dataclass-factory

Jul 20, 2022

Contents:

CHAPTER 1

Overview

1.1 Requirements

- python >= 3.6

You can use `dataclass_factory` with python 3.6 and `dataclass` library installed from pip.

From python 3.7 it has no external dependencies outside of the Python standard library.

1.2 Advantages

- No schemas or configuration needed for simple cases. Just create `Factory` and call `load`/`dump` methods
- Speed. It is up to 10 times faster than `marshmallow` and `dataclasses.asdict`
- Automatic name style conversion (e.g. `snake_case` to `CamelCase`)
- Automatic skipping of "internal use" fields (with leading underscore)
- Enums, typed dicts, tuples and lists are supported from the box
- Unions and Optionals are supported without need to define them in schema
- Generic dataclasses can be automatically parsed as well
- Cyclic-referenced structures (such as linked-lists or trees) also can be converted
- Per-field validators

1.3 Example

```python
from dataclasses import dataclass
import dataclass_factory


@dataclass
class Book:
    title: str
    price: int
    author: str = "Unknown author"


data = {
    "title": "Fahrenheit 451",
    "price": 100,
}

factory = dataclass_factory.Factory()
book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
serialized = factory.dump(book)
```

1.4 Supported types

- numeric types (`int`, `float`, `Decimal`, `complex`, `Fraction`)
- `bool`
- `str`, `bytearray`, `bytes`
- `List` and common protocols like `Iterable` are parsed as `list`
- `Tuple`, including something like `Tuple[int, ...]` or `Tuple[int, str, int]`
- `Dict`
- `Enum` is converted using its value
- `Optional`
- `Any`, using this type no conversion is done during parsing. But serialization is based on real data type
- `Union`
- `Literal` types, including variant from `typing_extensions`
- `TypedDict` types with checking of `total`, including variant from `typing_extensions`
- `dataclass` and `NamedTuple`
- `Generic` dataclasses
- Other standard types like `datetime`, `Path`, `UUID` and `IPV4Address`
- Custom classes can be parsed automatically using info from their `__init__` method. Serialization is done by calling *vars()* function and then processing real data types
- Or you can provide custom parser/serializer

CHAPTER 2

Quickstart

Dataclass factory analyzes your type hints and generates corresponding parsers based on retrieved information. For dataclasses it checks what fields are declared and then calls normal constructor. For others types behavior can differ.

Also you can configure it using miscellaneous schemas (see *Extended usage*).

2.1 Installation

Just use pip to install the library:

```
pip install dataclass_factory
```

2.2 Simple case

All you have to do to start parsing you dataclasses is create a Factory instance. Then call `load` or `dump` methods with corresponding type and everything is done automatically.

```
from dataclasses import dataclass
import dataclass_factory


@dataclass
class Book:
    title: str
    price: int
    author: str = "Unknown author"


data = {
    "title": "Fahrenheit 451",
    "price": 100,
```

(continues on next page)

(continued from previous page)

```
}

factory = dataclass_factory.Factory()
book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
serialized = factory.dump(book)
```

All typing information is retrieved from you annotations, so it is not required from you to provide any schema or even change your dataclass decorators or class bases.

In provided example `book.author == "Unknown author"` because normal dataclass constructor is called.

It is better to create factory only once, because all parsers are cached inside it after first usage. Otherwise, the structure of your classes will be analysed again and again for every new instance of Factory.

2.3 Nested objects

Nested objects are supported out of the box. It is surprising, but you do not have to do anything except defining your dataclasses. For example, your expect that author of Book is instance of Person, but in serialized form it is dictionary.

Declare your dataclasses as usual and then just parse your data.

```
from dataclasses import dataclass

import dataclass_factory


@dataclass
class Person:
    name: str


@dataclass
class Book:
    title: str
    price: int
    author: Person


data = {
    "title": "Fahrenheit 451",
    "price": 100,
    "author": {
        "name": "Ray Bradbury"
    }
}

factory = dataclass_factory.Factory()

# Book(title="Fahrenheit 451", price=100, author=Person("Ray Bradbury"))
book: Book = factory.load(data, Book)
serialized = factory.dump(book)
```

2.4 Lists and other collections

Want to parse collection of dataclasses? No changes required, just specify correct target type (e.g `List[SomeClass]` or `Dict[str, SomeClass]`).

```
from typing import List

from dataclasses import dataclass

import dataclass_factory


@dataclass
class Book:
    title: str
    price: int


data = [
    {
        "title": "Fahrenheit 451",
        "price": 100,
    },
    {
        "title": "1984",
        "price": 100,
    }
]

factory = dataclass_factory.Factory()
books = factory.load(data, List[Book])
serialized = factory.dump(books)
```

Fields also can contain any supported collections.

