python-ipfix Documentation

Release 0.9

Brian Trammell

July 02, 2013

CONTENTS

IPFIX implementation for Python 3.3.

This module provides a Python interface to IPFIX message streams, and provides tools for building IPFIX Exporting and Collecting Processes. It handles message framing and deframing, encoding and decoding IPFIX data records using templates, and a bridge between IPFIX ADTs and appropriate Python data types.

Before using any of the functions of this module, it is necessary to populate the information model with Information Elements. `ipfix.ie.use_iana_default()` populates the default IANA IPFIX Information Element Registry shipped with the module; this is the current registry as of release time. `ipfix.ie.use_5103_default()` populates the reverse counterpart IEs as in **RFC 5103**. The module also supports the definition of enterprise-specific Information Elements via `ipfix.ie.for_spec()` and `ipfix.ie.use_specfile()`; see `ipfix.ie` for more.

For reading and writing of records to IPFIX message streams with automatic message boundary management, see the `ipfix.reader` and `ipfix.writer` modules, respectively. For manual reading and writing of messages, see `ipfix.message`. In any case, exporters will need to define templates; see `ipfix.template`.

Warning: Variable-length IEs are ostensibly supported by this module, but have not been tested as of this release.

Reference documentation for each module is found in the subsections below.

CHAPTER

ONE

MODULE IPFIX.TYPES

Implementation of IPFIX abstract data types (ADT) and mappings to Python types.

Maps each IPFIX ADT to the corresponding Python type, as below:

IPFIX Type	Python Type
octetArray	bytes
unsigned8	int
unsigned16	int
unsigned32	int
unsigned64	int
signed8	int
signed16	int
signed32	int
signed64	int
float32	float
float64	float
boolean	bool
macAddress	bytes
string	str
dateTimeSeconds	datetime
dateTimeMilliseconds	datetime
dateTimeMicroseconds	datetime
dateTimeNanoseconds	datetime
ipv4Address	ipaddress
ipv6Address	ipaddress

Though client code generally will not use this module directly, it defines how each IPFIX abstract data type will be represented in Python, and the concrete IPFIX representation of each type. Type methods operate on buffers, as used internally by the `ipfix.message.MessageBuffer` class, so we'll create one to illustrate encoding and decoding:

```
>>> import ipfix.types
>>> buf = memoryview(bytearray(16))
```

Each of the encoding methods returns the offset into the buffer of the first byte after the encoded value; since we're always encoding to the beginning of the buffer in this example, this is equivalent to the length. We use this to bound the encoded value on subsequent decode.

Integers are represented by the python int type:

```
>>> unsigned32 = ipfix.types.for_name("unsigned32")
>>> length = unsigned32.encode_single_value_to(42, buf, 0)
>>> buf[0:length].tolist()
```

```
[0, 0, 0, 42]
>>> unsigned32.decode_single_value_from(buf, 0, length)
42
```

...floats by the float type, with the usual caveats about precision:

```
>>> float32 = ipfix.types.for_name("float32")
>>> length = float32.encode_single_value_to(42.03579, buf, 0)
>>> buf[0:length].tolist()
[66, 40, 36, 166]
>>> float32.decode_single_value_from(buf, 0, length)
42.035789489746094
```

...strings by the str type, encoded as UTF-8:

```
>>> string = ipfix.types.for_name("string")
>>> length = string.encode_single_value_to("Grüezi", buf, 0)
>>> buf[0:length].tolist()
[71, 114, 195, 188, 101, 122, 105]
>>> string.decode_single_value_from(buf, 0, length)
'Grüezi'
```

