fitdecode

Release 0.6.0

Nov 02, 2019

Contents

A FIT file parsing and decoding library written in Python3 (3.6+ only).

CHAPTER 1

Usage Example

Read a FIT file, frame by frame:

```
import fitdecode

with fitdecode.FitReader(src_file) as fit:
    for frame in fit:
        # The yielded frame object is of one of the following types:
        # * fitdecode.FitHeader
        # * fitdecode.FitDefinitionMessage
        # * fitdecode.FitDataMessage
        # * fitdecode.FitCRC

        if isinstance(frame, fitdecode.FitDataMessage):
            # Here, frame is a FitDataMessage object.
            # A FitDataMessage object contains decoded values that
            # are directly usable in your script logic.
            print(frame.name)
```

CHAPTER 2

Installation

fitdecode is available on PyPI:

```
$ pip install fitdecode
```

Or, to get the latest working version, you can clone fitdecode's source code repository before installing it:

```
$ git clone git@github.com:polyvertex/fitdecode.git
$ cd fitdecode
$ python setup.py test     # optional step to run unit tests
$ python setup.py install
```

Note that for convenience, the `cmd` directory located at the root of the source code tree can safely be added to your `PATH`, so that fitdecode commands can be called without the package to be installed.

CHAPTER 3

Overview

fitdecode is a non offensive and incompatible rewrite of the fitparse library, with some improvements and additional features, as well as efforts made to optimize both speed and memory usage.

Main differences between fitdecode and fitparse:

- fitdecode requires Python version 3.6 or greater
- fitdecode is faster
- fitdecode allows concurrent reading of multiple files by being thread-safe, in the sense that fitdecode's objects keep their state stored locally
- fitdecode high-level interface - FitReader - is not compatible with fitparse's FitFile
- fitdecode does not discard the FIT header and the CRC footer while iterating a file, which allow to get a complete 1:1 representation of the file that is being read
- This also allows the client to easily deal with so-called chained FIT files, as per FIT SDK definition (i.e. concatenated FIT files)
- CRC computation and matching are both optional. CRC can be matched, only computed, or fully ignored for faster reading.
- fitdecode offers optional access to records, headers and footers in their binary form, to allow FIT file cutting, stitching and filtering at binary level

CHAPTER 4

Why a new library?

A new library has been created instead of just offering to patch fitparse because many changes and adds in fitdecode break fitparse's backward compatibilty and because it allowed more freedom during the development of fitdecode.

CHAPTER 5

Documentation

Documentation is available at http://fitdecode.readthedocs.io/

CHAPTER 6

License

This project is distributed under the terms of the MIT license. See the LICENSE.txt file for details.

CHAPTER 7

Credits

fitdecode is largely based on the generic approach adopted by fitparse to define FIT types and to decode raw values. That includes the module `profile.py` and all the classes it refers to, as well as the script `generate_profile.py`.

CHAPTER 8

Contents

8.1 Reference

8.1.1 reader

class fitdecode.reader.**CrcCheck**

Defines the values expected by the `check_crc` parameter of *FitReader*'s constructor.

DISABLED = 0

CRC is not computed at all (fastest). `fitdecode.FitCRC` frame will still be yielded if present in the source FIT stream, but with meaningless values. In which case data processor's `fitdecode.DataProcessorBase.on_crc` method will still be called as well.

READONLY = 1

CRC is computed but *FitReader* will never try to match CRCs. So no `fitdecode.FitCRCError` will ever be raised.

ENABLED = 2

CRC is computed and matched by *FitReader*. `fitdecode.FitCRCError` is raised upon incorrect CRC values.

class fitdecode.reader.**FitReader**(*fileish*, ***, *processor=<object object>*, *check_crc=<CrcCheck.ENABLED: 2>*, *keep_raw_chunks=False*, *data_bag=<object object>*)

Parse the content of a FIT stream or storage.

