

---

# **pyfixbuf Documentation**

***Release 0.9.0***

**Carnegie Mellon University**

**Mar 01, 2022**



# CONTENTS

<b>1</b>	<b>Introduction to Pyfixbuf</b>	<b>1</b>
<b>2</b>	<b>Installation</b>	<b>3</b>
<b>3</b>	<b>API Documentation</b>	<b>5</b>
	<b>Python Module Index</b>	<b>51</b>
	<b>Index</b>	<b>53</b>



## INTRODUCTION TO PYFIXBUF

pyfixbuf is a Python API for [libfixbuf](#), an implementation of the *IPFIX RFC 7011* protocol used for building collecting and exporting processes. pyfixbuf can be used to write applications, often called mediators, that collect and export IPFIX. Mediators are useful in modifying, filtering, or adding to the content of the message before forwarding to another IPFIX collection point, or in converting IPFIX to another format (text, database, JSON, etc.).

libfixbuf is a compliant implementation of the IPFIX Protocol, as defined in the *Specification of the IPFIX Protocol for Export of IP Flow Information RFC 7011*. It supports the Information Model defined in *Information Model for IP Flow Information Export RFC 7012* extended as proposed by *Bidirectional Flow Export using IPFIX RFC 5103* to support information elements for representing biflows. It also supports *Exporting Type Information for IPFIX Information Elements RFC 5610* and structured data elements as described in *Export of Structured Data in IPFIX RFC 6313*.

libfixbuf, as well as pyfixbuf, supports UDP and TCP as transport protocols. It also supports operation as an IPFIX File Writer or IPFIX File Reader.



## INSTALLATION

pyfixbuf is compatible with Python versions 2.6, 2.7, and 3.3 or later.

pyfixbuf requires the `ipaddress` library, which is standard in Python 3 since v3.3. Users of Python 2.6 or 2.7 should install it through pypi or from <https://github.com/phihaq/ipaddress>.

pyfixbuf requires `libfixbuf` version 2.2 or later. `libfixbuf` should be built and installed before building and installing pyfixbuf. It may be necessary to set the `PKG_CONFIG_PATH` environment variable to the location of the `libfixbuf.pc` file before building pyfixbuf. Typically this file is located at `/usr/local/lib/pkgconfig` when `libfixbuf` is installed in the default location.

Building and installing pyfixbuf is done using the standard `setup.py` mechanism. The following commands should suffice in most cases:

```
python setup.py build
python setup.py install      #as root
```

You may need to use the `-E` option to `sudo` to preserve the `PKG_CONFIG_PATH` environment variable when installing.





## API DOCUMENTATION

### 3.1 *pyfixbuf* — Class Descriptions

#### Class Overview

The data for single flow is represented in *pyfixbuf* by the *Record* class. A *Record* contains one or more *Field* objects, where each *Field* describes some aspect of the flow.

The format of a flow record is flexible and is described by a *Template*, represented in *pyfixbuf* by the *Template* class. A template contains references to one or more Information Elements (*InfoElement*), where each element describes some aspect of the flow, such as *sourceIPv4Address*. The elements referenced by a template correspond to the fields in the *Record*. The *InfoElementSpec* class is used to specify the Information Elements used by a *Template*.

*libfixbuf* and *Pyfixbuf* support **RFC 6313** which defines object types to hold structured data. A *basicList* (*BL*) may contain multiple instances of a single Information Element. A *subTemplateList* (*STL*) contains zero or more *Record* instances that use that a single *Template*. Finally, the *subTemplateMultiList* (*STML*) type holds *Record* instances that may use different *Template* descriptions. In an *STML*, a group of *Records* that use the same *Template* are represented by the *STMLEntry* class.

An exporting process is created with the *Exporter* class, and a collecting process is created with the *Collector* class if it is file-based or the *Listener* class if it is socket-based. Internally, the *Listener* creates a new *Collector* object when a new connection is made to the socket.

The interface between a *Collector* or an *Exporter* and the *Record* and *Template* objects is managed by two classes:

- The message buffer, *Buffer*, is used to read data from a *Collector* and write data to a *Exporter*.
- The *Session* manages *Templates* and other state, such as message sequence numbers.

The universe of information elements (*InfoElement*) is maintained by an Information Model (*InfoModel*). An *InfoModel* instance contains the information elements defined by IANA. The *InfoModel* may be expanded by defining enterprise-specific elements, such as those defined by CERT (see also the *pyfixbuf.cert* module). The CERT elements are required when reading data produced by YAF or *super\_mediator*.

The *DataType*, *Units*, and *Semantic* classes are enumerations that describe properties of an *InfoElement*.

### 3.1.1 Record

A `Record` is one of the “core” interfaces to the IPFIX data through `libfixbuf`. This is the main object for manipulating the data prior to export and following import.

**class** `pyfixbuf.Record(model: InfoModel[, template: Template = None[, record: Record = None]])`  
Creates an empty `Record` given an [InfoModel](#), `model`, and optionally a `template` or a `record`.

The `Record` is returned from a collection [Buffer](#) or is added to an exporting [Buffer](#).

When adding elements to a `Record`, the `Record` should match a [Template](#). If the process is collecting, the `Record` should match the Internal Template. For an Exporting process, the `Record` should match the External Template, and there should be one `Record` for each External Template. A `Record` can not contain more Information Elements than it's associated `template`. Information Elements should be added to the `Record` in the same order as the [Template](#).

If a `template` is given to the constructor, all Information Elements that exist in the `template` are added to the `Record` in the same order as they exist in the Template. The `record` argument is ignored.

If a `record` is given, all Information Elements that exist in the `record` are added to the `Record` in the same order as they exist in the `record`.

One element must exist in the `Record` before exporting any data.

A `Record` maintains internal dictionaries for the elements that it contains. For this reason, if a template contains more than 1 of the same Information Element, elements must be added using the [add\\_element](#) method in order to give alternate key names to elements that are the same.

A `Record` may also be accessed similar to a list.

**add\_element**(`key_name: str[, type: DataType = DataType.OCTET_ARRAY[, element_name: str = None[, length: int = 0]]]`)

Appends a field named `key_name` to this `Record`, where `key_name` will be used to get or set the value of an Information Element.

The [InfoElement](#) may be specified in three ways: The `key_name` may match the name of an `InfoElement` in the [InfoModel](#), the `element_name` may name the `InfoElement`, or the `type` may be specified as `BASICLIST`, `SUBTEMPLATELIST`, `SUBTEMPLATEMULTILIST`, or `VARLEN` for `octetArray`.

When the `Record` contains duplicate Information Elements, the `key_name` should be unique to ease reference to the elements and the [InfoElement](#) should be specified using the `element_name` or `type` parameters.

The `length` parameter specifies the reduced-length encoding length for the Information Element, similar to the `length` attribute of [InfoElementSpec](#). This may only be applied to certain data types and must be smaller than the default length. When 0, the default length specified by the [InfoModel](#) is used.

When `type` is `BASICLIST`, the `element_name` parameter is used as the content type for the elements in the `basicList`. See [init\\_basic\\_list](#) and the [BL](#) constructor.

Elements must be added in the same order as they exist in the template.

Examples:

```
>>> my_rec = pyfixbuf.Record(model)
>>> my_rec.add_element("sourceTransportPort")
>>> my_rec.add_element("sourceTransportPort2", 0, "sourceTransportPort")
>>> my_rec.add_element("basicList")
>>> my_rec.add_element("basicList2", BASICLIST)
>>> my_rec.add_element("octetTotalCount", length=4)
```

In the above example, an empty `Record` was created. The corresponding template to the above `Record` would look something like:

```

>>> tmpl = Template(model)
>>> tmpl.add_spec_list([
...     pyfixbuf.InfoElementSpec("sourceTransportPort"),
...     pyfixbuf.InfoElementSpec("sourceTransportPort"),
...     pyfixbuf.InfoElementSpec("basicList"),
...     pyfixbuf.InfoElementSpec("basicList"),
...     pyfixbuf.InfoElementSpec("octetTotalCount", 4)])

```

As you can see, we have two sourceTransportPort elements and two basicList elements. A basicList is a list of one or more of the same Information Element. The Information Element in the basicList does not have to be initialized until data is added to the Record.

Since we have two sourceTransportPort fields, we must give a *key\_name* to one of the elements, in this case, sourceTransport2. Since sourceTransportPort2 is not a defined Information Element in the Information Model, the *element\_name* must be given to the method.

Similarly, in order to access the dictionary of elements in the Record, we had to give the second basicList a *key\_name*, basicList2. Since basicList2 is not a defined Information Element, it needs to be given the *type*, BASICLIST. Since *type* is not 0, it does not need an *element\_name*.

**add\_element\_list**(*name\_list*: Iterable[str])

Treats each string in *name\_list* as the name of an *InfoElement* and appends that information element to the Record. See the above method *add\_element*.

**clear\_all\_lists**()

Clears all the lists in the top level of the Record.

Any nested lists must be accessed and cleared manually.

**clear**()

Clears any memory allocated for the Record.

**init\_basic\_list**(*basic\_list\_key*: str[, *count*: int = 0[, *element\_name*: str = None ]])

Initializes a basicList for export with the given *basic\_list\_key* name to a list of *count* elements. If a name is not given to the *element\_name* keyword, it assumes the *basic\_list\_key* is a valid Information Element Name.

Examples:

```

>>> my_rec.add_element("bL", BASICLIST, "octetTotalCount")
>>> my_rec.add_element("basicList")
>>> my_rec.add_element("basicList2", BASICLIST)
>>> my_rec.init_basic_list("bL", 4)
>>> my_rec.init_basic_list("basicList", 3, "destinationTransportPort")
>>> my_rec.init_basic_list("basicList2", 2, "sourceIPv4Address")

```

In the above example, we have initialized three basicLists. The first initializes a basicList of octetTotalCounts by adding the element as as basicList to the record. Later we initialize the basicList to 4 items. The second does the initialization of the type, destinationTransportPort, when calling *init\_basic\_list* as opposed to the first, which is done when the basicList is added to the record. The third, basicList2, is initialized to two sourceIPv4Addresses.

It is perfectly acceptable to initialize a list to 0 elements. All basicLists in the Record must be initialized before appending the Record to the *Buffer*.

A basicList may be initialized via this method, or by using the *BL* and setting the basicList element in the Record to the *BL*.

**clear\_basic\_list**(*basic\_list\_key*: str)

Clears the basicList on this Record identified by *basic\_list\_key*, freeing any memory allocated for the list. This should be called after the Record has been appended to the *Buffer*. Does nothing if the type of the field identified by *basic\_list\_key* is not *DataType*.BASIC\_LIST, but does raises *KeyError* if *basic\_list\_key* is not known on this Record.

**\_\_getitem\_\_**(*key*: Union[str, int]) → Any

Implements the evaluation of `record[key]` for Record instances: Returns the value of the element with the given *key*.

If *key* is a string, it may the name of an *InfoElement* or the *key\_name* specified to the *add\_element* method. *key* may also be an integer corresponding to the positional index in the Record.

Raises an *Exception* when *key* is not in the Record. Use *get* for a similar function that does not raise an *Exception*. See also *get\_field*.

The return type depends on the Information Element type which was defined when initializing the *InfoElement*.

Element Type	Return Type
UINT*, INT*	long
FLOAT*	float
MILLISECONDS, MICROSECONDS	long
NANOSECONDS, SECONDS	long
OCTET_ARRAY	bytearray
BASICLIST	<i>BL</i>
STRING	string
IP (v4 or v6)	ipaddress object
MAC_ADDR	MAC Address String xx:xx:xx:xx:xx:xx
SUBTEMPLATelist	<i>STL</i>
SUBTEMPLATEMULTILIST	<i>STML</i>
Default (Undefined Type)	bytearray

Example:

```
>>> rec
<pyfixbuf.Record object at 0x10d0f49f0>
>>> rec['protocolIdentifier']
6
>>> rec[1]
80
>>> rec.template[1].name
'sourceTransportPort'
```

**\_\_setitem\_\_**(*key*: Union[str, int], *value*: Any)

Implements assignment to `record[key]` for Record instances: Sets the value of the element having the given *key* to *value*.

If *key* is a string, it may the name of an *InfoElement* or the *key\_name* specified to the *add\_element* method. *key* may also be an integer corresponding to the positional index in the Record.

**copy**(*other*: Record)

Copies all the matching elements from the *other* Record to this Record.

**is\_list**(*key*: str) → bool

Returns True or False depending on the type of the given *key*. Raises *KeyError* when *key* is not on this Record.

**get**(key: str, default: Any = None) → Any

Returns record[key] if key exists on this Record; otherwise returns *default*.

**get\_field**(key: str) → *Record.Field*

Returns a *Record.Field* object for key on this Record. Raises *KeyError* when key is not present.

**get\_stl\_list\_entry**(key: str) → *STL*

Gets the subTemplateList from this Record with the given key and returns a newly allocated *STL*. Returns *None* if the type of key is not *DataType.SUB\_TMPL\_LIST*. Raises *KeyError* if key is not a known key on this Record.

A *STL* may also be accessed by using `__getitem__`.

**get\_stml\_list\_entry**(key: str) → *STML*

Gets the subTemplateMultiList from this Record with the given key and returns a newly allocated *STML*. Returns *None* if the type of key is not *DataType.SUB\_TMPL\_MULTI\_LIST*. Raises *KeyError* if key is not a known key on this Record.

A *STML* may also be retrieved by using `__getitem__`.

**as\_dict**() → Dict[Union[str, Tuple(str, int)], Any]

Returns a dict that represents the Record.

The keys of the dictionary are normally strings, but if the Record's *Template* contains duplicate a *InfoElement*, the key for the **second** such element is a couple containing the name and the int 1, the third would be the name and the int 2, et cetera.

