `web-poet` standards. The extraction code should define the inputs it needs (such as "body of HTTP response", "Chrome DOM tree snapshot", "crawl state"); it shouldn't fetch these inputs itself.
2. Pass the Page Object class to a framework, in a way defined by the framework.
3. Receive a Page Object *instance* from the framework; call its extraction methods (e.g. `to_item`). Depending on the framework and on the task, the framework may be calling `to_item` (or other methods) automatically; in this case user code would be getting the extracted data, not a Page Object instance.

4.9 Summary

First, congratulations for making it through this document!

A take-away from this tutorial:

1. `web-poet` does very little on its own. Almost nothing, really. An important thing about `web-poet` is that if defines a standard for writing web scraping code.

   All these Page Objects are just Python classes, which receive some static data in `__init__` methods, and maybe provide some methods to extract the data.
2. `web-poet` prescribes certain things and limits what you can do, but not too much, and as a return you're getting better testability and reusability of your code.
3. Hopefully, now you understand how to write a web scraping framework which uses `web-poet`.
4. Basic `web-poet` usage looks similar to how one could have had refactored the extraction code anyways.

CHAPTER

FIVE

ADDITIONAL REQUESTS

Websites nowadays needs a lot of page interactions to display or load some key information. In most cases, these are done via AJAX requests. Some examples of these are:

- Clicking a button on a page to reveal other similar products.
- Clicking the *"Load More"* button to retrieve more images of a given item.
- Scrolling to the bottom of the page to load more items *(i.e. infinite scrolling)*.
- Hovering on a certain webpage element that reveals a tool-tip containing additional page info.

As such, performing additional requests inside Page Objects are inevitable to properly extract data for some websites.

> **Warning:** Additional requests made inside a Page Object aren't meant to represent the **Crawling Logic** at all. They are simply a low-level way to interact with today's websites which relies on a lot of page interactions to display its contents.

5.1 HttpRequest

Additional requests are defined using a simple data container that represents a generic HTTP Request: *HttpRequest*. Here's an example:

```
import json
import web_poet

request = web_poet.HttpRequest(
    url="https://www.api.example.com/product-pagination/",
    method="POST",
    headers={
        "Content-Type": "application/json;charset=UTF-8"
    },
    body=json.dumps(
        {
            "Page": page_num,
            "ProductID": product_id,
        }
    ).encode("utf-8"),
)

print(request.url)        # https://www.api.example.com/product-pagination/
```

(continues on next page)

(continued from previous page)

```
print(type(request.url))  # <class 'web_poet.page_inputs.http.RequestUrl'>
print(request.method)     # POST

print(type(request.headers)  # <class 'web_poet.page_inputs.HttpRequestHeaders'>
print(request.headers)       # <HttpRequestHeaders('Content-Type': 'application/json;
↪charset=UTF-8')>
print(request.headers.get("content-type"))    # application/json;charset=UTF-8
print(request.headers.get("does-not-exist"))  # None

print(type(request.body))  # <class 'web_poet.page_inputs.HttpRequestBody'>
print(request.body)        # b'{"Page": 1, "ProductID": 123}'
```

There are a few things to take note here:

- `method` is simply a **string**.
- `url` is represented by the *`RequestUrl`* class.
- `headers` is represented by the *`HttpRequestHeaders`* class which resembles a `dict`-like interface. It supports case-insensitive header-key lookups as well as multi-key storage.
  - See `multidict.CIMultiDict` for the set of features since *`HttpRequestHeaders`* simply inherits from it.
- `body` is represented by the *`HttpRequestBody`* class which is simply a subclass of the `bytes` class. Using the `body` param of *`HttpRequest`* needs to have an input argument in `bytes`. In our code example, we've converted it from `str` to `bytes` using the `encode()` string method.

Most of the time though, what you'll be defining would be `GET` requests. Thus, it's perfectly fine to define them as:

```
import web_poet

request = web_poet.HttpRequest("https://api.example.com/product-info?id=123")

print(request.url)        # https://api.example.com/product-info?id=123
print(type(request.url))  # <class 'web_poet.page_inputs.http.RequestUrl'>
print(request.method)     # GET

print(type(request.headers)  # <class 'web_poet.page_inputs.HttpRequestHeaders'>
print(request.headers)       # <HttpRequestHeaders()>
print(request.headers.get("content-type"))    # None
print(request.headers.get("does-not-exist"))  # None

print(type(request.body))  # <class 'web_poet.page_inputs.HttpRequestBody'>
print(request.body)        # b''
```

The key take aways are:

- The default value of `method` is `GET`.
- `headers` still holds *`HttpRequestHeaders`* which doesn't contain anything.
- The same is true for `body` holding an empty *`HttpRequestBody`*.

Now that we know how *`HttpRequest`* are structured, defining them doesn't execute the actual requests at all. In order to do so, we'll need to feed it into the *`HttpClient`* which is defined in the next section (see *HttpClient* tutorial section).

5.2 HttpResponse

HttpResponse is what comes after a *HttpRequest* has been executed. It's typically returned by the methods from *HttpClient* (see *HttpClient* tutorial section) which holds the information regarding the response.

HttpResponse can also be used as a Page Object dependency, e.g. *WebPage* uses it.

Note: The additional requests are expected to perform redirections except when the method is `HEAD`. This means that the *HttpResponse* that you'll be receiving is already the end of the redirection trail.

Let's check out an example to see its internals:

```
import web_poet

response = web_poet.HttpResponse(
    url="https://www.api.example.com/product-pagination/",
    body='{"data": "value "}'.encode("utf-8"),
    status=200,
    headers={"Content-Type": "application/json;charset=UTF-8"}
)

print(response.url)         # https://www.api.example.com/product-pagination/
print(type(response.url))   # <class 'web_poet.page_inputs.http.ResponseUrl'>

print(response.body)            # b'{"data": "value \xf0\x9f\x91\x8d"}'
print(type(response.body))      # <class 'web_poet.page_inputs.HttpResponseBody'>

print(response.status)          # 200
print(type(response.status))    # <class 'int'>

print(response.headers)         # <HttpResponseHeaders('Content-Type': 'application/json;
↪charset=UTF-8')>
print(type(response.headers))  # <class 'web_poet.page_inputs.HttpResponseHeaders'>
print(response.headers.get("content-type"))    # application/json;charset=UTF-8
print(response.headers.get("does-not-exist"))  # None

# These methods are also available:

print(response.body.declared_encoding())    # None
print(response.body.json())                 # {'data': 'value '}

print(response.headers.declared_encoding()) # utf-8

print(response.encoding)                    # utf-8
print(response.text)                        # {"data": "value "}
print(response.json())                      # {'data': 'value '}
```

Despite what the example above showcases, you won't be typically defining *HttpResponse* yourself as it's the implementing framework (see *Supporting Additional Requests*) that's responsible for it. Nonetheless, it's important to understand its underlying structure in order to better access its methods.

Here are the key take aways from the example above:

- `status` is simply an **int**.

- `url` is represented by the *ResponseUrl* class.
- `headers` is represented by the *HttpResponseHeaders* class. It's similar to *HttpRequestHeaders* where it inherits from `multidict.CIMultiDict`, granting it case-insensitive header-key lookups as well as multi-key storage.
  - The **encoding** can be derived using the *declared_encoding()* method. In this example, it was retrieved from the `Content-Type` header.
- `body` is represented by the *HttpResponseBody* class which is simply a subclass of the `bytes` class. Using the `body` param of *HttpResponse* needs to have an input argument in `bytes`. In our code example, we've converted it from `str` to `bytes` using the `encode()` string method.
  - Similar to the headers, the **encoding** can be derived using the *declared_encoding()*. In this case, it returned `None` since no encoding can be derived from the response body.
  - A *json()* method is also available to conveniently access decoded contents from JSON responses. It uses the derived **encoding** to properly decode the contents like the emoji.
- The *HttpResponse* class itself also have these convenient methods:
  - The *encoding()* property method returns the proper encoding of the response based on this hierarchy:
    - user-specified encoding (*using the* `_encoding` *attribute*)
    - BOM from the body
    - header encodings
    - body encodings
  - Instead of accessing the raw bytes values *(which doesn't represent the underlying content properly like the emoji)*, the *text()* property method can be used which takes into account the derived **encoding** when decoding the bytes value.
  - The *json()* method is available as a shortcut to *HttpResponseBody*'s *json()* method.

