

The INC File Format: Simple, INCLUSIVE Metadata for Everyone (Who Uses Tabular Data)

INC is

- INi-Csv

or

- Ini - aNd - Csv

or

- INCluded metadata

or

- Intrinsic aNd Connate metadata

Or we could have called it

Simple Pragmatic Annotated Metadata (SPAM)

Obligatory XKCD



<https://xkcd.com/927/>

Introduction

Metadata is information that describes or provides context for other data. It is often used to organize, manage, and search for digital content. Metadata is also important for preserving digital content over time. By providing information about how a file was created and how it should be accessed, metadata ensures that the content is more portable both across various systems and also across time. In addition, Metadata can also be used for security, by providing access-control, management and auditing information. It can also provide checks that the data has not become corrupted in some manner.

For data scientists, however, metadata is even more crucial. For data scientists, data isn't just a video to watch, it is *data* that will be processed through an algorithm. Metadata provides the crucial details for interpretation, such as units. It also provides the *covariates* – i.e., the variables against which results are indexed – that are the core of any study. For example, in textual analysis, you need to know the author of a text in order to compare different writers.

Metadata can also include information about the process used to collect the data, specifically issues that might affect data quality, such as sample sizes or missing data.

There has been a proliferation of datasets in the modern world, but although metadata's importance is a well-known, there are too many instances to count where lack of metadata has made an important dataset less valuable if not totally useless.

Part of the problem is that it can be hard to do metadata well, particularly for novices. Light-weight processes and tools alleviate this issue. There are a number of efforts to create light-weight data formats that facilitate "frictionless data." However, we argue there that they aren't working. If they were, we would not have endless conversations about metadata.

This note proposes a simple file format that incorporates metadata into a data file (here a CSV file) to create a very simple file format. The core philosophical basis for the proposal is that 80% of cases can be handled with 20% of the work. Generic, powerful, expressive formats like XML may be able to include all datasets, but their expressiveness comes at a cost. That cost is that they aren't used (enough), or are used inexpertly (which can sometimes be worse).

The success of the CSV format is a case in point. There are a large number of datasets that aren't suitable for CSV files, but nevertheless CSV is very, very useful because the number of tabular datasets is large. We seek to extend the CSV format and its simplicity with the addition of the simplest but most important metadata.

In more detail, our file format is based on a set of tenets (though note that these are not orthogonal, and have some overlaps):

1. **Simplicity:** The goal is to have a file format that is as easy to use as CSV. It should aim at working for the common case, so it must be flexible *enough*, but no more. *We will sacrifice expressivity for simplicity.*
2. **Inclusion:** Metadata must be **included inside** the file: Data that is held externally (e.g., in a central database), or maintained by a file system (e.g., edit dates) is easily separated from its data or corrupted. Metadata that is in the file itself is better protected.
3. **Readability:** Our format should be *easily* Machine-Readable (MR) and *easily* Human-Readable (HR), or really *Human Friendly* (HF). The need for MR is obvious and the ease of such is important to facilitate efficient I/O and data transfer. But HF is also important because it strongly facilitates ease of use and debugging of data protocols. In particular, here we mean HR by non-specialists, e.g., consumers of the data rather than programmers or data specialists. We also want the data to be human editable, so that a human can add metadata. That requires a higher level of ease of use than just the ability to see the character.
4. **Portability:** Data portability should be a core requirement for any new data format. It entails the file format be well documented, avoid machine specific details (like bit ordering) and be supportable by multiple code bases.
5. **Efficiency:** We need to reuse code and standards as much as possible. Reuse reduces the cost of development, but also ongoing maintenance and debugging. That is important for portability, for instance, because any standard like this should be supported by as many code bases as possible for as long as possible.
6. **Rigour:** Our format should be a rigorous container for non-rigorous material. That is, we will not provide means to enforce schemas about the material contained, but will for the structure of the overall container. In this way it is a little like the packet encapsulation of a communications packet, e.g., with a header and trailer that is precisely defined around arbitrary encapsulated data. Underlying this is a need to **inform rather than control content; and control rather than inform structure**. That is, content details should be up to the user, but structure should be tightly defined. This could be considered in opposition to [Postel's robustness principle](#), but we argue it is better to throw an error and prevent invalid data or metadata from corrupting future processes, than to warn, and potential allow uncontrolled issues to be amplified, as in Marshall Rose's criticism.
7. **Convention over code:** Related to the previous idea is the use of the software paradigm where the number of decisions made by developers is minimised, for instance by providing *sensible defaults* and common pre-defined terms. In doing so there is a potential loss of flexibility, but the advantage is expressed sometimes as "the principle of least astonishment." That is, unconventional, suprising and conflicting results are reduced.

And to reiterate, we aim at the common case of tabular data. Our key use case for this format is not serialisation of complex datasets, but storage of common table-based information, e.g., 2D arrays of data. Moreover, we envisage large-scale analysis, archival storage, and bulk copying of such files, not continuous streams of dynamic data.

We also envisage the common case where the associated metadata is quite simple, as is typical in many real applications. Because the data itself is in a simple table-based array, complex structural descriptions of the data, including schemas, are not needed. However, some information about the table of data can help remediate some of the common problems with CSV style files.

Why do it our way, and why do we need YAS?

Why YAS (Yet Another Standard), or more specifically, how does this differ from other efforts?

Perhaps the most contravertial aspect of this proposal is the idea that metadata should be included in data files themselves. It is seemingly more common for it to be held in a central repository.

But incorporation of metadata into a file is not a new concept – it is prevalent in many fields, perhaps most obviously in traditional analogue media like books. Most books contain a large proportion of their metadata internally, for instance the title and author on the cover, and copyright information on the first page. It's much easier to use data in this form, than the (old school) central library card catalogue.

