The latex2pydata package

Geoffrey M. Poore
gpoore@gmail.com
github.com/gpoore/latex2pydata/tree/main/latex

v0.2.0 from 2024/05/16

Abstract

latex2pydata is a LaTeX package for writing data to file using Python literal syntax. The data may then be loaded safely in Python using the `ast.literal_eval()` function or the latex2pydata Python package.

The original development of this package was funded by a TeX Development Fund grant from the TeX Users Group. latex2pydata is part of the 2023 grant for improvements to the minted package.

Contents

1 Introduction

The latex2pydata package is designed for passing data from LaTeX into Python. It writes data to file using Python literal syntax. The data may then be loaded safely in Python using the `ast.literal_eval()` function or the latex2pydata Python package.

The data that latex2pydata writes to file can take two forms. The top-level data structure can be configured as a Python dict. This is appropriate for representing a single LaTeX command or environment. The top-level data structure can also be configured as a list of dicts. This is useful for representing a sequence of LaTeX commands or environments. In both cases, all keys and values within dicts are written to file as Python string literals. Thus, the overall data is `dict[str, str]` or `list[dict[str, str]]`. This does not limit the data types that can be passed from LaTeX to Python, however. When data is loaded, the included schema functionality makes it possible to convert string values into other Python data types such as dicts, lists, sets, bools, and numbers.

The data is suitable for direct loading in Python with `ast.literal_eval()`. It is also possible to load data with the latex2pydata Python package, which serves as a wrapper for `ast.literal_eval()`. The Python package requires all keys to match the regex `[A-Za-z_][0-9A-Za-z_]*`. Periods in keys are interpreted as key paths and indicate sub-dicts. For example, the key path `main.sub` represents a key `main` in the main dict that maps to a sub-dict containing a key `sub`. This makes it convenient to represent nested dicts.

latex2pydata optionally supports writing metadata to file, including basic schema definitions for values. When the latex2pydata Python package loads data with a schema definition for a given value, the value is initially loaded as a string, which is the verbatim text sent from LaTeX. Then this string is evaluated with `ast.literal_eval()`. An error is raised if this process does not result in an object with the data type specified in the schema.

2 Example

```
\pydatasetfilename{\jobname.pydata}
\pydatawritedictopen
\pydatawritekeyvalue{key}{value with "quote" and \backslash\ ...}
\pydatawritedictclose
\pydataclosefilename{\jobname.pydata}
\VerbatimInput{\jobname.pydata}
```

```
{
"key": "value with \"quote\" and \\backslash\\ ...",
}
```

3 Design considerations

latex2pydata is intended for use with Python. Python literal syntax was chosen instead of JSON or another data format because it provides simpler compatibility with LaTeX.

- It must be possible to serialize the contents of a LaTeX environment verbatim. Python literal syntax supports multi-line string literals, so this is straightforward:

write an opening multi-line string delimiter to file, write the environment contents a line at a time (backslash-escaping any delimiter characters), and finally write a closing multi-line string delimiter. Meanwhile, JSON requires that all literal newlines in strings be replaced with "`\n`". The naive LaTeX implementation of this would be to accumulate the entire environment contents verbatim within a single macro and then perform newline substitutions. For long environment contents, this can lead to buffer memory errors (LaTeX's `buf_size`). It should be possible to avoid this, but only with more creative algorithms that bring additional complexity.

- Python literal syntax only requires that the backslash plus the string delimiter be escaped within strings. JSON has the additional requirement that command characters be escaped.

latex2pydata is designed for use with Python and there are no plans to add additional data formats for use with other languages. Choosing Python literal syntax does make latex2pydata less compatible with other programming languages than JSON or some other formats would be. However, the only data structures used are `dict[str, str]` and `list[dict[str, str]]`. It should be straightforward to implement a parser for this subset of Python literal syntax in other languages.

Data structures are limited to `dict[str, str]` and `list[dict[str, str]]` because the objective is to minimize the potential for errors during serialization and deserialization. These are simple enough data structures that basic checking for incomplete or malformed data is possible on the LaTeX side during writing or buffering. More complex data types, such as floating point numbers or deeply nested dicts, would be difficult to validate on the LaTeX side, so invalid values would tend to result in parse errors during deserialization in Python. The current approach still allows for a broad variety of data types via a schema, with the advantage that it can be easier to give useful error messages during schema validation than during deserialization parsing.

4 Usage

Load the package as usual: `\usepackage{latex2pydata}`. There are no package options.

4.1 Errors

Most LaTeX packages handle errors based on the `-interaction` and `-halt-on-error` command-line options, plus `\interactionmode` and associated macros. With the common `-interaction=nonstopmode`, LaTeX will continue after most errors except some related to missing external files.

latex2pydata is designed to force LaTeX to exit immediately after any latex2pydata errors. latex2pydata is designed for serializing data to file, typically so that an external program (restricted or unrestricted shell escape, or otherwise) can process the data and potentially generate output intended for LaTeX. Data that is known to be incomplete or malformed should not be passed to external programs, particularly via shell escape.

When latex2pydata forces LaTeX to exit immediately, there will typically be a message similar to "`! Emergency stop [...] cannot \read from terminal in nonstop modes`." This is due to the mechanism that latex2pydata uses to force LaTeX to exit. To debug, go back further up the log to find the latex2pydata error message that caused exiting.

4.2 File handling

All file handling commands operate globally (`\global`, `\gdef`, etc.).