...addresses as the IPv4Address and IPv6Address types in the ipaddress module:

```
>>> from ipaddress import ip_address
>>> ipv4Address = ipfix.types.for_name("ipv4Address")
>>> length = ipv4Address.encode_single_value_to(ip_address("198.51.100.27"), buf, 0)
>>> buf[0:length].tolist()
[198, 51, 100, 27]
>>> ipv4Address.decode_single_value_from(buf, 0, length)
IPv4Address('198.51.100.27')
>>> ipv6Address = ipfix.types.for_name("ipv6Address")
>>> length = ipv6Address.encode_single_value_to(ip_address("2001:db8::c0:ffee"), buf, 0)
>>> buf[0:length].tolist()
[32, 1, 13, 184, 0, 0, 0, 0, 0, 0, 0, 0, 0, 192, 255, 238]
>>> ipv6Address.decode_single_value_from(buf, 0, length)
IPv6Address('2001:db8::c0:ffee')
```

...and the timestamps of various precision as a python datetime, encoded as per RFC5101bis:

```
>>> from datetime import datetime
>>> dtfmt_in = "%Y-%m-%d %H:%M:%S.%f %z"
>>> dtfmt_out = "%Y-%m-%d %H:%M:%S.%f"
>>> dt = datetime.strptime("2013-06-21 14:00:03.456789 +0000", dtfmt_in)
```

dateTimeSeconds truncates microseconds:

```
>>> dateTimeSeconds = ipfix.types.for_name("dateTimeSeconds")
>>> length = dateTimeSeconds.encode_single_value_to(dt, buf, 0)
>>> buf[0:length].tolist()
[81, 196, 92, 99]
>>> dateTimeSeconds.decode_single_value_from(buf, 0, length).strftime(dtfmt_out)
'2013-06-21 14:00:03.000000'
```

dateTimeMilliseconds truncates microseconds to the nearest millisecond:

```
>>> dateTimeMilliseconds = ipfix.types.for_name("dateTimeMilliseconds")
>>> length = dateTimeMilliseconds.encode_single_value_to(dt, buf, 0)
>>> buf[0:length].tolist()
[0, 0, 1, 63, 103, 8, 228, 128]
```

```
>>> dateTimeMilliseconds.decode_single_value_from(buf, 0, length).strftime(dtfmt_out)
'2013-06-21 14:00:03.456000'
```

dateTimeMicroseconds exports microseconds fully in NTP format:

```
>>> dateTimeMicroseconds = ipfix.types.for_name("dateTimeMicroseconds")
>>> length = dateTimeMicroseconds.encode_single_value_to(dt, buf, 0)
>>> buf[0:length].tolist()
[81, 196, 92, 99, 116, 240, 32, 0]
>>> dateTimeMicroseconds.decode_single_value_from(buf, 0, length).strftime(dtfmt_out)
'2013-06-21 14:00:03.456789'
```

dateTimeNanoseconds is also supported, but is identical to dateTimeMicroseconds, as the datetime class in Python only supports microsecond-level timing.

class `ipfix.types.`**`IpfixType`**(*name*, *num*, *valenc*, *valdec*)
Abstract interface for all IPFIX types. Used internally.

exception `ipfix.types.`**`IpfixTypeError`**(**args*)
Raised when attempting to do an unsupported operation on a type

class `ipfix.types.`**`OctetArrayType`**(*name*, *num*, *valenc=<function _identity at 0x103aa8320>*, *valdec=<function _identity at 0x103aa8320>*)
Type encoded by byte array packing. Used internally.

class `ipfix.types.`**`StructType`**(*name*, *num*, *stel*, *valenc=<function _identity at 0x103aa8320>*, *valdec=<function _identity at 0x103aa8320>*)
Type encoded by struct packing. Used internally.

`ipfix.types.`**`decode_varlen`**(*buf*, *offset*)
Decode a IPFIX varlen encoded length; used internally by template

`ipfix.types.`**`encode_varlen`**(*buf*, *offset*, *length*)
Encode a IPFIX varlen encoded length; used internally by template

`ipfix.types.`**`for_name`**(*name*)
Return an IPFIX type for a given type name

Parameters **name** – the name of the type to look up

Returns IpfixType – type instance for that name

Raises IpfixTypeError

CHAPTER

TWO

MODULE IPFIX.IE

IESpec-based interface to IPFIX information elements, and interface to use the default IPFIX IANA Information Model

