fastavro Documentation

Release 1.8.3

Miki Tebeka

Sep 09, 2023

CONTENTS

The current Python *avro* package is dog slow.

On a test case of about 10K records, it takes about 14sec to iterate over all of them. In comparison the JAVA *avro* SDK does it in about 1.9sec.

fastavro is an alternative implementation that is much faster. It iterates over the same 10K records in 2.9sec, and if you use it with PyPy it'll do it in 1.5sec (to be fair, the JAVA benchmark is doing some extra JSON encoding/decoding).

If the optional C extension (generated by Cython) is available, then *fastavro* will be even faster. For the same 10K records it'll run in about 1.7sec.

CHAPTER

ONE

SUPPORTED FEATURES

- File Writer
- File Reader (iterating via records or blocks)
- Schemaless Writer
- Schemaless Reader
- JSON Writer
- JSON Reader
- Codecs (Snappy, Deflate, Zstandard, Bzip2, LZ4, XZ)
- Schema resolution
- Aliases
- Logical Types
- Parsing schemas into the canonical form
- Schema fingerprinting

CHAPTER

TWO

MISSING FEATURES

- Anything involving Avro's RPC features

CHAPTER

THREE

EXAMPLE

```
from fastavro import writer, reader, parse_schema

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}
parsed_schema = parse_schema(schema)

# 'records' can be an iterable (including generator)
records = [
    {u'station': u'011990-99999', u'temp': 0, u'time': 1433269388},
    {u'station': u'011990-99999', u'temp': 22, u'time': 1433270389},
    {u'station': u'011990-99999', u'temp': -11, u'time': 1433273379},
    {u'station': u'012650-99999', u'temp': 111, u'time': 1433275478},
]

# Writing
with open('weather.avro', 'wb') as out:
    writer(out, parsed_schema, records)

# Reading
with open('weather.avro', 'rb') as fo:
    for record in reader(fo):
        print(record)
```

CHAPTER

FOUR

DOCUMENTATION

CHAPTER

FIVE

FASTAVRO

5.1 fastavro.read

class reader(*fo: IO | AvroJSONDecoder, reader_schema: str | List | Dict | None = None, return_record_name: bool = False, return_record_name_override: bool = False, handle_unicode_errors: str = 'strict', return_named_type: bool = False, return_named_type_override: bool = False*)

Iterator over records in an avro file.

Parameters

- **fo** -- File-like object to read from
- **reader_schema** -- Reader schema
- **return_record_name** -- If true, when reading a union of records, the result will be a tuple where the first value is the name of the record and the second value is the record itself
- **return_record_name_override** -- If true, this will modify the behavior of return_record_name so that the record name is only returned for unions where there is more than one record. For unions that only have one record, this option will make it so that the record is returned by itself, not a tuple with the name.
- **return_named_type** -- If true, when reading a union of named types, the result will be a tuple where the first value is the name of the type and the second value is the record itself NOTE: Using this option will ignore return_record_name and return_record_name_override
- **return_named_type_override** -- If true, this will modify the behavior of return_named_type so that the named type is only returned for unions where there is more than one named type. For unions that only have one named type, this option will make it so that the named type is returned by itself, not a tuple with the name
- **handle_unicode_errors** -- Default *strict*. Should be set to a valid string that can be used in the errors argument of the string decode() function. Examples include *replace* and *ignore*

Example:

```
from fastavro import reader
with open('some-file.avro', 'rb') as fo:
    avro_reader = reader(fo)
    for record in avro_reader:
        process_record(record)
```

The *fo* argument is a file-like object so another common example usage would use an *io.BytesIO* object like so:

```
from io import BytesIO
from fastavro import writer, reader

fo = BytesIO()
writer(fo, schema, records)
fo.seek(0)
for record in reader(fo):
    process_record(record)
```

metadata

Key-value pairs in the header metadata

codec

The codec used when writing

writer_schema

The schema used when writing

reader_schema

The schema used when reading (if provided)

class block_reader(*fo: IO*, *reader_schema: str | List | Dict | None = None*, *return_record_name: bool = False*, *return_record_name_override: bool = False*, *handle_unicode_errors: str = 'strict'*, *return_named_type: bool = False*, *return_named_type_override: bool = False*)

Iterator over *Block* in an avro file.