`\pydatasetfilehandle` `{⟨filehandle⟩}`

Configure writing to file using an existing file handle created with `\newwrite`. This allows manual management of the file handle. For example:

```
\newwrite\testdata
\immediate\openout\testdata=\jobname.pydata\relax
\pydatasetfilehandle{\testdata}
...
\pydatareleasefilehandle{\testdata}
\immediate\closeout\testdata
```

To switch from one file handle to another, simply use `\pydatasetfilehandle` with the new file handle. When the file handle is no longer in use, `\pydatareleasefilehandle` is recommended (but not required) to remove references to the file handle and perform basic checking for incomplete or malformed data written to file.

`\pydatasetfilehandle` sets the file handle globally.

`\pydatareleasefilehandle` `{⟨filehandle⟩}`

When a file handle is no longer needed, remove references to it. Also perform basic checking for incomplete or malformed data written to file.

This should only be used once per opened file, after all data has been written, just before the file is closed. It is not needed when switching from one file handle to another when both files remain open; in that case, only `\pydatasetfilehandle` is needed. If `\pydatareleasefilehandle` is used before all data is written, or it is used multiple times while writing to the same file, then it is no longer possible to detect incomplete or malformed data.

`\pydatasetfilename` `{⟨filename⟩}`

Configure a file for writing based on filename, opening the file if necessary. For example:

```
\pydatasetfilename{\jobname.pydata}
```

This is not designed for manual management of the file handle. The file does not have to be closed manually since this will happen automatically at the end of the document. However, using `\pydataclosefilename{⟨filename⟩}` is recommended since it closes the file immediately and also performs basic checking for incomplete or malformed data written to file.

To switch from one file to another, simply use `\pydatasetfilename` with the new filename. When the file is no longer in use, `\pydataclosefilename` is recommended.

`\pydatasetfilename` sets the filename globally.

`\pydataclosefilename` `{⟨filename⟩}`

Close a file previously opened with `\pydatasetfilename`. Also perform basic checking for incomplete or malformed data written to file.

4.3 Metadata

latex2pydata optionally supports writing metadata to file, including basic schema definitions for values. When data is loaded with the latex2pydata Python package, the schema is used to perform type conversion and type checking. When a schema definition exists for a given value, the value is initially loaded as a string, and then (for non-string data types) it is evaluated with `ast.literal_eval()`. An error is raised if this process does not result in an object with the data type specified in the schema.

`\pydatasetschemamissing {⟨missing behavior⟩}`

This determines how the schema is processed when the schema is missing definitions for one or more key-value pairs. Options for ⟨*missing behavior*⟩:

- `error` (default): If a schema is defined then a complete schema is required. That is, a schema definition must exist for all key-value pairs or an error is raised.
- `rawstr`: The schema is enforced for all key-value pairs for which it is defined, and any other key-value pairs are kept with string values. These string values are the raw verbatim text passed from LaTeX.
- `evalany`: The schema is enforced for all key-value pairs for which it is defined, and any other key-value pairs have the value evaluated with `ast.literal_eval()`, with all value data types being permitted. Because all values without a schema definition are evaluated, any string values without a schema definition must be quoted and escaped as strings on the LaTeX side.

`\pydatasetschemakeytype {⟨key⟩}{⟨value type⟩}`

Define a key's schema. For example, `\pydatasetschemakeytype{key}{int}`.

⟨*value type*⟩ should be a standard Python type annotation, such as `list[int]` or `dict[str, float]`. See the latex2pydata Python package documentation for information about value data types that are currently supported.

`\pydataclearschema`

Delete the existing schema. If the schema is not deleted, it can be reused across multiple output files.

`\pydatawritemeta`

Write metadata, including schema, to a file previously configured with `\pydatasetfilename` or `\pydatasetfilehandle`. Metadata must always be the first thing written to file, before any data.

`\pydataclearmeta`

Clear all metadata. This includes deleting the schema and resetting schema missing behavior to the default.

4.4 Writing list and dict delimiters

The overall data structure, before any schema is applied by the latex2pydata Python package, can be either `list[dict[str, str]]` or `dict[str, str]`. This determines which data collection delimiters are needed.

Delimiters are written to the file previously configured via `\pydatasetfilehandle` or `\pydatasetfilename`.

`\pydatawritedictopen`

Write an opening dict delimiter `{` to file.

`\pydatawritedictclose`

Write a closing dict delimiter `}` to file.

`\pydatawritelistopen`

Write an opening list delimiter `[` to file.

`\pydatawritelistclose`

Write a closing list delimiter `]` to file.

4.5 Writing keys and values

All keys must be single-line strings of text without a newline. Both single-line and multi-line values are supported. Keys and values are written to the file previously configured via `\pydatasetfilehandle` or `\pydatasetfilename`.

Commands for writing keys and values may read these keys and values in one of two ways.

- Commands whose names contain `key` or `value` read these arguments verbatim, as described below.
- Commands whose names contain `edefkey` or `edefvalue` read these arguments normally, then expand the arguments via `\edef`, and finally interpret the result as verbatim text.

The latex2pydata commands that read keys and values verbatim have some limitations. When these commands are used inside other commands, they use macros from fvextra to attempt to interpret their arguments as verbatim. However, there are limitations in this case because the arguments are already tokenized:

- `#` and `%` cannot be used.
- Curly braces are only allowed in pairs.
- Multiple adjacent spaces will be collapsed into a single space.
- Be careful with backslashes. A backslash that is followed by one or more ASCII letters will cause a following space to be lost, if the space is not immediately followed by an ASCII letter.
- A single `^` is fine, but `^^` will serve as an escape sequence for an ASCII command character.

When the latex2pydata commands are used inside other commands that pass their arguments to the latex2pydata commands, it may be best to avoid these limitations by defining the other commands to read their arguments verbatim. Consider using the xparse package. It is also possible to use `\FVExtraReadVArg` from fvextra; for an example, see the implementation of `\pydatawritekey`.

