hexrec

Release 0.3.1

Andrea Zoppi

Feb 09, 2024

CONTENTS

CHAPTER

ONE

OVERVIEW

docs
tests
package

Library to handle hexadecimal record files

- Free software: BSD 2-Clause License

1.1 Introduction

The purpose of this library is to provide simple but useful methods to load, edit, and save hexadecimal record files.

In the field of embedded systems, hexadecimal record files are the most common way to share binary data to be written to the target non-volatile memory, such as a EEPROM or microcontroller code flash. Such binary data can contain compiled executable code, configuration data, volatile memory dumps, etc.

The most common file formats for hexadecimal record files are *Intel HEX* (.hex) and *Motorola S-record* (.srec). Other common formats for binary data exchange for embedded systems include the *Executable and Linkable Format* (.elf), hex dumps (by *hexdump* or *xxd*), and raw binary files (.bin).

A good thing about hexadecimal record files is that they are almost *de-facto*, so every time a supplier has to give away its binary data it is either in HEX or SREC, although ELF is arguably the most common for debuggable executables.

A bad thing is that their support in embedded software toolsets is sometimes flawed or only one of the formats is supported, while the supplier provides its binary data in the other format.

Another feature is that binary data is split into text record lines (thus their name) protected by some kind of checksum. This is good for data exchange and line-by-line writing to the target memory (in the old days), but this makes in-place editing by humans rather tedious as data should be split, and the checksum and other metadata have to be updated.

All of the above led to the development of this library, which allows to, for example:

- convert between hexadecimal record formats;
- merge/patch multiple hexadecimal record files of different formats;

- access every single record of a hexadecimal record file;
- build records through handy methods;
- edit sparse data in a virtual memory behaving like a `bytearray`;
- extract or update only some parts of the binary data.

1.2 Documentation

For the full documentation, please refer to:

https://hexrec.readthedocs.io/

1.3 Architecture

Within the `hexrec` package itself are the symbols of the most commonly used classes and functions.

As the core of this library are record files, the `hexrec.records` is the first module a user should look up. It provides high-level functions to deal with record files, as well as classes holding record data.

The `hexrec.records` allows to load `bytesparse` virtual memories, which are as easy to use as the native `bytearray`, but with sparse data blocks.

The `hexrec.utils` module provides some miscellaneous utility stuff.

`hexrec.xxd` is an emulation of the `xxd` command line utility delivered by `vim`.

The package can also be run as a command line tool, by running the `hexrec` package itself (`python -m hexrec`), providing some record file utilities. You can also create your own standalone executable, or download a precompiled one from the `pyinstaller` folder.

The codebase is written in a simple fashion, to be easily readable and maintainable, following some naive pythonic *K.I.S.S.* approach by choice.

1.4 Examples

To have a glimpse of the features provided by this library, some simple but common examples are shown in the following.

1.4.1 Convert format

It happens that some software tool only supports some hexadecimal record file formats, or the format given to you is not handled properly, or simply you prefer a format against another (*e.g.* SREC has *linear* addressing, while HEX is in a *segment:offset* fashion).

In this example, a HEX file is converted to SREC.

```
from hexrec import convert_file
convert_file('data.hex', 'data.srec')
```

This can also be done by running *hexrec* as a command line tool:

```
$ hexrec convert data.hex data.srec
```

Alternatively, by executing the package itself:

```
$ python -m hexrec convert data.hex data.srec
```

1.4.2 Merge files

It is very common that the board factory prefers to receive a single file to program the microcontroller, because a single file is simpler to manage for them, and might be faster for their workers or machine, where every second counts.

This example shows how to merge a bootloader, an executable, and some configuration data into a single file, in the order they are listed.

```
from hexrec import merge_files
input_files = ['bootloader.hex', 'executable.mot', 'configuration.s19']
merge_files(input_files, 'merged.srec')
```

This can also be done by running the *hexrec* package as a command line tool:

```
$ hexrec merge bootloader.hex executable.mot configuration.s19 merged.srec
```

Alternatively, these files can be merged manually via *virtual memory*:

```
from hexrec import load_memory, save_memory
from bytesparse import bytesparse
input_files = ['bootloader.hex', 'executable.mot', 'configuration.s19']
input_memories = [load_memory(filename) for filename in input_files]
merged_memory = bytesparse()
for input_memory in input_memories:
    merged_memory.write(0, input_memory)
save_memory('merged.srec', merged_memory)
```

1.4.3 Dataset generator

Let us suppose we are early in the development of the embedded system and we need to test the current executable with some data stored in EEPROM. We lack the software tool to generate such data, and even worse we need to test 100 configurations. For the sake of simplicity, the data structure consists of 4096 random values (0 to 1) of `float` type, stored in little-endian at the address `0xDA7A0000`.

```
import struct, random
from hexrec import save_chunk
for index in range(100):
    values = [random.random() for _ in range(4096)]
    data = struct.pack('<4096f', *values)
    save_chunk(f'dataset_{index:02d}.mot', data, 0xDA7A0000)
```

1.4.4 Write a CRC

Usually, the executable or the configuration data of an embedded system are protected by a CRC, so that their integrity can be self-checked.

Let us suppose that for some reason the compiler does not calculate such CRC the expected way, and we prefer to do it with a script.

This example shows how to load a HEX file, compute a CRC32 from the address `0x1000` to `0x3FFB` (`0x3FFC` exclusive), and write the calculated CRC to `0x3FFC` in big-endian as a SREC file. The rest of the data is left untouched.

```
import binascii, struct
from hexrec import save_memory
memory = load_memory('data.srec')
crc = binascii.crc32(memory[0x1000:0x3FFC]) & 0xFFFFFFFF  # remove sign
memory.write(0x3FFC, struct.pack('>L', crc))
save_memory('data_crc.srec', memory)
```

1.4.5 Trim for bootloader

When using a bootloader, it is very important that the application being written does not overlap with the bootloader. Sometimes the compiler still generates stuff like a default interrupt table which should reside in the bootloader, and we need to get rid of it, as well as everything outside the address range allocated for the application itself.

This example shows how to trim the application executable record file to the allocated address range `0x8000-0x1FFFF`. Being written to a flash memory, unused memory byte cells default to `0xFF`.

```
from hexrec import save_chunk
memory = load_memory('app_original.hex')
data = memory[0x8000:0x20000:b'\xFF']
save_chunk('app_trimmed.srec', data, 0x8000)
```

This can also be done by running the *hexrec* package as a command line tool:

```
$ hexrec cut -s 0x8000 -e 0x20000 -v 0xFF app_original.hex app_trimmed.srec
```

By contrast, we need to fill the application range within the bootloader image with `0xFF`, so that no existing application will be available again. Also, we need to preserve the address range `0x3F800-0x3FFFF` because it already contains some important data.

```
from hexrec import load_memory, save_memory
memory = load_memory('boot_original.hex')
memory.fill(0x8000, 0x20000, b'\xFF')
memory.clear(0x3F800, 0x40000)
save_memory('boot_fixed.srec', memory)
```

With the command line interface, it can be done via a two-pass processing, first to fill the application range, then to clear the reserved range. Please note that the first command is chained to the second one via standard output/input buffering (the virtual `-` file path, in `intel` format as per `boot_original.hex`).

```
$ hexrec fill -s 0x8000 -e 0x20000 -v 0xFF boot_original.hex - | \
  hexrec clear -s 0x3F800 -e 0x40000 -i intel - boot_fixed.srec
```

(newline continuation is backslash \ for a *Unix-like* shell, caret ^ for a *DOS* prompt).

1.4.6 Export ELF physical program

The following example shows how to export *physical program* stored within an *Executable and Linkable File* (*ELF*), compiled for a microcontroller. As per the previous example, only data within the range `0x8000-0x1FFFF` are kept, with the rest of the memory filled with the `0xFF` value.

```
from hexrec import save_memory
from bytesparse import bytesparse
from elftools.elf.elffile import ELFFile
with open('app.elf', 'rb') as elf_stream:
    elf_file = ELFFile(elf_stream)
    memory = bytesparse(start=0x8000, endex=0x20000)  # bounds set
    memory.fill(pattern=b'\xFF')  # between bounds
    for segment in elf_file.iter_segments(type='PT_LOAD'):
        addr = segment.header.p_paddr
        data = segment.data()
        memory.write(addr, data)
save_memory('app.srec', memory)
```

1.5 Installation

From PyPI (might not be the latest version found on *github*):

```
$ pip install hexrec
```

From the source code root directory:

```
$ pip install .
```

1.6 Development

To run the all the tests:

```
$ pip install tox
$ tox
```

CHAPTER

TWO

INSTALLATION

2.1 From *PyPI*

At the command line:

```
$ pip install hexrec
```

The package found on *PyPI* might be outdated with respect to the source repository.

2.2 From source

At the command line, at the root of the source directory:

```
$ pip install .
```

CHAPTER

THREE

COMMAND LINE INTERFACE

3.1 hexrec

A set of command line utilities for common operations with record files.

Being built with Click, all the commands follow POSIX-like syntax rules, as well as reserving the virtual file path - for command chaining via standard output/input buffering.

```
hexrec [OPTIONS] COMMAND [ARGS]...
```

3.1.1 clear

Clears an address range.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec clear [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.2 convert

Converts a file to another format.

INFILE is the list of paths of the input files. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec convert [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.3 crop

Selects data from an address range.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec crop [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-v, --value <value>

Byte value used to flood the address range. By default, no flood is performed.

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.4 cut

Selects data from an address range.

DEPRECATED: Use the *crop* command instead.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec cut [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-v, --value <value>

Byte value used to flood the address range. By default, no flood is performed.

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.5 delete

Deletes an address range.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec delete [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.6 fill

Fills an address range with a byte value.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec fill [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-v, --value <value>

Byte value used to fill the address range.

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.7 flood

Fills emptiness of an address range with a byte value.

`INFILE` is the path of the input file. Set to `-` to read from standard input; input format required.

`OUTFILE` is the path of the output file. Set to `-` to write to standard output.

```
hexrec flood [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-v, --value <value>

Byte value used to flood the address range.

-s, --start <start>

Inclusive start address. Negative values are referred to the end of the data. By default it applies from the start of the data contents.

-e, --endex <endex>

Exclusive end address. Negative values are referred to the end of the data. By default it applies till the end of the data contents.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.8 merge

Merges multiple files.

`INFILES` is the list of paths of the input files. Set any to `-` to read from standard input; input format required.

`OUTFILE` is the path of the output file. Set to `-` to write to standard output.

Every file of `INFILES` will overwrite data of previous files of the list where addresses overlap.

```
hexrec merge [OPTIONS] INFILES... OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

Arguments

INFILES

Required argument(s)

OUTFILE

Required argument

3.1.9 motorola

Motorola SREC specific

```
hexrec motorola [OPTIONS] COMMAND [ARGS]...
```

del-header

Deletes the header data record.

INFILE is the path of the input file; 'motorola' record type. Set to - to read from standard input.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec motorola del-header [OPTIONS] INFILE OUTFILE
```

Arguments

INFILE

Required argument

OUTFILE

Required argument

get-header

Gets the header data.

INFILE is the path of the input file; 'motorola' record type. Set to - to read from standard input.

```
hexrec motorola get-header [OPTIONS] INFILE
```

Options

-f, --format <format>

Header data format.

Options

ascii | hex | HEX | hex. | HEX. | hex- | HEX- | hex: | HEX: | **hex_** | **HEX_** | hex | HEX

Arguments

INFILE

Required argument

set-header

Sets the header data record.

INFILE is the path of the input file; 'motorola' record type. Set to - to read from standard input.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec motorola set-header [OPTIONS] HEADER INFILE OUTFILE
```

Options

-f, --format <format>

Header data format.