**\_\_len\_\_**() → int

Implements the built-in *len()* method for Record instances: Returns the number of elements in the Record.

**\_\_contains\_\_**(item: str) → bool

Implements the *in* operator for Record instances: Tests whether *add\_element* was called with a *key\_name* that is equal to *element*. If the Record was initialized with a *Template*, tests whether the *Template* included an *InfoElement* having the name *element*.

Example:

```
>>> rec
<pyfixbuf.Record object at 0x10d0f49f0>
>>> 'protocolIdentifier' in rec
True
```

**set\_template**(template: *Template*)

If this Record was not initialized with a *Template*, this method may be used to set the Record's *Template*. A Record must have a *Template* associated with it when assigning a Record to a subTemplateList element.

Examples:

```
>>> tmpl = pyfixbuf.Template(model)
>>> tmpl.add_spec_list([
...     pyfixbuf.InfoElementSpec("sourceTransportPort"),
...     pyfixbuf.InfoElementSpec("destinationTransportPort")])
>>> my_rec = pyfixbuf.Record(model)
>>> my_rec.add_element("sourceTransportPort", "destinationTransportPort")
>>> my_rec["sourceTransportPort"] = 13
>>> my_rec["destinationTransportPort"] = 15
>>> my_rec.set_template(tmpl)
>>> other_rec["subTemplateList"] = [my_rec]
```

**\_\_iter\_\_()** → Iterator[Any]

Implements the method to return an Iterator over the values in the Record. See also *iterfields*.

Example:

```
>>> for value in record:
...     print(value)
```

**iterfields()** → Iterator[Record.Field]

Returns an Iterator over the Record's values where each iteration returns a *Record.Field* object. To get a *Record.Field* for a single field, use *get\_field*.

Example:

```
>>> for field in record.iterfields():
...     print(field.ie.name, DataType.get_name(field.ie.type), field.value)
... 
```

**matches\_template(template: Template, exact: bool = False)** → bool

Returns True if this Record matches *Template* using the checks specified by *check\_template*. Returns False otherwise.

**count(element\_name: str)** → int

Counts the occurrences of the *element\_name* in the Record.

Examples:

```
>>> rec.add_element_list(["basicList", "basicList", "basicList"])
>>> rec.count("basicList")
3
>>> rec.count("sourceTransportPort")
0
```

**template : Template**

Returns the *Template* used by this Record.

### 3.1.2 Record.Field

**class Record.Field(name: str, instance: int, ie: InfoElement, length: int, value: Any)**

Represents a complete value field in a *Record*, and is implemented as a subclass of *collection.namedtuple*. This is the type of object returned by the *Record.iterfields* method. A *Record.Field* object includes the following attributes:

**name : str**

The field name provided as the *key* parameter to *Record.add\_element*. For a *Record* built from a *Template*, this is the name is the *InfoElement*.

**instance : int**

An integer that is non-zero when name is not unique. The value represents the number of times name occurs in the *Record* before this one.

**ie : InfoElement**

The canonical *InfoElement* that describes this value.

**length : int**

The length of this field specified to *Record.add\_element* or in the *InfoElementSpec* associated with the Record's *Template*. May be different than the length specified in the *InfoElement* due to reduced length encoding.

**value : Any**

The value of this field.

### 3.1.3 Template

The `Template` type implements an IPFIX Template or an IPFIX Options Template. IPFIX templates contain one or more Information Elements ([InfoElement](#)). If a certain sequence of elements is desired, each Information Element ([InfoElementSpec](#)) must be added to the template in the desired order. Templates are stored by Template ID and type (internal, external) per domain in a [Session](#). The valid range of Template IDs is 256 to 65535. Templates are given a template ID when they are added to a [Session](#). The only difference between Data Templates and Options Templates is that Options Templates have a scope associated with them, which gives the context of reported Information Elements in the Data Records.

An Internal Template is how fixbuf decides what the data should look like when it is transcoded. For this reason, an internal template should match the corresponding [Record](#), in terms of the order of Information Elements. An External Template is sent before the exported data so that the Collecting Process is able to process IPFIX messages without necessarily knowing the interpretation of all data records.

**class** `pyfixbuf.Template(model: InfoModel[, type: bool = False])`

Creates a new Template using the given *model*, an [InfoModel](#). An IPFIX Template is an ordered list of the Information Elements that are to be collected or exported. For export, the order of Information Elements in the Templates determines how the data will be exported.

If *type* is given, an Information Element Type Information Options Template will be created. The appropriate elements will automatically be added to the template and the scope will be sent. See [RFC 5610](#) for more information.

Once a Template has been added to a [Session](#), it cannot be altered.

A [Template](#) can be accessed like a dictionary or a list to retrieve a specific [InfoElementSpec](#).

An Information Model ([InfoModel](#)) is needed to allocate and initialize a new Template.

**copy()** → [Template](#)

Returns a copy of this [Template](#).

**add\_spec(spec: InfoElementSpec)**

Appends a given [InfoElementSpec](#) *spec* to the Template.

Once the Template has been added to a [Session](#), it cannot be altered.

**add\_spec\_list(specs: Iterable[InfoElementSpec])**

Appends each of the [InfoElementSpec](#) items in *specs* to the [Template](#).

Once the [Template](#) has been added to the [Session](#), it can not be altered.

**add\_element(name: str)**

Appends an Information Element with the given *name* to this [Template](#). This function may be used as an alternative to [add\\_spec](#).

This function creates an [InfoElementSpec](#) with the given element *name* and default length and adds it to the Template.

Once the Template has been added to the [Session](#), it can not be altered.

**get\_indexed\_ie(index: int) → InfoElement**

Returns the [InfoElement](#) at the given positional *index* in the [Template](#). Unlike the `__getitem__` method which returns the [InfoElementSpec](#), this method returns the [InfoElement](#) at a particular index.



`get_context()` → Any

Returns the *Template*'s context object as set by the callables registered with the *Session.add\_template\_callback* function.

`__contains__(element: Union[InfoElement, InfoElementSpec, str, int])` → bool

Implements the `in` operator for *Template* instances: Tests whether *element* is in the *Template*.

If *element* is an `int`, return `True` if it is non-negative and less than the length of the *Template*.

If *element* is a `str`, return `True` if an Information Element with the name *element* is included in the *Template*.

If *element* is an *InfoElement* or *InfoElementSpec*, return `True` if the element exists in the *Template*, `False` otherwise.

Examples:

```
>>> t = Template(InfoModel())
>>> t.add_element("protocolIdentifier")
>>> "protocolIdentifier" in t
True
>>> InfoElementSpec("protocolIdentifier") in t
True
```

`__getitem__(key: Union[str, int])` → *InfoElementSpec*

Implements the evaluation of `template[key]` for *Template* instances: Returns the *InfoElementSpec* represented by *key*.

If *key* is an `int`, it is treated as a positional index and an `IndexError` exception is raised when it is out of range.

If *key* is a `str`, it is treated as the name of the *InfoElementSpec* to find and `KeyError` Exception is raised when *key* is not the name of an *InfoElementSpec* on the *Template*. If an *InfoElementSpec* is repeated in the *Template*, querying by name always returns the first element.

Any other type for *key* raises a `TypeError`.

Examples:

```
>>> t = Template(InfoModel())
>>> t.add_element("protocolIdentifier")
>>> t[0]
<pyfixbuf.InfoElementSpec object at 0x107a0fc00>
>>> t["protocolIdentifier"]
<pyfixbuf.InfoElementSpec object at 0x107a0fc00>
```

`__len__()` → int

Implements the built-in `len()` method for *Template* instances: Returns the number of *InfoElementSpec* objects in the *Template*.

Examples:

```
>>> t = Template(InfoModel())
>>> t.add_element("protocolIdentifier")
>>> len(t)
1
```

`__iter__()` → `Iterator[InfoElementSpec]`

Implements the method to return an Iterator over the *InfoElementSpec* objects in this *Template*. See also *ie\_iter*.



**ie\_iter()** → Iterator[*InfoElement*]

Returns an Iterator over the *InfoElement* objects in this *Template*. See also `__iter__`.

Example:

```
>>> for ie in template:
...     print(ie.name, DataType.get_name(ie.type))
... 
```

**scope : int**

Returns the scope associated with the *Template*. Setting scope to zero sets the scope to encompass the entire template. Setting the scope to None removes the scope.

**template\_id : int**

Returns the template ID associated with the *Template*. Template ID can only be changed by adding the template to a *Session*.

**type : bool**

Returns True if template is an Information Element Type Information Template. Returns False otherwise. This attribute may not be changed.

**infomodel : InfoModel**

Returns the *InfoModel* associated with the *Template*. This attribute may not be changed.

**read\_only : bool**

Returns True if this template has been added to a *Session*. This attribute may not be set.

Examples:

```
>>> tmp1 = pyfixbuf.Template(model)
>>> spec = pyfixbuf.InfoElementSpec("sourceTransportPort")
>>> spec2 = pyfixbuf.InfoElementSpec("destinationTransportPort")
>>> tmp1.add_spec(spec)
>>> tmp1.add_spec(spec2)
>>> tmp2 = pyfixbuf.Template(model)
>>> tmp2.add_spec_list([pyfixbuf.InfoElementSpec("fooname"),
...                    pyfixbuf.InfoElementSpec("barname")])
>>> tmp2.scope = 2
>>> if "sourceTransportPort" in tmp1:
>>>     print "yes"
yes
```

### 3.1.4 BL

A *basicList* is a list of zero or more instances of an Information Element. Examples include a list of port numbers, or a list of host names. The BL object acts similar to a Python list with additional attributes.

**class** pyfixbuf.**BL**(*model*: *InfoModel*, *element*: Union[*InfoElement*, *InfoElementSpec*, str][, *count*: int = 0[, *semantic*: int = 0]])

A BL represents a *basicList*.

A *basicList* is a list of zero or more instances of an Information Element.

A *basicList* can be initialized through a *Record* via `init_basic_list` or by creating a BL object.

The constructor requires an *InfoModel* *model*, and a *InfoElementSpec*, *InfoElement*, or string *element*. Additionally, it takes an optional integer *count* which represents the number of elements in the list, and an optional integer *semantic* to express the relationship among the list items.

All basicLists in a Record must be initialized (even to 0) before appending a Record to a *Buffer*.

Examples:

```
>>> rec.add_element("basicList", BASICLIST)
>>> rec.add_element("basicList2", BASICLIST)
>>> bl = BL(model, "sourceTransportPort", 2)
>>> bl[0] = 80
>>> bl[1] = 23
>>> rec["basicList"] = bl
>>> rec.init_basic_list("basicList2", 4, "octetTotalCount")
>>> rec["basicList2"] = [99, 101, 104, 23]
```

**\_\_len\_\_()** → int

Implements the built-in *len()* method for BL instances: Returns the number of entries in the basicList.

Example:

```
>>> bl = BL(model, "sourceTransportPort", 5)
>>> len(bl)
5
```

**\_\_iter\_\_()** → Iterator[Any]

Implements the method to return an Iterator over the items in the BL.

Example:

```
>>> bl = record['basicList']
>>> bl.element.name
'httpContentType'
>>> for v in bl:
...     print(v)
...
'text/xml'
'text/plain'
```

**\_\_getitem\_\_(index: int)** → Any

Implements the evaluation of *bl[index]* for BL instances: Returns the value at position *index* in the basicList.

Example:

```
>>> bl = record['basicList']
>>> bl.element.name
'httpContentType'
>>> print(bl[0])
'text/xml'
```

**\_\_setitem\_\_(key: int, value: Any)**

Implements assignment to *bl[index]* for BL instances: Sets the value at position *index* in the basicList to *value*.

**copy(other: Iterable[Any])**

Copies the items in the list *other* to this BL, stopping when *other* is exhausted or the length of the BL is reached.

Raises *TypeError* when *other* does not support iteration.

**\_\_contains\_\_**(*item: Any*) → bool

Implements the `in` operator for BL instances: Tests whether this BL contains an element whose value is *item*.

Example:

```
>>> bl = record['basicList']
>>> bl.element.name
'httpContentType'
>>> 'text/plain' in bl
True
```

**\_\_str\_\_**() → str

Return `str(self)`.

**\_\_eq\_\_**(*other: list*) → bool

Determines whether *other* is equal to this BL.

Returns `True` when *other* and this `basicList` contain the same elements in the same order. Returns `False` otherwise. Raises `TypeError` when *other* does not support iteration.

Example:

```
>>> bl = record['basicList']
>>> bl.element.name
'httpContentType'
>>> bl == ['text/xml', 'text/plain']
True
```

**clear()**

Clears and frees the `basicList` data.

**semantic : int**

The `structured data semantic value` for this BL.

**element : InfoElement**

The `InfoElement` associated with this BL that was set when the class: *BL* was created. This attribute may not be changed.

Decoding Examples:

```
>>> bl = rec["basicList"]
>>> for items in bl:
...     print str(items) + '\n'
... bl.clear()
```

Encoding Examples:

```
>>> bl = BL(model, "httpUserAgent", 2)
>>> bl[0] = "Mozilla/Firefox"
>>> bl[1] = "Safari5.0"
>>> rec["basicList"] = bl
>>> if "Safari5.0" in bl:
...     print "Apple"
Apple
>>> print bl
["Mozilla/Firefox", "Safari5.0"]
```

### 3.1.5 STL

A subTemplateList is a list of zero or more instances of a structured data type where each entry corresponds to a single template. Since a single template is associated with an STL, a [Record](#) must also be associated with the STL. Access each entry (a Record) in the list by iterating through the STL.

**class** pyfixbuf.STL(*[record: Record = None, key\_name: str = None]*)

A STL represents a subTemplateList.

