bin-parser Documentation

Release stable

Jun 21, 2018

Contents:

This library provides general binary file parsing by interpreting documentation of a file structure and data types. By default, it supports basic data types like big-endian and little-endian integers, floats and doubles, variable length (delimited) strings, maps and bit fields (flags) and it can iterate over sub structures. Other data types are easily added.

The file structure and the types are stored in nested dictionaries. The structure is separated from the types, this way multiple file formats using the same types (within one project for example) can be easily supported without much duplication.

The design of the library is such that all operations can be reversed. This means that fully functional binary editing is possible using this implementation; first use the reader to convert a binary file to a serialised dictionary representation, this representation is easily edited using a text editor, and then use the writer to convert back to binary.

This idea is implemented in two languages; Python and JavaScript. All main development is done in Python. We chose YAML as our preferred serialised dictionary format, but other serialisation formats (JSON for example) can be used too.

Please see ReadTheDocs for the latest documentation.

CHAPTER 1

Introduction

1.1 Why this library?

Writing a parser for binary files requires reverse engineering skills, knowledge of encodings, and above all, patience and intuition. Once all knowledge is gathered, the person doing the reverse engineering usually writes a parser and, if we are lucky, leaves some documentation. We try to facilitate this process by providing the tools to do this in a uniform way. In essence, we document the knowledge we gain from the reverse engineering process and use this documentation directly in a parser.

Since the bulk of the types stored in binary files are standard, dedicated parsers contain a lot of boiler plate code. We try to minimise this by providing a framework where all knowledge is recorded in a human readable format (YAML files) while the obligatory boiler plate code is incorporated in the library.

1.2 Background

In the following example, we read two bytes from an input stream, convert the read data to an integer and store it in an output dictionary under the keys `weight` and `age`.

```
output['weight'] = s_char_to_int(input_handle.read(1))
output['age'] = s_char_to_int(input_handle.read(1))
```

This approach results in file specific literals (like `weight` and `1`) and data type conversions (`s_char_to_int()`) directly in source code. This has several disadvantages:

- It is difficult to see what the file format is. This can be deduced only from the source code of the developed parser.
- A separate piece of software needs to be implemented if the conversion needs to be reversed.
- The parser is not portable to other programming languages.

Within the framework of this library, we attempt to solve the aforementioned problems.

By first defining the types, we can reuse them easily:

```
---
s_char:
  function:
    name: struct
```

Now we can use the type `s_char` in our structure definition:

```
---
- name: weight
  type: s_char
- name: age
  type: s_char
```

By recording the file structure this way, the knowledge of the file format and the implementation of the parser are strictly separated. This has the following advantages:

- The file format is documented in a human readable way.
- Reading and writing of the file format is supported.
- The parser is portable.

1.3 Approach

In order to parse a binary file, the library needs two pieces of information: it needs to know what the structure of the binary file is and it needs to know which types are used. Both of these information sources are provided to the library as nested dictionaries.

1.3.1 Example: Personal information

Suppose we have a file (`person.dat`) that contains the following:

- An age (one byte integer).
- A name (zero delimited string).
- A weight (an other one byte integer).

To make a parser for this type of file, we need to create a file that contains the type definitions. We name this file `types.yml`.

```
---
types:
  s_char:
    function:
      name: struct
  text:
    delimiter:
      - 0x00
```

Then we create a file that contains the definition of the structure. This file we name `structure.yml`.

```
---
- name: age
  type: s_char
- name: name
```

(continues on next page)

(continued from previous page)

```
  type: text
- name: weight
  type: s_char
```

We can now call the command line interface as follows:

```
bin_parser read person.dat structure.yml types.yml person.yml
```

This will result in a new file, named `person.yml`, which contains the content of the input file (`person.dat`) in a human (and machine) readable format:

```
---
age: 36
name: John Doe
weight: 81
```

1.4 Limitations

The main assumption made is that the binary files are *linearly parsable*. File seeking or multiple passes over an input file are not supported. Also, there is no support for the chaining of data types, so currently, compressed and encrypted files are not supported.

