Pyfixmsg

Jan 22, 2021

Contents

`pyfixmsg` is a library for parsing, manipulating and serialising FIX messages, primarily geared towards testing. See the *example*.

CHAPTER 1

Objectives

- provide a rich API to compare and manipulate messages.
- (mostly) Message type agnostic,
- (mostly) value types agnostic
- pluggable : load specification XML files, custom specifications or build your own Specification class for repeating groups definitions and message types, define your own codec for custom serialisation or deserialisation quirks.

CHAPTER 2

Dependencies

- None for the core library.
- Optional lxml for faster parsing of xml specification files.
- Optional pytest to run the tests.
- Optional spec files from quickfix to get started with standard FIX specifications.

CHAPTER 3

Core classes

- *FixMessage*. Inherits from `dict`. The workhorse class. By default comes with a codec that will parse standard-looking `FIX`, but without support repeating groups.
- `RepeatingGroup`. defines repeating groups of a FixMessage.
- *Codec* defines how to parse a buffer into a FixMessage, and how to serialise it back
- *FixSpec* defines the `FIX` specification to follow. Only required for support of repeating group. Defined from Quickfix's spec XML files.

CHAPTER 4

How to run the tests

`py.test --spec=/var/tmp/FIX50.xml` will launch the tests against the spec file in /var/tmp. You will need to load the spec files from quickfix to get the tests to work.

The spec files are not included in this distribution.

CHAPTER 5

Notes

This is only a FIX message library. It doesn't include a FIX session management system or an order management core, or anything similar. It is purely message parsing-manipulation-serialisation. It is however easy to integrate into an order management or a exchange/broker simulator, etc.

5.1 API Documentation

5.1.1 FixMessage

class pyfixmsg.fixmessage.**FixMessage**(**args*, ***kwargs*)

Bases: *pyfixmsg.fixmessage.FixFragment*

Simple dictionary-like object, for use with FIX raw messages. Note that the tags are converted (when possible) to integers, and that the values are kept as strings. The default separator is `;`, but can be specified. Check the definition of `load_fix()` for details.

Example:

```
>>> fix = FixMessage()
>>> fix.load_fix(line)
>>> #print(fix)
{6: '0', 8: 'FIX.4.2',
10: '100', 10481: 'A', 14: '0',
15: 'EUR', 10641: 'blabla',
18: '1', 21: '2', 22: '5',
151: '1',
11951: 'HOOF7M0f4BGJ0rkaNTkkeAA',
....
```

FixMessage also have a `time` attribute, a `direction` attribute (inbound : 0, outbound : 1) and a `recipient` which is rather where it's been received from or sent to. FixMessages sort by default on time, and will be considered equal if the dictionary values are the same AND the time is the same.

This FixMessage is eager : it will parse the whole fix and store it locally. It is significantly faster in most usage patterns that we observed.

useful shortcut methods :

```
fix.tag_exact(tag, value)
fix.tag_iexact(tag, value)
fix.tag_contains(tag, value)
fix.tag_icontains(tag, value)
fix.tag_match_regex(tag, value)
```

Note : the tag_* methods don't support repeating groups

FragmentType
alias of *FixFragment*

__init__(**args*, ***kwargs*)
The constructor uses the `dict()` signature unmodified. You can set the following manually or through a factory function:

- `self.process` an opaque value (to store the process that received or processed the message, defaults to empty string).
- `self.separator` the default separator to use when parsing messages. Defaults to `';'`
- `self.time` the time the message has been created or received. Defaults to `datetime.utcnow()`
- `self.recipient` opaque value (to store for whom the message was intended)
- `self.direction` Whether the message was received (`0`), sent (`1`) or unknown (`None`)
- `self.typed_values` Whether the values in the message are typed. Defaults to `False`
- `self.raw_message` If constructed by class method `from_buffer`, keep the original format
- `self.codec` Default `Codec` to use to parse message. Defaults to a naive codec that doesn't support repeating groups

apply(*update*)
equivalent to `update()` but if any value in the update dictionary is None and the tag is present in the current message, that tag is removed. Note: naive about repeating groups

