diffx Documentation

Release 1.0

Christian Hammond

Sep 20, 2022

CONTENTS

If you're a software developer, you've probably worked with diff files. Git diffs, Subversion diffs, CVS diffs.. Some kind of diff. You probably haven't given it a second thought, really. You make some changes, run a command, a diff comes out. Maybe you hand it to someone, or apply it elsewhere, or put it up for review.

Diff files show the differences between two text files, in the form of inserted (+) and deleted (-) lines. Along with this, they contain some basic information used to identify the file (usually just the name/relative path within some part of the tree), maybe a timestamp or revision, and maybe some other information.

Most people and tools work with *Unified Diffs*. They look like this:

```
--- readme    2016-01-26 16:29:12.000000000 -0800
+++ readme    2016-01-31 11:54:32.000000000 -0800
@@ -1 +1,3 @@
 Hello there
+
+Oh hi!
```

Or this:

```
Index: readme
===================================================================
RCS file: /cvsroot/readme,v
retrieving version 1.1
retrieving version 1.2
diff -u -p -r1.1 -r1.2
--- readme    26 Jan 2016 16:29:12 -0000        1.1
+++ readme    31 Jan 2016 11:54:32 -0000        1.2
@@ -1 +1,3 @@
 Hello there
+
+Oh hi!
```

Or this:

```
diff --git a/readme b/readme
index d6613f5..5b50866 100644
--- a/readme
+++ b/readme
@@ -1 +1,3 @@
 Hello there
+
+Oh hi!
```

Or even this:

```
Index: readme
===================================================================
--- (revision 123)
+++ (working copy)
Property changes on: .
___________________________________________________________________
Modified: myproperty
## -1 +1 ##
-old value
+new value
```

Or this!

```
==== //depot/proj/logo.png#1 ==A== /src/proj/logo.png ====
Binary files /tmp/logo.png and /src/proj/logo.png differ
```

CHAPTER

ONE

HERE'S THE PROBLEM

Unified Diffs themselves are not a viable standard for modern development. They only standardize parts of what we consider to be a diff, namely the `---`/`+++` lines for file identification, `@@ ... @@` lines for diff hunk offsets/sizes, and `-`/`+` for inserted/deleted lines. They **don't** standardize encodings, revisions, metadata, or even how filenames or paths are represented!

This makes it *very* hard for patch tools, code review tools, code analysis tools, etc. to reliably parse any given diff and gather useful information, other than the changed lines, particularly if they want to support multiple types of source control systems. And there's a lot of good stuff in diff files that some tools, like code review tools or patchers, want.

You should see what GNU Patch has to deal with.

Unified Diffs have not kept up with where the world is going. For instance:

- A single diff can't represent a list of commits
- There's no standard way to represent binary patches
- Diffs don't know about text encodings (which is more of a problem than you might think)
- Diffs don't have any standard format for arbitrary metadata, so everyone implements it their own way.

We're long past the point where diffs should be able to do all this. Tools should be able to parse diffs in a standard way, and should be able to modify them without worrying about breaking anything. It should be possible to load a diff, any diff, using a Python module or Java package and pull information out of it.

Unified Diffs aren't going away, and they don't need to. We just need to add some extensibility to them. And that's completely doable, today.

CHAPTER

TWO

HERE'S THE GOOD NEWS

Unified Diffs, by nature, are *very* forgiving, and they're everywhere, in one form or another. As you've seen from the examples above, tools shove all kinds of data into them. Patchers basically skip anything they don't recognize. All they really lack is structure and standards.

Git's diffs are the closest things we have to a standard diff format (in that both Git and Mercurial support it, and Subversion pretends to, but poorly), and the closest things we have to a modern diff format (as they optionally support binary diffs and have a general concept of metadata, though it's largely Git-specific).

They're a good start, though still not formally defined. Still, we can build upon this, taking some of the best parts from Git diffs and from other standards, and using the forgiving nature of Unified Diffs to define a new, structured Unified Diff format.

CHAPTER

THREE

DIFFX FILES

We propose a new format called Extensible Diffs, or DiffX files for short. These are **fully backwards-compatible with existing tools**, while also being **future-proof** and remaining **human-readable**.

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..preamble: indent=4, length=319, mimetype=text/markdown
    Convert legacy header building code to Python 3.

