

---

# **python-ndn**

***Release 0.3a1-4***

**Xinyu Ma**

**Jul 18, 2021**



# CONTENTS

<b>1</b>	<b>Table Of Contents</b>	<b>1</b>
1.1	python-ndn	1
1.2	Installation	1
1.2.1	Instructions for developer	1
1.3	ndn.app package	2
1.3.1	Introduction	2
1.3.2	Keyword Arguments	2
1.3.3	Reference	3
1.4	ndn.encoding package	6
1.4.1	Introduction	6
1.4.2	FormalName and NonStrictName	7
1.4.3	Customized TLV Models	7
1.4.4	Reference	7
1.5	ndn.security package	20
1.5.1	Introduction	20
1.5.2	Signer	21
1.5.3	Validator	21
1.5.4	Keychain	21
1.6	ndn.schema package	25
1.6.1	Introduction	25
1.6.2	Examples	25
1.6.3	Reference	27
1.7	Miscellaneous packages	36
1.7.1	ndn.types package	36
1.7.2	ndn.utils package	37
1.8	Examples	37
1.8.1	Basic Applications	37
1.8.2	Customized TLV Models	39
1.9	Contribute and Support	41
1.10	Future plans	41
1.11	Authors	42
1.12	Changelog	42
1.12.1	0.3a1-3 (2021-05-22)	42
1.12.2	0.3a1-2 (2021-04-29)	42
1.12.3	0.3a1-1 (2021-01-31)	42
1.12.4	0.3a1 (2020-09-24)	42
1.12.5	0.2b2-2 (2020-05-26)	42
1.12.6	0.2b2-1 (2020-03-23)	43
1.12.7	0.2b2 (2020-02-18)	43
1.12.8	0.2b1 (2019-11-20)	43

<b>2 Indices and tables</b>	<b>45</b>
<b>Python Module Index</b>	<b>47</b>
<b>Index</b>	<b>49</b>

## TABLE OF CONTENTS

### 1.1 python-ndn

A Named Data Networking client library with AsyncIO support in Python 3.

It supports Python  $\geq 3.7$  and PyPy3  $\geq 7.1.1$ .

Please see our [documentation](#) if you have any issues.

### 1.2 Installation

Install the latest release with pip:

```
$ pip install python-ndn
```

Install the latest development version:

```
$ pip install -U git+https://github.com/named-data/python-ndn.git
```

#### 1.2.1 Instructions for developer

For development, pipenv is recommended:

```
$ pipenv install --dev
```

To setup a traditional python3 virtual environment with editable installation:

```
python3 -m venv venv  
. venv/bin/activate  
pip3 install -e ".[dev]"
```

Run all tests:

```
pipenv run test
```

Run static analysis:

```
pipenv run make lint
```

Please use python 3.8+ to generate the documentation.

```
pip3 install Sphinx sphinx-autodoc-typehints readthedocs-sphinx-ext \
    sphinx-rtd-theme pycryptodomex pygtrie

cd docs && make html
open _build/html/index.html
```

VSCode users can also use the development container obtained from the *.devcontainer* folder.

## 1.3 ndn.app package

### 1.3.1 Introduction

The *ndn.app* package contains the class *NDNApp*, which connects an NDN application and an NFD node.

*NDNApp* provides the functionalities similar to application Face in *ndn-cxx*, which include:

- Establish a connection to an NFD node.
- Express Interests and handle the Data coming back.
- Register and unregister a route with an Interest handling function.

### 1.3.2 Keyword Arguments

Some functions which create a Interest or Data packet accept a *kwargs*, which can be used to support diversity in arguments provided to create a packet.

#### MetaInfo

These arguments are used to fill in the MetaInfo field of a Data packet.

- **meta\_info** (*MetaInfo*) - the MetaInfo field of Data. All other related parameters will be ignored.
- **content\_type** (*int*) - *ContentType*. *ContentType.BLOB* by default.
- **freshness\_period** (*int*) - FreshnessPeriod in milliseconds. *None* by default.
- **final\_block\_id** (*BinaryStr*) - FinalBlockId. It should be an encoded *Component*. *None* by default.

#### InterestParameters

These arguments are used to fill in fields of an Interest packet.

- **interest\_param** (*InterestParam*) - a dataclass containing all parameters. All other related parameters will be ignored.
- **can\_be\_prefix** (*bool*) - *CanBePrefix*. *False* by default.
- **must\_be\_fresh** (*bool*) - *MustBeFresh*. *False* by default.
- **nonce** (*int*) - *Nonce*. A random number will be generated by default. To omit *Nonce*, please explicitly pass *None* to this argument.

- **lifetime** (*int*) - InterestLifetime in milliseconds. 4000 by default.

**Warning:** On Windows, a too small number may cause a memory failure of the NameTrie. Currently,  $\geq 10$  is safe.

- **hop\_limit** (*int*) - HopLimit. None by default.
- **forwarding\_hint** (*int*) - see *InterestParam*.

## Signature

These arguments are used to decide how the Interest or Data packet is signed and by which Signer. Supported arguments are different with each Keychain. Only those supported by the default Keychain are listed here. If there is a conflict, the earlier an argument is listed the higher priority it has.

---

**Note:** Only Interests with ApplicationParameters are signed. `b''` can be used if that field is not needed by the application.

---

- **signer** (*Signer*) - the Signer used to sign this packet. All other related parameters will be ignored. The Keychain will not be used.
- **no\_signature** (*bool*) - not signed. Not recommended.
- **digest\_sha256** (*bool*) - using SHA-256 digest to protect integrity only. `False` by default.
- **key** - using the specified Key to sign this packet. Either a Key object or the *NonStrictName* of a Key is acceptable.
- **identity** - using the default Key of the specified Identity to sign this packet. Either an Identity object or the *NonStrictName* of an Identity is acceptable. The default Identity will be used if all of the above arguments are omitted.

### 1.3.3 Reference

**class** `ndn.app.NDNApp`(*face=None, keychain=None*)  
An NDN application.

#### Variables

- **face** – the Face used to connection to a NFD node.
- **keychain** – the Keychain to store Identities and Keys, providing Signers.
- **int\_validator** – the default validator for Interest packets.
- **data\_validator** – the default validator for Data packets.

**express\_interest**(*name, app\_param=None, validator=None, need\_raw\_packet=False, \*\*kwargs*)  
Express an Interest packet.

The Interest packet is sent immediately and a coroutine used to get the result is returned. Awaiting on what is returned will block until the Data is received and return that Data. An exception is raised if unable to receive the Data.

#### Parameters

- **name** (*NonStrictName*) – the Name.

- **app\_param** (Optional[*BinaryStr*]) – the ApplicationParameters.
- **validator** (Optional[*Validator*]) – the Validator used to verify the Data received.
- **need\_raw\_packet** (*bool*) – if True, return the raw Data packet with TL.
- **kwargs** – *Keyword Arguments*.

**Returns** A tuple of (Name, MetaInfo, Content) after `await`. If `need_raw_packet` is True, return a tuple (Name, MetaInfo, Content, RawPacket).

**Return type** Coroutine[Any, None, Tuple[*FormalName*, *MetaInfo*, Optional[*BinaryStr*]]]

The following exception is raised by `express_interest`:

**Raises** *NetworkError* – the face to NFD is down before sending this Interest.

The following exceptions are raised by the coroutine returned:

**Raises**

- *InterestNack* – an NetworkNack is received.
- *InterestTimeout* – time out.
- *ValidationFailure* – unable to validate the Data packet.
- *InterestCanceled* – the face to NFD is shut down after sending this Interest.

**async main\_loop**(*after\_start=None*)

The main loop of NDNApp.

**Parameters** **after\_start** (Optional[Awaitable]) – the coroutine to start after connection to NFD is established.

**Return type** bool

**Returns** True if the connection is shutdown not by Ctrl+C. For example, manually or by the other side.

**prepare\_data**(*name*, *content=None*, *\*\*kwargs*)

Prepare a Data packet by generating, encoding and signing it.

**Parameters**

- **name** (*NonStrictName*) – the Name.
- **content** (Optional[*BinaryStr*]) – the Content.
- **kwargs** – *Keyword Arguments*.

**Returns** TLV encoded Data packet.

**put\_data**(*name*, *content=None*, *\*\*kwargs*)

Publish a Data packet.

**Parameters**

- **name** (*NonStrictName*) – the Name.
- **content** (Optional[*BinaryStr*]) – the Content.
- **kwargs** – *Keyword Arguments*.

**Returns** TLV encoded Data packet.

**put\_raw\_packet**(*data*)

Send a raw Data packet.



**Parameters** `data` (*BinaryStr*) – TLV encoded Data packet.

**Raises** *NetworkError* – the face to NFD is down.

**async register**(*name*, *func*, *validator=None*, *need\_raw\_packet=False*, *need\_sig\_ptrs=False*)

Register a route for a specific prefix dynamically.

#### Parameters

- **name** (*NonStrictName*) – the Name prefix for this route.
- **func** (Optional[Callable[[*FormalName*, *InterestParam*, Optional[*BinaryStr*]], None]]) – the onInterest function for the specified route. If *None*, the NDNApp will only send the register command to forwarder, without setting any callback function.
- **validator** (Optional[*Validator*]) – the Validator used to validate coming Interests.
- **need\_raw\_packet** (*bool*) – if True, pass the raw Interest packet to the callback as a keyword argument *raw\_packet*.
- **need\_sig\_ptrs** (*bool*) – if True, pass the Signature pointers to the callback as a keyword argument *sig\_ptrs*.

**Return type** *bool*

**Returns** True if the registration succeeded.

#### Raises

- **ValueError** – the prefix is already registered.
- *NetworkError* – the face to NFD is down now.

**route**(*name*, *validator=None*, *need\_raw\_packet=False*, *need\_sig\_ptrs=False*)

A decorator used to register a permanent route for a specific prefix.

This function is non-blocking and can be called at any time. If it is called before connecting to NFD, NDNApp will remember this route and automatically register it every time when a connection is established. Failure in registering this route to NFD will be ignored.

The decorated function should accept 3 arguments: Name, Interest parameters and ApplicationParameters.

#### Parameters

- **name** (*NonStrictName*) – the Name prefix for this route.
- **validator** (Optional[*Validator*]) – the Validator used to validate coming Interests. An Interest without ApplicationParameters and SignatureInfo will be considered valid without calling validator. Interests with malformed ParametersSha256DigestComponent will be dropped before going into the validator. Otherwise NDNApp will try to validate the Interest with the validator. Interests which fail to be validated will be dropped without raising any exception.
- **need\_raw\_packet** (*bool*) – if True, pass the raw Interest packet to the callback as a keyword argument *raw\_packet*.
- **need\_sig\_ptrs** (*bool*) – if True, pass the Signature pointers to the callback as a keyword argument *sig\_ptrs*.

#### Examples

```
app = NDNApp()

@app.route('/example/rpc')
```

(continues on next page)

(continued from previous page)

```
def on_interest(name: FormalName, param: InterestParam, app_param):
    pass
```

**run\_forever**(*after\_start=None*)

A non-async wrapper of `main_loop()`.

**Parameters** `after_start` (Optional[Awaitable]) – the coroutine to start after connection to NFD is established.

**Examples**

```
app = NDNApp()

if __name__ == '__main__':
    app.run_forever(after_start=main())
```

**set\_interest\_filter**(*name, func, validator=None, need\_raw\_packet=False, need\_sig\_ptrs=False*)

Set the callback function for an Interest prefix without sending a register command to the forwarder.

---

**Note:** All callbacks registered by `set_interest_filter` are removed when disconnected from the the forwarder, and will not be added back after reconnection. This behaviour is the same as `register`. Therefore, it is strongly recommended to use `route` for static routes.

---

**shutdown**()

Manually shutdown the face to NFD.

**async unregister**(*name*)

Unregister a route for a specific prefix.

**Parameters** `name` (*NonStrictName*) – the Name prefix.

**Return type** bool

**unset\_interest\_filter**(*name*)

Remove the callback function for an Interest prefix without sending an unregister command.