Parameters **update** (`dict`) – map of values to update the state with.

calculate_checksum()
calculates the standard fix checksum

checksum(*value=None*)
FIX checksum

copy()
Copy interface without using the copy module

fix
Legacy compatibility, will be removed shortly

classmethod from_buffer(*msg_buffer*, *fix_codec*)
Create a FixMessage from a buffer and a codec

Parameters

- **msg_buffer** (`str`) – a buffer as a string
- **fix_codec** (`Codec`) – an object with static encode() and decode() calls

Returns a FixMessage object

Return type `FixMessage`

from_wire(*msg*, *codec=None*)
Extract from a wire representation according to a codec

get_raw_message()
Return the original string from which the fix message was constructed

load_fix(*string*, *process=None*, *separator=';'*)
Parses a FIX message from a string using default codecs and separators.

Parameters

- **string** (`bytes`) – the string containing the FIX message to be parsed
- **process** (`unicode`) – Optional originator of the FIX message
- **separator** (`unicode`) – Character delimiting "tag=val" pairs. Optional. By default this is a ';' character. Specify `pyfixmsg.SEPARATOR` when parsing standard FIX.

Returns A parsed fix message

Return type `FixMessage`

output_fix(*separator=';'*, *calc_checksum=True*, *remove_length=False*)
ouputs itself as a vanilla FIX message. This forces the output to String fix but tries to reuse the spec from the current codec

set_len_and_chksum()
Assign length and checksum based on current contents

set_or_delete(*tag*, *value*)
Sets the tag if value is neither None or the empty string. Deletes the tag otherwise. Only works on top-level tags (not inside repeating groups)

tag_contains(*tag*, *value*, *case_insensitive=False*)
Returns True if self[tag] contains value. Returns False otherwise, or if the tag doesnt exist This is a string string comparison

tag_exact(*tag*, *value*, *case_insensitive=False*)
Returns True if self[tag] has the exact value. Returns False if the tag doesnt exist or is not exact

tag_exact_dict(*dictionary*)
check that all the keys and values of the passed dict are present and identical in the fixmsg

tag_ge(*tag*, *value*)
Test tag is greater or equal to value. Uses decimal comparison if possible. Returns False if tag absent

tag_gt(*tag*, *value*)
Test tag is greater than value. Uses decimal comparison if possible. Returns False if tag absent

tag_icontains(*tag*, *value*)
case-insensitive version of tag_contains

tag_iexact(*tag*, *value*)
Returns True if self[tag] has the exact value (case insensitive). Returns False if the tag doesnt exist or is not exact

tag_in(*tag*, *values*)
returns True if self[tag] is in values, false otherwise or if the tag doesnt exist

tag_le(*tag*, *value*)
Test tag is smaller or equal value. Uses decimal comparison if possible. Returns False if tag absent

tag_lt(*tag*, *value*)
Test tag is smaller than value. Uses decimal comparison if possible. Returns False if tag absent

tag_match_regex(*tag*, *regex*)
returns True of self[tag] matches regex, false otherwise or if the tag doesnt exist

tags
Note: this property is there to replace a self reference that existed before.

Deprecated.

to_wire(*codec=None*)
Return wire representation according to a codec

update_all(*tag*, *value*)
this will force a tag (that already exists!) to a value at all appearances

class pyfixmsg.fixmessage.**FixFragment**(**args*, ***kwargs*)
Bases: dict

Type designed to hold a collection of fix tags and values. This type is used directly for the contents of repeating groups. Whole fix messages are parsed from their wire representation to instances of the FixMessage type which inherits from this type.

__init__(**args*, ***kwargs*)
FixFragment constructor.

all_tags()
Returns a list of all the tag keys in this message, including flattened tags that are only present in repeating groups. The same tag will not appear twice in the list.

Returns A list of tag keys (usually strings or ints)

Return type list

anywhere(*tag*)
returns true if the tag is in the message or anywhere inside any contained repeating group

find_all(*tag*)
Generator. Find all instances of the tag in the message or inside repeating groups and returns the path to them one at a time.