    Header building for messages used old Python 2.6-era list comprehensions
    with tuples rather than modern dictionary comprehensions in order to build
    a message list. This change modernizes that, and swaps out six for a
    3-friendly `.items()` call.
#..meta: format=json, length=270
{
    "author": "Christian Hammond <christian@example.com>",
    "committer": "Christian Hammond <christian@example.com>",
    "committer date": "2021-06-02T13:12:06-07:00",
    "date": "2021-06-01T19:26:31-07:00",
    "id": "a25e7b28af5e3184946068f432122c68c1a30b23"
}
#..file:
#...meta: format=json, length=176
{
    "path": "/src/message.py",
    "revision": {
        "new": "f814cf74766ba3e6d175254996072233ca18a690",
        "old": "9f6a412b3aee0a55808928b43f848202b4ee0f8d"
    }
}
#...diff: length=629
--- /src/message.py
+++ /src/message.py
@@ -164,10 +164,10 @@
             not isinstance(headers, MultiValueDict)):
             # Instantiating a MultiValueDict from a dict does not ensure that
             # values are lists, so we have to ensure that ourselves.
-            headers = MultiValueDict(dict(
-                (key, [value])
-                for key, value in six.iteritems(headers)
-            ))
```

(continues on next page)

(continued from previous page)

```
+            headers = MultiValueDict({
+                key: [value]
+                for key, value in headers.items()
+            })

         if in_reply_to:
             headers['In-Reply-To'] = in_reply_to
```

DiffX files are built on top of Unified Diffs, providing structure and metadata that tools can use. Any DiffX file is a complete Unified Diff, and can even contain all the legacy data that Git, Subversion, CVS, etc. may want to store, while also structuring data in a way that any modern tool can easily read from or write to using **standard parsing rules.**

Let's summarize. Here are some things DiffX offers:

- Standardized rules for parsing diffs
- Formalized storage and naming of metadata for the diff and for each commit and file within
- Ability to extend the format without breaking existing parsers
- Multiple commits can be represented in one diff file
- Git-compatible diffs of binary content
- Knowledge of text encodings for files and diff metadata
- Compatibility with all existing parsers and patchers (for all standard diff features – new features will of course require support in tools, but can still be parsed)
- Mutability, allowing a tool to easily open a diff, record new data, and write it back out

DiffX is **not** designed to:

- Force all tools to support a brand new file format
- Break existing diffs in new tools or require tools to be rewritten
- Create any sort of vendor lock-in

CHAPTER

FOUR

WANT TO LEARN MORE?

If you want to know more about what diffs are lacking, or how they differ from each other (get it?), then read *The Problems with Diffs*.

If you want to get your hands dirty, check out the *DiffX File Format Specification*.

See *example DiffX files* to see this in action.

Other questions? We have a *FAQ* for you.

CHAPTER

FIVE

IMPLEMENTATIONS

- Python: *pydiffx*

CHAPTER

SIX

WHO'S USING DIFFX?

- Review Board from Beanbag. We built DiffX to solve long-standing problems we've encountered with diffs, and are baking support into all our products.

6.1 The Problems with Diffs

Diffs today have a number of problems that may not seem that obvious if you're not working closely with them. Parsing them, generating them, passing them between various systems.

We covered some of this on the front page, but let's go into more detail on the problems with diffs today.

6.1.1 Revision control systems represent data differently

There really isn't much of a standard in how you actually store information in diffs. All you really can depend on are the original and modified filenames (but not the format used to show them), and the file modifications.

A number of things have been bolted onto diffs and handled by GNU patch over the years, but very little has become standardized. This makes it very difficult to reliably store or parse metadata without writing a lot of custom code.

Git, for instance, needs to track data such as file modes, SHA1s, similarity information (for move/rename detection), and more. They do this with some strings that appear above the typical `---`/`+++` filename blocks that Git knows how to parse, but GNU patch will ignore. For instance, to handle a file move, you might get:

```
diff --git a/README b/README2
index 91bf7ab..dd93b71 100644
similarity index 95%
rename from README
rename to README2
--- a/README
+++ b/README
```

Perforce, on the other hand, doesn't encode any information on revisions or file modes, requiring that tools add their own metadata to the files. For example, Review Board adds this additional data for a moved file with changes (based on an existing extended Perforce diff format it adopted for compatibility):

```
Moved from: //depot/project/README
Moved to: //depot/project/README2
--- //depot/project/README  //depot/project/README#2
+++ //depot/project/README2  12-10-83 13:40:05
```

Or without changes:

```
==== //depot/project/README#2 ==MV== //depot/project/README2 ====
```

Let's look at a simple diff in CVS:

```
Index: README
===================================================================
RCS file: /path/to/README,v
retrieving revision 1.1
retrieving revision 1.2
diff -u -p -r1.1 -r1.2
--- README    07 May 2014 08:50:30 -0000      1.1
+++ README    10 Dec 2014 13:40:05 -0000      1.2
```

No real consistency, and the next revision control system that comes along will probably end up injecting its own arbitrary content in diffs.

6.1.2 Operations like moves/deletes are inconsistent

Diffs are pretty good at handling file modifications and, generally, the introduction of new files. Unfortunately, they fall short at handling other simple operations, like a deleted file or a moved/renamed file. Again, different implementations end up representing these operations in different ways.

For some time, Perforce's **`p4 diff`** wouldn't show deleted file content, prompting some companies to write their own wrapper.

TFS won't even show added or deleted content natively.

Git represents deleted files with:

```
diff --git a/README b/README
deleted file mode 100644
index 91bf7ab..0000000
--- a/README
+++ /dev/null
@@ -1,3 +0,0 @@
-All the lines
-are deleted
-one by one
```

Subverison, depending on the version and the way the diffs were built, may use:

```
Index: README
===================================================================
--- README      (revision 4)
+++ README      (working copy)
@@ -1,3 +0,0 @@
-All the lines
-are deleted
-one by one
```

Or it may be use:

```
Index: README
===================================================================
```

(continues on next page)

(continued from previous page)

```
--- README      (revision 4)
+++ README      (nonexistent)
@@ -1,3 +0,0 @@
-All the lines
-are deleted
-one by one
```

Or:

```
Index: README   (deleted)
===================================================================
--- README      (revision 4)
+++ README      (working copy)
@@ -1,3 +0,0 @@
-All the lines
-are deleted
-one by one
```

And that's not even factoring in the versions that localized "`(nonexistent)`" or "`(working copy)`" into other languages, in the diff!

Most are consistent with the removal of the lines, but that's about it. Some have metadata explicitly indicating a delete, but others don't differentiate between deleted files and removing all lines from files.

Copies/moves are worse. There is no standard at all, and SVN/Git/etc. have been forced to work around this by inventing their own formats and command line switches, which the patch tool needs to have special knowledge of.

6.1.3 No support for binary files

Binary files have no official support in diffs. Git has its own support for binary files in diffs, but GNU patch rejects them, requiring **`git apply`** to be used instead.

Very few systems even try to support binary files in diffs, instead simply adding a marker explaining the file has unspecified binary changes. This usually says `Binary files <file> and <file> differ`.

In the world of binary files in diffs, Git's way of handling them seems to be the current de-facto standard, as **`hg diff --git`** will generate these changes as well. Still, it's not very wide-spread yet.

6.1.4 Text encodings are unclear

When you view a diff, you have to essentially guess at the encoding. This can be done by trying a few encodings, or assuming an encoding if you know the encodings in the repository the diff is being applied to. This is pretty bad, though. Today, there's just no way to consistently know for sure how to properly decode text in a diff.

This manifests in the wild when working with international teams and different languages and sets of editors. If the encoding of a file has been changed from, say, UTF-8 to zh_CN, then any tool working with the diff and the source files will break, and it's hard to diagnose why at first.

6.1.5 They're limited to single commits

Tools will generally output a separate diff file for every commit, which means more files to keep track of and e-mail around, and means that the ordering must be respected when applying the changes or when uploading files to any services or software that needs to operate on them. This isn't a huge problem in practice, but ideally, a diff could just contain each commit.

DVCS is basically the standard for all modern source code management solutions, but that wasn't the case when Unified Diffs were first created. A new diff format should account for this.

6.1.6 Fixing these problems

These problems are all solvable, without breaking existing diffs.

Diffs have a lot of flexibility in what kind of "garbage" data is stored, so long as the diff contains at least one genuine modification to a file. Git, SVN, etc. diffs leverage this to store additional data.

We're leveraging this as well. We store an encoding marker at the top of the file and to break the diff into sections. Sections can contain options to control parsing behavior, metadata on the content represented by the section, and the content itself. The content may be standard text diff data (with or without implementation-specific metadata) or binary diff content.

Through this, it's also possible to extend the format by defining custom metadata, custom sections, and to specify custom parsing behavior in sections.

Diffs also don't have limits as to how many times a file shows up with modifications. Tools like **`patch`** and **`diffstat`** are more than happy to work with any entries that come up. That means we can safely store the diffs for a series of commits in one file and still be able to patch safely.

This is all done without breaking parsing/patching behavior for existing diffs, or causing incompatibilities between DiffX files and existing tools.

6.2 DiffX File Format Specification

Version
1.0

Last Updated
April 26, 2022

Copyright

6.2.1 Introduction

DiffX files are a superset of the *Unified Diff* format, intended to bring structure, parsing rules, and common metadata for diffs while retaining backwards-compatibility with existing software (such as tools designed to work with diffs built by Git, Subversion, CVS, or other software).

Scope

DiffX offers:

- Standardized rules for parsing diffs
- Formalized storage and naming of metadata for the diff and for each commit and file within
- Ability to extend the format without breaking existing parsers
- Multiple commits can be represented in one diff file
- Git-compatible diffs of binary content
- Knowledge of text encodings for files and diff metadata
- Compatibility with all existing parsers and patching tools (for all standard diff features – new features will of course require support in tools, but can still be parsed)
- Mutability, allowing a tool to easily open a diff, record new data, and write it back out

DiffX is not designed to:

- Force all tools to support a brand new file format
- Break existing diffs in new tools or require tools to be rewritten
- Create any sort of vendor lock-in

Filenames

Filenames can end in `.diffx` or in `.diff`.

It is expected that most diffs will retain the `.diff` file extension, though it might make sense for some tools to optionally write or export a `.diffx` file extension to differentiate from non-DiffX diffs.

Software should never assume a file is or is not a DiffX file purely based on the file extension. It must attempt to parse at least the file's *#diffx:* header according to this specification in order to determine the file format.

General File Structure

DiffX files are broken into hierarchical *sections*, which may contain free-form text, metadata, diffs, or subsections.

Each section is preceded by a *section header*, which may provide options to identify *content encodings*, content length information, and other parsing hints relevant to the section.

All DiffX-specific content has been designed in a way to all but ensure it will be ignored by most diff parsers (including GNU patch) if DiffX is not supported by the parser.

6.2.2 Section Definitions

DiffX files are grouped into hierarchical sections, each of which are preceded by a header that may list options that define how content or subsections are parsed.

Section Headers

Sections headers are indicated by a `#` at the start of the line, followed by zero or more periods (`.`) to indicate the nesting level, followed by the section name, `:`, and then optionally any parsing options for that section.

They are always encoded as ASCII strings, and are unaffected by the parent section's encoding (see *Encoding Rules*).

Section headers can be parsed with this regex:

```
^#(?P<level>\.{0,3})(?P<section_name>[a-z]+):\s*(?P<options>.*)$
```

For instance, the following are valid section headers:

```
#diffx: version=1.0
#.change:
#..meta: length=100, my-option=value, another-option=another-value
```

The following are not:

```
#diffx::
.preamble
#.change
#....diff:
```

Header Options

Headers may contain options that inform the parser of how to treat nested content or sections. The available options are dependent on the type of section.

Options are key/value pairs, each pair separated by a comma and space (`", "`), with the key and value separated by an equals sign (`"="`). Spaces are not permitted on either side of the `"="`.

Keys must be in the following format: `[A-Za-z][A-Za-z0-9_-]*`

Values must be in the following format: `[A-Za-z9-9/._-]+`

Each option pair can be parsed with this regex:

```
(?P<option_key>[A-Za-z][A-Za-z0-9_-]*)=(?P<option_value>[A-Za-z0-9/._-]+)
```

Note: It's recommended that diff generators write options in alphabetical order, to ensure consistent generation between implementations.

The following are valid headers with options:

```
#diffx: version=1.0
#.change:
#..meta: length=100, my-option=value, another-option=another-value
```

The following are not:

```
#diffx: 1.0
#..meta: option=100+
#..meta: option=value,option2=value
```

(continues on next page)

(continued from previous page)

```
#..meta: option=value, option2=value:
#..meta: _option=value
#..meta: my-option = value
```

Section IDs

The following are valid section IDs (as combinations of `level` and `section_name`):

- `diffx`
- `.meta`
- `.preamble`
- `.change`
- `..meta`
- `..preamble`
- `..file`
- `...meta`
- `...diff`

Anything else should raise a parsing error.

Section Order

Sections must appear in a specific order. Some sections are optional, some are required, and some may repeat themselves. You can refer to the order listed in *Section IDs*, or see *Section Hierarchy* for detailed information on each section and their valid subsections.

DiffX parsers can use the following state tree to determine which sections may appear next when parsing a section:

- `diffx`
  - `.preamble`
  - `.meta`
  - `.change`
- `.preamble`
  - `.meta`
  - `.change`
- `.meta`
  - `.change`
- `.change`
  - `..preamble`
  - `..meta`
  - `..file`
- `..preamble`

    - ..file
- ..meta
    - ..change
    - ..file
- ..file
    - ...meta
- ...meta
    - ...diff
    - ..file
    - .change
- ...diff
    - ..file
    - .change

Section Types

There are two types of DiffX sections:

1. *Container Sections* – Sections that contain one or more subsections
2. *Content Sections* – Sections that contain text content

Container Sections

Container sections contain no content of their own, but will contain one or more subsections.

The following are the container sections defined in this specification:

- *DiffX Main Section*
- *Change Section*
- *Changed File Section*

Options

Each container section may list the following option:

encoding (string – *optional*):

The default text encoding for child or grandchild preamble or metadata content sections.

This will typically be set once on the *DiffX Main Section*. It's recommended that diff generators use `utf-8`.

Encodings are not automatically applied to the *Changed File Diff Section*.

See *Encoding Rules*.

Listing 1: **Example**