If *record*, a [Record](#) object, and *key\_name*, a string, are provided, the subTemplateList for *key\_name* in the given *record* is initialized, otherwise a generic STL is initialized. Eventually a [Template](#) must be associated with the STL for encoding.

For decoding, a [Record](#) must be associated with the STL.

**set\_record**(*record: Record*)

Sets the [Record](#) on this STL to *record*.

**\_\_contains\_\_**(*name: str*) → bool

Implements the `in` operator for STL instances: Tests whether the [Template](#) associated with this STL contains an [InfoElement](#) having the name *name*.

**entry\_init**(*record: Record, template: Template[, count: int = 0]*)

Initializes the STL to *count* entries of the given [Record](#) and [Template](#).

This method should only be used to export a STL.

Each STL should be initialized before appending the [Record](#) to the [Buffer](#) even if it is initialized to 0.

Raises an `Exception` when *template* has a `template_id` of 0. You should add the *template* to a [Session](#) before using it for the STL.

The record that contains the STL should not be modified after calling `entry_init()`.

**\_\_iter\_\_**() → Iterator[[Record](#)]

Implements the method to return an Iterator over the [Record](#) objects in the STL.

Example:

```
>>> for record in stl:
...     print(record.as_dict())
```

**next**() → [Record](#)

Returns the next [Record](#) in the STL

**iter\_records**(*tmpl\_id: int = 0*) → Iterator[Records]

Returns an Iterator over the STL's records, including records in any nested [STML](#) or STL. If a template ID is passed, returns an iterator over all the [Record](#) objects with a template ID matching the passed value.

Example:

```
>>> for record in stl.iter_records():
...     print(record.template.template_id)
... 
```

**clear**()

Clears all entries in the list. Nested elements should be accessed and freed before calling this method. Frees any memory previously allocated for the list.

**\_\_getitem\_\_**(*item: Union[int, str]*) → Any

Implements the evaluation of `stl[item]` for STL instances:

If *item* is an `int`, returns the *Record* at that positional index.

If *item* is a `str`, finds the *InfoElement* with the name *item* in the STL's *Template* and returns the value for that element in the most recently accessed *Record* in the STL.

Returns `None` if *item* has an unexpected type.

**\_\_setitem\_\_**(*key*: `int`, *value*: *Record*)

Implements assignment to `stl[index]` for STL instances: Sets the *Record* at position *index* in this STL to *value*.

If this STL was not previously initialized via *entry\_init*, it is initialized with the given *Record*'s *Template* and a count of 1.

**\_\_len\_\_**() → `int`

Implements the built-in *len()* method for STL instances: Returns the number of *Record* objects in the STL.

**template\_id** : `int`

The template ID of the *Template* used for this STL.

**semantic** : `int`

The structured data semantic value for this STL.

Decoding Examples:

```
>>> stl = rec["dnsList"]
>>> stl.set_record(dnsRecord)
>>> for dnsRecord in stl:
...     dnsRecord = dnsRecord.as_dict()
...     for key,value in dnsRecord.items():
...         print key + ": " + str(value) + '\n'
... stl.clear()
```

Encoding Examples:

```
>>> stl = STL()
>>> stl.entry_init(dnsRecord, dnsTemplate, 2)
>>> dnsRecord["dnsName"] = "google.com"
>>> dnsRecord["rrType"] = 1
>>> stl[0] = dnsRecord
>>> dnsRecord["dnsName"] = "ns.google.com"
>>> dnsRecord["rrType"] = 2
>>> stl[1] = dnsRecord
>>> rec["subTemplateList"] = stl
```

### 3.1.6 STML

A *subTemplateMultiList* is a list of zero or more instances of a structured data record, where the data records do not necessarily have to reference the same template. A *subTemplateMultiList* is made up of one or more *STMLEntry* objects. Each *STMLEntry* in the STML typically has a different template associated with it, but that is not a requirement. The data in the STML is accessed by iterating through each *STMLEntry* in the list and setting a *Record* on the *STMLEntry*.

**class** `pyfixbuf.STML`([*record*: *Record* = `None`[, *key\_name*:*str* = `None`[, *type\_count*: `int` = `-1` ] ] )

A STML object represents a *subTemplateMultiList*.

If a *Record* object, *record*, and *key\_name*, a string, are provided, the STML object with *key\_name* will be initialized in the given *Record*. It is only necessary to initialize and give a *type\_count* if the *subTemplateMultiList* will

be exported. All subTemplateMultiLists in an exported *Record* must be initialized. It is acceptable to initialize an STML to 0 list entries.

A STML must be initialized with *record* and *key\_name* OR a *type\_count*. This object can be used to set a sub-TemplateMultiList element in a *Record*.

The subTemplateMultiList is initialized to *None* unless it is given a *type\_count*, in which case it will initialize the list and allocate memory in the given record.

*type\_count* is the number of different templates that the STML will contain. For example, if you plan to have an STML with entries of type Template ID 999 and 888, *type\_count* would be 2. *type\_count* would also be 2 even if both instances will use Template ID 999.

Examples:

```
>>> stml = my_rec["subTemplateMultiList"] # sufficient for collection
>>> stml = pyfixbuf.STML(rec, "subTemplateMultiList", 3) # STML with 3 entries for_
↳export
>>> stml = pyfixbuf.STML(type_count=2)
>>> stml = [record1, record2]
>>> stml2 = pyfixbuf.STML(type_count=3)
>>> stml2[0] = [record1, record2]
>>> stml2[1][0] = record3
>>> stml2[2].entry_init(record3, tmp13, 0) #all entries must be init'd - even to 0.
>>> rec["subTemplateMultiList"] = stml
```

#### **clear()**

Clears the entries in the subTemplateMultiList and frees any memory allocated.

#### **\_\_iter\_\_()** → Iterator[*STMLEntry*]

Implements the method to return an Iterator over the *STMLEntry* objects in the STML.

Example:

```
>>> for entry in stml:
...     for record in entry:
...         print(record.as_dict())
```

#### **next()** → *STMLEntry*

Returns the next *STMLEntry* in the STML.

#### **iter\_records(*tpl\_id: int = 0*)** → Iterator[Records]

Returns an Iterator over the STML's records, including records in any nested STML or *STL*. If a template ID is passed, returns an iterator over all the *Record* objects with a template ID matching the passed value.

Example:

```
>>> for record in stml.iter_records():
...     print(record.template.template_id)
...
```

#### **\_\_len\_\_()** → int

Implements the built-in *len()* method for STML instances: Returns the number of *STMLEntry* objects in the STML.

#### **\_\_contains\_\_(*name: str*)** → bool

Implements the *in* operator for STML instances: Tests whether the *Template* used by the first *STMLEntry* in this STML contains an *InfoElement* having the name *name*.

**\_\_getitem\_\_**(*index: int*) → *STMLEntry*

Implements the evaluation of `stml[index]` for STML instances: Returns the *STMLEntry* at position *index*.

Examples:

```
>>> entry = stml[0]
>>> stml[0].entry_init(record, template, 3)
```

**\_\_setitem\_\_**(*key: int, value: Iterable[Record]*)

Implements assignment to `stml[index]` for STML instances: Sets the *STMLEntry* at position *index* in the STML to the list of *Record* objects in *value*. *value* must be a list. All Records in the list should have the same `:class:Template`.

Examples:

```
>>> stml[0] = [rec1, rec2, rec3, rec4]
```

**semantic : int**

The structured data semantic value for this STML.

Decode Examples:

```
>>> stml = my_rec["subTemplateMultiList"]
>>> for entry in stml:
...     if "tcpSequenceNumber" in entry:
...         entry.set_record(tcp_rec)
...         for tcp_record in entry:
...             tcp_record = tcp_record.as_dict()
...             for key,value in tcp_record.items():
...                 print key + ": " + str(value) + '\n'
```

Encode Examples:

```
>>> stml = STML(type_count=3)
>>> stml.entry_init(rec, template, 2) #init first entry to 2 with template
>>> rec["sourceTransportPort"] = 3
>>> rec["destinationTransportPort"] = 5
>>> stml[0][0] = rec
>>> rec["sourceTransportPort"] = 6
>>> rec["destinationTransportPort"] = 7
>>> stml[0][1] = rec
>>> stml[1][0] = rec2 #init second entry to 1 item using rec2
>>> stml[2].entry_init(rec3, template3, 0) #init third entry to 0
```

### 3.1.7 STMLEntry

Each *STML* consists of one or more *STMLEntry* objects. Each *STMLEntry* is associated with a *Template*, and therefore should have a corresponding *Record*. An *STMLEntry* can contain zero or more instances of the associated *Record*.

**class** pyfixbuf.*STMLEntry*(*stml: STML*)

Creates an empty *STMLEntry* and associates it to the given *STML*, *stml*. There should be one *STMLEntry* for each different *Template* in the *STML*.

Each *STMLEntry* should be initialized using *entry\_init* to associate a *Record* and *Template* with the entry.

**entry\_init**(*record*: *Record*, *template*: *Template*[, *count*: *int* = 0 ])

Initializes the STMLEntry to the given *Record*, *record*, *Template*, *template*, and *count* instances of the *record* it will contain.

This should only be used for exporting a subTemplateMultiList. Entries in the *STML* must all be initialized, even if it is initialized to 0. This method is not necessary if a *Record* has a template associated with it. The application can simply set the STMLEntry to a list of *Record* objects and the STMLEntry will automatically be initialized.

Raises an Exception when *template* has a template\_id of 0. You should add the *template* to a *Session* before using it for the STMLEntry.

Examples:

```
>>> stml = pyfixbuf.STML(my_rec, "subTemplateMultiList", 1)
>>> stml[0].entry_init(my_rec, template, 2)
>>> my_rec["sourceTransportPort"] = 3
>>> stml[0][0] = my_rec
>>> my_rec["sourceTransportPort"] = 5
>>> stml[0][1] = my_rec
```

**set\_record**(*record*: *Record*)

Sets the *Record* on the STMLEntry to *record* in order to access its elements.

**\_\_contains\_\_**(*name*: *str*) → bool

Implements the *in* operator for STMLEntry instances: Tests whether the *Template* associated with this STMLEntry contains an *InfoElement* having the name *name*.

Alternatively, you can access the *Template* ID associated with this STMLEntry to determine the type of *Record* that should be used to access the elements.

**set\_template**(*template*: *Template*)

Assigns a *Template* to the STMLEntry. The *Template* must be valid.

This method may be used as an alternative to *entry\_init*. This is only required if the *Record* that will be assigned to the STMLEntry was not created with a *Template*. Using this method instead of *entry\_init* results in only allocating one item for the STMLEntry.

Raises an Exception when *template* has a template\_id of 0. You should add the *template* to a *Session* before using it for the STMLEntry.

**\_\_iter\_\_**() → Iterator[*Record*]

Implements the method to return an Iterator over the *Record* objects in the STMLEntry.

Example:

```
>>> for entry in stml:
...     for record in entry:
...         print(record.as_dict())
```

**next**() → *Record*

Retrieves the next *Record* in the STMLEntry.

**\_\_getitem\_\_**(*item*: Union[int, str]) → Any

Implements the evaluation of *stmlEntry[item]* for STMLEntry instances:

If *item* is an *int*, returns the *Record* at that positional index.

If *item* is a *str*, finds the *InfoElement* with the name *item* in the STMLEntry's *Template* and returns the value for that element in the first *Record* in the STMLEntry.



Returns None if *item* has an unexpected type.

**\_\_setitem\_\_**(*key: int, value: Record*)

Implements assignment to `stmEntry[index]` for STMEntry instances: Sets the *Record* at position *index* in this STMEntry to *value*.

If this STMEntry has not been previously initialized, it is initialized with the *Record*'s *Template* and a length of 1.

**\_\_len\_\_**() → int

Implements the built-in `len()` method for STMEntry instances: Returns the number of *Record* objects in the STMEntry.

**template\_id** : int

The Template ID of the *Template* that corresponds to this STMEntry in the *STML*.

Examples:

```
>>> stml = my_rec["subTemplateMultiList"]
>>> for entry in stml:
...     if "tcpSequenceNumber" in entry:
...         entry.set_record(tcp_rec)
...         for tcp_record in entry:
...             tcp_record = tcp_record.as_dict()
...             for key,value in tcp_record.items():
...                 print key + ": " + str(value) + '\n'
...     elif entry.template_id == 0xCE00:
...         entry.set_record(dns_rec)
...
>>> stml.clear()
```

### 3.1.8 Buffer

The Buffer implements a transcoding IPFIX Message buffer for both export and collection. The Buffer is one of the “core” interfaces to the fixbuf library. Each Buffer must be initialized to do either collecting or exporting.

**class** pyfixbuf.**Buffer**([*record: Record = None*, *auto: bool = False* ] )

Creates an uninitialized Buffer.

The Buffer must be initialized for collection or export using *init\_collection* or *init\_export* prior to use.

If *auto* is True and the Buffer is initialized for collection, a *Template* is automatically generated from the external template that is read from the *Collector*, the internal template on the Buffer is set to that template, and a matching *Record* is created to match that Template.

Example:

```
>>> collector = Collector()
>>> collector.init_file(sys.argv[1])
>>> session = Session(infomodel)
>>> buffer = Buffer(auto=True)
>>> buffer.init_collection(session, collector)
>>> for record in buffer:
>>>     print(record.as_dict())
```

If *record* is specified, it is set as the Buffer's cached Record for use during collection. The *set\_record* method may also be used to set the cached Record.

There is no requirement to set a cached `Record` on the `Buffer` since the `Buffer` creates one as needed. The cached `Record` is used during collection when its `Template` matches the `Buffer`'s current internal `Template`. Calling `set_internal_template` may clear the cached `Record`.

