Pyckson Documentation

Release 1.13.1

Jean Giard

May 04, 2023

Contents

1 User Documentation **3**
- 1.1 Installation . . . 3
- 1.2 Quickstart . . . 3
- 1.3 Advanced Usage . . . 5

2 Indices and tables **9**

Index **11**

Pyckson is a Python library to transform python objects to and from json.

Pyckson introspects your class __init__ signature to discover it's structure.

```
class Example:
    def __init__(self, things: List[int]):
        self.things = things
```

```
>>> from pyckson import serialize
>>> serialize(Example([1, 2, 3]))
{'things': [1, 2, 3]}
```

CHAPTER 1

User Documentation

1.1 Installation

1.1.1 Pip Install

To install Pyckson, simply run this simple command in your terminal of choice:

```
$ pip install pyckson
```

If you don't have pip installed, this Python installation guide can guide you through the process.

1.1.2 Get the Source Code

Pyckson is developed on GitHub, where the code is always available.

You can clone the public repository:

```
$ git clone https://github.com/antidot/Pyckson.git
```

Once you have a copy of the source, you can embed it in your own Python package, or install it into your site-packages easily:

```
$ python3 setup.py install
```

1.2 Quickstart

1.2.1 Writing Compatible Classes

Pyckson will assume that the classes you want to transform are written in a certain way :

- All class fields must be named parameters in your __init__ method

- All parameters must be assigned to the object with the same name
- All parameters must be type annotated

```
class Example:
    def __init__(self, foo: str, bar: List[int]):
        self.foo = foo
        self.bar = bar
```

1.2.2 Dataclasses

Pyckson also works with dataclasses in python3.7+

```
@dataclass
class Example:
    foo: str
    bar: List[int]
```

1.2.3 Conventions

Pyckson will produce and read json where your member names will have been tranformed to camelCase.

```
class CaseTest:
    def __init__(self, some_parameter: str):
        self.some_parameter = some_parameters
```

```
>>> pyckson.serialize(CaseTest('foo'))
{'someParameter': 'foo'}
```

1.2.4 Serializing Objects

The function *pyckson.serialize()* takes an object and return a dict-like structure.

```
>>> from pyckson import serialize
>>> serialize(Example('foo', [1, 2]))
{'foo': 'foo', 'bar': [1, 2]}
```

pyckson.**serialize**(*obj*)

Takes a object and produces a dict-like representation

Parameters **obj** – the object to serialize

You can also serialize lists, pyckson will handle the recursion

```
>>> from pyckson import serialize
>>> serialize([Example('foo', [1, 2]), Example('bar', [3, 4])])
[{'foo': 'foo', 'bar': [1, 2]}, {'foo': 'bar', 'bar': [3, 4]}]
```

1.2.5 Parsing Objects

The function *pyckson.parse()* takes a class and a dictionnary and return an instance of the class.

```
>>> form pyckson import parse
>>> example = parse(Example, {'foo': 'thing', 'bar': [1, 2, 3]})
>>> example
<__main__.Example object at 0x7fb177d86f28>
>>> example.foo
'thing'
>>> example.bar
[1, 2, 3]
```

pyckson.**parse**(*cls*, *value*)
Takes a class and a dict and try to build an instance of the class

Parameters

- **cls** – The class to parse
- **value** – either a dict, a list or a scalar value

Similarily to *pyckson.serialize()* you can also use the specific type *typing.List[cls]* to parse lists.

1.2.6 Utility Functions

Pyckson also includes some convenient wrappers to directly manipulate json strings with the json module.

pyckson.**dump**(*obj*, *fp*, ***kwargs*)
wrapper for `json.dump()`

pyckson.**dumps**(*obj*, ***kwargs*)
wrapper for `json.dumps()`

pyckson.**load**(*cls*, *fp*, ***kwargs*)
wrapper for `json.load()`

pyckson.**loads**(*cls*, *s*, ***kwargs*)
wrapper for `json.loads()`

1.3 Advanced Usage

1.3.1 Defaults

If you want to apply a specific pyckson behavior without having to annotate all your classes, you can configure a global decorator using `pyckson.set_defaults()`.

Most class level decorators are viable candidates. You can pass multiple arguments, or call the function multiple time to accumulate behaviors.

```
from pyckson import set_default, no_camel_case, date_formatter
from pyckson.date.arrow import ArrowStringFormatter
set_defaults(no_camel_case, date_formatter(ArrowStringFormatter()))
```

1.3.2 Enums

Pyckson can serialize and parse enums using the `Enum.name()` function.

Sometimes it's convenient to parse enums in a case-insensitive way, to do so you can use the *pyckson.caseinsensitive()* decorator.

pyckson.**caseinsensitive**(*cls*)
Annotation function to set an Enum to be case insensitive on parsing

Pyckson can also use the value part of the enum to perform parsing/serialization with the *pyckson.enumvalues()* decorator.

pyckson.**enumvalues**(*cls*)
Annotation function to set an Enum to use values instead of name for serialization

1.3.3 Dates

By default Pyckson does not apply any special treatment to date objects, meaning that if you use serialize you will get a dictionnary with date-type values, and json.dumps will not be able to serialize your object.

You can use the `pyckson.date_formatter()` decorator to override serialization behavior for fields of a class (it does not apply recursively), or configure it globally with `pyckson.set_defaults()`.

1.3.4 Custom Date Formatters

Pyckson provides two date formatters based on the arrow library : `pyckson.date.arrow.ArrowStringFormatter` and `pyckson.date.arrow.ArrowTimestampFormatter`.

To use them configure them appropriately like

```
import pyckson
from pyckson.date.arrow import ArrowStringFormatter
pyckson.set_defaults(pyckson.date_formatter(ArrowStringFormatter()))
```

or

```
import pyckson
from pyckson.date.arrow import ArrowStringFormatter

@pyckson.date_formatter(ArrowStringFormatter())
class Foo:
    def __init__(bar: datetime):
        self.bar = bar
```

You should then be able to properly serialize dates to json-strings

```
>>> pyckson.dumps(Foo(datetime(2018, 3, 8, 13, 58, 0)))
'{"bar": "2013-05-05T13:58:00+00:00"}'
```

1.3.5 Nulls serialisation

By default pyckson will not serialize optional empty attributes. You can switch this behavior and force it to assign the null value to the generated json.

```
import pyckson
pyckson.set_defaults(pyckson.explicit_nulls)
```

or

```
import pyckson

@pyckson.explicit_nulls()
class Foo:
    def __init__(bar: Optional[str] = None):
        self.bar = bar
```

You should see explicit null values in the output

```
>>> pyckson.dumps(Foo(bar=None))
'{"bar": null}'
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index

C

caseinsensitive() (*in module pyckson*), 6

D

dump() (*in module pyckson*), 5
dumps() (*in module pyckson*), 5

E

enumvalues() (*in module pyckson*), 6

L

load() (*in module pyckson*), 5
loads() (*in module pyckson*), 5

P

parse() (*in module pyckson*), 5

S

serialize() (*in module pyckson*), 4