asn1tools Documentation

Release 0.166.0

Erik Moqvist

Mar 10, 2023

Contents

CHAPTER 1

About

A Python package for ASN.1 parsing, encoding and decoding.

This project is *under development* and does only support a subset of the ASN.1 specification syntax.

Supported codecs:

- Basic Encoding Rules (BER)
- Distinguished Encoding Rules (DER)
- Generic String Encoding Rules (GSER)
- JSON Encoding Rules (JER)
- Basic Octet Encoding Rules (OER)
- Aligned Packed Encoding Rules (PER)
- Unaligned Packed Encoding Rules (UPER)
- XML Encoding Rules (XER)

Miscellaneous features:

- *C* source code generator for OER and UPER (with some limitations).

Project homepage: https://github.com/eerimoq/asn1tools

Documentation: http://asn1tools.readthedocs.org/en/latest

CHAPTER 2

Known limitations

- The `CLASS` keyword (X.681) and its friends are not yet supported.
- Parametrization (X.683) is not yet supported.
- The `EMBEDDED PDV` type is not yet supported.
- The `ANY` and `ANY DEFINED BY` types are not supported. They were removed from the ASN.1 standard 1994.
- `WITH COMPONENT` and `WITH COMPONENTS` constraints are ignored, except for OER `REAL`.
- The `DURATION` type is not yet supported.

CHAPTER 3

Installation

```
pip install asn1tools
```

CHAPTER 4

Example Usage

This is an example ASN.1 specification defining the messages of a fictitious Foo protocol (based on the FooProtocol on Wikipedia).

```
Foo DEFINITIONS ::= BEGIN

    Question ::= SEQUENCE {
        id        INTEGER,
        question  IA5String
    }

    Answer ::= SEQUENCE {
        id        INTEGER,
        answer    BOOLEAN
    }

END
```

4.1 Scripting

Compile the ASN.1 specification, and encode and decode a question using the default codec (BER).

```
>>> import asn1tools
>>> foo = asn1tools.compile_files('tests/files/foo.asn')
>>> encoded = foo.encode('Question', {'id': 1, 'question': 'Is 1+1=3?'})
>>> encoded
bytearray(b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?')
>>> foo.decode('Question', encoded)
{'id': 1, 'question': 'Is 1+1=3?'}
```

The same ASN.1 specification, but using the PER codec.

```
>>> import asn1tools
>>> foo = asn1tools.compile_files('tests/files/foo.asn', 'per')
>>> encoded = foo.encode('Question', {'id': 1, 'question': 'Is 1+1=3?'})
>>> encoded
bytearray(b'\x01\x01\tIs 1+1=3?')
>>> foo.decode('Question', encoded)
{'id': 1, 'question': 'Is 1+1=3?'}
```

See the examples folder for additional examples.

4.2 Command line tool

4.2.1 The shell subcommand

Use the command line shell to convert data between given formats. The default input codec is BER and output codec is GSER (produces human readable text).

```
> asn1tools shell

Welcome to the asn1tools shell!

$ help
Commands:
  compile
  convert
  exit
  help
$ compile tests/files/foo.asn
$ convert Question 300e020101160949732031 2b313d333f
question Question ::= {
    id 1,
    question "Is 1+1=3?"
}
$ compile --output-codec xer tests/files/foo.asn
$ convert Question 300e020101160949732031 2b313d333f
<Question>
    <id>1</id>
    <question>Is 1+1=3?</question>
</Question>
$ compile -o uper tests/files/foo.asn
$ convert Question 300e020101160949732031 2b313d333f
01010993cd03156c5eb37e
$ exit
>
```

4.2.2 The convert subcommand

Convert given encoded Question from BER to GSER (produces human readable text).

```
> asn1tools convert tests/files/foo.asn Question 300e020101160949732031 2b313d333f
question Question ::= {
    id 1,
    question "Is 1+1=3?"
```

(continues on next page)

(continued from previous page)

```
}
>
```

Convert given encoded Question from UPER to XER (xml).

```
> asn1tools convert -i uper -o xer tests/files/foo.asn Question 01010993cd03156c5eb37e
<Question>
    <id>1</id>
    <question>Is 1+1=3?</question>
</Question>
>
```

Convert given encoded Question from UPER to JER (json).

```
> asn1tools convert -i uper -o jer tests/files/foo.asn Question 01010993cd03156c5eb37e
{
    "id": 1,
    "question": "Is 1+1=3?"
}
>
```