Because the latex2pydata commands treat keys and values as verbatim, any desired macro expansion must be performed before passing the keys and values to the latex2pydata commands.

`\pydatawritekey {⟨key⟩}`

Write a key to file.

`\pydatawritevalue {⟨value⟩}`

Write a single-line value to file.

`\pydatawritekeyvalue {⟨key⟩}{⟨value⟩}`

Write a key and a single-line value to file simultaneously.

`\pydatawritekeyedefvalue {⟨key⟩}{⟨value⟩}`

Write a key and a single-line value to file simultaneously. The value is expanded via `\edef` before being interpreted as verbatim text and then written.

`pydatawritemlvalue` (*env.*)

Write a multi-line value to file.

This environment uses fvextra and fancyvrb internally to capture the environment contents verbatim. If a new environment is defined as a wrapper for `pydatawritemlvalue`, then `\VerbatimEnvironment` must be used at the beginning of the new environment definition. This configures fancyvrb to find the end of the new environment correctly.

`\pydatawritemlvaluestart`

`\pydatawritemlvalueline {⟨line⟩}`

`\pydatawritemlvalueend`

These commands allow writing a multi-line value to file one line at a time. ⟨*line*⟩ is interpreted verbatim.

4.6 Buffer

Key-value data can be written to file once a dict is opened with `\pydatawritedictopen`. It is also possible to accumulate key-value data in a "buffer." This is convenient when the data serves as input to an external program that generates cached content. Buffered data can be hashed in memory without being written to file, so the existence of cached content can be checked efficiently.

A buffer consists of a sequence of macros of the form `\⟨buffername⟩line⟨n⟩`, where each line of data corresponds to a macro and ⟨*n*⟩ is an integer greater than or equal to one (one-based indexing). The length of the buffer is stored in the counter `⟨buffername⟩length`. Buffers are limited to containing comma-separated key-value data, without any opening or closing dict delimiters `{}`.

All buffer commands that set the buffer or modify the buffer operate globally (`\global`, `\gdef`, etc.).

4.6.1 Creating and deleting buffers

`\pydatasetbuffername {⟨buffername⟩}`

Initialize a new buffer if ⟨*buffername*⟩ has not been used previously, and configure all buffer operations to use ⟨*buffername*⟩.

⟨*buffername*⟩ is used as a base name for creating the buffer line macros of the form `\⟨buffername⟩line⟨n⟩` and the buffer length counter `⟨buffername⟩length`.

`\pydataclearbuffername {⟨buffername⟩}`

Delete the specified buffer. `\let` all line macros `\⟨buffername⟩line⟨n⟩` to an undefined macro, and set the length counter `⟨buffername⟩length` to zero.

4.6.2 Special buffer operations

`\pydatabuffermdfivesum`

Calculate the MD5 hash of the current buffer, using `\pdf@mdfivesum` from pdftexcmds. This is fully expandable. For example:

```
\edef\hash{\pydatabuffermdfivesum}
```

`\pydatawritebuffer`

Write the current buffer to the file previously configured via `\pydatasetfilename` or `\pydatasetfilehandle`.

Writing the buffer does not modify the buffer in any way or delete it. To delete the buffer after writing, use `\pydataclearbuffername`.

4.6.3 Buffering keys and values

All keys must be single-line strings of text without a newline. Both single-line and multi-line values are supported. Keys and values are appended to the buffer previously configured via `\pydatasetbuffername`.

The latex2pydata commands read keys and values verbatim. Like the commands for writing keys and values, the commands for buffering keys and values have limitations when used inside other commands.

`\pydatabufferkey {⟨key⟩}`

Append a key to the buffer.

`\pydatabuffervalue {⟨value⟩}`

Append a single-line value to the buffer.

`\pydatabufferkeyvalue {⟨key⟩}{⟨value⟩}`

Append a key and a single-line value to the buffer simultaneously.

`\pydatabufferkeyedefvalue {⟨key⟩}{⟨value⟩}`

Append a key and a single-line value to the buffer simultaneously. The value is expanded via `\edef` before being interpreted as verbatim text and then buffered.

`pydatabuffermlvalue` (*env.*)

Append a multi-line value to the buffer.

This environment uses fvextra and fancyvrb internally to capture the environment contents verbatim. If a new environment is defined as a wrapper for `pydatabuffermlvalue`, then `\VerbatimEnvironment` must be used at the beginning of the new environment definition. This configures fancyvrb to find the end of the new environment correctly.

`\pydatabuffermlvaluestart`

`\pydatabuffermlvalueline` `{⟨line⟩}`

`\pydatabuffermlvalueend`

These commands allow buffering a multi-line value one line at a time. `⟨line⟩` is interpreted verbatim.

5 Implementation

5.1 Exception handling

`\pydata@error` Shortcut for error message. The `\batchmode\read -1 to \pydata@exitnow` forces an immediate exit with "`! Emergency stop [...] cannot \read from terminal in nonstop modes.`" Due to the potentially critical nature of written or buffered data, any errors in assembling the data should be treated as fatal.

```
\def\pydata@error#1{%
  \PackageError{latex2pydata}{#1}{}%
  \batchmode\read -1 to \pydata@exitnow}
```

`\pydata@warning` Shortcut for warning message.

```
\def\pydata@warning#1{%
  \PackageWarning{latex2pydata}{#1}}
```

5.2 Required packages

```
\RequirePackage{etoolbox}
\RequirePackage{fvextra}
\IfPackageAtLeastTF{fvextra}{2024/05/16}%
 {}{\pydata@error{package fvextra is outdated; upgrade to the latest version}}
\RequirePackage{pdftexcmds}
```

