# typeit documentation

*Release 3.9.1.9*

Maxim Avanov

Mar 08, 2023

# Contents

**typeit** infers Python types from a sample JSON/YAML data, and provides you with the tools for serialising and parsing it. It works superb on Python 3.8 and above.

# CHAPTER 1

# Quickstart Guide

## 1.1 Installation

```
$ pip install typeit
```

## 1.2 Using CLI tool

Once installed, `typeit` provides you with a CLI tool that allows you to generate a prototype Python structure of a JSON/YAML data that your app operates with.

For example, try the following snippet in your shell:

```
$ echo '{"first-name": "Hello", "initial": null, "last_name": "World"}' | typeit gen
```

You should see output similar to this:

```
from typing import Any, NamedTuple, Optional, Sequence
from typeit import TypeConstructor


class Main(NamedTuple):
    first_name: str
    initial: Optional[Any]
    last_name: str


overrides = {
    Main.first_name: 'first-name',
}


mk_main, serialize_main = TypeConstructor & overrides ^ Main
```

You can use this snippet as a starting point to improve further. For instance, you can clarify the `Optional` type of the `Main.initial` attribute, and rename the whole structure to better indicate the nature of the data:

```
# ... imports ...

class Person(NamedTuple):
    first_name: str
    initial: Optional[str]
    last_name: str


overrides = {
    Person.first_name: 'first-name',
}


mk_person, serialize_person = TypeConstructor & overrides ^ Person
```

`typeit` will handle creation of the constructor `mk_person :: Dict -> Person` and the serializer `serialize_person :: Person -> Dict` for you.

`TypeConstructor & overrides` produces a new type constructor that takes overrides into consideration, and `TypeConstructor ^ Person` reads as "type constructor applied on the Person structure" and essentially is the same as `TypeConstructor(Person)`, but doesn't require parentheses around overrides (and extensions):

```
(TypeConstructor & overrides & extension & ...)(Person)
```

If you don't like this combinator syntax, you can use a more verbose version that does exactly the same thing:

```
TypeConstructor.override(overrides).override(extension).apply_on(Person)
```

## 1.3 Overrides

As you might have noticed in the example above, `typeit` generated a snippet with a dictionary called `overrides`, which is passed to the `TypeConstructor` alongside our `Person` type:

```
overrides = {
    Person.first_name: 'first-name',
}


mk_person, serialize_person = TypeConstructor & overrides ^ Person
```

This is the way we can indicate that our Python structure has different field names than the original JSON payload. `typeit` code generator created this dictionary for us because the `first-name` attribute of the JSON payload is not a valid Python variable name (dashes are not allowed in Python variables).

Instead of relying on automatic dasherizing of this attribute (for instance, with a help of inflection package), which rarely works consistently across all possible corner cases, `typeit` explicitly provides you with a reference point in the code, that you can track and refactor with Intelligent Code Completion tools, should that necessity arise (but this doesn't meant that you cannot apply a global rule to override all attribute names, please refer to the *Constructor Flags* section of this manual for more details).

You can use the same `overrides` object to specify rules for attributes of any nested types, for instance:

```
class Address(NamedTuple):
    street: str
    city: str
    postal_code: str


class Person(NamedTuple):
    first_name: str
    initial: Optional[str]
    last_name: str
    address: Optional[Address]


overrides = {
    Person.first_name: 'first-name',
    Address.postal_code: 'postal-code',
}


mk_person, serialize_person = TypeConstructor & overrides ^ Person
```

**Note:** Because **dataclasses** do not provide class-level property attributes (`Person.first_name` in the example above), the syntax for their overrides needs to be slightly different:

```
@dataclass
class Person:
    first_name: str
    initial: Optional[str]
    last_name: str
    address: Optional[Address]


overrides = {
    (Person, 'first_name'): 'first-name',
    (Address, 'postal_code'): 'postal-code',
}
```

## 1.4 Handling errors

Let's take the snippet above and use it with incorrect input data. Here is how we would handle the errors:

```
invalid_data = {'initial': True}

try:
    person = mk_person(invalid_data)
except typeit.Error as err:
    for e in err:
        print(f'Invalid data for `{e.path}`; {e.reason}: {repr(e.sample)} was passed')
```

If you run it, you will see an output similar to this:

```
Invalid data for `first-name`; Required: None was passed
Invalid data for `initial`; None of the expected variants matches provided data: True␣
↪was passed
Invalid data for `last_name`; Required: None was passed
```

Instances of `typeit.Error` adhere iterator interface that you can use to iterate over all parsing errors that caused the exception.