CHAPTER 2

Installation

The software is distributed via PyPI and npm for the Python and JavaScript implementations respectively.

2.1 Python

The Python version of the package is installed with `pip`:

```
pip install bin-parser
```

2.2 JavaScript

The JavaScript version of the package is installed with `npm`:

```
npm install bin-parser
```

2.3 From source

The source is hosted on GitHub, to install the latest development version, use the following commands.

```
git clone https://github.com/jfjlaros/bin-parser.git
cd bin-parser
pip install .
```

For the JavaScript implementation, replace the last command with:

```
npm install .
```

CHAPTER 3

Command line usage

A command line interface is available for both implementations. Apart from some implementation details concerning standard streams, their behaviour is identical.

3.1 Python

To convert a binary file to YAML, use the `read` subcommand:

```
bin_parser read input.bin structure.yml types.yml output.yml
```

To convert a YAML file to binary, use the `write` subcommand:

```
bin_parser write input.yml structure.yml types.yml output.bin
```

3.2 JavaScript

To convert a binary file to YAML, use the `read` subcommand:

```
./node_modules/.bin/bin_parser read input.bin structure.yml types.yml \
  output.yml
```

To convert a YAML file to binary, use the `write` subcommand:

```
./node_modules/.bin/bin_parser write input.yml structure.yml types.yml \
  output.bin
```

Please note that when installing from source, the `bin_parser` executable is not installed. Instead run the script `cli.js` as follows:

```
nodejs javascript/cli.js
```

CHAPTER 4

Types

Types, constants, defaults and macros are defined in a nested dictionary which is usually serialised to YAML. This file, usually named `types.yml`, consists of three (optional) sections; `types`, `constants`, `defaults` and `macros`. In general the types file will look something like this:

```
---
constants:
  multiplier: 10
defaults:
  size: 2
types:
  s_char:
    size: 1
    function:
      name: struct
  text:
    delimiter:
      - 0x00
```

4.1 Types

A type consists of two subunits controlling two stages; the acquirement stage and the processing stage.

The acquirement stage is controlled by the `size` and `delimiter` parameters, the size is given in number of bytes, the delimiter is a list of bytes. Usually specifying one of these parameters is sufficient for the acquisition of the data, but in some cases, where for example we have to read a fixed sized block in which a string of variable size is stored, both parameters can be used simultaneously. Once the data is acquired, it is passed to the processing stage.

The processing stage is controlled by the `function` parameter, it denotes the function that is responsible for processing the acquired data. Additional parameters for this function can be supplied by the `args` parameter.

4.1.1 Basic types

In version 0.0.14 the `struct` type was introduced to replace basic types like `int`, `float`, etc. and simple compound data types. The formatting parameter `fmt` is used to control how a value is packed or unpacked. For example, a 4-byte little-endian integer uses the formatting string `'<i'` and a big-endian unsigned long uses the formatting string `'>L'`. To avoid any issues with serialisation to YAML (the > sign may cause problems), it is recommended to quote the string.

For a complete overview of the supported basic types, see the Python struct documentation or our extensive list of examples.

4.1.2 Examples

The following type is stored in two bytes and is processed by the `text` function:

```
id:
  size: 2
  function:
    name: text
```

This type is stored in a variable size array delimited by `0x00` and is processed by the `text` function:

```
comment:
  delimiter:
    - 0x00
  function:
    name: text
```

We can pass additional parameters to the `text` function, in this case split on the character `0x09`, like so:

```
comment:
  delimiter:
    - 0x00
  function:
    name: text
    args:
      split:
        - 0x09
```

A 2-byte little-endian integer is defined as follows:

```
int:
  size: 2
  function:
    name: struct
    args:
      fmt: '<h'
```

And a 4-byte big-endian float is defined as follows:

```
float:
  size: 4
  function:
    name: struct
    args:
      fmt: '>f'
```

4.1.3 Compound types

Simple compound types can also be created using the `struct` function. By default this will return a list of basic types, which can optionally be mapped using an annotation list. Additionally, a simple dictionary can be created by labeling the basic types.