5.3 Util

`\pydata@empty` Empty macro.

```
\def\pydata@empty{}
```

`\pydata@newglobalbool` `\pydata@provideglobalbool` Variants of etoolbox's `\newbool` and `\providebool` that create bools whose state is always global. When these global bools are used with `\setbool`, `\booltrue`, or `\boolfalse`, the global state is updated regardless of whether the command is prefixed with `\global`. These use a global variant of LaTeX's `\newif` internally.

```
\def\pydata@gnewif#1{%
  \count@\escapechar
  \escapechar\m@ne
  \global\let#1\iffalse
  \pydata@gif#1\iftrue
  \pydata@gif#1\iffalse
  \escapechar\count@}
\def\pydata@gif#1#2{%
  \expandafter\gdef\csname
    \expandafter\@gobbletwo\string#1\expandafter\@gobbletwo\string#2\endcsname
     {\global\let#1#2}}
\newrobustcmd*{\pydata@newglobalbool}[1]{%
  \begingroup
  \let\newif\pydata@gnewif
  \newbool{#1}%
  \endgroup}
\newrobustcmd*{\pydata@provideglobalbool}[1]{%
  \begingroup
  \let\newif\pydata@gnewif
  \providebool{#1}%
  \endgroup}
```

5.4 State

Track state of writing data and of buffering data. Notice that bools for tracking state are a special, custom variant that is always global.

`pydata@canwrite` Whether data can be written. False if a file handle has not been set or if the top-level data structure has been closed.

```
\pydata@newglobalbool{pydata@canwrite}
```

`pydata@hasmeta` Whether metadata was written. Metadata is a `dict[str, str | dict[str, str]]`.

```
\pydata@newglobalbool{pydata@hasmeta}
```

`pydata@topexists` Whether the top-level data structure has been configured. The top-level data structure can be a list or a dict. The overall data structure must be either `dict[str, str]` or `list[dict[str, str]]`.

```
\pydata@newglobalbool{pydata@topexists}
```

`pydata@topislist` Whether the top-level data structure is a list.

```
\pydata@newglobalbool{pydata@topislist}
```

`pydata@indict` Whether a dict has been opened.

```
\pydata@newglobalbool{pydata@indict}
```

`pydata@haskey` Whether a key has been written (waiting for a value).

```
\pydata@newglobalbool{pydata@haskey}
```

`\pydata@fhstartstate`
`\pydata@fhstopstate`
`\pydata@fhresetstate`
Start and stop state tracking for a file handle (`\newwrite`), or reset state after writing is complete. Each file handle has its own set of state bools of the form `pydata@`⟨*boolname*⟩`@`⟨*fh*⟩. When a file handle is in use, the values of these bools are copied into the `pydata@`⟨*boolname*⟩ bools; when the file handle is no longer in use, `pydata@`⟨*boolname*⟩ values are copied back into `pydata@`⟨*boolname*⟩`@`⟨*fh*⟩.

```
\def\pydata@fhstartstate#1{%
```

```
  \expandafter\pydata@fhstartstate@i\expandafter{\number#1}}
\newbool{pydata@fhnewstate}
\def\pydata@fhstartstate@i#1{%
  \ifcsname ifpydata@canwrite@#1\endcsname
    \boolfalse{pydata@fhnewstate}%
  \else
    \booltrue{pydata@fhnewstate}%
  \fi
  \def\do##1{%
    \pydata@provideglobalbool{pydata@##1@#1}%
    \ifbool{pydata@##1@#1}{\booltrue{pydata@##1}}{\boolfalse{pydata@##1}}}%
  \docsvlist{canwrite, hasmeta, topexists, topislist, indict, haskey}%
  \ifbool{pydata@fhnewstate}%
   {\booltrue{pydata@canwrite}}{}%
  \ifbool{pydata@fhisreleased@#1}%
   {\boolfalse{pydata@fhisreleased@#1}\booltrue{pydata@canwrite}}{}}
\def\pydata@fhstopstate#1{%
  \expandafter\pydata@fhstopstate@i\expandafter{\number#1}}
\def\pydata@fhstopstate@i#1{%
  \ifcsname ifpydata@canwrite@#1\endcsname
    \def\do##1{%
      \ifbool{pydata@##1}{\booltrue{pydata@##1@#1}}{\boolfalse{pydata@##1@#1}}%
      \boolfalse{pydata@##1}}%
    \docsvlist{canwrite, hasmeta, topexists, topislist, indict, haskey}%
  \fi}
\def\pydata@fhresetstate#1{%
  \expandafter\pydata@fhresetstate@i\expandafter{\number#1}}
\def\pydata@fhresetstate@i#1{%
  \def\do##1{%
    \boolfalse{pydata@##1@#1}}%
  \docsvlist{canwrite, hasmeta, topexists, topislist, indict, haskey}}
```

`pydata@bufferhaskey` Whether a key has been added to the buffer (waiting for a value).