An IESpec is a string representation of an IPFIX information element, including all the information required to define it, as documented in Section 9 of http://tools.ietf.org/html/draft-ietf-ipfix-ie-doctors. It has the format:

> name(pen/num)<type>[size]

To specify a new Information Element, a complete IESpec must be passed to for_spec():

```
>>> import ipfix.ie
>>> e = ipfix.ie.for_spec("myNewInformationElement(35566/1)<string>")
>>> e
InformationElement('myNewInformationElement', 35566, 1, ipfix.types.for_name('string'), 65535)
```

The string representation of an InformationElement is its IESpec:

```
>>> str(e)
'myNewInformationElement(35566/1)<string>[65535]'
```

To get an Information Element already specified, an incomplete specification can be passed; a name or number is enough:

```
>>> ipfix.ie.use_iana_default()
>>> str(ipfix.ie.for_spec("octetDeltaCount"))
'octetDeltaCount(0/1)<unsigned64>[8]'
>>> str(ipfix.ie.for_spec("(2)"))
'packetDeltaCount(0/2)<unsigned64>[8]'
```

Reduced-length encoding and fixed-length sequence types are supported by the for_length method; this is used internally by templates.

```
>>> str(e.for_length(32))
'myNewInformationElement(35566/1)<string>[32]'
```

Most client code will only need the `use_iana_default()`, `use_5103_default()`, and `use_specfile()` functions; client code using tuple interfaces will need `spec_list()` as well.

class `ipfix.ie.`**`InformationElement`**(*name*, *pen*, *num*, *ietype*, *length*)

An IPFIX Information Element (IE). This is essentially a five-tuple of name, element number (num), a private enterprise number (pen; 0 if it is an IANA registered IE), a type, and a length.

InformationElement instances should be obtained using the `for_spec()` or `for_template_entry()` functions.

`for_length`(*length*)

Get an instance of this IE for the specified length. Used to support reduced-length encoding (RLE).

Parameters **length** – length of the new IE

Returns this IE if length matches, or a new IE for the length

Raises ValueError

class `ipfix.ie.`**`InformationElementList`**(*iterable=None*)

A hashable ordered list of Information Elements.

Used internally by templates, and to specify the order of tuples to the tuple append and iterator interfaces. Get an instance by calling `spec_list()`

`ipfix.ie.`**`clear_infomodel`**()

Reset the cache of known Information Elements.

`ipfix.ie.`**`for_spec`**(*spec*)

Get an IE from the cache of known IEs, or create a new IE if not found, given an IESpec.

Parameters **spec** – IEspec, as in draft-ietf-ipfix-ie-doctors, of the form name(pen/num)<type>[size]; some fields may be omitted unless creating a new IE in the cache.

Returns an IE for the name

Raises ValueError

`ipfix.ie.`**`for_template_entry`**(*pen*, *num*, *length*)

Get an IE from the cache of known IEs, or create a new IE if not found, given a private enterprise number, element number, and length. Used internally by Templates.

Parameters

- **pen** – private enterprise number, or 0 for an IANA IE
- **num** – IE number (Element ID)
- **length** – length of the IE in bytes

Returns an IE for the given pen, num, and length. If the IE has not been previously added to the cache of known IEs, the IE will be named _ipfix_pen_num, and have octetArray as a type.

`ipfix.ie.`**`parse_spec`**(*spec*)

Parse an IESpec into name, pen, number, typename, and length fields

`ipfix.ie.`**`spec_list`**(*specs*)

Given a list or iterable of IESpecs, return a hashable list of IEs. Pass this as the ielist argument to the tuple export and iterator functions.

Parameters **specs** – list of IESpecs

Returns a new Information Element List, suitable for use with the tuple export and iterator functions in `message`

Raises ValueError

`ipfix.ie.`**`use_5103_default`**()

Load the module internal list of RFC 5103 reverse IEs for IANA registered IEs into the cache of known IEs. Normally, biflow-aware client code should call this just after use_iana_default().