For export, the user may use `set_internal_template` and `set_export_template` to specify the format of the record, and call `append` to append a `Record` to the `Buffer`.

**init\_collection**(*session*: `Session`, *collector*: `Collector`)

Initializes the `Buffer` for collection given the `Session`, *session*, and `Collector`, *collector*.

**init\_export**(*session*: `Session`, *exporter*: `Exporter`)

Initializes the `Buffer` for Export given the `Session`, *session*, and `Exporter`, *exporter*.

**set\_internal\_template**(*template\_id*: `int`)

Sets the internal `Template` to the one whose ID is *template\_id*. The `Buffer` must have an internal template set on it before collecting or exporting. Causes the `Buffer` to discard the cached `Record` specified in the call to `set_record` or to the constructor if its `Template` does not match that specified by *template\_id*.

**set\_export\_template**(*template\_id*: `int`)

Sets the external `Template` to the one whose ID is *template\_id*. The `Buffer` must have an export template set before appending (`append`) a `Record` to the `Buffer`. The export `Template` describes how fixbuf will write the given `Record` to the output stream.

**next\_record**(*record*: `Record`) → `Record`

Gets the next record on this `Buffer` in the form of the given `Record`, *record*. Returns `None` if the `Buffer` is empty. Raises an exception if an internal template is not set on the `Buffer` or if *record* does not match the internal template.

**next**([*record*: `Record` = `None`]) → `Record`

Returns the next `Record` from this `Buffer` initialized for collection. Raises a `StopIteration` Exception when there are no more `Record` objects. Raises `Exception` if an internal template has not been set on the `Buffer`.

**\_\_iter\_\_**() → `Iterator[Record]`

Implements the method to return an `Iterator` over the `Record` objects in this `Buffer` initialized for collection.

Example:

```
>>> for record in buffer:
...     print(record.as_dict())
```

**set\_record**(*record*: `Record`)

Sets the cached `Record` on this `Buffer` to *record*. If *record* is associated with a `Template` (and this `Buffer` is initialized for collection or export), calls `set_internal_template` with the *template\_id* of that `Template`.

**next\_template**() → `Template`

Retrieves the external `Template` that **will** be used to read the **next** `Record` from this `Buffer` initialized for collection. If no next `Record` is available, raises `StopIteration`.

If *ignore\_options* is `True`, skips options records and returns the `Template` for the next non-options `Record`.

See also `get_template`.

**get\_template**() → `Template`

Retrieves the external `Template` that **was** used to read the **current** `Record` from this `Buffer` initialized for collection. If no `Record` has been read, returns `None`.

See also `next_template`.

**append**(*rec: Record* [, *length: int* ])

Appends the first argument, a *Record*, to this Buffer initialized for export. If a second argument, an *int* representing a length, is given, appends only the first *length* number of bytes to the Buffer.

An internal and external *Template* must be set on the Buffer prior to appending a Record.

**write\_ie\_options\_record**(*name: str*, *template: Template*)

Appends an Information Element Type Information Record (**RFC 5610**) to this Buffer initialized for export. An Options Record is appended with information about the Information Element with the given *name*. *template* is the Information Element Type Options Template that was created by giving *type=True* to the *Template* constructor.

**auto\_insert**()

Tells this Buffer initialized for collection to watch the stream for Information Element Option Records (**RFC 5610**) and automatically create and insert an *InfoElement* to the *InfoModel* for each one seen. Information elements that have an enterprise number of 0 are ignored.

**ignore\_options**(*ignore: bool*)

Tells this Buffer initialized for collection how to handle Options Records.

If *ignore* is set to *True*, the Buffer ignores Options Templates and Records. By default, *ignore* is *False*, and the Buffer returns Options Records that the application must handle.

The application may use *next\_template* to retrieve the *Template* and determine if it is an Options Template.

**emit**()

Writes any pending *Record* objects in the Buffer to the *Exporter*.

**free**()

Frees the Buffer. This method may be invoked when using a Buffer for export to flush and close the stream.

Examples:

```
>>> buf = pyfixbuf.Buffer(my_rec)
>>> buf.init_collection(session, collector)
>>> buf.set_internal_template(999)
>>> for data in buf:
...     data = data.as_dict()
...     for key,value in data.items()
...         print key + ":" + str(value) + '\n'
```

Examples:

```
>>> buf = pyfixbuf.Buffer(my_rec)
>>> buf.init_export(session, exporter)
>>> buf.set_internal_template(999)
>>> buf.set_external_template(999)
>>> session.export_templates()
>>> while count < 10:
...     my_rec['sourceIPv4Address'] = "192.168.3.2"
...     my_rec['destinationIPv4Address'] = "192.168.4.5"
...     buf.append(my_rec)
>>> buf.emit()
```

Examples:

```
>> buf = pyfixbuf.Buffer(auto=True)
>> buf.init_collection(session, collector)
>> for data in buf:
...     data = data.as_dict()
...     for key,value in data.items():
...         print key + ":" + str(value) + '\n'
```

### 3.1.9 Session

The state of an IPFIX Transport Session is maintained in the `Session` object. This includes all IPFIX Message Sequence Number tracking, and internal and external template management. A `Session` is associated with an `InfoModel`. `Template` instances must be added before collecting (via a `Collector` or `Listener`) or exporting (see `Exporter`) any data.

**class** `pyfixbuf.Session(model: InfoModel)`

Creates an empty `Session` given an `InfoModel`.

**add\_template**(*template: Template*[, *template\_id: int = 0*]) → int

Adds the given `Template` *template* to the `Session` with the optional *template\_id*. This template may be used as both an internal and an external template. Use `add_internal_template` or `add_external_template` to be more selective on template usage.

If a *template\_id* is not given or 0, libfixbuf automatically chooses an unused template ID. *template\_id* is used for representing both the internal and external template. The valid range for *template\_id* is 256 to 65535 (0x100 to 0xffff).

When *template\_id* is specified and it is already in use by the `Session`, *template* replaces those `Template(s)`.

Returns the template ID of the added template.

**add\_internal\_template**(*template: Template*[, *template\_id: int = 0*]) → int

Adds the given *template* as an internal template to the session with the optionally given *template\_id*. An internal template determines how the data is presented when transcoded. See also `add_template`.

If *template\_id* is not set or 0, libfixbuf will automatically choose an unused value.

Returns the template ID of the added `Template`.

**add\_external\_template**(*template: Template*[, *template\_id: int = 0*]) → int

Adds the given *template* as an external template to the session with the optionally given *template\_id*. An external template is used when exporting IPFIX data. (libfixbuf automatically creates external templates when collecting IPFIX.) See also `add_template`.

If *template\_id* is not set or 0, libfixbuf will automatically choose an unused value.

Returns the template ID of the added `Template`.

**decode\_only**(*id\_list: Iterable[int]*)

When decoding, causes this `Session` to ignore a record in a subTemplateList (`STL`) or subTemplateMultiList (`STML`) **unless** the record has a template ID in the given *id\_list* of ints. The method is only used when the `Session` is bound to a `Collector` or `Listener`.

This method has no effect on records that are not in a subTemplateList or subTemplateMultiList. See also `ignore_templates`.

**ignore\_templates**(*id\_list: Iterable[int]*)

When decoding, causes this `Session` to ignore a record in a subTemplateList (`STL`) or subTemplateMultiList (`STML`) **when** the record has a template ID in the given *id\_list* of ints. The method is only used when the `Session` is bound to a `Collector` or `Listener`.

This method has no effect on records that are not in a `subTemplateList` or `subTemplateMultiList`. See also [decode\\_only](#).

**add\_template\_pair**(*external\_template\_id*: int, *internal\_template\_id*: int)

Specifies how to transcode data within structured data (a list).

That is, tells a [Collector](#) or [Listener](#) that data in a [STML](#) or [STL](#) whose [Template](#) is *external\_template\_id* is to be transcoded into the [Template](#) given by *internal\_template\_id*.

By default, libfixbuf transcodes each entry in the [STML](#) or [STL](#) with the external template it received, requiring the collector to free or clear any memory allocated for the list elements. The collector can change this behavior by adding a “template pair.” For each entry in the [STL](#) or [STML](#), if the entry has the given *external\_template\_id*, it will use the given *internal\_template\_id* to transcode the record. The *internal\_template\_id* must reference an internal template that was previously added to the [Session](#). If *internal\_template\_id* is 0, the entry will not be transcoded and will be ignored by libfixbuf.

Once a template pair has been added - the default is to ONLY decode entries that have an *external\_template\_id* in this template pair table. Therefore, any entries in a [STML](#) or [STL](#) that reference a template id not in this table will be dropped.

**export\_templates**()

Exports the templates associated with this [Session](#). This is necessary for an exporting session (see [Exporter](#)) and must be called before any records are appended to the [Buffer](#). [Buffer](#) must already have a [Session](#) associated with it using [Buffer.init\\_export](#).

**get\_template**(*template\_id*: int[, *internal*: bool = False]) → [Template](#)

Returns the [Template](#) with the given *template\_id*. By default, the external template is returned. Set *internal* to True to retrieve the internal template with the given *template\_id*. The returned [Template](#) may not be modified.

Raises an [Exception](#) if a [Template](#) with the given *template\_id* does not exist on the [Session](#).

**add\_template\_callback**(*callback*: Callable[[[Session](#), [Template](#), Any], Any])

Adds the callable *callback* to be called whenever a new external template is read from a [Collector](#) or [Listener](#).

Multiple callbacks may be added to a [Session](#). The callbacks may specify a **context** object that is stored with the [Template](#).

After setting the callback(s), when a new template is read the callbacks will be called in the order they are added, and each may modify the context object. After all callbacks are called, the [Template](#)’s context object will be set to the result. A [Template](#)’s context may be retrieved via [Template.get\\_context](#).

The *callback* must be a callable and it will be called with three arguments: *session*, *template*, and *context*. *session* is the [Session](#) from which the callback was added. *template* is the new [Template](#) and will have its *template\_id* property set to the external template ID.

The *context* argument is the current object that is to be stored as the [Template](#)’s context. The callback should either return a new object to use as the context or `None`. A return value of `None` indicates that the context object should not change.

To avoid undefined behavior, the callback functions should be added before the [Session](#) is bound to a [Buffer](#).

**domain** : int

The observation domain on the [Session](#).

Examples:

```
>>> session = pyfixbuf.Session(model)
>>> session.add_internal_template(289, tmpl)
```

(continues on next page)

(continued from previous page)

```
>>> auto_id = session.add_external_template(0, tmpl)
>>> session.decode_only([256, 257])
```

### 3.1.10 Collector

An `Collector` maintains the necessary information for the connection to a corresponding Exporting Process. A `Collector` is used for reading from an IPFIX file. See [Listener](#) for collecting IPFIX over a network.

#### **class** pyfixbuf.Collector

Creates an uninitialized [Collector](#). An IPFIX Collector manages the file it is reading from. Initialize the collector using [init\\_file](#).

#### **init\_file**(filename: str)

Initialize the [Collector](#) to read from the given *filename*. *filename* should be the path to a valid IPFIX File or the string "-" to read from the standard input.

Examples:

```
>>> collector = pyfixbuf.Collector()
>>> collector.init_file("path/to/in.ipfix")
```

### 3.1.11 Exporter

An `Exporter` maintains the information needed for its connection to a corresponding Collecting Process. An `Exporter` may be created to write to IPFIX files or to connect via the network using one of the supported IPFIX transport protocols.

#### **class** pyfixbuf.Exporter

Creates an empty [Exporter](#). Initialize the exporter using [init\\_file](#) or [init\\_net](#).

#### **init\_file**(filename: str)

Initializes the [Exporter](#) to write to the given *filename*.

#### **init\_net**(hostname: str[, transport: str = "tcp"[, port: int = 4739 ]])

Initializes the [Exporter](#) to write to the given *hostname*, *port* over the given *transport*.

Given *hostname* may be a hostname or IP address.

Acceptable values for *transport* are "tcp" and "udp". Default is "tcp".

Given *port* must be greater than 1024. Default is 4739.

Examples:

```
>>> exporter = pyfixbuf.Exporter()
>>> exporter.init_file("/path/to/out.ipfix")
>>> exporter2 = pyfixbuf.Exporter()
>>> exporter2.init_net("localhost", "udp", 18000)
```

### 3.1.12 Listener

The `Listener` manages the passive collection used to listen for connections from Exporting Processes.

**class** `pyfixbuf.Listener`(*session*: *Session*, *hostname*: *str*[, *transport*: *str* = "tcp"[, *port*: *int* = 4739 ] ])

Creates a `Listener` given the *session*, *hostname*, *transport*, and *port*.

*session* must be a valid instance of `Session`.

*hostname* is a string containing the address to bind to, a hostname or an IP Address. If *hostname* is `None`, all addresses are used.

*transport* is the transport protocol; it may contain "tcp" (the default) "udp", or "sctp". (Using "sctp" raises an exception unless libfixbuf has been compiled with SCTP support.)

*port* is the port number to listen on, and it must not be less than 1024. The default is 4739.

Examples:

```
>>> listener = Listener(session, hostname="localhost", port=18000)
```

**wait**([*record*: *Record* = `None` ]) → *Buffer*

Waits for a connection on the set host and port. Once a connection is made, returns a newly allocated *Buffer*.

If a *record* is given to *wait* then the returned *Buffer* will already be associated with a *Record*.

If no *record* is given, you must use *set\_record* on the *Buffer* before accessing the elements.

After receiving the *Buffer* you must set the internal template on the returned *Buffer* using *set\_internal\_template* before accessing the data.