---

**Note:** `unregister` will only remove the callback if the callback's name matches exactly the route's name. This is because there may be one route whose name is the prefix of another. To avoid cancelling unexpected routes, neither `unregister` nor `unset_interest_filter` behaves in a cascading manner. Please remove callbacks manually.

---

## 1.4 ndn.encoding package

### 1.4.1 Introduction

The `ndn.encoding` package contains classes and functions that help to encode and decode NDN Name, NameComponent, Data and Interest.

There are three parts of this package:

1. **TLV elements:** process TLV variables, Names and NameComponents.

2. **TlvModel**: design a general way to describe a TLV format. A TLV object can be described with a class derived from *TlvModel*, with members of type *Field*.
3. **NDN Packet Format v0.3**: functions used to encode and parse Interest and Data packets in *NDN Packet Format Spec 0.3*.

## 1.4.2 FormalName and NonStrictName

To increase the flexibility, API in python-ndn accepts Name arguments in a wide range of formats, i.e. *NonStrictName*, but returns an unified form, *FormalName*.

A Component is a NameComponent encoded in TLV format.

```
component = b'\x08\x09component'
```

A *FormalName* is a list of encoded Components.

```
formal_name = [bytearray(b'\x08\x06formal'), b'\x08\x04name']
```

A *NonStrictName* is any of below:

- A URI string.

```
casual_name_1 = "/non-strict/8=name"
```

- A list or iterator of Components, in the form of either encoded TLV or URI string.

```
casual_name_2 = [bytearray(b'\x08\x0anon-strict'), 'name']
casual_name_3 = (f'{x}' for x in range(3))
```

- An encoded Name of type bytes, bytearray or memoryview.

```
casual_name_4 = b'\x07\x12\x08\x0anon-strict\x08\x04name'
```

## 1.4.3 Customized TLV Models

See *Customized TLV Models*

## 1.4.4 Reference

### TLV Variables

#### ndn.encoding.tlv\_type.BinaryStr

A binary string is any of bytes, bytearray, memoryview.

alias of Union[bytes, bytearray, memoryview]

#### ndn.encoding.tlv\_type.FormalName

A FormalName is a list of encoded Components.

alias of List[Union[bytes, bytearray, memoryview]]

#### ndn.encoding.tlv\_type.NonStrictName

A NonStrictName is any of below:

- A URI string.

- A list or iterator of Components, in the form of either encoded TLV or URI string.
- An encoded Name of type bytes, bytearray or memoryview.

See also *FormalName and NonStrictName*

alias of Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]

`ndn.encoding.tlv_type.VarBinaryStr`

A variant binary string is a bytearray or a non-readonly memoryview.

alias of Union[bytearray, memoryview]

`ndn.encoding.tlv_type.is_binary_str(var)`

Check whether var is of type BinaryStr.

**Parameters** `var` – The variable to check.

**Returns** True if var is a *BinaryStr*.

`ndn.encoding.tlv_var.get_tl_num_size(val)`

Calculate the length of a TL variable.

**Parameters** `val` (int) – an integer standing for Type or Length.

**Return type** int

**Returns** The length of var.

`ndn.encoding.tlv_var.pack_uint_bytes(val)`

Pack an non-negative integer value into bytes

**Parameters** `val` (int) – the integer.

**Return type** bytes

**Returns** the buffer.

`ndn.encoding.tlv_var.parse_and_check_tl(wire, expected_type)`

Parse Type and Length, and then check:

- If the Type equals *expected\_type*;
- If the Length equals the length of *wire*.

**Parameters**

- **wire** (Union[bytes, bytearray, memoryview]) – the TLV encoded wire.
- **expected\_type** (int) – expected Type.

**Return type** memoryview

**Returns** a pointer to the memory of Value.

`ndn.encoding.tlv_var.parse_tl_num(buf, offset=0)`

Parse a Type or Length variable from a buffer.

**Parameters**

- **buf** (Union[bytes, bytearray, memoryview]) – the buffer.
- **offset** (int) – the starting offset.

**Return type** (<class 'int'>, <class 'int'>)

**Returns** a pair (value, size parsed).

`async ndn.encoding.tlv_var.read_tl_num_from_stream(reader, bio)`

Read a Type or Length variable from a StreamReader.

#### Parameters

- **reader** (StreamReader) – the StreamReader.
- **bio** (BytesIO) – the BytesIO to write whatever is read from the stream.

#### Return type int

**Returns** the value read.

`ndn.encoding.tlv_var.write_tl_num(val, buf, offset=0)`

Write a Type or Length value into a buffer.

#### Parameters

- **val** (int) – the value.
- **buf** (Union[bytearray, memoryview]) – the buffer.
- **offset** (int) – the starting offset.

#### Return type int

**Returns** the encoded length.

## Name and Component

### `ndn.encoding.name.Component` module

Component module is a collection of functions processing NDN NameComponents. In python-ndn, a NameComponent is always encoded in TLV form, of type bytes, bytearray or memoryview.

The types of NameComponent follows [Name Component Assignment policy](#). Type constants are following

Type	Description
TYPE_INVALID	Invalid name component type
TYPE_GENERIC	Implicit SHA-256 digest component
TYPE_IMPLICIT_SHA256	SHA-256 digest of Interest Parameters
TYPE_PARAMETERS_SHA256	Generic name component
TYPE_KEYWORD	Well-known keyword
TYPE_SEGMENT	Segment number
TYPE_BYTE_OFFSET	Byte offset
TYPE_VERSION	Version number
TYPE_TIMESTAMP	Unix timestamp in microseconds
TYPE_SEQUENCE_NUM	Sequence number

```
ndn.encoding.name.Component.CHARSET = {'%', '-', '.', '0', '1', '2', '3', '4', '5', '6',
'7', '8', '9', '=', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '_', 'a', 'b', 'c', 'd', 'e',
'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '~'}
```

The character set for NameComponent, which is unreserved characters + {'=', '%'}.

`ndn.encoding.name.Component.escape_str(val)`

Escape a string to a legal URI string. Any characters not in the [CHARSET](#) will be converted into percent-hexadecimal encoding. '%' itself will not be escaped. For hex digits, lowercase is used.

**Parameters** `val` (str) – the string to escape.

**Return type** str

**Returns** the URI string.

**Examples**

```
>>> from ndn.encoding.name import Component
>>> Component.escape_str('Kraus B6lter')
'Kraus%20B%C3%B6lter'
```

```
>>> Component.escape_str('all:%0a\tgcc -o a.out')
'all%3A%0a%09gcc%20-o%20a.out'
```

`ndn.encoding.name.Component.from_byte_offset(offset)`

Construct a Component from a byte offset.

**Parameters** `offset` (int) – the byte offset.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_bytes(val, typ=8)`

Construct a Component from bytes by adding a type and length.

**Parameters**

- `val` (Union[bytes, bytearray, memoryview]) – the value of the component.
- `typ` (int) – the type of the component. TYPE\_GENERIC by default.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_hex(val, typ=8)`

Construct a Component from hex string.

**Parameters**

- `val` (str) – a hexadecimal string.
- `typ` (int) – the type of the component. TYPE\_GENERIC by default.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_number(val, typ)`

Construct a Component from an integer.

**Parameters**

- `val` (int) – the integer.
- `typ` (int) – the type of the component.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_segment(segment)`

Construct a Component from an segment number.

**Parameters** `segment` (int) – the segment number.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_sequence_num(seq_num)`

Construct a Component from a sequence number.

**Parameters** `seq_num` (int) – the sequence number.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.from_str(val)`

Construct a Component from URI string.

**Parameters** `val` (str) – URI string. All characters should be from [CHARSET](#), otherwise it would raise a `ValueError`.

---

**Note:** Additional periods are not allowed here. To create a zero-size Component, just pass an empty string '' in.

---

**Return type** bytearray

**Returns** the component.

**Raises** `ValueError` – the string is not a legal URI.

`ndn.encoding.name.Component.from_timestamp(timestamp)`

Construct a Component from a timestamp number.

**Parameters** `timestamp` (int) – the timestamp

**Return type** bytearray

**Returns** the component.

**Examples**

```
>>> from ndn.encoding.name import Component
>>> from ndn.utils import timestamp
>>> Component.to_str(Component.from_timestamp(timestamp()))
'36=%00%00%01nH.%A7%90'
```

`ndn.encoding.name.Component.from_version(version)`

Construct a Component from a version number.

**Parameters** `version` (int) – the version number.

**Return type** bytearray

**Returns** the component.

`ndn.encoding.name.Component.get_type(component)`

Get the type from a Component.

**Parameters** `component` (Union[bytes, bytearray, memoryview]) – the component.

**Return type** int

**Returns** the type.

`ndn.encoding.name.Component.get_value(component)`

Get the value from a Component, in the form of memoryview.

**Parameters** `component` (Union[bytes, bytearray, memoryview]) – the component.

**Return type** memoryview

**Returns** the value.

`ndn.encoding.name.Component.to_number(component)`

Take the number encoded in the component out.

**Parameters** `component` (Union[bytes, bytearray, memoryview]) – the component.

**Return type** int

**Returns** an integer, which is the value of the component.

`ndn.encoding.name.Component.to_str(component)`

Convert a Component into a URI string. Returns an empty string '' for a 0-size Component.

**Parameters** `component` (Union[bytes, bytearray, memoryview]) – the component.

**Return type** str

**Returns** a URI string.

## `ndn.encoding.name.Name` module

Name module is a collection of functions processing NDN Names.

`ndn.encoding.name.Name.TYPE_NAME = 7`

The TLV type of NDN Name.

`ndn.encoding.name.Name.from_bytes(buf)`

Decode the Name from its TLV encoded form.

**Parameters** `buf` (Union[bytes, bytearray, memoryview]) – encoded Name.

**Returns** Decoded Name.

**Return type** *FormalName*

**Raises** **ValueError** – if the Type is not `TYPE_NAME`.

`ndn.encoding.name.Name.from_str(val)`

Construct a Name from a URI string.

**Parameters** `val` (str) – URI string. Character out of `ndn.encoding.name.Component.CHARSET` will be escaped automatically. Leading and tailing '/' will be removed.

---

**Note:** Additional periods are not allowed here. To create a zero-size Component, use two slashes to surround it. Also, there should be no scheme identifier and authority component in the URI.

---

**Return type** List[bytearray]

**Returns** *FormalName*.

**Examples**

```
>>> from ndn.encoding.name import Name
>>> Name.from_str("example/name")
[bytearray(b'\x08\x07example'), bytearray(b'\x08\x04name')]
```



```
>>> Name.from_str("/a//32=b/")
[bytearray(b'\x08\x01a'), bytearray(b'\x08\x00'), bytearray(b'\x20\x01b
↪')] ]
```

```
>>> Name.from_str('/a../b')
[bytearray(b'\x08\x01a'), bytearray(b'\x08\x02..'), bytearray(b'\x08\
↪x01b')] ]
```

`ndn.encoding.name.Name.is_prefix(lhs, rhs)`

Test if a Name is a prefix of another Name.

**Parameters**

- **lhs** (*NonStrictName*) – prefix to be tested.
- **rhs** (*NonStrictName*) – full name to test on.

**Return type** bool

**Returns** True if lhs is a prefix of rhs.

`ndn.encoding.name.Name.normalize(name)`

Convert a NonStrictName to a FormalName. If name is a binary string, decode it. If name is a str, encode it into FormalName. If name is a list, encode all str elements into Components.

**Parameters** **name** (*NonStrictName*) – the NonStrictName.

**Returns** the FormalName. It may be a swallow copy of name.

**Return type** *FormalName*

**Raises** **TypeError** – if the name or one of its element has a unrecognized type.

**Examples**

```
>>> from ndn.encoding.name import Name
>>> Name.normalize(f'{i}' for i in range(3))
[bytearray(b'\x08\x010'), bytearray(b'\x08\x011'), bytearray(b'\x08\x012
↪')] ]
```

```
>>> Name.normalize(['', b'\x01\x00'])
[bytearray(b'\x08\x08\xd0\x90\xd0\xbb\xd0\xb5\xd0\xba'), b'\x01\x00']
```

`ndn.encoding.name.Name.to_bytes(name)`

Encode a Name via TLV encoding.