We've only explored a JSON response as a result from an additional request. Let's take a look at another example having an HTML response:

```
import web_poet

response = web_poet.HttpResponse(
    url="https://www.api.example.com/product-pagination/",
    body=(
        '<html>'
        '  <head>'
        '    <title>Some page</title>'
        '    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">'
        '  </head>'
        '  <body>Sample content </body>'
        '</html>'
    ).encode("utf-8"),
    status=200,
    headers={}
)

print(response.headers.declared_encoding()) # None
print(response.body.declared_encoding())    # utf-8
print(response.encoding)                    # utf-8
```

(continues on next page)

(continued from previous page)

```
print(response.body.json())  # JSONDecodeError
print(response.json())       # JSONDecodeError

print(type(response.selector))  # <class 'parsel.selector.Selector'>

print(response.selector.css("body ::text").get())     # Sample content
print(response.css("body ::text").get())              # Sample content

print(response.selector.xpath("//body/text()").get()) # Sample content
print(response.xpath("//body/text()").get())          # Sample content
```

The key take aways for this example are:

- The **encoding** is derived from the body inside the `meta` tags since the `headers` is empty for this example.
- Since we now have an HTML response, using *`json()`* method would raise a `JSONDecodeError` as a JSON document cannot be parsed from it.
- The *`selector()`* property is an instance of `parsel.selector.Selector`; there are also *`css()`* and *`xpath()`* methods.
  - Usually there's no need to use *`selector()`*, as *`css()`* and *`xpath()`* are available.

5.3 HttpClient

The main interface for executing additional requests would be *`HttpClient`*. It also has full support for `asyncio` enabling developers to perform additional requests asynchronously using `asyncio.gather()`, `asyncio.wait()`, etc. This means that `asyncio` could be used anywhere inside the Page Object, including the *`to_item()`* method.

In the previous section, we've explored how *`HttpRequest`* is defined. Let's see a few quick examples to see how to execute additional requests using the *`HttpClient`*.

5.3.1 Executing a HttpRequest instance

```
import attrs
import web_poet


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

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
        }

        # Simulate clicking on a button that says "View All Images"
        request = web_poet.HttpRequest(f"https://api.example.com/v2/images?id={item[
```

(continues on next page)

(continued from previous page)

```
↪'product_id']}")
        response: web_poet.HttpResponse = await self.http.execute(request)

        item["images"] = response.css(".product-images img::attr(src)").getall()
        return item
```

As the example suggests, we're performing an additional request that allows us to extract more images in a product page that might not be otherwise be possible. This is because in order to do so, an additional button needs to be clicked which fetches the complete set of product images via AJAX.

There are a few things to take note of this example:

- Recall from the *HttpRequest* tutorial section that the default method is `GET`. Thus, the `method` parameter can be omitted for simple `GET` requests.
- We're now using the `async/await` syntax inside the *`to_item()`* method.
- The response from the additional request is of type *`HttpResponse`*.

Tip: Check out the *Batch requests* tutorial section to see how to execute a group of *`HttpRequest`* in batch.

Fortunately, there are already some quick shortcuts on how to perform single additional requests using the *`request()`*, *`get()`*, and *`post()`* methods of *`HttpClient`*. These already define the *`HttpRequest`* and executes it as well.

5.3.2 A simple `GET` request

Let's use the example from the previous section and use the *`get()`* method on it.

```
import attrs
import web_poet


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

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
        }

        # Simulates clicking on a button that says "View All Images"
        response: web_poet.HttpResponse = await self.http.get(
            f"https://api.example.com/v2/images?id={item['product_id']}"
        )
        item["images"] = response.css(".product-images img::attr(src)").getall()
        return item
```

There are a few things to take note in this example:

- A `GET` request can be done via *`HttpClient`*'s *`get()`* method.

- There is no need create an instance of *HttpRequest* when *get()* is used.

5.3.3 A POST request with *header* and *body*

Let's see another example which needs `headers` and `body` data to process additional requests.

In this example, we'll paginate related items in a carousel. These are usually lazily loaded by the website to reduce the amount of information rendered in the DOM that might not otherwise be viewed by all users anyway.

Thus, additional requests inside the Page Object are typically needed for it:

```
import attrs
import web_poet


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

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
            "related_product_ids": self.parse_related_product_ids(self),
        }

        # Simulates "scrolling" through a carousel that loads related product items
        response: web_poet.HttpResponse = await self.http.post(
            url="https://www.api.example.com/related-products/",
            headers={
                "Content-Type": "application/json;charset=UTF-8"
            },
            body=json.dumps(
                {
                    "Page": 2,
                    "ProductID": item["product_id"],
                }
            ).encode("utf-8"),
        )
        item["related_product_ids"].extend(self.parse_related_product_ids(response))
        return item

    @staticmethod
    def parse_related_product_ids(response_page) -> List[str]:
        return response_page.css("#main .related-products ::attr(product-id)").getall()
```

Here's the key takeaway in this example:

- Similar to *HttpClient*'s *get()* method, a *post()* method is also available. It is often used to submit forms.

5.3.4 Other Single Requests

The *get()* and *post()* methods are merely quick shortcuts for *request()*:

```
client = HttpClient()

url = "https://api.example.com/v1/data"
headers = {"Content-Type": "application/json;charset=UTF-8"}
body = b'{"data": "value"}'

# These are the same:
response = await client.get(url)
response = await client.request(url, method="GET")

# The same goes for these:
response = await client.post(url, headers=headers, body=body)
response = await client.request(url, method="POST", headers=headers, body=body)
```

Thus, apart from the common `GET` and `POST` HTTP methods, you can use *request()* for them (*e.g.* `HEAD`, `PUT`, `DELETE`, etc).

5.3.5 Batch requests

We can also choose to process requests by **batch** instead of sequentially or one by one (e.g. using *execute()*). The *batch_execute()* method can be used for this which accepts an arbitrary number of *HttpRequest* instances.

Let's modify the example in the previous section to see how it can be done.

The difference for this code example from the previous section is that we're increasing the pagination from only the **2nd page** into the **10th page**. Instead of calling a single *post()* method, we're creating a list of *HttpRequest* to be executed in batch using the *batch_execute()* method.

```
from typing import List

import attrs
import web_poet


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

    default_pagination_limit = 10

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
            "related_product_ids": self.parse_related_product_ids(self),
        }

        requests: List[web_poet.HttpRequest] = [
            self.create_request(item["product_id"], page_num=page_num)
```

(continues on next page)

(continued from previous page)

```
            for page_num in range(2, self.default_pagination_limit)
        ]
        responses: List[web_poet.HttpResponse] = await self.http.batch_execute(*requests)
        related_product_ids = [
            id_
            for response in responses
            for product_ids in self.parse_related_product_ids(response)
            for id_ in product_ids
        ]

        item["related_product_ids"].extend(related_product_ids)
        return item

    def create_request(self, product_id, page_num=2):
        # Simulates "scrolling" through a carousel that loads related product items
        return web_poet.HttpRequest(
            url="https://www.api.example.com/product-pagination/",
            method="POST",
            headers={
                "Content-Type": "application/json;charset=UTF-8"
            },
            body=json.dumps(
                {
                    "Page": page_num,
                    "ProductID": product_id,
                }
            ).encode("utf-8"),
        )

    @staticmethod
    def parse_related_product_ids(response_page) -> List[str]:
        return response_page.css("#main .related-products ::attr(product-id)").getall()
```

The key takeaways for this example are:

- An *HttpRequest* can be instantiated to represent a Generic HTTP Request. It only contains the HTTP Request information for now and isn't executed yet. This is useful for creating factory methods to help create requests without any download execution at all.
- *HttpClient* has a *batch_execute()* method that can process a list of *HttpRequest* instances asynchronously together.