Web page design is another example where a header field is used to populate many common forms of metadata (using HTML). In this case it is particularly obvious why the metadata must be incorporated – web pages are inherently mobile, and it would be very easy for any separate record to become detached.

Digital photographs are yet another example where including metadata (EXIF data) in the file was considered essential. But this idea didn't start with digital photographs – it was common, back in the day, to write some information about a photograph on the back of the photograph. That way, even if the photo was displaced (from its album or other store) the critical information about it would not be forgotten.

Yet another example of “in-file” metadata is in filename, which are commonly chosen to incorporate some aspect of metadata (such as, e.g., author name). This has obvious limitations (name length), but like file-system metadata is highly vulnerable when data is mobile between systems, which may change file names, either by truncation or case changes.

So it is common to have both internal and external metadata to some degree, but the proportion depends on the application. Here we note that data scientists are not cataloguers: one major tradition towards metadata comes from the bibliographic context of organising and providing search and recovery mechanisms for objects or resources. Data scientists don't use metadata in the same manner (in many cases). In a traditional library application, for instance, an historian might wish to find and access all of the source material from a set of authors, then take a subsection of this material to read closely. The number of sources that can be examined closely is small, and the majority of metadata for the sources is used to find the sources to be used. A data scientist would more likely be performing *distant reading* in which case the entire corpus would be analysed, and grouped (through software) by author, date or other variables. The difference is that in the first case the metadata is used primarily to source the material, and in the second it is used in the analysis en masse of the material. A centralised repository of metadata can serve for either, but is oriented primarily at the first case. Metadata contained in the files is less accessible for search, but is more directly useful for analysis.

Moreover, metadata held separately can easily be lost. This is most important for long-term archival of data, particularly when the data might be moved at some point. Increasingly, research organisations are realising that long-term storage and sharing of data are important parts of scientific reproducibility and hence credibility. But it is not just enough to hold the data, it must be usable. It is very easy that, over time, things that seem obvious (e.g. units) can be forgotten, let alone file formats or software used to access a file. Metadata has an important function in preservation and sharing of such data.

But a separate file can easily become dislocated from the data when that data is moved or shared. Metadata is also sometimes considered a function of a file system (which keeps details such as last-modified dates), but this can also become easily corrupted when files are shared. So there are a large number of places where include metadata can be very helpful.

So why YAS (Yet Another Standard)? There are standards like XML that can incorporate metadata into a file. We will consider this in more detail below, but essentially it comes down to the fact that although everyone acknowledges the importance of metadata, it is still done very badly or not at all. And part of the reason is that it is hard is that complex standards like XML make it harder. We want to make it easier for a common set of cases.

XML allows arbitrary hierarchy and structure for data and metadata. However, for very many applications metadata can be represented as flat, or at most 1-level hierarchy. There is much metadata that can be represented otherwise, but that isn't needed for almost everything that might be used to describe a table of data. A lot of the issues come from metadata that must capture a wide variety of resources ranging from photos, to videos, to audio, to text, to geospatial, to medical and to genetic data. Our use case here is metadata specifically for a 2D array of data. This case is restricted, but very useful.

XML allows arbitrary hierarchy, but at the cost of complexity. At the least, there are 10 ways to do anything in XML, and it is not obvious which to use. And software to work with XML is correspondingly complex and hard to use. Allowing links, complex hierarchies, and arbitrary data is therefore overkill, and comes with a big cost. Fundamentally, XML (and related formats) are trying to solve the larger problem of serialisation of arbitrarily-structured data.

Finally, CSV has been a successful tool for data portability. But although CSV is a commonly used tool, it is not a strict format, but rather a style of formats that might, more correctly, be called delimiter-tabular data. In CSV the delimiters are commas, and end-of-line (EOL) characters, but even if we limit ourselves to those decisions, there is much variability in CSV, for instance in whether it allows comments, how headers work, how missing data is represented, and how quotation marks are used. Most modern toolboxes for working with CSV files provide a mechanism to work with the many subtle variations, but they usually require some manual intervention. But with a small number of hints included in the file's metadata, all of these issues can be alleviated.

We are not the first to propose such modifications – for instance CSVY uses YAML to provide metadata in a header for a file. This is one of the inspirations for our approach, but CSV-YAML inherits YAML's problems, which are non-trivial. We seek to avoid these problems, and simplify everything to its core.

Use cases

In the following we elaborate on the proposed use cases for such a file format, to help understand how powerful even such a simple mechanism as we propose could be.

(i) Data collected as part of a medical study

A typical medical study would aggregate data from

1. Experimental results (e.g., the treatment administered, and survival-time of patients); and
2. Covariates about the patients (e.g., age, gender, ...).

Much other data is collected these days, but it is still very common to see data with this type of structure that is strongly amenable to tabular format. It is incredibly common that such data is held in a spreadsheet, for instance.

The metadata about such experiments is very important for (a) its interpretation, (b) later incorporation into metastudies. For instance, returning to the note about scientific reproducibility, there is an increasing organisation around the management of data, for instance through Data Management Plans (DMP), which are now expected as an automatic requirement for a project in many organisation. A secondary desire of this project is that it enables an automated pathway from a structured DMP to the data itself. That is, we should be able to parse the DMP, and create the required metadata, in particular data for rights management.

In a nutshell, if a study is pre-registered, it would be ideal to be able to match the registration details with the data automatically, and thus validate the data at a non-trivial level.