Transparently supports "chained FIT Files" as per SDK's definition. A *FitHeader* object is yielded during iteration to mark the beginning of each new "FIT File".

Usage:

```
import fitdecode

with fitdecode.FitReader(src_file) as fit:
    for frame in fit:
```

(continues on next page)

(continued from previous page)

```
        # The yielded frame object is of one of the following types:
        # * fitdecode.FitHeader
        # * fitdecode.FitDefinitionMessage
        # * fitdecode.FitDataMessage
        # * fitdecode.FitCRC

        if isinstance(frame, fitdecode.FitDataMessage):
            # Here, frame is a FitDataMessage object.
            # A FitDataMessage object contains decoded values that
            # are directly usable in your script logic.
            print(frame.name)
```

Data processing:

- You can specify your own data processor object using the *processor* argument.
- The argument can be left untouched so that *DefaultDataProcessor* is used.
- Otherwise, it can be set to None or any other false value to skip data processing entirely. This can speed up things a bit if your intent is only to manipulate the file at binary level (i.e. chunks), in which case *keep_raw_chunks* must be set to true.

Raw chunks:

- "raw chunk" or sometimes "frame", is the name given in fitdecode to the *bytes* block that represents one of the four FIT entities: *FitHeader*, *FitDefinitionMessage*, *FitDataMessage* and *FitCRC*.
- While iterating a file with *FitReader*, you can for instance cut, stitch and/or reconstruct the file being read by using the *FitChunk* object attached to any of the four aforementioned entities, as long as the *keep_raw_chunks* option is true.

Data bag:

- A *data_bag* object can be passed to the constructor and then be retrieved via the *data_bag* property.
- *data_bag* can be of any type (a dict by default) and will never be altered by this class.
- A "data bag" is useful if you wish to store some context-sensitive data during the decoding of a file.
- A typical use case is from a data processor that cannot hold its own context-sensitive data due to its instance being shared with other readers and/or by multiple threads (typically *DefaultDataProcessor*).

data_bag = None
the *data_bag* object that was passed to the constructor, or, by default, a dict object

processor
Read-only access to the data processor object.

last_header
The last read *FitHeader* object. May be None.

last_timestamp
The last timestamp value (int).

Often useful in FIT files since some data fields rely on it like timestamp_16 and timestamp_ms for instance.

Hint: you usually want to use this property from your own processor class derived from on of the processors available from *fitdecode.processors*.

file_id
The last read file_id *FitDataMessage* object. May be None.

local_mesg_defs
Read-only access to the dict of local message types of the current "FIT file".

It is cleared by *close()* (or __exit__()), and also each time a FIT file header is reached (i.e. at the beginning of a file, or after a *FitCRC*).

local_dev_types
Read-only access to the dict of developer types of the current "FIT file".

It is cleared by *close()* (or __exit__()), and also each time a FIT file header is reached (i.e. at the beginning of a file, or after a *FitCRC*).

close()
Close the file handle (constructor's *fileish*) and clear the internal state.

8.1.2 processors

fitdecode.processors.**FIT_UTC_REFERENCE = 631065600**
Datetimes (uint32) represent seconds since this FIT_UTC_REFERENCE (unix timestamp for UTC 00:00 Dec 31 1989).

fitdecode.processors.**FIT_DATETIME_MIN = 268435456**
date_time typed fields for which value is below FIT_DATETIME_MIN represent the number of seconds elapsed since device power on.

class fitdecode.processors.**DataProcessorBase**
Data processing base class.

This class does nothing. It is meant to be derived.

The following methods are called by fitdecode.FitReader:

- *on_header*, each time a fitdecode.FitHeader is reached
- *on_crc*, each time a fitdecode.FitCRC (the FIT footer) is reached

This is convenient if you wish to reset some context-sensitive state in- between two chained FIT files for example.

Bear in mind that a malformed/corrupted file may miss either of these entities (header and/or CRC footer).