If multiple buffers are in use, all buffers use the same `pydata@bufferhaskey`. Inconsistent state is avoided by requiring that `\pydatasetbuffername` can only be invoked when `pydata@bufferhaskey` is false.

```
\pydata@newglobalbool{pydata@bufferhaskey}
```

5.5 File handle

`\pydata@filehandle` File handle for writing data.

```
\let\pydata@filehandle\relax
```

`\pydata@checkfilehandle` Check whether file handle has been set.

```
\def\pydata@checkfilehandle{%
  \ifx\pydata@filehandle\relax
    \pydata@error{Undefined file handle; use \string\pydatasetfilehandle}%
  \fi}
```

`\pydatasetfilehandle`
`\pydatareleasefilehandle` Set and release file handle. Release isn't strictly required, but it is necessary for basic data checking on the LaTeX side.

```
\def\pydatasetfilehandle#1{%
  \if\relax\detokenize{#1}\relax
```

```
    \pydata@error{Missing file handle}%
  \fi
  \ifx\pydata@filehandle\relax
  \else\ifx\pydata@filehandle#1\relax
  \else
    \pydata@fhstopstate{\pydata@filehandle}%
  \fi\fi
  \ifx\pydata@filehandle#1\relax
  \else
    \global\let\pydata@filehandle#1\relax
    \pydata@provideglobalbool{pydata@fhisreleased@\number#1}%
    \pydata@fhstartstate{#1}%
  \fi}
\def\pydatareleasefilehandle#1{%
  \ifcsname ifpydata@canwrite@\number#1\endcsname
  \else
    \pydata@error{Unknown file handle #1}%
  \fi
  \ifx\pydata@filehandle#1\relax
    \pydata@fhstopstate{#1}%
    \global\let\pydata@filehandle\relax
  \fi
  \ifbool{pydata@canwrite@\number#1}%
   {\ifbool{pydata@haskey@\number#1}%
     {\pydata@error{Incomplete data: key is waiting for value}}{}%
    \ifbool{pydata@indict@\number#1}%
     {\pydata@error{Incomplete data: dict is not closed}}{}%
    \ifbool{pydata@topislist@\number#1}%
     {\pydata@error{Incomplete data: list is not closed}}{}}%
   {}%
  \pydata@fhresetstate{#1}%
  \booltrue{pydata@fhisreleased@\number#1}}
```

`\pydatasetfilename`
`\pydataclosefilename`

Shortcut for creating a `\newwrite` and then passing the file handle to `\pydatasetfilehandle`. File handles are global. If the close macro is not invoked, then basic data checking on the LaTeX side will not be performed. However, TeX will automatically close open writes at the end of the compile.

```
\def\pydatasetfilename#1{%
  \if\relax\detokenize{#1}\relax
    \pydata@error{Missing filename}%
  \fi
  \ifcsname pydata@fh@#1\endcsname
  \else
    \expandafter\newwrite\csname pydata@fh@#1\endcsname
  \fi
  \pydata@provideglobalbool{pydata@fileisopen@#1}%
  \ifbool{pydata@fileisopen@#1}%
   {}%
   {\expandafter\immediate\expandafter\openout\csname pydata@fh@#1\endcsname=#1\relax
    \booltrue{pydata@fileisopen@#1}}%
  \expandafter\pydatasetfilehandle\expandafter{\csname pydata@fh@#1\endcsname}}
\def\pydataclosefilename#1{%
  \ifcsname pydata@fh@#1\endcsname
    \ifbool{pydata@fileisopen@#1}%
```

```
     {\expandafter\pydatareleasefilehandle\expandafter{\csname pydata@fh@#1\endcsname}%
      \expandafter\immediate\expandafter\closeout\csname pydata@fh@#1\endcsname
      \boolfalse{pydata@fileisopen@#1}}%
     {}%
  \else
    \pydata@error{Unknown file name "#1"}%
  \fi}
```

5.6 Buffer

Key-value data can be written directly to file once a dict is opened. It is also possible to accumulate key-value data in a "buffer." This is convenient when the data serves as input to an external program that generates cached content. Buffered data can be hashed in memory without being written to file, so the existence of cached content can be checked efficiently.

The buffer consists of a sequence of macros of the form `\<buffer_name>line<n>`, where each line of data corresponds to a macro and `<n>` is an integer greater than or equal to one. The length of the buffer is stored in the counter `<buffer_name>length`. The buffer includes comma-separated key-value data, without any opening or closing dict delimiters `{}`.

`pydata@bufferindex` Counter for looping through buffers.

```
\newcounter{pydata@bufferindex}
\setcounter{pydata@bufferindex}{0}
```

`\pydatasetbuffername` `\pydata@buffername` `\pydata@bufferlinename` `\pydata@bufferlengthname` Set the buffer base name and create a corresponding length counter if it does not exist.

```
\def\pydatasetbuffername#1{%
  \ifbool{pydata@bufferhaskey}%
   {\pydata@error{Cannot change buffers when a buffered key is waiting for a value}}%
   {}%
  \gdef\pydata@buffername{#1}%
  \gdef\pydata@bufferlinename{#1line}%
  \gdef\pydata@bufferlengthname{#1length}%
  \ifcsname c@\pydata@bufferlengthname\endcsname
  \else
    \expandafter\newcounter\expandafter{\pydata@bufferlengthname}%
  \fi}
\pydatasetbuffername{pydata@defaultbuffer}
```

`\pydatawritebuffer` Write existing buffer macros to file handle.

```
\def\pydatawritebuffer{%
  \ifnum\expandafter\value\expandafter{\pydata@bufferlengthname}<1\relax
    \pydata@error{Cannot write empty buffer}%
  \fi
  \pydata@checkfilehandle
  \ifbool{pydata@indict}{}{\pydata@error{Cannot write buffer unless in a dict}}%
  \ifbool{pydata@haskey}%
   {\pydata@error{Cannot write buffer when file has a key waiting for a value}}{}%
  \ifbool{pydata@bufferhaskey}%
   {\pydata@error{Cannot write buffer when a buffered key is waiting for a value}}{}%
  \setcounter{pydata@bufferindex}{1}%
  \loop\unless\ifnum\value{pydata@bufferindex}>%
      \expandafter\value\expandafter{\pydata@bufferlengthname}\relax
    \immediate\write\pydata@filehandle{%
      \csname\pydata@bufferlinename\arabic{pydata@bufferindex}\endcsname}%
    \stepcounter{pydata@bufferindex}%
  \repeat
  \setcounter{pydata@bufferindex}{0}}
```