Tip: The *batch_execute()* method can execute multiple *HttpRequest* instances. For example, it could be a mixture of `GET` and `POST` requests or even representing requests for various parts of the page altogether.

Processing the additional requests in batch is useful since it takes advantage of async execution which could be faster in certain cases *(assuming you're allowed to perform HTTP requests in parallel)*.

Nonetheless, you can still use the *batch_execute()* method to execute a single *HttpRequest* instance.

Note: The *batch_execute()* method is a simple wrapper over `asyncio.gather()`. Developers are free to use other functionalities available inside `asyncio` to handle multiple requests.

For example, `asyncio.as_completed()` can be used to process the first response from a group of requests as early as possible. However, the order could be shuffled.

5.4 Handling Exceptions in Page Objects

Let's have a look at how we could handle exceptions when performing additional requests inside Page Objects. For this example, let's improve the code snippet from the previous subsection named: *A simple GET request*.

```
import logging

import attrs
import web_poet

logger = logging.getLogger(__name__)


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

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
        }

        try:
            # Simulates clicking on a button that says "View All Images"
            response: web_poet.HttpResponse = await self.http.get(
                f"https://api.example.com/v2/images?id={item['product_id']}"
            )
        except web_poet.exceptions.HttpRequestError as err:
            logger.warning(
                f"Unable to request images for product ID '{item['product_id']}' "
                f"using this request: {err.request}"
            )
        except web_poet.exceptions.HttpResponseError as err:
            logger.warning(
                f"Received a {err.response.status} response status for product ID "
                f"'{item['product_id']}' from this URL: {err.request.url}"
            )
        else:
            item["images"] = response.css(".product-images img::attr(src)").getall()

        return item
```

In this code example, the code became more resilient on cases where it wasn't possible to retrieve more images using the website's public API. It could be due to anything like *SSL errors*, *connection errors*, *page not found*, etc.

Using *`HttpClient`* to execute requests raises exceptions with the base class of type *`web_poet.exceptions.http.HttpError`* irregardless of how the HTTP Downloader is implemented. From our example above, we could've simply

used the *web_poet.exceptions.http.HttpError* base error. However, it's ambiguous in the sense that the error could originate during the HTTP Request execution or when receiving the HTTP Response.

A more specific *web_poet.exceptions.http.HttpRequestError* exception is raised when the *HttpRequest* was being handled while the *web_poet.exceptions.http.HttpResponseError* is raised when receiving a response with an HTTP error. Notice from the example that the exceptions have the attributes like `request` and `response` which are respective instance of *HttpRequest* and *HttpResponse*. Accessing them would be useful to debug and log the problems.

Note that *web_poet.exceptions.http.HttpResponseError* only occurs when receiving responses with status codes in the `400-5xx` range. However, this behavior could be altered by using the `allow_status` param in the methods of *HttpClient*.

Note: In the future, more specific exceptions which inherits from the base *web_poet.exceptions.http.HttpError* exception would be available. This should allow developers writing Page Objects to properly identify what went wrong and act specifically based on the problem.

Let's take another example when executing requests in batch as opposed to using single requests via these methods of the *HttpClient*: *request()*, *get()*, and *post()*.

For this example, let's improve the code snippet from the previous subsection named: *Batch requests*.

```
import logging
from typing import List, Union

import attrs
import web_poet


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

    default_pagination_limit = 10

    async def to_item(self):
        item = {
            "url": self.url,
            "name": self.css("#main h3.name ::text").get(),
            "product_id": self.css("#product ::attr(product-id)").get(),
            "related_product_ids": self.parse_related_product_ids(self),
        }

        requests: List[web_poet.HttpRequest] = [
            self.create_request(item["product_id"], page_num=page_num)
            for page_num in range(2, self.default_pagination_limit)
        ]

        try:
            responses: List[web_poet.HttpResponse] = await self.http.batch_
↪execute(*requests)
        except web_poet.exceptions.HttpError:
            logger.warning(
                f"Unable to request for more related products for product ID: {item[
```

(continues on next page)

(continued from previous page)

```
↪'product_id']}"
            )
        else:
            related_product_ids = []
            for response in responses:
                related_product_ids.extend(
                    [
                        id_
                        for product_ids in self.parse_related_product_ids(response)
                        for id_ in product_ids
                    ]
                )
            item["related_product_ids"].extend(related_product_ids)

        return item

    def create_request(self, product_id, page_num=2):
        # Simulates "scrolling" through a carousel that loads related product items
        return web_poet.HttpRequest(
            url="https://www.api.example.com/product-pagination/",
            method="POST",
            headers={
                "Content-Type": "application/json;charset=UTF-8"
            },
            body=json.dumps(
                {
                    "Page": page_num,
                    "ProductID": product_id,
                }
            ).encode("utf-8"),
        )

    @staticmethod
    def parse_related_product_ids(response_page) -> List[str]:
        return response_page.css("#main .related-products ::attr(product-id)").getall()
```

Handling exceptions using *batch_execute()* remains largely the same. However, the main difference is that you may be wasting perfectly good responses just because a single request from the batch ruined it. Notice that we're using the base exception class of *web_poet.exceptions.http.HttpError* to account for any type of errors, both during the HTTP Request execution and when receiving the response.

An alternative approach would be salvaging good responses altogether. For example, you've sent out 10 *HttpRequest* and only 1 of them had an exception during processing. You can still get the data from 9 of the *HttpResponse* by passing the parameter `return_exceptions=True` to *batch_execute()*.

This means that any exceptions raised during the HTTP execution are returned alongside any of the successful responses. The return type of *batch_execute()* could be a mixture of *HttpResponse* and *web_poet.exceptions.http.HttpError* (*and its exception subclasses*).

Here's an example:

```
# Revised code snippet from the to_item() method

requests: List[web_poet.HttpRequest] = [
```

(continues on next page)

(continued from previous page)

```
    self.create_request(item["product_id"], page_num=page_num)
    for page_num in range(2, self.default_pagination_limit)
]

responses: List[Union[web_poet.HttpResponse, web_poet.exceptions.HttpError]] = (
    await self.http.batch_execute(*requests, return_exceptions=True)
)

related_product_ids = []
for i, response in enumerate(responses):
    if isinstance(response, web_poet.exceptions.HttpError):
        logger.warning(
            f"Unable to request related products for product ID '{item['product_id']}' "
            f"using this request: {requests[i]}. Reason: {response}."
        )
        continue
    related_product_ids.extend(
        [
            id_
            for product_ids in self.parse_related_product_ids(response)
            for id_ in product_ids
        ]
    )

item["related_product_ids"].extend(related_product_ids)
return item
```

From the example above, we're now checking the list of responses to see if any exceptions are included in it. If so, we're simply logging it down and ignoring it. In this way, perfectly good responses can still be processed through.

CHAPTER

SIX

FIELDS

6.1 Background

It is common for Page Objects not to put all the extraction code to the `to_item()` method, but create properties or methods to extract individual attributes, a method or property per attribute:

```
import attrs
from web_poet import ItemPage, HttpResponse


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

    @property
    def name(self):
        return self.response.css(".name").get()

    @property
    def price(self):
        return self.response.css(".price").get()

    def to_item(self) -> dict:
        return {
            'name': self.name,
            'price': self.price
        }
```

This approach has 2 main advantages:

1. Often the code looks cleaner this way, it's easier to follow.
2. The resulting page object becomes more flexible and reusable: if not all data extracted in the `to_item()` method is needed, user can use properties for individual attributes. It's more efficient than running `to_item()` and only using some of the result.

However, writing and maintaining `to_item()` method can get tedious, especially if there is a lot of properties.