Options

ascii | hex | HEX | hex. | HEX. | hex- | HEX- | hex: | HEX: | **hex_** | **HEX_** | hex | HEX

Arguments

HEADER

Required argument

INFILE

Required argument

OUTFILE

Required argument

3.1.10 reverse

Reverses data.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec reverse [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options
ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options
ascii_hex | binary | intel | mos | motorola | tektronix

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.11 shift

Shifts data addresses.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

OUTFILE is the path of the output file. Set to - to write to standard output.

```
hexrec shift [OPTIONS] INFILE OUTFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options
ascii_hex | binary | intel | mos | motorola | tektronix

-o, --output-format <output_format>

Forces the output file format. By default it is that of the input file.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

-n, --amount <amount>

Address shift to apply.

Arguments

INFILE

Required argument

OUTFILE

Required argument

3.1.12 validate

Validates a record file.

INFILE is the path of the input file. Set to - to read from standard input; input format required.

```
hexrec validate [OPTIONS] INFILE
```

Options

-i, --input-format <input_format>

Forces the input file format. Required for the standard input.

Options

ascii_hex | binary | intel | mos | motorola | tektronix

Arguments

INFILE

Required argument

3.1.13 xxd

Emulates the xxd command.

Please refer to the xxd manual page to know its features and caveats.

Some parameters were changed to satisfy the POSIX-like command line parser.

```
hexrec xxd [OPTIONS] INFILE OUTFILE
```

Options

-a, --autoskip

Toggles autoskip.

A single '*' replaces null lines.

-b, --bits

Binary digits.

Switches to bits (binary digits) dump, rather than hexdump. This option writes octets as eight digits of '1' and '0' instead of a normal hexadecimal dump. Each line is preceded by a line number in hexadecimal and followed by an ASCII (or EBCDIC) representation. The argument switches -r, -p, -i do not work with this mode.

-c, --cols <cols>

Formats <cols> octets per line. Max 256.

Defaults: normal 16, -i 12, -p 30, -b 6.

-E, --ebcdic, --EBCDIC

Uses EBCDIC charset.

Changes the character encoding in the right-hand column from ASCII to EBCDIC. This does not change the hexadecimal representation. The option is meaningless in combinations with -r, -p or -i.

-e, --endian

Switches to little-endian hexdump.

This option treats byte groups as words in little-endian byte order. The default grouping of 4 bytes may be changed using -g. This option only applies to hexdump, leaving the ASCII (or EBCDIC) representation unchanged. The switches -r, -p, -i do not work with this mode.

-g, --groupsize <groupsize>

Byte group size.

Separates the output of every <groupsize> bytes (two hex characters or eight bit-digits each) by a whitespace. Specify <groupsize> 0 to suppress grouping. <groupsize> defaults to 2 in normal mode, 4 in little-endian mode and 1 in bits mode. Grouping does not apply to -p or -i.

-i, --include

Output in C include file style.

A complete static array definition is written (named after the input file), unless reading from standard input.

-l, --length, --len <length>

Stops after writing <length> octets.

-o, --offset <offset>

Adds <offset> to the displayed file position.

-p, --postscript, --plain, --ps

Outputs in postscript continuous hexdump style.

Also known as plain hexdump style.

-q, --quadword

Uses 64-bit addressing.

-r, --revert

Reverse operation.

Convert (or patch) hexdump into binary. If not writing to standard output, it writes into its output file without truncating it. Use the combination -r and -p to read plain hexadecimal dumps without line number information and without a particular column layout. Additional Whitespace and line breaks are allowed anywhere.

-k, --seek <oseek>

Output seeking.

When used after -r reverts with -o added to file positions found in hexdump.

-s <iseek>

Input seeking.

Starts at <s> bytes absolute (or relative) input offset. Without -s option, it starts at the current file position. The prefix is used to compute the offset. '+' indicates that the seek is relative to the current input position. '-' indicates that the seek should be that many characters from the end of the input. '+-' indicates that the seek should be that many characters before the current stdin file position.

-U, --upper-all

Uses upper case hex letters on address and data.

-u, --upper

Uses upper case hex letters on data only.

-v, --version

Prints the package version number.

Arguments

INFILE

Required argument

OUTFILE

Required argument

CHAPTER

FOUR

REFERENCE

hexrec	
hexrec.formats	Record formats
hexrec.formats.ascii_hex	ASCII-hex format.
hexrec.formats.binary	Binary format.
hexrec.formats.intel	Intel HEX format.
hexrec.formats.mos	MOS Technology format.
hexrec.formats.motorola	Motorola S-record format.
hexrec.formats.tektronix	Tektronix extended HEX format.
hexrec.records	Hexadecimal record management.
hexrec.utils	Generic utility functions.
hexrec.xxd([infile, outfile, autoskip, ...])	Emulation of the xxd utility core.

4.1 hexrec

4.2 hexrec.formats

Record formats

This is a collection of commonly used hexadecimal record file types, and the like.

4.3 hexrec.formats.ascii_hex

ASCII-hex format.

See also:

https://srecord.sourceforge.net/man/man5/srec_ascii_hex.5.html

Classes

Record	ASCII-hex record.

4.3.1 hexrec.formats.ascii_hex.Record

class hexrec.formats.ascii_hex.**Record**(*address*, *tag*, *data*, *checksum=None*)

ASCII-hex record.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.
- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_data	Builds a data record.
build_standalone	Makes a sequence of data records standalone.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.
REGEX	Regular expression for parsing a record text line.
TAG_TYPE	Associated Python class for tags.

EXTENSIONS: Sequence[str] = ()

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

REGEX = re.compile("^(\\$A(?P<address>[0-9A-Fa-f]{4})[,.][%',]?)?(?P<data>([0-9A-Fa-f]{2}[%',])*([0-9A-Fa-f]{2})?)(\\$S(?P<checksum>[0-9A-Fa-f]{4})[,.][%',]?)?$")

Regular expression for parsing a record text line.

TAG_TYPE: Optional[Type[*Tag*]] = None

Associated Python class for tags.

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=None*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns
str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__

list of weak references to the object (if defined)

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters
path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_data(*address*, *data*)

Builds a data record.

Parameters

- **address** (*int*) – Record address, or None.
- **data** (*bytes*) – Record data, or None.

Returns
record – A data record.

Examples

```
>>> Record.build_data(0x1234, b'Hello, World!')
...
Record(address=0x1234, tag=None, count=13,
       data=b'Hello, World!', checksum=None)
```

```
>>> Record.build_data(None, b'Hello, World!')
...
Record(address=None, tag=None, count=13,
       data=b'Hello, World!', checksum=None)
```

```
>>> Record.build_data(0x1234, None)
...
Record(address=0x1234, tag=None, count=0, data=None, checksum=None)
```

classmethod build_standalone(*data_records*, **args*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of records*) – Sequence of data records.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Yields

record – Records for a standalone record file.

check()

Performs consistency checks.

Raises

ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters

records (*list of records*) – Sequence of records.

Raises

ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns

int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns

bool – count field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters
records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is `None`.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

is_data()

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

classmethod load_blocks(*path*)

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
`Memory` – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
bytes or str – Data for output, according to the file type.

`overlaps`(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters
other (*record*) – Record to compare with *self*.

Returns
bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

`classmethod parse_record`(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
record – Parsed record.

Note: This method must be overridden.

`classmethod read_blocks`(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters

records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (Memory) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, *address=0*, *columns=16*, *align=Ellipsis*, *standalone=True*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. Maximum of 128 columns.
- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.

Yields

record – Data split into records.

Raises

ValueError – Address, size, or column overflow.

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (*Memory*) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.4 hexrec.formats.binary

Binary format.

This format is actually used to hold binary chunks of raw data (*bytes*).

Classes

Record	Binary record.

4.4.1 hexrec.formats.binary.Record

class hexrec.formats.binary.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)

Binary record.

This record type is actually just a container for binary data.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.

- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_data	Builds a data record.
build_standalone	Makes a sequence of data records standalone.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a hexadecimal record line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.
TAG_TYPE	Associated Python class for tags.

EXTENSIONS: Sequence[str] = ('.bin', '.dat', '.raw')

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = b''

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

TAG_TYPE: Optional[Type[*Tag*]] = None

Associated Python class for tags.

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns

str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__

list of weak references to the object (if defined)

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters

path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_data(*address*, *data*)

Builds a data record.

Parameters

- **address** (*int*) – Data address.
- **data** (*bytes*) – Record data.

Returns
record – Data record.

Example

```
>>> Record.build_data(0x1234, b'Hello, World!')
...
Record(address=0x00001234, tag=0, count=13,
       data=b'Hello, World!', checksum=0x69)
```

classmethod build_standalone(*data_records*, **args*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of records*) – Sequence of data records.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Yields
record – Records for a standalone record file.

check()

Performs consistency checks.

Raises
ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters
records (*list of records*) – Sequence of records.

Raises
ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns

int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D000000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns

bool – *count* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
```

(continues on next page)

(continued from previous page)

```
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters
records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is `None`.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

is_data()

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

classmethod load_blocks(*path*)

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
Memory – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal()

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

bytes or str – Data for output, according to the file type.

overlaps(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters

other (*record*) – Record to compare with *self*.

Returns

bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

classmethod parse_record(*line*, **args*, ***kwargs*)

Parses a hexadecimal record line.

Parameters

line (*str*) – Text line to parse.

Returns

record – Parsed record.

Warning: Since it parses raw hex data, it is not possible to set address to a value different than `0`.

Example

```
>>> line = '48656C6C 6F2C2057 6F726C64 21'
>>> Record.parse_record(line)
...
Record(address=0x00000000, tag=0, count=13,
       data=b'Hello, World!', checksum=0x69)
```

classmethod read_blocks(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters
records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.

- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (Memory) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, *address=0*, *columns=None*, *align=Ellipsis*, *standalone=True*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. If `None`, the whole *data* is put into a single record.
- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.

Yields

record – Data split into records.

Raises

ValueError – Address or size overflow.

classmethod unmarshal(*data*, **args*, *address=0*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (*Memory*) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.5 hexrec.formats.intel

Intel HEX format.

See also:

https://en.wikipedia.org/wiki/Intel_HEX

Classes

Record	Intel HEX record.
Tag	Intel HEX tag.

4.5.1 hexrec.formats.intel.Record

class hexrec.formats.intel.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)

Intel HEX record.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.

- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_data	Builds a data record.
build_end_of_file	Builds an end-of-file record.
build_extended_linear_address	Builds an extended linear address record.
build_extended_segment_address	Builds an extended segment address record.
build_standalone	Makes a sequence of data records standalone.
build_start_linear_address	Builds an start linear address record.
build_start_segment_address	Builds an start segment address record.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
terminate	Builds a record termination sequence.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	Automatically supported file extensions.
LINE_SEP	Separator between record lines.
REGEX	Regular expression for parsing a record text line.

EXTENSIONS: Sequence[str] = ('.hex', '.ihex', '.mcs')

Automatically supported file extensions.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

REGEX = re.compile('^:(?P<count>[0-9A-Fa-f]{2})(?P<offset>[0-9A-Fa-f]{4})(?P<tag>[0-9A-Fa-f]{2})(?P<data>([0-9A-Fa-f]{2}){,255})(?P<checksum>[0-9A-Fa-f]{2})$')

Regular expression for parsing a record text line.

TAG_TYPE

Associated Python class for tags.

alias of *Tag*

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns

str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__

list of weak references to the object (if defined)

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters

path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_data(*address*, *data*)

Builds a data record.

Parameters

- **address** (*int*) – Record address.
- **data** (*bytes*) – Record data.

Returns
record – Data record.

Example

```
>>> str(Record.build_data(0x1234, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

classmethod build_end_of_file()

Builds an end-of-file record.

Returns
record – End-of-file record.

Example

```
>>> str(Record.build_end_of_file())
':00000001FF'
```

classmethod build_extended_linear_address(*address*)

Builds an extended linear address record.

Parameters
address (*int*) – Extended linear address. The 16 least significant bits are ignored.

Returns
record – Extended linear address record.

Raises
ValueError – Address overflow.

Example

```
>>> str(Record.build_extended_linear_address(0x12345678))
':020000041234B4'
```

classmethod build_extended_segment_address(*address*)

Builds an extended segment address record.