Examples:

```
>>> buf = listener.wait()
>>> buf.set_record(my_rec)
>>> buf.set_internal_template(999)
>>> for data in buf:
>>> ...
```

### 3.1.13 InfoModel

The `InfoModel` type implements an IPFIX Information Model (**RFC 7012**) which holds the known Information Elements (*InfoElement*).

libfixbuf adds, by default, the **IANA managed Information Elements** to the Information Model. IANA's Information Elements have a enterprise number of 0; a non-zero enterprise number is called a private enterprise number (PEN).

To process data from **YAF** or *super\_mediator*, enterprise-specific information elements must be loaded into the information model. These information elements use the CERT PEN, 6871. One may load all CERT defined information elements into an `InfoModel`, *model*, by importing the `pyfixbuf.cert` package and running `pyfixbuf.cert.add_elements_to_model` with *model* as its argument.

There are two alternate ways to add those elements to an `InfoModel`:

1. Download the *XML file* that defines those elements and invoke the model's `InfoModel.read_from_xml_file` method.
2. Invoke `InfoModel.add_element_list` on the model and pass it one of the `pyfixbuf.yaflists` variables.



**class** pyfixbuf.InfoModel

An IPFIX Information Model stores all of the Information Elements (*InfoElement*) that can be collected or exported by a Collecting or Exporting Process.

The constructor creates a new Information Model and adds the default IANA-managed Information Elements.

**add\_element**(*element*: *InfoElement*)

Adds the given *InfoElement* to the *InfoModel*.

**add\_element\_list**(*elements*: *Iterable*[*InfoElement*])

Adds each of the *InfoElement* objects in *elements* to the *InfoModel*.

**get\_element\_length**(*name*: *str*[, *type*: *int*]) → *length*

Returns the in-memory size of an *InfoElement*.

Specifically, searches the *InfoModel* for an *InfoElement* named *name* and if found, returns the length of a value of that *InfoElement* when the value is part of an in-memory *Record*. If the *InfoElement* is of variable length or a list, returns the size of the data structure that holds the value within *libfixbuf*.

However, if *type* is given and is one of *BASICLIST*, *SUBTEMPLATELIST*, or *SUBTEMPLATEMULTILIST*, ignores *name* and returns the length of the list data structure within *libfixbuf*. Any other integer value for *type* is ignored.

Raises *TypeError* if *name* is not a *str* or if *type* is given and is not an *int*. Raises *KeyError* if *name* is not ignored and not the name of a known *InfoElement*.

**get\_element**([*name*: *str*[, *id*: *int*[, *ent*: *int*]]]) → *InfoElement*

Returns the *InfoElement* given the *name* or *id* and *ent*. Raises *KeyError* when the element is not found.

Example:

```
>>> ie = model.get_element('protocolIdentifier')
>>> ie.name
'protocolIdentifier'
>>> ie = model.get_element(id = 4)
>>> ie.name
'protocolIdentifier'
>>> ie = m.get_element(id = 4, ent = CERT_PEN)
Traceback ...
KeyError: 'No element 6871/4'
```

**get\_element\_type**(*name*: *str*) → *DataType*

Returns the type of the Information Element as defined in the *InfoModel* given the *InfoElement* *name*.

**add\_options\_element**(*rec*: *Record*)

Adds the information element contained in the Options *Record*. Use this method for incoming Options Records that contain Information Element Type Information.

**read\_from\_xml\_data**(*xml\_data*: *Any*)

Adds to this *InfoModel* the Information Elements found in the XML data stored in *xml\_data*, which is an object that supports the buffer call interface. See also *read\_from\_xml\_file*.

**read\_from\_xml\_file**(*filename*: *str*)

Adds to this *InfoModel* the Information Elements found in the XML file in *filename*. See also *read\_from\_xml\_data*.

Examples:

```
>>> model = pyfixbuf.InfoModel()
>>> model.add_element(foo);
```

(continues on next page)



(continued from previous page)

```
>>> model.add_element_list([foo, bar, flo])
>>> model.add_element_list(pyfixbuf.yaflists.YAF_DNS_LIST) # adds all YAF DNS DPI_
↳elements
>>> length = model.get_element_length("sourceTransportPort")
>>> print length
2
```

### 3.1.14 InfoElement

An Information Element describes a single piece of data in a *Template* and the *Record* it describes. An IPFIX Information Model (*InfoModel*) holds all Information Elements. Each Information Element consists of a unique and meaningful name, an enterprise number, a numeric identifier, a length, and a data type (*DataType*). The enterprise number is 0 for information elements defined by IANA. A non-zero enterprise number is called a private enterprise number (PEN).

Giving an *InfoElement* an *DataType* allows Python to return the appropriate type of object when the value is requested. Otherwise, the value of the *InfoElement* is *bytearray*. If the type is *DataType.STRING* or one of the list values, the IE length should be *VARLEN*. *InfoElements* that have type *DataType.OCTET\_ARRAY* may or may not be variable length.

Units (*Units*), semantics (*Semantic*), minimum value, maximum value, and description are all optional parameters to further describe an information element. If the process is exporting Information Element Type Option Records (**RFC 5610**), this information may help the collecting process identify the type of information contained in the value of an Information Element.

```
class pyfixbuf.InfoElement(name: str, enterprise_number: int, id: int[, length: int = VARLEN[, reversible:
    bool = False[, endian: bool = False[, type: DataType =
    DataType.OCTET_ARRAY[, units: Units = Units.NONE[, min: int = 0[, max:
    int = 0[, semantic: Semantic = Semantic.DEFAULT[, description: str = None ] ]
    ]]]]]])
```

Creates a new Information Element (IE) using the given *name*, *enterprise\_number*, and *id*, and optional *length*, *reversible* flag, *endian* flag, *datatype*, *units*, *min*, *max*, *semantic*, and *description*. An Information Element identifies a type of data to be stored and transmitted via IPFIX.

If no *length* is provided, the IE is defined as having a variable length. All Strings should be variable length.

If *endian* is *True*, the IE is assumed to be an integer and will be converted to and from network byte order upon transcoding.

If *reversible* is *True*, a second IE is created for the same information in the reverse direction. (The reversed IE's name is created by capitalizing the first character of *name* and prepending the string *reverse*.)

If *type* is set, pyfixbuf will know how to print values of this type. Otherwise the type of the element will be *DataType.OCTET\_ARRAY* and it is returned as a Python *bytearray*. The *type* may be a *DataType* value or an integer.

*units* optionally defines the units of an Information Element. See *Units* for the available units.

*min* optionally defines the minimum value of an Information Element.

*max* optionally defines the maximum value of an Information Element.

*semantic* optionally defines the semantics of an Information Element. See *Semantic* for the known semantics.

*description* optionally contains a human-readable description of an Information Element.

**name : str**

The name, a string, associated with the *InfoElement*.

**enterprise\_number : int**

The Enterprise Number associated with the InfoElement. Default Information Elements have a enterprise number of 0. `enterprise_number` is a 32-bit unsigned integer (1–4,294,967,295).

**id : int**

The Information Element ID that, with the enterprise number, uniquely identifies the Information Element. `id` is an unsigned 15-bit integer (1–32767).

**length : int**

The length of this Information Element as defined in the Information Model. The lengths of elements having type `DataType.STRING`, `DataType.OCTET_ARRAY`, and `DataType.LIST` are 65535 (VARLEN).

**type : DataType**

The data type associated with the Information Element. The values are stored in the [DataType](#) enumeration. If type is not defined, the default type is 0, `DataType.OCTET_ARRAY`. If the Information Element is defined as VARLEN, the default type is 14, `DataType.STRING`.

**units : Units**

The units associated with the Information Element. The values are stored in the `Units` enumeration. If units are not defined, the default is `Units.NONE`.

**min : int**

If a range is defined with the Information Element, min is the minimum value accepted. Valid values are 0 -  $2^{64}-1$ .

**max : int**

If a range is defined for an Information Element, max is the maximum value accepted. Valid values are 0 -  $2^{64}-1$ .

**semantic : Semantic**

Semantic value for an Information Element. The values are stored in the [Semantic](#) enumeration. The default semantic is 0, `Semantic.DEFAULT`.

**description : str**

Description of an Information Element. This is a string. Default is None.

**reversible : bool**

True if an Information Element is defined as reversible.

**endian : bool**

True if an Information Element is defined as numeric value that should potentially be byte-swapped when transcoding to or from network byte order.

**as\_dict() → dict**

Returns a dictionary of key value pairs suitable for use as keyword arguments to `InfoElement`'s constructor.

**ent : int**

An alias for `enterprise_number`.

Examples:

```
>>> foo = pyfixbuf.InfoElement('fooname', CERT_PEN, 722, units=pyfixbuf.Units.WORDS)
>>> bar = pyfixbuf.InfoElement('barname', 123, 565, 1, reversible=True, endian=True)
>>> foo2 = pyfixbuf.InfoElement('fooname2', 0, 888, 3, type=pyfixbuf.DataType.OCTET_
↳ ARRAY)
>>> flo = pyfixbuf.InfoElement('flo_element', 0, 452, 8, endian=True, type=8)
```

### 3.1.15 DataType

The `DataType` class holds the values for the [IPFIX Information Element Data Types](#) defined by Section 3.1 of [RFC 7012](#). This class may not be instantiated, and all methods are static.

#### **class** pyfixbuf.DataType

An enumeration containing the DataTypes supported by pyfixbuf.

In the following table, the RLE column indicates whether the type supports reduced length encoding.

Type	Integer Value	Length	RLE	Python Return Type
DataType.OCTET_ARRAY	0	VARLEN	Yes	bytearray
DataType.UINT8	1	1	No	int
DataType.UINT16	2	2	Yes	long
DataType.UINT32	3	4	Yes	long
DataType.UINT64	4	8	Yes	long
DataType.INT8	5	1	No	long
DataType.INT16	6	2	Yes	long
DataType.INT32	7	4	Yes	long
DataType.INT64	8	8	Yes	long
DataType.FLOAT32	9	4	No	float
DataType.FLOAT64	10	8	Yes	float
DataType.BOOL	11	1	No	bool
DataType.MAC_ADDR	12	6	No	string
DataType.STRING	13	VARLEN	No	string
DataType.SECONDS	14	4	No	long
DataType.MILLISECONDS	15	8	No	long
DataType.MICROSECONDS	16	8	No	long
DataType.NANOSECONDS	17	8	No	long
DataType.IP4ADDR	18	4	No	ipaddress.IPv4Address
DataType.IP6ADDR	19	16	No	ipaddress.IPv6Address
DataType.BASIC_LIST	20	VARLEN	No	pyfixbuf.BL
DataType.SUB_TMPL_LIST	21	VARLEN	No	pyfixbuf.STL
DataType.SUB_TMPL_MULTI_LIST	22	VARLEN	No	pyfixbuf.STML

**classmethod** `get_name(value: int) → str`

Returns the string name associated with *value* in this enumeration.

Raises `KeyError` if *value* is not known to this enumeration. Raises `TypeError` if *value* is not an instance of an int.

**classmethod** `to_string(value: int) → str`

Returns a string that includes the class name and the name associated with *value* in this enumeration; e.g., `DataType(UINT8)` or `Units(PACKETS)`.

Returns a string even when *value* is not known to this enumeration.

**classmethod** `by_name(name: String) → DataType`

Returns the value associated with the name *name* in this enumeration.

Raises `KeyError` if this enumeration does not have a value associated with *name*. Raises `TypeError` if *name* is not a str.

**static** `check_type(data_type: int, value: Any) → bool`  
Returns True if a field whose [InfoElement](#)'s `DataType` is *dt* may be set to *value* given it the Pythonic type of *value*.

**static** `get_length(data_type: int) → int`  
Returns the standard length of a `DataType`

**static** `refine_type_for_length(data_type: int, len: int) → DataType`  
Chooses a `DataType` given a `DataType` and a length

**static** `supports_RLE(data_type: int) → bool`  
Returns True if the `DataType` supports reduced length encoding

### 3.1.16 Semantic

The `Semantic` class holds the values for the [IPFIX Information Element Semantics](#) defined by Section 3.2 of [RFC 7012](#) and Section 3.10 of [RFC 5610](#). This class may not be instantiated, and all methods are static.

**class** `pyfixbuf.Semantic`

An enumeration containing the available Semantic values for Information Elements ([InfoElement](#)). This is not for the semantics of structured data items.

Semantic	Integer Value
<code>Semantic.DEFAULT</code>	0
<code>Semantic.QUANTITY</code>	1
<code>Semantic.TOTALCOUNTER</code>	2
<code>Semantic.DELTACOUNTER</code>	3
<code>Semantic.IDENTIFIER</code>	4
<code>Semantic.FLAGS</code>	5
<code>Semantic.LIST</code>	6
<code>Semantic.SNMPCOUNTER</code>	7
<code>Semantic.SNMPGAUGE</code>	8

**classmethod** `get_name(value: int) → String`  
Returns the string name associated with *value* in this enumeration.  
  
Raises `KeyError` if *value* is not known to this enumeration. Raises `TypeError` if *value* is not an instance of an `int`.

**classmethod** `to_string(value: int) → String`  
Returns a string that includes the class name and the name associated with *value* in this enumeration; e.g., `DataType(UINT8)` or `Units(PACKETS)`.  
  
Returns a string even when *value* is not known to this enumeration.

**classmethod** `by_name(name: String) → Semantic`  
Returns the value associated with the name *name* in this enumeration.  
  
Raises `KeyError` if this enumeration does not have a value associated with *name*. Raises `TypeError` if *name* is not a `str`.

### 3.1.17 Units

The `Units` class holds the values for the [IPFIX Information Element Units](#) initially defined by Section 3.7 of [RFC 5610](#). This class may not be instantiated, and all methods are static.

#### `class pyfixbuf.Units`

An enumeration containing the Units supported by pyfixbuf.