6.2 @field decorator

To aid writing Page Objects in this style, `web-poet` provides the *@web_poet.field* decorator:

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


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

    @field
    def name(self):
        return self.response.css(".name").get()

    @field
    def price(self):
        return self.response.css(".price").get()
```

ItemPage has a default `to_item()` implementation: it uses all the properties created with the *@field* decorator, and returns a dict with the result, where keys are method names, and values are property values. In the example above, `to_item()` returns a `{"name": ..., "price": ...}` dict with the extracted data.

Methods annotated with the *@field* decorator become properties; for a `page = MyPage(...)` instance you can access them as `page.name`.

It's important to note that the default *ItemPage.to_item()* implementation is an `async def` function - make sure to await its result: `item = await page.to_item()`

6.3 Asynchronous fields

The reason *ItemPage* provides an async `to_item` method by default is that both regular and `async def` fields are supported.

For example, you might need to send *Additional Requests* to extract some of the attributes:

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


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

    @field
    def name(self):
        return self.response.css(".name").get()

    @field
    async def price(self):
```

(continues on next page)

(continued from previous page)

```
        resp = await self.http.get("...")
        return resp.json()['price']
```

6.3.1 Using Page Objects with async fields

If you want to use a Page Object with async fields without calling its `to_item` method, make sure to await the field when needed, and not await it when that's not needed:

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

This is not ideal, because now the code which needs to use a page object must be aware if a field is sync or async. If a field needs to be changed from being sync to `async def` (or the other way around), e.g. because of a website change, all the code which uses this page object must be updated.

One approach to solve it is to always define all fields as `async def`. It works, but it makes the page objects harder to use in non-async environments.

Instead of doing this, you can also use *ensure_awaitable()* utility function when accessing the fields:

```
from web_poet.utils import ensure_awaitable

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

Now any field can be converted from sync to async, or the other way around, and the code would keep working.

6.4 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):
    return s.replace('\t', ' ')

class MyPage(ItemPage):
    response: HttpResponse

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

Note that while processors can be applied to async fields, they need to be sync functions themselves.

It's also possible to implement field cleaning and processing in `to_item` but in that case accessing a field directly will return the value without processing, so it's preferable to use field processors instead.

6.5 Item Classes

In all previous examples, `to_item` methods are returning `dict` instances. It is common to use item classes (e.g. dataclasses or attrs instances) instead of unstructured dicts to hold the data:

```
import attrs
from web_poet import ItemPage, HttpResponse

@attrs.define
class Product:
    name: str
    price: str


@attrs.define
class ProductPage(ItemPage):
    # ...
    def to_item(self) -> Product:
        return Product(
            name=self.name,
            price=self.price
        )
```

web_poet.fields supports it, by allowing to parametrize *ItemPage* with an item class:

```
@attrs.define
class ProductPage(ItemPage[Product]):
    # ...
```

When *ItemPage* is parametrized with an item class, its `to_item()` method starts to return item instances, instead of `dict` instances. In the example above `ProductPage.to_item` method returns `Product` instances.

Defining an item class may be an overkill if you only have a single Page Object, but item classes are of a great help when

- you need to extract data in the same format from multiple websites, or
- if you want to define the schema upfront.

6.5.1 Error prevention

Item classes play particularly well with the *@field* decorator, preventing some of the errors, which may happen if results are plain "dicts".

Consider the following badly written page object:

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

@attrs.define
class Product:
    name: str
    price: str
```

(continues on next page)

(continued from previous page)

```
@attrs.define
class ProductPage(ItemPage[Product]):
    response: HttpResponse

    @field
    def nane(self):
        return self.response.css(".name").get()
```

Because the `Product` item class is used, a typo ("nane" instead of "name") is detected at runtime: the creation of a `Product` instance would fail with a `TypeError`, because of the unexpected keyword argument "nane".

After fixing it (renaming "nane" method to "name"), another error is going to be detected: the `price` argument is required, but there is no extraction method for this attribute, so `Product.__init__` will raise another `TypeError`, indicating that a required argument is missing.

Without an item class, none of these errors are detected.

6.5.2 Changing Item Class

Let's say there is a Page Object implemented, which outputs some standard item. Maybe there is a library of such Page Objects available. But for a particular project we might want to output an item of a different type:

- some attributes of the standard item might not be needed;
- there might be a need to implement extra attributes, which are not available in the standard item;
- names of attributes might be different.

There are a few ways to approach it. If items are very different, using the original Page Object as a dependency is a good approach:

```
import attrs
from my_library import FooPage, StandardItem
from web_poet import ItemPage, HttpResponse, field, ensure_awaitable

@attrs.define
class CustomItem:
    new_name: str
    new_price: str

@attrs.define
class CustomFooPage(ItemPage[CustomItem]):
    response: HttpResponse
    standard: FooPage

    @field
    async def new_name(self):
        orig_name = await ensure_awaitable(self.standard.name)
        orig_brand = await ensure_awaitable(self.standard.brand)
        return f"{orig_brand}: {orig_name}"

    @field
    async def new_price(self):
        ...
```

However, if items are similar, and share many attributes, this approach could lead to boilerplate code. For example, you might be extending an item with a new field, and it'd be required to duplicate definitions for all other fields.

Instead of using dependency injection you can make your Page Object a subclass of the original Page Object; that's a nice way to add a new field to the item:

```
import attrs
from my_library import FooPage, StandardItem
from web_poet import field, Returns

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

@attrs.define
class CustomFooPage(FooPage, Returns[CustomItem]):

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

Note how *Returns* is used as one of the base classes of `CustomFooPage`; it allows to change the item class returned by a page object.

Removing fields (as well as renaming) is a bit more tricky.

The caveat is that by default *ItemPage* uses all fields defined as `@field` to produce an item, passing all these values to item's `__init__` method. So, if you follow the previous example, and inherit from the "base", "standard" Page Object, there could be a `@field` from the base class which is not present in the `CustomItem`. It'd be still passed to `CustomItem.__init__`, causing an exception.

One way to solve it is to make the original Page Object a dependency instead of inheriting from it, as explained in the beginning.

Alternatively, you can use `skip_nonitem_fields=True` class argument - it tells *to_item()* to skip `@fields` which are not defined in the item:

```
@attrs.define
class CustomItem:
    # let's pick only 1 attribute from StandardItem, nothing more
    name: str

class CustomFooPage(FooPage, Returns[CustomItem], skip_nonitem_fields=True):
    pass
```

Here, `CustomFooPage.to_item` only uses `name` field of the `FooPage`, ignoring all other fields defined in `FooPage`, because `skip_nonitem_fields=True` is passed, and `name` is the only field `CustomItem` supports.

To recap:

- Use `Returns[NewItemType]` to change the item class in a subclass.
- Don't use `skip_nonitem_fields=True` when your Page Object corresponds to an item exactly, or when you're only adding fields. This is a safe approach, which allows to detect typos in field names, even for optional fields.
- Use `skip_nonitem_fields=True` when it's possible for the Page Object to contain more `@fields` than defined in the item class, e.g. because Page Object is inherited from some other base Page Object.

6.6 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 web_poet import ItemPage, HttpResponse, HttpClient

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

    async def to_item(self):
        api_url = self.response.css("...").get()
        api_response = await self.http.get(api_url).json()
        return {
            'name': self.response.css(".name ::text").get(),
            'price': api_response["price"],
            'sku': api_response["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 web_poet import ItemPage, HttpResponse, HttpClient, field, cached_method

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

    @cached_method
    async def api_response(self):
        api_url = self.response.css("...").get()
        return await self.http.get(api_url).json()

    @field
    def name(self):
        return self.response.css(".name ::text").get()

    @field
    async def price(self):
        api_response = await self.api_response()
        return api_response["price"]

    @field
    async def sku(self):
        api_response = await self.api_response()
        return api_response["sku"]
```

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`.

6.6.1 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.