Parameters
address (*int*) – Extended segment address. The 20 least significant bits are ignored.

Returns
record – Extended segment address record.

Example

```
>>> str(Record.build_extended_segment_address(0x12345678))
':020000020123D8'
```

classmethod build_standalone(*data_records*, *start=Ellipsis*, **args*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of records*) – A sequence of data records.
- **start** (*int*) – Program start address. If `Ellipsis`, it is assigned the minimum data record address. If `None`, the start address records are not output.

Yields
record – Records for a standalone record file.

classmethod build_start_linear_address(*address*)

Builds an start linear address record.

Parameters
address (*int*) – Start linear address.

Returns
record – Start linear address record.

Raises
ValueError – Address overflow.

Example

```
>>> str(Record.build_start_linear_address(0x12345678))
':0400000512345678E3'
```

classmethod build_start_segment_address(*address*)

Builds an start segment address record.

Parameters
address (*int*) – Start segment address.

Returns
record – Start segment address record.

Raises
ValueError – Address overflow.

Example

```
>>> str(Record.build_start_segment_address(0x12345678))
':0400000312345678E5'
```

check()
Performs consistency checks.

Raises
ValueError – a field is inconsistent.

classmethod check_sequence(*records*)
Consistency check of a sequence of records.

Parameters
records (*list of records*) – Sequence of records.

Raises
ValueError – A field is inconsistent.

compute_checksum()
Computes the checksum.

Returns
int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns

bool – *count* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters

records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is None.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

`is_data()`

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

`classmethod load_blocks(path)`

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
Memory – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

bytes or str – Data for output, according to the file type.

overlaps(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters

other (*record*) – Record to compare with *self*.

Returns

bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
```

(continues on next page)

(continued from previous page)

```
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

classmethod parse_record(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Parsed record.

Note: This method must be overridden.

---

classmethod read_blocks(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

`Memory` – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Intel HEX, stores records by *segment/offset* addressing. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters

records (*list of records*) – Sequence of records to be converted to *flat* addressing, in-place.

Example

```
>>> records = [
...     Record.build_extended_linear_address(0x76540000),
...     Record.build_data(0x00003210, b'Hello, World!'),
... ]
>>> records
[Record(address=0x00000000,
             tag=<Tag.EXTENDED_LINEAR_ADDRESS: 4>, count=2,
             data=b'vT', checksum=0x30),
 Record(address=0x00003210, tag=<Tag.DATA: 0>, count=13,
             data=b'Hello, World!', checksum=0x48)]
>>> Record.readdress(records)
>>> records
[Record(address=0x76540000,
             tag=<Tag.EXTENDED_LINEAR_ADDRESS: 4>, count=2,
             data=b'vT', checksum=0x30),
 Record(address=0x76543210, tag=<Tag.DATA: 0>, count=13,
             data=b'Hello, World!', checksum=0x48)]
```

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (Memory) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, *address=0*, *columns=16*, *align=Ellipsis*, *standalone=True*, *start=None*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. Maximum of 255 columns.
- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.
- **start** (*int*) – Program start address. If `Ellipsis`, it is assigned the minimum data record address. If `None`, no start address records are output.

Yields

record – Data split into records.

Raises

ValueError – Address, size, or column overflow.

classmethod terminate(*start=None*)

Builds a record termination sequence.

The termination sequence is made of:

An extended linear address record at `0`. # A start linear address record at *start*. # An end-of-file record.

Parameters

start (*int*) – Program start address. If `None`, the start address records are not output.

Returns

list of records – Termination sequence.

Example

```
>>> list(map(str, Record.terminate(0x12345678)))
[':020000040000FA', ':0400000512345678E3', ':00000001FF']
```

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (Memory) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.5.2 hexrec.formats.intel.Tag

class hexrec.formats.intel.**Tag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Intel HEX tag.

Methods

__init__	
as_integer_ratio	Return integer ratio.
bit_count	Number of ones in the binary representation of the absolute value of self.
bit_length	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes	Return the integer represented by the given array of bytes.
to_bytes	Return an array of bytes representing an integer.

Attributes

DATA	Binary data.
END_OF_FILE	End of file.
EXTENDED_SEGMENT_ADDRESS	Extended segment address.
START_SEGMENT_ADDRESS	Start segment address.
EXTENDED_LINEAR_ADDRESS	Extended linear address.
START_LINEAR_ADDRESS	Start linear address.
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

DATA = 0

Binary data.

END_OF_FILE = 1

End of file.

EXTENDED_LINEAR_ADDRESS = 4

Extended linear address.

EXTENDED_SEGMENT_ADDRESS = 2

Extended segment address.

START_LINEAR_ADDRESS = 5

Start linear address.

START_SEGMENT_ADDRESS = 3

Start segment address.

__abs__()
abs(self)

__add__(*value*, /)
Return self+value.

__and__(*value*, /)
Return self&value.

__bool__()
True if self else False

__ceil__()
Ceiling of an Integral returns itself.

classmethod __contains__(*member*)
Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

__dir__()
Returns all members and all public methods

__divmod__(*value*, /)
Return divmod(self, value).

__eq__(*value*, /)
Return self==value.

__float__()
float(self)

__floor__()
Flooring an Integral returns itself.

__floordiv__(*value*, /)
Return self//value.

__format__(*format_spec*, /)
Default object formatter.

__ge__(*value*, /)
Return self>=value.

__getattribute__(*name*, /)
Return getattr(self, name).

classmethod __getitem__(*name*)
Return the member matching *name*.

__gt__(*value*, /)
Return self>value.

__hash__()
Return hash(self).

__index__()
Return self converted to an integer, if self is suitable for use as an index into a list.

__init__(**args*, ***kwds*)

__int__()
int(self)

__invert__()
~self

classmethod __iter__()
Return members in definition order.

__le__(*value*, /)
Return self<=value.

classmethod __len__()
Return the number of members (no aliases)

__lshift__(*value*, /)
Return self<<value.

__lt__(*value*, /)
Return self<value.

__mod__(*value*, /)
Return self%value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__neg__()
-self

__new__(*value*)

__or__(*value*, /)
Return self|value.

__pos__()
+self

__pow__(*value*, *mod=None*, /)
Return pow(self, value, mod).

__radd__(*value*, /)
Return value+self.

__rand__(*value*, /)
Return value&self.

__rdivmod__(*value*, /)
Return divmod(value, self).

__reduce_ex__(*proto*)
Helper for pickle.

__repr__()
Return repr(self).

__rfloordiv__(*value*, /)
Return value//self.

__rlshift__(*value*, /)
Return value<<self.

__rmod__(*value*, /)
Return value%self.

__rmul__(*value*, /)
Return value*self.

__ror__(*value*, /)
Return value|self.

__round__()
Rounding an Integral returns itself.

Rounding with an ndigits argument also returns an integer.

__rpow__(*value*, *mod=None*, /)
Return pow(value, self, mod).

__rrshift__(*value*, /)
Return value>>self.

__rshift__(*value*, /)
Return self>>value.

__rsub__(*value*, /)
Return value-self.

__rtruediv__(*value*, /)
Return value/self.

__rxor__(*value*, /)
Return value^self.

__sizeof__()
Returns size in memory, in bytes.

__str__()
Return repr(self).

__sub__(*value*, /)
Return self-value.

__truediv__(*value*, /)
Return self/value.

__trunc__()
Truncating an Integral returns itself.

__xor__(*value*, /)
Return self^value.

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, *, *signed=False*)

Return the integer represented by the given array of bytes.

bytes
: Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
: The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
: Indicates whether two's complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator
the numerator of a rational number in lowest terms

real
the real part of a complex number

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)
Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

4.6 hexrec.formats.mos

MOS Technology format.

See also:

https://srecord.sourceforge.net/man/man5/srec_mos_tech.5.html

Classes

Record	MOS Technology record.

4.6.1 hexrec.formats.mos.Record

class hexrec.formats.mos.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)
MOS Technology record.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.

- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.
- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_data	Builds a data record.
build_standalone	Makes a sequence of data records standalone.
build_terminator	Builds a terminator record.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.
REGEX	Regular expression for parsing a record text line.
TAG_TYPE	Associated Python class for tags.

EXTENSIONS: Sequence[str] = ('.mos',)

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

REGEX = re.compile('^;(?P<count>[0-9A-Fa-f]{2})(?P<address>[0-9A-Fa-f]{4})(?P<data>([0-9A-Fa-f]{2}){,255})(?P<checksum>[0-9A-Fa-f]{4})$')

Regular expression for parsing a record text line.

TAG_TYPE: Optional[Type[*Tag*]] = None

Associated Python class for tags.

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns

str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__

list of weak references to the object (if defined)

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters

path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_data(*address*, *data*)

Builds a data record.

Parameters

- **address** (*int*) – Record address.
- **data** (*bytes*) – Record data.

Returns
record – Data record.

Example

```
>>> Record.build_data(0x1234, b'Hello, World!')
...
Record(address=0x1234, tag=None, count=13,
       data=b'Hello, World!', checksum=0x04AA)
```

classmethod build_standalone(*data_records*, **args*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters
data_records (*list of records*) – A sequence of data records.

Yields
record – Records for a standalone record file.

classmethod build_terminator(*record_count*)

Builds a terminator record.

The terminator record holds the number of data records in the *address* fields. Also the *checksum* field is actually set to the record count.

Parameters
record_count (*int*) – Number of previous records.

Returns
record – A terminator record.

Example

```
>>> Record.build_data(0x1234, b'Hello, World!')
...
Record(address=0x00001234, tag=0, count=13,
       data=b'Hello, World!', checksum=0x69)
```

check()

Performs consistency checks.

Raises
ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters
records (*list of records*) – Sequence of records.

Raises
ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns
int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns

bool – *count* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters

records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is None.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

`is_data()`

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

`classmethod load_blocks(path)`

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
Memory – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

bytes or str – Data for output, according to the file type.

overlaps(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters

other (*record*) – Record to compare with *self*.

Returns

bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
```

(continues on next page)

(continued from previous page)

```
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

classmethod parse_record(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Parsed record.

Note: This method must be overridden.

---

classmethod read_blocks(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters
records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (*Memory*) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, *address=0*, *columns=16*, *align=Ellipsis*, *standalone=True*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. Maximum of 128 columns.
- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.

Yields

record – Data split into records.

Raises

ValueError – Address, size, or column overflow.

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (Memory) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.

- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

4.7 hexrec.formats.motorola

Motorola S-record format.

See also:

https://en.wikipedia.org/wiki/SREC_(file_format)

Classes

Record	Motorola S-record.
Tag	Motorola S-record tag.

4.7.1 hexrec.formats.motorola.Record

class hexrec.formats.motorola.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)

Motorola S-record.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.
- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_count	Builds a count record.
build_data	Builds a data record.
build_header	Builds a header record.
build_standalone	Makes a sequence of data records standalone.
build_terminator	Builds a terminator record.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fit_count_tag	Fits the record count tag.
fit_data_tag	Fits a data tag by address.
fix_tags	Fix record tags.
get_header	Gets the header record.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.

continues on next page

Table 3 – continued from previous page

load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
set_header	Sets the header data.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.
MATCHING_TAG	Maps the terminator tag to its matching data tag.
REGEX	Regular expression for parsing a record text line.
TAG_TO_ADDRESS_LENGTH	Maps a tag to its address byte length, if available.
TAG_TO_COLUMN_SIZE	Maps a tag to its maximum column size, if available.

EXTENSIONS: Sequence[str] = ('.mot', '.s19', '.s28', '.s37', '.srec', '.exo')

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

MATCHING_TAG: Sequence[Optional[int]] = (None, None, None, None, None, None, None, 3, 2, 1)

Maps the terminator tag to its matching data tag.

REGEX = re.compile('^S[0-9]([0-9A-Fa-f]{2}){4,264}$')

Regular expression for parsing a record text line.

TAG_TO_ADDRESS_LENGTH: Sequence[Optional[int]] = (2, 2, 3, 4, None, None, None, 4, 3, 2)

Maps a tag to its address byte length, if available.