Units	Integer Value
Units.NONE	0
Units.BITS	1
Units.OCTETS	2
Units.PACKETS	3
Units.FLOWS	4
Units.SECONDS	5
Units.MILLISECONDS	6
Units.MICROSECONDS	7
Units.NANOSECONDS	8
Units.WORDS	9
Units.MESSAGES	10
Units.HOPS	11
Units.ENTRIES	12
Units.FRAMES	13
Units.PORTS	14
UNITS.INFERRED	15

#### `classmethod get_name(value: int) → String`

Returns the string name associated with *value* in this enumeration.

Raises `KeyError` if *value* is not known to this enumeration. Raises `TypeError` if *value* is not an instance of an `int`.

#### `classmethod to_string(value: int) → String`

Returns a string that includes the class name and the name associated with *value* in this enumeration; e.g., `DataType(UINT8)` or `Units(PACKETS)`.

Returns a string even when *value* is not known to this enumeration.

#### `classmethod by_name(name: String) → Units`

Returns the value associated with the name *name* in this enumeration.

Raises `KeyError` if this enumeration does not have a value associated with *name*. Raises `TypeError` if *name* is not a `str`.

### 3.1.18 InfoElementSpec

An Information Element Specification (`InfoElementSpec`) is used to name an Information Element (`InfoElement`) for inclusion in a `Template`. The Information Element must have already been defined and added to the Information Model (`InfoModel`). An `InfoElementSpec` contains the exact name of the defined Information Element and an optional length override.

**class** `pyfixbuf.InfoElementSpec`(*name*: `str`[, *length*: `int` = 0])

Creates a new Information Element Specification using the given *name*, and optional override *length*. An IPFIX Template is made up of one or more `InfoElementSpec`s.

The given *name* must be a defined Information Element (`InfoElement`) in the Information Model (`InfoModel`) before adding the `InfoElementSpec` to a `Template`.

If *length* is nonzero, it is used instead of the default length of this Information Element for reduced-length encoding. Not all Information Element data types support reduced-length encoding, and *length* must be smaller than the default length. When 0, the default length provided by the `InfoElement` in the `InfoModel` is used.

Note that the values of *name* and *length* are only checked when the `InfoElementSpec` is added to a `Template`. When an `InfoElementSpec` whose *length* is zero is added to a `Template`, the *length* of that `InfoElementSpec` is modified to reflect the default length of the `InfoElement`.

Examples:

```
>>> spec1 = pyfixbuf.InfoElementSpec("fooname")
>>> spec2 = pyfixbuf.InfoElementSpec("sourceTransportPort")
>>> spec3 = pyfixbuf.InfoElementSpec("flo_element", 4)
```

**name** : `str`

The Information Element Specification name.

**length** : `int`

The length override for the Information Element Specification. A value of 0 indicates the length of the element is the default length specified for that `InfoElement` in the `InfoModel`.

### 3.1.19 Global Variables

pyfixbuf defines the following global constants:

`pyfixbuf.__version__`

The version of pyfixbuf.

`pyfixbuf.VARLEN`

The size used to indicate an `InfoElement` has a variable length. (65535)

`pyfixbuf.CERT_PEN`

The private enterprise number assigned by IANA to the CERT division within the Software Engineering Institute. (6871)

`pyfixbuf.BASICLIST`

An alias for `DataType.BASIC_LIST`. The `Record.add_element` and `InfoModel.get_element_length` methods accept this value to help when constructing a `Record`.

`pyfixbuf.SUBTEMPLATELIST`

An alias for `DataType.SUB_TMPL_LIST`. The `Record.add_element` and `InfoModel.get_element_length` methods accept this value to help when constructing a `Record`.

**pyfixbuf.SUBTEMPLATEMULTILIST**

An alias for `DataType.SUB_TMPL_MULTI_LIST`. The `Record.add_element` and `InfoModel.get_element_length` methods accept this value to help when constructing a `Record`.

## 3.2 *pyfixbuf.cert* — Information Elements for NetSA Tools

The *pyfixbuf.cert* module provides functions to update an *pyfixbuf.InfoModel* with the Information Elements defined by CERT and used by the NetSA tools such as YAF and *super\_mediator*. These functions load the information elements from the `cert_ipfix.xml` file, which is included as a resource in the *pyfixbuf* distribution. To load these elements into your Information Model, use:

```
import pyfixbuf
import pyfixbuf.cert
```

```
model = pyfixbuf.InfoModel()
pyfixbuf.cert.add_elements_to_model(model)
```

`pyfixbuf.cert.add_elements_to_model(model: InfoModel)`

Adds to the *pyfixbuf.InfoModel* *model* the text returned by *info\_element\_xml*.

`pyfixbuf.cert.info_element_xml() → str`

Returns a string containing the XML text that defines the Information Elements defined by the CERT at SEI and used by YAF and other NetSA tools.

Note: The length of this string is approximately 198k.

## 3.3 *pyfixbuf.yaflists* — Pre-defined Information Element Lists

The *pyfixbuf.yaflists* module defines variables which specify lists of CERT enterprise-specific Information Elements. The Elements may be added to an Information Model (*pyfixbuf.InfoModel*) by invoking *pyfixbuf.InfoModel.add\_element\_list* with one of the list variables as an argument.

**NOTE:** The following variables are outdated and incomplete should not be used in new code. Please change your code so it adds the CERT Information Elements to your model by loading them from the *pyfixbuf.cert* package, as shown in this example:

```
# create your model as normal
model = pyfixbuf.InfoModel()

# add this:
import pyfixbuf.cert
pyfixbuf.cert.add_elements_to_model(model)
```

As of *pyfixbuf-0.9.0*, these variables are no longer imported into the *pyfixbuf* module. To use them, you must explicitly import them:

```
import pyfixbuf
from pyfixbuf.yaflists import YAF_LIST, YAF_DNS_LIST, YAF_DPI_LIST
from pyfixbuf.yaflists import YAF_FLOW_STATS_LIST, YAF_FTP_LIST
from pyfixbuf.yaflists import YAF_HTTP_LIST, YAF_IMAP_LIST, YAF_RTSP_LIST
from pyfixbuf.yaflists import YAF_SIP_LIST, YAF_SLP_LIST, YAF_SMTP_LIST
from pyfixbuf.yaflists import YAF_SSL_LIST, YAF_STATS_LIST
```

The `pyfixbuf.InfoElement` objects in these lists use the CERT private enterprise number (PEN) 6871. Each list contains Elements that are related to a particular internet protocol (e.g., HTTP, DNS, SMTP). The variables `YAF_LIST` and `YAF_STATS_LIST` are necessary for reading the IPFIX streams created by YAF when its deep-packet inspection feature is disabled.

### 3.3.1 YAF\_LIST

Information Element	ID TYPE	Description
initialTCPFlags	14 UINT8	Initial sequence number of the forward direction of the flow
unionTCPFlags	15 UINT8	Union of TCP flags of all packets other than the initial packet in the forward direction of the flow
reverseFlowDeltaMilliseconds	21 UINT32	Difference in time in milliseconds between first packet in forward direction and first packet in reverse direction
silkAppLabel	33 UINT16	Application label, defined as the primary well-known port associated with a given application.
osName	36 STRING	p0f OS Name for the forward flow based on the SYN packet and p0f SYN Fingerprints.
payload	36 OCTET ARRAY	Initial n bytes of forward direction of flow payload.
osVersion	37 STRING	p0f OS Version for the forward flow based on the SYN packet and p0f SYN Fingerprints.
firstPacketBanner	38 OCTET ARRAY	IP and transport headers for first packet in forward direction to be used for external OS Fingerprints.
secondPacketBanner	39 OCTET ARRAY	IP and transport headers for first packet in forward direction to be used for external OS Fingerprints.
flowAttributes	40 UINT16	Miscellaneous flow attributes for the forward direction of the flow
osFingerPrint	107 STRING	p0f OS Fingerprint for the forward flow based on the SYN packet and p0f SYN fingerprints.
yafFlowKeyHash	106 UINT32	The 32 bit hash of the 5-tuple and VLAN that is used as they key to YAF's internal flow table.

### 3.3.2 YAF\_STATS\_LIST

Information Element	ID TYPE	Description
expiredFragmentCount	100 UINT32	Total amount of fragments that have been expired since yaf start time.
assembledFragmentCount	101 UINT32	Total number of packets that been assembled from a series of fragments since yaf start time.
meanFlowRate	102 UINT32	The mean flow rate of the yaf flow sensor since yaf start time, rounded to the nearest integer.
meanPacketRate	103 UINT32	The mean packet rate of the yaf flow sensor since yaf start time, rounded to the nearest integer.
flowTableFlushEventCount	104 UINT32	Total number of times the yaf flow table has been flushed since yaf start time.
flowTablePeakCount	105 UINT32	The maximum number of flows in the yaf flow table at any one time since yaf start time.



### 3.3.3 YAF\_FLOW\_STATS\_LIST

Information Element	ID	TYPE	Description
smallPacketCount	50	UINT32	The number of packets that contain less than 60 bytes of payload.
nonEmptyPacketCount	50	UINT32	The number of packets that contain at least 1 byte of payload.
dataByteCount	50	UINT64	Total bytes transferred as payload.
averageInterarrivalTime	50	UINT64	Average number of milliseconds between packets.
standardDeviationInterarrivalTime	50	UINT64	Standard deviation of the interarrival time for up to the first ten packets.
firstNonEmptyPacketSize	50	UINT16	Payload length of the first non-empty packet.
maxPacketSize	50	UINT16	The largest payload length transferred in the flow.
firstEightNonEmptyPacketDirections	50	UINT8	Represents directionality for the first 8 non-empty packets. 0 for forward direction, 1 for reverse direction.
standardDeviationPayloadLength	50	UINT16	The standard deviation of the payload length for up to the first 10 non empty packets.
tcpUrgCount	50	UINT32	The number of TCP packets that have the URGENT Flag set.
largePacketCount	51	UINT32	The number of packets that contain at least 220 bytes of payload.

### 3.3.4 YAF\_HTTP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
httpServerString	110	STRING
httpUserAgent	111	STRING
httpGet	112	STRING
httpConnection	113	STRING
httpVersion	114	STRING
httpReferer	115	STRING
httpLocation	116	STRING
httpHost	117	STRING
httpContentLength	118	STRING
httpAge	119	STRING
httpAccept	120	STRING
httpAcceptLanguage	121	STRING
httpContentType	122	STRING
httpResponse	123	STRING
httpCookie	220	STRING
httpSetCookie	221	STRING
httpAuthorization	252	STRING
httpVia	253	STRING
httpX-Forwarded-For	254	STRING
httpRefresh	256	STRING
httpIMEI	257	STRING
httpIMSI	258	STRING
httpMSISDN	259	STRING
httpSubscriber	260	STRING
httpExpires	255	STRING
httpAcceptCharset	261	STRING

continues on next page

Table 1 – continued from previous page

Information Element	ID	TYPE
httpAcceptEncoding	262	STRING
httpAllow	263	STRING
httpDate	264	STRING
httpExpect	265	STRING
httpFrom	266	STRING
httpProxyAuthentication	267	STRING
httpUpgrade	268	STRING
httpWarning	269	STRING
httpDNT	270	STRING
httpX-Forwarded-Proto	271	STRING
httpX-Forwarded-Host	272	STRING
httpX-Forwarded-Server	273	STRING
httpX-DeviceID	274	STRING
httpX-Profile	275	STRING
httpLastModified	276	STRING
httpContentEncoding	277	STRING
httpContentLanguage	278	STRING
httpContentLocation	279	STRING
httpX-UA-Compatible	280	STRING

### 3.3.5 YAF\_SLP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
slpVersion	128	UINT8
slpMessageType	129	UINT8
slpString	130	STRING

### 3.3.6 YAF\_FTP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
ftpReturn	131	STRING
ftpUser	132	STRING
ftpPass	133	STRING
ftpType	134	STRING
ftpRespCode	135	STRING

### 3.3.7 YAF\_IMAP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
imapCapability	136	STRING
imapLogin	137	STRING
imapStartTLS	138	STRING
imapAuthenticate	139	STRING
imapCommand	140	STRING
imapExists	141	STRING
imapRecent	142	STRING

### 3.3.8 YAF\_RTSP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
rtspURL	143	STRING
rtspVersion	144	STRING
rtspReturnCode	145	STRING
rtspContentLength	146	STRING
rtspCommand	147	STRING
rtspContentType	148	STRING
rtspTransport	149	STRING
rtspCSeq	150	STRING
rtspLocation	151	STRING
rtspPacketsReceived	152	STRING
rtspUserAgent	153	STRING
rtspJitter	154	STRING

### 3.3.9 YAF\_SIP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
sipInvite	155	STRING
sipCommand	156	STRING
sipVia	157	STRING
sipMaxForwards	158	STRING
sipAddress	159	STRING
sipContentLength	160	STRING
sipUserAgent	161	STRING

### 3.3.10 YAF\_SMTP\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
smtpHello	162	STRING
smtpFrom	163	STRING
smtpTo	164	STRING
smtpContentType	165	STRING
smtpSubject	166	STRING
smtpFilename	167	STRING
smtpContentDisposition	168	STRING
smtpResponse	169	STRING
smtpEnhanced	170	STRING
smtpSize	222	STRING
smtpDate	251	STRING