(ii) Results of the analysis of an archive

Libraries have amazing archives, for instance the NSW library has an archive of hundreds of the diaries of WW1 ANZACs. The data itself is a complex resource, but summary statistics of each resource are highly amenable to tabular form. Metadata for these summaries comes in two parts

1. Metadata for each diary (e.g., author, dates, ...)
2. Metadata for the summary (e.g., how it was created)

The latter is of interest here, in particular. It might describe the software and algorithm used to create the summaries, or it might detail statistical parameters used in creating the summary.

Note here that the metadata for the summary is quite small and quite simple compared to the original data.

(iii) Data created through a simulation

Simulations are an increasingly used tool to study complex systems. One of the features that make them desirable is their ability to generate vast amounts of data providing large-numbers of statistical samples of similar processes, or a simulations of a range of scenarios to answer “What if?” questions or optimise a system.

It is very beneficial (from computation cost, scientific validity and verification, and just simple ease of use points of view) to store intermediate simulation data, e.g., the lists of events, or measured values, for later analysis. But it is very important in such cases to document the parameters, metaparameters and details such as software versions used to create each file (corresponding to a single simulation run).

Files are created asynchronously, often on parallel/distributed computers, and often in tranches (as one learns more about the problem). Sometimes bugs are found in code, and some data needs to be regenerate (but maybe not all). Sometimes new parameter values are chosen for exploration. All of this requires careful metadata handling, and the easiest approach is to contain the metadata in the file corresponding to each simulation run.

The amount of metadata per simulation run is tiny, and it is usually very simple, often just a few numbers.

Summary of case studies

In each of these cases:

0. The data has an important aspect that is naturally tabular, hence the use of CSV to store the data.
1. The metadata is static, or nearly so. It is created, usually alongside the data, and changes rarely.
2. The metadata is small compared to the data. Some inefficiency is tolerable because it isn't a significant overhead compared to the data. Metadata is often a few hundred bytes, as compared to mega-, giga- or terabyte sized data.
3. Metadata is simple, but not, however, naturally tabular (e.g., the elements of metadata often take different types, don't have common units or common lengths). It doesn't need massive data structures, tables within tables, or multi-level hierarchies.
4. There are some core metadata that are common to many use cases (e.g., the source of data), but there are also some that are tied closely to the application and have no meaning outside that application (e.g., parameter values in a simulation).
5. It is advantageous for metadata to be (easily) human readable, but it must also be machine readable.

Our proposal

The simplest approach to all of the above is to co-opt existing formats and code, but there is a wide range of potential alternatives.

We choose to combine INI and CSV formats, arguing that this results in a simpler result than a bespoke approach to encapsulate data and metadata all together. Metadata and data have different characteristics (e.g., metadata is much smaller than data, typically) and hence different tools are needed.

HF implies that a typical user should not have to understand the INC standard. The standard's structure should be obvious, with very limited use of special characters or structures, and it should have as few syntactical elements that differ from convention as possible. That is one of the reasons INI files are chosen, and we refuse to add serious additional concepts to the format.

CSV is chosen because it is flexible, powerful, and very often used, but also has some simple flaws that can be corrected through our format. Our intention is that the file does not lose any CSV information, and is (with only minor tweaks) compatible with CSV readers and writers.

The overall file will be encoded in the UTF-8 variant of unicode (with no BOM). There are many advantages to (i) choosing a single encoding (notably portability and simplicity), and (ii) using UTF-8, namely:

- its variable length encoding is relatively efficient (amongst Unicode encodings) for digits, which form a large component of many datasets;
- it has become the dominant form of Unicode in many contexts (web and related, email, ...);

- is the natural Unicode string format for many languages (Go, Julia, Rust, Swift, ...);
- it is supported by most modern text editors and much other software;
- it is backward compatible (at a byte level) with ASCII; and
- it avoids (and recommends against) the need for a BOM (Byte Order Mark) at the start of a file

<https://en.wikipedia.org/wiki/UTF-8>

Component formats

Initialisation (INI) Files

The INI format is really a loose class of formats often used for operating system configuration files that takes the form of (name, value) pairs given simply as `name = value`, with some simple structural elements (sections). The name comes from their main use as *initialisation* files.

INI files has some characteristics that are useful for us. Configuration languages/files like INI are usually working on static, predefined concepts and data. Their interpreters are rarely required to evaluate complex expressions or do any more than trivial calculations.

Configurations languages are typically declarative (as is INI), though they may look imperative. However, to a large extent, the order of statements within configuration languages is unimportant, e.g., there is no modifiable state in these languages. The statements are instead expressing constraints on the final state of the configuration, and hence the language is declarative. This may vary somewhat (here we allow section blocks, for instance).

The syntax of configuration languages is usually much more limited than a fully fledged programming language also because they just don't need features like recursion.

These features make such languages easier to work with in many ways, for instance:

- low need for ordering;
- minimal interactions across a file; and
- constrained syntax;

and all these features make them easier to correctly write and read such files.

They are aimed at being *predictable*. They are at being What You See Is What You Get (WYSIWYG). They are therefore ideal for many tasks, when they are sufficient for the task.

Configuration languages have a set of functions that make them particularly suited for our task as well. In particular:

- Setting parameters;
- Grouping processes or parameters;
- Binding or linking or mapping components together; and
- Expressing constraints on valid or invalid behaviour.

Some aspects of configuration files are not needed here: for instance, they often have aids to modularisation and reuse such as the ability to include other (configuration) files. We argue that this would defeat the point of including metadata in the file. Configuration languages usually also include many system-specific components, and most of these will not be needed here.

Why INI in particular? Much of the reasoning is included below, but the most basic reason is its simplicity and readability. It does what we need, and nothing else.

TOML and YAML are intended to do a similar job, but (as we will talk about in detail below) they add too many features and despite their goal to be simple, and hence allow potentially very confusing structures that aren't necessary here.