`ipfix.ie.`**`use_iana_default`**()

Load the module internal list of IANA registered IEs into the cache of known IEs. Normally, client code should call this before using any other part of this module.

`ipfix.ie.`**`use_specfile`**(*filename*)

Load a file listing IESpecs into the cache of known IEs

Parameters **filename** – name of file containing IESpecs to open

Raises ValueError

CHAPTER

THREE

MODULE IPFIX.TEMPLATE

Representation of IPFIX templates. Provides template-based packing and unpacking of data in IPFIX messages.

For reading, templates are handled internally. For writing, use `from_ielist()` to create a template.

See `ipfix.message` for examples.

exception `ipfix.template.`**`IpfixDecodeError`**(**args*)
Raised when decoding a malformed IPFIX message

exception `ipfix.template.`**`IpfixEncodeError`**(**args*)
Raised on internal encoding errors, or if message MTU is too small

class `ipfix.template.`**`Template`**(*tid=0*, *iterable=None*)
An IPFIX Template.

A template is an ordered list of IPFIX Information Elements with an ID.

`append`(*ie*)
Append an IE to this Template

`count`()
Count IEs in this template

`decode_from`(*buf*, *offset*, *packplan=None*)
Decodes a record into a tuple containing values in template order

`decode_iedict_from`(*buf*, *offset*, *recinf=None*)
Decodes a record from a buffer into a dict keyed by IE

`decode_namedict_from`(*buf*, *offset*, *recinf=None*)
Decodes a record from a buffer into a dict keyed by IE name.

`decode_tuple_from`(*buf*, *offset*, *recinf=None*)
Decodes a record from a buffer into a tuple, ordered as the IEs in the InformationElementList given as recinf.

`encode_iedict_to`(*buf*, *offset*, *rec*, *recinf=None*)
Encodes a record from a dict containing values keyed by IE

`encode_namedict_to`(*buf*, *offset*, *rec*, *recinf=None*)
Encodes a record from a dict containing values keyed by IE name

`encode_template_to`(*buf*, *offset*, *setid*)
Encodes the template to a buffer. Encodes as a Template if setid is TEMPLATE_SET_ID, as an Options Template if setid is OPTIONS_SET_ID.

`encode_to`(*buf*, *offset*, *vals*, *packplan=None*)
Encodes a record from a tuple containing values in template order

encode_tuple_to(*buf*, *offset*, *rec*, *recinf=None*)
Encodes a record from a tuple containing values ordered as the IEs in the InformationElementList given as recinf. If recinf is not given, assumes the tuple contains all IEs in the template in template order.

finalize()
Compile a default packing plan. Called after append()ing all IEs.

fixlen_count()
Count of fixed-length IEs in this template before the first variable-length IE; this is the size of the portion of the template which can be encoded/decoded efficiently.

packplan_for_ielist(**args*, ***kwds*)
Given a list of IEs, devise and cache a packing plan. Used by the tuple interfaces.

class ipfix.template.**TemplatePackingPlan**(*tmpl*, *indices*)
Plan to pack/unpack a specific set of indices for a template. Used internally by Templates for efficient encoding and decoding.

ipfix.template.**decode_template_from**(*buf*, *offset*, *setid*)
Decodes a template from a buffer. Decodes as a Template if setid is TEMPLATE_SET_ID, as an Options Template if setid is OPTIONS_SET_ID.

ipfix.template.**from_ielist**(*tid*, *ielist*)
Create a template from a template ID and an information element list (itself available from `ipfix.ie.spec_list()`).

Parameters

- **tid** – Template ID, must be between 256 and 65535.
- **ielist** – List of Information Elements for the Template, see `ipfix.ie.spec_list()`.

Returns A new Template, ready to use for writing to a Message

CHAPTER

FOUR

MODULE IPFIX.MESSAGE

Provides the MessageBuffer class for encoding and decoding IPFIX Messages.