### 3.3.11 YAF\_DNS\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
dnsQueryResponse	174	UINT8
dnsQRType	175	UINT16
dnsAuthoritative	176	UINT8
dnsNXDomain	177	UINT8
dnsRRSection	178	UINT8
dnsQName	179	STRING
dnsCName	180	STRING
dnsMXPreference	181	UINT16
dnsMXExchange	182	STRING
dnsNSDName	183	STRING
dnsPTRDName	184	STRING
dnsTTL	199	UINT32
dnsTXTData	208	STRING
dnsSOASerial	209	UINT32
dnsSOARefresh	210	UINT32
dnsSOARetry	211	UINT32
dnsSOAExpire	212	UINT32
dnsSOAMinimum	213	UINT32
dnsSOAMName	214	STRING
dnsSOARName	215	STRING
dnsSRVPriority	216	UINT16
dnsSRVWeight	217	UINT16
dnsSRVPort	218	UINT16
dnsSRVTarget	219	STRING
dnsID	226	UINT16
dnsAlgorithm	227	UINT8
dnsKeyTag	228	UINT16
dnsSigner	229	STRING

continues on next page

Table 2 – continued from previous page

Information Element	ID	TYPE
dnsSignature	230	OCTET ARRAY
dnsDigest	231	OCTET ARRAY
dnsPublicKey	232	OCTET ARRAY
dnsSalt	233	OCTET ARRAY
dnsHashData	234	OCTET ARRAY
dnsIterations	235	UINT16
dnsSignatureExpiration	236	UINT32
dnsSignatureInception	237	UINT32
dnsDigestType	238	UINT8
dnsLabels	239	UINT8
dnsTypeCovered	240	UINT16
dnsFlags	241	UINT16

### 3.3.12 YAF\_SSL\_LIST

Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
sslCipher	185	UINT32
sslClientVersion	186	UINT8
sslServerCipher	187	UINT32
sslCompressionMethod	188	UINT8
sslCertVersion	189	UINT8
sslCertSignature	190	STRING
sslCertIssuerCountryName	191	STRING
sslCertIssuerOrgName	192	STRING
sslCertIssuerOrgUnitName	193	STRING
sslCertIssuerZipCode	194	STRING
sslCertIssuerState	195	STRING
sslCertIssuerCommonName	196	STRING
sslCertIssuerLocalityName	197	STRING
sslCertIssuerStreetAddress	198	STRING
sslCertSubCountryName	200	STRING
sslCertSubOrgName	201	STRING
sslCertSubOrgUnitName	202	STRING
sslCertSubZipCode	203	STRING
sslCertSubState	204	STRING
sslCertSubCommonName	205	STRING
sslCertSubLocalityName	206	STRING
sslCertSubStreetAddress	207	STRING
sslCertSerialNumber	208	STRING
sslObjectType	245	UINT8
sslObjectValue	246	STRING
sslCertValidityNotBefore	247	STRING
sslCertValidityNotAfter	248	STRING
sslCertPublicKeyAlgorithm	249	STRING
sslCertPublicKeyLength	250	UINT16
sslRecordVersion	288	UINT16

### 3.3.13 YAF\_DPI\_LIST

This list contains miscellaneous Information Elements from the remaining protocols YAF decodes. Descriptions of each Information Element can be found at <http://tools.netsa.cert.org/yaf/yafdpi.html>.

Information Element	ID	TYPE
mysqlUsername	223	STRING
mysqlCommandCode	224	UINT8
mysqlCommandText	225	STRING
pop3TextMessage	124	STRING
ircTextMessage	125	STRING
tftpFilename	126	STRING
tftpMode	127	STRING
dhcpFingerPrint	242	STRING
dhcpVendorCode	243	STRING
dnp3SourceAddress	281	UINT16
dnp3DestinationAddress	282	UINT16
dnp3Function	283	UINT8
dnp3ObjectData	284	OCTET_ARRAY
modbusData	285	OCTET_ARRAY
ethernetIPData	286	OCTET_ARRAY
rtpPayloadType	287	UINT8

## 3.4 Pyfixbuf Examples

### 3.4.1 Collector Example

The pyfixbuf API aims to follow the original C library. The following example follows the traditional method of collecting IPFIX with libfixbuf:

1. Create an information model
2. Add Private Enterprise Number (PEN) Information Elements to the model.
3. Create an IPFIX template(s).
4. Define what the template(s) will contain.
5. Add the elements to the template.
6. Create an IPFIX collector (file vs TCP vs UDP)
7. Create a session.
8. Add the template(s) to the session.
9. Create an incoming data buffer.
10. Associate the collector and the session to the buffer.
11. Set the internal template on the buffer.
12. Read the data from the buffer into records.

```
#!/usr/bin/env python
```

(continues on next page)

(continued from previous page)

```

import sys
# Import pyfixbuf
import pyfixbuf
import pyfixbuf.cert

# Import times from netsa-python for nice timestamp formats
from netsa.data.times import *

# Test that the number of arguments is correct
if ( len (sys.argv) != 2):
    print "Must supply exactly one IPFIX file to read"
    sys.exit()

# Create an InfoModel
infomodel = pyfixbuf.InfoModel()

# Add enterprise-specific information elements defined by CERT
pyfixbuf.cert.add_elements_to_model(infomodel)

# Create a Template
tmpl = pyfixbuf.Template(infomodel)

# Create a Stats Template to receive YAF Stats (Options) Records
stats_tmpl = pyfixbuf.Template(infomodel)

# Add some elements to the internal template
# This is a normal YAF flow record

data_list = [pyfixbuf.InfoElementSpec("flowStartMilliseconds"),
              pyfixbuf.InfoElementSpec("flowEndMilliseconds"),
              pyfixbuf.InfoElementSpec("octetTotalCount"),
              pyfixbuf.InfoElementSpec("reverseOctetTotalCount"),
              pyfixbuf.InfoElementSpec("packetTotalCount"),
              pyfixbuf.InfoElementSpec("reversePacketTotalCount"),
              pyfixbuf.InfoElementSpec("sourceIPv4Address"),
              pyfixbuf.InfoElementSpec("destinationIPv4Address"),
              pyfixbuf.InfoElementSpec("sourceTransportPort"),
              pyfixbuf.InfoElementSpec("destinationTransportPort"),
              pyfixbuf.InfoElementSpec("flowAttributes"),
              pyfixbuf.InfoElementSpec("reverseFlowAttributes"),
              pyfixbuf.InfoElementSpec("protocolIdentifier"),
              pyfixbuf.InfoElementSpec("flowEndReason"),
              pyfixbuf.InfoElementSpec("silkAppLabel"),
              pyfixbuf.InfoElementSpec("subTemplateMultiList")]

tmpl.add_spec_list(data_list)

# Add elements to the stats template (this is a subset of the YAF stats)

stats_list = [pyfixbuf.InfoElementSpec("exportedFlowRecordTotalCount"),
              pyfixbuf.InfoElementSpec("packetTotalCount"),
              pyfixbuf.InfoElementSpec("droppedPacketTotalCount"),

```

(continues on next page)

(continued from previous page)

```
        pyfixbuf.InfoElementSpec("ignoredPacketTotalCount")]
```

```
stats_tmpl.add_spec_list(stats_list)
```

```
# Create a collector
collector = pyfixbuf.Collector()

# Initialize the collector to read an IPFIX file
collector.init_file(sys.argv[1])

# Create a session
session = pyfixbuf.Session(infomodel)

# Add your data template to the session
session.add_internal_template(tmpl, 999)

# Add the stats template to the session
session.add_internal_template(stats_tmpl, 911)

# Create a Record for each Template and/or each SubTemplate
# The following rec will contain all the elements in the data template
rec = pyfixbuf.Record(infomodel, tmpl)

# The following rec will contain all the elements in the stats template
statsrec = pyfixbuf.Record(infomodel, stats_tmpl)

# Create a TCP Record, since YAF exports TCP information in the
# subTemplateMultiList by default

tcprec = pyfixbuf.Record(infomodel)

# Since we don't need a template for this TCP Record because
# it belongs in the subTemplateMultiList, we have to add
# the TCP elements using the addElement method

tcp_elements = ["tcpSequenceNumber", "initialTCPFlags", "unionTCPFlags",
                "reverseInitialTCPFlags", "reverseUnionTCPFlags",
                "reverseTcpSequenceNumber"]

tcprec.add_element_list(tcp_elements)

# Create a new buffer for collection - rec matches our internal template
buf = pyfixbuf.Buffer(rec)

# Initialize the buffer for collection
buf.init_collection(session, collector)

# Set the internal template on the buffer
buf.set_internal_template(999)

# Now we can get the elements from the buffer
```

(continues on next page)



(continued from previous page)

```

for data in buf:
    print "-----FLOW-----"
    for field in data.iterfields():
        if field.ie.type == DataType.MILLISECONDS:
            # use netsa-python to print times
            print field.key + ": " + str(make_datetime(field.value/1000))
        # print every element that is not a subtemplatemultilist
        elif field.ie.type != DataType.SUB_TMPL_MULTI_LIST:
            print field.key + ": " + str(field.value)

    # retrieve STML
    stml = data["subTemplateMultiList"]
    # Iterate through entries in STML
    for entry in stml:
        # Is it a TCP Template?
        if "tcpSequenceNumber" in entry:
            # set the tcprec on the entry
            entry.set_record(tcprec)
            # iterate through records in this entry of the stml
            for record in entry:
                for field in record.iterfields():
                    print field.key + ": " + str(field.value)

    # clear the STML
    stml.clear()

    # Now check to see if the next record is a stats record
    # by checking the next template on the buffer

    try:
        tpl_next = buf.next_template()
    except StopIteration:
        break
    # if a template has scope - it's an options template
    if ( tpl_next.scope ):
        # Set the internal template to the stats template
        buf.set_internal_template(911)
        # get the next record in the buffer as a stats record
        stats = buf.next_record(statsrec)
        print "----STATS----"
        if (stats != None):
            stats = stats.as_dict()
            # print all the items in stats
            for key,value in stats.items():
                print key + ": " + str(value)
            # Set the internal template back to the data template
            buf.set_internal_template(999)

```

It may be the case that the IPFIX data can change often and the application needs to be able to collect everything that the records contain. In that case, pyfixbuf can be used to build Records on the fly based on the templates that it receives. This is slightly different than the traditional way of reading IPFIX. Typically, the application knows what kind of data it wants and libfixbuf will populate only the fields the application cares about. In the following example, the application wants to view the contents of every IPFIX record in the file.

```
#!/usr/bin/env python

import sys
# Import pyfixbuf
import pyfixbuf
# If using the CERT information elements
import pyfixbuf.cert

# Import times from netsa-python for nice timestamp formats
from netsa.data.times import *

# Test that the number of arguments is correct
if ( len (sys.argv) != 2):
    print "Must supply exactly one IPFIX file to read"
    sys.exit()

# Create an InfoModel
infomodel = pyfixbuf.InfoModel()

# If necessary, augment model with private enterprise information elements
#pyfixbuf.cert.add_elements_to_model(infomodel)

# Create a collector
collector = pyfixbuf.Collector()

# Initialize the collector to read an IPFIX file
collector.init_file(sys.argv[1])

# Create a session
session = pyfixbuf.Session(infomodel)

# Create a new buffer for collection
buf = pyfixbuf.Buffer(auto=True)

# Initialize the buffer for collection
buf.init_collection(session, collector)

# Initialize a record counter
count = 0

# Read the data
for data in buf:

    print "-----FLOW %d-----" % count
    for field in data.iterfields():
        if field.ie.type == DataType.MILLISECONDS:
            # use netsa-python to print times
            print field.key + ": " + str(make_datetime(field.value/1000))
            # print every element that is not a subtemplatemultilist
        elif field.ie.type != DataType.SUB_TMPL_MULTI_LIST:
            print str(field.key) + ": " + str(field.value)
    # retrieve STML
    if "subTemplateMultiList" in data:
```

(continues on next page)

(continued from previous page)

```

stml = data["subTemplateMultiList"]
# Iterate through entries in STML
for entry in stml:
    for record in entry:
        for field in record.iterfields():
            if field.ie.type != DataType.SUB_TMPL_LIST:
                print str(field.key) + ": " + str(field.value)
        if "subTemplateList" in record:
            stl = record["subTemplateList"]
            for sub in stl:
                for field in sub.iterfields():
                    print str(field.key) + ": " + str(field.value)
            stl.clear()

# clear the STML
stml.clear()
count += 1

```

### 3.4.2 Conversion Example

pyfixbuf is often used for converting comma-separated value (CSV) records or non-IPFIX records to IPFIX so they can be imported by an IPFIX collector tool, such as [Analysis Pipeline](#). The following code provides an example of converting CSV to IPFIX. The CSV are DNS records that were converted from NMSG to CSV with `nmsgtool`. Specifically, this example transforms the A and AAAA records from CSV to IPFIX records so they can be read and analyzed by Analysis Pipeline.

```

#!/usr/bin/env python
## -----
## nmsg_to_pipeline.py
## sample IPFIX converter/exporter using pyfixbuf.
## Takes a csv file that has format <domain name>,<type>,<ttl>,<ip>
## -----

import sys
import pyfixbuf
import pyfixbuf.cert
import csv

# Test that the argument number is correct
if (len(sys.argv) < 3):
    print "Must supply an IPFIX file to write to."
    print ("Usage: nmsg_to_pipeline.py nmsg_csv_file.txt" +
          " <ipfix file or domain/ip> <port_number>")
    sys.exit()

# Create the information model with the standard IPFIX elements
infomodel = pyfixbuf.InfoModel()

# Add YAF's IPFIX elements
pyfixbuf.cert.add_elements_to_model(infomodel)

```

(continues on next page)