Notably, TOML allows many ways to achieve a given end. That is appealing to programmers, but can reduce human readability because

- a human reader is only familiar with one 'dialect', or way to do things;

- a mixture of styles can be at best confusing;
- TOML allows whitespace to be ignored (e.g., in names), but this creates ambiguity in the mind of the reader – better to have WYSIWYG;
- machine written outputs will often look completely different, even when produced by the same code, making it hard for a human reader to compare two files.

Our INI does not remove all options, but only allow those that are needed for other reasons. For instance, order independence allows some variation in a file's appearance, but is needed to simplify many other aspects of the file structure.

We provide specific details before, but let us also talk about the data component of the file first.

Comma Separated Value (CSV) Files

The CSV (Comma Separated Value) format is a common approach to store *tabular* data (2D arrays or tables of data). Tabular data is a very common form of data as evidenced by the common use of spreadsheets and CSV files.

The success of these files is documented in Lyn et al. (2021), but also note its use in high-profile places such as the UK government (<https://www.gov.uk/guidance/using-csv-file-format>); the IMF (<https://www.imf.org/external/help/csv.htm>); and the US Library of Congress (<https://www.loc.gov/preservation/digital/formats/fdd/fdd000323.shtml>). Usage of CSV goes back to the early days of computers, back to the IBM Fortran compiler c1972.

When we talk of CSV we use the common convention that we are really discussing DSV (Delimiter Separated Value) files, as we are agnostic about the details of such files (as will be described below). That includes TSV (tab-separated) and PSV (pipe-separated) data (two other common and approximately equivalent formats). It is noteworthy that the examples above both provide extensive documentation on how to use CSV in their dialect.

<https://web.archive.org/web/20171227070107/http://www.catb.org/esr/writings/taoup/html/ch05s02.html>

There are really two separators in a CSV file: (i) the comma, used to separate fields in a record, and (ii) the EOL character(s) used to separate records. So a standard CSV file has one record per line and each record has the same number (and order) of fields, each separated by a comma.

That is where the loose idea of the format becomes more complicated. CSV files sometimes have

- one or more header lines that provide a name for each field of the records,
- comments (strings that are not part of the data),
- a syntax to allow commas to appear as part of the data, typically by quoting,
- a means to distinguish quotes and other characters that are not part of the data from those that are.

The problems are exacerbated in UNICODE documents where there can be multiple symbols that meet a definition, e.g., UNICODE has a list of over 20 characters commonly associated with quote marks (see <https://unicode-table.com/en/sets/quotation-marks/>), of which many are hard to distinguish visually, e.g., U+0022 and U+FF02.

All of these (and other details) lead to an additional layer of complexity. In addition, CSV files do not usually provide details such as the *type* of each field in a record, although a common assumption is that a field will have a uniform type across each record. Thus, software reading CSVs often has to infer the type, which can be challenging in cases, particularly where special strings are interspersed in numerical data, for instance, to indicate missing data.

However, most of these details could be easily included in the metadata for the file, which if read before the data, could provide the hints needed by software to easily circumvent these issues. That is one goal here.

Our file format

The top level of our file format is simply:

```

1  ---
2  INI (Metadata) PART
3  ---
4  CSV DATA PART
```

The INI Part will be a precisely defined and constrained variant of the class of INI formats (see below). It is used to store the metadata.

The CSV DATA Part will (here but maybe not in the future) be a DSV format, most commonly CSV, but we will allow a large part of that family.

We do not allow any extra complexity.

For instance, a file will contain exactly 1 dataset. There is no syntax to contain more than one table in a single file. That is a common place where data formats come unstuck – in order to allow such, they provide some mechanism, which then multiplies the complexity of the other parts because now metadata must include notation about which parts it applies to.

By default, all INC files are encoded in UTF-8 (the default recommended for XML and other file formats). We *might* allow that alternative codings should be possible, with the proviso that implementations be able to autodetect which variant encoding the file is, and to this end the file should include an “encoding”

Minor points:

- We actually allow the - - - to be any three-or-more consecutive dashes, noting that the “dash” character could be any of the Unicode symbols in the category *Punctuation*, dash denoted by Pd, and including some 20 or so symbols including dashes, hyphens and the minus sign (there are others <https://www.compart.com/en/unicode/category/Pd>) and matched by the PCRE regex `\{Pd\}`.

<https://www.regular-expressions.info/unicode.html>

- Why use - - - as the delimiter between parts? That comes from the common use of this symbol in similar formats such as CSVY. But also, three dashes is an uncommon symbol in metadata, and so its use will only rarely cause conflict. In such cases we require that three-or-more consecutive dashes be escaped using double quotes, ", specifically by U+0022 or U+FF02.
- Why have a - - - symbol at the start and middle but not the end? Placing it in the header provides a positive begin/end for the header. Avoiding it at the end avoids the need to escape three-or-more dashes that appear in the data where we might have limited ability to cause an arbitrary string to be quoted (we aim to use standard CSV packages for this component, and they might not escape strings like this). Note that this implies that if a regex is used to match to the header boundaries, then it must not be *greedy*. As a minor additional reason, many CSV file formats are agnostic about the need for an end-of-line indicator at the end of the file and we want to maintain this agnosticism.
- The - - - lines can contain whitespace and comments in order to add to human readability and writability, but should have nothing else, and there can be no whitespace between the dashes (normally, we will ignore whitespace, but not in this instance).

The INI-Part format

INI files consist primarily of a set of *properties* defined by (name, value) pairs in the form `name = value`, with one property defined per line of the file.