Continuously convert encoded Questions read from standard input. Any line that cannot be converted is printed as is, in this example the dates.

```
> cat encoded.txt
2018-02-24 11:22:09
300e0201011609497320312b313d333f
2018-02-24 11:24:15
300e0201021609497320322b323d353f
> cat encoded.txt | asn1tools convert tests/files/foo.asn Question -
2018-02-24 11:22:09
question Question ::= {
    id 1,
    question "Is 1+1=3?"
}
2018-02-24 11:24:15
question Question ::= {
    id 2,
    question "Is 2+2=5?"
}
>
```

4.2.3 The convert subcommand with a cache

Convert given encoded PCCH-Message from UPER to GSER with the `--cache-dir` option set to `my_cache`. Using a cache significantly reduces the command execution time after the first call.

```
> time asn1tools convert --cache-dir my_cache -i uper tests/files/3gpp/rrc_8_6_0.asn␣
↪PCCH-Message 28
pcch-message PCCH-Message ::= {
    message c1 : paging : {
        systemInfoModification true,
        nonCriticalExtension {
        }
    }
```

(continues on next page)

(continued from previous page)

```
}

real    0m2.090s
user    0m1.977s
sys     0m0.032s
> time asn1tools convert --cache-dir my_cache -i uper tests/files/3gpp/rrc_8_6_0.asn␣
↪PCCH-Message 28
pcch-message PCCH-Message ::= {
    message c1 : paging : {
        systemInfoModification true,
        nonCriticalExtension {
        }
    }
}

real    0m0.276s
user    0m0.197s
sys     0m0.026s
>
```

4.2.4 The parse subcommand

Parse given ASN.1 specification and write it as a Python dictionary to given file. Use the created file to convert given encoded Question from BER to GSER (produces human readable text). The conversion is significantly faster than passing .asn-file(s) to the convert subcommand, especially for larger ASN.1 specifications.

```
> asn1tools parse tests/files/foo.asn foo.py
> asn1tools convert foo.py Question 300e020101160949732031 2b313d333f
question Question ::= {
    id 1,
    question "Is 1+1=3?"
}
>
```

4.2.5 The generate C source subcommand

Generate OER or UPER C source code from an ASN.1 specification.

No dynamic memory is used in the generated code. To achieve this all types in the ASN.1 specification must have a known maximum size, i.e. `INTEGER (0..7)`, `OCTET STRING (SIZE(12))`, etc.

Below is an example generating OER C source code from tests/files/c_source/c_source.asn.

```
> asn1tools generate_c_source --namespace oer tests/files/c_source/c_source.asn
Successfully generated oer.h and oer.c.
```

The same as above, but generate UPER C source code instead of OER.

```
> asn1tools generate_c_source --codec uper --namespace uper tests/files/c_source/c_
↪source.asn
Successfully generated uper.h and uper.c.
```

The same as the first example, but also generate fuzz testing C source code for libFuzzer.

```
> asn1tools generate_c_source --namespace oer --generate-fuzzer tests/files/c_source/
↪c_source.asn
Successfully generated oer.h and oer.c.
Successfully generated oer_fuzzer.c and oer_fuzzer.mk.

Run "make -f oer_fuzzer.mk" to build and run the fuzzer. Requires a
recent version of clang.
```

See oer.h, oer.c, uper.h, uper.c, oer_fuzzer.c and oer_fuzzer.mk for the contents of the generated files.

Limitations by design:

- Only the types `BOOLEAN`, `INTEGER`, `NULL`, `OCTET STRING`, `BIT STRING`, `ENUMERATED`, `SEQUENCE`, `SEQUENCE OF`, and `CHOICE` are supported. The OER generator also supports `REAL`.
- All types must have a known maximum size, i.e. `INTEGER (0..7)`, `OCTET STRING (SIZE(12))`.
- `INTEGER` must be 64 bits or less.
- `REAL` must be IEEE 754 binary32 or binary64. binary32 is generated as `float` and binary64 as `double`.
- Recursive types are not supported.

Known limitations:

- Extension additions (`...`) are only supported in the OER generator. See compact_extensions_uper for how to make UPER `CHOICE` and `SEQUENCE` extendable without using `...`.
- Named numbers in `ENUMERATED` are not yet supported.

Other OER and/or UPER C code generators:

- https://github.com/vlm/asn1c
- https://github.com/ttsiodras/asn1scc

See the benchmark example for a comparison of *asn1c*, *asn1scc* and *asn1tools*.