Parameters

- **fo** -- Input stream
- **reader_schema** -- Reader schema
- **return_record_name** -- If true, when reading a union of records, the result will be a tuple where the first value is the name of the record and the second value is the record itself
- **return_record_name_override** -- If true, this will modify the behavior of return_record_name so that the record name is only returned for unions where there is more than one record. For unions that only have one record, this option will make it so that the record is returned by itself, not a tuple with the name.
- **return_named_type** -- If true, when reading a union of named types, the result will be a tuple where the first value is the name of the type and the second value is the record itself NOTE: Using this option will ignore return_record_name and return_record_name_override
- **return_named_type_override** -- If true, this will modify the behavior of return_named_type so that the named type is only returned for unions where there is more than one named type. For unions that only have one named type, this option will make it so that the named type is returned by itself, not a tuple with the name
- **handle_unicode_errors** -- Default *strict*. Should be set to a valid string that can be used in the errors argument of the string decode() function. Examples include *replace* and *ignore*

Example:

```
from fastavro import block_reader
with open('some-file.avro', 'rb') as fo:
    avro_reader = block_reader(fo)
    for block in avro_reader:
        process_block(block)
```

metadata

Key-value pairs in the header metadata

codec

The codec used when writing

writer_schema

The schema used when writing

reader_schema

The schema used when reading (if provided)

class Block(*bytes_*, *num_records*, *codec*, *reader_schema*, *writer_schema*, *named_schemas*, *offset*, *size*, *options*)

An avro block. Will yield records when iterated over

num_records

Number of records in the block

writer_schema

The schema used when writing

reader_schema

The schema used when reading (if provided)

offset

Offset of the block from the beginning of the avro file

size

Size of the block in bytes

schemaless_reader(*fo: IO*, *writer_schema: str | List | Dict*, *reader_schema: str | List | Dict | None = None*, *return_record_name: bool = False*, *return_record_name_override: bool = False*, *handle_unicode_errors: str = 'strict'*, *return_named_type: bool = False*, *return_named_type_override: bool = False*) → None | str | float | int | Decimal | bool | bytes | List | Dict

Reads a single record written using the *schemaless_writer()*

Parameters

- **fo** -- Input stream
- **writer_schema** -- Schema used when calling schemaless_writer
- **reader_schema** -- If the schema has changed since being written then the new schema can be given to allow for schema migration
- **return_record_name** -- If true, when reading a union of records, the result will be a tuple where the first value is the name of the record and the second value is the record itself
- **return_record_name_override** -- If true, this will modify the behavior of return_record_name so that the record name is only returned for unions where there is more than one record. For unions that only have one record, this option will make it so that the record is returned by itself, not a tuple with the name.
- **return_named_type** -- If true, when reading a union of named types, the result will be a tuple where the first value is the name of the type and the second value is the record itself NOTE: Using this option will ignore return_record_name and return_record_name_override
- **return_named_type_override** -- If true, this will modify the behavior of return_named_type so that the named type is only returned for unions where there is more

than one named type. For unions that only have one named type, this option will make it so that the named type is returned by itself, not a tuple with the name

- **handle_unicode_errors** -- Default *strict*. Should be set to a valid string that can be used in the errors argument of the string decode() function. Examples include *replace* and *ignore*

Example:

```
parsed_schema = fastavro.parse_schema(schema)
with open('file', 'rb') as fp:
    record = fastavro.schemaless_reader(fp, parsed_schema)
```

Note: The `schemaless_reader` can only read a single record.

is_avro(*path_or_buffer: str | IO*) → bool

Return True if path (or buffer) points to an Avro file. This will only work for avro files that contain the normal avro schema header like those create from *writer()*. This function is not intended to be used with binary data created from *schemaless_writer()* since that does not include the avro header.

