json-five

Release 0.1.0

Spencer Phillip Young

Aug 04, 2023

CONTENTS:

GitHub

CHAPTER

ONE

QUICKSTART

1.1 Installation

```
pip install json-five
```

1.2 Basic Usage

Suppose you have a JSON5 compliant file `my-json-file.json`

```
// This is a JSON5 file!
{'foo': 'bar'}
```

You can load this file to Python like so:

```
import json5
with open('my-json-file.json') as f:
    data = json5.load(f)
```

You can also work directly with strings

```
import json5
json_string = '{json5 /* identifiers dont need quotes */: "values do though"}'
data = json5.loads(json_string)
```

Want to do more? Check out *Extending json-five* to dive deeper!

CHAPTER

TWO

EXTENDING JSON-FIVE

2.1 The `json` way

`json5.load` and `json5.loads` support a similar interface to the stdlib `json` module. Specifically, you can provide the following arguments that have the same meaning as in `json.load`:

- `parse_int`
- `parse_float`
- `parse_constant`
- `object_hook`
- `object_pairs_hook`

This is convenient if you have existing code that uses these arguments with the `json` module, but want to also support JSON5. These options are also useful as a simple way to customize parsing of json types.

Additionally, a new hook keyword argument, `parse_json5_identifiers`, is available to help users control the output of parsing identifiers. By default, JSON5 Identifiers in object keys are returned as a `JsonIdentifier` object, which is a subclass of `str` (meaning it's compatible anywhere `str` is accepted). This helps keep keys the same round-trip, rather than converting unquoted identifiers into quoted strings, such that `dumps(loads(text)) == text` (in this case).

You can change this behavior with the `parse_json5_identifiers` keyword argument with a callable that receives the *JsonIdentifier* object and its return value is used instead. For example, you can specify `parse_json5_identifiers=str` to convert identifiers to normal strings, such that `dumps(loads('{foo: "bar"}')) == '{"foo": "bar"}'`.

However, this package does not support the `cls` keyword found in the standard library `json` module. If you want to implement custom serializers/deserializers, read on about custom loaders/dumpers.

2.2 Custom Loaders and Dumpers

This package uses "Loaders" as part of the deserialization of JSON text to Python. "Dumpers" are used to serialize Python objects to JSON text.

The entry points for loaders and dumpers are the `load` and `dump` methods, respectively. You can override these methods to implement custom loading of models or dumping of objects.

2.2.1 Extending the default loader

The default loader takes in a model and produces, in the default case, Python objects.

As a simple example, you can extend the default loader with your own to customize loading of lists. Here, I'll create a custom loader that, when it encounters an array (`json5.model.JSONArray`) with with only one value, it will return the single value, rather than a single-item array.

```
from json5.loader import DefaultLoader, loads
from json5.model import JSONArray


class MyCustomLoader(DefaultLoader):
    def load(self, node):
        if isinstance(node, JSONArray):
            return self.json_array_to_python(node)
        else:
            return super().load(node)

    def json_array_to_python(self, node):
        if len(node.values) == 1:
            return self.load(node.values[0])
        else:
            return super().json_array_to_python(node)
```

The `loads` function accepts a `loader` keyword argument, where the custom loader can be passed in.

```
json_string = "{foo: ['bar', 'baz'], bacon: ['eggs']}"
loads(json_string)  # Using the regular default loader
# {'foo': ['bar', 'baz'], 'bacon': ['eggs']}

loads(json_string, loader=MyCustomLoader())  # use the custom loader instead
# {'foo': ['bar', 'baz'], 'bacon': 'eggs'}
```

2.2.2 Extending the default dumper

Extending the dumper follows a similar principle as extending the loader.

As an example, I'll make a custom dumper that dumps booleans `True` and `False` to integers instead of the JSON `true` or `false`.

```
from json5.dumper import DefaultDumper, dumps

class MyCustomDumper(DefaultDumper):
    def dump(self, node):
        if isinstance(node, bool):
            return self.bool_to_json(node)
        else:
            return super().dump(node)

    def bool_to_json(self, node):
        super().dump(int(node.value))
```

And you can see the effects

```
>>> dumps([True, False])
'[true, false]'
>>> dumps([True, False], dumper=MyCustomDumper())
'[1, 2]'
```

2.3 Other loaders/dumpers and tools

Besides the default loader, there is also the `ModelLoader` which simply returns the raw model with no additional processing.

Besides the default dumper, there is also the `ModelDumper` which takes a model and serializes it to JSON.

The `json5.dumper.modelize` function can take python objects and convert them to a model.

```
from json5.dumper import modelize
obj = ['foo', 123, True]
modelize(obj)
```

The resulting model:

```
JSONArray(
    values=[
        SingleQuotedString(characters='foo', raw_value="'foo'"),
        Integer(raw_value='123', value=123, is_hex=False),
        BooleanLiteral(value=True),
    ],
    trailing_comma=None,
)
```

CHAPTER

THREE

HOW THIS PACKAGE WORKS

This is an overview of how the internals of this package work. The code demonstrated here is not necessarily intended to be used by users!