INI is an informal format defined by usage, but more variables have the following common features:

- A small number of reserved symbols, in our case the characters `[`, `]`, `=` and `;`
- Sections defined by a word in square brackets, providing a single layer of hierarchy.
- Comments preceeded by a particular character (often `;` or `#`).
- Names for sections and properties are case insensitive and must exclude reserved characters (but allow all other characters).
- The order of properties in a section is irrelevant (but properties are grouped into section by ordering).
- Whitespace lines (once comments are removed) are ignored.

We refine these as follows:

- We add the string - - - to the reserved symbols.

- We use ; or # for comments, to be consistent with common INI formats.
- Sections are monolithic, i.e., all properties of a section are listed together under the one section heading (which facilitates ease of use if the file becomes large), That is, a section is defined by a word in square brackets, and the properties that follow it until the next section declaration. There is a global section at the start.
- Duplicate names in a section are not allowed and throw an error.

Some INI formats allow nested sections, but we do not use that feature here.

We also make a number of further restrictions that are not common in INI formats, but we argue that these restrictions allow common cases while making the whole system more stable.

1. Names must exclude reserved symbols and whitespace. We do not provide an escape syntax or means to quote names to allow arbitrary strings. Doing so is an example of the type of unneeded complexity introduced by tools such as TOML.
2. Lines cannot be continued. This might seem a limitation to human readability, but most text editors today allow soft wrapping of text, so in the rare cases of long metadata strings, a single line is not a restriction, and it makes parsing significantly simpler.
3. All values are *casstable* strings or integers. There are, for instance, no numbers or dates, though the strings can hold these and they can be interpreted as such later. Explicit typing compromises HF, and implicit typing introduces the [Norway problem](#).
4. Leading and trailing whitespace are stripped from names and values (unless quoted for values).
5. Reserved symbols can be included in values by escaping.

Obviously many people will want to change those details, but they greatly enhance the rigour and readability of a file. Moreover, the number of cases where metadata NEEDS more is a very small percentage.

The most obvious issue is the inclusion of typing for values. TOML, YAML and others use implicit typing to determine if a value is a number, string, Boolean or date, or sometimes other types. This is a bad idea (see [here](#) for why). A simple problem case is the use of locale dependent decimal indicators (the . or , are both used, at least).

We leave type inference up to the higher-layers (described later), where code can be more specific and informed, rather than forcing the file format to implicitly define types. The one exception to this is we allow automatic inference of integers. Integer values (in metadata) are very common, and easy to infer (you might need to write "007" is James Bond needs it) and so it seems a small extra convenience with little cost.

Line continuation is also notionally easy but harder than it might seem to do well. For instance, one very common use for metadata is search for given records. If the metadata is kept on a common line, it facilitates search using simple unix tools like grep to search for (name,value) pairs that match a criteria. Allowing multiline strings greatly complicates such a search. Moreover, as soon as we allow line continuation, human readability of a file is likely to be compromised at some point.

Finally, excluding reserved characters and whitespace from names is common practice in almost all programming languages because of the extra parsing complexity this might impose. Some approaches try to allow whitespace in names, so (it seems) to allow more natural names for properties, but this is just not needed—camel-case or underscore separated strings obviates the need. For instance `quote_string` instead of `"quote string"`. And this then avoids problems when these names are later passed up to another programming language where it is highly likely that symbol names don't allow whitespace.

Mini-Schemas

One guiding principle here is "convention over code." That is, using the software paradigm where the number of decisions made by developers is minimised. Thus we define only one *reserved* section, the `[structure]` section. This section is to be used to describe the following CSV field.

However, metadata without conventions or descriptions can lose some of its utility. There are, for instance, several standards for metadata, eg the National Information Standards Organization (NISO) <https://groups.niso.org/higherlogic/ws/public/download/17446/Understanding%20Metadata.pdf>, which groups metadata as

- descriptive
- structure

- admin (but this is really 3 subgroups)

For instance, we might have a metadata INI block along the lines of

```

1  [description]  # for finding or understanding a resource
2      title
3      authors
4      subject
5      publication_date
6      creation_date
7  [preservation] # (admin) long-term management of files
8      checksum
9      ???
10 [technical]    # (admin) for decoding and "rendering" a file
11     file type
12     file size
13     creation date/time
14     details
15 [rights]       # (admin) intellectual property rights and access control
16     copyright
17     license
18     rights_holder
19     accessible_to
20 [structural]   #

```

Alternatively the Dublin core <https://www.ietf.org/rfc/rfc5013.txt>

```

1  title
2  creator
3  subject
4  description
5  publisher
6  contributor
7  date
8  type
9  format
10 identifier
11 source
12 language
13 coverage
14 rights

```

Internet Anonymous FTP Archives (IAFA) for Internet data, <https://www.w3.org/Conferences/WWW4/Papers/52/> also defines certain fields.

Our metadata allows comments to add description, but comments are not intended to be machine read. A metadata schema can provide:

- additional descriptions;
- a list of required fields (for a particular type of data);
- information about how to further parse a particular field (it's type);
- restrictions or checks on the provided metadata.

In keeping with the core philosophy of INC, our mini-schemas:

1. a very lightweight

2. reuse the INI syntax so that an INC reader can also read the mini-schemas.

Also, schema use is entirely optional, so having this mechanism presents no barrier to entry.

The Data Part

The data part contains a tabular-delimiter file, most commonly a CSV file. It's formatting details can be contained in the metadata.

However, in the spirit of sensible defaults, we take the default that this would be a file with as defined in RFC 4180 <https://www.ietf.org/rfc/rfc4180.txt> with

- 1 header line,
- comma delimiters between fields and end-of-line characters delimiting records, technically a CRLF (a carriage-return and line-feed),
- comments preceeded by a # symbol,
- fields may be enclosed in double quote marks, notably to escape commas or end-of-line characters to be held inside a string value,
- a quote mark can be included in a field value by using a pair of quotes ""
- the last line may or may not end with a CRLF symbol.