In the following example, we read three unsigned bytes, by providing a list of labels, the first byte is labelled `r`, the second one `g`, and the last one `b`. If the values are 0, 255 and 128 respectively, the resulting dictionary will be: `{'r': 0, 'g': 255, 'b': 128}`.

```
colour:
  size: 3
  function:
    name: struct
    args:
      fmt: 'BBB'
      labels: [r, g, b]
```

Values can also be mapped using an annotation list to improve readability. This procedure replaces specific values by their annotation and leaves other values unaltered. Note that mapping multiple values to the same annotation will break reversibility of the parser.

In the following example, we read one 4-byte little-endian unsigned integer and provide annotation for the maximum and minimum value. If the value is 0, the result will be `unknown`, if the value is 10, the result will be 10 as well.

```
date:
  size: 4
  function:
    name: struct
    args:
      fmt: '<I'
      annotation:
        0xffffffff: defined
        0x00000000: unknown
```

Labels and annotation lists can be combined.

4.2 Constants

A constant can be used as an alias in `structure.yml`. Using constants can make conditional statements and loops more readable.

4.3 Defaults

To save some space and time writing types definitions, the following default values are used:

- `size` defaults to `1`.
- `function` defaults to the name of the type.
- If no name is given, the type defaults to `raw` and the destination is a list named __raw__.

So, for example, since a byte is of size 1, we can omit the `size` parameter in the type definition:

```
byte:
  function:
    name: struct
```

In the next example the function `text` will be used.

```
text:
  size: 2
```

And if we need an integer of size one which we want to name `struct`, we do not need to define anything.

If the following construction is used in the structure, the type will default to `raw`:

```
- name:
  size: 20
```

4.3.1 Overrides

The following defaults can be overridden by adding an entry in the `defaults` section:

- `delimiter` (defaults to `[]`).
- `name` (defaults to `''`).
- `size` (defaults to 1).
- `type` (defaults to `text`).
- `unknown_destination` (defaults to `__raw__`).
- `unknown_type` (defaults to `raw`).

4.4 Macros

Macros were introduced in version 0.0.15 to define complex compound types. A macro is equivalent to a sub structure, which are also used in the structure definition either as is, or as the body of a loop or conditional statement.

In the following example, we have a substructure that occurs more than once in our binary file. We have two persons, of which the name, age, weight and height are stored. Using a flat file structure will result in something similar to this:

```
---
- name: name_1
- name: age_1
  type: u_char
- name: weight_1
  type: u_char
- name: height_1
  type: u_char
- name: name_2
- name: age_2
  type: u_char
- name: weight_2
  type: u_char
- name: height_2
  type: u_char
```

Note that we have to choose new variable names for every instance of a person. This makes downstream processing quite tedious. Furthermore, code duplication makes maintenance tedious.

The `structure` directive can be used to group variables in a substructure. This solves the variable naming issue, but it does not solve the maintenance issue.

```
---
- name: person_1
  structure:
  - name: name
  - name: age
    type: u_char
  - name: weight
    type: u_char
  - name: height
    type: u_char
- name: person_2
  structure:
  - name: name
  - name: age
    type: u_char
  - name: weight
    type: u_char
  - name: height
    type: u_char
```

We can define a macro in the `types.yml` file by adding a section named `macros` where we describe the structure of the group of variables.

```
---
types:
  u_char:
    function:
      name: struct
      args:
        fmt: 'B'
  text:
    delimiter:
      - 0x00
macros:
  person:
    - name: name
    - name: age
      type: u_char
    - name: weight
      type: u_char
    - name: height
      type: u_char
```

This macro can then be used in the `structure.yml` file in almost the same we we use a basic type.

```
---
- name: person_1
  macro: person
- name: person_2
  macro: person
```

A common substructure in binary formats is a data field preceded by its length, e.g., a string preceded by its length as a little endian 32-bit unsigned integer: `\x0b\x00\x00\x00hello world`. In the size_string example we show

how we can use a macro to facilitate this.

Macros can also be used to define variable types, i.e., a type that depends on the value of a previously defined variable. In the var_type example, we show how this can be accomplished.