Parameters

path_or_buffer -- Path to file

5.2 fastavro.write

writer(*fo: IO | AvroJSONEncoder*, *schema: str | List | Dict*, *records: Iterable[Any]*, *codec: str = 'null'*, *sync_interval: int = 16000*, *metadata: Dict[str, str] | None = None*, *validator: bool = False*, *sync_marker: bytes = b''*, *codec_compression_level: int | None = None*, ***, *strict: bool = False*, *strict_allow_default: bool = False*, *disable_tuple_notation: bool = False*)

Write records to fo (stream) according to schema

Parameters

- **fo** -- Output stream
- **schema** -- Writer schema
- **records** -- Records to write. This is commonly a list of the dictionary representation of the records, but it can be any iterable
- **codec** -- Compression codec, can be 'null', 'deflate' or 'snappy' (if installed)
- **sync_interval** -- Size of sync interval
- **metadata** -- Header metadata
- **validator** -- If true, validation will be done on the records
- **sync_marker** -- A byte string used as the avro sync marker. If not provided, a random byte string will be used.
- **codec_compression_level** -- Compression level to use with the specified codec (if the codec supports it)
- **strict** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states
- **strict_allow_default** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states unless it is a missing field that has a default value in the schema

- **disable_tuple_notation** -- If set to True, tuples will not be treated as a special case. Therefore, using a tuple to indicate the type of a record will not work

Example:

```
from fastavro import writer, parse_schema

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}
parsed_schema = parse_schema(schema)

records = [
    {u'station': u'011990-99999', u'temp': 0, u'time': 1433269388},
    {u'station': u'011990-99999', u'temp': 22, u'time': 1433270389},
    {u'station': u'011990-99999', u'temp': -11, u'time': 1433273379},
    {u'station': u'012650-99999', u'temp': 111, u'time': 1433275478},
]

with open('weather.avro', 'wb') as out:
    writer(out, parsed_schema, records)
```

The *fo* argument is a file-like object so another common example usage would use an *io.BytesIO* object like so:

```
from io import BytesIO
from fastavro import writer

fo = BytesIO()
writer(fo, schema, records)
```

Given an existing avro file, it's possible to append to it by re-opening the file in *a+b* mode. If the file is only opened in *ab* mode, we aren't able to read some of the existing header information and an error will be raised. For example:

```
# Write initial records
with open('weather.avro', 'wb') as out:
    writer(out, parsed_schema, records)

# Write some more records
with open('weather.avro', 'a+b') as out:
    writer(out, None, more_records)
```

Note: When appending, any schema provided will be ignored since the schema in the avro file will be re-used. Therefore it is convenient to just use None as the schema.

schemaless_writer(*fo: IO*, *schema: str | List | Dict*, *record: Any*, ***, *strict: bool = False*, *strict_allow_default: bool = False*, *disable_tuple_notation: bool = False*)

Write a single record without the schema or header information

Parameters

- **fo** -- Output file
- **schema** -- Schema
- **record** -- Record to write
- **strict** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states
- **strict_allow_default** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states unless it is a missing field that has a default value in the schema
- **disable_tuple_notation** -- If set to True, tuples will not be treated as a special case. Therefore, using a tuple to indicate the type of a record will not work

Example:

```
parsed_schema = fastavro.parse_schema(schema)
with open('file', 'wb') as fp:
    fastavro.schemaless_writer(fp, parsed_schema, record)
```

Note: The `schemaless_writer` can only write a single record.

5.2.1 Using the tuple notation to specify which branch of a union to take

Since this library uses plain dictionaries to represent a record, it is possible for that dictionary to fit the definition of two different records.

For example, given a dictionary like this:

```
{"name": "My Name"}
```

It would be valid against both of these records:

```
child_schema = {
    "name": "Child",
    "type": "record",
    "fields": [
        {"name": "name", "type": "string"},
        {"name": "favorite_color", "type": ["null", "string"]},
    ]
}

pet_schema = {
    "name": "Pet",
    "type": "record",
    "fields": [
        {"name": "name", "type": "string"},
        {"name": "favorite_toy", "type": ["null", "string"]},
    ]
}
```