If you're wondering how to use this package, see *QuickStart* instead.

3.1 Deserializing JSON to Python; the journey

The first step in deserialization is tokenizing. Text, assuming it is conforming to the JSON5 spec, is parsed into _tokens_. The tokens are then _parsed_ to produce a representative _model_ of the JSON structure. Finally, that model is _loaded_ where each node in the model is turned into an instance of a Python data type.

Let's explore this process interactively.

3.1.1 tokenizing

Tokenizing is the first step in turning JSON text into Python objects. Let's look at tokenizing a very simple empty JSON object `{}`

```
>>> from json5.tokenizer import tokenize
>>> json_string = "{}"
>>> tokens = tokenize(json_string)
>>> for token in tokens:
...     print(token)
...
Token(type='LBRACE', value='{', lineno=1, index=0)
Token(type='RBRACE', value='}', lineno=1, index=1)
```

As you can see, this broke down into two tokens: the left brace and the right brace.

For good measure, let's see a slightly more complex tokenization

```
for token in tokenize("{foo: 'bar'}"):
    print(token)

Token(type='LBRACE', value='{', lineno=1, index=0)
Token(type='NAME', value='foo', lineno=1, index=1)
Token(type='COLON', value=':', lineno=1, index=4)
Token(type='WHITESPACE', value=' ', lineno=1, index=5)
Token(type='SINGLE_QUOTE_STRING', value="'bar'", lineno=1, index=6)
Token(type='RBRACE', value='}', lineno=1, index=11)
```

These tokens will be used to build a model in the next step.

3.1.2 Parsing and models

As the text is processed into tokens, the stream of tokens is parsed into a model representing the JSON structure.

Let's start with the same simple example of an empty JSON object `{}`

```
>>> from json5.tokenizer import tokenize
>>> from json5.parser import parse_tokens
>>> tokens = tokenize("{}")
>>> model = parse_tokens(tokens)
>>> model
JSONText(value=JSONObject(key_value_pairs=[], trailing_comma=None))
```

The tokens were parsed to produce a model. Each production (part) in the model more or less represents a part of the JSON5 grammar. `JSONText` is always the root production of the model for any JSON5 document.

Let's look at a more complex model for the JSON text `{foo:  0xC0FFEE}` – This model has been 'prettified' for this doc:

```
JSONText(
    value=JSONObject(
        key_value_pairs=[
            KeyValuePair(
                key=Identifier(name='foo'),
                value=Integer(raw_value='0xC0FFEE', value=12648430, is_hex=True),
            )
        ],
        trailing_comma=None,
    )
)
```

You can also build model objects 'manually' without any source text.

```
from json5.model import *
model = JSONText(value=JSONObject(KeyValuePair(key=Identifier('bacon'),␣
↪value=Infinity())))
```

3.1.3 Loading

Once we have a model in-hand, we can use it to generate Python object representation from the model. To do this, specialized classes, called Loaders, are used. Loaders take a model and produce something else, like Python data types.

In this example, we'll just create a model instead of parsing one from text and turn it into Python using the default loader (the default loader is used when calling `loads` by default.

```
>>> from json5.loader import DefaultLoader
>>> from json5.model import *
>>> loader = DefaultLoader()
>>> model = JSONText(value=JSONObject(KeyValuePair(key=Identifier('bacon'),␣
↪value=Infinity())))
>>> loader.load(model)
{'bacon': inf}
```

3.2 Serializing to JSON

Objects can be serialized to JSON using _dumpers_. A dumper takes and object and writes JSON text representing the object. The default dumper dumps python objects directly to JSON text.

```
>>> from json5 import dumps
>>> dumps(['foo', 'bar', 'baz'])
'["foo", "bar", "baz"]'
```

CHAPTER

FOUR

WORKING WITH COMMENTS; ROUND-TRIP SUPPORT

In order to work with comments, you must work with the raw model.

Each node in the model has two special attributes: `.wsc_before` and `.wsc_after`. These attributes are a list of any whitespace or comments that appear before or after the node.

```
from json5.loader import loads, ModelLoader
from json5.dumper import dumps, ModelDumper
from json5.model import BlockComment
json_string = """{"foo": "bar"}"""
model = loads(json_string, loader=ModelLoader())
print(model.value.key_value_pairs[0].value.wsc_before)  # [' ']
model.value.key_value_pairs[0].key.wsc_before.append(BlockComment("/* comment */"))
dumps(model, dumper=ModelDumper()) # '{/* comment */"foo": "bar"}'
```

This section will be expanded with time, the API for working with comments will likely change alot in future versions.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search