CHAPTER 5

Structure

After having defined the basic types, the structure of the binary file can be recorded in a separate nested dictionary which is usually serialised to YAML. This file, usually named `structure.yml` contains the general structure of the binary file.

5.1 Flat structure

A simple flat structure is recorded as a list in which, for every variable, we supply a name and a type. In the following example we see the definition of a simple flat structure containing two short integers and one text field.

```
---
- name: year_of_birth
  type: short
- name: name
  type: text
- name: balance
  type: short
```

5.2 Loops and conditionals

Both loops and conditionals (except the `for` loop) are controlled by an evaluation of a logic statement. The statement is formulated by specifying one or two *operands* and one *operator*. The operands are either constants, variables or literals. The operator is one of the following:

operator	binary	explanation
not	no	Not.
and	yes	And.
or	yes	Or.
xor	yes	Exclusive or.
eq	yes	Equal.
ne	yes	Not equal.
ge	yes	Greater then or equal.
gt	yes	Greater then.
le	yes	Less then or equal.
lt	yes	Less then.
mod	yes	Modulo.
contains	yes	Is a sub string of.

A simple test for truth or non-zero can be done by supplying one operand and no operator.

5.3 for loops

A simple for loop can be made as follows.

```
- name: fixed_size_list
  for: 2
  structure:
    - name: item
    - name: value
      type: s_char
```

The size can also be given by a variable.

```
- name: size_of_list
  type: s_char
- name: variable_size_list
  for: size_of_list
  structure:
    - name: item
    - name: value
      type: s_char
```

5.4 while loops

The do-while loop reads the structure as long as the specified condition is met. Evaluation is done at the end of each cycle, the resulting list is therefore at least of size 1.

```
- name: variable_size_list
  do_while:
    operands:
      - value
      - 2
    operator: ne
  structure:
```

(continues on next page)

(continued from previous page)

```
    - name: item
    - name: value
      type: s_char
```

The `while` loop first reads the first element of the structure and if the specified condition is met, the rest of the structure is read. Evaluation is done at the start of the cycle, the resulting list can therefore be of size 0. The element used in the last evaluation (the one that terminates the loop), does not have an associated structure, so its value is stored in the variable specified by the `term` keyword.

```
- name: variable_size_list
  while:
    operands:
      - value
      - 2
    operator: ne
    term: list_term
  structure:
    - name: value
      type: s_char
    - name: item
```

When using this structure on the input `\x01hello\x00\x03world\x00\x02`, the result will be as follows.

```
list_term: 2
variable_size_list:
- item: hello
  value: 1
- item: world
  value: 3
```

5.5 Conditionals

A variable or structure can be read conditionally using the `if` statement.

```
- name: something
  type: s_char
- name: item
  if:
    operands:
      - something
      - 2
    operator: eq
```

5.6 Complex conditionals

More complex conditional statements can be built by using nesting. The following example evaluates the expression `(1 == 2) or True`.

```
- name: item
  if:
    operands:
```

(continues on next page)

(continued from previous page)

```
      - operands:
          - 1
          - 2
        operator: eq
      - true
    operator: or
```

Also see complex_eval for a working example.

5.7 Notes on evaluation

Since we use a general way of evaluating expressions, there are usually multiple ways of writing such an expression. For example, the following statements are equal:

Implicit truth test.

```
- name: item
  if:
    operands:
      - something
```

Explicit truth test.

```
- name: item
  if:
    operands:
      - something
      - true
    operator: eq
```

Explicit non-false test.

```
- name: item
  if:
    operands:
      - something
      - false
    operator: ne
```

CHAPTER 6

Library

While the command line interface can be used to parse a binary file when the correct types and structure files are provided, it may be useful to have a dedicated interface for specific file types. It could also be that the current library does not provide all functions required for a specific file type. In these cases, direct interfacing to the library is needed.

6.1 Basic usage

For both implementations we provide a `BinReader` and a `BinWriter` object that are initialised with the input file and the types and structure definitions.