```
#.change: type=encoding
```

Content Sections

There are three types of content sections:

- *Preamble Sections*
- *Metadata Sections*
- *Changed File Diff Section*

The following are the content sections defined in this specification:

- *DiffX Preamble Section*
- *DiffX Metadata Section*
- *Change Preamble Section*
- *Change Metadata Section*
- *Changed File Metadata Section*
- *Changed File Diff Section*

Options

Each container section supports the following options:

`encoding` (string – *optional*):
The default text encoding for the content of this section.

This will typically be set once on the *DiffX Main Section*. It's recommended that diff generators use `utf-8`. However, this can be useful if existing content using another encoding is being wrapped in DiffX.

See *Encoding Rules*.

Listing 2: **Example**

```
#..preamble: encoding=utf-32, length=217
```

`length` (integer – *required*):
The length of the section's content in bytes.

This is used by parsers to read the content for a section (up to but not including the following section or subsection), regardless of the encoding used within the section.

The length does not include the section header or its trailing newline, or any subsections. It's the length from the end of the header to the start of the next section/subsection.

Listing 3: **Example**

```
#.meta: length=100
```

`line_endings` (string – *recommended*):
The known type of line endings used within the content.

If specified, this must be either `dos` (*CRLF* line endings – `\r\n`) or `unix` (*LF* line endings – `\n`).

If a diff generator knows the type of line endings being used for content, then it should include this. This is particularly important for diff content, to aid diff parsers in splitting the lines and preserving or stripping the correct line endings.

If this option is not specified, diff parsers should determine whether the first line ends with a *CRLF* or *LF* by reading up until the first *LF* and determine whether it's preceded by a *CR*.

Design Rationale

Diffs have been encountered in production usage that use DOS line endings but include Line Feed characters as part of the line's data, and in these situations, knowing the line endings up-front will aid in parsing.

Diffs have also been found that use a CRCRLF (`\r\r\n`) line feeds, as a result of a diff generator (in one known case, an older version of Perforce) being confused when diffing files from another operating system with non-native line endings. This edge case was considered but rejected, as it's ultimately a bug that should be handled before the diff is put into a DiffX file.

Preamble Sections

Metadata sections can appear directly under the *DiffX main section* or within a particular *change section*.

This section contains human-readable text, often representing a commit message, a sumamry of a complete set of changes across several files or diffs, or a merge commit's text.

This content is free-form text, but *cannot* contain anything that looks like modifications to a diff file, DiffX section information, or lines specific to a variant of a diff format. Tools should prefix each line with a set number of spaces to avoid this, setting the *indent option* to inform parsers of this number.

Preamble sections **must** end in a newline, in the section's encoding.

Preamble sections may also include a *mimetype option* help indicate whether the text is something other than plain text (such as Markdown)

See *Encoding Rules* for information on how to encode content within preamble sections.

Options

This supports the *common content section options*, along with:

`indent` (integer – *recommended*):

The number of spaces content is indented within this preamble.

In order to prevent user-provided text from breaking parsing (by introducing DiffX headers or diff data), diff generators may want to indent the content a number of spaces. This option is a hint to parsers to say how many spaces should be removed from preamble text.

A suggested value would be `4`. If left off, the default is `0`.

When writing the file, indentation MUST be applied *after* encoding the text, to ensure maximum compatibility with diff parsers.

When reading the file, indentation MUST be stripped *before* decoding the text.

Note: The order in which indentation is applied is important.

Indentation must be ASCII spaces (`0x20`), applied after the content is encoded, and stripped before it's decoded, in order to avoid encoded characters at column 0 being picked up by diff parsers as syntax.

Listing 4: **Example**

```
#.preamble: indent=4, length=55
    This content won't break parsing if it adds:

    #.change:
```

mimetype (**string** – *optional*)**:**
The mimetype of the text, as a hint to the parser.

Supported mimetypes at this time are:

- `text/plain` (default)
- `text/markdown`

Other types may be used in the future, but only if first covered by this specification. Note that consumers of the diff file are not required to render the text in these formats. It is merely a hint.

Listing 5: **Example**

```
#.preamble: length=40, mimetype=text/markdown
Here is a **description** of the change.
```

Metadata Sections

Metadata sections can appear directly under the *DiffX main section*, within a particular *change section*, or within a particular *changed file's section*.

Metadata sections contain structured JSON content. It MUST be outputted in a pretty-printed (rather than minified) format, with dictionary keys sorted and 4 space indentation. This is important for keeping output consistent across JSON implementations.

Metadata sections **must** end in a newline, in the section's encoding.

Design Rationale

JSON is widely-supported in most languages. Its syntax is unlikely to cause any conflicts with existing diff parsers (due to { and } having no special meaning in diffs, and indented content being sufficient to prevent any metadata content from appearing as DiffX, unified diff, or SCM-specific syntax.

An example metadata section with key/value pairs, lists, and strings may look like:

```
#.meta: format=json, length=209
{
    "dictionary key": {
        "sub key": {
            "sub-sub key": "value"
        }
    },
    "list key": [
       123,
       "value"
    ],
```

(continues on next page)

(continued from previous page)

```
    "some boolean": true,
    "some key": "Some string"
}
```

Options

This supports the *common content section options*, along with:

`format` (string – *recommended*):

This would indicate the metadata format. Currently, only `json` is officially supported, and is the default if not provided.

It's recommended that diff generators always provide this option in order to be explicit about the metadata format. They must not introduce their own format options without proposing it for the DiffX specification.

Diff parsers must always check for the presence of this option. If provided, it must confirm that the value is a format it can parse, and provide a suitable failure if it cannot understand the format.

New format options will only be introduced along with a DiffX specification version change.

Custom Metadata

While this specification covers many standard metadata keys, certain types of diffs, or diff generators, will need to provide custom metadata.

All custom metadata should be nested under an appropriate vendor key. For example:

```
#.meta: format=json, length=70
{
    "myscm": {
        "key1": "value",
        "key2": 123
    }
}
```

Vendors can propose to include custom metadata in the DiffX specification, effectively promoting it out of the vendor key, if it may be useful outside of the vendor's toolset.

6.2.3 Section Hierarchy

DiffX files are structured according to the following hierarchy:

- *DiffX Main Section* (*required*)
  - *DiffX Main Preamble Section* (*optional*)
  - *DiffX Main Metadata Section* (*optional*)
  - *Change (commit) Sections* (*one or more required*)
    - *Change Preamble Section* (*optional*)
    - *Change Metadata Section* (*optional*)
    - *File Sections* (*one or more required*)

- *File Metadata Section* (*required*)
- *File Diff Section* (*optional*)

DiffX Main Section

Type: *Container Section*

These sections cover the very top of a DiffX file. Each of these sections can only appear once per file.

DiffX Main Header

The first line of a DiffX file must be the start of the file section. This indicates to the parser that this is a DiffX-formatted file, and can provide options for parsing the file.

If not specified in a file, then the file cannot be treated as a DiffX file.

Options

This supports the *common container section options*, along with:

`encoding` (string – *recommended*):
The default text encoding of the DiffX file.

This does *not* cover diff content, which is treated as binary data by default.

See *Encoding Rules* for encoding rules.

Important: If unspecified, the parser cannot assume a particular encoding. This is to match behavior with existing *Unified Diff* files. It is strongly recommended that all tools that generate DiffX files specify an encoding option, with `utf-8` being the recommended encoding.

Listing 6: **Example**

```
#diffx: encoding=utf-8, version=1.0
```

`version` (string – *required*):
The DiffX specification version (currently `1.0`).

Listing 7: **Example**

```
#diffx: version=1.0
```

Subsections

- *DiffX Preamble Section* (*optional*)
- *DiffX Metadata Section* (*optional*)
- *Change Sections* (*required*)

Example

```
#diffx: encoding=utf-8, version=1.0
...
```

DiffX Preamble Section

Type: *Preamble Section*

This section contains human-readable text describing the diff as a whole. This can summarize a complete set of changes across several files or diffs, or perhaps even a merge commit's text.

You'll often see Git commit messages (or similar) at the top of a *Unified Diff* file. Those do not belong in this section. Instead, place those in the *Change Preamble section*.

Options

This supports all of the *common preamble section options*.

Example

```
#diffx: encoding=utf-8, version=1.0
#.preamble: indent=4, length=80
    Any free-form text can go here.

    It can span as many lines as you like.
```

DiffX Metadata Section

Type: *Metadata Section*

This section provides metadata on the diff file as a whole. It can contain anything that the diff generator wants to provide.

While diff generators are welcome to add additional keys, they are encouraged to either submit them for inclusion in this specification, or stick them under a namespace. For instance, a hypothetical Git-specific key for a clone URL would look like:

```
#diffx: encoding=utf-8, version=1.0
#.meta: format=json, length=82
{
    "git": {
        "clone url": "https://github.com/beanbaginc/diffx"
    }
}
```

Options

This supports all of the *common metadata section options*.

Metadata Keys

`stats` (dictionary – *recommended*):
A dictionary of statistics on the commits, containing the following sub-keys:

`changes` (integer – *recommended*):
The total number of *Change sections* in the DiffX file.

`files` (integer – *recommended*):
The total number of *File sections* in the DiffX file.

`insertions` (integer – *recommended*):
The total number of insertions (+ lines) made across all *File Diff sections*.

`deletions` (integer – *recommended*):
The total number of deletions (- lines) made across all *File Diff sections*.

Listing 8: **Example**

```
{
    "stats": {
        "changes": 4,
        "files": 2,
        "insertions": 30,
        "deletions": 15
    }
}
```

Example