## 1.5 Supported types

- `bool`
- `int`
- `float`
- `bytes`
- `str`
- `dict`
- `set` and `frozenset`
- `typing.Any` passes any value as is
- `typing.NewType`
- `typing.Union` including nested structures
- `typing.Sequence`, `typing.List` including generic collections with `typing.TypeVar`;
- `typing.Set` and `typing.FrozenSet`
- `typing.Tuple`
- `typing.Dict`
- `typing.Mapping`
- `typing.Literal` (`typing_extensions.Literal` on Python prior 3.8);
- `typing.Generic[T, U, ...]`
- `typeit.sums.SumType`
- `typeit.custom_types.JsonString` - helpful when dealing with JSON strings encoded into JSON strings;
- `enum.Enum` derivatives
- `pathlib.Path` derivatives
- `pyrsistent.typing.PVector`
- `pyrsistent.typing.PMap`
- Forward references and recursive definitions
- Regular classes with annotated `__init__` methods (*dataclasses.dataclass* are supported as a consequence of this).

## 1.6 Sum Type

There are many ways to describe what a Sum Type (Tagged Union) is. Here's just a few of them:

- Wikipedia describes it as "a data structure used to hold a value that could take on several different, but fixed, types. Only one of the types can be in use at any one time, and a tag explicitly indicates which one is in use. It can be thought of as a type that has several "cases", each of which should be handled correctly when that type is manipulated";
- or you can think of Sum Types as data types that have more than one constructor, where each constructor accepts its own set of input data;
- or even simpler, as a generalized version of Enums, with some extra features.

`typeit` provides a limited implementation of Sum Types, that have functionality similar to default Python Enums, plus the ability of each tag to hold a value.

A new SumType is defined with the following signature:

```
from typeit.sums import SumType

class Payment(SumType):
    class Cash:
        amount: Money

    class Card:
        amount: Money
        card: CardCredentials

    class Phone:
        amount: Money
        provider: MobilePaymentProvider

    class JustThankYou:
        pass
```

`Payment` is a new Tagged Union (which is another name for a Sum Type, remember), that consists of four distinct possibilities: `Cash`, `Card`, `Phone`, and `JustThankYou`. These possibilities are called tags (or variants, or constructors) of `Payment`. In other words, any instance of `Payment` is either `Cash` or `Card` or `Phone` or `JustThankYou`, and is never two or more of them at the same time.

Now, let's observe the properties of this new type:

```
>>> adam_paid = Payment.Cash(amount=Money('USD', 10))
>>> jane_paid = Payment.Card(amount=Money('GBP', 8),
...                          card=CardCredentials(number='1234 5678 9012 3456',
...                                               holder='Jane Austen',
...                                               validity='12/24',
...                                               secret='***'))
>>> fred_paid = Payment.JustThankYou()
>>>
>>> assert type(adam_paid) is type(jane_paid) is type(fred_paid) is Payment
>>>
>>> assert isinstance(adam_paid, Payment)
>>> assert isinstance(jane_paid, Payment)
>>> assert isinstance(fred_paid, Payment)
>>>
>>> assert isinstance(adam_paid, Payment.Cash)
>>> assert isinstance(jane_paid, Payment.Card)
```

(continues on next page)

(continued from previous page)

```
>>> assert isinstance(fred_paid, Payment.JustThankYou)
>>>
>>> assert not isinstance(adam_paid, Payment.Card)
>>> assert not isinstance(adam_paid, Payment.JustThankYou)
>>>
>>> assert not isinstance(jane_paid, Payment.Cash)
>>> assert not isinstance(jane_paid, Payment.JustThankYou)
>>>
>>> assert not isinstance(fred_paid, Payment.Cash)
>>> assert not isinstance(fred_paid, Payment.Card)
>>>
>>> assert not isinstance(adam_paid, Payment.Phone)
>>> assert not isinstance(jane_paid, Payment.Phone)
>>> assert not isinstance(fred_paid, Payment.Phone)
>>>
>>> assert Payment('Phone') is Payment.Phone
>>> assert Payment('phone') is Payment.Phone
>>> assert Payment(Payment.Phone) is Payment.Phone
>>>
>>> paid = Payment(adam_paid)
>>> assert paid is adam_paid
```