TAG_TO_COLUMN_SIZE: Sequence[Optional[int]] = (None, 252, 251, 250, None, None, None, None, None, None)

Maps a tag to its maximum column size, if available.

TAG_TYPE

Associated Python class for tags.

alias of *Tag*

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

> **Warning:** Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns
str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__
list of weak references to the object (if defined)

_get_checksum()
int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)
Opens a file for input.

Parameters
path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)
Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_count(*record_count*)
Builds a count record.

Parameters
record_count (*int*) – Record count.

Returns
record – Count record.

Raises
ValueError – Count error.

Examples

```
>>> str(Record.build_count(0x1234))
'S5031234B6'
```

```
>>> str(Record.build_count(0x123456))
'S6041234565F'
```

classmethod build_data(*address*, *data*, *tag=None*)

Builds a data record.

Parameters

- **address** (*int*) – Record start address.
- **data** (*bytes*) – Some program data.
- **tag** (*tag*) – Data tag record. If None, automatically selects the fitting one.

Returns

record – Data record.

Raises

ValueError – Tag error.

Examples

```
>>> str(Record.build_data(0x1234, b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> str(Record.build_data(0x1234, b'Hello, World!',
...                       tag=Tag.DATA_16))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> str(Record.build_data(0x123456, b'Hello, World!',
...                       tag=Tag.DATA_24))
'S21112345648656C6C6F2C20576F726C6421E9'
```

```
>>> str(Record.build_data(0x12345678, b'Hello, World!',
...                       tag=Tag.DATA_32))
'S3121234567848656C6C6F2C20576F726C642170'
```

classmethod build_header(*data*)

Builds a header record.

Parameters

data (*bytes*) – Header string data.

Returns

record – Header record.

Example

```
>>> str(Record.build_header(b'Hello, World!'))
'S010000048656C6C6F2C20576F726C642186'
```

classmethod build_standalone(*data_records*, *start=None*, *tag=None*, *header=b''*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of records*) – A sequence of data records.
- **start** (*int*) – Program start address. If `None`, it is assigned the minimum data record address.
- **tag** (*tag*) – Data tag record. If `None`, automatically selects the fitting one.
- **header** (*bytes*) – Header byte data.

Yields

record – Records for a standalone record file.

classmethod build_terminator(*start*, *last_data_tag=Tag.DATA_16*)

Builds a terminator record.

Parameters

- **start** (*int*) – Program start address.
- **last_data_tag** (*tag*) – Last data record tag to match.

Returns

record – Terminator record.

Examples

```
>>> str(Record.build_terminator(0x1234))
'S9031234B6'
```

```
>>> str(Record.build_terminator(0x1234, Tag.DATA_16))
'S9031234B6'
```

```
>>> str(Record.build_terminator(0x123456, Tag.DATA_24))
'S8041234565F'
```

```
>>> str(Record.build_terminator(0x12345678, Tag.DATA_32))
'S70512345678E6'
```

check()

Performs consistency checks.

Raises

ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters
records (*list of records*) – Sequence of records.

Raises
ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns
int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns
bool – *count* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fit_count_tag(*record_count*)

Fits the record count tag.

Parameters
record_count (*int*) – Record count.

Returns
tag – Fitting record count tag.

Raises
ValueError – Count overflow.

Examples

```
>>> Record.fit_count_tag(0x0000000)
<Tag.COUNT_16: 5>
```

```
>>> Record.fit_count_tag(0x00FFFF)
<Tag.COUNT_16: 5>
```

```
>>> Record.fit_count_tag(0x010000)
<Tag.COUNT_24: 6>
```

```
>>> Record.fit_count_tag(0xFFFFFF)
<Tag.COUNT_24: 6>
```

classmethod fit_data_tag(*endex*)

Fits a data tag by address.

Depending on the value of *endex*, get the data tag with the smallest supported address.

Parameters
endex (*int*) – Exclusive end address of the data.

Returns
tag – Fitting data tag.

Raises
ValueError – Address overflow.

Examples

```
>>> Record.fit_data_tag(0x00000000)
<Tag.DATA_16: 1>
```

```
>>> Record.fit_data_tag(0x0000FFFF)
<Tag.DATA_16: 1>
```

```
>>> Record.fit_data_tag(0x00010000)
<Tag.DATA_16: 1>
```

```
>>> Record.fit_data_tag(0x00FFFFFF)
<Tag.DATA_24: 2>
```

```
>>> Record.fit_data_tag(0x01000000)
<Tag.DATA_24: 2>
```

```
>>> Record.fit_data_tag(0xFFFFFFFF)
<Tag.DATA_32: 3>
```

```
>>> Record.fit_data_tag(0x100000000)
<Tag.DATA_32: 3>
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters
records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_header(*records*)

Gets the header record.

Parameters
records (*list of records*) – A sequence of records.

Returns
record – The header record, or `None`.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Collected metadata:

- *columns*: maximum data columns per line found, or `None`.
- *start*: program execution start address found, or `None`.
- *count*: last *count* record found, or `None`.
- *header*: last *header* record data found, or `None`.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

is_data()

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

classmethod load_blocks(*path*)

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
`Memory` – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

bytes or str – Data for output, according to the file type.

overlaps(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters

other (*record*) – Record to compare with *self*.

Returns

bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
```

(continues on next page)

(continued from previous page)

```
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

classmethod parse_record(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Parsed record.

Note: This method must be overridden.

---

classmethod read_blocks(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters
records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (*Memory*) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod set_header(*records*, *data*)

Sets the header data.

If existing, the header record is updated in-place. If missing, the header record is prepended.

Parameters

- **records** (*list of records*) – A sequence of records.
- **data** (*bytes*) – Optional header data.

Returns

list of records – Updated record list.

classmethod split(*data*, *address=0*, *columns=16*, *align=Ellipsis*, *standalone=True*, *start=0*, *tag=None*, *header=b''*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. Maximum columns: 252 for *S1*, 251 for *S2*, 250 for *S3*.

- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.
- **start** (*int*) – Program start address. If `Ellipsis`, it is assigned the minimum data record address. If `None`, no start address records are output.
- **tag** (*tag*) – Data tag record. If `None`, automatically selects the fitting one.
- **header** (*bytes*) – Header byte data.

Yields
record – Data split into records.

Raises
ValueError – Address, size, or column overflow.

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (Memory) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.7.2 hexrec.formats.motorola.Tag

class hexrec.formats.motorola.**Tag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Motorola S-record tag.

Methods

__init__	
as_integer_ratio	Return integer ratio.
bit_count	Number of ones in the binary representation of the absolute value of self.
bit_length	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes	Return the integer represented by the given array of bytes.
to_bytes	Return an array of bytes representing an integer.

Attributes

HEADER	Header string.
DATA_16	16-bit address data record.
DATA_24	24-bit address data record.
DATA_32	32-bit address data record.
COUNT_16	16-bit record count.
COUNT_24	24-bit record count.
START_32	32-bit start address.
START_24	24-bit start address.
START_16	16-bit start address.
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

COUNT_16 = 5

16-bit record count. Optional.

COUNT_24 = 6

24-bit record count. Optional.

DATA_16 = 1

16-bit address data record.

DATA_24 = 2

24-bit address data record.

DATA_32 = 3

32-bit address data record.

HEADER = 0

Header string. Optional.

START_16 = 9

16-bit start address. Terminates *DATA_16*.

START_24 = 8

24-bit start address. Terminates *DATA_24*.

START_32 = 7

32-bit start address. Terminates *DATA_32*.

_RESERVED = 4

Reserved tag.

__abs__()

abs(self)

__add__(*value*, /)

Return self+value.

__and__(*value*, /)

Return self&value.

__bool__()

True if self else False

__ceil__()

Ceiling of an Integral returns itself.

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

__dir__()

Returns all members and all public methods

__divmod__(*value*, /)

Return divmod(self, value).

__eq__(*value*, /)

Return self==value.

__float__()
float(self)

__floor__()
Flooring an Integral returns itself.

__floordiv__(*value*, /)
Return self//value.

__format__(*format_spec*, /)
Default object formatter.

__ge__(*value*, /)
Return self>=value.

__getattribute__(*name*, /)
Return getattr(self, name).

classmethod __getitem__(*name*)
Return the member matching *name*.

__gt__(*value*, /)
Return self>value.

__hash__()
Return hash(self).

__index__()
Return self converted to an integer, if self is suitable for use as an index into a list.

__init__(**args*, ***kwds*)

__int__()
int(self)

__invert__()
~self

classmethod __iter__()
Return members in definition order.

__le__(*value*, /)
Return self<=value.

classmethod __len__()
Return the number of members (no aliases)

__lshift__(*value*, /)
Return self<<value.

__lt__(*value*, /)
Return self<value.

__mod__(*value*, /)
Return self%value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__neg__()
-self

__new__(*value*)

__or__(*value*, /)
Return self|value.

__pos__()
+self

__pow__(*value*, *mod=None*, /)
Return pow(self, value, mod).

__radd__(*value*, /)
Return value+self.

__rand__(*value*, /)
Return value&self.

__rdivmod__(*value*, /)
Return divmod(value, self).

__reduce_ex__(*proto*)
Helper for pickle.

__repr__()
Return repr(self).

__rfloordiv__(*value*, /)
Return value//self.

__rlshift__(*value*, /)
Return value<<self.

__rmod__(*value*, /)
Return value%self.

__rmul__(*value*, /)
Return value*self.

__ror__(*value*, /)
Return value|self.

__round__()
Rounding an Integral returns itself.

Rounding with an ndigits argument also returns an integer.

__rpow__(*value*, *mod=None*, /)
Return pow(value, self, mod).

__rrshift__(*value*, /)
Return value>>self.

__rshift__(*value*, /)

Return self>>value.

__rsub__(*value*, /)

Return value-self.

__rtruediv__(*value*, /)

Return value/self.

__rxor__(*value*, /)

Return value^self.

__sizeof__()

Returns size in memory, in bytes.

__str__()

Return repr(self).

__sub__(*value*, /)

Return self-value.

__truediv__(*value*, /)

Return self/value.

__trunc__()

Truncating an Integral returns itself.

__xor__(*value*, /)

Return self^value.

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()

Returns self, the complex conjugate of any int.

denominator

the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

bytes
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Indicates whether two's complement is used to represent the integer.

imag

the imaginary part of a complex number

numerator

the numerator of a rational number in lowest terms

real

the real part of a complex number

to_bytes(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

4.8 hexrec.formats.tektronix

Tektronix extended HEX format.

See also:

https://en.wikipedia.org/wiki/Tektronix_extended_HEX

Classes

Record	Tektronix extended HEX record.
Tag	

4.8.1 hexrec.formats.tektronix.Record

class hexrec.formats.tektronix.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)

Tektronix extended HEX record.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.
- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Methods

__init__	
build_data	Builds a data record.
build_standalone	Makes a sequence of data records standalone.
build_terminator	Builds a terminator record.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.

continues on next page

Table 4 – continued from previous page

fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.
REGEX	Regular expression for parsing a record text line.

EXTENSIONS: Sequence[str] = ('.tek',)

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

REGEX =
re.compile('^%(?P<count>[0-9A-Fa-f]{2})(?P<tag>[68])(?P<checksum>[0-9A-Fa-f]{2})8(?
P<address>[0-9A-Fa-f]{8})(?P<data>([0-9A-Fa-f]{2}){,255})$')

Regular expression for parsing a record text line.

TAG_TYPE

Associated Python class for tags.

alias of *Tag*

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns

str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

__weakref__

list of weak references to the object (if defined)

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters
path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_data(*address*, *data*)

Builds a data record.

Parameters

- **address** (*int*) – Record address.
- **data** (*bytes*) – Record data.

Returns
record – Data record.

Example