All of these are changeable with metadata from the [structure] section. We model the fields of the [structure] section on the keyword arguments to Julia's CSV.jl package read and write commands.

Example

Output from a simulator of the M/M/1 queue. The simulation starts with the queue empty and generates an arrival at time zero to get the ball rolling. The state of the system is recorded at event epochs.

Note that the indentation used here is purely decorative (to enhance readability) and is not needed.

```
1  ---
2  filename    = "simulation_1.inc"
3  created_by  = "mm1_simulator.jl"
4  created_on  = "2023/01/10 00:03:24"
5  source      = "simulation"
6  [structure]      # data format details, such as units
7  format = "CSV"
8  delimiter = ","
9  header_lines = 1
10 comment_character = "#"
11 quote_string = ""
12 checksum = 1332351132
13 rows = 4
14 columns = 3
15 col_labels = ["Time (s)", "# of customers", "Event type"]
16 col_types = ["FLOAT", "INT", "STRING"]
17 col_description = ["timestamp (seconds from the start of the simulation)",
18                   "number of customers in the system",
19                   "event indicator (what event triggered the record)"]
20 [parameters] # parameters of code generating the data
21 seed = 100 # pseudo-random number seed used in this realisation
22 λ = 1.5 # arrival rate in customers per second
23 μ = 2.3 # service rate customers per second
```

```
24 |     strange_parameter = "---"
25 | ---
26 | time,customers,event
27 | 0.000,1,Arrival
28 | 0.203,2,Arrival
29 | 0.431,1,Departure
30 | 1.222,2,Arrival
31 | # simulation finished here
```

Why not X?

There is a tendency, in creating formats, that people want to make them extensible and all-singing and dancing. They don't want to specify the use case clearly – they want a kitchen sink.

In creating something that can do everything, it does nothing well.

The philosophy here is to use the minimum that provides what we need for a clear, and common use case.

That should be enough, but it might pay to address potential competitors explicitly, which we do below.

Why not just CSV?

[CSV is incredibly successful, by itself.](#)

We can include comments in CSV (or at least many CSV libraries admit the possibility that a CSV file contains comments). Metadata can be stored in comments, and so included, so why don't we just use CSV?

However

1. CSV is not a standardised file format, but rather a class of formats with common attributes. The goal here is to place a rigorous framework around the CSV component such that MR is facilitated (a machine could, for instance, determine the delimiter type before trying to read the file), and
2. Standardise how the metadata is contained to help make the file more portable.

Why not CSVY (YAML-CSV)

This proposal is in part inspired by [CSVY](#), and other tools that use YAML frontmatter, e.g., [Jekyll](#) but the CSVY format has problems, primarily because YAML has problems.

These largely come down to the fact that YAML is trying to do too much. It is seeking to be yet-another data-serialisation package. We don't need that for most metadata.

It provides feature (e.g., hierarchy, line continuation, data type inference, ...) that seem superficially appealing, but which create well documented problems (see), for instance:

- Implicit typing can cause problems, e.g., the Norway problem.
- Duplicate keys can cause ambiguity.
- Flow style allows/encourages JSON-like code, and hence complexifies.
- Node anchors allow deduplication of code, which might save space (arguably in this use case) but at the cost of HR.
- It allows concepts (e.g., explicit tags) that make the results incomprehensible to non-programmers.
- YAML allows Python code to be executed, [raising serious security issues](#).

The response has been “Strict” YAML, which is (almost) a subset of YAML that seeks to remove the most problematic features. But if you are removing those features, why not go one step further through the thought process and move to the most simple approach that works for most cases.

Here we want to re-iterate that most metadata can be expressed as simple name-value pairs with little hierarchy (we found one level can be useful for organisation).

Why not use Strict YAML in a CSVY file

Strict YAML addresses many of the issues in YAML, but retains the underlying syntax, and most of its advantages.

It is an enticing alternative.

However, its main advantage over our approach is the ability to represent more levels of hierarchy, and we argue there that (i) this is not needed for this type of data, and (ii) every additional level of hierarchy adds a layer of confusion.

Incidentally, the **Strict YAML** page lists reasons why not to use INI. Hierarchy is one, but the other is the variations in the INI format, which we aim to eliminate when it is used for our purpose.

Why not use TOML

TOML is in some sense a child of INI. It inherits similar syntax and ideas.

TOML also has many problems, argued [here](#) and [here](#) and [here](#):

- It is verbose, and while we are not too concerned about saving space, verbosity is potentially a problem for HR.
- TOML's hierarchy is difficult to infer from its syntax, and this would also be true in INI except we limit hierarchy to one level.
- TOML also has too many features: e.g., data and time parsing.

You might say, why not use TOML, but restrict it in some manner. However, as soon as you call it TOML, you run into people who will want to use some feature from TOML. Allow variations, and they will be used. So we just don't allow it from the start.

More deeply TOML's rules cannot be inferred from the language – some parts of a file require a detailed understanding of TOML to be readable, so the full language doesn't reach our standard for HR.

However, one goal here is that our INI format be a strict subset of TOML, so that TOML libraries could (in principle) be used to read the frontmatter/metadata of the file.