Also, the following methods are called (still by fitdecode.FitReader) for each field of every data message, in that order:

- *on_process_type*
- *on_process_field*
- *on_process_unit*
- *on_process_message*

By default, the above processor methods call the following methods if they exist (hence the aforementioned caching):

```
def process_type_<type_name>(reader, field_data)
def process_field_<field_name>(reader, field_data)  # can be unknown_XYZ but NOT␣
↪recommended
def process_units_<unit_name>(reader, field_data)
def process_message_<mesg_name>(reader, data_message)
```

process_* methods are not expected to return any value and may alter the content of the passed *field_data* (fitdecode.FieldData) and *data_message* (fitdecode.FitDataMessage) arguments if needed.

See also:

DefaultDataProcessor, *StandardUnitsDataProcessor*

on_header(*reader*, *fit_header*)

on_crc(*reader*, *fit_crc*)

on_process_type(*reader*, *field_data*)

on_process_field(*reader*, *field_data*)

on_process_unit(*reader*, *field_data*)

on_process_message(*reader*, *data_message*)

class fitdecode.processors.**DefaultDataProcessor**

This is the default data processor used by fitdecode.FitReader. It derives from *DataProcessorBase*.

This data processor converts some raw values to more comfortable ones.

See also:

StandardUnitsDataProcessor, *DataProcessorBase*

process_type_bool(*reader*, *field_data*)

Just bool any bool typed FIT field unless value is None

process_type_date_time(*reader*, *field_data*)

Convert date_time typed field values into datetime.datetime object if possible.

That is, if value is not None and greater or equal than *FIT_DATETIME_MIN*.

The resulting datetime.datetime object is timezone-aware (UTC).

process_type_local_date_time(*reader*, *field_data*)

Convert date_time typed field values into datetime.datetime object unless value is None.

The resulting datetime.datetime object **IS NOT** timezone-aware, but this method assumes UTC at object construction to ensure consistency.

process_type_localtime_into_day(*reader*, *field_data*)

Convert localtime_into_day typed field values into datetime.time object unless value is None.

process_message_hr(*reader*, *data_message*)

Convert populated event_timestamp component values of the hr to datetime.datetime objects

class fitdecode.processors.**StandardUnitsDataProcessor**

A *DefaultDataProcessor* that also:

- Converts distance and total_distance fields to km (standard's default is m)
- Converts all speed and *_speeds fields (by name) to km/h (standard's default is m/s)
- Converts GPS coordinates (i.e. FIT's semicircles type) to deg

See also:

DefaultDataProcessor, *DataProcessorBase*

on_process_field(*reader*, *field_data*)
Convert all `*_speed` fields using *process_field_speed*.

All other units will use the default method.

process_field_distance(*reader*, *field_data*)

process_field_total_distance(*reader*, *field_data*)

process_field_speed(*reader*, *field_data*)

process_units_semicircles(*reader*, *field_data*)

8.1.3 records

class fitdecode.records.**FitChunk**(*index*, *offset*, *bytes*)

index
zero-based index of this frame in the file

offset
the offset at which this frame starts in the file

bytes
the frame itself as a *bytes* object

class fitdecode.records.**FitHeader**(*header_size*, *proto_ver*, *profile_ver*, *body_size*, *crc*, *crc_matched*, *chunk*)

frame_type = 1

header_size

proto_ver

profile_ver

body_size

crc
may be null

crc_matched

chunk
FitChunk or `None` (depends on `keep_raw_chunks` option)

class fitdecode.records.**FitCRC**(*crc*, *matched*, *chunk*)

frame_type = 2

crc

matched

chunk
FitChunk or `None` (depends on `keep_raw_chunks` option)

class fitdecode.records.**FitDefinitionMessage**(*is_developer_data*, *local_mesg_num*, *time_offset*, *mesg_type*, *global_mesg_num*, *endian*, *field_defs*, *dev_field_defs*, *chunk*)

frame_type = 3

is_developer_data

local_mesg_num

time_offset

mesg_type

global_mesg_num

endian

field_defs
list of *FieldDefinition*

dev_field_defs
list of *DevFieldDefinition*

chunk
FitChunk or `None` (depends on `keep_raw_chunks` option)

name

all_field_defs

class fitdecode.records.**FitDataMessage**(*is_developer_data*, *local_mesg_num*, *time_offset*, *def_mesg*, *fields*, *chunk*)

frame_type = 4

is_developer_data
Is this a "developer" message?