```
>>> str(Record.build_data(0x12345678, b'Hello, World!'))
'%236E081234567848656C6C6F2C20576F726C6421'
```

classmethod build_standalone(*data_records*, **args*, *start=None*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of record*) – A sequence of data records.
- **start** (*int*) – Program start address. If None, it is assigned the minimum data record address.

Yields

record – Records for a standalone record file.

classmethod build_terminator(*start*)

Builds a terminator record.

Parameters

start (*int*) – Program start address.

Returns

record – Terminator record.

Example

```
>>> str(Record.build_terminator(0x12345678))
'%0983D812345678'
```

check()

Performs consistency checks.

Raises

ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters

records (*list of records*) – Sequence of records.

Raises

ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns

int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns

bool – count field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters
records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is `None`.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

is_data()

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

classmethod load_blocks(*path*)

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
`Memory` – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
bytes or str – Data for output, according to the file type.

`overlaps`(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters
other (*record*) – Record to compare with *self*.

Returns
bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

`classmethod parse_record`(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
record – Parsed record.

Note: This method must be overridden.

`classmethod read_blocks`(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters
stream (*stream*) – Input stream of the blocks to read.

Returns
Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters

records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (Memory) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, *address=0*, *columns=16*, *align=Ellipsis*, *standalone=True*, *start=None*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **address** (*int*) – Start address of the first data record being split.
- **columns** (*int*) – Maximum number of columns per data record. Maximum of 128 columns.
- **align** (*int*) – Aligns record addresses to such number. If `Ellipsis`, its value is resolved after *columns*.
- **standalone** (*bool*) – Generates a sequence of records that can be saved as a standalone record file.
- **start** (*int*) – Program start address. If `Ellipsis`, it is assigned the minimum data record address. If `None`, no start address is set.

Yields

record – Data split into records.

Raises

ValueError – Address, size, or column overflow.

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via *build_data()* and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (Memory) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.8.2 hexrec.formats.tektronix.Tag

class hexrec.formats.tektronix.**Tag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Methods

__init__	
as_integer_ratio	Return integer ratio.
bit_count	Number of ones in the binary representation of the absolute value of self.
bit_length	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes	Return the integer represented by the given array of bytes.
to_bytes	Return an array of bytes representing an integer.

Attributes

DATA	
TERMINATOR	
denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

__abs__()

abs(self)

__add__(*value*, /)

Return self+value.

__and__(*value*, /)

Return self&value.

__bool__()

True if self else False

__ceil__()

Ceiling of an Integral returns itself.

classmethod __contains__(*member*)

Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

__dir__()

Returns all members and all public methods

__divmod__(*value*, /)

Return divmod(self, value).

__eq__(*value*, /)

Return self==value.

__float__()

float(self)

__floor__()

Flooring an Integral returns itself.

__floordiv__(*value*, /)

Return self//value.

__format__(*format_spec*, /)

Default object formatter.

__ge__(*value*, /)

Return self>=value.

__getattribute__(*name*, /)
Return getattr(self, name).

classmethod __getitem__(*name*)
Return the member matching *name*.

__gt__(*value*, /)
Return self>value.

__hash__()
Return hash(self).

__index__()
Return self converted to an integer, if self is suitable for use as an index into a list.

__init__(**args*, ***kwds*)

__int__()
int(self)

__invert__()
~self

classmethod __iter__()
Return members in definition order.

__le__(*value*, /)
Return self<=value.

classmethod __len__()
Return the number of members (no aliases)

__lshift__(*value*, /)
Return self<<value.

__lt__(*value*, /)
Return self<value.

__mod__(*value*, /)
Return self%value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__neg__()
-self

__new__(*value*)

__or__(*value*, /)
Return self|value.

__pos__()
+self

__pow__(*value*, *mod=None*, /)
Return pow(self, value, mod).

__radd__(*value*, /)
Return value+self.

__rand__(*value*, /)
Return value&self.

__rdivmod__(*value*, /)
Return divmod(value, self).

__reduce_ex__(*proto*)
Helper for pickle.

__repr__()
Return repr(self).

__rfloordiv__(*value*, /)
Return value//self.

__rlshift__(*value*, /)
Return value<<self.

__rmod__(*value*, /)
Return value%self.

__rmul__(*value*, /)
Return value*self.

__ror__(*value*, /)
Return value|self.

__round__()
Rounding an Integral returns itself.

Rounding with an ndigits argument also returns an integer.

__rpow__(*value*, *mod=None*, /)
Return pow(value, self, mod).

__rrshift__(*value*, /)
Return value>>self.

__rshift__(*value*, /)
Return self>>value.

__rsub__(*value*, /)
Return value-self.

__rtruediv__(*value*, /)
Return value/self.

__rxor__(*value*, /)
Return value^self.

__sizeof__()
Returns size in memory, in bytes.

__str__()
Return repr(self).

__sub__(*value*, /)
Return self-value.

__truediv__(*value*, /)
Return self/value.

__trunc__()
Truncating an Integral returns itself.

__xor__(*value*, /)
Return self^value.

as_integer_ratio()
Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()
Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()
Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

conjugate()
Returns self, the complex conjugate of any int.

denominator
the denominator of a rational number in lowest terms

from_bytes(*byteorder='big'*, *, *signed=False*)
Return the integer represented by the given array of bytes.

bytes
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Indicates whether two's complement is used to represent the integer.

`imag`
the imaginary part of a complex number

`numerator`
the numerator of a rational number in lowest terms

`real`
the real part of a complex number

`to_bytes`(*length=1*, *byteorder='big'*, ***, *signed=False*)
Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

4.9 hexrec.records

Hexadecimal record management.

The core of this library are *hexadecimal record files*. Such files are used to store binary data in text form, where each byte octet is represented in hexadecimal format. Over the whole byte addressing range of the memory to store (typically 32-bit addressing), only the relevant data is kept.

The hexadecimal data text is split into *record lines*, which give the name to this family of file formats. Each line should at least be marked with the *address* of its first byte, so that it is possible to load data from sparse records.

Usually not only plain data records exist, but also records holding metadata, such as: a *terminator* record, the *record count*, the *start address* to set the program counter upon loading an executable, a generic *header string*, and so on. Each record line is thus marked with a *tag* to indicate which kind of data it holds.

Record lines are commonly protected by a *checksum*, so that each line can be checked for (arguably weak) consistency. A *count* number is used to measure the record line length some way.

Summarizing, a record line holds the following fields:

- a *tag* to tell which kind of (meta)data is hold;
- some bytes of actual *data*, or tag-specific;

- the *address* of its first data byte, or tag-specific;
- the *count* of record line characters;
- a *checksum* to protect the record line.

This module provides functions and classes to handle hexadecimal record files, from the record line itself, to high-level procedures.

Module attributes

RECORD_TYPES	Registered record types.

4.9.1 hexrec.records.RECORD_TYPES

hexrec.records.**RECORD_TYPES: MutableMapping[str, Type[*Record*]] = {}**

Registered record types.

Functions

blocks_to_records	Converts blocks to records.
convert_file	Converts a record file to another record type.
convert_records	Converts records to another type.
find_record_type	Finds the record type class.
find_record_type_name	Finds the record type name.
get_data_records	Extracts data records.
load_blocks	Loads blocks from a record file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a record file.
merge_files	Merges record files.
merge_records	Merges data records.
records_to_blocks	Converts records to blocks.
register_default_record_types	
save_blocks	Saves blocks to a record file.
save_chunk	Saves a data chunk to a record file.
save_memory	Saves a virtual memory to a record file.
save_records	Saves records to a record file.

4.9.2 hexrec.records.blocks_to_records

hexrec.records.**blocks_to_records**(*blocks*, *record_type*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Converts blocks to records.

Parameters

- **blocks** (*list of blocks*) – A sequence of non-contiguous blocks, sorted by start address.
- **record_type** (*type*) – Output record type.

- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Returns
list of records – Records holding data from *blocks*.

Example

```
>>> from hexrec.utils import chop_blocks
>>> from bytesparse import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> data = bytes(range(256))
>>> blocks = list(chop_blocks(data, 16))
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> records_to_blocks(records) == Memory.collapse_blocks(blocks)
True
```

4.9.3 hexrec.records.convert_file

hexrec.records.**convert_file**(*input_file*, *output_file*, *input_type=None*, *output_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Converts a record file to another record type.

> **Warning:** Only binary data is kept; metadata will be overwritten by the call to *Record.build_standalone()*.

Parameters

- **input_file** (*str*) – Path of the input file.
- **output_file** (*str*) – Path of the output file.
- **input_type** (*type*) – Explicit input record type. If `None`, it is guessed from the file extension.
- **output_type** (*type*) – Explicit output record type. If `None`, it is guessed from the file extension.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> motorola = list(MotorolaRecord.split(bytes(range(256))))
>>> intel = list(IntelRecord.split(bytes(range(256))))
>>> save_records('bytes.mot', motorola)
>>> convert_file('bytes.mot', 'bytes.hex')
>>> load_records('bytes.hex') == intel
True
```

4.9.4 hexrec.records.convert_records

hexrec.records.**convert_records**(*records*, *input_type=None*, *output_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Converts records to another type.

Parameters

- **records** (*list of records*) – A sequence of *Record* elements. Sequence generators supported if *input_type* is specified.
- **input_type** (*type*) – Explicit type of *records* elements. If None, it is taken from the first element of the (indexable) *records* sequence.
- **output_type** (*type*) – Explicit output type. If None, it is reassigned as *input_type*.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Returns

list of records – Converted records.

Examples

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> motorola = list(MotorolaRecord.split(bytes(range(256))))
>>> intel = list(IntelRecord.split(bytes(range(256))))
>>> converted = convert_records(motorola, output_type=IntelRecord)
>>> converted == intel
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> motorola = list(MotorolaRecord.split(bytes(range(256))))
>>> intel = list(IntelRecord.split(bytes(range(256))))
>>> converted = convert_records(intel, output_type=MotorolaRecord)
>>> converted == motorola
True
```

4.9.5 hexrec.records.find_record_type

hexrec.records.**find_record_type**(*file_path*)

Finds the record type class.

Checks if the extension of *file_path* is a know record type, and returns its mapped type class.

Parameters
file_path (*str*) – File path to get the file extension from.

Returns
str – Record type class.

Raises
KeyError – Unsupported extension.

Example

```
>>> from hexrec.records import find_record_type_name
>>> find_record_type('dummy.mot').__name__
'MotorolaRecord'
```

4.9.6 hexrec.records.find_record_type_name

hexrec.records.**find_record_type_name**(*file_path*)

Finds the record type name.

Checks if the extension of *file_path* is a know record type, and returns its mapped name.

Parameters
file_path (*str*) – File path to get the file extension from.

Returns
str – Record type name.

Raises
KeyError – Unsupported extension.

Example

```
>>> from hexrec.records import find_record_type_name
>>> find_record_type_name('dummy.mot')
'motorola'
```

4.9.7 hexrec.records.get_data_records

hexrec.records.**get_data_records**(*records*)

Extracts data records.

Parameters
records (*list of records*) – Sequence of records.

Returns
list of records – Sequence of data records.

Example

```
>>> from hexrec.utils import chop_blocks
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> data = bytes(range(256))
>>> blocks = list(chop_blocks(data, 16))
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> all(r.is_data() for r in get_data_records(records))
True
```

4.9.8 hexrec.records.load_blocks

hexrec.records.**load_blocks**(*path*, *record_type=None*)

Loads blocks from a record file.

Parameters

- **path** (*str*) – Path of the input file.
- **record_type** (*type*) – Explicit record type. If None, it is guessed from the file extension.

Returns
list of blocks – Blocks loaded from *path*.

Example

```
>>> blocks = [[n, bytes(range(n, n + 16))] for n in range(0, 256, 16)]
>>> save_blocks('bytes.mot', blocks)
>>> load_blocks('bytes.mot') == blocks
True
```

4.9.9 hexrec.records.load_memory

hexrec.records.**load_memory**(*path*, *record_type=None*)

Loads a virtual memory from a file.

Parameters

- **path** (*str*) – Path of the input file.
- **record_type** (*type*) – Explicit record type. If None, it is guessed from the file extension.

Returns
Memory – Virtual memory holding data from *path*.

Example

```
>>> blocks = [[n, bytes(range(n, n + 16))] for n in range(0, 256, 16)]
>>> blocks = Memory.collapse_blocks(blocks)
>>> memory = Memory.from_blocks(blocks)
>>> save_memory('bytes.mot', memory)
>>> load_memory('bytes.mot') == memory
True
```

4.9.10 hexrec.records.load_records

hexrec.records.**load_records**(*path*, *record_type=None*)

Loads records from a record file.

Parameters

- **path** (*str*) – Path of the input file.
- **record_type** (*type*) – Explicit record type. If None, it is guessed from the file extension.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> records = list(MotorolaRecord.split(bytes(range(256))))
>>> save_records('bytes.mot', records)
>>> load_records('bytes.mot') == records
True
```

4.9.11 hexrec.records.merge_files

hexrec.records.**merge_files**(*input_files*, *output_file*, *input_types=None*, *output_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Merges record files.