2.5 Error handling

Currently parser doesn't throw any specific exception in case of parser failure. Errors are the same as thrown by corresponding constructors. In normal cases all suitable exceptions are described in `dataclass_factory.PARSER_EXCEPTIONS`

```
from dataclasses import dataclass

import dataclass_factory


@dataclass
class Book:
    title: str
    price: int
    author: str = "Unknown author"


data = {
    "title": "Fahrenheit 451"
```

(continues on next page)

(continued from previous page)

```
}

factory = dataclass_factory.Factory()

try:
    book: Book = factory.load(data, Book)
except dataclass_factory.PARSER_EXCEPTIONS as e:
    # Cannot parse:  <class 'TypeError'> __init__() missing 1 required positional␣
↪argument: 'price'
    print("Cannot parse: ", type(e), e)
```

2.6 Validation

Validation of data can be done in two cases:

- per-field validations
- whole structure validation

In first case you can use `@validate` decorator with field name to check the data.

Here are details:

- validator can be called before parsing field data (set `pre=True`) or after it.
- field validators are applied after all name transformations (name styles, structure flattening). So use field name as it is called in your class
- validator can be applied to multiple fields. Just provide multiple names
- validator can be applied to any field separately. Just do not set any field name
- validator must return data if checks are succeeded. Data can be same as passed to it or anything else. Validator CAN change data
- field validators cannot be set in default schema
- multiple decorators can be used on single method
- multiple validators can be applied to single field
- use different method names to prevent overwriting

```
from dataclasses import dataclass

from dataclass_factory import validate, Factory, Schema, NameStyle


class MySchema(Schema):
    SOMETHING = "Some string"

    @validate("int_field")  # use original field name in class
    def validate_field(self, data):
        if data > 100:
            raise ValueError
        return data * 100  # validator can change value

    # this validator will be called before parsing field
    @validate("complex_field", pre=True)
```

(continues on next page)

(continued from previous page)

```
    def validate_field_pre(self, data):
        return data["value"]

    @validate("info")
    def validate_stub(self, data):
        return self.SOMETHING  # validator can access schema fields


@dataclass
class My:
    int_field: int
    complex_field: int
    info: str


factory = Factory(schemas={
    My: MySchema(name_style=NameStyle.upper_snake)  # name style does not affect how␣
↪validators are bound to fields
})

result = factory.load({"INT_FIELD": 1, "COMPLEX_FIELD": {"value": 42}, "INFO":
↪"ignored"}, My)
assert result == My(100, 42, "Some string")
```

If you want to check whole structure, your can any check in `pre_parse` or `post_parse` step. Idea is the same:

- `pre_parse` is called before structure parsing is done (but even before data is flattened and names are processed).
- `post_parse` is called after successful parsing
- only one `pre_parse` and one `post_parse` methods can be in class.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    author: str = "Unknown author"


data = {
    "title": "Fahrenheit 451",
    "price": 100,
}
invalid_data = {
    "title": "1984",
    "price": -100,
}


def validate_book(book: Book) -> Book:
    if not book.title:
```

(continues on next page)

(continued from previous page)

```
        raise ValueError("Empty title")
    if book.price <= 0:
        raise ValueError("InvalidPrice")
    return book


factory = dataclass_factory.Factory(schemas={Book: Schema(post_parse=validate_book)})
book = factory.load(data, Book)  # No error
other_book = factory.load(invalid_data, Book)  # ValueError: InvalidPrice
```

CHAPTER 3

Extended usage

You can configure the factory during its creation. You can't change the settings later because they affect parsers, which are created only once for each instance of a factory.

Most of the configuration is done via Schemas. You can set default schema or one per type:

```
factory = Factory(default_schema=Schema(...), schemas={ClassA: Schema(...)})
```

3.1 More verbose errors

Currently, errors are not very verbose. But you can make them a bit better using `debug_path` of a factory. It is disabled by default because affects performance.

In this mode `InvalidFieldError` is thrown when some dataclass field cannot be parsed. It contains `field_path` which is a path to the field in provided data (key and indexes).

3.2 Working with field names

3.2.1 Name mapping

In some cases, you have json with keys that leave much to be desired. For example, they might contain spaces or just have unclear meanings. The simplest way to fix it is to set a custom name mapping. You can call fields as you want and the factory will translate them using your mapping.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


```

(continues on next page)

(continued from previous page)

```
@dataclass
class Book:
    title: str
    price: int


data = {
    "title": "Fahrenheit 451",
    "book price": 100,
}

book_schema = Schema(name_mapping={
    "price": "book price"
})
factory = dataclass_factory.Factory(schemas={Book: book_schema})
book: Book = factory.load(data, Book)
serialized = factory.dump(book)
```

Fields absent in mapping are not translated and used with their original names (as in dataclass specification).

3.2.2 Stripping underscore

It is often unnecessary to fill name mapping. One of the most common cases is dictionary keys which are python keywords. For example, you cannot use the string `from` as a field name, but it is very likely to see in APIs. Usually, it is solved by adding a trailing underscore (e.g. `from_`).

Dataclass factory will trim trailing underscores so you won't meet this case.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Period:
    from_: int
    to_: int


data = {
    "from": 1,
    "to": 100,
}

factory = dataclass_factory.Factory()
period = factory.load(data, Period)
```

Sometimes this behavior is unwanted, so you can disable this feature by setting `trim_trailing_underscore=False` in Schema (in default schema of the concrete one). Also, you can re-enable it for certain types.

```
from dataclasses import dataclass

import dataclass_factory
```

(continues on next page)

(continued from previous page)

```
from dataclass_factory import Schema


@dataclass
class Period:
    from_: int
    to_: int


data = {
    "from_": 1,
    "to_": 100,
}

factory = dataclass_factory.Factory(default_schema=Schema(trim_trailing_
↪underscore=False))
period = factory.load(data, Period)
factory.dump(period)
```

3.2.3 Name styles

Sometimes json keys are quite normal but ugly. For example, they are named using CamelCase, but PEP8 recommends you to use snake_case. Of cause, you can prepare name mapping, but it is too much to write for such a stupid thing.

The library can translate such names automatically. You need to declare fields as recommended by PEP8 (e.g. *field_name*) and set corresponding `name_style`. As usual, if no style is set for a certain type, it will be taken from the default schema.

By the way, you cannot convert names that do not follow snake_case style. In this case, the only valid style is `ignore`

```
from dataclasses import dataclass

from dataclass_factory import Factory, Schema, NameStyle

factory = Factory(default_schema=Schema(
    name_style=NameStyle.camel
))


@dataclass
class Person:
    first_name: str
    last_name: str


person = Person("ivan", "petrov")

serial_person = {
    "FirstName": "ivan",
    "LastName": "petrov"
}

assert factory.dump(person) == serial_person
```

Following name styles are supported:

- `snake` (snake_case)

- `kebab` (kebab-case)
- `camel_lower` (camelCaseLower)
- `camel` (CamelCase)
- `lower` (lowercase)
- `upper` (UPPERCASE)
- `upper_snake` (UPPER_SNAKE_CASE)
- `camel_snake` (Camel_Snake)
- `dot` (dot.case)
- `camel_dot` (Camel.Dot)
- `upper_dot` (UPPER.DOT)
- `ignore` (not real style, but just does no conversion)

3.3 Selecting and skipping fields

You have several ways to skip processing of some fields.

Note: Skipped fields MUST NOT be required in class constructor, otherwise parsing will fail

3.3.1 Only and exclude

If you know exactly what fields must be parsed/serialized and want to ignore all others just set them as `only` parameter of schema. Also, you can provide a list with excluded names via `exclude`.

It affects both parsing and serializing.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    extra: str = ""


data = {
    "title": "Fahrenheit 451",
    "price": 100,
    "extra": "some extra string"
}

# using `only`:
factory = dataclass_factory.Factory(schemas={Book: Schema(only=["title", "price"])})
book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
```

(continues on next page)

(continued from previous page)

```
serialized = factory.dump(book)  # no `extra` key will be in serialized

# using `exclude`
factory = dataclass_factory.Factory(schemas={Book: Schema(exclude=["extra"])})
book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
serialized = factory.dump(book)  # no `extra` key will be in serialized
```

3.3.2 Only mapped

Already have `name_mapping` and do not want to repeat all names in `only` parameter? Just set `only_mapped=True`. It will ignore all fields which are not described in name mapping.