local_mesg_num
The **local** definition number of this message

time_offset
Time offset in case header was compressed. `None` otherwise.

def_mesg
FitDefinitionMessage

fields
list of *FieldData*

chunk
FitChunk or `None` (depends on `keep_raw_chunks` option)

name
Message name

global_mesg_num
The **global** definition number of this message

mesg_type
The *MessageType* object this message is associated to

has_field(*field_name_or_num*)
Is the desired field present in this message?

field_name_or_num is the name of the field (`str`), or its definition number (`int`).

See also:

get_field, *get_fields*, *get_value*, *get_values*

get_field(*field_name_or_num*, *idx=0*)
Get the desired `FieldData` object.

field_name_or_num is the name of the field (`str`), or its definition number (`int`).

idx is the zero-based index of the specified field **among other fields with the same name/number**. I.e. not the index of the field in the list of fields of this message. That is, `idx=0` is the first *field_name_or_num* field found in this message.

idx is useful in case a message contains multiple fields with the same *field_name_or_num*.

See also:

`get_fields`, `get_value`, `get_values`, `has_field`

get_fields(*field_name_or_num*)
Like `get_field` but **yield** every `FieldData` object matching *field_name_or_num* fields in this message - i.e. generator.

See also:

`get_field`, `get_value`, `get_values`, `has_field`

get_value(*field_name_or_num*, ***, *idx=0*, *fallback=<object object>*, *raw_value=False*, *fit_type=None*, *py_type=<object object>*)
Get the value (or raw_value) of a field specified by its name or its definition number (*field_name_or_num*), with optional type checking.

idx has the same meaning than for `get_field`.

fallback can be specified to avoid `KeyError` being raised in case no field matched *field_name_or_num*.

fit_type can be a `str` to indicate a given FIT type is expected (as defined in FIT profile; e.g. `date_time`, `manufacturer`, ...), in which case `TypeError` may be raised in case of a type mismatch.

py_type can be a Python type or a `tuple` of types to expect (as passed to `isinstance`), in which case `TypeError` may be raised in case of a type mismatch.

raw_value can be set to a true value so that the returned value is field's `raw_value` property instead of `value`. This does not impact the way *fit_type* and *py_type* are interpreted.

Special case: *field_name_or_num* can be `None`, in which case the field will be selected using *idx* only. In this case, *idx* is interpreted to be the zero-based index in the list of fields.

See also:

`get_values`, `get_field`, `get_fields`, `has_field`

get_values(*field_name_or_num*, ***, *raw_value=False*, *fit_type=None*, *py_type=<object object>*)
Like `get_value` but **yield** every value of all the fields that match *field_name_or_num* - i.e. generator.

It is not possible to specify a *fallback* value so `KeyError` will always be raised in case the specified field was not found.

The other arguments have the same meaning than for `get_value`.

See also:

`get_value`, `get_field`, `get_fields`, `has_field`

8.1.4 types

class fitdecode.types.**BaseType**(*name*, *identifier*, *fmt*, *parse*)

enum = None

name

identifier

fmt

size

parse

type_num
"Base Type Number" as per SDK definition

class fitdecode.types.**FieldType**(*name*, *base_type*, *enum=None*)

name

base_type

enum

class fitdecode.types.**Field**(*name*, *type*, *def_num*, *scale=None*, *offset=None*, *units=None*, *components=None*, *subfields=None*)

field_type = 'field'