This becomes a problem when a schema contains a union of these two similar records as it is not clear which record the dictionary represents. For example, if you used the previous dictionary with the following schema, it wouldn't be clear if the record should be serialized as a *Child* or a *Pet*:

```
household_schema = {
    "name": "Household",
    "type": "record",
    "fields": [
        {"name": "address", "type": "string"},
        {
            "name": "family_members",
            "type": {
                "type": "array", "items": [
                    {
                        "name": "Child",
                        "type": "record",
                        "fields": [
                            {"name": "name", "type": "string"},
                            {"name": "favorite_color", "type": ["null", "string"]},
                        ]
                    }, {
                        "name": "Pet",
                        "type": "record",
                        "fields": [
                            {"name": "name", "type": "string"},
                            {"name": "favorite_toy", "type": ["null", "string"]},
                        ]
                    }
                ]
            }
        },
    ]
}
```

To resolve this, you can use a tuple notation where the first value of the tuple is the fully namespaced record name and the second value is the dictionary. For example:

```
records = [
    {
        "address": "123 Drive Street",
        "family_members": [
            ("Child", {"name": "Son"}),
            ("Child", {"name": "Daughter"}),
            ("Pet", {"name": "Dog"}),
        ]
    }
]
```

5.2.2 Using the record hint to specify which branch of a union to take

In addition to the tuple notation for specifying the name of a record, you can also include a special *-type* attribute (note that this attribute is *-type*, not *type*) on a record to do the same thing. So the example above which looked like this:

```
records = [
    {
        "address": "123 Drive Street",
        "family_members": [
            ("Child", {"name": "Son"}),
            ("Child", {"name": "Daughter"}),
            ("Pet", {"name": "Dog"}),
        ]
    }
]
```

Would now look like this:

```
records = [
    {
        "address": "123 Drive Street",
        "family_members": [
            {"-type": "Child", "name": "Son"},
            {"-type": "Child", "name": "Daughter"},
            {"-type": "Pet", "name": "Dog"},
        ]
    }
]
```

Unlike the tuple notation which can be used with any avro type in a union, this *-type* hint can only be used with records. However, this can be useful if you want to make a single record dictionary that can be used both in and out of unions.

5.3 fastavro.json_read

json_reader(*fo: ~typing.IO*, *schema: str | ~typing.List | ~typing.Dict*, *reader_schema: str | ~typing.List | ~typing.Dict | None = None*, ***, *decoder=<class 'fastavro.io.json_decoder.AvroJSONDecoder'>*) → *reader*

Iterator over records in an avro json file.

Parameters

- **fo** -- File-like object to read from
- **schema** -- Original schema used when writing the JSON data
- **reader_schema** -- If the schema has changed since being written then the new schema can be given to allow for schema migration
- **decoder** -- By default the standard AvroJSONDecoder will be used, but a custom one could be passed here

Example:

```
from fastavro import json_reader

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ]
}

with open('some-file', 'r') as fo:
    avro_reader = json_reader(fo, schema)
    for record in avro_reader:
        print(record)
```

5.4 fastavro.json_write

json_writer(*fo: ~typing.IO, schema: str | ~typing.List | ~typing.Dict, records: ~typing.Iterable[~typing.Any], *, write_union_type: bool = True, validator: bool = False, encoder=<class 'fastavro.io.json_encoder.AvroJSONEncoder'>, strict: bool = False, strict_allow_default: bool = False, disable_tuple_notation: bool = False*) → None

Write records to fo (stream) according to schema

Parameters

- **fo** -- File-like object to write to
- **schema** -- Writer schema
- **records** -- Records to write. This is commonly a list of the dictionary representation of the records, but it can be any iterable
- **write_union_type** -- Determine whether to write the union type in the json message. If this is set to False the output will be clear json. It may however not be decodable back to avro record by *json_read*.
- **validator** -- If true, validation will be done on the records
- **encoder** -- By default the standard AvroJSONEncoder will be used, but a custom one could be passed here
- **strict** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states
- **strict_allow_default** -- If set to True, an error will be raised if records do not contain exactly the same fields that the schema states unless it is a missing field that has a default value in the schema
- **disable_tuple_notation** -- If set to True, tuples will not be treated as a special case. Therefore, using a tuple to indicate the type of a record will not work