Merges multiple record files where each file overwrites overlapping data of the previous files.

> **Warning:** Only binary data is kept; metadata will be overwritten by the call to *Record.build_standalone()*.

Parameters

- **input_files** (*list of str*) – A sequence of file paths to merge.
- **output_file** (*str*) – Path of the output file. It can target an input file.
- **input_types** (*list of types*) – Selects the record type for each of the sequences in *data_records*. None will guess from file extension.

- **output_type** (*type*) – Selects the output record type. None will guess from file extension.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> merge_files(['merge1.mot', 'merge2.hex'], 'merged.tek')
```

4.9.12 hexrec.records.merge_records

hexrec.records.**merge_records**(*records*, *input_types=None*, *output_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Merges data records.

Merges multiple sequences of data records where each sequence overwrites overlapping data of the previous sequences.

Parameters

- **records** (*list of records*) – A vector of record sequences. If *input_types* is not None, sequence generators are supported for the vector and its nested sequences. Only *data* records are kept.
- **input_types** (*list of types*) – Selects the record type for each of the sequences in *data_records*. None will choose that of the first element of the (indexable) sequence.
- **output_type** (*type*) – Selects the output record type. None will choose that of the first *input_types*.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Returns

list of records – Merged records.

Example

```
>>> from hexrec.utils import chop_blocks
>>> from bytesparse import Memory
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> data1 = bytes(range(0, 32))
>>> data2 = bytes(range(96, 128))
>>> blocks1 = list(chop_blocks(data1, 16, start=0))
>>> blocks2 = list(chop_blocks(data2, 16, start=96))
```

(continues on next page)

(continued from previous page)

```
>>> records1 = blocks_to_records(blocks1, MotorolaRecord)
>>> records2 = blocks_to_records(blocks2, IntelRecord)
>>> IntelRecord.readdress(records2)
>>> merged_records = merge_records([records1, records2])
>>> merged_blocks = records_to_blocks(merged_records)
>>> merged_blocks == Memory.collapse_blocks(blocks1 + blocks2)
True
```

4.9.13 hexrec.records.records_to_blocks

hexrec.records.**records_to_blocks**(*records*)

Converts records to blocks.

Extracts all the data records, collapses them in the order they compare in *records*, and merges the collapsed blocks. Returns sequence of non-contiguous blocks, sorted by start address.

Parameters
records (*list of records*) – Sequence of records to convert to blocks.

Returns
list of blocks – Blocks holding data from *records*.

Example

```
>>> from hexrec.utils import chop_blocks
>>> from bytesparse import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> data = bytes(range(256))
>>> blocks = list(chop_blocks(data, 16))
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> records_to_blocks(records) == Memory.collapse_blocks(blocks)
True
```

4.9.14 hexrec.records.register_default_record_types

hexrec.records.**register_default_record_types**()

4.9.15 hexrec.records.save_blocks

hexrec.records.**save_blocks**(*path*, *blocks*, *record_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a record file.

Parameters

- **path** (*str*) – Path of the output file.
- **blocks** (*list of blocks*) – Sequence of blocks to save.
- **record_type** (*type*) – Explicit record type. If None, it is guessed from the file extension.

- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> blocks = [[n, bytes(range(n, n + 16))] for n in range(0, 256, 16)]
>>> save_blocks('bytes.hex', blocks)
>>> load_blocks('bytes.hex') == blocks
True
```

4.9.16 hexrec.records.save_chunk

hexrec.records.**save_chunk**(*path*, *chunk*, *address=0*, *record_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a data chunk to a record file.

Parameters

- **path** (*str*) – Path of the output file.
- **chunk** (*bytes*) – A chunk of data.
- **address** (*int*) – Address of the data chunk.
- **record_type** (*type*) – Explicit record type. If `None`, it is guessed from the file extension.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> data = bytes(range(256))
>>> save_chunk('bytes.mot', data, 0x12345678)
>>> load_blocks('bytes.mot') == [[0x12345678, data]]
True
```

4.9.17 hexrec.records.save_memory

hexrec.records.**save_memory**(*path*, *memory*, *record_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a record file.

Parameters

- **path** (*str*) – Path of the output file.
- **memory** (Memory) – A virtual memory.
- **record_type** (*type*) – Explicit record type. If None, it is guessed from the file extension.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> blocks = [[n, bytes(range(n, n + 16))] for n in range(0, 256, 16)]
>>> blocks = Memory.collapse_blocks(blocks)
>>> memory = Memory.from_blocks(blocks)
>>> save_memory('bytes.hex', memory)
>>> load_memory('bytes.hex') == memory
True
```

4.9.18 hexrec.records.save_records

hexrec.records.**save_records**(*path*, *records*, *output_type=None*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves records to a record file.

Parameters

- **path** (*str*) – Path of the output file.
- **records** (*list of records*) – Sequence of records to save.
- **output_type** (*type*) – Output record type. If None, it is guessed from the file extension.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> records = list(IntelRecord.split(bytes(range(256))))
>>> save_records('bytes.hex', records)
>>> load_records('bytes.hex') == records
True
```

Classes

Record	Abstract record type.
Tag	Abstract record tag.

4.9.19 hexrec.records.Record

class hexrec.records.**Record**(*address*, *tag*, *data*, *checksum=Ellipsis*)

Abstract record type.

A record is the basic structure of a record file.

This is an abstract class, so it provides basic generic methods shared by most of the *Record* implementations. Please refer to the actual subclass for more details.

Variables

- **address** (*int*) – Tells where its *data* starts in the memory addressing space, or an address with a special meaning.
- **tag** (*int*) – Defines the logical meaning of the *address* and *data* fields.
- **data** (*bytes*) – Byte data as required by the *tag*.
- **count** (*int*) – Counts its fields as required by the *Record* subclass implementation.
- **checksum** (*int*) – Computes the checksum as required by most *Record* implementations.

Parameters

- **address** (*int*) – Record *address* field.
- **tag** (*int*) – Record *tag* field.
- **data** (*bytes*) – Record *data* field.
- **checksum** (*int*) – Record *checksum* field. `Ellipsis` makes the constructor compute its actual value automatically. `None` assigns `None`.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0x1234, None, b'Hello, World!')
...
Record(address=0x00001234, tag=None, count=13,
       data=b'Hello, World!', checksum=0x69)
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0x1234, MotorolaTag.DATA_16, b'Hello, World!')
...
Record(address=0x00001234, tag=<MotorolaTag.DATA_16: 1>,
       count=16, data=b'Hello, World!', checksum=0x40)
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!')
...
Record(address=0x00001234, tag=<IntelTag.DATA: 0>, count=13,
       data=b'Hello, World!', checksum=0x44)
```

Methods

__init__	
build_standalone	Makes a sequence of data records standalone.
check	Performs consistency checks.
check_sequence	Consistency check of a sequence of records.
compute_checksum	Computes the checksum.
compute_count	Computes the count.
fix_tags	Fix record tags.
get_metadata	Retrieves metadata from records.
is_data	Tells if it is a data record.
load_blocks	Loads blocks from a file.
load_memory	Loads a virtual memory from a file.
load_records	Loads records from a file.
marshal	Marshals a record for output.
overlaps	Checks if overlapping occurs.
parse_record	Parses a record from a text line.
read_blocks	Reads blocks from a stream.
read_memory	Reads a virtual memory from a stream.
read_records	Reads records from a stream.
readdress	Converts to flat addressing.
save_blocks	Saves blocks to a file.
save_memory	Saves a virtual memory to a file.
save_records	Saves records to a file.
split	Splits a chunk of data into records.
unmarshal	Unmarshals a record from input.
update_checksum	Updates the *checksum* field via *compute_count()*.
update_count	Updates the *count* field via *compute_count()*.
write_blocks	Writes blocks to a stream.
write_memory	Writes a virtual memory to a stream.
write_records	Saves records to a stream.

Attributes

tag	
count	
address	
data	
checksum	
EXTENSIONS	File extensions typically mapped to this record type.
LINE_SEP	Separator between record lines.

EXTENSIONS: Sequence[str] = ()

File extensions typically mapped to this record type.

LINE_SEP: Union[bytes, str] = '\n'

Separator between record lines.

If subclass of `bytes`, it is considered as a binary file.

TAG_TYPE

Associated Python class for tags.

alias of *Tag*

__eq__(*other*)

Equality comparison.

Returns

bool – The *address*, *tag*, and *data* fields are equal.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record1 == record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(1, b'Hello, World!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord.build_data(0, b'Hello, World!')
>>> record2 = BinaryRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> record1 = MotorolaRecord.build_header(b'Hello, World!')
>>> record2 = MotorolaRecord.build_data(0, b'hello, world!')
>>> record1 == record2
False
```

__hash__()

Computes the hash value.

Computes the hash of the *Record* fields. Useful to make the record hashable although it is a mutable class.

Returns

int – Hash of the *Record* fields.

Warning: Be careful with hashable mutable objects!

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> hash(BinaryRecord(0x1234, None, b'Hello, World!'))
...
7668968047460943252
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> hash(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                             b'Hello, World!'))
...
7668968047460943265
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> hash(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
...
7668968047460943289
```

__init__(*address*, *tag*, *data*, *checksum=Ellipsis*)

__lt__(*other*)

Less-than comparison.

Returns

bool – *address* less than *other*'s.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x1234, None, b'')
>>> record2 = BinaryRecord(0x4321, None, b'')
>>> record1 < record2
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0x4321, None, b'')
>>> record2 = BinaryRecord(0x1234, None, b'')
>>> record1 < record2
False
```

__repr__()

Return repr(self).

__str__()

Converts to text string.

Builds a printable text representation of the record, usually the same found in the saved record file as per its *Record* subclass requirements.

Returns

str – A printable text representation of the record.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> str(BinaryRecord(0x1234, None, b'Hello, World!'))
'48656C6C6F2C20576F726C6421'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> str(MotorolaRecord(0x1234, MotorolaTag.DATA_16,
...                    b'Hello, World!'))
'S110123448656C6C6F2C20576F726C642140'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> str(IntelRecord(0x1234, IntelTag.DATA, b'Hello, World!'))
':0D12340048656C6C6F2C20576F726C642144'
```

_get_checksum()

int: The *checksum* field itself if not None, the value computed by *compute_count()* otherwise.

classmethod _open_input(*path*)

Opens a file for input.

Parameters
path (*str*) – File path.

Returns
stream – An input stream handle.

classmethod _open_output(*path*)

Opens a file for output.

Parameters
path (*str*) – File path.

Returns
stream – An output stream handle.

classmethod build_standalone(*data_records*, **args*, ***kwargs*)

Makes a sequence of data records standalone.

Parameters

- **data_records** (*list of records*) – Sequence of data records.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Yields
record – Records for a standalone record file.

check()

Performs consistency checks.

Raises
ValueError – a field is inconsistent.

classmethod check_sequence(*records*)

Consistency check of a sequence of records.

Parameters
records (*list of records*) – Sequence of records.

Raises
ValueError – A field is inconsistent.

compute_checksum()

Computes the checksum.

