Items

The main goal in scraping is to extract structured data from unstructured sources, typically, web pages. Scrapy spiders can return the extracted data as Python dicts. While convenient and familiar, Python dicts lack structure: it is easy to make a typo in a field name or return inconsistent data, especially in a larger project with many spiders.

To define common output data format Scrapy provides the `Item` class. `Item` objects are simple containers used to collect the scraped data. They provide a dictionary-like API with a convenient syntax for declaring their available fields.

Various Scrapy components use extra information provided by Items: exporters look at declared fields to figure out columns to export, serialization can be customized using Item fields metadata, `trackref` tracks Item instances to help find memory leaks (see Debugging memory leaks with trackref), etc.

Declaring Items

Items are declared using a simple class definition syntax and `Field` objects. Here is an example:

```
import scrapy

class Product(scrapy.Item):
    name = scrapy.Field()
    price = scrapy.Field()
    stock = scrapy.Field()
    last_updated = scrapy.Field(serializer=str)
```

Note

Those familiar with Django will notice that Scrapy Items are declared similar to Django Models, except that Scrapy Items are much simpler as there is no concept of different field types.

Item Fields

`Field` objects are used to specify metadata for each field. For example, the serializer function for the `last_updated` field illustrated in the example above.

You can specify any kind of metadata for each field. There is no restriction on the values accepted by `Field` objects. For this same reason, there is no reference list of all available metadata keys. Each key defined in `Field` objects could be used by a different component, and only those components know about it. You can also define and use any other `Field` key in your project too, for your own needs. The main goal of `Field` objects is to provide a way to define all field metadata in one place. Typically, those components whose behaviour depends on each field use certain field keys to configure that behaviour. You must refer to their documentation to see which metadata keys are used by each component.

It's important to note that the `Field` objects used to declare the item do not stay assigned as class attributes. Instead, they can be accessed through the `Item.fields` attribute.

Working with Items

Here are some examples of common tasks performed with items, using the `Product` item declared above. You will notice the API is very similar to the dict API.

Creating items

```
>>> product = Product(name='Desktop PC', price=1000)
>>> print product
Product(name='Desktop PC', price=1000)
```

Getting field values

```
>>> product['name']
Desktop PC
>>> product.get('name')
Desktop PC

>>> product['price']
1000

>>> product['last_updated']
Traceback (most recent call last):
    ...
KeyError: 'last_updated'

>>> product.get('last_updated', 'not set')
not set

>>> product['lala'] # getting unknown field
```

```
Traceback (most recent call last):
    ...
KeyError: 'lala'

>>> product.get('lala', 'unknown field')
'unknown field'

>>> 'name' in product  # is name field populated?
True

>>> 'last_updated' in product  # is last_updated populated?
False

>>> 'last_updated' in product.fields  # is last_updated a declared field?
True

>>> 'lala' in product.fields  # is lala a declared field?
False
```

Setting field values

```
>>> product['last_updated'] = 'today'
>>> product['last_updated']
today

>>> product['lala'] = 'test' # setting unknown field
Traceback (most recent call last):
    ...
KeyError: 'Product does not support field: lala'
```

Accessing all populated values

To access all populated values, just use the typical dict API:

```
>>> product.keys()
['price', 'name']

>>> product.items()
[('price', 1000), ('name', 'Desktop PC')]
```

Other common tasks

Copying items:

```
>>> product2 = Product(product)
>>> print product2
Product(name='Desktop PC', price=1000)

>>> product3 = product2.copy()
>>> print product3
Product(name='Desktop PC', price=1000)
```

Creating dicts from items:

```
>>> dict(product) # create a dict from all populated values
{'price': 1000, 'name': 'Desktop PC'}
```

Creating items from dicts:

```
>>> Product({'name': 'Laptop PC', 'price': 1500})
Product(price=1500, name='Laptop PC')

>>> Product({'name': 'Laptop PC', 'lala': 1500}) # warning: unknown field in dict
Traceback (most recent call last):
    ...
KeyError: 'Product does not support field: lala'
```

Extending Items

You can extend Items (to add more fields or to change some metadata for some fields) by declaring a subclass of your original Item.

For example:

```
class DiscountedProduct(Product):
    discount_percent = scrapy.Field(serializer=str)
    discount_expiration_date = scrapy.Field()
```

You can also extend field metadata by using the previous field metadata and appending more values, or changing existing values, like this:

```
class SpecificProduct(Product):
    name = scrapy.Field(Product.fields['name'], serializer=my_serializer)
```

That adds (or replaces) the `serializer` metadata key for the `name` field, keeping all the previously existing metadata values.

Item objects

class scrapy.item.Item([arg])

Return a new Item optionally initialized from the given argument.

Items replicate the standard dict API, including its constructor. The only additional attribute provided by Items is:

fields

A dictionary containing all declared fields for this Item, not only those populated. The keys are the field names and the values are the `Field` objects used in the Item declaration.

Field objects

class scrapy.item.Field([arg])

The `Field` class is just an alias to the built-in dict class and doesn't provide any extra functionality or attributes. In other words, `Field` objects are plain-old Python dicts. A separate class is used to support the item declaration syntax based on class attributes.

Previous Next