Example:

```
from fastavro import json_writer, parse_schema

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}
parsed_schema = parse_schema(schema)

records = [
    {u'station': u'011990-99999', u'temp': 0, u'time': 1433269388},
    {u'station': u'011990-99999', u'temp': 22, u'time': 1433270389},
    {u'station': u'011990-99999', u'temp': -11, u'time': 1433273379},
    {u'station': u'012650-99999', u'temp': 111, u'time': 1433275478},
]

with open('some-file', 'w') as out:
    json_writer(out, parsed_schema, records)
```

5.5 fastavro.schema

parse_schema(*schema: str | List | Dict*, *named_schemas: Dict[str, Dict] | None = None*, ***, *expand: bool = False*, *_write_hint: bool = True*, *_force: bool = False*, *_ignore_default_error: bool = False*) → str | List | Dict

Returns a parsed avro schema

It is not necessary to call parse_schema but doing so and saving the parsed schema for use later will make future operations faster as the schema will not need to be reparsed.

Parameters

- **schema** -- Input schema
- **named_schemas** -- Dictionary of named schemas to their schema definition
- **expand** -- If true, named schemas will be fully expanded to their true schemas rather than being represented as just the name. This format should be considered an output only and not passed in to other reader/writer functions as it does not conform to the avro specification and will likely cause an exception
- **_write_hint** -- Internal API argument specifying whether or not the __fastavro_parsed marker should be added to the schema
- **_force** -- Internal API argument. If True, the schema will always be parsed even if it has been parsed and has the __fastavro_parsed marker
- **_ignore_default_error** -- Internal API argument. If True, when a union has the wrong default value, an error will not be raised.

Example:

```
from fastavro import parse_schema
from fastavro import writer

parsed_schema = parse_schema(original_schema)
with open('weather.avro', 'wb') as out:
    writer(out, parsed_schema, records)
```

Sometimes you might have two schemas where one schema references another. For the sake of example, let's assume you have a *Parent* schema that references a *Child* schema`. If you were to try to parse the parent schema on its own, you would get an exception because the child schema isn't defined. To accommodate this, we can use the *named_schemas* argument to pass a shared dictionary when parsing both of the schemas. The dictionary will get populated with the necessary schema references to make parsing possible. For example:

```
from fastavro import parse_schema

named_schemas = {}
parsed_child = parse_schema(child_schema, named_schemas)
parsed_parent = parse_schema(parent_schema, named_schemas)
```

fullname(*schema: Dict*) → str

Returns the fullname of a schema

Parameters
schema -- Input schema

Example:

```
from fastavro.schema import fullname

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}

fname = fullname(schema)
assert fname == "test.Weather"
```

expand_schema(*schema: str | List | Dict*) → str | List | Dict

Returns a schema where all named types are expanded to their real schema

NOTE: The output of this function produces a schema that can include multiple definitions of the same named type (as per design) which are not valid per the avro specification. Therefore, the output of this should not be passed to the normal *writer*/*reader* functions as it will likely result in an error.

Parameters
schema (*dict*) -- Input schema

Example:

```
from fastavro.schema import expand_schema

original_schema = {
    "name": "MasterSchema",
    "namespace": "com.namespace.master",
    "type": "record",
    "fields": [{
        "name": "field_1",
        "type": {
            "name": "Dependency",
            "namespace": "com.namespace.dependencies",
            "type": "record",
            "fields": [
                {"name": "sub_field_1", "type": "string"}
            ]
        }
    }, {
        "name": "field_2",
        "type": "com.namespace.dependencies.Dependency"
    }]
}

expanded_schema = expand_schema(original_schema)

assert expanded_schema == {
    "name": "com.namespace.master.MasterSchema",
    "type": "record",
    "fields": [{
        "name": "field_1",
        "type": {
            "name": "com.namespace.dependencies.Dependency",
            "type": "record",
            "fields": [
                {"name": "sub_field_1", "type": "string"}
            ]
        }
    }, {
        "name": "field_2",
        "type": {
            "name": "com.namespace.dependencies.Dependency",
            "type": "record",
            "fields": [
                {"name": "sub_field_1", "type": "string"}
            ]
        }
    }]
}
```

load_schema(*schema_path: str*, ***, *repo:* AbstractSchemaRepository | *None = None*, *named_schemas: Dict[str, Dict] | None = None*, *_write_hint: bool = True*, *_injected_schemas: Set[str] | None = None*) → str | List | Dict

Returns a schema loaded from repository.

