deserialize

May 24, 2023

Contents:

A library to make deserialization easy. To get started, just run *pip install deserialize*

CHAPTER 1

How it used to be

Without the library, if you want to convert:

```
{
    "a": 1,
    "b": 2
}
```

into a dedicated class, you had to do something like this:

```
class MyThing:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    @staticmethod
    def from_json(json_data):
        a_value = json_data.get("a")
        b_value = json_data.get("b")

        if a_value is None:
            raise Exception("'a' was None")
        elif b_value is None:
            raise Exception("'b' was None")
        elif type(a_value) != int:
            raise Exception("'a' was not an int")
        elif type(b_value) != int:
            raise Exception("'b' was not an int")

        return MyThing(a_value, b_value)

my_instance = MyThing.from_json(json_data)
```

CHAPTER 2

How it is now

With *deserialize* all you need to do is this:

```
import deserialize

class MyThing:
    a: int
    b: int

my_instance = deserialize.deserialize(MyThing, json_data)
```

That's it. It will pull out all the data and set it for you type checking and even checking for null values.

If you want null values to be allowed though, that's easy too:

```
from typing import Optional

class MyThing:
    a: Optional[int]
    b: Optional[int]
```

Now *None* is a valid value for these.

Types can be nested as deep as you like. For example, this is perfectly valid:

```
class Actor:
    name: str
    age: int

class Episode:
    title: str
    identifier: st
    actors: List[Actor]

class Season:
    episodes: List[Episode]
```

(continues on next page)

(continued from previous page)

```
    completed: bool

class TVShow:
    seasons: List[Season]
    creator: str
```

2.1 Custom Keys

It may be that you want to name your properties in your object something different to what is in the data. This can be for readability reasons, or because you have to (such as if your data item is named *__class__*). This can be handled too. Simply use the *key* annotation as follows:

```
@deserialize.key("identifier", "id")
class MyClass:
    value: int
    identifier: str
```

This will now assign the data with the key *id* to the field *identifier*. You can have multiple annotations to override multiple keys.

2.2 Unhandled Fields

Usually, if you don't specify the field in your definition, but it does exist in the data, you just want to ignore it. Sometimes however, you want to know if there is extra data. In this case, when calling *deserialize(...)* you can set *throw_on_unhandled=True* and it will raise an exception if any fields in the data are unhandled.

Additionally, sometimes you want this, but know of a particular field that can be ignored. You can mark these as allowed to be unhandled with the decorator *@allow_unhandled("key_name")*.

2.3 Ignored Keys

You may want some properties in your object that aren't loaded from disk, but instead created some other way. To do this, use the *ignore* decorator. Here's an example:

```
@deserialize.ignore("identifier")
class MyClass:
    value: int
    identifier: str
```

When deserializing, the library will now ignore the *identifier* property.

2.4 Parsers

Sometimes you'll want something in your object in a format that the data isn't in. For example, if you get the data:

```
{
    "successful": True,
    "timestamp": 1543770752
}
```

You may want that to be represented as:

```
class Result:
    successful: bool
    timestamp: datetime.datetime
```

By default, it will fail on this deserialization as the value in the data is not a timestamp. To correct this, use the *parser* decorator to tell it a function to use to parse the data. E.g.

```
@deserialize.parser("timestamp", datetime.datetime.fromtimestamp)
class Result:
    successful: bool
    timestamp: datetime.datetime
```

This will now detect when handling the data for the _key_ *timestamp* and run it through the parser function supplied before assigning it to your new class instance.

The parser is run _before_ type checking is done. This means that if you had something like *Optional[datetime.datetime]*, you should ensure your parser can handle the value being *None*. Your parser will obviously need to return the type that you have declared on the property in order to work.

2.5 Subclassing

Subclassing is supported. If you have a type *Shape* for example, which has a subclass *Rectangle*, any properties on *Shape* are supported if you try and decode some data into a ‘rectangle object.

