PyJackson

Release 0.0.26

Jul 07, 2020

Contents

CHAPTER 1

Overview

docs	
tests	
package	

PyJackson is a serialization library based on type hinting

1.1 Example

Just type hint *__init__* and you are ready to go:

```
import pyjackson


class MyPayload:
    def __init__(self, string_field: str, int_field: int):
        self.string_field = string_field
        self.int_field = int_field


pyjackson.serialize(MyPayload('value', 10))  # {'string_field': 'value', 'int_field':␣
↪10}

pyjackson.deserialize({'string_field': 'value', 'int_field': 10}, MyPayload)  #␣
↪MyPayload('value', 10)
```

(continues on next page)

(continued from previous page)

More features and examples here and in examples dir.

1.2 Installation

```
pip install pyjackson
```

1.3 Documentation

https://pyjackson.readthedocs.io/

1.4 Development

To run all tests run:

```
tox
```

1.5 Licence

- Free software: Apache Software License 2.0

CHAPTER 2

Installation

At the command line:

```
pip install pyjackson
```

CHAPTER 3

Usage

3.1 Quickstart

To use PyJackson in a project, define a class with type hinted constructor arguments

```
class MyPayload:
    def __init__(self, string_field: str, int_field: int):
        self.string_field = string_field
        self.int_field = int_field
```

Now you are able to serialize instance of your class to dict and back with *serialize()* and *deserialize()*

```
instance = MyPayload('value', 10)
payload = pyjackson.serialize(instance)  # {'string_field': 'value', 'int_field': 10}

new_instance = pyjackson.deserialize(payload, MyPayload)  # MyPayload('value', 10)
```

It also works with nested structures and supports *typing* module generic annotations

```
class PayloadList:
    def __init__(self, payload_list: typing.List[MyPayload]):
        self.payload_list = payload_list


plist = PayloadList([instance])
payloads = pyjackson.serialize(plist)
# {'payload_list': [{'string_field': 'value', 'int_field': 10}]}
```

This code can be found in *examples/quickstart.py*

3.2 Type Hierarchy

If you have a hierarchy of types and you want to be able to deserialize them using base type, you need to register your base type with *type_field()* decorator. First argument is a name of class field, where you will put aliases for child types.

```
from pyjackson import deserialize, serialize
from pyjackson.decorators import type_field


@type_field('type_alias')
class Parent:
    type_alias = 'parent'  # also could be None for abstract parents


class Child1(Parent):
    type_alias = 'child1'

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


class Child2(Parent):
    type_alias = 'child2'

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


serialize(Child1(1), Parent)  # {'type_alias': 'child1', 'a': 1}
deserialize({'type_alias': 'child2', 'b': 'b'}, Parent)  # Child2('b')
```

3.3 Custom Serialization

If you want custom serialization logic for one of your class, or if you need to serialize external types with no type hints, you can implement custom serializer for them. Serializer is bound to the type you register and will be used when this type is encountered.

For example, you have class without type hints, or *__init__* differ from actual fields. Just implement StaticSerializer

```
class External:
    def __init__(self, a):
        self.b = a


class ExternalSerializer(StaticSerializer):
    real_type = External

    @classmethod
    def serialize(cls, instance: External) -> dict:
        return {'a': instance.b}

    @classmethod
    def deserialize(cls, obj: dict) -> object:
        return External(obj['a'])
```

Now you can serialize External

```
payload = serialize(External('value'))  # {'a': 'value'}
new_instance = deserialize(payload, External)  # External('value')
```

Like with simple types, it can be used in nested structures

```
class Container:
    def __init__(self, externals: List[External]):
        self.externals = externals


container_payload = serialize(Container([External('value')]))
new_container = deserialize(container_payload, Container)
```

If you need to parametrize yor serializer, you can implement generic Serializer, adding your parameters to serializers *__init__*. For example, you want to serialize lists of certain size with same values.

```
class SizedListSerializer(Serializer):
    real_type = list

    def __init__(self, size: int):
        self.size = size

    def serialize(self, instance: list) -> dict:
        if len(set(instance)) != 1:
            raise ValueError('Cannot serialize list with different values')
        return {'value': instance[0]}

    def deserialize(self, obj: dict) -> object:
        value = obj['value']
        return [value for _ in range(self.size)]


serializer = SizedListSerializer(3)

list_payload = serialize([1, 1, 1], serializer)  # {'value': 1}
new_list = deserialize(list_payload, serializer)  # [1, 1 ,1]
```