6.1.1 Python

To use the library from our own code, we need to use the following:

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

from bin_parser import BinReader


parser = BinReader(
    open('balance.dat', 'rb').read(),
    yaml.safe_load(open('structure.yml')),
    yaml.safe_load(open('types.yml')))

print('{}\n'.format(parser.parsed['name']))
print('{}\n'.format(parser.parsed['year_of_birth']))
print('{}\n'.format(parser.parsed['balance']))
```

The `BinReader` object stores the original data in the `data` member variable and the parsed data in the `parsed` member variable.

6.1.2 JavaScript

Similarly, in JavaScript, we use the following:

```
#!/usr/bin/env node

'use strict';

var fs = require('fs'),
    yaml = require('js-yaml');

var BinParser = require('../../javascript/index');

var parser = new BinParser.BinReader(
  fs.readFileSync('balance.dat'),
  yaml.load(fs.readFileSync('structure.yml')),
  yaml.load(fs.readFileSync('types.yml')),
  {})

console.log(parser.parsed.name);
console.log(parser.parsed.year_of_birth);
console.log(parser.parsed.balance);
```

6.2 Defining new types

See prince for a working example of a reader and a writer in both Python and JavaScript.

6.2.1 Python

Types can be added by subclassing the BinReadFunctions class. Suppose we need a function that inverts all bits in a byte. We first have to make a subclass that implements this function:

```
from bin_parser import BinReadFunctions

class Invert(BinReadFunctions):
    def inv(self, data):
        return data ^ 0xff
```

By default, the new type will read one byte and process it with the inv function. In this case there is no need to define the type in types.yml.

Now we can initialise the parser using an instance of the new class:

```
parser = bin_parser.BinReader(
    open('something.dat', 'rb').read(),
    yaml.safe_load(open('structure.yml')),
    yaml.safe_load(open('types.yml')),
    functions=Invert())
```

6.2.2 JavaScript

Similarly, in JavaScript, we make a prototype of the BinReadFunctions function.

```
var Functions = require('../../../javascript/functions');

function Invert() {
  this.inv = function(data) {
    return data ^ 0xff;
  };

  Functions.BinReadFunctions.call(this);
}
```

Now we can initialise the parser with the prototyped function:

```
var parser = new BinParser.BinReader(
  fs.readFileSync('something.dat'),
  yaml.load(fs.readFileSync('structure.yml')),
  yaml.load(fs.readFileSync('types.yml')),
  {'functions': new Invert()});
```

CHAPTER 7

Extras

In this section we discuss a number of additional features and programs included in this project.

7.1 Debugging

The parser and the writer support four debug levels, controlled via the `-d` option of the command line interface.

level	description
0	No debugging.
1	Show general debugging information and internal variables.
2	Show general debugging information and parsing details.
3	Show all debugging information.

7.1.1 General debugging information

The section `DEBUG INFO` contains some general debugging information.

For the parser it contains:

- The file position after the parsing has finished and the size of the file. Something is wrong if these two values are not equal.
- The number of bytes that have been parsed and assigned to variables. This is all the data that has not been assigned to the `__raw__` list.

For the writer this section only contains the number of bytes written.

7.1.2 Internal variables

The section `INTERNAL VARIABLES` contains the internal key-value store used for referencing previously read variables.

7.1.3 Parsing details

The section named `PARSING DETAILS` contains a detailed trace of the parsing or writing process. Every line represents either a conversion or information about substructures.

For the parser, a conversion line contains the following fields:

field	description
1 :	File position.
2	Field content.
(3)	Field size (not used for strings).
--> 4	Variable name.

In the following example, we see how the file from our balance example is parsed.

```
0x000000: cf 07 (2) --> year_of_birth
0x000002: John Doe --> name
0x00000b: 8a 0c (2) --> balance
```

For the writer, a conversion line contains the following fields:

field	description
1 :	File position.
2	Variable name.
--> 3	Field content.

In the following example, we see how the file from our balance example is written.