This interface allows direct control over Messages; for reading or writing records automatically from/to streams, see `ipfix.reader` and `ipfix.writer`, respectively.

To create a message buffer:

```
>>> import ipfix.message
>>> msg = ipfix.message.MessageBuffer()
>>> msg
<MessageBuffer domain 0 length 0>
```

To prepare the buffer to write records:

```
>>> msg.begin_export(8304)
>>> msg
<MessageBuffer domain 8304 length 16 (writing)>
```

Note that the buffer grows to contain the message header.

To write records to the buffer, first you'll need a template:

```
>>> import ipfix.ie
>>> ipfix.ie.use_iana_default()
>>> import ipfix.template
>>> tmpl = ipfix.template.from_ielist(256,
...        ipfix.ie.spec_list(("flowStartMilliseconds",
...                            "sourceIPv4Address",
...                            "destinationIPv4Address",
...                            "packetDeltaCount")))
>>> tmpl
<Template ID 256 count 4 scope 0>
```

To add the template to the message:

```
>>> msg.add_template(tmpl)
>>> msg
<MessageBuffer domain 8304 length 40 (writing set 2)>
```

Note that `MessageBuffer.add_template()` exports the template when it is written by default, and that the current set ID is 2 (template set).

Now, a set must be created to add records to the message; the set ID must match the ID of the template. MessageBuffer automatically uses the template matching the set ID for record encoding.

```
>>> msg.export_ensure_set(256)
>>> msg
<MessageBuffer domain 8304 length 44 (writing set 256)>
```

Records can be added to the set either as dictionaries keyed by IE name:

```
>>> from datetime import datetime
>>> from ipaddress import ip_address
>>> rec = { "flowStartMilliseconds" : datetime.strptime("2013-06-21 14:00:00",
...                                                     "%Y-%m-%d %H:%M:%S"),
...         "sourceIPv4Address" : ip_address("10.1.2.3"),
...         "destinationIPv4Address" : ip_address("10.5.6.7"),
...         "packetDeltaCount" : 27 }
>>> msg.export_namedict(rec)
>>> msg
<MessageBuffer domain 8304 length 68 (writing set 256)>
```

or as tuples in template order:

```
>>> rec = (datetime.strptime("2013-06-21 14:00:02", "%Y-%m-%d %H:%M:%S"),
...        ip_address("10.8.9.11"), ip_address("10.12.13.14"), 33)
>>> msg.export_tuple(rec)
>>> msg
<MessageBuffer domain 8304 length 92 (writing set 256)>
```

Variable-length information elements will be encoded using the native length of the passed value:

```
>>> ipfix.ie.for_spec("myNewInformationElement(35566/1)<string>")
InformationElement('myNewInformationElement', 35566, 1, ipfix.types.for_name('string'), 65535)
>>> tmpl = ipfix.template.from_ielist(257,
...        ipfix.ie.spec_list(("flowStartMilliseconds",
...                            "myNewInformationElement")))
>>> msg.add_template(tmpl)
>>> msg.export_ensure_set(257)
>>> msg
<MessageBuffer domain 8304 length 116 (writing set 257)>
>>> rec = { "flowStartMilliseconds" : datetime.strptime("2013-06-21 14:00:04",
...                                   "%Y-%m-%d %H:%M:%S"),
...         "myNewInformationElement" : "Grüezi, Y'all" }
>>> msg.export_namedict(rec)
>>> msg
<MessageBuffer domain 8304 length 139 (writing set 257)>
```

Attempts to write past the end of the message (set via the mtu parameter, default 65535) result in `EndOfMessage` being raised.