```
#diffx: encoding=utf-8, version=1.0
#.meta: format=json, length=111
{
   "stats": {
       "changes": 4,
       "files": 2,
       "insertions": 30,
       "deletions": 15
   }
}
```

Change Sections

Change Section

Type: *Container Section*

A DiffX file will have one or more change sections. Each can represent a simple change to a series of files (perhaps generated locally on the command line) or a commit in a repository.

Each change section can have an optional preamble and metadata. It must have one or more file sections.

Subsections

- *Change Preamble Section* (*optional*)
- *Change Metadata Section* (*optional*)
- *Changed File Sections* (*required*)

Options

This supports the *common container section options*.

Example

```
#diffx: encoding=utf-8, version=1.0
#.change:
...
```

Change Preamble Section

Type: *Preamble Section*

Many diffs based on commits contain a commit message before any file content. We refer to this as the "preamble." This content is free-form text, but should not contain anything that looks like modifications to a diff file, in order to remain compatible with existing diff behavior.

Options

This supports all of the *common preamble section options*.

Example

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..preamble: indent=4, length=111
    Any free-form text can go here.

    It can span as many lines as you like. Represents the commit message.
```

Change Metadata Section

Type: *Metadata Section*

The change metadata sections contains metadata on the commit/change the diff represents, or anything else that the diff tool chooses to provide.

Diff generators are welcome to add additional keys, but are encouraged to either submit them as a standard, or stick them under a namespace. For instance, a hypothetical Git-specific key for a clone URL would look like:

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..meta: format=json, length=82
{
    "git": {
        "clone url": "https://github.com/beanbaginc/diffx"
    }
}
```

Options

This supports all of the *common metadata section options*.

Metadata Keys

`author` (string – *recommended*):

The author of the commit/change, in the form of `Full Name <email>`.

This is the person or entity credited with making the changes represented in the diff.

Diffs against a source code repository will usually have an author, whereas diffs against a local file might not. This field is not required, but is strongly recommended when suitable information is available.

Listing 9: **Example**

```
{
    "author": "Ann Chovey <achovey@example.com>"
}
```

`author date` (string – *recommended*):

The date/time that the commit/change was authored, in ISO 8601 format.

This can distinguish the date in which a commit was authored (e.g., when the diff was last generated, when the original commit was made, or when a change was put up for review) from the date in which it was officially placed in a repository.

Not all source code management systems differentiate between when a change was authored and when it was committed to a repository. In this case, a diff generator may opt to either:

1. Include the key and set it to the same value as *date*.
2. Leave the key out entirely.

If the key is not present, diff parsers should assume the value of *date* (if provided).

If it is present, it is expected to contain a date equal to or older than *date* (which must also be present).

Listing 10: **Example**

```
{
    "author date": "2021-05-24T18:21:06Z",
    "date": "2021-06-01T12:34:30Z"
}
```

`committer` (string – *recommended*):
The committer of the commit/change, in the form of `Full Name <email>`.

This can distinguish the person or entity responsible for placing a change in a repository from the author of that change. For example, it may be a person or an identifier for an automated system that lands a change provided by an author in a review request or pull request.

Not all source code management systems track authors and committers separately. In this case, a diff generator may opt to either:

1. Include the key and set it to the same value as *author*.
2. Leave the key out entirely.

If the key is not present, diff parsers should assume the value of *author* (if present).

If present, *author* must also be present.

Listing 11: **Example**

```
{
    "author": "Ann Chovey <achovey@example.com>",
    "committer": "John Dory <jdory@example.com>"
}
```

`date` (string – *recommended*):
The date/time the commit/change was placed in the repository, in ISO 8601 format.

This can distinguish the date in which a commit was officially placed in a repository from the date in which the change was authored.

For most source code management systems, this will be equal to the date of the commit.

For changes to local code, this may be left out, or it may equal the date/time in which the diff was generated.

Listing 12: **Example**

```
{
    "date": "2021-06-01T12:34:30Z"
}
```

`id` (string – *recommended*):
The unique ID of the change.

This value depends on the revision control system. For example, the following would be used on these systems:

- **Git:** The commit ID
- **Mercurial:** The changeset ID
- **Subversion:** The commit revision (if generating from an existing commit)

Not all revision control systems may be able to supply an ID. For example, on Subversion, there's no ID associated with pending changes to a repository. In this case, `id` can either be `null` or omitted entirely.

Listing 13: **Example**

```
{
    "id": "939dba397f0a577201f56ac72efb6f983ce69262"
}
```

`parent ids` (list of string – *optional*):
A list of parent change IDs.

This value depends on the revision control system, and may contain zero or more values.

For example, Git and Mercurial may list 1 parent ID in most cases, but may list 2 if representing a merge commit. The first commit in a tree may have no ID.

Having this information can help tools that need to know the history in order to analyze or apply the change.

If present, *id* must also be present.

Listing 14: **Example**