Returns
int – *checksum* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> hex(record.compute_checksum())
'0x69'
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> hex(record.compute_checksum())
'0x86'
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> hex(record.compute_checksum())
'0x8a'
```

compute_count()

Computes the count.

Returns
bool – *count* field value based on the current fields.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record = BinaryRecord(0, None, b'Hello, World!')
>>> str(record)
'48656C6C6F2C20576F726C6421'
>>> record.compute_count()
13
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> record = MotorolaRecord(0, MotorolaTag.DATA_16,
...                         b'Hello, World!')
>>> str(record)
'S110000048656C6C6F2C20576F726C642186'
>>> record.compute_count()
16
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> record = IntelRecord(0, IntelTag.DATA, b'Hello, World!')
>>> str(record)
':0D00000048656C6C6F2C20576F726C64218A'
>>> record.compute_count()
13
```

classmethod fix_tags(*records*)

Fix record tags.

Updates record tags to reflect modified size and count. All the checksums are updated too. Operates in-place.

Parameters
records (*list of records*) – A sequence of records. Must be in-line mutable.

classmethod get_metadata(*records*)

Retrieves metadata from records.

Metadata is specific of each record type. The most common metadata are:

- *columns*: maximum data columns per line.
- *start*: program execution start address.
- *count*: some count of record lines.
- *header*: some header data.

When no such information is found, its keyword is either skipped or its value is `None`.

Parameters
records (*list of records*) – Records to scan for metadata.

Returns
dict – Collected metadata.

is_data()

Tells if it is a data record.

Tells whether the record contains plain binary data, i.e. it is not a *special* record.

Returns
bool – The record contains plain binary data.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> BinaryRecord(0, None, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.DATA_16,
...                b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.formats.motorola import Tag as MotorolaTag
>>> MotorolaRecord(0, MotorolaTag.HEADER,
...                b'Hello, World!').is_data()
False
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.DATA, b'Hello, World!').is_data()
True
```

```
>>> from hexrec.formats.intel import Record as IntelRecord
>>> from hexrec.formats.intel import Tag as IntelTag
>>> IntelRecord(0, IntelTag.END_OF_FILE, b'').is_data()
False
```

classmethod load_blocks(*path*)

Loads blocks from a file.

Each line of the input file is parsed via `parse_block()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> MotorolaRecord.load_blocks('load_blocks.mot')
[[0, b'abc'], [16, b'def']]
```

classmethod load_memory(*path*)

Loads a virtual memory from a file.

Parameters
path (*str*) – Path of the record file to load.

Returns
Memory – Loaded virtual memory.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_blocks.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> memory = MotorolaRecord.load_memory('load_blocks.mot')
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod load_records(*path*)

Loads records from a file.

Each line of the input file is parsed via `parse()`, and collected into the returned sequence.

Parameters
path (*str*) – Path of the record file to load.

Returns
list of records – Sequence of parsed records.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> with open('load_records.mot', 'wt') as f:
...     f.write('S0030000FC\n')
...     f.write('S1060000616263D3\n')
...     f.write('S1060010646566BA\n')
...     f.write('S5030002FA\n')
...     f.write('S9030000FC\n')
>>> records = MotorolaRecord.load_records('load_records.mot')
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

marshal(**args*, ***kwargs*)

Marshals a record for output.

Parameters

- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

bytes or str – Data for output, according to the file type.

overlaps(*other*)

Checks if overlapping occurs.

This record and another have overlapping *data*, when both *address* fields are not `None`.

Parameters

other (*record*) – Record to compare with *self*.

Returns

bool – Overlapping.

Examples

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
>>> record2 = BinaryRecord(1, None, b'def')
>>> record1.overlaps(record2)
True
```

```
>>> from hexrec.formats.binary import Record as BinaryRecord
>>> record1 = BinaryRecord(0, None, b'abc')
```

(continues on next page)

(continued from previous page)

```
>>> record2 = BinaryRecord(3, None, b'def')
>>> record1.overlaps(record2)
False
```

classmethod parse_record(*line*, **args*, ***kwargs*)

Parses a record from a text line.

Parameters

- **line** (*str*) – Record line to parse.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

record – Parsed record.

Note: This method must be overridden.

---

classmethod read_blocks(*stream*)

Reads blocks from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

list of blocks – Sequence of parsed blocks.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> MotorolaRecord.read_blocks(stream)
[[0, b'abc'], [16, b'def']]
```

classmethod read_memory(*stream*)

Reads a virtual memory from a stream.

Read blocks from the input stream into the returned sequence.

Parameters

stream (*stream*) – Input stream of the blocks to read.

Returns

Memory – Loaded virtual memory.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> memory = MotorolaRecord.read_memory(stream)
>>> memory.to_blocks()
[[0, b'abc'], [16, b'def']]
```

classmethod read_records(*stream*)

Reads records from a stream.

For text files, each line of the input file is parsed via `parse()`, and collected into the returned sequence.

For binary files, everything to the end of the stream is parsed as a single record.

Parameters
stream (*stream*) – Input stream of the records to read.

Returns
list of records – Sequence of parsed records.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> _ = stream.seek(0, io.SEEK_SET)
>>> records = MotorolaRecord.read_records(stream)
>>> records
[Record(address=0x00000000, tag=<Tag.HEADER: 0>, count=3,
        data=b'', checksum=0xFC),
 Record(address=0x00000000, tag=<Tag.DATA_16: 1>, count=6,
        data=b'abc', checksum=0xD3),
 Record(address=0x00000010, tag=<Tag.DATA_16: 1>, count=6,
        data=b'def', checksum=0xBA),
 Record(address=0x00000000, tag=<Tag.COUNT_16: 5>, count=3,
        data=b'\x00\x02', checksum=0xFA),
 Record(address=0x00000000, tag=<Tag.START_16: 9>, count=3,
        data=b'', checksum=0xFC)]
```

classmethod readdress(*records*)

Converts to flat addressing.

Some record types, notably the *Intel HEX*, store records by some *segment/offset* addressing flavor. As this library adopts *flat* addressing instead, all the record addresses should be converted to *flat* addressing after loading. This procedure readdresses a sequence of records in-place.

> **Warning:** Only the *address* field is modified. All the other fields hold their previous value.

Parameters
records (*list*) – Sequence of records to be converted to *flat* addressing, in-place.

classmethod save_blocks(*path*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves blocks to a file.

Each block of the *blocks* sequence is converted into a record via `build_data()` and written to the output file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **blocks** (*list of blocks*) – Sequence of blocks to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> MotorolaRecord.save_blocks('save_blocks.mot', blocks)
>>> with open('save_blocks.mot', 'rt') as f: text = f.read()
>>> text
'S00300000FC\nS1060000616263D3\nS1060010646566BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_memory(*path*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Saves a virtual memory to a file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **memory** (`Memory`) – Virtual memory to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory(blocks=[[0, b'abc'], [16, b'def']])
>>> MotorolaRecord.save_memory('save_memory.mot', memory)
>>> with open('save_memory.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod save_records(*path*, *records*)

Saves records to a file.

Each record of the *records* sequence is converted into text via `str()`, and stored into the output text file.

Parameters

- **path** (*str*) – Path of the record file to save.
- **records** (*list*) – Sequence of records to store.

Example

```
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> MotorolaRecord.save_records('save_records.mot', records)
>>> with open('save_records.mot', 'rt') as f: text = f.read()
>>> text
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod split(*data*, **args*, ***kwargs*)

Splits a chunk of data into records.

Parameters

- **data** (*bytes*) – Byte data to split.
- **args** (*tuple*) – Further positional arguments for overriding.
- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns

list – List of records.

Note: This method must be overridden.

---

classmethod unmarshal(*data*, **args*, ***kwargs*)

Unmarshals a record from input.

Parameters

- **data** (*bytes or str*) – Input data, according to the file type.
- **args** (*tuple*) – Further positional arguments for overriding.

- **kwargs** (*dict*) – Further keyword arguments for overriding.

Returns
record – Unmarshaled record.

update_checksum()

Updates the *checksum* field via *compute_count()*.

update_count()

Updates the *count* field via *compute_count()*.

classmethod write_blocks(*stream*, *blocks*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes blocks to a stream.

Each block of the *blocks* sequence is converted into a record via `build_data()` and written to the output stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **blocks** (*list of blocks*) – Sequence of records to store.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> stream = io.StringIO()
>>> MotorolaRecord.write_blocks(stream, blocks)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_memory(*stream*, *memory*, *split_args=None*, *split_kwargs=None*, *build_args=None*, *build_kwargs=None*)

Writes a virtual memory to a stream.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **memory** (`Memory`) – Virtual memory to save.
- **split_args** (*list*) – Positional arguments for *Record.split()*.
- **split_kwargs** (*dict*) – Keyword arguments for *Record.split()*.
- **build_args** (*list*) – Positional arguments for *Record.build_standalone()*.
- **build_kwargs** (*dict*) – Keyword arguments for *Record.build_standalone()*.

Example

```
>>> import io
>>> from hexrec.records import Memory
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> memory = Memory.from_blocks([[0, b'abc'], [16, b'def']])
>>> stream = io.StringIO()
>>> MotorolaRecord.write_memory(stream, memory)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

classmethod write_records(*stream*, *records*)

Saves records to a stream.

Each record of the *records* sequence is stored into the output file.

Parameters

- **stream** (*stream*) – Output stream of the records to write.
- **records** (*list of records*) – Sequence of records to store.

Example

```
>>> import io
>>> from hexrec.formats.motorola import Record as MotorolaRecord
>>> from hexrec.records import blocks_to_records
>>> blocks = [[0, b'abc'], [16, b'def']]
>>> records = blocks_to_records(blocks, MotorolaRecord)
>>> stream = io.StringIO()
>>> MotorolaRecord.write_records(stream, records)
>>> stream.getvalue()
'S0030000FC\nS1060000616263D3\nS106001064656 6BA\nS5030002FA\nS9030000FC\n'
```

4.9.20 hexrec.records.Tag

class hexrec.records.**Tag**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Abstract record tag.

Methods

is_data	bool: *value* is a data record tag.
__init__	
as_integer_ratio	Return integer ratio.
bit_count	Number of ones in the binary representation of the absolute value of self.
bit_length	Number of bits necessary to represent self in binary.
conjugate	Returns self, the complex conjugate of any int.
from_bytes	Return the integer represented by the given array of bytes.
to_bytes	Return an array of bytes representing an integer.

Attributes

denominator	the denominator of a rational number in lowest terms
imag	the imaginary part of a complex number
numerator	the numerator of a rational number in lowest terms
real	the real part of a complex number

__abs__()
abs(self)

__add__(*value*, /)
Return self+value.

__and__(*value*, /)
Return self&value.

__bool__()
True if self else False

__ceil__()
Ceiling of an Integral returns itself.

classmethod __contains__(*member*)
Return True if member is a member of this enum raises TypeError if member is not an enum member

note: in 3.12 TypeError will no longer be raised, and True will also be returned if member is the value of a member in this enum

__dir__()
Returns all members and all public methods

__divmod__(*value*, /)
Return divmod(self, value).

__eq__(*value*, /)
Return self==value.

__float__()
float(self)

__floor__()
Flooring an Integral returns itself.

__floordiv__(*value*, /)
Return self//value.

__format__(*format_spec*, /)
Default object formatter.

__ge__(*value*, /)
Return self>=value.

__getattribute__(*name*, /)
Return getattr(self, name).

classmethod __getitem__(*name*)
Return the member matching *name*.

__gt__(*value*, /)
Return self>value.

__hash__()
Return hash(self).

__index__()
Return self converted to an integer, if self is suitable for use as an index into a list.

__init__(**args*, ***kwds*)

__int__()
int(self)

__invert__()
~self

classmethod __iter__()
Return members in definition order.