You can also use serializers in type hints

```
class OtherContainer:
    def __init__(self, sized_list: SizedListSerializer(3)):
        self.sized_list = sized_list


other_payload = serialize(OtherContainer([2, 2, 2]))  # {'sized_list': {'value': 2}}
new_other_container = deserialize(other_payload, OtherContainer)  # OtherContainer([2,
↪ 2, 2])
```

You can find this code in *examples/custom_serialization.py*

CHAPTER 4

pyjackson package

pyjackson.**deserialize**(*obj, as_class: Union[Type[CT_co], pyjackson.generics.Serializer]*)
Convert python dict into given class

Parameters

- **obj** – dict (or list or any primitive) to deserialize
- **as_class** – type or serializer

Returns deserialized instance of as_class (or real_type of serializer)

Raise DeserializationError

pyjackson.**dump**(*fp*, *obj*, *as_class: type = None*)
Serialize obj to JSON as *as_class* and write it to file-like *fp*

Parameters

- **fp** – file-like object to write
- **obj** – object to serialize
- **as_class** – type or serializer

Returns bytes written

pyjackson.**dumps**(*obj*, *as_class: type = None*)
Serialize obj to JSON string as *as_class*

Parameters

- **obj** – object to serialize
- **as_class** – type or serializer

Returns JSON string representation

pyjackson.**load**(*fp*, *as_class: type*)
Deserialize content of file-like *fp* to *as_class* instance

Parameters

- **fp** – file-like object to read
- **as_class** – type or serializer

Returns deserialized instance of as_class (or real_type of serializer)

pyjackson.**loads**(*payload: str*, *as_class: type*)
Deserialize *payload* to *as_class* instance

Parameters

- **payload** – JSON string
- **as_class** – type or serializer

Returns deserialized instance of as_class (or real_type of serializer)

pyjackson.**read**(*path: str, as_class: Type[T]*) → T
Deserialize object from file in *path* as as_class

Parameters

- **path** – path to file with JSON representation
- **as_class** – type or serializer

Returns deserialized instance of as_class (or real_type of serializer)

pyjackson.**serialize**(*obj, as_class: Union[Type[CT_co], pyjackson.generics.Serializer] = None*)
Convert object into JSON-compatible dict (or other structure)

Parameters

- **obj** – object to serialize
- **as_class** – type to serialize as or serializer

Returns JSON-compatible object

pyjackson.**write**(*path: str*, *obj*, *as_class: type = None*)
Serialize *obj* to JSON and write it to *path*

Parameters

- **path** – path to write JSON representation
- **obj** – object to serialize
- **as_class** – type or serializer

Returns bytes written

4.1 Submodules

4.1.1 pyjackson.core module

class pyjackson.core.**Position**
Bases: enum.Enum

Enum to change field with type information position

INSIDE = 0

OUTSIDE = 1

class pyjackson.core.**Unserializable**
Bases: object

Mixin type to signal that type is not serializable. *pyjackson.serialize()* will throw explicit error if called with instance of Unserializable (or object with nested Unserializable)

class pyjackson.core.**Comparable**
Bases: object

class pyjackson.core.**Field**(*name: str*, *type: type*, *has_default: bool*, *default: Any = None*)
Bases: *pyjackson.core.Comparable*

class pyjackson.core.**Signature**(*args*, *output*)
Bases: tuple

args
Alias for field number 0

output
Alias for field number 1

4.1.2 pyjackson.decorators module

class pyjackson.decorators.**cached_property**(*method*)
Bases: object

pyjackson.decorators.**make_string**(**fields*, *include_name=True*)
Decorator to create a *__str__* method for class based on *__init__* arguments

Usage: directly `@make_string()` on class declaration to include all fields or `@make_string(*fields, include_name)()` to alter params

Parameters

- **fields** – list of strings with field names
- **include_name** – whether to include class name

pyjackson.decorators.**as_list**(*cls: Type[CT_co]*)
Mark class to serialize it to list instead of dict