6.7 Field metadata

`web-poet` allows to store arbitrary information for each field, using `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()
my_field_meta = fields_dict["my_field"].meta
```

(continues on next page)

(continued from previous page)

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

CHAPTER

SEVEN

APPLY RULES

7.1 Basic Usage

7.1.1 @handle_urls

web-poet provides a *handle_urls()* decorator, which allows to declare how a page object can be used (applied):

- for which websites / URL patterns it works,
- which data type (item classes) it can return,
- which page objects it can replace (override; more on this later).

```
from web_poet import ItemPage, handle_urls
from my_items import MyItem

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

`handle_urls("example.com")` can serve as documentation, but it also enables getting the information about page objects programmatically. The information about all page objects decorated with *handle_urls()* is stored in `web_poet.default_registry`, which is an instance of *RulesRegistry*. In the example above, the following *ApplyRule* is added to the registry:

```
ApplyRule(
    for_patterns=Patterns(include=('example.com',), exclude=(), priority=500),
    use=<class 'MyPage'>,
    instead_of=None,
    to_return=<class 'my_items.MyItem'>,
    meta={}
)
```

Note how `rule.to_return` is set to `MyItem` automatically. Such rules can be used by libraries like scrapy-poet. For example, if a spider needs to extract `MyItem` from some page on the `example.com` website, scrapy-poet now knows that `MyPage` page object can be used.

7.1.2 Specifying the URL patterns

`handle_urls()` decorator uses url-matcher library to define the URL rules. Some examples:

```
# page object can be applied on any URL from the example.com domain,
# or from any of its subdomains
@handle_urls("example.com")

# page object can be applied on example.com pages under /products/ path
@handle_urls("example.com/products/")

# page object can be applied on any URL from example.com, but only if
# it contains "productId=..." in the query string
@handle_urls("example.com?productId=*")
```

The string passed to *handle_urls()* is converted to a `url_matcher.matcher.Patterns` instance. Please consult with the url-matcher documentation to learn more about the possible rules; it is pretty flexible. You can exclude patterns, use wildcards, require certain query parameters to be present and ignore others, etc. Unlike regexes, this mini-language "understands" the URL structure.

7.2 Overrides

handle_urls() can be used to declare that a particular Page Object could (and should) be used *instead of* some other Page Object on certain URL patterns:

```
from web_poet import ItemPage, handle_urls
from my_items import Product
from my_pages import DefaultProductPage

@handle_urls("site1.example.com", instead_of=DefaultProductPage)
class Site1ProductPage(ItemPage[Product]):
    # ...

@handle_urls("site2.example.com", instead_of=DefaultProductPage)
class Site2ProductPage(ItemPage[Product]):
    # ...
```

This concept is a bit more advanced than the basic `handle_urls` usage ("this Page Object can return `MyItem` on example.com website").

A common use case is a "generic", or a "template" spider, which uses some default implementation of the extraction, and allows to replace it ("override") on specific websites or URL patterns.

This default page extraction (`DefaultProductPage` in the example) can be based on semantic markup, Machine Learning, heuristics, or just be empty. Page Objects which can be used instead of the default (`Site1ProductPage`, `Site2ProductPage`) are commonly written using XPath or CSS selectors, with website-specific rules.

Libraries like scrapy-poet allow to create such "generic" spiders by using the information declared via `handle_urls(..., instead_of=...)`.

7.2.1 Example Use Case

Let's explore an example use case for the Overrides concept.

Suppose we're using Page Objects for our broadcrawl project which explores eCommerce websites to discover product pages. It wouldn't be entirely possible for us to create parsers for all websites since we don't know which sites we're going to crawl beforehand.

However, we could at least create a generic Page Object to support parsing of some fields in well-known locations of product information like `<title>`. This enables our broadcrawler to at least parse some useful information. Let's call such a Page Object to be `GenericProductPage`.

Assuming that one of our project requirements is to fully support parsing of the *top 3 eCommerce websites*, then we'd need to create a Page Object for each one to parse more specific fields.

Here's where the Overrides concept comes in:

1. The `GenericProductPage` is used to parse all eCommerce product pages *by default*.
2. Whenever one of our declared URL rules matches with a given page URL, then the Page Object associated with that rule *overrides (or replaces)* the default `GenericProductPage`.

This enables us to conveniently declare which Page Object would be used for a given webpage *(based on a page's URL pattern)*.

Let's see this in action by declaring the Overrides in the Page Objects below.

7.2.2 Creating Overrides

To simplify the code examples in the next few subsections, let's assume that these item classes have been predefined:

```
import attrs


@attrs.define
class Product:
    product_title: str
    regular_price: float


@attrs.define
class SimilarProduct:
    product_title: str
    regular_price: float
```

Page Object

Let's take a look at how the following code is structured:

```
from web_poet import handle_urls, WebPage


class GenericProductPage(WebPage):
    def to_item(self) -> Product:
        return Product(product_title=self.css("title::text").get())
```

(continues on next page)

(continued from previous page)

```
@handle_urls("some.example", instead_of=GenericProductPage)
class ExampleProductPage(WebPage):
    ...  # more specific parsing


@handle_urls("another.example", instead_of=GenericProductPage, exclude="/digital-goods/")
class AnotherExampleProductPage(WebPage):
    ...  # more specific parsing


@handle_urls(["dual.example/shop/?product=*", "uk.dual.example/store/?pid=*"], instead_
↪of=GenericProductPage)
class DualExampleProductPage(WebPage):
    ...  # more specific parsing
```

The code above declares that:

- The Page Objects return `Product` and `SimilarProduct` item classes. Returning item classes is a preferred approach as explained in the *Fields* section.
- For sites that match the `some.example` pattern, `ExampleProductPage` would be used instead of `GenericProductPage`.
- The same is true for `DualExampleProductPage` where it is used instead of `GenericProductPage` for two URL patterns which works as something like:
  - (match) https://www.dual.example/shop/electronics/?product=123
  - (match) https://www.dual.example/shop/books/paperback/?product=849
  - (NO match) https://www.dual.example/on-sale/books/?product=923
  - (match) https://www.uk.dual.example/store/kitchen/?pid=776
  - (match) https://www.uk.dual.example/store/?pid=892
  - (NO match) https://www.uk.dual.example/new-offers/fitness/?pid=892
- On the other hand, `AnotherExampleProductPage` is used instead of `GenericProductPage` when we're handling pages that match the `another.example` URL Pattern, which doesn't contain `/digital-goods/` in its URL path.

Tip: The URL patterns declared in the `@handle_urls` decorator can still be further customized. You can read some of the specific parameters in the *API section* of *web_poet.handle_urls()*.

Item Class

An alternative approach for the Page Object Overrides example above is to specify the returned item class. For example, we could change the previous example into the following:

```
from web_poet import handle_urls, WebPage


class GenericProductPage(WebPage[Product]):
    def to_item(self) -> Product:
        return Product(product_title=self.css("title::text").get())


@handle_urls("some.example")
class ExampleProductPage(WebPage[Product]):
    ...  # more specific parsing


@handle_urls("another.example", exclude="/digital-goods/")
class AnotherExampleProductPage(WebPage[Product]):
    ...  # more specific parsing


@handle_urls(["dual.example/shop/?product=*", "uk.dual.example/store/?pid=*"])
class DualExampleProductPage(WebPage[Product]):
    ...  # more specific parsing
```

Let's break this example down:

- The URL patterns are exactly the same as with the previous code example.
- The `@handle_urls` decorator determines the item class to return (i.e. `Product`) from the decorated Page Object.
- The `instead_of` parameter can be omitted in lieu of the derived Item Class from the Page Object which becomes the `to_return` attribute in *`ApplyRule`* instances. This means that:
  - If a `Product` item class is requested for URLs matching with the "some.example" pattern, then the `Product` item class would come from the `to_item()` method of `ExampleProductPage`.
  - Similarly, if a page with a URL matches with "another.example" without the "/digital-goods/" path, then the `Product` item class comes from the `AnotherExampleProductPage` Page Object.
  - However, if a `Product` item class is requested matching with the URL pattern of "dual.example/shop/?product=*", a `SimilarProduct` item class is returned by the `DualExampleProductPage`'s `to_item()` method instead.