Will recursively load referenced schemas attempting to load them from same repository, using *schema_path* as

schema name.

If *repo* is not provided, *FlatDictRepository* is used. *FlatDictRepository* will try to load schemas from the same directory assuming files are named with the convention <*full_name*>.*avsc*.

Parameters

- **schema_path** -- Full schema name, or path to schema file if default repo is used.
- **repo** -- Schema repository instance.
- **named_schemas** -- Dictionary of named schemas to their schema definition
- **_write_hint** -- Internal API argument specifying whether or not the __fastavro_parsed marker should be added to the schema
- **_injected_schemas** -- Internal API argument. Set of names that have been injected

Consider the following example with default FlatDictRepository...

namespace.Parent.avsc:

```
{
    "type": "record",
    "name": "Parent",
    "namespace": "namespace",
    "fields": [
        {
            "name": "child",
            "type": "Child"
        }
    ]
}
```

namespace.Child.avsc:

```
{
    "type": "record",
    "namespace": "namespace",
    "name": "Child",
    "fields": []
}
```

Code:

```
from fastavro.schema import load_schema

parsed_schema = load_schema("namespace.Parent.avsc")
```

load_schema_ordered(*ordered_schemas: List[str]*, *, *_write_hint: bool = True*) → str | List | Dict

Returns a schema loaded from a list of schemas.

The list of schemas should be ordered such that any dependencies are listed before any other schemas that use those dependencies. For example, if schema *A* depends on schema *B* and schema B depends on schema *C*, then the list of schemas should be [C, B, A].

Parameters

- **ordered_schemas** -- List of paths to schemas

- **_write_hint** -- Internal API argument specifying whether or not the __fastavro_parsed marker should be added to the schema

Consider the following example...

Parent.avsc:

```
{
    "type": "record",
    "name": "Parent",
    "namespace": "namespace",
    "fields": [
        {
            "name": "child",
            "type": "Child"
        }
    ]
}
```

namespace.Child.avsc:

```
{
    "type": "record",
    "namespace": "namespace",
    "name": "Child",
    "fields": []
}
```

Code:

```
from fastavro.schema import load_schema_ordered

parsed_schema = load_schema_ordered(
    ["path/to/namespace.Child.avsc", "path/to/Parent.avsc"]
)
```

to_parsing_canonical_form(*schema: str | List | Dict*) → str

Returns a string represening the parsing canonical form of the schema.

For more details on the parsing canonical form, see here: https://avro.apache.org/docs/current/spec.html#Parsing+Canonical+Form+for+Schemas

Parameters

schema -- Schema to transform

fingerprint(*parsing_canonical_form: str*, *algorithm: str*) → str

Returns a string represening a fingerprint/hash of the parsing canonical form of a schema.

For more details on the fingerprint, see here: https://avro.apache.org/docs/current/spec.html#schema_fingerprints

Parameters

- **parsing_canonical_form** -- The parsing canonical form of a schema
- **algorithm** -- The hashing algorithm

5.6 fastavro.validation

validate(*datum: Any*, *schema: str | List | Dict*, *field: str = ''*, *raise_errors: bool = True*, *strict: bool = False*, *disable_tuple_notation: bool = False*) → bool

Determine if a python datum is an instance of a schema.

Parameters

- **datum** -- Data being validated
- **schema** -- Schema
- **field** -- Record field being validated
- **raise_errors** -- If true, errors are raised for invalid data. If false, a simple True (valid) or False (invalid) result is returned
- **strict** -- If true, fields without values will raise errors rather than implicitly defaulting to None
- **disable_tuple_notation** -- If set to True, tuples will not be treated as a special case. Therefore, using a tuple to indicate the type of a record will not work

Example:

```
from fastavro.validation import validate
schema = {...}
record = {...}
validate(record, schema)
```

validate_many(*records: Iterable[Any]*, *schema: str | List | Dict*, *raise_errors: bool = True*, *strict: bool = False*, *disable_tuple_notation: bool = False*) → bool

Validate a list of data!