`\pydataclearbuffername` Delete the buffer: `\let` all line macros to an undefined macro, and set length to zero.

```
\def\pydataclearbuffername#1{%
  \def\pydata@clearbuffername{#1}%
  \ifcsname c@#1length\endcsname
  \else
    \pydata@error{Buffer #1 does not exist}%
  \fi
  \setcounter{pydata@bufferindex}{1}%
  \loop\unless\ifnum\value{pydata@bufferindex}>\value{#1length}\relax
    \expandafter\global\expandafter\let
      \csname#1line\arabic{pydata@bufferindex}\endcsname\pydata@undefined
    \stepcounter{pydata@bufferindex}%
  \repeat
  \setcounter{#1length}{0}%
  \setcounter{pydata@bufferindex}{0}%
  \ifx\pydata@clearbuffername\pydata@buffername
    \boolfalse{pydata@bufferhaskey}%
  \fi}
```

`\pydatabuffermdfivesum` Calculate buffer MD5.

```
\def\pydatabuffermdfivesum{%
  \pdf@mdfivesum{%
    \ifnum\expandafter\value\expandafter{\pydata@bufferlengthname}<1
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi
    {}{\pydatabuffermdfivesum@i{1}}}}
\def\pydatabuffermdfivesum@i#1{%
  \csname\pydata@bufferlinename#1\endcsname^^J%
  \ifnum\expandafter\value\expandafter{\pydata@bufferlengthname}=#1
    \expandafter\@gobble
  \else
    \expandafter\@firstofone
  \fi
  {\expandafter\pydatabuffermdfivesum@i\expandafter{\the\numexpr#1+1 }}}
```

5.7 String processing

Ensure correct catcode for double quotation mark, which will be used for delimiting all Python string literals.

```
\begingroup
\catcode`\"=12\relax
```

`\pydata@escstrtext` Escape string text by replacing \ with \\ and " with \". Any text that requires expansion must be expanded prior to escaping. The string text is processed with `\detokenize` to ensure catcodes and prepare it for writing. This is redundant in cases where text has already been processed with `\FVExtraDetokenizeVArg`.

```
\begingroup
\catcode`\!=0
!catcode`!\=12
!gdef!pydata@escstrtext#1{%
  !expandafter!pydata@escstrtext@i!detokenize{#1}\!FV@Sentinel}
!gdef!pydata@escstrtext@i#1\#2!FV@Sentinel{%
  !if!relax!detokenize{#2}!relax
    !expandafter!@firstoftwo
  !else
    !expandafter!@secondoftwo
  !fi
   {!pydata@escstrtext@ii#1"!FV@Sentinel}%
   {!pydata@escstrtext@ii#1\\"!FV@Sentinel!pydata@escstrtext@i#2!FV@Sentinel}}
!gdef!pydata@escstrtext@ii#1"#2!FV@Sentinel{%
  !if!relax!detokenize{#2}!relax
    !expandafter!@firstoftwo
  !else
    !expandafter!@secondoftwo
  !fi
   {#1}%
   {#1\"!pydata@escstrtext@ii#2!FV@Sentinel}}
!endgroup
```

`\pydata@quotestr` Escape a string then quote it with ".

```
\gdef\pydata@quotestr#1{%
  "\pydata@escstrtext{#1}"}
```

`\pydata@mlstropen` `\pydata@mlstrclose` Multi-line string delimiters. The opening delimiter has a trailing backslash to prevent the string from starting with a newline.

```
\begingroup
\catcode`\!=0
!catcode`!\=12
!gdef!pydata@mlstropen{"""\}
!gdef!pydata@mlstrclose{"""}
!endgroup
```

End " catcode.

```
\endgroup
```

5.8 Metadata

`\pydata@schema` Macro storing key-value schema data.

```
\def\pydata@schema{}
```

`\pydatasetschemamissing` `\pydata@schemamissing` Define behavior for missing key-value pairs in a schema.

```
\let\pydata@schemamissing@error\relax
\let\pydata@schemamissing@rawstr\relax
\let\pydata@schemamissing@evalany\relax
\def\pydatasetschemamissing#1{%
  \ifcsname pydata@schemamissing@\detokenize{#1}\endcsname
  \else
```

```
    \pydata@error{Invalid schema missing setting #1}%
  \fi
  \gdef\pydata@schemamissing{#1}}
\pydatasetschemamissing{error}
```

`\pydatasetschemakeytype` Define a key's schema. For example, `\pydatasetschemakeytype{key}{int}`.

```
\begingroup
\catcode`\:=12\relax
\catcode`\,=12\relax
\gdef\pydatasetschemakeytype#1#2{%
  \ifbool{pydata@hasmeta}{\pydata@error{Must create schema before writing metadata}}{}%
  \ifbool{pydata@topexists}{\pydata@error{Must create schema before writing data}}{}%
  \expandafter\def\expandafter\pydata@schema\expandafter{%
    \pydata@schema\pydata@quotestr{#1}: \pydata@quotestr{#2}, }}
\endgroup
```

`\pydataclearschema` Delete existing schema. This isn't done automatically upon writing so that a schema can be defined and then reused.

```
\def\pydataclearschema{%
  \gdef\pydata@schema{}}
```

`\pydataclearmeta` Delete existing metadata. This isn't done automatically upon writing so that metadata can be defined and then reused.

```
\def\pydataclearmeta{%
  \pydatasetschemamissing{error}%
  \pydataclearschema}
```