As you can see, every variant constructs an instance of the same type `Payment`, and yet, every instance is identified with its own tag. You can use this tag to branch your business logic, like in a function below:

```
def notify_restaurant_owner(channel: Broadcaster, payment: Payment):
    if isinstance(payment, Payment.JustThankYou):
        channel.push(f'A customer said Big Thank You!')
    else:  # Cash, Card, Phone instances have the `payment.amount` attribute
        channel.push(f'A customer left {payment.amount}!')
```

And, of course, you can use Sum Types in signatures of your serializable data:

```
from typing import NamedTuple, Sequence
from typeit import TypeConstructor

class Payments(NamedTuple):
    latest: Sequence[Payment]

mk_payments, serialize_payments = TypeConstructor ^ Payments

json_ready = serialize_payments(Payments(latest=[adam_paid, jane_paid, fred_paid]))
payments = mk_payments(json_ready)
```

## 1.7 Constructor Flags

Constructor flags allow you to define global overrides that affect all structures (toplevel and nested) in a uniform fashion.

`typeit.flags.GlobalNameOverride` - useful when you want to globally modify output field names from pythonic *snake_style* to another naming convention scheme (*camelCase*, *dasherized-names*, etc). Here's a few examples:

```
import inflection

class FoldedData(NamedTuple):
    field_three: str

class Data(NamedTuple):
    field_one: str
    field_two: FoldedData

constructor, to_serializable = TypeConstructor & GlobalNameOverride(inflection.
↪camelize) ^ Data

data = Data(field_one='one',
            field_two=FoldedData(field_three='three'))

serialized = to_serializable(data)
```

the *serialized* dictionary will look like

```
{
    'FieldOne': 'one',
    'FieldTwo': {
        'FieldThree': 'three'
    }
}
```

`typeit.flags.NonStrictPrimitives` - disables strict checking of primitive types. With this flag, a type constructor for a structure with a `x: int` attribute annotation would allow input values of `x` to be strings that could be parsed as integer numbers. Without this flag, the type constructor will reject those values. The same rule is applicable to combinations of floats, ints, and bools:

```
construct, deconstruct = TypeConstructor ^ int
nonstrict_construct, nonstrict_deconstruct = TypeConstructor & NonStrictPrimitives ^␣
↪int

construct('1')              # raises typeit.Error
construct(1)                # OK
nonstrict_construct('1')    # OK
nonstrict_construct(1)      # OK
```

`typeit.flags.SumTypeDict` - switches the way SumType is parsed and serialized. By default, SumType is represented as a tuple of `(<tag>, <payload>)` in a serialized form. With this flag, it will be represented and parsed from a dictionary:

```
{
    <TAG_KEY>: <tag>,
    <payload>
}
```

i.e. the tag and the payload attributes will be merged into a single mapping, where `<TAG_KEY>` is the key by which the `<tag>` could be retrieved and set while parsing and serializing. The default value for `TAG_KEY` is `type`, but you can override it with the following syntax:

```
# Use "_type" as the key by which SumType's tag can be found in the mapping
mk_sum, serialize_sum = TypeConstructor & SumTypeDict('_type') ^ int
```