**Parameters** **name** (*NonStrictName*) – Name to encode.

**Return type** bytes

**Returns** Encoded Name.

`ndn.encoding.name.Name.to_str(name)`

Convert an NDN Name to a URI string.

**Parameters** **name** (*NonStrictName*) – the input NDN Name.

**Return type** str

**Returns** the URI.

**Examples**

```
>>> from ndn.encoding.name import Name
>>> Name.to_str('')
'/%CE%A3%CF%80%CF%85%CF%81%CE%AF%CE%B4%CF%89%CE%BD'
```

## TLV Model

### **exception** ndn.encoding.tlv\_model.DecodeError

Raised when there is a critical field (Type is odd) that is unrecognized, redundant or out-of-order.

### **exception** ndn.encoding.tlv\_model.IncludeBaseError

Raised when IncludeBase is used to include a non-base class.

### **class** ndn.encoding.tlv\_model.IncludeBase(*base*)

Include all fields from a base class.

### **class** ndn.encoding.tlv\_model.Field(*type\_num, default=None*)

Field of *TlvModel*. A field with value None will be omitted in encoding TLV. There is no required field in a *TlvModel*, i.e. any Field can be None.

#### Variables

- **name** (*str*) – The name of the field
- **type\_num** (*int*) – The Type number used in TLV encoding
- **default** – The default value used for parsing and encoding.
  - If this field is absent during parsing, **default** is used to fill in this field.
  - If this field is not explicitly assigned to None before encoding, **default** is used.

#### **\_\_get\_\_**(*instance, owner*)

Get the value of this field in a specific instance. Simply call *get\_value()* if *instance* is not None.

#### Parameters

- **instance** – the instance that this field is being accessed through.
- **owner** – the owner class of this field.

**Returns** the value of this field.

#### **\_\_set\_\_**(*instance, value*)

Set the value of this field.

#### Parameters

- **instance** – the instance whose field is being set.
- **value** – the new value.

### **abstract** **encode\_into**(*val, markers, wire, offset*)

Encode this field into wire. Must be called after *encoded\_length()*.

#### Parameters

- **val** – value of this field
- **markers** (dict) – encoding marker variables
- **wire** (Union[bytearray, memoryview]) – buffer to encode
- **offset** (int) – offset of this field in wire

**Return type** int

**Returns** encoded length with TL. It is expected to be the same as `encoded_length()` returns.

**abstract encoded\_length**(*val, markers*)

Preprocess value and get encoded length of this field. The function may use `markers[f'{self.name}##encoded_length']` to store the length with TL. Other marker variables starting with `f'{self.name}##'` may also be used. Generally, marker variables are only used to store temporary values and avoid duplicated calculation. One field should not access to another field's marker by its name.

This function may also use other marker variables. However, in that case, this field must be unique in a `TlvModel`. Usage of marker variables should follow the name convention defined by specific `TlvModel`.

#### Parameters

- **val** – value of this field
- **markers** (dict) – encoding marker variables

#### Return type int

**Returns** encoded length with TL. It is expected as the exact length when encoding this field. The only exception is `SignatureValueField` (invisible to application developer).

**get\_value**(*instance*)

Get the value of this field in a specific instance. Most fields use `instance.__dict__` to access the value.

**Parameters** **instance** – the instance that this field is being accessed through.

**Returns** the value of this field.

**abstract parse\_from**(*instance, markers, wire, offset, length, offset\_btl*)

Parse the value of this field from an encoded wire.

#### Parameters

- **instance** – the instance to parse into.
- **markers** (dict) – encoding marker variables. Only used in special cases.
- **wire** (Union[bytes, bytearray, memoryview]) – the TLV encoded wire.
- **offset** (int) – the offset of this field's Value in wire.
- **length** (int) – the Length of this field's Value.
- **offset\_btl** (int) – the offset of this field's TLV.

```
assert offset == (offset_btl
                  + get_tl_num_size(self.type_num)
                  + get_tl_num_size(length))
```

**Returns** the value.

**skipping\_process**(*markers, wire, offset*)

Called when this field does not occur in `wire` and thus be skipped.

#### Parameters

- **markers** (dict) – encoding marker variables.
- **wire** (Union[bytes, bytearray, memoryview]) – the TLV encoded wire.
- **offset** (int) – the offset where this field should have been if it occurred.

**class** `ndn.encoding.tlv_model.ProcedureArgument` (*default=None*)

A marker variable used during encoding or parsing. It does not have a value. Instead, it provides a way to access a specific variable in `markers`.

`__get__(instance, owner)`

**Returns** itself.

`__set__(instance, value)`

This is not allowed and will raise a `TypeError` if called.

`get_arg(markers)`

Get its value from markers

**Parameters** `markers` (dict) – the markers dict.

**Returns** its value.

`set_arg(markers, val)`

Set its value in markers.

**Parameters**

- **markers** (dict) – the markers dict.
- **val** – the new value.

`class ndn.encoding.tlv_model.OffsetMarker(default=None)`

A marker variable that records its position in TLV wire in terms of offset.

`class ndn.encoding.tlv_model.UintField(type_num, default=None, fixed_len=None)`

NonNegativeInteger field.

Type: `int`

Its Length is 1, 2, 4 or 8 when present.

**Variables** `fixed_len` (`int`) – the fixed value for Length if it's not `None`. Only 1, 2, 4 and 8 are acceptable.

`class ndn.encoding.tlv_model.BoolField(type_num, default=None)`

Boolean field.

Type: `bool`

Its Length is always 0. When present, its Value is `True`. When absent, its Value is `None`, which is equivalent to `False`.

---

**Note:** The default value is always `None`.

---

`class ndn.encoding.tlv_model.NameField(default=None)`

NDN Name field. Its Type is always `Name.TYPE_NAME`.

Type: `NonStrictName`

`class ndn.encoding.tlv_model.BytesField(type_num, default=None)`

Field for \*OCTET.

Type: `BinaryStr`

---

**Note:** Do not assign it with a `str`.

---

`class ndn.encoding.tlv_model.ModelField(type_num, model_type, copy_in_fields=None, copy_out_fields=None, ignore_critical=False)`

Field for nested TlvModel.

Type: *TlvModel*

#### Variables

- **model\_type** (*TlvModelMeta*) – the type of its value.
- **ignore\_critical** (bool) – whether to ignore critical fields (whose Types are odd).

**class** ndn.encoding.tlv\_model.RepeatedField(*element\_type*)

Field for an array of a specific type. All elements will be directly encoded into TLV wire in order, sharing the same Type. The `type_num` of `element_type` is used.

Type: list

**Variables** `element_type` (*Field*) – the type of elements in the list.

**Warning:** Please always create a new *Field* instance. Don't use an existing one.

**class** ndn.encoding.tlv\_model.TlvModelMeta(*name, bases, attrs*)

Metaclass for TlvModel, used to collect fields.

**class** ndn.encoding.tlv\_model.TlvModel

Used to describe a TLV format.

**Variables** `_encoded_fields` (*List[Field]*) – a list of *Field* in order.

`__eq__` (*other*)

Compare two TlvModels

**Parameters** `other` – the other TlvModel to compare with.

**Returns** whether all Fields are equal.

**asdict** (*dict\_factory=<class 'dict'>*)

Return a dict to represent this TlvModel.

**Parameters** `dict_factory` – class of dict.

**Returns** the dict.

**encode** (*wire=None, offset=0, markers=None*)

Encode the TlvModel.

#### Parameters

- **wire** (Union[bytearray, memoryview, None]) – the buffer to contain the encoded wire. A new bytearray will be created if it's None.
- **offset** (int) – the starting offset.
- **markers** (Optional[dict]) – encoding marker variables.

**Return type** Union[bytearray, memoryview]

**Returns** wire.

#### Raises

- **ValueError** – some field is assigned with improper value.
- **TypeError** – some field is assigned with value of wrong type.
- **IndexError** – wire does not have enough length.
- **struct.error** – a negative number is assigned to any non-negative integer field.

**encoded\_length**(*markers=None*)

Get the encoded Length of this TlvModel.

**Parameters** **markers** (Optional[dict]) – encoding marker variables.

**Return type** int

**Returns** the encoded Length.

**classmethod** **parse**(*wire, markers=None, ignore\_critical=False*)

Parse a TlvModel from TLV encoded wire.

**Parameters**

- **wire** (Union[bytes, bytearray, memoryview]) – the TLV encoded wire.
- **markers** (Optional[dict]) – encoding marker variables.
- **ignore\_critical** (bool) – whether to ignore unknown critical fields.

**Returns** parsed TlvModel.

**Raises**

- **DecodeError** – a critical field is unrecognized, redundant or out-of-order.
- **IndexError** – the Length of a field exceeds the size of wire.

### NDN Packet Format 0.3

**class** ndn.encoding.ndn\_format\_0\_3.**ContentType**

Numbers used in ContentType.

Type	Description
BLOB	Payload identified by the data name
LINK	A list of delegations
KEY	Public Key
NACK	Application-level NACK

**class** ndn.encoding.ndn\_format\_0\_3.**Delegation**

**class** ndn.encoding.ndn\_format\_0\_3.**InterestParam**(*can\_be\_prefix=False, must\_be\_fresh=False, nonce=None, lifetime=4000, hop\_limit=None, forwarding\_hint=<factory>*)

A dataclass collecting the parameters of an Interest, except ApplicationParameters.

**Variables**

- **can\_be\_prefix** (*bool*) – CanBePrefix. False by default.
- **must\_be\_fresh** (*bool*) – MustBeFresh. False by default.
- **nonce** (*int*) – Nonce. None by default.
- **lifetime** (*int*) – InterestLifetime in milliseconds. 4000 by default.
- **hop\_limit** (*int*) – HopLimit. None by default.
- **forwarding\_hint** (List [ Tuple [ int , *NonStrictName* ]]) – ForwardingHint. The type should be list of pairs of Preference and Name. e.g.: [(1, "/ndn/name1"), (2, ["ndn", "name2"])]

**class** ndn.encoding.ndn\_format\_0\_3.**KeyLocator**

```
class ndn.encoding.ndn_format_0_3.Links
```

```
class ndn.encoding.ndn_format_0_3.MetaInfo(content_type=0, freshness_period=None,
                                           final_block_id=None)
```

```
class ndn.encoding.ndn_format_0_3.SignatureInfo
```

```
class ndn.encoding.ndn_format_0_3.SignaturePtrs(signature_info=None,
                                                  signature_covered_part=<factory>,
                                                  signature_value_buf=None,
                                                  digest_covered_part=<factory>,
                                                  digest_value_buf=None)
```

A set of pointers used to verify a packet.

#### Variables

- **signature\_info** (*SignatureInfo*) – the SignatureInfo.
- **signature\_covered\_part** (List [ memoryview ]) – a list of pointers, each of which points to a memory covered by signature.
- **signature\_value\_buf** (memoryview) – a pointer to SignatureValue (TL excluded).
- **digest\_covered\_part** (List [ memoryview ]) – a list of pointers, each of which points to a memory covered by ParametersSha256DigestComponent.
- **digest\_value\_buf** (memoryview) – a pointer to ParametersSha256DigestComponent (TL excluded).

```
class ndn.encoding.ndn_format_0_3.SignatureType
```

Numbers used in SignatureType.

Type	Description
NOT_SIGNED	Not signed
DIGEST_SHA256	SHA-256 digest (only for integrity protection)
SHA256_WITH_RSA	RSA signature over a SHA-256 digest
SHA256_WITH_ECDSA	An ECDSA signature over a SHA-256 digest
HMAC_WITH_SHA256	SHA256 hash-based message authentication codes
NULL	An empty signature for testing and experimentation

```
class ndn.encoding.ndn_format_0_3.TypeNumber
```

TLV Type numbers used in [NDN Packet Format 0.3](#).

Constant names are changed to PEP 8 style, i.e., all upper cases with underscores separating words.

```
ndn.encoding.ndn_format_0_3.make_data(name, meta_info, content=None, signer=None)
```

Make a Data packet.

#### Parameters

- **name** (*NonStrictName*) – the Name field.
- **meta\_info** (*MetaInfo*) – the MetaInfo field.
- **content** (Optional [ *BinaryStr* ]) – the Content.
- **signer** (Optional[*Signer*]) – a Signer to sign this Interest. None if it is unsigned.