`\pydatawritemeta` Write metadata to file, including any schema.

```
\begingroup
\catcode`\:=12\relax
\catcode`\#=12\relax
\catcode`\,=12\relax
\gdef\pydatawritemeta{%
  \ifbool{pydata@canwrite}%
   {}{\pydata@error{Data was already written; cannot write metadata}}%
  \ifbool{pydata@hasmeta}{\pydata@error{Already wrote metadata}}{}%
  \ifbool{pydata@topexists}{\pydata@error{Must write metadata before writing data}}{}%
  \edef\pydata@meta@exp{%
    # latex2pydata metadata:
    \@charlb
    \pydata@quotestr{schema_missing}:
    \expandafter\pydata@quotestr\expandafter{\pydata@schemamissing},
    \pydata@quotestr{schema}:
    \ifx\pydata@schema\pydata@empty
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi
     {None}{\@charlb\pydata@schema\@charrb},
    \@charrb}%
  \immediate\write\pydata@filehandle{\pydata@meta@exp}%
  \booltrue{pydata@hasmeta}}
\endgroup
```

5.9 Collection delimiters

\pydatawritelistopen \pydatawritelistclose Write list delimiters. These are only used when the top-level data structure is a list: `list[dict[str, str]]`.

```
\begingroup
\catcode`\[=12\relax
\catcode`\]=12\relax
\gdef\pydatawritelistopen{%
  \pydata@checkfilehandle
  \ifbool{pydata@canwrite}%
   {}{\pydata@error{Data structure is closed; cannot write delim}}%
  \ifbool{pydata@topexists}%
   {\pydata@error{Top-level data structure already exists}}{}%
  \immediate\write\pydata@filehandle{[}%
  \booltrue{pydata@topexists}%
  \booltrue{pydata@topislist}}
\gdef\pydatawritelistclose{%
  \ifbool{pydata@topexists}%
   {}{\pydata@error{No data structure is open; cannot write delim}}%
  \ifbool{pydata@topislist}%
   {}{\pydata@error{Top-level data structure is not a list}}%
  \ifbool{pydata@haskey}%
   {\pydata@error{Cannot close data structure when key is waiting for value}}{}%
  \immediate\write\pydata@filehandle{]}%
  \boolfalse{pydata@topexists}%
  \boolfalse{pydata@topislist}%
  \boolfalse{pydata@hasmeta}%
  \boolfalse{pydata@canwrite}}
\endgroup
```

\pydatawritedictopen \pydatawritedictclose Write dict delimiters. These are not the top-level data structure for `list[dict[str, str]]` but are the top-level data structure for `dict[str, str]`.

```
\begingroup
\catcode`\,=12\relax
\gdef\pydatawritedictopen{%
  \ifbool{pydata@topislist}%
   {\ifbool{pydata@indict}{\pydata@error{Already in a dict; cannot nest}}{}%
    \immediate\write\pydata@filehandle{\@charlb}%
    \booltrue{pydata@indict}}%
   {\pydata@checkfilehandle
    \ifbool{pydata@canwrite}%
     {}{\pydata@error{Data structure is closed; cannot write delim}}%
    \ifbool{pydata@topexists}%
     {\pydata@error{Top-level data structure already exists}}{}%
    \immediate\write\pydata@filehandle{\@charlb}%
    \booltrue{pydata@topexists}%
    \booltrue{pydata@indict}}}
\gdef\pydatawritedictclose{%
  \ifbool{pydata@indict}{}{\pydata@error{No dict is open; cannot write delim}}%
  \ifbool{pydata@haskey}%
   {\pydata@error{Cannot close data structure when key is waiting for value}}{}%
  \ifbool{pydata@topislist}%
   {\immediate\write\pydata@filehandle{\@charrb,}%
    \boolfalse{pydata@indict}}%
  {\immediate\write\pydata@filehandle{\@charrb}%
   \boolfalse{pydata@indict}%
   \boolfalse{pydata@topexists}%
   \boolfalse{pydata@hasmeta}%
   \boolfalse{pydata@canwrite}}
\endgroup
```

5.10 Keys and values

\pydatawritekey
\pydatabufferkey

Write key to file or append it to the buffer.

```
\begingroup
\catcode`\:=12\relax
\gdef\pydatawritekey{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatawritekey@i}}}
\gdef\pydatawritekey@i#1{%
  \ifbool{pydata@indict}{}{\pydata@error{Cannot write a key unless in a dict}}%
  \ifbool{pydata@haskey}{\pydata@error{Cannot write a key when waiting for a value}}{}%
  \immediate\write\pydata@filehandle{%
    \pydata@quotestr{#1}:%
  }%
  \booltrue{pydata@haskey}}
\gdef\pydatabufferkey{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatabufferkey@i}}}
\gdef\pydatabufferkey@i#1{%
  \ifbool{pydata@bufferhaskey}%
   {\pydata@error{Cannot buffer a key when waiting for a value}}{}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@quotestr{#1}:%
    }%
  \booltrue{pydata@bufferhaskey}}
\endgroup
```

\pydatawritevalue
\pydatabuffervalue

Write a value to file or append it to the buffer.

```
\begingroup
\catcode`\,=12\relax
\gdef\pydatawritevalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatawritevalue@i}}}
\gdef\pydatawritevalue@i#1{%
  \ifbool{pydata@haskey}{}{\pydata@error{Cannot write value when waiting for a key}}%
  \immediate\write\pydata@filehandle{%
    \pydata@quotestr{#1},%
  }%
  \boolfalse{pydata@haskey}}
\gdef\pydatabuffervalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatabuffervalue@i}}}
\gdef\pydatabuffervalue@i#1{%
  \ifbool{pydata@bufferhaskey}%
   {}{\pydata@error{Cannot buffer value when waiting for a key}}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
```

```
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@quotestr{#1},%
    }%
  \boolfalse{pydata@bufferhaskey}}
\endgroup
```