Here's an example how this flag changes the behaviour of the parser:

```
>>> class Payment(typeit.sums.SumType):
...     class Cash:
...         amount: str
...     class Card:
...         number: str
...         amount: str
...
>>> _, serialize_std_payment = typeit.TypeConstructor ^ Payment
>>> _, serialize_dict_payment = typeit.TypeConstructor & typeit.flags.SumTypeDict ^␣
↪Payment
>>> _, serialize_dict_v2_payment = typeit.TypeConstructor & typeit.flags.SumTypeDict('
↪$type') ^ Payment
>>>
>>> payment = Payment.Card(number='1111 1111 1111 1111', amount='10')
>>>
>>> print(serialize_std_payment(payment))
('card', {'number': '1111 1111 1111 1111', 'amount': '10'})

>>> print(serialize_dict_payment(payment))
{'type': 'card', 'number': '1111 1111 1111 1111', 'amount': '10'}

>>> print(serialize_dict_v2_payment(payment))
{'$type': 'card', 'number': '1111 1111 1111 1111', 'amount': '10'}
```

## 1.8 Extensions

See a cookbook for *Structuring Docker Compose Config*.

# CHAPTER 2

# Cookbook

## 2.1 Structuring Docker Compose Config

### 2.1.1 Sketching

Let's assume you have a docker-compose config to spin up Postgres and Redis backends:

```yaml
# Source code of ./docker-compose.yml
---
version: "2.0"
services:
  postgres:
    image: postgres:11.3-alpine
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: database
    ports:
      - 5433:5432

  redis:
    image: redis:5.0.4-alpine
    ports:
      - 6380:6379
```

Let's also assume that you want to manipulate this config from your Python program, but you don't like to deal with it as a dictionary, because your IDE doesn't hint you about available keys in dictionaries, and because you don't want to accidentally mix up host/guest ports of your containerized services. Hence, you decide to parse this config and put it into an appropriate Python representation that you would call `DockerConfig`.

And because writing boilerplate logic of this kind is always tiresome and is error-prone when done manually, you employ `typeit` for the task and do preliminary sketching with it:

```
$ typeit gen -s ./docker-compose.yml > ./docker_config.py
```

The command will generate `./docker_config.py` with definitions similar to this:

```python
# Source code of ./docker_config.py

from typing import Any, NamedTuple, Optional, Sequence
from typeit import TypeConstructor


class ServicesRedis(NamedTuple):
    image: str
    ports: Sequence[str]


class ServicesPostgresEnvironment(NamedTuple):
    POSTGRES_USER: str
    POSTGRES_PASSWORD: str
    POSTGRES_DB: str


class ServicesPostgres(NamedTuple):
    image: str
    environment: ServicesPostgresEnvironment
    ports: Sequence[str]


class Services(NamedTuple):
    postgres: ServicesPostgres
    redis: ServicesRedis


class Main(NamedTuple):
    version: str
    services: Services


mk_main, serialize_main = TypeConstructor ^ Main
```

Neat! This already is a good enough representation to play with, and we can verify that it does work as expected:

```python
# Source code of ./__init__.py

import yaml
from . import docker_config as dc

with open('./docker-compose.yml', 'rb') as f:
    config_dict = yaml.safe_load(f)

config = dc.mk_main(config_dict)
assert isinstance(config, dc.Main)
assert isinstance(config.services.postgres, dc.ServicesPostgres)
assert config.services.postgres.ports == ['5433:5432']
assert dc.serialize_main(config) == conf_dict
```

Now, let's refactor it a bit, so that `Main` becomes `DockerConfig` as we wanted, and `DockerConfig.version` is restricted to `"2.0"` and `"2.1"` only (and doesn't allow any random string):

```python
# Source code of ./__init__.py

from typing import Literal
# from typing_extensions import Literal  # on python < 3.8

class DockerConfig(NamedTuple):
    version: Literal['2.0', '2.1']
    services: Services


mk_config, serialize_config = TypeConstructor ^ DockerConfig
```

Looks good! There is just one thing that we still want to improve - service ports. And for that we need to extend our `TypeConstructor`.

