SimpleFIX Documentation

Release 1.0.13

David Arnold

Sep 08, 2023

Contents:

CHAPTER 1

Introduction

FIX (Financial Information eXchange) Protocol is a widely-used, text-based protocol for interaction between parties in financial trading. Banks, brokers, clearing firms, exchanges, and other general market participants use FIX protocol for all phases of electronic trading.

Typically, a FIX implementation exists as a FIX Engine: a standalone service that acts as a gateway for other applications (matching engines, trading algos, etc) and implements the FIX protocol. The most popular Open Source FIX engine is probably one of the versions of QuickFIX.

This package provides a *simple* implementation of the FIX application-layer protocol. It does no socket handling, and does not implement FIX recovery or any message persistence. It supports the creation, encoding, and decoding of FIX messages.

CHAPTER 2

Installation

simplefix has a few dependencies. Firstly, it is known to run on Python 2.6, 2.7, 3.3, 3.4, 3.5, 3.6, and 3.7. It will not run on Python 2.5 or earlier.

You can install it using pip:

```
$ pip install simplefix
```

or using easy_install:

```
$ easy_install simplefix
```

It's usually a good idea to install simplefix into a virtualenv, to avoid issues with incompatible versions and system packaging schemes.

2.1 Getting the code

You can also get the code from PyPI or GitHub. You can either clone the public repository:

```
$ git clone git://github.com/da4089/simplefix.git
```

Download the tarball:

```
$ curl -OL https://github.com/da4089/simplefix/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/da4089/simplefix/zipball/master
```

Once you have a copy of the source you can install it into your site-packages easily:

```
$ python setup.py install
```

CHAPTER 3

Importing

You can import the *simplefix* module maintaining its internal structure, or you can import some or all bindings directly.

```
import simplefix

fix_msg = simplefix.Message()
```

Or

```
from simplefix import *

fix_msg = Message()
```

Note that the "import *" form is explicitly supported, with the exposed namespace explicitly managed to contain only the public features of the package.

All the example code in this document will use the first form, which is recommended.

CHAPTER 4

Creating Messages

To create a FIX message, first create an instance of the FixMessage class.

```
msg = simplefix.FixMessage()
```

You can then add fields to the message as required. You should add the standard header tags 8, 34, 35, 49, 52, and 56 to all messages, unless you're deliberately creating a malformed message for testing or similar.

4.1 Simple Fields

For most tags, using `append_pair()` is the easiest way to add a field to the message.

```
message.append_pair(1, "MC435967")
message.append_pair(54, 1)
message.append_pair(44, 37.0582)
```

Note that any type of value can be used: it will be explicitly converted to a string before encoding the message.

With a few exceptions, the message retains the order in which fields are added. The exceptions are that fields BeginString (8), Length (9), MsgType (35), and Checksum (10) are encoded in their required locations, regardless of what order they were added to the Message.

4.2 Header Fields

The Message class does not distinguish header fields from body fields, with one exception.

To enable fields to be added to the FIX header after body fields have already been added, there's an optional keyword parameter to the `append_pair()` method (and other append field methods). If this `header` parameter is set to `True`, the field is inserted after any previously added header fields, starting at the beginning of the message.

This is normally used for setting things like MsgSeqNum (34) and SendingTime (52) immediately prior to encoding and sending the message.

```
message.append_pair(8, "FIX.4.4")
message.append_pair(35, 0)
message.append_pair(49, "SENDER")
message.append_pair(56, "TARGET")
message.append_pair(112, "TR0003692")
message.append_pair(34, 4684, header=True)
message.append_time(52, header=True)
```

In the example above, field 34 would be inserted at the beginning of the message. After encoding, the order of fields would be: 8, 9, 35, 34, 52, 49, 56, 112, 10.

It's not necessary, but field 49 and 56 could also be written with `header` set `True`, in which case, they'd precede 34 ane 52 when encoded.

See `append_time()` below for details of that method.