Example, navigate all paths for a given tag:

```
>>> for path in msg.find_all(self, tag):
...   # path here is a list of ints or str keys
...   path_msg = msg
...   for key in path:
...     path_msg = path_msg[key]
...     # [...] do something at each level in the path
...   path_msg[tag] = # [...] do something with the last level of the path
```

Returns a generator of paths where each path is a list of string or integer indices into the message

Return type Generator of list of int or str

classmethod from_dict(*tags_dict*)
Create a FixMessage from a dictionary.

Parameters **tags_dict** (dict of int to str or int or float or long) – dictionary of FIX tags to values

Returns a FixMessage object

Return type FixMessage

length()
Length of the body of the message in bytes

5.1.2 Repeating groups

class pyfixmsg.**RepeatingGroup**(**args*, ***kwargs*)
Bases: list

Implementation of repeating groups for pyfixmsg.FixMessage. The repeating group will look like {opening_tag:[FixMessage,FixMessage]} in the fix message a repeating group behaves like a list. You can add two repeating groups, or append a FixMessage to one.

__init__(**args*, ***kwargs*)
Maintains list's signature unchanged.

Sets * self.number_tag (the tag that contains the number of elements in the group) * self.first_tag (the first repeated tag) * self.standard (reserved)

all_tags()
Returns a list of all the tag keys in any member of this repeating group, The same tag will not appear twice in the generated sequence. The count tag for the repeating group is *not* included, it is considered as part of the parent message. Order is not guaranteed. @return: A list of tag keys (usually strings or ints) @rtype: C{list}

classmethod create_repeating_group(*tag*, *standard=True*, *first_tag=None*)
creates a group with member. Can't use __init__ as it would mean overriding the list __init__ which sounds dangerous

entry_tag
returns the entry tag for the group and its value as a tuple

find_all(*tag*)
Generator. Find all instances of the tag in the message or inside repeating groups and returns the path to them one at a time.

Example, navigate all paths for a given tag:

```
>>> for path in msg.find_all(self, tag):
...   # path here is a list of ints or str keys
...   path_msg = msg
...   for key in path:
...     path_msg = path_msg[key]
...     # [...] do something at each level in the path
...   path_msg[tag] = # [...] do something with the last level of the path
```

@return: a generator of paths where each path is a list of string or integer indices into the message @rtype: Generator of C{list} of C{int} or C{str}

length()
Length of the body of the message in bytes

class pyfixmsg.**RepeatingGroupFactory**(*tag*, *standard=True*, *first_tag=None*)
Bases: object

An easy way to create a repeating group for a given tag, without having to define all the tags yourself, takes the standard ones

__init__(*tag*, *standard=True*, *first_tag=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

get_r_group(**fix_messages*)
factory method. I'm not familiar with the factory design pattern, it shows ;-)

5.1.3 Codec

class pyfixmsg.codecs.stringfix.**Codec**(*spec=None*, *no_groups=False*, *fragment_class=<type 'dict'>*, *decode_as=None*, *decode_all_as_347=False*)

Bases: object

FIX codec. Initialise with a *FixSpec* to support repeating groups.

This class is used to transform the serialised FIX message into an instance of fragment_class, default dict Tags are assumed to be all of type int, repeating groups are lists of fragment_class

Values can either bytes or unicode or a mix thereof, depending on the constructor arguments.

__init__(*spec=None*, *no_groups=False*, *fragment_class=<type 'dict'>*, *decode_as=None*, *decode_all_as_347=False*)

Parameters

- **spec** – the *FixSpec* instance to use to parse messages. If spec is not defined repeating groups will not be parsed correctly, and the logic to handle encoded tags will not be functional.
- **no_groups** – set to True to ignore repeating groups
- **fragment_class** – Which dict-like object to return when parsing messages. Also sets the type of members of repeating groups
- **decode_as** – what encoding to decode all tags. Defaults to None, which returns the raw byte strings. setting to a non-None value makes both non-numerical tags and values to be unicode, using this value for decode.
- **decode_all_as_347** – whether to trust tag 347 to decode all other tags or only the Encoded* ones. If set to False, use 347 normally for Encoded* tags, respect decode_as for all other tags. If 347 is not present on the message, the values are left encoded.