3.3.3 Skip Internal

More simplified case is to skip so-called *internal use* fields, those fields which name starts with underscore. You can skip them from parsing and serialization using `skip_internal` option of schema.

It is disabled by default. It affects both parsing and serializing.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    _total: int = 0


data = {
    "title": "Fahrenheit 451",
    "price": 100,
    "_total": 1000,
}

factory = dataclass_factory.Factory(default_schema=Schema(skip_internal=True))
book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
serialized = factory.dump(book)  # no `_total` key will be produced
```

3.3.4 Omit default

If you have defaults for some fields, it is unnecessary to store them in serialized representation. For example, this may be `None`, empty list or something else. You can omit them when serializing using `omit_default` option. Those values that are **equal** to default, will be stripped from the resulting dict.

It is disabled by default. It affects only serialising.

```
from typing import Optional, List

from dataclasses import dataclass, field

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: Optional[int] = None
    authors: List[str] = field(default_factory=list)


data = {
    "title": "Fahrenheit 451",
}

factory = dataclass_factory.Factory(default_schema=Schema(omit_default=True))
book = Book(title="Fahrenheit 451", price=None, authors=[])
serialized = factory.dump(book)  # no `price` and `authors` key will be produced
assert data == serialized
```

3.4 Structure flattening

Another case of ugly API is a too complex hierarchy of data. You can fix it using already known `name_mapping`. Earlier, you used it to rename fields, but also you can use it to map a name to a nested value by specifying a path to it.

Integers in the path are treated as list indices, strings - as dict keys. It affects parsing and serializing.

For example, you have an author of a book with only field - name (see *Nested objects*). You can expand this dict and store the author name directly in your Book class.

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    author: str


data = {
    "title": "Fahrenheit 451",
    "price": 100,
    "author": {
        "name": "Ray Bradbury"
    }
}
```

(continues on next page)

(continued from previous page)

```
book_schema = Schema(
    name_mapping={
        "author": ("author", "name")
    }
)
factory = dataclass_factory.Factory(schemas={Book: book_schema})

# Book(title="Fahrenheit 451", price=100, author="Ray Bradbury")
book: Book = factory.load(data, Book)
serialized = factory.dump(book)
assert serialized == data
```

We can modify example above to store author as a list with name

```
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    author: str


data = {
    "title": "Fahrenheit 451",
    "price": 100,
    "author": ["Ray Bradbury"]
}

book_schema = Schema(
    name_mapping={
        "author": ("author", 0)
    }
)
factory = dataclass_factory.Factory(schemas={Book: book_schema})

# Book(title="Fahrenheit 451", price=100, author="Ray Bradbury")
book: Book = factory.load(data, Book)
serialized = factory.dump(book)
assert serialized == data
```

3.4.1 Automatic naming during flattening

If names somewhere in "complex" structure are the same, as in your class you can simplify your schema using ellipsis (`...`). There are two simple rules:

- `...` as as a key in `name_mapping` means *Any* field. Path will be applied to every field that is not declared explicitly in mapping
- `...` inside path in `name_mapping` means that original name of field will be reused. If name style or other rules are provided the will be applied to the name.

Examples:

```python
from dataclasses import dataclass

import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    author: str


data = {
    "book": {
        "title": "Fahrenheit 451",
        "price": 100,
    },
    "author": {
        "name": "Ray Bradbury"
    }
}

book_schema = Schema(
    name_mapping={
        "author": (..., "name"),
        ...: ("book", ...)
    }
)
factory = dataclass_factory.Factory(schemas={Book: book_schema})

# Book(title="Fahrenheit 451", price=100, author="Ray Bradbury")
book: Book = factory.load(data, Book)
serialized = factory.dump(book)
assert serialized == data
```

3.5 Parsing unknown fields

By default, all extra fields that are absent in the target structure are ignored. But this behavior is not necessary. For now, you can select from several variants setting `unknown` attribute of Schema