Parameters **cls** – class to mark

pyjackson.decorators.**type_field**(*field_name*, *position: pyjackson.core.Position = <Position.INSIDE: 0>*, *allow_reregistration=False*)
Class decorator for polymorphic hierarchies to define class field name, where subclass's type alias will be stored Use it on hierarchy root class, add class field with defined name to any subclasses The same field name will be used during deserialization

Parameters

- **field_name** – class field name to put alias for type
- **position** – where to put type alias
- **allow_reregistration** – whether to allow reregistration of same alias or throw error

pyjackson.decorators.**real_types**(**types*)
Register multiple real types for one serializer

pyjackson.decorators.**rename_fields**(***field_mapping*)
Change name of fields in payload. This behavior is inheritable and overridable for child classes

Parameters **field_mapping** – str-str mapping of field name (from constructor) to field name in payload

pyjackson.decorators.**camel_case**(*cls*)
Change snake_case field names to camelCase names

4.1.3 pyjackson.errors module

exception pyjackson.errors.**PyjacksonError**
Bases: Exception

General Pyjackson exception

exception pyjackson.errors.**DeserializationError**
Bases: *pyjackson.errors.PyjacksonError*

Deserialization exception

exception pyjackson.errors.**SerializationError**
Bases: *pyjackson.errors.PyjacksonError*

Serialization exception

exception pyjackson.errors.**UnserializableError**(*obj*)
Bases: *pyjackson.errors.SerializationError*

Raised when unserializable object is being serialized

4.1.4 pyjackson.generics module

class pyjackson.generics.**Serializer**
Bases: object

A base for defining custom serializers. # TODO definitely more docs here

real_type = None

deserialize(*obj: dict*) → object

serialize(*instance: object*) → dict

class pyjackson.generics.**StaticSerializer**
Bases: *pyjackson.generics.Serializer*

An easier way to define a serializer if it has no 'generic' arguments.

classmethod deserialize(*obj: dict*) → object

classmethod serialize(*instance: object*) → dict

4.1.5 pyjackson.pydantic_ext module

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

PyJackson could always use more documentation, whether as part of the official PyJackson docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/mike0sv/pyjackson/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *pyjackson* for local development:

1. Fork pyjackson (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:mike0sv/pyjackson.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Mikhail Sveshnikov - https://github.com/mike0sv

CHAPTER 7

Changelog

7.1 0.0.26 (2020-07-07)

- Experimental pydantic support

7.2 0.0.25 (2020-03-23)

- Support for int and float keys in dicts

7.3 0.0.24 (2020-02-22)

- Support for python 3.8

7.4 0.0.23 (2019-12-16)

- Fixed bug in subtype resolving

7.5 0.0.21 (2019-11-25)

- Fixed default type name

7.6 0.0.19 (2019-11-25)

- Allow subtype reregistration flag

7.7 0.0.18 (2019-11-22)

- Added support for full class path in type field (with importing logic)

7.8 0.0.17 (2019-11-21)

- Added Any support for serde skipping

7.9 0.0.16 (2019-11-15)

- Raise on subtype resolve error and fix for camel case forward ref resolving

7.10 0.0.15 (2019-11-11)

- Set class docstring and qualname of hierarchy root to be valid

7.11 0.0.14 (2019-11-05)

- Added decorator for camel case field renaming

7.12 0.0.13 (2019-11-03)

- Added decorator for field renaming

7.13 0.0.12 (2019-10-28)

- Fixed is_serializable for Field

7.14 0.0.11 (2019-10-28)

- Fixed is_serializable for Signature

7.15 0.0.10 (2019-10-16)

- Set class name and module of hierarchy root to be valid

7.16 0.0.9 (2019-10-09)

- Removed empty Serialzier __init__ method and fix for staticmethod in serializer

7.17 0.0.8 (2019-10-07)

- Changed is_collection to not include dict type

7.18 0.0.7 (2019-10-04)

- Added datetime.datetime serializer

7.19 0.0.6 (2019-10-02)

- Added Tuple[X, Y] and Tuple[X, ...] support

7.20 0.0.5 (2019-09-30)

- Fixed comparison of serializers

7.21 0.0.4 (2019-09-17)

- Added some examples and minor fixes

7.22 0.0.3 (2019-09-17)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index