parse(*buff*, *delimiter='='*, *separator='\x01'*)
Parse a FIX message. The FIX message is expected to be a bytestring and the output is a dictionary-like object which type is determined by the fragment_class constructor argument and which keys are int and values unicode. Note that if there is a non-int tag in the message, this will be stored as a key in the original format (i.e. bytestring)

Parameters

- **buff** (bytestr or unicode) – Buffer to parse
- **delimiter** (unicode) – A character that separate key and values inside the FIX message. Generally '='. Note the type: because of the way the buffer is tokenised, this needs to be unicode (or str in python 2.7*).
- **separator** (unicode) – A character that separate key+value pairs inside the FIX message. Generally ''. See type observations above.

serialise(*msg*, *separator='\x01'*, *delimiter='='*, *encoding=None*)
Serialise a message into a bytestring.

Parameters

- **msg** (dict-like interface) – the message to serialse
- **delimiter** – as in parse()
- **separator** – as in parse()
- **encoding** (str) – encoding mode

5.1.4 FixTag

class pyfixmsg.reference.**FixTag**(*name*, *tag*, *tagtype=None*, *values=()*)
Bases: object

Fix tag representation. A fix tag has name, tag (number), type and valid values (enum)

__init__(*name*, *tag*, *tagtype=None*, *values=()*)

Parameters

- **name** (str) – Tag name
- **tag** (int) – Tag number
- **tagtype** (str) – Type as in quickfix's XML reference documents
- **values** (tuple((str, str)) with the first element of each tuple being the value of the enum, the second the name of the value.) – The valid enum values

add_enum_value(*name*, *value*)
Add a value to the tag's enum values.

del_enum_value(*name=None*, *value=None*)
Delete a value from the tag's enum values. Specify name or value using keyword arguments. If specifying both, they must both match the known name and value otherwise ValueError is raised.

enum_by_name(*name*)
Retrieve an enum value by name

enum_by_value(*value*)
Retrieve an enum value by value

5.1.5 FixSpec

class pyfixmsg.reference.**FixSpec**(*xml_file*, *eager=False*)
Bases: object

A python-friendly representation of a FIX spec. This class is built from an XML file sourced from Quickfix (http://www.quickfixengine.org/).

It contains the Message Types supported by the specification, as a map (FixSpec.msg_types) of message type value ('D', '6', '8', etc..) to MessageType class, and all the fields supported in the spec as a TagReference instance (FixSpec.tags) which can be accessed by tag name or number.

__init__(*xml_file*, *eager=False*)

Parameters

- **xml_file** (str) – path to a quickfix specification xml file
- **eager** (bool) – whether to eagerly populate tags maps for speedy lookup or only on first access

5.2 Examples

```
#!/bin/env python
"""
Examples illustrating the usage of FixMessage and associated objects
"""

from __future__ import print_function

import os
import sys
import decimal
import argparse
from copy import copy
from random import randint

sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)), '..'))

from pyfixmsg import RepeatingGroup
from pyfixmsg.fixmessage import FixMessage, FixFragment
from pyfixmsg.reference import FixSpec, FixTag
from pyfixmsg.codecs.stringfix import Codec

if sys.version_info.major >= 3:
    unicode = str  # pylint: disable=C0103,W0622


def main(spec_filename):
    """
    Illustration of the usage of FixMessage.

    :param spec_filename: a specification file from Quickfix.org
    :type spec_filename: ``str``
    """

    # For this example you need FIX 4.2 specification
    # refer to: path_to_quickfix/FIX42.xml (MS: fsf/quickfix/1.14.3.1ms)
    spec = FixSpec(spec_filename)
    codec = Codec(spec=spec,  # The codec will use the given spec to find repeating groups
                  fragment_class=FixFragment)  # The codec will produce FixFragment objects inside repeating groups

    # (default is dict). This is required for the find_all() and anywhere()
    # methods to work. It would fail with AttributeError otherwise.

    def fixmsg(*args, **kwargs):
        """
        Factory function. This allows us to keep the dictionary __init__
        arguments unchanged and force the codec to our given spec and avoid
        passing codec to serialisation and parsing methods.

        The codec defaults to a reasonable parser but without repeating groups.

        An alternative method is to use the ``to_wire`` and ``from_wire`` methods
        to serialise and parse messages and pass the codec explicitly.
        """
```