(continued from previous page)

```

# Create the "outer" template
tmpl = pyfixbuf.Template(infomodel)

# Add elements we want in our template
a_list = [
    pyfixbuf.InfoElementSpec("dnsName"),
    pyfixbuf.InfoElementSpec("dnsA"),
    pyfixbuf.InfoElementSpec("dnsTTL"),
    pyfixbuf.InfoElementSpec("dnsRRType")]

# Add elements to our template
tmpl.add_spec_list(a_list)

aaaa_list = [
    pyfixbuf.InfoElementSpec("dnsName"),
    pyfixbuf.InfoElementSpec("dnsAAAA"),
    pyfixbuf.InfoElementSpec("dnsTTL"),
    pyfixbuf.InfoElementSpec("dnsRRType")]

tmplaaaa = pyfixbuf.Template(infomodel)
tmplaaaa.add_spec_list(aaaa_list)

# Create the exporter
exporter = pyfixbuf.Exporter()

# Create the IPFIX file to write to or open a network socket
if (len(sys.argv) == 3):
    exporter.init_file(sys.argv[2])
else:
    exporter.init_net(hostname=sys.argv[2], port=sys.argv[3], transport='tcp')

# Create the session
session = pyfixbuf.Session(infomodel)

# Create internal and external templates since this is an exporter
session.add_internal_template(tmpl, 999)
session.add_external_template(tmpl, 999)

session.add_internal_template(tmplaaaa, 1000)
session.add_external_template(tmplaaaa, 1000)

# Create the records to fill for export
rec = pyfixbuf.Record(infomodel, tmpl)
reca = pyfixbuf.Record(infomodel, tmplaaaa)

# Create the buffer for exporter
buf = pyfixbuf.Buffer(rec)

# Make the buffer an export buffer
buf.init_export(session, exporter)

# Set the internal template on the buffer

```

(continues on next page)

(continued from previous page)

```

buf.set_internal_template(999)

# Export the templates to the file
session.export_templates()

# Set the export template
buf.set_export_template(999)

# Open NMSG CSV file
f = open(sys.argv[1], 'r')

csv.field_size_limit(sys.maxsize)

c = csv.reader(f, delimiter=',')

count = 0

for row in c:
    if (row[1] == "A(1)" or row[1] == "1"):
        try:
            rec['dnsA'] = row[3]
        except:
            print "row[3] is " + row[3]
        rec['dnsName'] = row[0]
        rec['dnsTTL'] = int(row[2])
        rec['dnsRRType'] = 1

        buf.set_internal_template(999)
        buf.set_export_template(999)
        buf.append(rec)

        count += 1

#some records have more than 1 IPv4Address on a line
if (len(row) > 4):
    k = len(row) - 4
    while (k):
        rec['dnsA'] = row[3+k]
        rec['dnsName'] = row[0]
        rec['dnsTTL'] = int(row[2])
        rec['dnsRRType'] = 1

        buf.append(rec)
        count += 1
        k -= 1
    elif (row[1] == "AAAA(28)" or row[1] == "28"):
        try:
            reca['dnsAAAA'] = row[3]
        except:
            print "row[3] is " + row[3]
        reca['dnsName'] = row[0]
        reca['dnsTTL'] = int(row[2])

```

(continues on next page)

(continued from previous page)

```
reca['dnsRRType'] = 28

buf.set_internal_template(1000)
buf.set_export_template(1000)
buf.append(reca)

count += 1

if (len(row) > 4):
    k = len(row) - 4
    while (k):
        reca['dnsAAAA'] = row[3+k]
        reca['dnsName'] = row[0]
        reca['dnsTTL'] = int(row[2])
        reca['dnsRRType'] = 28

        buf.append(reca)
        count += 1
        k -= 1

buf.emit()
print "FINISHED sending %d records" % count

f.close()
```

Additional examples are included with the pyfixbuf source code distribution in the `samples` directory.

## PYTHON MODULE INDEX

### p

pyfixbuf, [5](#)  
pyfixbuf.cert, [35](#)  
pyfixbuf.yaflists, [35](#)





## Symbols

\_\_contains\_\_() (pyfixbuf.BL method), 14  
 \_\_contains\_\_() (pyfixbuf.Record method), 9  
 \_\_contains\_\_() (pyfixbuf.STL method), 16  
 \_\_contains\_\_() (pyfixbuf.STML method), 18  
 \_\_contains\_\_() (pyfixbuf.STMLEntry method), 20  
 \_\_contains\_\_() (pyfixbuf.Template method), 12  
 \_\_eq\_\_() (pyfixbuf.BL method), 15  
 \_\_getitem\_\_() (pyfixbuf.BL method), 14  
 \_\_getitem\_\_() (pyfixbuf.Record method), 8  
 \_\_getitem\_\_() (pyfixbuf.STL method), 16  
 \_\_getitem\_\_() (pyfixbuf.STML method), 18  
 \_\_getitem\_\_() (pyfixbuf.STMLEntry method), 20  
 \_\_getitem\_\_() (pyfixbuf.Template method), 12  
 \_\_iter\_\_() (pyfixbuf.BL method), 14  
 \_\_iter\_\_() (pyfixbuf.Buffer method), 22  
 \_\_iter\_\_() (pyfixbuf.Record method), 9  
 \_\_iter\_\_() (pyfixbuf.STL method), 16  
 \_\_iter\_\_() (pyfixbuf.STML method), 18  
 \_\_iter\_\_() (pyfixbuf.STMLEntry method), 20  
 \_\_iter\_\_() (pyfixbuf.Template method), 12  
 \_\_len\_\_() (pyfixbuf.BL method), 14  
 \_\_len\_\_() (pyfixbuf.Record method), 9  
 \_\_len\_\_() (pyfixbuf.STL method), 17  
 \_\_len\_\_() (pyfixbuf.STML method), 18  
 \_\_len\_\_() (pyfixbuf.STMLEntry method), 21  
 \_\_len\_\_() (pyfixbuf.Template method), 12  
 \_\_setitem\_\_() (pyfixbuf.BL method), 14  
 \_\_setitem\_\_() (pyfixbuf.Record method), 8  
 \_\_setitem\_\_() (pyfixbuf.STL method), 17  
 \_\_setitem\_\_() (pyfixbuf.STML method), 19  
 \_\_setitem\_\_() (pyfixbuf.STMLEntry method), 21  
 \_\_str\_\_() (pyfixbuf.BL method), 15  
 \_\_version\_\_ (in module pyfixbuf), 34

## A

add\_element() (pyfixbuf.InfoModel method), 28  
 add\_element() (pyfixbuf.Record method), 6  
 add\_element() (pyfixbuf.Template method), 11  
 add\_element\_list() (pyfixbuf.InfoModel method), 28  
 add\_element\_list() (pyfixbuf.Record method), 7

add\_elements\_to\_model() (in module pyfixbuf.cert), 35  
 add\_external\_template() (pyfixbuf.Session method), 24  
 add\_internal\_template() (pyfixbuf.Session method), 24  
 add\_options\_element() (pyfixbuf.InfoModel method), 28  
 add\_spec() (pyfixbuf.Template method), 11  
 add\_spec\_list() (pyfixbuf.Template method), 11  
 add\_template() (pyfixbuf.Session method), 24  
 add\_template\_callback() (pyfixbuf.Session method), 25  
 add\_template\_pair() (pyfixbuf.Session method), 25  
 append() (pyfixbuf.Buffer method), 22  
 as\_dict() (pyfixbuf.InfoElement method), 30  
 as\_dict() (pyfixbuf.Record method), 9  
 auto\_insert() (pyfixbuf.Buffer method), 23

## B

BASICLIST (in module pyfixbuf), 34  
 BL (class in pyfixbuf), 13  
 Buffer (class in pyfixbuf), 21  
 by\_name() (pyfixbuf.DataType class method), 31  
 by\_name() (pyfixbuf.Semantic class method), 32  
 by\_name() (pyfixbuf.Units class method), 33

## C

CERT\_PEN (in module pyfixbuf), 34  
 check\_type() (pyfixbuf.DataType static method), 31  
 clear() (pyfixbuf.BL method), 15  
 clear() (pyfixbuf.Record method), 7  
 clear() (pyfixbuf.STL method), 16  
 clear() (pyfixbuf.STML method), 18  
 clear\_all\_lists() (pyfixbuf.Record method), 7  
 clear\_basic\_list() (pyfixbuf.Record method), 7  
 Collector (class in pyfixbuf), 26  
 copy() (pyfixbuf.BL method), 14  
 copy() (pyfixbuf.Record method), 8  
 copy() (pyfixbuf.Template method), 11  
 count() (pyfixbuf.Record method), 10

## D

`DataType` (class in `pyfixbuf`), 31  
`decode_only()` (`pyfixbuf.Session` method), 24

## E

`emit()` (`pyfixbuf.Buffer` method), 23  
`entry_init()` (`pyfixbuf.STL` method), 16  
`entry_init()` (`pyfixbuf.STMLEntry` method), 19  
`export_templates()` (`pyfixbuf.Session` method), 25  
`Exporter` (class in `pyfixbuf`), 26

## F

`free()` (`pyfixbuf.Buffer` method), 23

## G

`get()` (`pyfixbuf.Record` method), 8  
`get_context()` (`pyfixbuf.Template` method), 11  
`get_element()` (`pyfixbuf.InfoModel` method), 28  
`get_element_length()` (`pyfixbuf.InfoModel` method), 28  
`get_element_type()` (`pyfixbuf.InfoModel` method), 28  
`get_field()` (`pyfixbuf.Record` method), 9  
`get_indexed_ie()` (`pyfixbuf.Template` method), 11  
`get_length()` (`pyfixbuf.DataType` static method), 32  
`get_name()` (`pyfixbuf.DataType` class method), 31  
`get_name()` (`pyfixbuf.Semantic` class method), 32  
`get_name()` (`pyfixbuf.Units` class method), 33  
`get_stl_list_entry()` (`pyfixbuf.Record` method), 9  
`get_stml_list_entry()` (`pyfixbuf.Record` method), 9  
`get_template()` (`pyfixbuf.Buffer` method), 22  
`get_template()` (`pyfixbuf.Session` method), 25

## I

`ie_iter()` (`pyfixbuf.Template` method), 13  
`ignore_options()` (`pyfixbuf.Buffer` method), 23  
`ignore_templates()` (`pyfixbuf.Session` method), 24  
`info_element_xml()` (in module `pyfixbuf.cert`), 35  
`InfoElement` (class in `pyfixbuf`), 29  
`InfoElementSpec` (class in `pyfixbuf`), 34  
`InfoModel` (class in `pyfixbuf`), 27  
`init_basic_list()` (`pyfixbuf.Record` method), 7  
`init_collection()` (`pyfixbuf.Buffer` method), 22  
`init_export()` (`pyfixbuf.Buffer` method), 22  
`init_file()` (`pyfixbuf.Collector` method), 26  
`init_file()` (`pyfixbuf.Exporter` method), 26  
`init_net()` (`pyfixbuf.Exporter` method), 26  
`is_list()` (`pyfixbuf.Record` method), 8  
`iter_records()` (`pyfixbuf.STL` method), 16  
`iter_records()` (`pyfixbuf.STML` method), 18  
`iterfields()` (`pyfixbuf.Record` method), 10

## L

`Listener` (class in `pyfixbuf`), 27

## M

`matches_template()` (`pyfixbuf.Record` method), 10  
module  
    `pyfixbuf`, 5  
    `pyfixbuf.cert`, 35  
    `pyfixbuf.yaflists`, 35

## N

`next()` (`pyfixbuf.Buffer` method), 22  
`next()` (`pyfixbuf.STL` method), 16  
`next()` (`pyfixbuf.STML` method), 18  
`next()` (`pyfixbuf.STMLEntry` method), 20  
`next_record()` (`pyfixbuf.Buffer` method), 22  
`next_template()` (`pyfixbuf.Buffer` method), 22

## P

`pyfixbuf`  
    module, 5  
`pyfixbuf.cert`  
    module, 35  
`pyfixbuf.yaflists`  
    module, 35

## R

`read_from_xml_data()` (`pyfixbuf.InfoModel` method), 28  
`read_from_xml_file()` (`pyfixbuf.InfoModel` method), 28  
`Record` (class in `pyfixbuf`), 6  
`Record.Field` (class in `pyfixbuf`), 10  
`refine_type_for_length()` (`pyfixbuf.DataType` static method), 32

## RFC

RFC 5103, 1  
RFC 5610, 1, 11, 23, 29, 32, 33  
RFC 6313, 1, 5  
RFC 7011, 1  
RFC 7012, 1, 27, 31, 32

## S

`Semantic` (class in `pyfixbuf`), 32  
`Session` (class in `pyfixbuf`), 24  
`set_export_template()` (`pyfixbuf.Buffer` method), 22  
`set_internal_template()` (`pyfixbuf.Buffer` method), 22  
`set_record()` (`pyfixbuf.Buffer` method), 22  
`set_record()` (`pyfixbuf.STL` method), 16  
`set_record()` (`pyfixbuf.STMLEntry` method), 20  
`set_template()` (`pyfixbuf.Record` method), 9  
`set_template()` (`pyfixbuf.STMLEntry` method), 20  
`STL` (class in `pyfixbuf`), 16  
`STML` (class in `pyfixbuf`), 17  
`STMLEntry` (class in `pyfixbuf`), 19

SUBTEMPLATELIST (*in module pyfixbuf*), 34  
SUBTEMPLATEMULTILIST (*in module pyfixbuf*), 34  
supports\_RLE() (*pyfixbuf.DataType static method*), 32

## T

Template (*class in pyfixbuf*), 11  
to\_string() (*pyfixbuf.DataType class method*), 31  
to\_string() (*pyfixbuf.Semantic class method*), 32  
to\_string() (*pyfixbuf.Units class method*), 33

## U

Units (*class in pyfixbuf*), 33

## V

VARLEN (*in module pyfixbuf*), 34

## W

wait() (*pyfixbuf.Listener method*), 27  
write\_ie\_options\_record() (*pyfixbuf.Buffer method*), 23