More detailed critique of TOML (vs INI) can be found at <https://github.com/madmurphy/libconfini/wiki/An-INI-critique-of-TOML>. Here we list a few issues considered, but not problematic for our application, which is not configuration files:

- case sensitivity is fine (in some places)
- minor verbosity issue (quotes, ...)
- square bracket arrays, where square brackets are used for section names
- empty strings for section names

Others that are problems, but minor exclusions

- Empty strings for names
- syntax for dates
- quoting rules

1. In INI files everything is a castable string. Application always receives a string and it can decide.
- 2.

Why not CSVW

CSVW – CSV on the Web – is a standard to add metadata to CSV files.

It shares some of the goals and motivations of this project. It includes standards to disambiguate column types, declare names of columns and so on. It talks of “machine readable footnotes.” which is very much in the spirit of what we aim to do here.

However, as with most similar projects its scope is wider. It is focussed on tabular data, and provides a lightweight means to add metadata. However, the types of metadata encompassed include the ability to relate tables together to build databases of tables.

However, the most fundamental objection is that CSVW is intended to create a separate metadata file (for one or more data files). Separate files have some advantages, i.e., they can avoid repetition (in describing multiple data files), and using a different format for the metadata is conceptually easier for some people to grasp if the files are separate), but we argue here that there is a need to include at least some metadata IN the file to which it refers.

Why not XML or JSON?

The issues of XML are well documented, but fundamentally it is trying to solve a different problem. Its problem, and that of JSON is to serialise potentially complex (not table-based) data. Both XML and JSON also allow arbitrary metadata as an intrinsic, but do not do so in an easy-to-get-right manner because they need to allow for every case.

The difficulties with understanding XML and JSON often results in them often being used badly.

There are many other data formats in this category. They do much more than INC, but at a cost that we don't want to pay for the common majority of files.

Why not some kind of database?

A common strategy for metadata is to hold it in a database. We have already argued extensively for the need to include metadata in the data file, not separate, but there is another reason that a database is not what we need here. The primary use for databases arises in transactional data, where properties such as atomicity and consistency are crucial. The common case for our data is a static set of files. Databases are overkill in those settings, but bring in a vast amount of overhead.

Why '=' and not a colon, or some other symbol?

Most INI files use the = sign, as we are here. We could have used '+' but that would have been IMHO less readable. Similarly, I personally find colons harder to read. Moreover, the colon (and many other symbols) seems more likely to be needed in descriptions and similar items, and so reserving it seems unproductive.

An alternative, a multicharacter symbol like '<-' (sometimes used for assignment in programming languages) uses two characters instead of one for arguable benefit.

Why not use Frictionless Data?

There is a group of people putting forward software and standards specifically labelled **frictionless data**, which is the goal in our work as well, and the philosophy of the group is very much what we are going for. We want many of the same things (simplicity; HR and MR; reuse of standards and formats; and portability, for instance).

But the frictionless data standard also does much more than we do here. For instance, it can act as a container for a package of files. And it is aimed at a much wider variety of data from tabular data (that we deal with here) to geospatial data and so on. So another fundamental goal of frictionless data is to create extensibility.

As soon as you open the extensibility COW, you create a much more complex problem.

Why not Excel (or spreadsheets in general)

This is an interesting question. Many people use a spreadsheet exactly for this purpose, i.e., to have a simple way to include metadata with their tabular data. It can be done in a separate sheet of a few cells above the main data. But there are problems:

- A spreadsheet does much more than store data – it can store, for instance calculations.
- It links formatting to the data, which can result in overloaded fields (for instance, users sometimes use colour to provide meaning).
- Despite massively improved portability of spreadsheet file formats, there is still a fundamental issue in relying on a format that is tied to software, not defined independently.
- The editability of a spreadsheet (most people access them through a read/write application) can easily lead to data rot.
- Dates are often represented in a non-HF manner (superficially they are easy, but that ease is mediated through the spreadsheet interface).
- Spreadsheets enable “non-rectangular” data.
- Most importantly, such metadata is almost always added in a *ad hoc* fashion.

All of these can be addressed through careful processing but almost never are. The general “niceness” of most spreadsheet applications encourages the use of features that are counter-productive for storing data, and for storing its metadata in a structured and portable manner.

These issues (and one approach to address them through process) are explained eloquently in Bromana and Woo (2018). They note

- error rates up to 88%;
- horror stories: ...

Spreadsheets are also often used for data entry, but the data entry and data analysis and storage have different requirements, and are rarely separated.

Other concerns

One general concern might be that when a large number of files are involved, this approach to metadata wastes much space. Repeating information in each file of a dataset is indeed a compromise between space and ease, but we would point out several remediations:

- Disk space is no longer so tight that a few extra bytes in each file need cause concern.
- In memory storage of the files can drop the metadata that isn't needed.
- Most modern compression tools, when applied to a group of files with repeated strings will do a very good job of reducing the overhead of such repetition.

More generally, we acknowledge that we violate the DRY (Don't Repeat Yourself) principal here. Each file of a similar group of files might have fields that are identical. A standard CS approach to such a problem would be to add a layer of abstraction/redirection and hence reuse. For example, in configuration files, it is common to allow an `include` statement that pulls in another file, often through simple text replacement. Such includes could replace common materials such as are repeated across a group of files. However, we have resisted the temptation to do so principally for the reasons for including metadata in the file in the first place.

Another concern is that updating our file-included metadata is harder than updating a central repository, but it is envisaged that these types of files are commonly generated and managed through software, and updating a group of files is then no harder than a central point in such cases.

Finally, metadata included in a file is not invulnerable, particularly to malicious code. That is not the goal here. The goal is to solve 90% of problems with 10% of the effort. If your metadata is crucial, e.g., in a forensic application, then this is not the solution. But your solution will be much more complex and hard to use.

Implementation

Part of the goal here is to develop a standard for which implementation is VERY easy, enabling the rapid spread of this format to many contexts.