Messages can be written to a stream using `MessageBuffer.write_message()`, or dumped to a byte array for transmission using `MessageBuffer.to_bytes()`. The message must be reset before starting to write again.

```
>>> b = msg.to_bytes()
>>> msg.begin_export()
>>> msg
<MessageBuffer domain 8304 length 16 (writing)>
```

Reading happens more or less in reverse. To begin, a message is read from a byte array using `MessageBuffer.from_bytes()`, or from a stream using `MessageBuffer.read_message()`.

```
>>> msg.from_bytes(b)
>>> msg
<MessageBuffer domain 8304 length 139 (deframed 4 sets)>
```

Both of these methods scan the message in advance to find the sets within the message. The records within these sets can then be accessed by iterating over the message. As with export, the records can be accessed as a dictionary mapping IE names to values or as tuples. The dictionary interface is designed for general IPFIX processing applications, such as collectors accepting many types of data, or diagnostic tools for debugging IPFIX export:

```
>>> for rec in msg.namedict_iterator():
...     print(sorted(rec.items()))
...
[('destinationIPv4Address', IPv4Address('10.5.6.7')), ('flowStartMilliseconds', datetime.datetime(201
[('destinationIPv4Address', IPv4Address('10.12.13.14')), ('flowStartMilliseconds', datetime.datetime
[('flowStartMilliseconds', datetime.datetime(2013, 6, 21, 12, 0, 4)), ('myNewInformationElement', "Gr
```

The tuple interface for reading messages is designed for applications with a specific internal data model. It can be much faster than the dictionary interface, as it skips decoding of IEs not requested by the caller, and can skip entire sets not containing all the requested IEs. Requested IEs are specified as an `ipfix.ie.InformationElementList` instance, from `ie.spec_list()`:

```
>>> ielist = ipfix.ie.spec_list(["flowStartMilliseconds", "packetDeltaCount"])
>>> for rec in msg.tuple_iterator(ielist):
...     print(rec)
...
(datetime.datetime(2013, 6, 21, 12, 0), 27)
(datetime.datetime(2013, 6, 21, 12, 0, 2), 33)
```

Notice that the variable-length record written to the message are not returned by this iterator, since that record doesn't include a packetDeltaCount IE. The record is, however, still there:

```
>>> ielist = ipfix.ie.spec_list(["myNewInformationElement"])
>>> for rec in msg.tuple_iterator(ielist):
...     print(rec)
...
("Grüezi, Y'all",)
```

exception `ipfix.message.`**`EndOfMessage`**(**args*)

Exception raised when a write operation on a Message fails because there is not enough space in the message.

class `ipfix.message.`**`MessageBuffer`**

Implements a buffer for reading or writing IPFIX messages.

`active_template_ids`()

Get an iterator over all active template IDs in the current domain. Provided to allow callers to export some or all active Templates across multiple Messages.

Returns a template ID iterator

`add_template`(*tmpl*, *export=True*)

Add a template to this MessageBuffer. Adding a template makes it available for use for exporting records; see `export_new_set()`.

Parameters

- **tmpl** – the template to add
- **export** – If True, export this template to the MessageBuffer after adding it.

Raises EndOfMessage

`begin_export`(*odid=None*)

Start exporting a new message. Clears any previous message content, but keeps template information intact. Sets the message sequence number.

Parameters **odid** – Observation domain ID to use for export. By default, uses the observation domain ID of the previous message. Note that templates are scoped to observation domain, so templates will need to be added after switching to a new observation domain ID.

Raises IpfixEncodeError

delete_template(*tid*, *export=True*)

Delete a template by ID from this MessageBuffer.

Parameters

- **tid** – ID of the template to delete
- **export** – if True, export a Template Withdrawal for this Template after deleting it

Raises EndOfMessage

export_ensure_set(*setid*)

Ensure that the current set for export has the given Set ID. Starts a new set if not using `export_new_set()`

Parameters **setid** – Set ID of the new Set; corresponds to the Template ID of the Template that will be used to encode records into the Set. The require Template must have already been added to the MessageBuffer, see `add_template()`.

Raises IpfixEncodeError, EndOfMessage

export_namedict(*rec*)

Export a record to the message, using the template for the current Set ID. The record is a dictionary mapping IE names to values. The dictionary must contain a value for each IE in the template. Keys in the dictionary not in the template will be ignored.