Parameters

- **records** -- List of records to validate
- **schema** -- Schema
- **raise_errors** -- If true, errors are raised for invalid data. If false, a simple True (valid) or False (invalid) result is returned
- **strict** -- If true, fields without values will raise errors rather than implicitly defaulting to None
- **disable_tuple_notation** -- If set to True, tuples will not be treated as a special case. Therefore, using a tuple to indicate the type of a record will not work

Example:

```
from fastavro.validation import validate_many
schema = {...}
records = [{...}, {...}, ...]
validate_many(records, schema)
```

5.7 fastavro.utils

generate_one(*schema: str | List | Dict*) → Any

Returns a single instance of arbitrary data that conforms to the schema.

Parameters

schema -- Schema that data should conform to

Example:

```
from fastavro import schemaless_writer
from fastavro.utils import generate_one

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}

with open('weather.avro', 'wb') as out:
    schemaless_writer(out, schema, generate_one(schema))
```

generate_many(*schema: str | List | Dict*, *count: int*) → Iterator[Any]

A generator that yields arbitrary data that conforms to the schema. It will yield a number of data structures equal to what is given in the count

Parameters

- **schema** -- Schema that data should conform to
- **count** -- Number of objects to generate

Example:

```
from fastavro import writer
from fastavro.utils import generate_many

schema = {
    'doc': 'A weather reading.',
    'name': 'Weather',
    'namespace': 'test',
    'type': 'record',
    'fields': [
        {'name': 'station', 'type': 'string'},
        {'name': 'time', 'type': 'long'},
        {'name': 'temp', 'type': 'int'},
    ],
}
```

(continues on next page)

(continued from previous page)

```
with open('weather.avro', 'wb') as out:
    writer(out, schema, generate_many(schema, 5))
```

anonymize_schema(*schema: str | List | Dict*) → str | List | Dict

Returns an anonymized schema

Parameters
schema -- Schema to anonymize

Example:

```
from fastavro.utils import anonymize_schema

anonymized_schema = anonymize_schema(original_schema)
```

5.8 Logical Types

Fastavro supports the following official logical types:

- decimal
- uuid
- date
- time-millis
- time-micros
- timestamp-millis
- timestamp-micros
- local-timestamp-millis
- local-timestamp-micros

Fastavro is missing support for the following official logical types:

- duration

5.8.1 How to specify logical types in your schemas

The docs say that when you want to make something a logical type, you just need to add a *logicalType* key. Unfortunately, this means that a common confusion is that people will take a pre-existing schema like this:

```
schema = {
    "type": "record",
    "name": "root",
    "fields": [
        {
            "name": "id",
            "type": "string",
        },
    ]
}
```

And then add the uuid logical type like this:

```
schema = {
    "type": "record",
    "name": "root",
    "fields": [
        {
            "name": "id",
            "type": "string",
            "logicalType": "uuid",  # This is the wrong place to add this key
        },
    ]
}
```

However, that adds the *logicalType* key to the *field* schema which is not correct. Instead, we want to group it with the *string* like so:

```
schema = {
    "type": "record",
    "name": "root",
    "fields": [
        {
            "name": "id",
            "type": {
                "type": "string",
                "logicalType": "uuid",  # This is the correct place to add this key
            },
        },
    ]
}
```

5.8.2 Custom Logical Types

The Avro specification defines a handful of logical types that most implementations support. For example, one of the defined logical types is a microsecond precision timestamp. The specification states that this value will get encoded as an avro *long* type.

For the sake of an example, let's say you want to create a new logical type for a microsecond precision timestamp that uses a string as the underlying avro type.