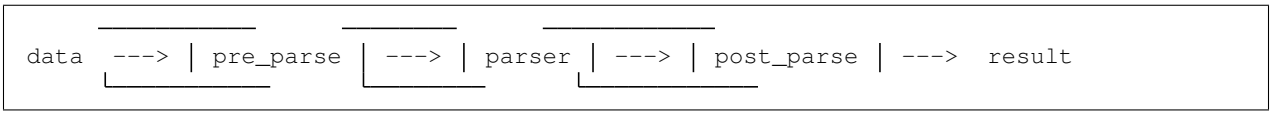
- `Unknown.SKIP` - default behavior. All unknown fields are ignored (skipped)
- `Unknown.FORBID` - `UnknownFieldsError` is raised in case of any unknown field is found
- `Unknown.STORE` - all unknown fields passed unparsed to the constructor of a class. Your `__init__` must be ready for this
- Field name (`str`). The specified field is filled with all unknowns and the parser of the corresponding type is called. For simple cases, you can annotate that field with `Dict` type. In the case of serialization, this field is also serialized and the result is merged up with the current result.
- Several field names (sequence of `str`). The behavior is very similar to the case with one field name. All unknowns are collected to a single dict and it is passed to parsers of each provided field (be careful modifying data at `pre_parse` step). Also, their dump results are merged when serializing

```
from typing import Optional, Dict

from dataclasses import dataclass

from dataclass_factory import Factory, Schema


@dataclass
class Sub:
    b: str


@dataclass
class Data:
    a: str
    unknown: Optional[Dict] = None
    sub: Optional[Sub] = None


serialized = {
    "a": "A1",
    "b": "B2",
    "c": "C3",
}

factory = Factory(default_schema=Schema(unknown=["unknown", "sub"]))
data = factory.load(serialized, Data)
assert data == Data(a="A1", unknown={"b": "B2", "c": "C3"}, sub=Sub("B2"))
```

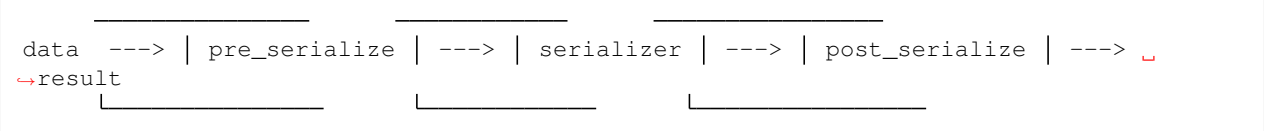
3.6 Additional steps

Most of the work is done automatically, but you may want to do some additional processing.

Real parsing process has following flow:



The same is for serializing:



So the return value of `pre_parse` is passed to `parser`, and return value of `post_parse` is used as the total result of the parsing process. You can add your logic at any step, but mind the main difference:

- `pre_parse` and `post_serialize` work with serialized representation of data (e.g. dict for dataclasses)
- `post_parse` and `pre_serialize` work with instances of your classes.

So if you want to do some validation - it is better to do at `post_parse` step. And if you want to do *polymorphic parsing* - check if a type is suitable before parsing is started at `pre_parse`.

Another case is to change the representation of some fields: serialize json to string, split values and so on.

```
import json
from typing import List

from dataclasses import dataclass

from dataclass_factory import Schema, Factory


@dataclass
class Data:
    items: List[str]
    name: str


def post_serialize(data):
    data["items"] = json.dumps(data["items"])
    return data


def pre_parse(data):
    data["items"] = json.loads(data["items"])
    return data


def post_parse(data: Data) -> Data:
    if not data.name:
        raise ValueError("Name must not be empty")
    return data


data_schema = Schema[Data](
    post_serialize=post_serialize,
    pre_parse=pre_parse,
    post_parse=post_parse,
)
factory = Factory(schemas={Data: data_schema})

data = Data(['a', 'b'], 'My Name')
serialized = {'items': '["a", "b"]', 'name': 'My Name'}
assert factory.dump(data) == serialized
assert factory.load(serialized, Data) == data

try:
    factory.load({'items': '[]', 'name': ''}, Data)
except ValueError as e:
    print("Error detected:", e)  # Error detected: Name must not be empty
```

3.7 Schema inheritance

In some cases, it might be useful to subclass Schema instead of just creating instances normally.

```
from typing import Any

from dataclasses import dataclass

```

(continues on next page)