**Return type** Union[bytearray, memoryview]

**Returns** TLV encoded Data packet.

`ndn.encoding.ndn_format_0_3.make_interest(name, interest_param, app_param=None, signer=None, need_final_name=False)`

Make an Interest packet.

**Parameters**

- **name** (*NonStrictName*) – the Name field.
- **interest\_param** (*InterestParam*) – basic parameters of the Interest.
- **app\_param** (Optional [ *BinaryStr* ]) – the ApplicationParameters field.
- **signer** (Optional[*Signer*]) – a Signer to sign this Interest. None if it is unsigned.
- **need\_final\_name** (bool) – if True, also return the final Name with ParametersSha256DigestComponent.

**Returns** TLV encoded Interest packet. If `need_final_name`, return a tuple of the packet and the final Name.

`ndn.encoding.ndn_format_0_3.parse_data(wire, with_tl=True)`

Parse a TLV encoded Data.

**Parameters**

- **wire** (*BinaryStr*) – the buffer.
- **with\_tl** (bool) – True if the packet has Type and Length. False if wire only has the Value part.

**Returns** a Tuple of Name, MetaInfo, Content and *SignaturePtrs*.

**Return type** Tuple [ *FormalName* , *MetaInfo* , Optional [ *BinaryStr* ] , *SignaturePtrs* ]

`ndn.encoding.ndn_format_0_3.parse_interest(wire, with_tl=True)`

Parse a TLV encoded Interest.

**Parameters**

- **wire** (*BinaryStr*) – the buffer.
- **with\_tl** (bool) – True if the packet has Type and Length. False if wire only has the Value part.

**Returns** a Tuple of Name, InterestParameters, ApplicationParameters and *SignaturePtrs*.

**Return type** Tuple [ *FormalName* , *InterestParam* , Optional [ *BinaryStr* ] , *SignaturePtrs* ]

## 1.5 ndn.security package

### 1.5.1 Introduction

The `ndn.security` package provides basic tools for security use.



## 1.5.2 Signer

A *Signer* is a class used to sign a packet during encoding.

```
class ndn.encoding.Signer
```

```
    abstract get_signature_value_size()
```

Get the size of SignatureValue. If the size is variable, return the maximum possible value.

**Return type** int

**Returns** the size of SignatureValue.

```
    abstract write_signature_info(signature_info)
```

Fill in the fields of SignatureInfo.

**Parameters** *signature\_info* – a blank SignatureInfo object.

```
    abstract write_signature_value(wire, contents)
```

Calculate the SignatureValue and write it into wire. The length of wire is exactly what *get\_signature\_value\_size()* returns. Basically this function should return the same value except for ECDSA.

**Parameters**

- **wire** (Union[bytearray, memoryview]) – the buffer to contain SignatureValue.
- **contents** (List[Union[bytearray, memoryview]]) – a list of memory blocks that needs to be covered.

**Return type** int

**Returns** the actual size of SignatureValue.

## 1.5.3 Validator

A *Validator* is a async function called to validate an Interest or Data packet. It takes 2 arguments: a *FormalName* and a *SignaturePtrs*, and returns whether the packet is validated.

## 1.5.4 Keychain

A *Keychain* is a class which contains Identities, Keys associated with Identities and associated Certificates.

```
class ndn.security.keychain.Keychain
```

The abstract Keychain class, derived from `collections.abc.Mapping`. It behaves like an immutable dict from *FormalName* to Identity. The implementation of Identity varies with concrete implementations. Generally, its methods should also accept *NonStrictName* as inputs. This includes operators such as `in` and `[]`.

```
    abstract get_signer(sign_args)
```

Get a signer from sign\_args.

**Parameters** *sign\_args* (Dict[str, Any]) – the signing arguments provided by the application.

**Returns** a signer.

**Return type** *Signer*

## KeychainDigest

**class** ndn.security.keychain.keychain\_digest.**KeychainDigest**

A signer which has no Identity and always returns a SHA-256 digest signer.

**get\_signer**(*sign\_args*)

Get a signer from sign\_args.

**Parameters** **sign\_args** (Dict[str, Any]) – the signing arguments provided by the application.

**Returns** a signer.

**Return type** *Signer*

## KeychainSqlite3

This is the default Keychain.

**class** ndn.security.keychain.keychain\_sqlite3.**Certificate**(*id, key, name, data, is\_default*)

A dataclass for a Certificate.

### Variables

- **id** (*int*) – its id in the database.
- **key** (*FormalName*) – the Name of the associated Key.
- **name** (*FormalName*) – its Name.
- **data** (*bytes*) – the content.
- **is\_default** (*bool*) – whether this is the default Identity.

**class** ndn.security.keychain.keychain\_sqlite3.**Identity**(*pib, row\_id, name, is\_default*)

An Identity. It behaves like an immutable dict from *FormalName* to *Key*.

### Variables

- **row\_id** (*int*) – its id in the database.
- **name** (*FormalName*) – its Name.
- **is\_default** (*bool*) – whether this is the default Identity.

**default\_key**()

Get the default Key.

**Return type** *Key*

**Returns** the default Key.

**del\_key**(*name*)

Delete a specific Key.

**Parameters** **name** (*NonStrictName*) – the Name of the Key to delete.

**has\_default\_key**()

Whether it has a default Key.

**Return type** bool

**Returns** True if there is one.

**new\_key**(*key\_type*)

Create a new key with default arguments.

**Parameters** `key_type` (`str`) – the type of the Key. Can be `ec` or `rsa`.

**Return type** `Key`

**Returns** the new Key.

`set_default_key(name)`

Set the default Key.

**Parameters** `name` (`NonStrictName`) – the Name of the new default Key.

**class** `ndn.security.keychain.keychain_sqlite3.Key`(`pib`, `identity`, `row_id`, `name`, `key_bits`, `is_default`)

A Key. It behaves like an immutable dict from `FormalName` to `Certificate`.

#### Variables

- `row_id` (`int`) – its id in the database.
- `identity` (`FormalName`.) – the Name of the associated Identity.
- `name` (`FormalName`) – its Name.
- `key_bits` (`bytes`) – the key bits of the public key.
- `is_default` (`bool`) – whether this is the default Identity.

`default_cert()`

Get the default Certificate.

**Return type** `Certificate`

**Returns** the default Certificate.

`del_cert(name)`

Delete a specific Certificate.

**Parameters** `name` (`NonStrictName`) – the Name of the Key to delete.

`has_default_cert()`

Whether it has a default Certificate.

**Return type** `bool`

**Returns** True if there is one.

`set_default_cert(name)`

Set the default Certificate.

**Parameters** `name` (`NonStrictName`) – the Name of the new default Certificate.

**class** `ndn.security.keychain.keychain_sqlite3.KeychainSqlite3`(`path`, `tpm`)

Store public information in a Sqlite3 database and private keys in a TPM.

#### Variables

- `path` (`str`) – the path to the database. The default path is `~/.ndn/pib.db`.
- `tpm` (`Tpm`) – an instance of TPM.
- `tpm_locator` (`str`) – a URI string describing the location of TPM.

`default_identity()`

Get the default Identity.

**Return type** `Identity`

**Returns** the default Identity.

**del\_cert**(*name*)

Delete a specific Certificate.

**Parameters** **name** (*NonStrictName*) – the Certificate Name.

**del\_identity**(*name*)

Delete a specific Identity.

**Parameters** **name** (*NonStrictName*) – the Identity Name.

**del\_key**(*name*)

Delete a specific Key.

**Parameters** **name** (*NonStrictName*) – the Key Name.

**get\_signer**(*sign\_args*)

Get a signer from *sign\_args*.

**Parameters** **sign\_args** (Dict[str, Any]) – the signing arguments provided by the application.

**Returns** a signer.

**Return type** *Signer*

**has\_default\_identity**()

Whether there is a default Identity. :rtype: bool :return: True if there is one.

**new\_identity**(*name*)

Create a new Identity without a default Key. This is used to control the Keychain in a fine-grained way.

**Parameters** **name** (*NonStrictName*) – the Name of the new Identity.

**Return type** *Identity*

**Returns** the Identity created.

**new\_key**(*id\_name*, *key\_type='ec'*, *\*\*kwargs*)

Generate a new key for a specific Identity.

**Parameters**

- **id\_name** (*NonStrictName*) – the Name of Identity.
- **key\_type** (str) – the type of key. Can be one of the following:
  - ec: ECDSA key.
  - rsa: RSA key.
- **kwargs** – keyword arguments.

**Keyword Arguments**

- **key\_size** (int) - key size in bit.
- **key\_id** (Union[*BinaryStr*, str]) - a one-Component ID of the Key.
- **key\_id\_type** (str) - the method to generate the ID if *key\_id* is not specified. Can be random or sha256.

**Return type** *Key*

**Returns** the new Key.

**set\_default\_identity**(*name*)

Set the default Identity.

**Parameters** **name** (*NonStrictName*) – the Name of the new default Identity.

**shutdown()**

Close the connection.

**touch\_identity(*id\_name*)**

Get an Identity with specific name. Create a new one if it does not exist. The newly created one will automatically have a default ECC Key and self-signed Certificate.

**Parameters** *id\_name* (*NonStrictName*) – the Name of Identity.

**Return type** *Identity*

**Returns** the specified Identity.

## 1.6 ndn.schema package

**Warning:** Name Tree Schema (NTSchema) is experimental and capricious. The current implementation is treated as a proof-of-concept demo.

### 1.6.1 Introduction

The `ndn.schema` package provides an implementation of Name Tree Schema, an application framework that organizes application functionalities by the application namespace. Modularized NDN libraries can be developed based on it, and application developers can use those libraries as building blocks.

The core concept of NTSchema is the namespace schema tree. The schema tree is a tree structure that contains all possible naming conventions of an application. Different from a tree of names, its edge may be a pattern variable instead of a specific name component. For example, the path `/<Identity>/KEY/<KeyID>` can be used to represent a naming convention of a key, where specific keys – like `/Alice/KEY/%01` and `/Bob/KEY/%c2` match with it.

Two main components of NTSchema are custom nodes and policies. In the schema tree, every node represents a namespace. After matching with a specific name, a node can be used to produce and consume data. For example, if we call `matched_node = tree.match('/Alice/KEY/%01')`, it will return a matching of node `/<Identity>/KEY/<KeyID>` with variable setting `Identity='Alice'`, `KeyID='\x01'`. Then we call `matched_node.provide(key_data)`, it will generate the key with data `key_data` and make it available. When we call `key_data = matched_node.need()`, it will try to fetch the key. A custom node will have customized pipeline to handle `provide` and `need` function calls. Policies are annotations attached to nodes, that specifies user-defined policies that are security, storage, etc.

### 1.6.2 Examples

#### 1 - File Sharing

Assume that Alice has several devices and wants to share some files among them. To simplify the case, we assume that all devices have Alice's key, which can be used as the trust anchor. A file may be large, so segmentation is needed.

## Design

First, let's start with the namespace design. There are two kinds of object in the system, one is the key, the other is the file.

For the file, an option is [RDR protocol](#). RDR protocol handles the version discovery and segmentation. There is no need to know the implementation details, since NTScheme allows we use an existing protocols as a black box. In short, RDR has:

- A metadata packet that contains a version number of the content.
- A series of data packets containing segmented data content.

For the key, we can use a single Data packet to contain the certificate.

---

**Note:** This example is only used for demo, which is different from the real-world scenario.

- RDR is not necessary in this scenario, since there is only one version for each file.
  - In real world, Alice may want to have a trust anchor instead of sharing a single key.
- 

The whole namespace design is shown as follows:

In the figure, `/file/<FileName>` is the file object and `/<IDName>/KEY/<KeyID>/self/<CertID>` represents the certificate. Here, `<variable>` is a pattern variable that matches exactly one name component. The real names may be `/file/foo.txt` and `/Alice/KEY/%29/self/%F6`. Also, note that `/file/<FileName>` is an object composed of multiple data packets, which are managed by [RDRNode](#) and not exposed to the programmer.