Specifying the item class that a Page Object returns makes it possible for web-poet frameworks to make Page Object usage transparent to end users.

For example, a web-poet framework could implement a function like:

```
item = get_item(url, item_class=Product)
```

Here there is no reference to the Page Object being used underneath, you only need to indicate the desired item class, and the web-poet framework automatically determines the Page Object to use based on the specified URL and the specified item class.

Note, however, that web-poet frameworks are encouraged to also allow getting a Page Object instead of an item class instance, for scenarios where end users wish access to Page Object attributes and methods.

Combination

Of course, you can use the combination of both which enables you to specify in either contexts of Page Objects and item classes.

```
from web_poet import handle_urls, WebPage


class GenericProductPage(WebPage[Product]):
    def to_item(self) -> Product:
        return Product(product_title=self.css("title::text").get())


@handle_urls("some.example", instead_of=GenericProductPage)
class ExampleProductPage(WebPage[Product]):
    ...  # more specific parsing


@handle_urls("another.example", instead_of=GenericProductPage, exclude="/digital-goods/")
class AnotherExampleProductPage(WebPage[Product]):
    ...  # more specific parsing


@handle_urls(["dual.example/shop/?product=*", "uk.dual.example/store/?pid=*"], instead_
↪of=GenericProductPage)
class DualExampleProductPage(WebPage[SimilarProduct]):
    ...  # more specific parsing
```

See the next *Retrieving all available rules* section to observe what are the actual *ApplyRule* that were created by the `@handle_urls` decorators.

7.3 Working with rules

7.3.1 Retrieving all available rules

The *get_rules()* method from the `web_poet.default_registry` allows retrieval of all *ApplyRule* in the given registry. Following from our example above in the *Combination* section, using it would be:

```
from web_poet import default_registry

# Retrieves all ApplyRules that were registered in the registry
rules = default_registry.get_rules()

for r in rules:
    print(r)
# ApplyRule(for_patterns=Patterns(include=('some.example',), exclude=(), priority=500),␣
↪use=<class 'ExampleProductPage'>, instead_of=<class 'GenericProductPage'>, to_return=
↪<class 'Product'>, meta={})
# ApplyRule(for_patterns=Patterns(include=('another.example',), exclude=('/digital-goods/',
↪), priority=500), use=<class 'AnotherExampleProductPage'>, instead_of=<class
↪'GenericProductPage'>, to_return=<class 'Product'>, meta={})
# ApplyRule(for_patterns=Patterns(include=('dual.example/shop/?product=*', 'uk.dual.
```

(continues on next page)

(continued from previous page)

```
↪example/store/?pid=*'), exclude=(), priority=500), use=<class 'DualExampleProductPage'>,␣
↪instead_of=<class 'GenericProductPage'>, to_return=<class 'SimilarProduct'>, meta={})
```

Remember that using `@handle_urls` to annotate the Page Objects would result in the *ApplyRule* to be written into `web_poet.default_registry`.

Warning: *get_rules()* relies on the fact that all essential packages/modules which contains the *web_poet.handle_urls()* decorators are properly loaded *(i.e imported)*.

Thus, for cases like importing and using Page Objects from other external packages, the `@handle_urls` decorators from these external sources must be read and processed properly. This ensures that the external Page Objects have all of their *ApplyRule* present.

This can be done via the function named *consume_modules()*. Here's an example:

```
from web_poet import default_registry, consume_modules

consume_modules("external_package_A.po", "another_ext_package.lib")
rules = default_registry.get_rules()
```

The next section explores this caveat further.

7.3.2 Using rules from External Packages

Developers have the option to import existing Page Objects alongside the *ApplyRule* attached to them. This section aims to showcase different scenarios that come up when using multiple Page Object Projects.

Using all available ApplyRules from multiple Page Object Projects

Let's suppose we have the following use case before us:

- An **external** Python package named `ecommerce_page_objects` is available which contains Page Objects for common websites.
- Another similar **external** package named `gadget_sites_page_objects` is available for even more specific websites.
- Your project's objective is to handle as much eCommerce websites as you can.
  - Thus, you'd want to use the already available packages above and perhaps improve on them or create new Page Objects for new websites.

Remember that all of the *ApplyRule* are declared by annotating Page Objects using the *web_poet.handle_urls()* via `@handle_urls`. Thus, they can easily be accessed using the *get_rules()* of `web_poet.default_registry`.

This can be done something like:

```
from web_poet import default_registry, consume_modules

#  Remember that this wouldn't retrieve any rules at all since the
# ``@handle_urls`` decorators are NOT properly loaded.
rules = default_registry.get_rules()
print(rules)  # []
```

(continues on next page)

(continued from previous page)

```
#  Instead, you need to run the following so that all of the Page
# Objects in the external packages are recursively imported.
consume_modules("ecommerce_page_objects", "gadget_sites_page_objects")
rules = default_registry.get_rules()

# The collected rules would then be as follows:
print(rules)
# 1. ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_1.EcomSite1'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
# 4. ApplyRule(for_patterns=Patterns(include=['site_3.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_3.GadgetSite3'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
```

Note: Once *consume_modules()* is called, then all external Page Objects are recursively imported and available for the entire runtime duration. Calling *consume_modules()* again makes no difference unless a new set of modules are provided.

Using only a subset of the available ApplyRules

Suppose that the use case from the previous section has changed wherein a subset of *ApplyRule* would be used. This could be achieved by using the *search()* method which allows for convenient selection of a subset of rules from a given registry.

Here's an example of how you could manually select the rules using the *search()* method instead:

```
from web_poet import default_registry, consume_modules
import ecommerce_page_objects, gadget_sites_page_objects

consume_modules("ecommerce_page_objects", "gadget_sites_page_objects")

ecom_rules = default_registry.search(instead_of=ecommerce_page_objects.EcomGenericPage)
print(ecom_rules)
# ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[], priority=500),␣
↪use=<class 'ecommerce_page_objects.site_1.EcomSite1'>, instead_of=<class 'ecommerce_page_
↪objects.EcomGenericPage'>, to_return=None, meta={})
# ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[], priority=500),␣
↪use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class 'ecommerce_page_
↪objects.EcomGenericPage'>, to_return=None, meta={})

gadget_rules = default_registry.search(use=gadget_sites_page_objects.site_3.GadgetSite3)
print(gadget_rules)
# ApplyRule(for_patterns=Patterns(include=['site_3.example'], exclude=[], priority=500),␣
↪use=<class 'gadget_sites_page_objects.site_3.GadgetSite3'>, instead_of=<class 'gadget_
```

(continues on next page)

(continued from previous page)

```
↪sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})

rules = ecom_rules + gadget_rules
print(rules)
# ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[], priority=500),␣
↪use=<class 'ecommerce_page_objects.site_1.EcomSite1'>, instead_of=<class 'ecommerce_page_
↪objects.EcomGenericPage'>, to_return=None, meta={})
# ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[], priority=500),␣
↪use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class 'ecommerce_page_
↪objects.EcomGenericPage'>, to_return=None, meta={})
# ApplyRule(for_patterns=Patterns(include=['site_3.example'], exclude=[], priority=500),␣
↪use=<class 'gadget_sites_page_objects.site_3.GadgetSite3'>, instead_of=<class 'gadget_
↪sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
```

As you can see, using the *search()* method allows you to conveniently select for *ApplyRule* which conform to a specific criteria. This allows you to conveniently drill down to which *ApplyRule* you're interested in using.

Creating a new registry

After gathering all the pre-selected rules, we can then store it in a new instance of *RulesRegistry* in order to separate it from the `default_registry` which contains all of the rules. We can use the `RulesRegistry(rules=...)` for this:

```
from web_poet import RulesRegistry

my_new_registry = RulesRegistry(rules=rules)
```

Improving on external Page Objects

There would be cases wherein you're using Page Objects with *ApplyRule* from external packages only to find out that a few of them lacks some of the fields or features that you need.