(continued from previous page)

```
import dataclass_factory
from dataclass_factory import Schema


@dataclass
class Book:
    title: str
    price: int
    _author: str = "Unknown author"


data = {
    "title": "Fahrenheit 451",
    "price": 100,
}


class DataSchema(Schema[Any]):
    skip_internal = True

    def post_parse(self, data):
        print("parsing done")
        return data


factory = dataclass_factory.Factory(schemas={Book: DataSchema(trim_trailing_
↪underscore=False)})

book: Book = factory.load(data, Book)  # Same as Book(title="Fahrenheit 451",␣
↪price=100)
serialized = factory.dump(book)
```

Note: In versions <2.9: Factory created a copy of a schema for each type filling in missed args. If you need to get access to some data in schema, get a working instance of the schema with `Factory.schema` method

Note: Single schema instance can be used multiple time simultaneously because of multithreading or recursive structures. Be careful modifying data in the schema

3.8 Json-schema

You can generate json schema for your classes.

Note that factory does it lazily and caches the result. So, if you need definitions for all of your classes, create schema for each top-level class using the `json_schema` method and then collect all definitions using `json_schema_definitions`

```
import json
from enum import Enum
from typing import Dict, Union
```

(continues on next page)

(continued from previous page)

```
from dataclasses import dataclass, field

from dataclass_factory import Factory, Schema


class A(Enum):
    X = "x"
    Y = 1


@dataclass
class Data:
    a: A
    dict_: Dict[str, Union[int, float]]
    dictw_: Dict[str, Union[int, float]] = field(default_factory=dict)
    optional_num: int = 0


factory = Factory(schemas={A: Schema(description="My super `A` class")})
print(json.dumps(factory.json_schema(Data), indent=2))
print(json.dumps(factory.json_schema_definitions(), indent=2))
```

Result of `json_schema` call is

```
{
  "title": "Data",
  "type": "object",
  "properties": {
    "a": {
      "$ref": "#/definitions/A"
    },
    "dict": {
      "$ref": "#/definitions/typing.Dict[str, typing.Union[int, float]]"
    },
    "dictw": {
      "$ref": "#/definitions/typing.Dict[str, typing.Union[int, float]]",
      "default": {}
    },
    "optional_num": {
      "type": "integer",
      "default": 0
    }
  },
  "additionalProperties": true,
  "required": [
    "a",
    "dict"
  ]
}
```

Result of `json_schema_definitions` call is

```
{
  "definitions": {
    "A": {
      "title": "A",
      "description": "My super `A` class",
```

(continues on next page)

(continued from previous page)


```
      "enum": [
        "x",
        1
      ]
    },
    "typing.Union[int, float]": {
      "title": "typing.Union[int, float]",
      "anyOf": [
        {
          "type": "integer"
        },
        {
          "type": "number"
        }
      ]
    },
    "typing.Dict[str, typing.Union[int, float]]": {
      "title": "typing.Dict[str, typing.Union[int, float]]",
      "type": "object",
      "additionalProperties": {
        "$ref": "#/definitions/typing.Union[int, float]"
      }
    },
    "Data": {
      "title": "Data",
      "type": "object",
      "properties": {
        "a": {
          "$ref": "#/definitions/A"
        },
        "dict": {
          "$ref": "#/definitions/typing.Dict[str, typing.Union[int, float]]"
        },
        "dictw": {
          "$ref": "#/definitions/typing.Dict[str, typing.Union[int, float]]",
          "default": {}
        },
        "optional_num": {
          "type": "integer",
          "default": 0
        }
      },
      "additionalProperties": true,
      "required": [
        "a",
        "dict"
      ]
    }
  }
}
```


Note: Not all features of dataclass factory are supported currently. You cannot generate json-schema if you use structure-flattening, additional parsing of unknown fields or init-based parsing. Also, if you have custom parsers or pre-parse step, schema might be incorrect.