Then, let's move to the policies part. We want to ensure the following requirements:

- All data packets are stored in memory, so if another node requests this file, the current node can serve it. This applies to both the producer – which loads the file from the disk and create packets, and the consumer – which receives the file from another node.
- Data packets of the file must be signed by Alice's key. The certificate can be preloaded into memory when the program starts.

Let's attach these two policies onto the namespace schema tree we have:

The [MemoryCachePolicy](#) indicates all data packets are stored in memory. And [SignedBy](#) requires data packets with prefix `/file/<FileName>` to be signed by key `/<IDName>/KEY/<KeyID>`. We can add restrictions, such as `IDName == 'Alice'`, to limit the identity.

## Coding

With NTScheme, we can translate our design into code directly:

```
# Make schema tree
root = Node()
root['/<IDName>/KEY/<KeyID>/self/<CertID>'] = Node()
root['/file/<FileName>'] = RDRNode()

# Set policies
id_name = Name.Component.get_value(app.keychain.default_identity().name[0])
```

(continues on next page)

(continued from previous page)

```

cache = MemoryCache()
root.set_policy(policy.Cache, MemoryCachePolicy(cache))
root['/file/<FileName>'].set_policy(
    policy.DataValidator,
    SignedBy(root['/<IDName>/KEY/<KeyID>'],
              subject_to=lambda _, vars: vars['IDName'] == id_name))

```

The full source code can be found in `examples/rdrnode.py`.

### 1.6.3 Reference

#### Namespace Schema Tree

**exception** `ndn.schema.schema_tree.LocalResourceNotExistError(name)`

Raised when trying to fetch a local resource that does not exist. Used only when `LocalOnly` is attached to the node.

**class** `ndn.schema.schema_tree.MatchedNode(root, node, name, pos, env, policies)`

`MatchedNode` represents a matched static tree node. That is, a node with all name patterns on the path from the root to it assigned to some value. For example, if the tree contains a node `N` on the path `/a/<b>/<c>`, and the user use the Name `/a/x/y` to match, then a matched node `(N, {'b': 'x', 'c': 'y'})` will be returned.

#### Variables

- **root** (`Node`) – the root of the static tree.
- **node** (`Node`) – the matched node of the static tree.
- **name** (`FormalName`) – the name used to match.
- **pos** (`int`) – an integer indicating the length the name is matched. Generally, it equals the length of name.
- **env** (`Dict[str, Any]`) – a dict containing the value all pattern variables matched on the path.
- **policies** (`Dict[Type[policy.Policy], policy.Policy]`) – a dict collecting all policies that apply to this node. For each type of policy, the one attached on the nearest ancestor is collected here.

**app()**

The `NDNApp` the static tree is attached to.

**Return type** `NDNApp`

**Returns** the `NDNApp`.

**async express**(`app_param=None, **kwargs`)

Try to fetch the data, called by the node's need function. It will search the local cache, and examines the local resource. If the corresponding Data cannot be found in the two places, it encrypts the `app_param` and expresses the Interest.

---

**Note:** This function only sends out an Interest packet when the Data is not cached locally.

---

#### Parameters

- **app\_param** (Union[bytes, bytearray, memoryview, None]) – the ApplicationParameter of the Interest.
- **kwargs** – other parameters of the Interest.

**Returns** whatever `process_data` returns. Generally this function is only called at the default node, so the return value is a tuple of the content and a dict containing metadata.

**finer\_match**(*new\_name*)

Do a finer match based on current match. `new_name` must include current `name` as its prefix. For example, if the current match name is `/a/b` and we want to get the matched node for `/a/b/c`, then we can call `finer_match` with `/a/b/c`.

**Parameters** **new\_name** (List[Union[bytes, bytearray, memoryview]]) – the new name to be matched. Must include current name as its prefix.

**Returns** the new matched node.

**need**(\*\**kwargs*)

Consume an object corresponding to this node. Specific node type may have customized processing pipeline. For example, a `SegmentedNode` can do reassembly here. By default it sends an Interest packet to fetch a `Data`.

`MatchedNode`'s `need` simply calls the node's `need` function.

**Parameters** **kwargs** – arguments from user input.

**Returns** the object needed, whose format is defined by specific node type. By default, it returns a tuple of the content and a dict of metadata.

**async on\_data**(*meta\_info*, *content*, *raw\_packet*)

Called when a `Data` packet comes. It saves the `Data` packet into the cache, decrypts the content, and calls the node's `process_data` function.

**Parameters**

- **meta\_info** (*MetaInfo*) – the `MetaInfo` of the incoming `Data` packet.
- **content** (Union[bytes, bytearray, memoryview, None]) – the content of the `Data`.
- **raw\_packet** (Union[bytes, bytearray, memoryview]) – the raw `Data` packet.

**Returns** whatever `process_data` returns.

**async on\_interest**(*param*, *app\_param*, *raw\_packet*)

Called when an Interest packet comes. It looks up the cache and returns a `Data` packet if it exists. Otherwise, it decrypts `ApplicationParameters` and calls the node's `process_int` function.

**Parameters**

- **param** (*InterestParam*) – the parameters of the incoming Interest.
- **app\_param** (Union[bytes, bytearray, memoryview, None]) – the ApplicationParameters of the Interest.
- **raw\_packet** (Union[bytes, bytearray, memoryview]) – the raw Interest packet.

**provide**(*content*, \*\**kwargs*)

Produce an object corresponding to this node, and make all generated `Data` packets available. Specific node type may have customized processing pipeline. For example, a `SegmentedNode` can do segmentation here. By default it makes a `Data` packet out of content and put it into the cache.

`MatchedNode`'s `provide` simply calls the node's `provide` function.

**Parameters**



- **content** – the content of the object.
- **kwargs** – other arguments from user input. Defined by specific node type.

**async put\_data**(*content=None, send\_packet=False, \*\*kwargs*)

Generate the Data packet out of content. This function encrypts the content, encodes and signs the packet, saves it into the cache, and optionally sends it to the face. This function is called by the node's provide function.

#### Parameters

- **content** (Union[bytes, bytearray, memoryview, None]) – the Data content.
- **send\_packet** (bool) – whether sends the Data packet to the face.
- **kwargs** – other arguments generating the Data packet.

**class ndn.schema.schema\_tree.Node**(*parent=None*)

Node represents a node in the static namespace tree.

#### Variables

- **policies** (*Dict*[*Type*[*policy.Policy*], *policy.Policy*]) – policies attached to this node
- **prefix** (*FormalName*) – the prefix of the root node of the tree. Generally not set for other nodes.
- **app** (*Optional*[*NDNApp*]) – the *NDNApp* this static tree is attached to. Only available at the root.

**async attach**(*app, prefix*)

Attach this node to a specified *NDNApp*, register all name prefixes. This node becomes the root node of the application static tree. *prefix* is the prefix of the tree, which will be prepended to all names under this tree. For example, if *prefix='/a/blog'*, then the node with path */articles* from this node will become */a/blog/articles*.

**Warning:** The way to register prefixes is still under discussion. Currently, we register the nodes that we can reach without going through a pattern. Also, there is no *detach* function yet, and no means to change the static tree after it's attached.

#### Parameters

- **app** (*NDNApp*) – the *NDNApp* to be attached to.
- **prefix** (Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]) – the prefix of the static tree.

**Returns** whether succeeded or not.

**exist**(*key*)

If it has a child with specified name component or nme pattern.

**Parameters** **key** – a name component (bytes) or a patten (tuple).

**Returns** whether the child node exists

**get\_policy**(*typ*)

Get the policy of specified type that applies to this node. It can be attached to this node or a parent of this node.

**Parameters** **typ** (*Type*[*Policy*]) – a policy type

**Returns** the policy. None if there does not exist one.

**match**(*name*)

Start from this node, go the path that matches with the name, and return the node it reaches when it cannot go further.

**Parameters** **name** (Union[Iterable[Union[bytes, bytearray, memoryview, str]], str, bytes, bytearray, memoryview]) – an NDN name.

**Returns** a *MatchedNode*, which contains the destination node and variables matched.

**async need**(*match*, *\*\*kwargs*)

Consume an object corresponding to this node. Specific node type can override this function to have customized processing pipeline. For example, a *SegmentedNode* can do reassembly here. By default it sends an Interest packet to fetch a Data.

**Parameters**

- **match** – the matched node object of this node.
- **kwargs** – other arguments from user input.

**Returns** This is defined by the node type. By default it returns what *process\_data()* returns. That is, a tuple of context and metadata dict.

**async on\_register**(*root*, *app*, *prefix*, *cached*)

Called when the root node *root* is attached to *app*, and the *attach()* wants to register prefixed under the subtree rooted at this node.

**Parameters**

- **root** – the root of the static tree.
- **app** (*NDNApp*) – the *NDNApp* to be attached to.
- **prefix** (List[Union[bytes, bytearray, memoryview]]) – the prefix of the static tree.
- **cached** (bool) – If there is a cache policy that applies to this node.

**Returns** whether succeeded or not.

**async process\_data**(*match*, *meta\_info*, *content*, *raw\_packet*)

Processing an incoming Data packet. Specific node type can override this function to have customized processing pipeline. By default it returns the content.

**Parameters**

- **match** – the matched node object of this node.
- **meta\_info** (*MetaInfo*) – the *MetaInfo* of the Data packet.
- **content** (Union[bytes, bytearray, memoryview, None]) – the content of the Data packet.
- **raw\_packet** (Union[bytes, bytearray, memoryview]) – the raw Data packet.

**Returns** a tuple, whose first element is data content after processing, and second is a dict [str, Any] containing metadata.

**async process\_int**(*match*, *param*, *app\_param*, *raw\_packet*)

Processing an incoming Interest packet. Specific node type can override this function to have customized processing pipeline.

---

**Note:** This function will not be called if the Interest packet is satisfied with a cached Data packet.

---

**Parameters**

- **match** – the matched node object of this node.
- **param** (*InterestParam*) – the parameters of the Interest packet.
- **app\_param** (Union[bytes, bytearray, memoryview, None]) – the ApplicationParameters of the Interest packet.
- **raw\_packet** (Union[bytes, bytearray, memoryview]) – the raw Interest packet.

**async provide**(*match, content, \*\*kwargs*)

Produce an object corresponding to this node, and make all generated Data packets available. Specific node type can override this function to have customized processing pipeline. For example, a SegmentedNode can do segmentation here. By default it makes a Data packet out of content and put it into the cache.

**Parameters**

- **match** – the matched node object of this node.
- **content** – the content of the object.
- **kwargs** – other arguments from user input.

**set\_policy**(*typ, value*)

Attach a policy to this node.

**Parameters**

- **typ** (Type[*Policy*]) – the policy type.
- **value** (*Policy*) – the policy to be attached to this node.

**exception** ndn.schema.schema\_tree.**NodeExistsError**(*pattern*)

Raised when trying to create a node which already exists.

**Utils**

ndn.schema.util.**NamePattern**

NamePattern is a list containing mixed name components and variable patterns. A variable pattern is a capturing pattern that matches with exactly one name component. It is a tuple containing 3 variables:

- The 1st element is reserved and always 0. This is a quick and dirty solution in this PoC implementation. It will be used if we want to support multiple name components matching patterns.
- The 2nd element is the TLV type of the name component to be matched.
- The 3rd element is the name of the pattern variable.

alias of List[Union[bytes, bytearray, memoryview, Tuple[int, int, str]]]

ndn.schema.util.**norm\_pattern**(*name*)

This function returns a normalized name pattern from a string, just like normalizing a name.

**Parameters** **name** (str) – the name pattern string.

**Return type** List[Union[bytes, bytearray, memoryview, Tuple[int, int, str]]]

**Returns** normalized name pattern.

## Custom Nodes

**class** `ndn.schema.simple_node.LocalResource`(*parent=None, data=None*)

`LocalResource` is a custom node that preloads some data. When `need()` is called, it returns the loaded data directly. This node type does not interact with the network.

**async need**(*match, \*\*kwargs*)

Consume an object corresponding to this node. Specific node type can override this function to have customized processing pipeline. For example, a `SegmentedNode` can do reassembly here. By default it sends an Interest packet to fetch a `Data`.

### Parameters

- **match** – the matched node object of this node.
- **kwargs** – other arguments from user input.