(continues on next page)

(continued from previous page)

```
        returned = FixMessage(*args, **kwargs)
        returned.codec = codec
        return returned

    ########################
    #    Vanilla tag/value
    ########################
    data = (b'8=FIX.4.2|9=97|35=6|49=ABC|56=CAB|34=14|52=20100204-09:18:42|'
            b'23=115685|28=N|55=BLAH|54=2|44=2200.75|27=S|25=H|10=248|')
    msg = fixmsg().load_fix(data, separator='|')

    # The message object is a dictionary, with integer keys
    # and string values. No validation is performed.
    # The spec contains the information, however so it can be retrieved.
    print("Message Type {} ({})".format(msg[35],
                                        spec.msg_types[msg[35]].name))
    print("Price {} (note type: {}, spec defined type {})".format(
        msg[44], type(msg[44]), spec.tags.by_tag(44).type
    ))

    check_tags = (55, 44, 27)
    for element, required in spec.msg_types[msg[35]].composition:
        if isinstance(element, FixTag) and element.tag in check_tags:
            if required:
                print("{} is required on this message type".format(element.name))
            else:
                print("{} is not required on this message type".format(element.name))
    print("Spec also allows looking up enums: {}  is {}".format(msg[54],
                                                               spec.tags.by_tag(54).
↪enum_by_value(msg[54])))

    # Although the values are stored as string there are rich operators provided that
    # allow to somewhat abstract the types
    print("exact comparison with decimal:", msg.tag_exact(44, decimal.Decimal("2200.75
↪")))
    print("exact comparing with int:", msg.tag_exact(54, 2))
    print("lower than with float:", msg.tag_lt(44, 2500.0))
    print("greater than with float:", msg.tag_gt(23, 110000.1))
    print("contains, case sensitive and insensitive:", msg.tag_contains(55, "MI"),␣
↪msg.tag_icontains(55, "blah"))

    # Tags manipulation is as for a dictionary
    msg[56] = "ABC.1"  # There is no enforcement of what tags are used for, so␣
↪changing 56 is no worry for the lib
    msg.update({55: 'ABC123.1', 28: 'M'})

    # note regex matching
    print("tag 56 changed", msg.tag_match_regex(56, r"..M\.N"))
    print("Tag 55 and 28 changed to {} and {}".format(msg[55], msg[28]))

    # There are additional tools for updating the messages' content though
    none_or_one = randint(0, 1) or None
    msg.set_or_delete(27, none_or_one)
    msg.apply({25: None, 26: 2})

    if none_or_one is None:
        print("Got None, the tag is deleted")
```

(continues on next page)

(continued from previous page)

```
        assert 27 not in msg
    else:
        print("Got None, the tag is maintained")
        assert msg[27] == 1

    assert 25 not in msg
    assert msg.tag_exact(26, '2')

    ########################
    #    copying messages
    ########################

    # Because messages maintain a reference to the codec and the spec
    # a deepcopy() of messages is extremely inefficient. It is a lot faster
    # to serialise-deserialise to copy a message, which is what copy() does.
    # Alternatively do a shallow copy through dict constructor.
    new_msg = msg.copy()
    assert new_msg is not msg
    msg.set_len_and_chksum()
    # Note that this reverts the type of values that were set manually
    assert new_msg[26] != msg[26]
    print("tag 26 before {}, after {}".format(type(msg[26]), type(new_msg[26])))
    assert all(new_msg.tag_exact(t, msg[t]) for t in msg)

    msg = fixmsg().load_fix(data, separator='|')

    # if no types have changed, the copy() method returns a message that is identical
    # to the original one
    new_msg = copy(msg)
    assert new_msg == msg

    # note you can also use the method copy()
    new_msg = msg.copy()
    assert new_msg == msg
    assert new_msg is not msg

    # and codec is not copied
    assert new_msg.codec is msg.codec

    # note that msg equality is not true when using dict constructor
    new_msg = FixMessage(msg)
    assert new_msg != msg
    # That's because codec, time, recipient and direction are not preserved
    # when using this technique, only tags are
    assert dict(new_msg) == dict(msg)

    ########################
    #    Repeating Groups
    ########################

    # Repeating groups are indexed by count tag (268 below)
    # and stored as a list of FixMessages (technically FixFragments, but close enough)
    data = (b'8=FIX.4.2|9=196|35=X|49=A|56=B|34=12|52=20100318-03:21:11.364'
            b'|262=A|268=2|279=0|269=0|278=BID|55=EUR/USD|270=1.37215'
            b'|15=EUR|271=2500000|346=1|279=0|269=1|278=OFFER|55=EUR/USD'
            b'|270=1.37224|15=EUR|271=2503200|346=1|10=171|')
    msg = fixmsg().load_fix(data, separator='|')
```