CHAPTER 5

Contributing

1. Fork the repository.
2. Install prerequisites.

```
pip install -r requirements.txt
```

3. Implement the new feature or bug fix.
4. Implement test case(s) to ensure that future changes do not break legacy.
5. Run the tests.

```
make test
```

6. Create a pull request.

CHAPTER 6

Specifications

ASN.1 specifications released by ITU and IETF.

6.1 General

- X.680: Specification of basic notation
- X.681: Information object specification
- X.682: Constraint specification
- X.683: Parameterization of ASN.1 specifications

6.2 Encodings

- X.690: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)
- X.691: Specification of Packed Encoding Rules (PER)
- X.693: XML Encoding Rules (XER)
- X.696: Specification of Octet Encoding Rules (OER)
- RFC 3641: Generic String Encoding Rules (GSER) for ASN.1
- Overview of the JSON Encoding Rules (JER)

CHAPTER 7

Basic Usage

asn1tools.**compile_files**(*filenames*, *codec='ber'*, *any_defined_by_choices=None*, *encoding='utf-8'*, *cache_dir=None*, *numeric_enums=False*)

Compile given ASN.1 specification file(s) and return a *Specification* object that can be used to encode and decode data structures with given codec *codec*. *codec* may be one of `'ber'`, `'der'`, `'gser'`, `'jer'`, `oer`, `'per'`, `'uper'` and `'xer'`.

encoding is the text encoding. This argument is passed to the built-in function *open()*.

cache_dir specifies the compiled files cache location in the file system. Give as `None` to disable the cache. By default the cache is disabled. The cache key is the concatenated contents of given files and the codec name. Using a cache will significantly reduce the compile time when recompiling the same files. The cache directory is automatically created if it does not exist. Remove the cache directory *cache_dir* to clear the cache.

Give *numeric_enums* as `True` for numeric enumeration values instead of strings.

```
>>> foo = asn1tools.compile_files('foo.asn')
```

Give *cache_dir* as a string to use a cache.

```
>>> foo = asn1tools.compile_files('foo.asn', cache_dir='my_cache')
```

class asn1tools.compiler.**Specification**(*modules*, *decode_length*, *type_checkers*, *constraints_checkers*)

This class is used to encode and decode ASN.1 types found in an ASN.1 specification.

Instances of this class are created by the factory functions *compile_files()*, *compile_string()* and *compile_dict()*.

types

A dictionary of all unique types in the specification. Types found in two or more modules are not part of this dictionary.

```
>>> question = foo.types['Question']
>>> question
Sequence(Question, [Integer(id), IA5String(question)])
```

(continues on next page)

(continued from previous page)

```
>>> question.encode({'id': 1, 'question': 'Is 1+1=3?'})
b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?'
```

modules
A dictionary of all modules in the specification. Unlike *types*, this attribute contains every type, even if the type name was found in two or more modules.

```
>>> question = foo.modules['Foo']['Question']
>>> question
Sequence(Question, [Integer(id), IA5String(question)])
>>> question.encode({'id': 1, 'question': 'Is 1+1=3?'})
b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?'
```

encode(*name*, *data*, *check_types=True*, *check_constraints=False*, ***kwargs*)
Encode given dictionary *data* as given type *name* and return the encoded data as a bytes object.

If *check_types* is `True` all objects in *data* are checked against the expected Python type for its ASN.1 type. Set *check_types* to `False` to minimize the runtime overhead, but instead get less informative error messages.

See *Types* for a mapping table from ASN.1 types to Python types.

If *check_constraints* is `True` all objects in *data* are checked against their ASN.1 type constraints. A ConstraintsError exception is raised if the constraints are not fulfilled. Set *check_constraints* to `False` to skip the constraints check and minimize the runtime overhead, but instead get less informative error messages and allow encoding of values not fulfilling the constraints.

```
>>> foo.encode('Question', {'id': 1, 'question': 'Is 1+1=3?'})
b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?'
```

decode(*name*, *data*, *check_constraints=False*)
Decode given bytes object *data* as given type *name* and return the decoded data as a dictionary.

If *check_constraints* is `True` all objects in *data* are checked against their ASN.1 type constraints. A ConstraintsError exception is raised if the constraints are not fulfilled. Set *check_constraints* to `False` to skip the constraints check and minimize the runtime overhead, but instead allow decoding of values not fulfilling the constraints.

```
>>> foo.decode('Question', b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?')
{'id': 1, 'question': 'Is 1+1=3?'}
```

decode_with_length(*name*, *data*, *check_constraints=False*)
Same as *decode()*, but also returns the byte length of the decoded data.