To do this, there are a few functions that need to be defined. First, we need an encoder function that will encode a datetime object as a string. The encoder function is called with two arguments: the data and the schema. So we could define this like so:

```
def encode_datetime_as_string(data, schema):
    return datetime.isoformat(data)

# or

def encode_datetime_as_string(data, *args):
    return datetime.isoformat(data)
```

Then, we need a decoder function that will transform the string back into a datetime object. The decoder function is called with three arguments: the data, the writer schema, and the reader schema. So we could define this like so:

```
def decode_string_as_datetime(data, writer_schema, reader_schema):
    return datetime.fromisoformat(data)

# or

def decode_string_as_datetime(data, *args):
    return datetime.fromisoformat(data)
```

Finally, we need to tell *fastavro* to use these functions. The schema for this custom logical type will use the type *string* and can use whatever name you would like as the *logicalType*. In this example, let's suppose we call the logicalType *datetime2*. To have the library actually use the custom logical type, we use the name of <*avro_type*>-<*logical_type*>, so in this example that name would be *string-datetime2* and then we add those functions like so:

```
fastavro.write.LOGICAL_WRITERS["string-datetime2"] = encode_datetime_as_string
fastavro.read.LOGICAL_READERS["string-datetime2"] = decode_string_as_datetime
```

And you are done. Now if the library comes across a schema with a logical type of *datetime2* and an avro type of *string*, it will use the custom functions. For a complete example, see here:

```
import io
from datetime import datetime

import fastavro
from fastavro import writer, reader


def encode_datetime_as_string(data, *args):
    return datetime.isoformat(data)

def decode_string_as_datetime(data, *args):
    return datetime.fromisoformat(data)

fastavro.write.LOGICAL_WRITERS["string-datetime2"] = encode_datetime_as_string
fastavro.read.LOGICAL_READERS["string-datetime2"] = decode_string_as_datetime


writer_schema = fastavro.parse_schema({
    "type": "record",
    "name": "root",
    "fields": [
        {
            "name": "some_date",
            "type": [
                "null",
                {
                    "type": "string",
                    "logicalType": "datetime2",
                },
            ],
        },
    ]
})
```

(continues on next page)

(continued from previous page)

```
records = [
    {"some_date": datetime.now()}
]

bio = io.BytesIO()

writer(bio, writer_schema, records)

bio.seek(0)

for record in reader(bio):
    print(record)
```

5.9 fastavro command line script

A command line script is installed with the library that can be used to dump the contents of avro file(s) to the standard output.

Usage:

```
usage: fastavro [-h] [--schema] [--codecs] [--version] [-p] [file [file ...]]

iter over avro file, emit records as JSON

positional arguments:
  file          file(s) to parse

optional arguments:
  -h, --help    show this help message and exit
  --schema      dump schema instead of records
  --codecs      print supported codecs
  --version     show program's version number and exit
  -p, --pretty  pretty print json
```

5.9.1 Examples

Read an avro file:

```
$ fastavro weather.avro

{"temp": 0, "station": "011990-99999", "time": -619524000000}
{"temp": 22, "station": "011990-99999", "time": -619506000000}
{"temp": -11, "station": "011990-99999", "time": -619484400000}
{"temp": 111, "station": "012650-99999", "time": -655531200000}
{"temp": 78, "station": "012650-99999", "time": -655509600000}
```

Show the schema:

```
$ fastavro --schema weather.avro
```

(continues on next page)

(continued from previous page)

```
{
 "type": "record",
 "namespace": "test",
 "doc": "A weather reading.",
 "fields": [
  {
   "type": "string",
   "name": "station"
  },
  {
   "type": "long",
   "name": "time"
  },
  {
   "type": "int",
   "name": "temp"
  }
 ],
 "name": "Weather"
}
```

CHAPTER

SIX

FASTAVRO.IO

6.1 fastavro.io

6.1.1 fastavro.io.json_decoder

class AvroJSONDecoder(*fo: IO*)

Decoder for the avro JSON format.

NOTE: All attributes and methods on this class should be considered private.

Parameters

fo -- File-like object to reader from

6.1.2 fastavro.io.json_encoder

class AvroJSONEncoder(*fo: IO*, *, *write_union_type: bool = True*)

Encoder for the avro JSON format.

NOTE: All attributes and methods on this class should be considered private.

Parameters

- **fo** -- Input stream
- **write_union_type** -- Determine whether to write the union type in the json message.

CHAPTER

SEVEN

FASTAVRO.REPOSITORY

7.1 fastavro.repository

7.1.1 fastavro.repository.base

class AbstractSchemaRepository

- genindex
- modindex
- search

INDEX