(continues on next page)

(continued from previous page)

```
    print("Message Type {} ({})".format(msg[35],
                                        spec.msg_types[msg[35]].name))
    print("Repeating group {} looks like {}".format(spec.tags.by_tag(268).name,
                                                    msg[268]))
    print("Accessing repeating groups at depth: tag 278 "
          "in the second member of the group is '{}'".format(msg[268][1][278]))

    # finding out if a tag is present can be done at depth
    print("Utility functions like anywhere() allow you to find out "
          "if a tag is present at depth, or find the path to it: {} is present : {}; "
          "it can be found at the following paths {}".format(278, msg.anywhere(278),
                                                             list(msg.find_all(278))))

    ########################
    # Customise the spec
    ########################
    # Sometimes it may be desirable to tweak the spec a bit, especially add a custom tag
    # or a custom repeating group.

    # Create tag
    spec.tags.add_tag(10001, "MyTagName")
    assert spec.tags.by_tag(10001).name == "MyTagName"
    assert spec.tags.by_name("MyTagName").tag == 10001
    # Change enum values
    spec.tags.by_tag(54).add_enum_value(name="SELLFAST", value="SF")
    assert spec.tags.by_tag(54).enum_by_value("SF") == "SELLFAST"

    # Add repeating Group
    # let's add repeating group 268 to msg type 'D'
    # for illustration purposes, we'll create the group from scratch
    data = (b'8=FIX.4.2|9=196|35=D|49=A|56=B|34=12|52=20100318-03:21:11.364'
            b'|262=A|268=2|279=0|269=0|278=BID|55=EUR/USD|270=1.37215'
            b'|15=EUR|271=2500000|346=1|279=0|269=1|278=OFFER|55=EUR/USD'
            b'|270=1.37224|15=EUR|271=2503200|346=1|10=171|')
    before = FixMessage()
    before.codec = Codec(spec=spec)
    before.load_fix(data, separator='|')

    # The composition is a iterable of pairs (FixTag, bool), with the bool indicating whether
    # the tag is required or not (although it's not enforced in the codec at this time
    composition = [(spec.tags.by_tag(i), False) for i in (279, 269, 278, 55, 270, 15, 271, 346)]

    # add_group takes a FixTag and the composition
    spec.msg_types['D'].add_group(spec.tags.by_tag(268), composition)

    after = FixMessage()
    after.codec = Codec(spec=spec, fragment_class=FixFragment)
    after.load_fix(data, separator='|')

    assert isinstance(before[268], (str, unicode))  # 268 is not parsed as a repeating group
    assert before[270] == '1.37224'  # 268 is not parsed as a repeating group, so 271 takes the second value

```

(continues on next page)

(continued from previous page)

```
    assert isinstance(after[268], RepeatingGroup)  # After the change, 268 becomes a repeating group
    assert list(after.find_all(270)) == [[268, 0, 270], [268, 1, 270]]  # and both 270 can be found


if __name__ == "__main__":
    PARSER = argparse.ArgumentParser()
    PARSER.add_argument("spec_xml")
    main(PARSER.parse_args().spec_xml)
```

Index

Symbols

A

C

D

E

F

G

L

O

P

R

S

T

U