name

type
FieldType

def_num

scale

offset

units

components

subfields

class fitdecode.types.**SubField**(*name*, *def_num*, *type*, *scale=None*, *offset=None*, *units=None*, *components=None*, *ref_fields=None*)

field_type = 'subfield'

name

def_num

type

scale

offset

units

components

ref_fields

class fitdecode.types.**DevField**(*dev_data_index*, *name*, *def_num*, *type*, *units*, *native_field_num*)

field_type = 'devfield'

dev_data_index

name

def_num

type

units

native_field_num

scale

offset

components

subfields

class fitdecode.types.**ReferenceField**(*name*, *def_num*, *value*, *raw_value*)

name

def_num

value

raw_value

class fitdecode.types.**ComponentField**(*name*, *def_num*, *scale=None*, *offset=None*, *units=None*, *accumulate=None*, *bits=None*, *bit_offset=None*)

field_type = 'component'

name

def_num

scale

offset

units

accumulate

bits

bit_offset

render(*raw_value*)

class fitdecode.types.**MessageType**(*name*, *mesg_num*, *fields*)

name

mesg_num

fields

class fitdecode.types.**FieldDefinition**(*field*, *def_num*, *base_type*, *size*)

field
Field

def_num

base_type

size

is_dev

name

type

class fitdecode.types.**DevFieldDefinition**(*field*, *dev_data_index*, *def_num*, *size*)

field

dev_data_index

def_num

size

base_type

is_dev

name

type

class fitdecode.types.**FieldData**(*field_def*, *field*, *parent_field*, *value*, *raw_value*, *units=None*)

field_def
FieldDefinition object

field

parent_field

value

raw_value

units

name
Field's name as defined in FIT global profile.

If name was not found in global profile, a string is created with the form: `unknown_{def_num}` where `def_num` is the field's definition number.

This value is **NOT** compatible with *is_named*.

See also:

name_or_num

name_or_num
Field's name as defined in FIT global profile.

If name was not found in global profile, `self.def_num` is returned (`int`).

This value is compatible with *is_named*.

See also:

name

def_num
Field's definition number (`int`)

base_type
Field's *BaseType*

is_base_type
Field's *BaseType*

type

field_type

is_expanded
Flag to indicate whether this field has been generated through expansion

is_named(*name_or_num*)
Check if this field has the specified name (`str`) or definition number (`int`)

`fitdecode.types.`**parse_string**(*byteslike*)

8.1.5 exceptions

exception `fitdecode.exceptions.`**FitError**

exception `fitdecode.exceptions.`**FitHeaderError**

exception `fitdecode.exceptions.`**FitCRCError**

exception `fitdecode.exceptions.`**FitEOFError**(*expected*, *got*, *offset*, *message=''*)

expected = None
number of expected bytes

got = None
number of bytes read

offset = None
the file offset from which reading took place

exception `fitdecode.exceptions.`**FitParseError**(*offset*, *message=''*)

offset = None
the file offset from which reading took place

8.1.6 utils

`fitdecode.utils.`**scrub_method_name**(*method_name*, *convert_units=False*)
Create a valid Python name out of *method_name*

`fitdecode.utils.`**get_mesg_type**(*mesg_name_or_num*)
Get a `fitdecode.MessageType` from `profile`, by its name or its global number.

Raise `ValueError` if type was not found.

`fitdecode.utils.`**get_mesg_num**(*mesg_name*)
Get the global number of a message as defined in `profile`, by its name

Raise `ValueError` if type was not found.

fitdecode.utils.**get_mesg_field**(*mesg_name_or_num*, *field_name_or_num*)
Get the *fitdecode.types.Field* object of a particular field from a particular message.

Raise ValueError if message or field was not found.

fitdecode.utils.**get_mesg_field_num**(*mesg_name_or_num*, *field_name*)
Get the definition number of a particular field from a particular message.

Raise ValueError if message or field was not found.

fitdecode.utils.**get_field_type**(*field_name*)
Get fitdecode.FieldType by name from profile.