```
0x000000: year_of_birth --> 1999
0x000002: name --> John Doe
0x00000b: balance --> 3210
```

The start of a substructure is indicated by `--` followed by the name of the substructure, the end of a substructure is indicated by `-->` followed by the name of the substructure.

7.2 `make_skeleton`

To facilitate the development of support for a new file type, the `make_skeleton` command can be used to generate a definition stub. It takes an example file and a delimiter as input and outputs a structure and types files definition. The input file is scanned for occurrences of the delimiter and creates a field of type `raw` for the preceding bytes. All fields are treated as delimited variable length strings that are processed by the `raw` function, as a result, all fixed sized fields are appended to the start of these strings.

7.2.1 Example

Suppose we know that the string delimiter in our balance example is `0x00`. We can create a stub for the structure and types definitions as follows:

```
make_skeleton -d 0x00 balance.dat structure.yml types.yml
```

The `-d` parameter can be used multiple times for multi-byte delimiters.

This will generate the following types definition:

```
---
types:
  raw:
    delimiter:
    - 0x00
    function:
      name: raw
  text:
    delimiter:
    - 0x00
```

with the following structure definition:

```
---
- name: field_000000
  type: raw
- name: field_000001
  type: raw
```

The performance of these generated definitions can be assessed by using the parser in debug mode:

```
bin_parser read -d 2 \
  balance.dat structure.yml types.yml balance.yml 2>&1 | less
```

which gives the following output:

```
0x000000: <CF>^GJohn Doe --> field_000000
0x00000b: <8A>^L --> field_000001
```

We see that the first field has two extra bytes preceding the text field. This is an indication that one or more fields need to be added to the start of the structure definition. If we also know that in this file format only strings and 16-bit integers are used, we can change the definitions as follows.

We remove the `raw` type and add a type for parsing 16-bit integers:

```
---
types:
  short:
    size: 2
    function:
      name: struct
      args:
        fmt: '<h'
  text:
    delimiter:
      - 0x00
```

and we change the structure to enable parsing of the newly found integers:

```
---
- name: number_1
  type: short
- name: name
  type: text
- name: number_2
  type: short
```

By iterating this process, reverse engineering of these types of file formats is greatly simplified.

7.3 compare_yaml

Since YAML files are serialised dictionaries or JavaScript objects, the order of the keys is not fixed. Also, differences in indentation, line wrapping and other formatting differences can lead to false positive detection of differences when using rudimentary tools like `diff`.

`compare_yaml` takes two YAML files as input and outputs differences in the content of these files:

```
compare_yaml input_1.yaml input_2.yaml
```

The program recursively compares the contents of dictionaries (keys), lists and values. The following differences are reported:

- Missing keys at any level.
- Lists of unequal size.
- Differences in values.

When a difference is detected, no further recursive comparison attempted, so the list reported differences is not guaranteed to be complete. Conversely, if no differences are reported, then the YAML files are guaranteed to have the same content.

7.4 sync_test

To keep the Python- and JavaScript implementations in sync, we use a shell script that compares the output of both the parser and the writer for various examples.

```
./extras/sync_test
```

This will perform a parser test and an invariance test for all examples.

7.4.1 Parser test

This test uses the Python- and JavaScript implementation to convert from binary to YAML. `compare_yaml` is used to check for any differences.

7.4.2 Invariance test

This test performs the following steps:

1. Use the Python implementation to convert from binary to YAML.
2. Use the Python implementation to convert the output of step 1 back to binary.
3. Use the JavaScript implementation to convert the output of step 1 back to binary.
4. Use the Python implementation to convert the output of step 2 to YAML.

The output of step 1 and 4 is compared using `compare_yaml` to assure that the generated YAML is invariant under conversion to binary and back in the Python implementation. The two generated binary files in step 2 and 3 are compared with `diff` to confirm that the Python- and JavaScript implementations behave identically.

Note that the original binary may not be invariant under conversion to YAML and back. This is the case when variable length strings within fixed sized fields are used.

CHAPTER 8

Contributors

- Jeroen F.J. Laros <J.F.J.Laros@lumc.nl> (Original author, maintainer)
- Daniel S. Katz <d.katz@ieee.org>
- Jamie Ross <jamie.ross@electricityexchange.ie>
- Matthew Fernandez <matthew.fernandez@gmail.com>
- Robert Haines <rhaines@manchester.ac.uk>

Find out who contributed:

```
git shortlog -s -e
```