Let's suppose that we wanted to use *all* of the *ApplyRule* similar to this section: *Using all available ApplyRules from multiple Page Object Projects*. However, the `EcomSite1` Page Object needs to properly handle some edge cases where some fields are not being extracted properly. One way to fix this is to subclass the said Page Object and improve its `to_item()` method, or even creating a new class entirely. For simplicity, let's have the first approach as an example:

```
from web_poet import default_registry, consume_modules, handle_urls
import ecommerce_page_objects, gadget_sites_page_objects

consume_modules("ecommerce_page_objects", "gadget_sites_page_objects")
rules = default_registry.get_rules()

# The collected rules would then be as follows:
print(rules)
# 1. ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_1.EcomSite1'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
```

(continues on next page)

(continued from previous page)

```
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
# 4. ApplyRule(for_patterns=Patterns(include=['site_3.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_3.GadgetSite3'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})

@handle_urls("site_1.example", instead_of=ecommerce_page_objects.EcomGenericPage,␣
↪priority=1000)
class ImprovedEcomSite1(ecommerce_page_objects.site_1.EcomSite1):
    def to_item(self):
        ...  # call super().to_item() and improve on the item's shortcomings

rules = default_registry.get_rules()
print(rules)
# 1. ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_1.EcomSite1'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
# 4. ApplyRule(for_patterns=Patterns(include=['site_3.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_3.GadgetSite3'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
# 5. ApplyRule(for_patterns=Patterns(include=['site_1.example'], exclude=[],␣
↪priority=1000), use=<class 'my_project.ImprovedEcomSite1'>, instead_of=<class 'ecommerce_
↪page_objects.EcomGenericPage'>, to_return=None, meta={})
```

Notice that we're adding a new *ApplyRule* for the same URL pattern for `site_1.example`.

When the time comes that a Page Object needs to be selected when parsing `site_1.example` and it needs to replace `ecommerce_page_objects.EcomGenericPage`, rules **#1** and **#5** will be the choices. However, since we've assigned a much **higher priority** for the new rule in **#5** than the default `500` value, rule **#5** will be chosen because of its higher priority value.

More details on this in the *Priority Resolution* subsection.

Handling conflicts when using Multiple External Packages

You might've observed from the previous section that retrieving the list of all *ApplyRule* from two different external packages may result in a conflict.

We can take a look at the rules for **#2** and **#3** when we were importing all available rules:

```
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'ecommerce_page_objects.EcomGenericPage'>, to_return=None, meta={})
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'gadget_sites_page_objects.GadgetGenericPage'>, to_return=None, meta={})
```

However, it's technically **NOT** a *conflict*, **yet**, since:

- `ecommerce_page_objects.site_2.EcomSite2` would only be used in **site_2.example** if `ecommerce_page_objects.EcomGenericPage` is to be replaced.
- The same case with `gadget_sites_page_objects.site_2.GadgetSite2` wherein it's only going to be utilized for **site_2.example** if the following is to be replaced: `gadget_sites_page_objects.GadgetGenericPage`.

It would be only become a conflict if both rules for **site_2.example** *intend to replace the* **same** *Page Object*.

However, let's suppose that there are some *ApplyRule* which actually result in a conflict. To give an example, let's suppose that rules **#2** and **#3** *intends to replace the* **same** *Page Object*. It would look something like:

```
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'common_items.ProductGenericPage'>, to_return=None, meta={})
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'common_items.ProductGenericPage'>, to_return=None, meta={})
```

Notice that the `instead_of` param are the same and only the `use` param remained different.

There are two main ways we recommend in solving this.

1. Priority Resolution

If you notice, the `for_patterns` attribute of *ApplyRule* is an instance of url_matcher.Patterns. This instance also has a `priority` param where a higher value will be chosen in times of conflict.

Note: The url-matcher library is the one responsible breaking such `priority` conflicts *(amongst others)*. It's specifically discussed in this section: rules-conflict-resolution.

Unfortunately, updating the `priority` value directly isn't possible as the `url_matcher.Patterns` is a **frozen** *dataclass*. The same is true for *ApplyRule*. This is made by design so that they are hashable and could be deduplicated immediately without consequences of them changing in value.

The only way that the `priority` value can be changed is by creating a new *ApplyRule* with a different `priority` value (*higher if it needs more priority*). You don't necessarily need to *delete* the **old** *ApplyRule* since they will be resolved via `priority` anyways.

Creating a new *ApplyRule* with a higher priority could be as easy as:

1. Subclassing the Page Object in question.
2. Declare a new *web_poet.handle_urls()* decorator with the same URL pattern and Page Object to override but with a much higher priority.

Here's an example:

```
from web_poet import default_registry, consume_modules, handle_urls
import ecommerce_page_objects, gadget_sites_page_objects, common_items

@handle_urls("site_2.example", instead_of=common_items.ProductGenericPage, priority=1000)
class EcomSite2Copy(ecommerce_page_objects.site_1.EcomSite1):
    def to_item(self):
        return super().to_item()
```

Now, the conflicting **#2** and **#3** rules would never be selected because of the new *ApplyRule* having a much higher priority (see rule **#4**):

```
# 2. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'ecommerce_page_objects.site_2.EcomSite2'>, instead_of=<class
↪'common_items.ProductGenericPage'>, to_return=None, meta={})
# 3. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=500), use=<class 'gadget_sites_page_objects.site_2.GadgetSite2'>, instead_of=
↪<class 'common_items.ProductGenericPage'>, to_return=None, meta={})

# 4. ApplyRule(for_patterns=Patterns(include=['site_2.example'], exclude=[],␣
↪priority=1000), use=<class 'my_project.EcomSite2Copy'>, instead_of=<class 'common_items.
↪ProductGenericPage'>, to_return=None, meta={})
```

A similar idea was also discussed in the *Improving on external Page Objects* section.

2. Specifically Selecting the Rules

When the last resort of `priority`-resolution doesn't work, then you could always specifically select the list of *ApplyRule* you want to use.

We **recommend** in creating an **inclusion**-list rather than an **exclusion**-list since the latter is quite brittle. For instance, an external package you're using has updated its rules and the exlusion strategy misses out on a few rules that were recently added. This could lead to a *silent-error* of receiving a different set of rules than expected.

This **inclusion**-list approach can be done by importing the Page Objects directly and creating instances of *ApplyRule* from it. You could also import all of the available *ApplyRule* using *get_rules()* to sift through the list of available rules and manually selecting the rules you need.

Most of the time, the needed rules are the ones which uses the Page Objects we're interested in. You can use *search()* to get them (see *Using only a subset of the available ApplyRules*):

```
from web_poet import default_registry, consume_modules
import package_A, package_B, package_C

consume_modules("package_A", "package_B", "package_C")

rules = default_registry.search(use=package_A.PageObject1) + \
        default_registry.search(use=package_B.PageObject2) + \
        default_registry.search(use=package_C.PageObject3)

# ApplyRule(for_patterns=Patterns(include=['site_A.example'], exclude=[], priority=500),␣
↪use=<class 'package_A.PageObject1'>, instead_of=<class 'GenericPage'>, to_return=None,␣
↪meta={})
# ApplyRule(for_patterns=Patterns(include=['site_B.example'], exclude=[], priority=500),␣
↪use=<class 'package_B.PageObject2'>, instead_of=<class 'GenericPage'>, to_return=None,␣
↪meta={})
# ApplyRule(for_patterns=Patterns(include=['site_C.example'], exclude=[], priority=500),␣
↪use=<class 'package_C.PageObject3'>, instead_of=<class 'GenericPage'>, to_return=None,␣
↪meta={})
```

Another example:

```
from url_matcher import Patterns
from web_poet import default_registry, consume_modules
import package_A, package_B, package_C
```

(continues on next page)