\pydatawritekeyvalue
\pydatawritekeyedefvalue
\pydatabufferkeyvalue
\pydatabufferkeyedefvalue

Write a key and a single-line value to file simultaneously, or append them to the buffer.

```
\begingroup
\catcode`\:=12\relax
\catcode`\,=12\relax
\gdef\pydatawritekeyvalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatawritekeyvalue@i}}}
\gdef\pydatawritekeyvalue@i#1{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatawritekeyvalue@ii{#1}}}}
\gdef\pydatawritekeyvalue@ii#1#2{%
  \ifbool{pydata@indict}{}{\pydata@error{Cannot write a key unless in a dict}}%
  \ifbool{pydata@haskey}{\pydata@error{Cannot write a key when waiting for a value}}{}%
  \immediate\write\pydata@filehandle{%
    \pydata@quotestr{#1}: \pydata@quotestr{#2},%
  }}
\gdef\pydatawritekeyedefvalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatawritekeyedefvalue@i}}}
\gdef\pydatawritekeyedefvalue@i#1#2{%
  \edef\pydata@tmp{#2}%
  \expandafter\pydatawritekeyedefvalue@ii\expandafter{\pydata@tmp}{#1}}
\gdef\pydatawritekeyedefvalue@ii#1#2{%
  \FVExtraDetokenizeVArg{\pydatawritekeyvalue@ii{#2}}{#1}}
\gdef\pydatabufferkeyvalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatabufferkeyvalue@i}}}
\gdef\pydatabufferkeyvalue@i#1{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatabufferkeyvalue@ii{#1}}}}
\gdef\pydatabufferkeyvalue@ii#1#2{%
  \ifbool{pydata@bufferhaskey}%
   {\pydata@error{Cannot buffer a key when waiting for a value}}{}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@quotestr{#1}: \pydata@quotestr{#2},%
    }}
\gdef\pydatabufferkeyedefvalue{%
  \FVExtraReadVArg{\FVExtraDetokenizeVArg{\pydatabufferkeyedefvalue@i}}}
\gdef\pydatabufferkeyedefvalue@i#1#2{%
  \edef\pydata@tmp{#2}%
  \expandafter\pydatabufferkeyedefvalue@ii\expandafter{\pydata@tmp}{#1}}
\gdef\pydatabufferkeyedefvalue@ii#1#2{%
  \FVExtraDetokenizeVArg{\pydatabufferkeyvalue@ii{#2}}{#1}}
\endgroup
```

\pydatawritemlvaluestart
\pydatawritemlvalueline
\pydatawritemlvalueend
\pydatabuffermlvaluestart
\pydatabuffermlvalueline
\pydatabuffermlvalueend

Write a line of a multi-line value to file or append it to the buffer. Write the end delimiter of the value to file or append it to the buffer.

```
\begingroup
\catcode`\,=12\relax
```

```
\gdef\pydatawritemlvaluestart{%
  \ifbool{pydata@haskey}{}{\pydata@error{Cannot write value when waiting for a key}}%
  \immediate\write\pydata@filehandle{%
    \pydata@mlstropen
  }}
\gdef\pydatawritemlvalueline#1{%
  \ifbool{pydata@haskey}{}{\pydata@error{Cannot write value when waiting for a key}}%
  \immediate\write\pydata@filehandle{%
    \pydata@escstrtext{#1}%
  }}
\gdef\pydatawritemlvalueend{%
  \ifbool{pydata@haskey}{}{\pydata@error{Cannot write value when waiting for a key}}%
  \immediate\write\pydata@filehandle{%
    \pydata@mlstrclose,%
  }%
  \boolfalse{pydata@haskey}}
\gdef\pydatabuffermlvaluestart{%
  \ifbool{pydata@bufferhaskey}%
   {}{\pydata@error{Cannot buffer value when waiting for a key}}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@mlstropen
    }}
\gdef\pydatabuffermlvalueline#1{%
  \ifbool{pydata@bufferhaskey}%
   {}{\pydata@error{Cannot buffer value when waiting for a key}}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@escstrtext{#1}%
    }}
\gdef\pydatabuffermlvalueend{%
  \ifbool{pydata@bufferhaskey}%
   {}{\pydata@error{Cannot buffer value when waiting for a key}}%
  \expandafter\stepcounter\expandafter{\pydata@bufferlengthname}%
  \expandafter\xdef\csname
    \pydata@bufferlinename\expandafter\arabic\expandafter{\pydata@bufferlengthname}%
    \endcsname{%
      \pydata@mlstrclose,%
    }%
  \boolfalse{pydata@bufferhaskey}}
\endgroup
```

pydatawritemlvalue

```
\newenvironment{pydatawritemlvalue}%
 {\VerbatimEnvironment
  \pydatawritemlvaluestart
  \begin{VerbatimWrite}[writer=\pydatawritemlvalueline]}%
 {\end{VerbatimWrite}}
\AfterEndEnvironment{pydatawritemlvalue}{\pydatawritemlvalueend}
```

pydatabuffermlvalue

```
\newenvironment{pydatabuffermlvalue}%
 {\VerbatimEnvironment
  \pydatabuffermlvaluestart
  \begin{VerbatimBuffer}[bufferer=\pydatabuffermlvalueline]}%
 {\end{VerbatimBuffer}%
  \pydatabuffermlvalueend}
```