Raise ValueError if type was not found.

fitdecode.utils.**compute_crc**(*byteslike*, *, *crc=0*, *start=0*, *end=None*)
Compute the CRC as per FIT definition, of *byteslike* object, from offset *start* (included) to *end* (excluded)

fitdecode.utils.**blocking_read**(*istream*, *size=-1*, *nonblocking_reads_delay=0.06*)
Read from *istream* and do not return until *size bytes* have been read unless EOF has been reached.

Return all the data read so far. The length of the returned data may still be less than *size* in case EOF has been reached.

nonblocking_reads_delay specifies the number of seconds (float) to wait before trying to read from *istream* again in case BlockingIOError has been raised during previous call.

8.2 Change Log

8.2.1 v0.6.0 (2019-11-02)

- Added *FitReader.last_timestamp* property
- Fixed: *FitReader* was raising KeyError instead of *FitParseError* when a dev_type was not found
- *FitParseError* message contains more details upon malformed file in some cases
- FIT SDK profile upgraded to v21.16
- README's usage example slightly improved

8.2.2 v0.5.0 (2019-04-11)

- Added fitdecode.DataProcessorBase class
- check_crc - the parameter to fitdecode.FitReader's constructor - can now be either "enabled", "read-only" or "disabled" (fix #1)
- Minor speed improvements

8.2.3 v0.4.0 (2019-04-10)

- Added fitdecode.FitDataMessage.has_field
- fitdecode.FitDataMessage.get_fields is now a generator
- fitdecode.FitDataMessage.get_values is now a generator

- `fitdecode.DefaultDataProcessor` now converts `hr.event_timestamp` values that were populated from `hr.event_timestamp_12` components to `datetime.datetime` objects for convenience
- `fitjson` and `fittxt` utilities: * Added support for input files with Unicode characters * Still write output file even if an error occurred while parsing FIT file
- Fixed handling of some FIT fields that are both scaled and components. See https://github.com/dtcooper/python-fitparse/issues/84
- Improved support for malformed FIT files. See https://github.com/dtcooper/python-fitparse/issues/62
- `generate_profile` utility slightly improved
- Added some unit tests
- Minor improvements and corrections

8.2.4 v0.3.0 (2018-07-27)

- Added *fitdecode.utils.get_mesg_field*
- Added *fitdecode.utils.get_mesg_field_num*
- Minor improvements and corrections

8.2.5 v0.2.0 (2018-07-16)

- Added *FieldData.name_or_num*
- Added *FitDataMessage.get_fields*
- Added *FitDataMessage.get_values*
- Improved *FitDataMessage.get_field* (*idx* arg)
- Improved *FitDataMessage.get_value* (*idx* arg)
- Completed documentation of *FitDataMessage*
- Improved documentation of *FieldData*
- *FitReader*'s internal state is reset as well after a *FitCRC* has been yielded (i.e. not only when a FIT header is about to be read), in order to avoid incorrect behavior due to malformed FIT stream

8.2.6 v0.1.0 (2018-07-14)

- Added class property `frame_type` (read-only) to *FitHeader*, *FitCRC*, *FitDefinitionMessage* and *FitDataMessage* (`records` module) to ease and speed up type checking
- Added *FitDataMessage.get_value* method
- `string` values with no null byte are still decoded (in full length)
- `cmd` directory added to the source code tree for convenience

8.2.7 v0.0.1 (2018-07-08)

- First release

8.2.8 v0.0.0 (2018-05-31)

- Birth!

8.3 To Do

```
* Project and CI
  * Pipenv
  * AppVeyor CI
  * .gitlab-ci.yml
  * Coverage.py???
  * Check for PyPy compliance once it's compatible with 3.6+ (3.5 only atm)
    and add it to .travis.yml

* Improve docs
```

CHAPTER 9

Indices and tables

- genindex
- modindex (API)

Python Module Index

f

Index

G

H

I

L

S

T

U

V