Parameters **rec** – the record to export, as a dictionary

Raises EndOfMessage

export_needs_flush()

True if content has been written to this MessageBuffer since the last call to `begin_export()`

export_new_set(*setid*)

Start exporting a new Set with the given set ID. Creates a new set even if the current Set has the given set ID; client code should in most cases use `export_ensure_set()` instead.

Parameters **setid** – Set ID of the new Set; corresponds to the Template ID of the Template that will be used to encode records into the Set. The require Template must have already been added to the MessageBuffer, see `add_template()`.

Raises IpfixEncodeError, EndOfMessage

export_record(*rec*, *encode_fn=<function Template.encode_namedict_to at 0x103ad84d0>*, *recinf=None*)

Low-level interface to record export.

Export a record to a MessageBuffer, using the template associated with the Set ID given to the most recent `export_new_set()` or `export_ensure_set()` call, and the given encode function. By default, the record is assumed to be a dictionary mapping IE names to values (i.e., the same as `export_namedict()`).

Parameters

- **encode_fn** – Function used to encode a record; must be an (unbound) "encode" instance method of the `ipfix.template.Template` class.
- **recinf** – Record information opaquely passed to decode function

Raises EndOfMessage

export_template(*tid*)

Export a template to this Message given its template ID.

Parameters **tid** – ID of template to export; must have been added to this message previously with `add_template()`.

Raises EndOfMessage, KeyError

export_tuple(*rec*, *ielist=None*)

Export a record to the message, using the template for the current Set ID. The record is a tuple of values, in template order by default. If ielist is given, the tuple is in the order if IEs in that list instead. The tuple must contain one value for each IE in the template; values for IEs in the ielist not in the template will be ignored.

Parameters

- **rec** – the record to export, as a tuple
- **ielist** – optional information element list describing the order of the rec tuple

Raises EndOfMessage

from_bytes(*bytes*)

Read an IPFIX message from a byte array.

This populates message header fields and the internal setlist. Call for each new message before iterating over records when reading from a byte array.

Parameters **bytes** – a byte array containing a complete IPFIX message.

Raises IpfixDecodeError

get_export_time()

Return the export time of this message. When reading, returns the export time as read from the message header. When writing, this is the argument of the last call to `set_export_time()`, or, if :attr:auto_export_time is True, the time of the last message export.

Returns export time of the last message read/written.

namedict_iterator()

Iterate over all records in the Message, as dicts mapping IE names to values.

Returns a name dictionary iterator

read_message(*stream*)

Read a IPFIX message from a stream.

This populates message header fields and the internal setlist. Call for each new message before iterating over records when reading from a stream.

Parameters **stream** – stream to read from

Raises IpfixDecodeError

record_iterator(*decode_fn=<function Template.decode_namedict_from at 0x103ad8290>*, *tmplaccept_fn=<function accept_all_templates at 0x103adecb0>*, *recinf=None*)

Low-level interface to record iteration.

Iterate over records in an IPFIX message previously read with `read_message()` or `from_bytes()`. Automatically handles templates in set order. By default, iterates over each record in the stream as a dictionary mapping IE name to value (i.e., the same as `namedict_iterator()`)

Parameters

- **decode_fn** – Function used to decode a record; must be an (unbound) "decode" instance method of the `ipfix.template.Template` class.
- **tmplaccept_fn** – Function returning True if the given template is of interest to the caller, False if not. Default accepts all templates. Sets described by templates for which this function returns False will be skipped.
- **recinf** – Record information opaquely passed to decode function

Returns an iterator over records decoded by decode_fn.

set_export_time(*dt=None*)

Set the export time for the next message written with `write_message()` or `to_bytes()`. Disables automatic export time updates. By default, sets the export time to the current time.

Parameters **dt** – export time to set, as a datetime

template_for_id(*tid*)

Retrieve a Template for a given ID in the current domain.

Parameters **tid** – template ID to get

Returns the template

Raises KeyError

to_bytes()

Convert this MessageBuffer to a byte array, suitable for writing to a binary file, socket, or datagram. Finalizes the message by rewriting the message header with current length, and export time.