Note that there are some aspects of implementations which should be clear, but perhaps need to be restated:

- Comments are for humans. If you read a file containing comments (either in the INI or CSV part), those comments are not read into memory. And so if the file is then rewritten, the comments will be lost.
- Order (within a section) of the INI properties is not preserved when a file is rewritten.
- Some cleaning (e.g., removal of whitespace) will occur when a file is rewritten.
- Some INI properties may change on rewriting, e.g., the “written_on” date.
- Some additional quoting or escapes may be added.
- Default values for certain (unspecified) properties will be added.

All of these features mean that a file that is read and then rewritten will differ from the original in a potentially large number of ways. Hence, **we recommend you do not rewrite over the original file**, unless the intention is specifically to clean it up.

However, in other respects we aim to preserve the file as much as possible, e.g., by keeping the same CSV dialect.

General Implementation Principles

Principles (the ones not already stated):

0. **Don't try to read the whole file at once:** the data might be big, but the metadata is small. Including all metadata in frontmatter allows this to be read line-by-line without trying to ingest the entire file at once. Data-size indications in the meta-data might then be used to inform the read strategy for the data (e.g., by choosing to use memory-mapped I/O if the file is too large for system memory).
1. **Throw an error, not a warning:** warnings are too often ignored. We want to prevent metadata loss through incorrect structure, so preventing errors is more important than being nice to users.
2. **Errors need to be meaningful:** error messages should be HF, and we should have an agreed set of messages for all code bases.
3. **The user should have few options:** Don't ask the user for many inputs, e.g., very few flags. The user might like control, but control leads to misuse of the control mechanism to subvert the protocol.
4. **A simple CSV file should be readable without a header:** to make this backward compatible with plain-old-CSV.
5. **The data structure we read metadata into should be the same (as much as is possible) for all implementation.** The data structure at present is a 2D dictionary of strings, e.g., an associative array with two indices (section, name), e.g., in Julia

```
1 | Dict{String, Dict{String, String}} }
```

that we access using syntax similar to

```
1 | metadata[section_name][property_name]
```

All names (for properties and sections) are strings as are all values.

Uniformity of this representation across code-bases enhances usability and portability.

6. **Optionally, the implementation should provide a pretty-printer for metadata**, which shouldn't be too much harder than providing a write routine.

Related

CSV is really a conceptual framework, rather than a standard. Within it there are many dialects (Shafranovich, 2005). There is some desire to standardise these, but the proliferation of variants makes that problematic. Rather than force a standardisation, one opportunity here is the standardise a framework around delimited tabular data files (in their metadata) to more precisely specify the dialect being used.

Lightweight processes are often used, but not often formally studied. One exception is Spinellis and Guruprasad (1997), who explain much of what we assert here:

- reuse of existing tools is facilitated;
- you get to use features that target the specific use cases (the tabular data and metadata components of a dataset are very different in most cases);
- such approaches facilitate simplicity, and hence ease of use; and
- they specifically comment on the use of self-documentation (for code) as a use case, which is very similar to the use of metadata in this context.

Conclusion

This memo provides a description of a simple file format that is intended for storing tabular data along with its metadata.

The goal here is simplicity, but also a degree of rigour.

The length of this document (as compared to many similar efforts) speaks to the detail required for even a simple file format to be specified completely. But also, this is not a document just documenting the format itself, but also the decision process used to form it.

References

- Karl W. Broman & Kara H. Woo (2018) Data Organization in Spreadsheets, *The American Statistician*, 72:1, 2-10, DOI: 10.1080/00031305.2017.1375989
- Kathleen Burnett, Kwong Bor Ng, and Soyeon Park (1999), "A Comparison of the Two Traditions of Metadata Development," *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*. 50(13):1209–1217.
- Rachel Heery (2007), "Review of metadata formats," *Program: electronic library and information systems*, vol. 30, no. 4, October 1996, pp. 345-373, <https://www.emerald.com/insight/content/doi/10.1108/eb047236/full/pdf?title=review-of-metadata-formats>
- Y. Shafranovich (2005), "Common Format and MIME Type for Comma-Separated Values (CSV) Files," IETF RFC 4180, <https://www.ietf.org/rfc/rfc4180.txt>.
- D. Spinellis & V. Guruprasad (1997), "Lightweight Languages as Software Engineering Tools," *Proceedings of the Conference on Domain-Specific Languages* Santa Barbara, California, https://www.usenix.org/legacy/publications/library/proceedings/dsl97/full_papers/spinellis/spinellis.pdf

others

- Sharing biological data: why, when, and how Samantha L. Wilson¹, Gregory P. Way², Wout Bittremieux^{3,4}, Jean-Paul Armache⁵, Melissa A. Haendel⁶ and Michael M. Hoffman <https://carpenter-singh-lab.broadinstitute.org/files/anne/files/FEBS%20Letters%20-%202021%20-%20Wilson%20-%20Sharing%20biological%20data%20-%20why%20-%20when%20-%20and%20how.pdf>
- @misc{<https://doi.org/10.48550/arxiv.2106.15005>, doi = {10.48550/ARXIV.2106.15005}, url = {<https://arxiv.org/abs/2106.15005>}, author = {Bartram, Lyn and Correll, Michael and Tory, Melanie}, keywords = {Human-Computer Interaction (cs.HC), FOS: Computer and information sciences, FOS: Computer and information sciences}, title = {Untidy Data: The Unreasonable Effectiveness of Tables}, publisher = {arXiv}, year = {2021}, copyright = {Creative Commons Attribution 4.0 International} }
- <https://towardsdatascience.com/everything-you-didnt-want-to-have-to-know-about-csv-665d0755e28>