__le__(*value*, /)
Return self<=value.

classmethod __len__()
Return the number of members (no aliases)

__lshift__(*value*, /)
Return self<<value.

__lt__(*value*, /)
Return self<value.

__mod__(*value*, /)
Return self%value.

__mul__(*value*, /)
Return self*value.

__ne__(*value*, /)
Return self!=value.

__neg__()
-self

__new__(*value*)

__or__(*value*, /)
Return self|value.

__pos__()
+self

__pow__(*value*, *mod=None*, /)
Return pow(self, value, mod).

__radd__(*value*, /)
Return value+self.

__rand__(*value*, /)
Return value&self.

__rdivmod__(*value*, /)
Return divmod(value, self).

__reduce_ex__(*proto*)
Helper for pickle.

__repr__()
Return repr(self).

__rfloordiv__(*value*, /)
Return value//self.

__rlshift__(*value*, /)
Return value<<self.

__rmod__(*value*, /)
Return value%self.

__rmul__(*value*, /)
Return value*self.

__ror__(*value*, /)
Return value|self.

__round__()
Rounding an Integral returns itself.

Rounding with an ndigits argument also returns an integer.

__rpow__(*value*, *mod=None*, /)
Return pow(value, self, mod).

__rrshift__(*value*, /)
Return value>>self.

__rshift__(*value*, /)
Return self>>value.

__rsub__(*value*, /)

Return value-self.

__rtruediv__(*value*, /)

Return value/self.

__rxor__(*value*, /)

Return value^self.

__sizeof__()

Returns size in memory, in bytes.

__str__()

Return repr(self).

__sub__(*value*, /)

Return self-value.

__truediv__(*value*, /)

Return self/value.

__trunc__()

Truncating an Integral returns itself.

__xor__(*value*, /)

Return self^value.

as_integer_ratio()

Return integer ratio.

Return a pair of integers, whose ratio is exactly equal to the original int and with a positive denominator.

```
>>> (10).as_integer_ratio()
(10, 1)
>>> (-10).as_integer_ratio()
(-10, 1)
>>> (0).as_integer_ratio()
(0, 1)
```

bit_count()

Number of ones in the binary representation of the absolute value of self.

Also known as the population count.

```
>>> bin(13)
'0b1101'
>>> (13).bit_count()
3
```

bit_length()

Number of bits necessary to represent self in binary.

```
>>> bin(37)
'0b100101'
>>> (37).bit_length()
6
```

`conjugate()`

Returns self, the complex conjugate of any int.

`denominator`

the denominator of a rational number in lowest terms

`from_bytes`(*byteorder='big'*, ***, *signed=False*)

Return the integer represented by the given array of bytes.

bytes
Holds the array of bytes to convert. The argument must either support the buffer protocol or be an iterable object producing bytes. Bytes and bytearray are examples of built-in objects that support the buffer protocol.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Indicates whether two's complement is used to represent the integer.

`imag`

the imaginary part of a complex number

`classmethod is_data`(*value*)

bool: *value* is a data record tag.

`numerator`

the numerator of a rational number in lowest terms

`real`

the real part of a complex number

`to_bytes`(*length=1*, *byteorder='big'*, ***, *signed=False*)

Return an array of bytes representing an integer.

length
Length of bytes object to use. An OverflowError is raised if the integer is not representable with the given number of bytes. Default is length 1.

byteorder
The byte order used to represent the integer. If byteorder is 'big', the most significant byte is at the beginning of the byte array. If byteorder is 'little', the most significant byte is at the end of the byte array. To request the native byte order of the host system, use `sys.byteorder' as the byte order value. Default is to use 'big'.

signed
Determines whether two's complement is used to represent the integer. If signed is False and a negative integer is given, an OverflowError is raised.

4.10 hexrec.utils

Generic utility functions.

Functions

check_empty_args_kwargs	Checks for empty positional and keyword arguments.
chop	Chops a vector.
chop_blocks	Chops a sequence of items into blocks.
columnize	Splits and wraps a line into columns.
hexlify	Splits ans wraps byte data into hexadecimal columns.
parse_int	Parses an integer.
unhexlify	Converts a hexadecimal text line into bytes.

4.10.1 hexrec.utils.check_empty_args_kwargs

hexrec.utils.**check_empty_args_kwargs**(*args*, *kwargs*)

Checks for empty positional and keyword arguments.

Both *args* and *kwargs* must be either `None` or equivalent to `False`. If the check is not satisfied, a `ValueError` exception is raised.

Parameters

- **args** (*list*) – List of unexpected positional arguments, or `Null`.
- **kwargs** (*dict*) – List of unexpected keyword arguments, or `Null`.

Raises

ValueError – Condition not satisfied.

4.10.2 hexrec.utils.chop

hexrec.utils.**chop**(*vector*, *window*, *align_base=0*)

Chops a vector.

Iterates through the vector grouping its items into windows.

Parameters

- **vector** (*items*) – Vector to chop.
- **window** (*int*) – Window length.
- **align_base** (*int*) – Offset of the first window.

Yields

list or items – *vector* slices of up to *window* elements.

Examples

```
>>> list(chop(b'ABCDEFG', 2))
['AB', 'CD', 'EF', 'G']
```

```
>>> ':'.join(chop('ABCDEFG', 2))
'AB:CD:EF:G'
```

```
>>> list(chop('ABCDEFG', 4, 3))
['A', 'BCDE', 'FG']
```

4.10.3 hexrec.utils.chop_blocks

hexrec.utils.**chop_blocks**(*items*, *window*, *align_base=0*, *start=0*)

Chops a sequence of items into blocks.

Iterates through the vector grouping its items into windows.

Parameters

- **items** (*items*) – Sequence of items to chop.
- **window** (*int*) – Window length.
- **align_base** (*int*) – Offset of the first window.
- **start** (*int*) – Start address.

Yields

items – *items* slices of up to *window* elements.

Examples

9	10	11	12	13	14	15	16	17
	[A	B]	[C	D]	[E	F]	[G]	

```
>>> list(chop_blocks(b'ABCDEFG', 2, start=10))
[[10, b'AB'], [12, b'CD'], [14, b'EF'], [16, b'G']]
```

~~~

12	13	14	15	16	17	18	19	20
	[A]	[B	C	D	E]	[F	G]	

```
>>> list(chop_blocks(b'ABCDEFG', 4, 3, 10))
[[13, b'A'], [14, b'BCDE'], [18, b'FG']]
```
~~~

4.10.4 hexrec.utils.columnize

hexrec.utils.**columnize**(*line*, *width*, *sep=''*, *newline='\\n'*, *window=1*)

Splits and wraps a line into columns.

A text line is wrapped up to a width limit, separated by a given newline string. Each wrapped line is then split into columns by some window size, separated by a given separator string.

Parameters

- **line** (*str*) – Line of text to columnize.
- **width** (*int*) – Maximum line width.
- **sep** (*str*) – Column separator string.
- **newline** (*str*) – Line separator string.
- **window** (*int*) – Splitted column length.

Returns

str – A wrapped and columnized text.

Examples

```
>>> columnize('ABCDEFGHIJKLMNOPQRSTUVWXYZ', 6)
'ABCDEF\nGHIJKL\nMNOPQR\nSTUVWX\nYZ'
```

```
>>> columnize('ABCDEFGHIJKLMNOPQRSTUVWXYZ', 6, sep=' ', window=3)
'ABC DEF\nGHI JKL\nMNO PQR\nSTU VWX\nYZ'
```

4.10.5 hexrec.utils.hexlify

hexrec.utils.**hexlify**(*data*, *width=None*, *sep=''*, *newline='\\n'*, *window=2*, *upper=True*)

Splits ans wraps byte data into hexadecimal columns.

A chunk of byte data is converted into a hexadecimal text line, and then columnized as per *columnize()*.

Parameters

- **data** (*bytes*) – Byte data.
- **width** (*int*) – Maximum line width, or None.
- **sep** (*str*) – Column separator string.
- **newline** (*str*) – Line separator string.
- **window** (*int*) – Splitted column length.
- **upper** (*bool*) – Uppercase hexadecimal digits.

Returns

str – A wrapped and columnized hexadecimal representation of the data.

Example

```
>>> hexlify(b'Hello, World!', sep='.')
'48.65.6C.6C.6F.2C.20.57.6F.72.6C.64.21'
```

```
>>> hexlify(b'Hello, World!', 6, ' ')
'48 65 6C\n6C 6F 2C\n20 57 6F\n72 6C 64\n21'
```

4.10.6 hexrec.utils.parse_int

hexrec.utils.**parse_int**(*value*)

Parses an integer.

Parameters

value (Union[str, Any]) – A generic object to convert to integer. In case *value* is a `str` (case-insensitive), it can be either prefixed with `0x` or postfixed with `h` to convert from an hexadecimal representation, or prefixed with `0b` from binary; a prefix of only `0` converts from octal. A further suffix of `k` or `m` scales as *kibibyte* or *mebibyte*. A `None` value evaluates as `None`. Any other object class will call the standard `int()`.

Returns

int – None if *value* is `None`, its integer conversion otherwise.

Examples

```
>>> parse_int('-0xABk')
-175104
```

```
>>> parse_int(None)
```

```
>>> parse_int(123)
123
```

```
>>> parse_int(135.7)
135
```

4.10.7 hexrec.utils.unhexlify

hexrec.utils.**unhexlify**(*hexstr*)

Converts a hexadecimal text line into bytes.

Parameters

hexstr (*str*) – A hexadecimal text line. Whitespace is removed, and the resulting total length must be a multiple of 2.

Returns

bytes – Text converted into byte data.

Example

```
>>> unhexlify('48656C6C 6F2C2057 6F726C64 21')
b'Hello, World!'
```

4.11 hexrec.xxd

Emulation of the xxd utility.

CHAPTER

FIVE

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

5.1 Bug reports

When reporting a bug please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.2 Documentation improvements

hexrec could always use more documentation, whether as part of the official hexrec docs, in docstrings, or even on the web in blog posts, articles, and such.

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/TexZK/hexrec/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that code contributions are welcome :)

5.4 Development

To set up *hexrec* for local development:

1. Fork hexrec (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/hexrec.git
```

3. Create a branch for local development:

```
git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

5. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`).
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

CHAPTER

SIX

AUTHORS

- Andrea Zoppi - main developer - https://github.com/TexZK

6.1 Special thanks

- Scott Finneran - main developer of the SRecord project - https://srecord.sourceforge.net/

CHAPTER

SEVEN

CHANGELOG

7.1 0.3.1 (2024-01-23)

- Added support for Python 3.12.
- Added Motorola header editing.
- Minor fixes and changes.

7.2 0.3.0 (2023-02-21)

- Added support for Python 3.11, removed 3.6.
- Deprecated `hexrec.blocks` module entirely.
- Using `bytesparse` for virtual memory management.
- Improved repository layout.
- Improved testing and packaging workflow.
- Minor fixes and changes.

7.3 0.2.3 (2021-12-30)

- Removed dependency of legacy pathlib package; using Python's own module instead.
- Added support for Python 3.10.
- Fixed maximum SREC length.
- Changed pattern offset behavior.
- Some alignment to the `bytesparse.Memory` API; deprecated code marked as such.

7.4 0.2.2 (2020-11-08)

- Added workaround to register default record types.
- Added support for Python 3.9.
- Fixed insertion bug.
- Added empty space reservation.

7.5 0.2.1 (2020-03-05)

- Fixed flood with empty span.

7.6 0.2.0 (2020-02-01)

- Added support for current Python versions (3.8, PyPy 3).
- Removed support for old Python versions (< 3.6, PyPy 2).
- Major refactoring to allow an easier integration of new record formats.

7.7 0.1.0 (2019-08-13)

- Added support for Python 3.7.

7.8 0.0.4 (2018-12-22)

- New command line interface made with Click.
- More testing and fixing.
- Some refactoring.
- More documentation.

7.9 0.0.3 (2018-12-04)

- Much testing and fixing.
- Some refactoring.
- More documentation.

7.10 0.0.2 (2018-08-29)

- Major refactoring.
- Added most of the documentation.
- Added first drafts to manage blocks of data.
- Added first test suites.

7.11 0.0.1 (2018-06-27)

- First release on PyPI.
- Added first drafts to manage record files.
- Added first emulation of xxd.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

S

T

U

W