**Returns** This is defined by the node type. By default it returns what `process_data()` returns. That is, a tuple of context and metadata dict.

**async on\_register**(*root, app, prefix, cached*)

Called when the root node `root` is attached to `app`, and the `attach()` wants to register prefixed under the subtree rooted at this node.

### Parameters

- **root** – the root of the static tree.
- **app** – the `NDNApp` to be attached to.
- **prefix** – the prefix of the static tree.
- **cached** (bool) – If there is a cache policy that applies to this node.

**Returns** whether succeeded or not.

**async provide**(*match, content, \*\*kwargs*)

Produce an object corresponding to this node, and make all generated `Data` packets available. Specific node type can override this function to have customized processing pipeline. For example, a `SegmentedNode` can do segmentation here. By default it makes a `Data` packet out of `content` and put it into the cache.

### Parameters

- **match** – the matched node object of this node.
- **content** – the content of the object.
- **kwargs** – other arguments from user input.

**class** `ndn.schema.simple_node.RDRNode`(*parent=None, \*\*kwargs*)

`RDRNode` represents a versioned and segmented object whose encoding follows the RDR protocol. Its `provide` function generates the metadata packet, and `need` function handles version discovery.

**class** `MetaData`(*parent=None*)

**make\_metadata**(*match*)

**async need**(*match, \*\*kwargs*)

Consume an object corresponding to this node. Specific node type can override this function to have customized processing pipeline. For example, a `SegmentedNode` can do reassembly here. By default it sends an Interest packet to fetch a `Data`.

### Parameters

- **match** – the matched node object of this node.

- **kwargs** – other arguments from user input.

**Returns** This is defined by the node type. By default it returns what `process_data()` returns. That is, a tuple of context and metadata dict.

**async process\_int**(*match, param, app\_param, raw\_packet*)

Processing an incoming Interest packet. Specific node type can override this function to have customized processing pipeline.

---

**Note:** This function will not be called if the Interest packet is satisfied with a cached Data packet.

---

#### Parameters

- **match** – the matched node object of this node.
- **param** – the parameters of the Interest packet.
- **app\_param** – the ApplicationParameters of the Interest packet.
- **raw\_packet** – the raw Interest packet.

#### class MetaDataValue

**async need**(*match, \*\*kwargs*)

Consume an object corresponding to this node. Specific node type can override this function to have customized processing pipeline. For example, a SegmentedNode can do reassembly here. By default it sends an Interest packet to fetch a Data.

#### Parameters

- **match** – the matched node object of this node.
- **kwargs** – other arguments from user input.

**Returns** This is defined by the node type. By default it returns what `process_data()` returns. That is, a tuple of context and metadata dict.

**async provide**(*match, content, \*\*kwargs*)

Produce an object corresponding to this node, and make all generated Data packets available. Specific node type can override this function to have customized processing pipeline. For example, a SegmentedNode can do segmentation here. By default it makes a Data packet out of content and put it into the cache.

#### Parameters

- **match** – the matched node object of this node.
- **content** – the content of the object.
- **kwargs** – other arguments from user input.

**class** `ndn.schema.simple_node.SegmentedNode`(*parent=None, timeout=4000, retry\_times=3, segment\_size=4400*)

SegmentedNode represents a segmented object. The segmented object is composed with multiple Data packets, whose name have a suffix “/seg=seg\_no” attached to the object’s name. The provide function handles segmentation, and the need function handles reassembly.

---

**Note:** Currently, the fetching pipeline is a simple one-by-one pipeline. where only one Interest will be in-flight at one time.

---

**async need**(*match, \*\*kwargs*)

Consume an object corresponding to this node. Specific node type can override this function to have customized processing pipeline. For example, a SegmentedNode can do reassembly here. By default it sends an Interest packet to fetch a Data.

#### Parameters

- **match** – the matched node object of this node.
- **kwargs** – other arguments from user input.

**Returns** This is defined by the node type. By default it returns what `process_data()` returns. That is, a tuple of context and metadata dict.

**async process\_int**(*match, param, app\_param, raw\_packet*)

Processing an incoming Interest packet. Specific node type can override this function to have customized processing pipeline.

---

**Note:** This function will not be called if the Interest packet is satisfied with a cached Data packet.

---

#### Parameters

- **match** – the matched node object of this node.
- **param** – the parameters of the Interest packet.
- **app\_param** – the ApplicationParameters of the Interest packet.
- **raw\_packet** – the raw Interest packet.

**async provide**(*match, content, \*\*kwargs*)

Produce an object corresponding to this node, and make all generated Data packets available. Specific node type can override this function to have customized processing pipeline. For example, a SegmentedNode can do segmentation here. By default it makes a Data packet out of content and put it into the cache.

#### Parameters

- **match** – the matched node object of this node.
- **content** – the content of the object.
- **kwargs** – other arguments from user input.

## Policies

### Policy Types

**class** `ndn.schema.policy.Cache`

Cache policy determines how Data packets are stored.

**class** `ndn.schema.policy.DataEncryption`

DataEncryption policy is a type used to indicate the Data encryption policy. Used as the type argument of `set_policy`.

**class** `ndn.schema.policy.DataSigning`

DataSigning policy is a type used to indicate the Data signer. Used as the type argument of `set_policy`.

**class** `ndn.schema.policy.DataValidator`

DataValidator policy describes how to verify a Data packet.

**class** `ndn.schema.policy.Encryption`

Encryption policy encrypts and decrypts content. When a user uses encryption policy, he needs to specify whether its *InterestEncryption* or *DataEncryption*.

**class ndn.schema.policy.InterestEncryption**

InterestSigning policy is a type used to indicate the Interest encryption policy. Used as the type argument of set\_policy.

**class ndn.schema.policy.InterestSigning**

InterestSigning policy is a type used to indicate the Interest signer. Used as the type argument of set\_policy.

**class ndn.schema.policy.InterestValidator**

InterestValidator policy describes how to verify an Interest packet.

**class ndn.schema.policy.LocalOnly**

LocalOnly means the Data should be stored in the local storage. It prevents the node from sending Interest packets.

**class ndn.schema.policy.Policy**

Policy is an annotation attached to a node.

**class ndn.schema.policy.Register**

Register policy indicates the node should be registered as a prefix in the forwarder.

**class ndn.schema.policy.Signing**

Signing policy gives a signer used to sign a packet. When a user uses signing policy, he needs to specify whether its *InterestSigning* or *DataSigning*.

**Trust Policies****class ndn.schema.simple\_trust.SignedBy**(key, subject\_to=None)

SignedBy policy represents the trust schema, specifying the key used to signed the Interest or Data packet. It does the follows:

- Match the key used to sign the packet in the static tree. The real key must match the node specified by key. Otherwise, the validation fails.
- Call the checker subject\_to with two matching variable dict. Fail if the checker returns False.
- Call the need function of the matched key node to get the public key. Fail if the key cannot be fetched.
- Verify the signature.

---

**Note:** Theoretically, SignedBy should also give the signer used to sign outgoing packets. However, this function is missing in current implementation.

---

For example,

```
# This checker checks the Author of Data is the same as the Author of the key.
def check_author(data_env, key_env):
    return data_env['Author'] == key_env['Author']

root = Node()
root['/author/<Author>/KEY/<KeyID>/self/<CertID>'] = Node()
root['/blog/<Author>/<Category>/<Date>'] = Node()
# The Data "/blog/<Author>/<Category>/<Date>" should be signed by
# the key "/author/<Author>/KEY/<KeyID>" with the same author.
root['/blog/<Author>/<Category>/<Date>'].set_policy(
    policy.DataValidator,
    SignedBy(root['/author/<Author>/KEY/<KeyID>'], subject_to=check_author))
```

## Cache Policies

**class** ndn.schema.simple\_cache.**MemoryCache**

MemoryCache is a simple cache class that supports searching and storing Data packets in the memory.

**async** **save**(*name*, *packet*)

Save a Data packet with name into the memory storage.

**Parameters**

- **name** (List[Union[bytes, bytearray, memoryview]]) – the Data name.
- **packet** (Union[bytes, bytearray, memoryview]) – the raw Data packet.

**async** **search**(*name*, *param*)

Search for the data packet that satisfying an Interest packet with name specified.

**Parameters**

- **name** (List[Union[bytes, bytearray, memoryview]]) – the Interest name.
- **param** (*InterestParam*) – the parameters of the Interest. Not used in current implementation.

**Returns** a raw Data packet or None.

**class** ndn.schema.simple\_cache.**MemoryCachePolicy**(*cache*)

MemoryCachePolicy stores Data packets in memory.

## 1.7 Miscellaneous packages

### 1.7.1 ndn.types package

**exception** ndn.types.**InterestCanceled**

Raised when an Interest is cancelled due to the loss of connection to NFD.

---

**Note:** A very large packet may cause NFD shutting down the connection. More specifically,

- The face is shutdown.
  - All pending Interests are cancelled with this exception.
  - `App.run_forever()` returns True.
- 

**exception** ndn.types.**InterestNack**(*reason*)

Raised when receiving a NetworkNack.

**Variables** **reason** (*int*) – reason for Nack.

**exception** ndn.types.**InterestTimeout**

Raised when an Interest times out.

**exception** ndn.types.**NetworkError**

Raised when trying to send a packet before connecting to NFD.

ndn.types.**Route**

An OnInterest callback function for a route.



alias of Callable[[List[Union[bytes, bytearray, memoryview]], [ndn.encoding.ndn\\_format\\_0\\_3.InterestParam](#), Optional[Union[bytes, bytearray, memoryview]]], None]

**exception** `ndn.types.ValidationFailure`(*name*, *meta\_info*, *content*)

Raised when failing to validate a Data packet.

#### Variables

- **name** (*FormalName*) – the Name of Data.
- **meta\_info** (*MetaInfo*) – the MetaInfo.
- **content** (Optional[*BinaryStr*]) – the Content of Data.

`ndn.types.Validator`

A validator used to validate an Interest or Data packet.

alias of Callable[[List[Union[bytes, bytearray, memoryview]], [ndn.encoding.ndn\\_format\\_0\\_3.SignaturePtrs](#)], Coroutine[Any, None, bool]]

## 1.7.2 ndn.utils package

`ndn.utils.gen_nonce()`

Generate a random nonce.

**Returns** a random 32-bit unsigned integer.

`ndn.utils.gen_nonce_64()`

Generate a random 64-bit nonce.

**Returns** a random 64-bit unsigned integer.

`ndn.utils.timestamp()`

Generate a timestamp number.

**Returns** the time in milliseconds since the epoch as an integer

## 1.8 Examples

Here lists some examples of python-ndn.

### 1.8.1 Basic Applications

#### Connect to NFD

NDNApp connects to an NFD node and provides interface to express and process Interests. The following code initializes an NDNApp instance with default configuration.

```
from ndn.app import NDNApp
app = NDNApp()
app.run_forever()
```

If there is a main function for the application, use the `after_start` argument.

```
from ndn.app import NDNApp
app = NDNApp()

async def main():
    # Do something
    app.shutdown() # Close the connection and shutdown

app.run_forever(after_start=main())
```

## Consumer

A consumer can use `express_interest` to express an Interest. If a Data is received and validated, it returns the Name, MetaInfo and Content of Data. Otherwise, an exception is thrown.

```
from ndn.encoding import Name

async def main():
    try:
        data_name, meta_info, content = await app.express_interest(
            # Interest Name
            '/example/testApp/randomData',
            must_be_fresh=True,
            can_be_prefix=False,
            # Interest lifetime in ms
            lifetime=6000)
        # Print out Data Name, MetaInfo and its content.
        print(f'Received Data Name: {Name.to_str(data_name)}')
        print(meta_info)
        print(bytes(content) if content else None)
    except InterestNack as e:
        # A NACK is received
        print(f'Nacked with reason={e.reason}')
    except InterestTimeout:
        # Interest times out
        print(f'Timeout')
    except InterestCanceled:
        # Connection to NFD is broken
        print(f'Canceled')
    except ValidationFailure:
        # Validation failure
        print(f'Data failed to validate')
    finally:
        app.shutdown()
```

## Producer

A producer can call `route` to register a permanent route. Route registration can be done before application is started. NDNApp will automatically announce that route to the NFD node.

```
@app.route('/example/testApp')
def on_interest(name, interest_param, application_param):
    app.put_data(name, content=b'content', freshness_period=10000)
```

## 1.8.2 Customized TLV Models

### Encoding

python-ndn provides a descriptive way to define a specific TLV format, called TLV model. Every object can be described by a class derived from `TlvModel`. Elements of a TLV object is expressed as an instance variable of `Field`. Fields are encoded in order.

```
from ndn.encoding import *

class Model(TlvModel):
    # Model = [Name] [IntVal] [StrVal] [BoolVal]
    name = NameField() # Name = NAME-TYPE TLV-LENGTH ...
    int_val = UintField(0x03) # IntVal = INT-VAL-TYPE TLV-LENGTH nonNegativeInteger
    str_val = BytesField(0x02) # StrVal = STR-VAL-TYPE TLV-LENGTH *OCTET
    bool_val = BoolField(0x01) # BoolVal = BOOL-VAL-TYPE 0

model = Model()
model.name = '/name'
model.str_val = b'bit string'
assert model.encode() == b'\x07\x06\x08\x04name\x02\nbit string'

model = Model.parse(b'\x07\x06\x08\x04name\x02\nbit string')
assert model.str_val == b'bit string'
```

There is *no required* fields in a TLV model. Every `Field` is `None` by default, which means it will not be encoded.