Returns message as a byte array

tuple_iterator(*ielist*)

Iterate over all records in the Message containing all the IEs in the given ielist. Records are returned as tuples in ielist order.

Parameters **ielist** – an instance of `ipfix.ie.InformationElementList` listing IEs to return as a tuple

Returns a tuple iterator for tuples as in ielist order

write_message(*stream*)

Convenience method to write a message to a stream; see `to_bytes()`.

CHAPTER

FIVE

MODULE IPFIX.READER

Interface to read IPFIX Messages from a stream.

class `ipfix.reader.`**`MessageStreamReader`**(*stream*)

Reads records from a stream of IPFIX messages.

Uses an `ipfix.message.MessageBuffer` internally, and continually reads messages from the given stream into the buffer, iterating over records, until the end of the stream. Use `from_stream()` to get an instance.

Suitable for reading from IPFIX files (see **RFC 5655**) as well as from UDP or TCP sockets with `socketserver.StreamRequestHandler`. When opening a stream from a file, use mode='rb'.

`records_as_dict`()

Iterate over all records in the stream, as dicts mapping IE names to values.

Returns a name dictionary iterator

`records_as_tuple`(*ielist*)

Iterate over all records in the stream containing all the IEs in the given ielist. Records are returned as tuples in ielist order.

Parameters **ielist** – an instance of `ipfix.ie.InformationElementList` listing IEs to return as a tuple

Returns a tuple iterator for tuples in ielist order

`ipfix.reader.`**`from_stream`**(*stream*)

Get a MessageStreamReader for a given stream

Parameters **stream** – stream to read

Returns a `MessageStreamReader` wrapped around the stream.

CHAPTER

SIX

MODULE IPFIX.WRITER

class `ipfix.writer.`**`MessageStreamWriter`**(*stream*, *mtu=65535*)

Writes records to a stream of IPFIX messages.

Uses an `ipfix.message.MessageBuffer` internally, and continually writes records into messages, exporting messages to the stream each time the maximum message size (MTU) is reached. Use `to_stream()` to get an instance.

Suitable for writing to IPFIX files (see **RFC 5655**) as well as to TCP sockets. When writing a stream to a file, use mode='wb'.

..warning: This class is not yet suitable for UDP export; this is an open issue to be fixed in a subsequent release.

`add_template`(*tmpl*)

Add a template to this Writer. Adding a template makes it available for use for exporting records; see `set_export_template()`.

Parameters **tmpl** – the template to add

`export_namedict`(*rec*)

Export a record to the message, using the current template The record is a dictionary mapping IE names to values. The dictionary must contain a value for each IE in the template. Keys in the dictionary not in the template will be ignored.

Parameters **rec** – the record to export, as a dictionary

`flush`()

Export an in-progress Message immediately.

Used internally to manage message boundaries, but can also be used to force immediate export (e.g. to reduce delay due to buffer dwell time), as well as to finish write operations on a Writer before closing the underlying stream.

`set_domain`(*odid*)

Sets the observation domain for subsequent messages sent with this Writer.

Parameters **odid** – Observation domain ID to use for export. Note that templates are scoped to observation domain, so templates will need to be added after switching to a new observation domain ID.

`set_export_template`(*tid*)

Set the template to be used for export by subsequent calls to `export_namedict()` and `export_tuple()`.

Parameters **tid** – Template ID of the Template that will be used to encode records to the Writer. The corresponding Template must have already been added to the Writer, see `add_template()`.

`ipfix.writer.`**`to_stream`**(*stream*, *mtu=65535*)

Get a MessageStreamWriter for a given stream

Parameters

- **stream** – stream to write
- **mtu** – maximum message size in bytes; defaults to 65535, the largest possible ipfix message.

Returns a `MessageStreamWriter` wrapped around the stream.

CHAPTER

SEVEN

MODULE IPFIX.IEUTILS

CHAPTER

EIGHT

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

i

A

B

C

D

E

F

G

I