4.3 Pre-composed Pairs

In some cases, your FIX application might have the message content as pre-composed "tag=value" strings. In this case, as an optimisation, the `append_string()` or `append_strings()` methods can be used.

```
BEGIN_STRING = "8=FIX.4.2"
STR_SEQ = ["49=SENDER", "56=TARGET"]

message.append_string(BEGIN_STRING, header=True)
message.append_strings(STR_SEQ, header=True)
```

As with `append_pair()`, note that these methods have an optional keyword parameter to ensure that their fields are inserted before body fields.

4.4 Timestamps

The FIX protocol defines four time types: UTCTimestamp, UTCTimeOnly, TZTimestamp, and TZTimeOnly. Field values of these types can be added using dedicated functions, avoiding the need to translate and format time values in the application code.

```
message.append_utc_timestamp(52, precision=6, header=True)
message.append_tz_timestamp(1132, my_datetime)
message.append_utc_time_only(1495, start_time)
message.append_tz_time_only(1079, maturity_time)
```

The first parameter to these functions is the field's tag number. The second parameter is optional: if None or not supplied, it defaults to the current time, otherwise it must be a Unix epoch time (like from `time.time()`), or a `datetime` instance.

There are two keyword parameters: `precision` which can be 0 for just seconds, 3 for milliseconds, or 6 for microseconds; and `header` to insert this field in the header rather than the body.

In addition, there are a set of methods for creating correctly formatted time only values from their components:

```
message.append_utc_time_only_parts(1495, 7, 0, 0, 0, 0)
message.append_tz_time_only_parts(1079, 20, 0, 0, offset=-300)
```

As usual, the first parameter to these functions is the field's tag number. The next three parameters are the hour, minute, and seconds of the time value, followed by optional milliseconds and microseconds values.

The timezone for the TZTimeOnly field is set using an offset value, the number of minutes east of UTC. Thus CET will be offset 60 minutes, and New York offset -240 minutes (four hours west).

Finally, remember that time fields can always be set using a string value if the application already has the value in the correct format or prefers to manage the formatting itself.

4.5 Repeating Groups

There is no specific support for creating repeating groups in Messages. The count field must be appended first, followed by the group's member's fields.

Consequently, it's not an error to append two fields with the same tag, but note that the count fields are not added automatically.

4.6 Data Fields

There are numerous defined fields in the FIX protocol that use the *data* type. These fields consist of two parts: a length, which must come first, immediately followed by the value field, whose value may include the ASCII SOH character, the ASCII NUL character, and in fact any 8-bit byte value.

To append a data field to a message, the `append_data()` method can be used. It will correctly add both the length field and the value field.

```
message.append_data(95, 96, "RAW DATA \x00\x01 VALUE")
```

which will result in the FIX message content (where represents the SOH):

95=1796=RAW DATA \x00\x01 VALUE

CHAPTER 5

Encoding

Once all fields are set, calling `encode()` will return a byte buffer containing the correctly formatted FIX message, with fields in the required order, and automatically added and set values for the BodyLength (9) and Checksum (10) fields.

```
byte_buffer = message.encode()
```

5.1 Raw Mode

Note that if you want to manually control the ordering of all fields, or the value of the BodyLength (9) or Checksum (10) fields, there's a 'raw' flag to the `encode()` method that disables the default automatic functionality.

```
byte_buffer = message.encode(True)
```

This is primarily useful for creating known-bad messages for testing purposes.

CHAPTER 6

Parsing Messages

To extract FIX messages from a byte buffer, such as that received from a socket, you should first create an instance of the `FixParser` class. For each byte string received, append it to the internal reassembly buffer using `append_buffer()`. At any time, you can call `get_message()`: if there's no complete message in the parser's internal buffer, it'll return None, otherwise, it'll return a `FixMessage` instance.

Once you've received a `FixMessage` from `get_message()`, you can: check the number of fields with `count()`, retrieve the value of a field using `get()` or the built-in "[ ]" syntax, or iterate over all the fields using "for ... in ...".

Members of repeating groups can be accessed using `get(tag, nth)`, where the "nth" value is an integer indicating the number of the group to use (note that the first group is number one, not zero).

CHAPTER 7

Indices and tables

- genindex
- modindex
- search