### Nested Model

python-ndn allows a TLV model to be a field (`ModelField`) of another TLV model, which enables a hierarchical structure. Also, a TLV model does not contain the outer Type and Length. This can be solved by encapsulating it into another TLV model.

```
class Inner(TlvModel):
    # Inner = [Val1]
    val1 = UintField(0x01) # Val1 = 1 TLV-LENGTH nonNegativeInteger

class Outer(TlvModel):
    # Outer = [Val2]
    val2 = ModelField(0x02, Inner) # Val2 = 2 TLV-LENGTH Inner

obj = Outer()
obj.val2 = Inner()
obj.val2.val1 = 255
assert obj.encode() == b'\x02\x03\x01\x01\xff'
```

## Repeated Model

*RepeatedField* is an array of a specific type of field. When encoding, elements are encoded in order.

```
class WordArray(TlvModel):
    words = RepeatedField(UintField(0x01, fixed_len=2)) # WordArray = *Words
                                                    # Words = 1 2 2OCTET

array = WordArray()
array.words = [i for i in range(3)]
assert array.encode() == b'\x01\x02\x00\x00\x01\x02\x00\x01\x01\x02\x00\x02'
```

## Derivation

To avoid duplication, a *TlvModel* can extend 1 or more other *TlvModels*. However, to indicate the locations of base classes in the TLV encoded wire, there must be a field for every base class to explicitly include its base class. These fields must have the value *IncludeBase*. *TlvModel* instances' Include fields cannot be assigned, and will be ignored during encoding and parsing.

```
class Base(TlvModel):
    m2 = UintField(0x02) # Base = [M2]

class Derived(Base):
    m1 = UintField(0x01) # Derived = [M1] [M2] [M3]
    _base = IncludeBase(Base)
    m3 = UintField(0x03)

obj = Derived()
obj.m1, obj.m2, obj.m3 = range(1, 4)
assert obj.encode() == b'\x01\x01\x01\x02\x01\x02\x03\x01\x03'
```

## Overriding

The derived class can override fields of its base classes. To override a field, declare a field with the same name *after* the including. Overriding fields will be encoded in their *original* places, irrelevant to the order of declaration.

```
class A1(TlvModel):
    m1 = UintField(0x01) # A1 = [M1]

class A2(A1):
    _a1 = IncludeBase(A1) # A2 = [M1] [M2]
    m2 = UintField(0x02)

class B1(TlvModel):
    x = UintField(0x0a) # B1 = [X] [A1] [Y]
    a = ModelField(0x03, A1)
    y = UintField(0x0b)

class B2(B1):
    IncludeBase(B1) # B2 = [X] [A2] [Y]
    a = ModelField(0x03, A2)
```

## Parsing

A TlvModel can be parsed from a wire. All fields are parsed in order. Out of order or unknown fields are ignored if they are non-critical. An unknown critical field leads to *DecodeError* .

```

from ndn.encoding import *

class Model(TlvModel):
    name = NameField()           # Model = [Name] [IntVal] [StrVal] [BoolVal]
    int_val = UintField(0x03)    # Name = NAME-TYPE TLV-LENGTH ...
    str_val = BytesField(0x02)  # IntVal = INT-VAL-TYPE TLV-LENGTH nonNegativeInteger
    bool_val = BoolField(0x01)  # StrVal = STR-VAL-TYPE TLV-LENGTH *OCTET
                                # BoolVal = BOOL-VAL-TYPE 0

model = Model.parse(b'\x07\x06\x08\x04name\x02\nbit string')
assert Name.to_str(model.name) == '/name'
assert model.str_val == b'bit string'

```

## Signature

Please contact the developer if you have to have a Signature field in your model.

## 1.9 Contribute and Support

- Please submit any changes via [GitHub](#) pull requests.
- If you use any existing code, please make sure it is compatible to *Apache v2*
- Ensure that your code complies to [PEP8](#).
- Ensure that your code works with *Python 3.6* and *PyPy 7.1.1*.
- Please contact the author first if your changes are not back compatible or import new dependencies.
- Write unit tests. There is no need to cover all code, but please cover as much as you can.
- Add your name to `AUTHORS.rst`.

## 1.10 Future plans

Future releases will include:

- Add more documentation
- Add more tests
- **Security tools:**
  - Schematized trust (including its necessary components such as validator, certificate storage, etc.)
  - Name-based Access Control
- **Application supports:**
  - Sync protocol
  - DLedger

## 1.11 Authors

- Xinyu Ma <<https://zjkmxy.github.io>>
- Zhaoning Kong <<https://jonnykong.com>>
- Eric Newberry <<https://ericnewberry.com>>
- Junxiao Shi <<https://yoursunny.com>>

## 1.12 Changelog

### 1.12.1 0.3a1-3 (2021-05-22)

- Support Unix socket on Windows 10.
- Fix semaphore running in a different event loop bug.

### 1.12.2 0.3a1-2 (2021-04-29)

- Handle ConnectionResetError.
- Drop Python 3.6 support.

### 1.12.3 0.3a1-1 (2021-01-31)

- Transfer the repo to named-data/python-ndn.
- Fix cocoapy to make it work on MacOS 11 Big Sur.
- Add more supports to NDNLv2 (CongestionMark).
- Add dispatcher and set\_interest\_filter.

### 1.12.4 0.3a1 (2020-09-24)

- Fix the bug that registering multiple prefixes at the same time leads to 403 error.
- Add Name Tree Schema.
- Add .devcontainer for VSCode Remote Containers and GitHub Codespaces.

### 1.12.5 0.2b2-2 (2020-05-26)

- Change the default sock file path from /var/run/nfd.sock to /run/nfd.sock on Linux.
- Add FIB and CS management data structures
- Add make\_network\_nack
- Recognize NDN\_CLIENT\_\* environment variables

### 1.12.6 0.2b2-1 (2020-03-23)

- Fix RuntimeWarning for hanging coroutine when main\_loop raises an exception.
- Fix the issue when after\_start throws an exception, the application gets stuck.
- Set raw\_packet of express\_interest and on\_interest to be the whole packet with TL fields.

### 1.12.7 0.2b2 (2020-02-18)

- Switch to Apache License 2.0.
- Add NDNApp.get\_original\_packet\_value.
- Improve NDNApp.route and NDNApp.express\_interest to give access the original packet and signature pointers of packets.
- Fix typos in the documentation.
- Support more alternate URI format of Name Component (seg, off, v, t and seq)
- Update Python version to 3.8 and add PyPy 7.2.0 in GitHub Action.
- Fix Name.to\_str so its output for [b'\x08\x00'] is correct.

### 1.12.8 0.2b1 (2019-11-20)

The initial release.





## INDICES AND TABLES

- genindex
- modindex
- search



## PYTHON MODULE INDEX

### n

- `ndn.app`, 3
- `ndn.encoding.name.Component`, 9
- `ndn.encoding.name.Name`, 12
- `ndn.encoding.ndn_format_0_3`, 18
- `ndn.encoding.tlv_model`, 14
- `ndn.encoding.tlv_type`, 7
- `ndn.encoding.tlv_var`, 8
- `ndn.schema.policy`, 34
- `ndn.schema.schema_tree`, 27
- `ndn.schema.simple_cache`, 36
- `ndn.schema.simple_node`, 32
- `ndn.schema.simple_trust`, 35
- `ndn.schema.util`, 31
- `ndn.security.keychain.keychain_digest`, 22
- `ndn.security.keychain.keychain_sqlite3`, 22
- `ndn.types`, 36
- `ndn.utils`, 37



## Symbols

`__eq__()` (*ndn.encoding.tlv\_model.TlvModel* method), 17  
`__get__()` (*ndn.encoding.tlv\_model.Field* method), 14  
`__get__()` (*ndn.encoding.tlv\_model.ProcedureArgument* method), 15  
`__set__()` (*ndn.encoding.tlv\_model.Field* method), 14  
`__set__()` (*ndn.encoding.tlv\_model.ProcedureArgument* method), 16

## A

`app()` (*ndn.schema.schema\_tree.MatchedNode* method), 27  
`asdict()` (*ndn.encoding.tlv\_model.TlvModel* method), 17  
`attach()` (*ndn.schema.schema\_tree.Node* method), 29

## B

`BinaryStr` (*in module ndn.encoding.tlv\_type*), 7  
`BoolField` (*class in ndn.encoding.tlv\_model*), 16  
`BytesField` (*class in ndn.encoding.tlv\_model*), 16

## C

`Cache` (*class in ndn.schema.policy*), 34  
`Certificate` (*class in ndn.security.keychain.keychain\_sqlite3*), 22  
`CHARSET` (*in module ndn.encoding.name.Component*), 9  
`ContentType` (*class in ndn.encoding.ndn\_format\_0\_3*), 18

## D

`DataEncryption` (*class in ndn.schema.policy*), 34  
`DataSigning` (*class in ndn.schema.policy*), 34  
`DataValidator` (*class in ndn.schema.policy*), 34  
`DecodeError`, 14  
`default_cert()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 23  
`default_identity()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 23  
`default_key()` (*ndn.security.keychain.keychain\_sqlite3.Identity* method), 22

`del_cert()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 23  
`del_cert()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 23  
`del_identity()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24  
`del_key()` (*ndn.security.keychain.keychain\_sqlite3.Identity* method), 22  
`del_key()` (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24  
`Delegation` (*class in ndn.encoding.ndn\_format\_0\_3*), 18

## E

`encode()` (*ndn.encoding.tlv\_model.TlvModel* method), 17  
`encode_into()` (*ndn.encoding.tlv\_model.Field* method), 14  
`encoded_length()` (*ndn.encoding.tlv\_model.Field* method), 15  
`encoded_length()` (*ndn.encoding.tlv\_model.TlvModel* method), 17  
`Encryption` (*class in ndn.schema.policy*), 34  
`escape_str()` (*in module ndn.encoding.name.Component*), 9  
`exist()` (*ndn.schema.schema\_tree.Node* method), 29  
`express()` (*ndn.schema.schema\_tree.MatchedNode* method), 27  
`express_interest()` (*ndn.app.NDNApp* method), 3

## F

`Field` (*class in ndn.encoding.tlv\_model*), 14  
`finer_match()` (*ndn.schema.schema\_tree.MatchedNode* method), 28  
`FormalName` (*in module ndn.encoding.tlv\_type*), 7  
`from_byte_offset()` (*in module ndn.encoding.name.Component*), 10  
`from_bytes()` (*in module ndn.encoding.name.Component*), 10  
`from_bytes()` (*in module ndn.encoding.name.Name*), 12  
`from_hex()` (*in module ndn.encoding.name.Component*), 10