### 2.1.2 Extending

At the moment our `config.services.postgres.ports` value is represented as a list of one string element `['5433:5432']`. It is still unclear which of those numbers belongs to what endpoint in a host <-> container network binding. You may remember Docker documentation saying that the actual format is `"host_port:container_port"`, however, it is inconvenient to spread this implicit knowledge across your Python codebase. Let's annotate these ports by introducing a new data type:

```python
# Source code of ./docker_config.py

class PortMapping(NamedTuple):
    host_port: int
    container_port: int
```

We want to use this type for port mappings instead of `str` in `ServicesRedis` and `ServicesPostgres` definitions:

```python
# Source code of ./docker_config.py

class ServicesRedis(NamedTuple):
    image: str
    ports: Sequence[PortMapping]


class ServicesPostgres(NamedTuple):
    image: str
    environment: ServicesPostgresEnvironment
    ports: Sequence[PortMapping]
```

This looks good, however, our type constructor doesn't know anything about conversion rules between a string value that comes from the YAML config and `PortMapping`. We need to explicitly define this rule:

```python
# Source code of ./docker_config.py

import typeit

class PortMappingSchema(typeit.schema.primitives.Str):
    def deserialize(self, node, cstruct: str) -> PortMapping:
        """ Converts input string value ``cstruct`` to ``PortMapping``
        """
```

(continues on next page)

(continued from previous page)

```
        ports_str = super().deserialize(node, cstruct)
        host_port, container_port = ports_str.split(':')
        return PortMapping(
            host_port=int(host_port),
            container_port=int(container_port)
        )

    def serialize(self, node, appstruct: PortMapping) -> str:
        """ Converts ``PortMapping`` back to string value suitable for YAML config
        """
        return super().serialize(
            node,
            f'{appstruct.host_port}:{appstruct.container_port}'
        )
```

Next, we need to tell our type constructor that all `PortMapping` values can be constructed with `PortMappingSchema` conversion schema:

```
# Source code of ./docker_config.py

Typer = typeit.TypeConstructor & PortMappingSchema[PortMapping]
```

We named the new extended type constructor `Typer`, and we're done with the task! Let's take a look at the final result.

### 2.1.3 Final Result

Here's what we get as the final solution for our task:

```
# Source code of ./docker_config.py

from typing import NamedTuple, Sequence
from typing import Literal
# from typing_extensions import Literal  # on python < 3.8

import typeit


class PortMapping(NamedTuple):
    host_port: int
    container_port: int


class PortMappingSchema(typeit.schema.primitives.Str):
    def deserialize(self, node, cstruct: str) -> PortMapping:
        """ Converts input string value ``cstruct`` to ``PortMapping``
        """
        ports_str = super().deserialize(node, cstruct)
        host_port, container_port = ports_str.split(':')
        return PortMapping(
            host_port=int(host_port),
            container_port=int(container_port)
        )

    def serialize(self, node, appstruct: PortMapping) -> str:
```

(continues on next page)

(continued from previous page)

```
        """ Converts ``PortMapping`` back to string value suitable
        for YAML config
        """
        return super().serialize(
            node,
            f'{appstruct.host_port}:{appstruct.container_port}'
        )


class ServicesRedis(NamedTuple):
    image: str
    ports: Sequence[PortMapping]


class ServicesPostgresEnvironment(NamedTuple):
    POSTGRES_USER: str
    POSTGRES_PASSWORD: str
    POSTGRES_DB: str


class ServicesPostgres(NamedTuple):
    image: str
    environment: ServicesPostgresEnvironment
    ports: Sequence[PortMapping]


class Services(NamedTuple):
    postgres: ServicesPostgres
    redis: ServicesRedis


class DockerConfig(NamedTuple):
    version: Literal['2', '2.1']
    services: Services


Typer = typeit.TypeConstructor & PortMappingSchema[PortMapping]
mk_config, serialize_config = Typer ^ DockerConfig
```

Let's test it!

```
# Source code of ./__init__.py

import yaml
from . import docker_config as dc

with open('./docker-compose.yml', 'rb') as f:
    config_dict = yaml.safe_load(f)

config = dc.mk_config(config_dict)

assert isinstance(config, dc.DockerConfig)
assert isinstance(config.services.postgres, dc.ServicesPostgres)
assert isinstance(config.services.postgres.ports[0], dc.PortMapping)
assert isinstance(config.services.redis.ports[0], dc.PortMapping)
assert dc.serialize_config(config) == config_dict
```

### 2.1.4 Notes

- Under the hood, `typeit` relies on Colander - a schema parsing and validation library that you may need to familiarise yourself with in order to understand `PortMappingSchema` definition.

# CHAPTER 3

# Indices and tables

- genindex
- modindex
- search