CHAPTER 4

Specific types behavior

4.1 Structures

Out of the box dataclass factory supports three types of structures:

- dataclasses
- TypedDict (`total` is also checked)
- other classes with annotated `__init__`

Feature	dataclass	TypedDict	Class with __init__
Parsing	x	x	x
Name conversion	x	x	x
Omit default	x		
Skip internal	x	x	
Serializing	x	x	x

4.2 Custom parsers and serializers

Not all types are supported out of the box. For example, it is unclear how to parse datetime: it can be represented as unixtime or iso format or something else. If you meet unsupported type you can provide your own parser and serializer:

```
from datetime import datetime, timezone

from dataclasses import dataclass

from dataclass_factory import Schema, Factory


```

(continues on next page)

(continued from previous page)

```
@dataclass
class Author:
    name: str
    born_at: datetime


def parse_timestamp(data):
    print("parsing timestamp")
    return datetime.fromtimestamp(data, tz=timezone.utc)


unixtime_schema = Schema(
    parser=parse_timestamp,
    serializer=datetime.timestamp
)

factory = Factory(
    schemas={
        datetime: unixtime_schema,
    }
)
expected_author = Author("Petr", datetime(1970, 1, 2, 3, 4, 56, tzinfo=timezone.utc))
data = {'born_at': 97496, 'name': 'Petr'}
author = factory.load(data, Author)
print(author, expected_author)
assert author == expected_author

serialized = factory.dump(author)
assert data == serialized
```

4.3 Common schemas

We have subpackage called `dataclass_factory.schema_helpers` with some common schemas:

- `unixtime_schema` - converts `datetime` to unixtime and vice versa
- `isotime_schema` - converts `datetime` to string containing ISO 8601. Supported only on Python 3.7+
- `uuid_schema` - converts `UUID` objects to string

4.4 Self referenced types

Just place correct annotations and use factory.

```
from typing import Any, Optional

from dataclasses import dataclass

import dataclass_factory


@dataclass
class LinkedItem:
```

(continues on next page)

(continued from previous page)

```
    value: Any
    next: Optional["LinkedItem"] = None


data = {
    "value": 1,
    "next": {
        "value": 2
    }
}

factory = dataclass_factory.Factory()
items = factory.load(data, LinkedItem)
# items = LinkedItem(1, LinkedItem(2))
```

4.5 Generic classes

Generic classes supported out of the box. The difference is that if no schema found for concrete class, it will be taken for generic.

```
from dataclasses import dataclass
from typing import TypeVar, Generic

from dataclass_factory import Factory, Schema

T = TypeVar("T")


@dataclass
class FakeFoo(Generic[T]):
    value: T


factory = Factory(schemas={
    FakeFoo[str]: Schema(name_mapping={"value": "s"}),
    FakeFoo: Schema(name_mapping={"value": "i"}),
})
data = {"i": 42, "s": "Hello"}
assert factory.load(data, FakeFoo[str]) == FakeFoo("Hello")  # found schema for␣
↪concrete type
assert factory.load(data, FakeFoo[int]) == FakeFoo(42)  # schema taken from generic␣
↪version
assert factory.dump(FakeFoo("hello"), FakeFoo[str]) == {"s": "hello"}  # concrete␣
↪type is set explicitly
assert factory.dump(FakeFoo("hello")) == {"i": "hello"}  # generic type is detected␣
↪automatically
```

Note: Always pass concrete type as as second argument of `dump` method. Otherwise it will be treated as generic due to type erasure.

4.6 Polymorphic parsing

Very common case is to select class based on information in data.

If required fields differ between classes, no configuration required. But sometimes you want to make a selection more explicitly. For example, if data field "type" equals to "item" data should be parsed as Item, if it is "group" then Group class should be used.

For such case you can use `type_checker` from `schema_helpers` module. It creates a function, which should be used on `pre_parse` step. By default it checks `type` field of data, but you can change it

```
from typing import Union

from dataclasses import dataclass

from dataclass_factory import Factory, Schema
from dataclass_factory.schema_helpers import type_checker


@dataclass
class Item:
    name: str
    type: str = "item"


@dataclass
class Group:
    name: str
    type: str = "group"


Something = Union[Item, Group]  # Available types

factory = Factory(schemas={
    Item: Schema(pre_parse=type_checker("item", field="type")),
    Group: Schema(pre_parse=type_checker("group")),  # `type` is default name for␣
↪checked field
})

assert factory.load({"name": "some name", "type": "group"}, Something) == Group("some␣
↪name")
```

If you need you own pre_parse function, you can set it as parameter for `type_checker` factory.

For more complex cases you can write your own function. Just raise `ValueError` if you detected that current class is not acceptable for provided data, and parser will go to the next one in `Union`

CHAPTER 5

Indices and tables

- genindex
- modindex
- search