- from\_number() (in module *ndn.encoding.name.Component*), 10  
 from\_segment() (in module *ndn.encoding.name.Component*), 10  
 from\_sequence\_num() (in module *ndn.encoding.name.Component*), 11  
 from\_str() (in module *ndn.encoding.name.Component*), 11  
 from\_str() (in module *ndn.encoding.name.Name*), 12  
 from\_timestamp() (in module *ndn.encoding.name.Component*), 11  
 from\_version() (in module *ndn.encoding.name.Component*), 11
- ## G
- gen\_nonce() (in module *ndn.utils*), 37  
 gen\_nonce\_64() (in module *ndn.utils*), 37  
 get\_arg() (*ndn.encoding.tlv\_model.ProcedureArgument* method), 16  
 get\_policy() (*ndn.schema.schema\_tree.Node* method), 29  
 get\_signature\_value\_size() (*ndn.encoding.Signer* method), 21  
 get\_signer() (*ndn.security.keychain.Keychain* method), 21  
 get\_signer() (*ndn.security.keychain.keychain\_digest.KeychainDigest* method), 22  
 get\_signer() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24  
 get\_tl\_num\_size() (in module *ndn.encoding.tlv\_var*), 8  
 get\_type() (in module *ndn.encoding.name.Component*), 11  
 get\_value() (in module *ndn.encoding.name.Component*), 11  
 get\_value() (*ndn.encoding.tlv\_model.Field* method), 15
- ## H
- has\_default\_cert() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 23  
 has\_default\_identity() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24  
 has\_default\_key() (*ndn.security.keychain.keychain\_sqlite3.Identity* method), 22
- ## I
- Identity (class in *ndn.security.keychain.keychain\_sqlite3*), 22  
 IncludeBase (class in *ndn.encoding.tlv\_model*), 14  
 IncludeBaseError, 14  
 InterestCanceled, 36  
 InterestEncryption (class in *ndn.schema.policy*), 34  
 InterestNack, 36  
 InterestParam (class in *ndn.encoding.ndn\_format\_0\_3*), 18  
 InterestSigning (class in *ndn.schema.policy*), 35  
 InterestTimeout, 36  
 InterestValidator (class in *ndn.schema.policy*), 35  
 is\_binary\_str() (in module *ndn.encoding.tlv\_type*), 8  
 is\_prefix() (in module *ndn.encoding.name.Name*), 13
- ## K
- Key (class in *ndn.security.keychain.keychain\_sqlite3*), 23  
 Keychain (class in *ndn.security.keychain*), 21  
 KeychainDigest (class in *ndn.security.keychain.keychain\_digest*), 22  
 KeychainSqlite3 (class in *ndn.security.keychain.keychain\_sqlite3*), 23  
 KeyLocator (class in *ndn.encoding.ndn\_format\_0\_3*), 18
- ## L
- Links (class in *ndn.encoding.ndn\_format\_0\_3*), 18  
 LocalOnly (class in *ndn.schema.policy*), 35  
 LocalResource (class in *ndn.schema.simple\_node*), 32  
 LocalResourceNotExistError, 27
- ## M
- MainDigest (class in *ndn.encoding.ndn\_format\_0\_3*), 19  
 main\_loop() (*ndn.app.NDNApp* method), 4  
 make\_data() (in module *ndn.encoding.ndn\_format\_0\_3*), 19  
 make\_interest() (in module *ndn.encoding.ndn\_format\_0\_3*), 19  
 make\_metadata() (*ndn.schema.simple\_node.RDRNode.MetaData* method), 32  
 match() (*ndn.schema.schema\_tree.Node* method), 30  
 MatchedNode (class in *ndn.schema.schema\_tree*), 27  
 MemoryCache (class in *ndn.schema.simple\_cache*), 36  
 MemoryCachePolicy (class in *ndn.schema.simple\_cache*), 36  
 MetaInfo (class in *ndn.encoding.ndn\_format\_0\_3*), 19  
 MetadataField (class in *ndn.encoding.tlv\_model*), 16  
 module  
 ndn.app, 3  
 ndn.encoding.name.Component, 9  
 ndn.encoding.name.Name, 12  
 ndn.encoding.ndn\_format\_0\_3, 18  
 ndn.encoding.tlv\_model, 14  
 ndn.encoding.tlv\_type, 7  
 ndn.encoding.tlv\_var, 8  
 ndn.schema.policy, 34  
 ndn.schema.schema\_tree, 27  
 ndn.schema.simple\_cache, 36  
 ndn.schema.simple\_node, 32  
 ndn.schema.simple\_trust, 35  
 ndn.schema.util, 31

ndn.security.keychain.keychain\_digest, 22  
 ndn.security.keychain.keychain\_sqlite3,  
 22  
 ndn.types, 36  
 ndn.utils, 37

## N

NameField (class in ndn.encoding.tlv\_model), 16  
 NamePattern (in module ndn.schema.util), 31  
 ndn.app  
 module, 3  
 ndn.encoding.name.Component  
 module, 9  
 ndn.encoding.name.Name  
 module, 12  
 ndn.encoding.ndn\_format\_0\_3  
 module, 18  
 ndn.encoding.tlv\_model  
 module, 14  
 ndn.encoding.tlv\_type  
 module, 7  
 ndn.encoding.tlv\_var  
 module, 8  
 ndn.schema.policy  
 module, 34  
 ndn.schema.schema\_tree  
 module, 27  
 ndn.schema.simple\_cache  
 module, 36  
 ndn.schema.simple\_node  
 module, 32  
 ndn.schema.simple\_trust  
 module, 35  
 ndn.schema.util  
 module, 31  
 ndn.security.keychain.keychain\_digest  
 module, 22  
 ndn.security.keychain.keychain\_sqlite3  
 module, 22  
 ndn.types  
 module, 36  
 ndn.utils  
 module, 37  
 NDNApp (class in ndn.app), 3  
 need() (ndn.schema.schema\_tree.MatchedNode  
 method), 28  
 need() (ndn.schema.schema\_tree.Node method), 30  
 need() (ndn.schema.simple\_node.LocalResource  
 method), 32  
 need() (ndn.schema.simple\_node.RDRNode method), 33  
 need() (ndn.schema.simple\_node.RDRNode.MetaData  
 method), 32  
 need() (ndn.schema.simple\_node.SegmentedNode  
 method), 33

NetworkError, 36  
 new\_identity() (ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3  
 method), 24  
 new\_key() (ndn.security.keychain.keychain\_sqlite3.Identity  
 method), 22  
 new\_key() (ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3  
 method), 24  
 Node (class in ndn.schema.schema\_tree), 29  
 NodeExistsError, 31  
 NonStrictName (in module ndn.encoding.tlv\_type), 7  
 norm\_pattern() (in module ndn.schema.util), 31  
 normalize() (in module ndn.encoding.name.Name), 13

## O

OffsetMarker (class in ndn.encoding.tlv\_model), 16  
 on\_data() (ndn.schema.schema\_tree.MatchedNode  
 method), 28  
 on\_interest() (ndn.schema.schema\_tree.MatchedNode  
 method), 28  
 on\_register() (ndn.schema.schema\_tree.Node  
 method), 30  
 on\_register() (ndn.schema.simple\_node.LocalResource  
 method), 32

## P

pack\_uint\_bytes() (in module ndn.encoding.tlv\_var),  
 8  
 parse() (ndn.encoding.tlv\_model.TlvModel class  
 method), 18  
 parse\_and\_check\_tl() (in module  
 ndn.encoding.tlv\_var), 8  
 parse\_data() (in module  
 ndn.encoding.ndn\_format\_0\_3), 20  
 parse\_from() (ndn.encoding.tlv\_model.Field method),  
 15  
 parse\_interest() (in module  
 ndn.encoding.ndn\_format\_0\_3), 20  
 parse\_tl\_num() (in module ndn.encoding.tlv\_var), 8  
 Policy (class in ndn.schema.policy), 35  
 prepare\_data() (ndn.app.NDNApp method), 4  
 ProcedureArgument (class in ndn.encoding.tlv\_model),  
 15  
 process\_data() (ndn.schema.schema\_tree.Node  
 method), 30  
 process\_int() (ndn.schema.schema\_tree.Node  
 method), 30  
 process\_int() (ndn.schema.simple\_node.RDRNode.MetaData  
 method), 33  
 process\_int() (ndn.schema.simple\_node.SegmentedNode  
 method), 34  
 provide() (ndn.schema.schema\_tree.MatchedNode  
 method), 28  
 provide() (ndn.schema.schema\_tree.Node method), 31

- provide() (*ndn.schema.simple\_node.LocalResource* method), 32
- provide() (*ndn.schema.simple\_node.RDRNode* method), 33
- provide() (*ndn.schema.simple\_node.SegmentedNode* method), 34
- put\_data() (*ndn.app.NDNApp* method), 4
- put\_data() (*ndn.schema.schema\_tree.MatchedNode* method), 29
- put\_raw\_packet() (*ndn.app.NDNApp* method), 4
- ## R
- RDRNode (class in *ndn.schema.simple\_node*), 32
- RDRNode.MetaData (class in *ndn.schema.simple\_node*), 32
- RDRNode.MetaDataValue (class in *ndn.schema.simple\_node*), 33
- read\_tl\_num\_from\_stream() (in module *ndn.encoding.tlv\_var*), 8
- Register (class in *ndn.schema.policy*), 35
- register() (*ndn.app.NDNApp* method), 5
- RepeatedField (class in *ndn.encoding.tlv\_model*), 17
- Route (in module *ndn.types*), 36
- route() (*ndn.app.NDNApp* method), 5
- run\_forever() (*ndn.app.NDNApp* method), 6
- ## S
- save() (*ndn.schema.simple\_cache.MemoryCache* method), 36
- search() (*ndn.schema.simple\_cache.MemoryCache* method), 36
- SegmentedNode (class in *ndn.schema.simple\_node*), 33
- set\_arg() (*ndn.encoding.tlv\_model.ProcedureArgument* method), 16
- set\_default\_cert() (*ndn.security.keychain.keychain\_sqlite3.Key* method), 23
- set\_default\_identity() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24
- set\_default\_key() (*ndn.security.keychain.keychain\_sqlite3.Identity* method), 23
- set\_interest\_filter() (*ndn.app.NDNApp* method), 6
- set\_policy() (*ndn.schema.schema\_tree.Node* method), 31
- shutdown() (*ndn.app.NDNApp* method), 6
- shutdown() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 24
- SignatureInfo (class in *ndn.encoding.ndn\_format\_0\_3*), 19
- SignaturePtrs (class in *ndn.encoding.ndn\_format\_0\_3*), 19
- SignatureType (class in *ndn.encoding.ndn\_format\_0\_3*), 19
- SignedBy (class in *ndn.schema.simple\_trust*), 35
- Signer (class in *ndn.encoding*), 21
- Signing (class in *ndn.schema.policy*), 35
- skipping\_process() (*ndn.encoding.tlv\_model.Field* method), 15
- ## T
- timestamp() (in module *ndn.utils*), 37
- TlvModel (class in *ndn.encoding.tlv\_model*), 17
- TlvModelMeta (class in *ndn.encoding.tlv\_model*), 17
- to\_bytes() (in module *ndn.encoding.name.Name*), 13
- to\_number() (in module *ndn.encoding.name.Component*), 12
- to\_str() (in module *ndn.encoding.name.Component*), 12
- to\_str() (in module *ndn.encoding.name.Name*), 13
- touch\_identity() (*ndn.security.keychain.keychain\_sqlite3.KeychainSqlite3* method), 25
- TYPE\_NAME (in module *ndn.encoding.name.Name*), 12
- TypeName (class in *ndn.encoding.ndn\_format\_0\_3*), 19
- ## U
- UIntField (class in *ndn.encoding.tlv\_model*), 16
- unregister() (*ndn.app.NDNApp* method), 6
- unset\_interest\_filter() (*ndn.app.NDNApp* method), 6
- ## V
- ValidationFailure, 37
- Validator (in module *ndn.types*), 37
- VarBinaryStr (in module *ndn.encoding.tlv\_type*), 8
- ## W
- write\_signature\_info() (*ndn.encoding.Signer* method), 21
- write\_signature\_value() (*ndn.encoding.Signer* method), 21
- write\_tl\_num() (in module *ndn.encoding.tlv\_var*), 9