Use to get the length of indefinite-length BER encoded data.

```
>>> foo.decode_with_length('Question',
                           b'0\x0e\x02\x01\x01\x16\x09Is 1+1=3?')
({'id': 1, 'question': 'Is 1+1=3?'}, 16)
```

decode_length(*data*)
Decode the length of given data *data*. Returns None if not enough data was given to decode the length.

This method only works for BER and DER codecs with definite length in the first data encoding. Other codecs and combinations lacks length information in the data.

```
>>> foo.decode_length(b'\x30\x0e\x02\x01\x01')
16
```

CHAPTER 8

Types

ASN.1 types are mapped to Python 3 types as shown in the table below. In Python 2, INTEGER may be `long` and all string types are `unicode`.

ASN.1 type	Python type	Example
BOOLEAN	`bool`	`True`
INTEGER	`int`	`87`
REAL	`float`	`33.12`
NULL	–	`None`
BIT STRING	`tuple(bytes, int)`	`(b'\x50', 4)`
OCTET STRING	`bytes`	`b'\x44\x1e\xff'`
OBJECT IDENTIFIER	`str`	`'1.33.2'`
ENUMERATED	`str` or `int` (1)	`'one'` or `1`
SEQUENCE	`dict`	`{'a': 52, 'b': 1}`
SEQUENCE OF	`list`	`[1, 3]`
SET	`dict`	`{'foo': 'bar'}`
SET OF	`list`	`[3, 0, 7]`
CHOICE	`tuple(str, object)`	`('a', 5)`
UTF8String	`str`	`'hello'`
NumericString	`str`	`'234359'`
PrintableString	`str`	`'goo'`
IA5String	`str`	`'name'`
VisibleString	`str`	`'gle'`
GeneralString	`str`	`'abc'`
BMPString	`str`	`'ko'`
GraphicString	`str`	`'a b'`
TeletexString	`str`	`'ßø'`
UniversalString	`str`	`'ääö'`
UTCTime	`datetime.datetime`	`datetime(2018, 6, 11, 11, 4, 59)`
GeneralizedTime	`datetime.datetime`	`datetime(2018, 1, 31, 5, 0, 47)`
DATE	`datetime.date`	`date(1985, 4, 12)`
TIME-OF-DAY	`datetime.time`	`time(15, 27, 46)`
DATE-TIME	`datetime.datetime`	`datetime(1985, 4, 12, 15, 27, 30)`
ObjectDescriptor	–	–

(1) Compile with `numeric_enums=True` for numeric enumeration values instead of strings.

CHAPTER 9

Advanced Usage

asn1tools.**compile_string**(*string*, *codec='ber'*, *any_defined_by_choices=None*, *numeric_enums=False*)

Compile given ASN.1 specification string and return a *Specification* object that can be used to encode and decode data structures with given codec *codec*. *codec* may be one of `'ber'`, `'der'`, `'gser'`, `'jer'`, `oer`, `'per'`, `'uper'` and `'xer'`.

Give *numeric_enums* as `True` for numeric enumeration values instead of strings.

```
>>> with open('foo.asn') as fin:
...     foo = asn1tools.compile_string(fin.read())
```

asn1tools.**compile_dict**(*specification*, *codec='ber'*, *any_defined_by_choices=None*, *numeric_enums=False*)

Compile given ASN.1 specification dictionary and return a *Specification* object that can be used to encode and decode data structures with given codec *codec*. *codec* may be one of `'ber'`, `'der'`, `'gser'`, `'jer'`, `oer`, `'per'`, `'uper'` and `'xer'`.

Give *numeric_enums* as `True` for numeric enumeration values instead of strings.

```
>>> foo = asn1tools.compile_dict(asn1tools.parse_files('foo.asn'))
```

asn1tools.**parse_files**(*filenames*, *encoding='utf-8'*)

Parse given ASN.1 specification file(s) and return a dictionary of its/their contents.

The dictionary can later be compiled with *compile_dict()*.

encoding is the text encoding. This argument is passed to the built-in function *open()*.

```
>>> foo = asn1tools.parse_files('foo.asn')
```

asn1tools.**parse_string**(*string*)

Parse given ASN.1 specification string and return a dictionary of its contents.

The dictionary can later be compiled with *compile_dict()*.

```
>>> with open('foo.asn') as fin:
...     foo = asn1tools.parse_string(fin.read())
```

Index