2.6 Raw Storage

It can sometimes be useful to keep a reference to the raw data that was used to construct an object. To do this, simply set the *raw_storage_mode* paramater to *RawStorageMode.ROOT* or *RawStorageMode.ALL*. This will store the data in a parameter named *__deserialize_raw__* on the root object, or on all objects in the tree respectively.

2.7 Default Values

Some data will come to you with fields missing. In these cases, a default is often known. To do this, simply decorate your class like this:

```
@deserialize.default("value", 0)
class IntResult:
    successful: bool
    value: int
```

If you pass in data like *{“successful”: True}* this will deserialize to a default value of *0* for *value*. Note, that this would not deserialize since *value* is not optional: *{“successful”: True, “value”: None}*.

2.8 Post-processing

Not everything can be set on your data straight away. Some things need to be figured out afterwards. For this you need to do some post-processing. The easiest way to do this is through the *@constructed* decorator. This decorator takes a function which will be called whenever a new instance is constructed with that instance as an argument. Here's an example which converts polar coordinates from using degrees to radians:

```
data = {
    "angle": 180.0,
    "magnitude": 42.0
}

def convert_to_radians(instance):
    instance.angle = instance.angle * math.pi / 180

@deserialize.constructed(convert_to_radians)
class PolarCoordinate:
    angle: float
    magnitude: float

pc = deserialize.deserialize(PolarCoordinate, data)

print(pc.angle, pc.magnitude)

>>> 3.141592653589793 42.0
```

2.9 Default Values

Some data will come to you with fields missing. In these cases, a default is often known. To do this, simply decorate your class like this:

```
@deserialize.default("value", 0)
class IntResult:
    successful: bool
    value: int
```

If you pass in data like *{"successful": True}* this will deserialize to a default value of *0* for *value*. Note, that this would not deserialize since *value* is not optional: *{"successful": True, "value": None}*.

2.10 Post-processing

Not everything can be set on your data straight away. Some things need to be figured out afterwards. For this you need to do some post-processing. The easiest way to do this is through the *@constructed* decorator. This decorator takes a function which will be called whenever a new instance is constructed with that instance as an argument. Here's an example which converts polar coordinates from using degrees to radians:

```
data = {
    "angle": 180.0,
    "magnitude": 42.0
}

def convert_to_radians(instance):
```

(continues on next page)

(continued from previous page)

```
    instance.angle = instance.angle * math.pi / 180

@deserialize.constructed(convert_to_radians)
class PolarCoordinate:
    angle: float
    magnitude: float

pc = deserialize.deserialize(PolarCoordinate, data)

print(pc.angle, pc.magnitude)

>>> 3.141592653589793 42.0
```

2.11 Downcasting

Data often comes in the form of having the type as a field in the data. This can be difficult to parse. For example:

```
data = [
    {
        "data_type": "foo",
        "foo_prop": "Hello World",
    },
    {
        "data_type": "bar",
        "bar_prop": "Goodbye World",
    }
]
```

Since the fields differ between the two, there's no good way of parsing this data. You could use optional fields on some base class, try multiple deserializations until you find the right one, or do the deserialization based on a mapping you build of the *data_type* field. None of those solutions are elegant though, and all have issues if the types are nested. Instead, you can use the *downcast_field* and *downcast_identifier* decorators.

downcast_field is specified on a base class and gives the name of the field that contains the type information. *downcast_identifier* takes in a base class and an identifier (which should be one of the possible values of the *downcast_field* from the base class). Internally, when a class with a downcast field is detected, the field will be extacted, and a subclass with a matching identifier will be searched for. If no such class exists, an *UndefinedDowncastException* will be thrown.

Here's an example which would handle the above data:

```
@deserialize.downcast_field("data_type")
class MyBase:
    type_name: str


@deserialize.downcast_identifier(MyBase, "foo")
class Foo(MyBase):
    foo_prop: str


@deserialize.downcast_identifier(MyBase, "bar")
class Bar(MyBase):
    bar_prop: str
```

(continues on next page)

(continued from previous page)

```
result = deserialize.deserialize(List[MyBase], data)
```

Here, *result[0]* will be an instance of *Foo* and *result[1]* will be an instance of *Bar*.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search