(continued from previous page)

```
consume_modules("package_A", "package_B", "package_C")

rule_from_A = default_registry.search(use=package_A.PageObject1)
print(rule_from_A)
# [ApplyRule(for_patterns=Patterns(include=['site_A.example'], exclude=[], priority=500),␣
↪use=<class 'package_A.PageObject1'>, instead_of=<class 'GenericPage'>, to_return=None,␣
↪meta={})]

rule_from_B = default_registry.search(instead_of=GenericProductPage)
print(rule_from_B)
# []

rule_from_C = default_registry.search(for_patterns=Patterns(include=["site_C.example"]))
print(rule_from_C)
# [
#     ApplyRule(for_patterns=Patterns(include=['site_C.example'], exclude=[],␣
↪priority=500), use=<class 'package_C.PageObject3'>, instead_of=<class 'GenericPage'>, to_
↪return=None, meta={}),
#     ApplyRule(for_patterns=Patterns(include=['site_C.example'], exclude=[],␣
↪priority=1000), use=<class 'package_C.PageObject3_improved'>, instead_of=<class
↪'GenericPage'>, to_return=None, meta={})
# ]

rules = rule_from_A + rule_from_B + rule_from_C
```

CHAPTER

EIGHT

RETRIES

The responses of some websites can be unreliable. For example, sometimes a request can get a response that may only include a part of the data to be extracted, no data at all, or even data unrelated to your request, but sending a follow-up, identical request can get you the expected data.

Pages objects are responsible for handling these scenarios, where issues with response data can only be detected during extraction.

8.1 Retrying Page Object Input

When the bad response data comes from the inputs that your web-poet framework supplies to your page object, your page object must raise *Retry*:

```
from web_poet import WebPage
from web_poet.exceptions import Retry

class MyPage(WebPage):

    def to_item(self) -> dict:
        if not self.css(".expected"):
            raise Retry
        return {}
```

As a result, your web-poet framework will retry the source requests and create a new instance of your page object with the new inputs.

8.2 Retrying Additional Requests

When the bad response data comes from *additional requests*, you must handle retries on your own.

The page object 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, HttpRequest, WebPage

@attrs.define
class MyPage(WebPage):
    http: HttpClient

    @retry(stop=stop_after_attempt(3))
    async def get_data(self):
        request = HttpRequest("https://toscrape.com/")
        response = await self.http.execute(request)
        if not response.css(".expected"):
            raise ValueError
        return response.css(".data").get()

    async def to_item(self) -> dict:
        try:
            data = await self.get_data()
        except ValueError:
            return {}
        return {"data": data}
```

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

NINE

USING PAGE PARAMS

In some cases, Page Objects might require additional information to be passed to them. Such information can dictate the behavior of the Page Object or affect its data entirely depending on the needs of the developer.

If you can recall from the previous basic tutorials, one essential requirement of Page Objects that inherit from *WebPage* would be *HttpResponse*. This holds the HTTP response information that the Page Object is trying to represent.

In order to standardize how to pass arbitrary information inside Page Objects, we'll need to use *PageParams* similar on how we use *HttpResponse* as a requirement to instantiate Page Objects:

```
import attrs
import web_poet

@attrs.define
class SomePage(web_poet.WebPage):
    # The HttpResponse attribute is inherited from WebPage
    page_params: web_poet.PageParams

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

# It uses Python's dict interface.
page_params = web_poet.PageParams({"arbitrary_value": 1234, "cool": True})

page = SomePage(response=response, page_params=page_params)
```

However, similar with *HttpResponse*, developers using *PageParams* shouldn't care about how they are being passed into Page Objects. This will depend on the framework that would use **web-poet**.

Let's checkout some examples on how to use it inside a Page Object.

9.1 Controlling item values

```
import attrs
import web_poet


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

(continues on next page)

(continued from previous page)

```
    default_tax_rate = 0.10

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

    @staticmethod
    def calculate_price_with_tax(item):
        tax_rate = self.page_params.get("tax_rate") or 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.

9.2 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 paginations being made.

```
from typing import List

import attrs
import web_poet


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

    default_max_pages = 5

    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") or self.default_max_pages
        requests = [
            self.create_next_page_request(page_num)
            for page_num in range(2, max_pages + 1)
        ]
```

(continues on next page)

(continued from previous page)

```
        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
        ]

    @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 in case the *PageParams* instance did not provide it.

CHAPTER

TEN

SUPPORTING ADDITIONAL REQUESTS

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

10.1 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:

10.1.1 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

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

    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.

10.1.2 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

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

    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.

10.2 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.

10.3 Exception Handling

In the previous *Handling Exceptions in Page Objects* section, we can see how 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.

10.3.1 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 attrs
import web_poet

import aiohttp
import requests
import urllib


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

    async def to_item(self):
        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


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

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

10.3.2 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

ELEVEN

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

TWELVE

API REFERENCE

12.1 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()`.

property selector: Selector

Cached instance of `parsel.selector.Selector`.

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

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

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_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`.

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

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_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*, *body=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*, *encoding: Optional[str] = None*)

Bases: SelectableMixin

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.

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*.

css(*query*) → SelectorList

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

property selector: Selector

Cached instance of `parsel.selector.Selector`.

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.client.**HttpClient**(*request_downloader: Optional[Callable] = 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: Optional[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: Optional[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: Optional[Union[str, int, List[Union[str, int]]]] = None*) → *HttpResponse*

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

async execute(*request:* HttpRequest, *, *allow_status: Optional[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: Optional[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.

12.2 Pages

class web_poet.pages.**Injectable**

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**(**args*, ***kwds*)

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

Base Page Object, with a default *to_item()* implementation which supports web-poet fields.

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

Shortcut to HTML Response's content.

property item_cls: Type[ItemT]

Item class

property selector: Selector

Cached instance of `parsel.selector.Selector`.

async to_item() → ItemT

Extract an item from a web page

property url

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**(**args*, ***kwds*)

Bases: Generic[ItemT]

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

property item_cls: Type[ItemT]

Item class

12.3 Mixins

class web_poet.mixins.**ResponseShortcutsMixin**

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

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

property html

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

12.4 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.

12.5 Exceptions

12.5.1 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.**Retry**

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

See *Retries*.

12.5.2 HTTP Exceptions

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

exception web_poet.exceptions.http.**HttpError**(*msg: Optional[str] = None*, *request: Optional[*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) – The *HttpRequest* instance that was used.

exception web_poet.exceptions.http.**HttpRequestError**(*msg: Optional[str] = None*, *request: Optional[*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: Optional[str] = None*, *response: Optional[*HttpResponse*] = None*, *request: Optional[*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) – The *HttpRequest* instance that was used.
- **response** (HttpResponse) – The *HttpResponse* instance that was received, if any. Note that this wouldn't exist if the problem ocurred when executing the *HttpRequest*.

12.6 Apply Rules

See the tutorial section on *Apply 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 (see *Item Class* section for some examples). 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. This is the concept of **overrides** (see the *Overrides* section for more info`).

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 *Item Classes* section. Code example in *Combination* subsection.
- **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*)

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 *Apply 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 (see *Creating a new registry*).

However, it might be applicable in certain scenarios like storing custom rules to separate it from the `default_registry`. This *example* from the tutorial section may provide some context.

---

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.

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*)

12.7 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()*.

12.8 utils

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

CHAPTER

THIRTEEN

CONTRIBUTING

web-poet is an open-source project. Your contribution is very welcome!

13.1 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

13.2 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.

13.3 Testing

We use tox to run tests with different Python versions:

```
tox
```

The command above also runs type checks; we use mypy.

CHAPTER

FOURTEEN

CHANGELOG

14.1 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.

14.2 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.

14.3 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.

14.4 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.

14.5 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.

14.6 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

14.7 0.1.1 (2021-06-02)

- `base_url` and `urljoin` shortcuts

14.8 0.1.0 (2020-07-18)

- Documentation
- WebPage, ItemPage, ItemWebPage, Injectable and ResponseData are available as top-level imports (e.g. `web_poet.ItemPage`)

14.9 0.0.1 (2020-04-27)

Initial release.

CHAPTER

FIFTEEN

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.

PYTHON MODULE INDEX

W

INDEX

W

X