```
{
    "parent ids": [
        "939dba397f0a577201f56ac72efb6f983ce69262"
    ]
}
```

`stats` (dictionary – *recommended*):
A dictionary of statistics on the change.

This can be useful information to provide to diff analytics tools to help quickly determine the size and scope of a change.

`files` (integer – *required*):
The total number of *File sections* in this change section.

`insertions` (integer – *recommended*):
The total number of insertions (+ lines) made across all *File Diff sections* in this change section.

`deletions` (integer – *recommended*):
The total number of deletions (- lines) made across all *File Diff sections* in this change section.

Listing 15: **Example**

```
{
    "stats": {
        "files": 10,
        "deletions": 75,
        "insertions": 43
    }
}
```

Changed File Sections

Changed File Section

Type: *Container Section*

The file section simply contains two subsections: `#...meta:` and `#...diff:`. The metadata section is required, but the diff section may be optional, depending on the operation performed on the file.

Subsections

- *Changed File Metadata Section* (*required*)
- *Changed File Diff Section* (*optional*)

Options

This supports the *common container section options*.

Example

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
...
```

Changed File Metadata Section

Type: *Metadata Section*

The file metadata section contains metadata on the file. It may contain information about the file itself, operations on the file, etc.

At a minimum, a filename must be provided. Unless otherwise specified, the expectation is that the change is purely a content change in an existing file. This is controlled by an `op` option.

For usage in a revision control system, the `revision` options must be provided. It should be possible for the parser to have enough information between the revision and the filename to fetch a copy of the file from a matching repository.

The rest of the information is purely optional, but may be beneficial to clients, particularly those wanting to display information on file mode changes or that want to quickly display statistics on the file.

Diff generators are welcome to add additional keys, but are encouraged to either submit them as a standard, or stick them under a namespace. For instance, a hypothetical Git-specific key for a submodule reference would look like:

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...meta: format=json, length=65
{
    "git": {
        "submodule": "vendor/somelibrary"
    }
}
```

Options

This supports all of the *common metadata section options*.

Metadata Keys

`mimetype` (string or dictionary – *recommended*):

The mimetype of the file as a string. This is especially important for binary files.

When possible, the encoding of the file should be recorded in the mimetype through the standard `; charset=..` parameter. For instance, `text/plain; charset=utf-8`.

The mimetype value can take one of two forms:

1. The mimetype is the same between the original and modified files.

   If the mimetype is not changing (or the file is newly-added), then this will be a single value string.

   Listing 16: **Example**

   ```
   {
       "mimetype": "image/png"
   }
   ```

2. The mimetype has changed.

   If the mimetype has changed, then this should contain the following subkeys instead:

   `old` (string – *required*):
   The old mimetype of the file.

   `new` (string – *required*):
   The new mimetype of the file.

   Listing 17: **Example**

   ```
   {
       "mimetype": {
           "old": "text/plain; charset=utf-8",
           "new": "text/html; charset=utf-8"
       }
   }
   ```

op (string – *recommended*):

The operation performed on the file.

If not specified, this defaults to `modify`.

The following values are supported:

`create`:

The file is being created.

Listing 18: **Example**

```
{
    "op": "create",
    "path": "/src/main.py"
}
```

`delete`:

The file is being deleted.

Listing 19: **Example**

```
{
    "op": "delete",
    "path": "/src/compat.py"
}
```

`modify` (default):

The file or its permissions are being modified (but not renamed/copied/moved).

Listing 20: **Example**

```
{
    "op": "modify",
    "path": "/src/tests.py"
}
```

`copy`:

The file is being copied without modifications. The `path` key must have `old` and `new` values.

Listing 21: **Example**

```
{
    "op": "copy",
    "path": {
        "old": "/images/logo.png",
        "new": "/test-data/images/sample-image.png"
    }
}
```

`move`:

The file is being moved or renamed without modifications. The `path` key must have `old` and `new` values.

Listing 22: **Example**

```
{
    "op": "move",
```

(continues on next page)

(continued from previous page)

```
    "path": {
        "old": "/src/tests.py",
        "new": "/src/tests/test_utils.py"
    }
}
```

copy-modify:
The file is being copied with modifications. The `path` key must have `old` and `new` values.

Listing 23: **Example**

```
{
    "op": "copy-modify",
    "path": {
        "old": "/test-data/payload1.json",
        "new": "/test-data/payload2.json"
    }
}
```

move-modify:
The file is being moved with modifications. The `path` key must have `old` and `new` values.

Listing 24: **Example**

```
{
    "op": "move-modify",
    "path": {
        "old": "/src/utils.py",
        "new": "/src/encoding.py"
    }
}
```

path (string or dictionary – *required*):
The path of the file either within a repository a relative path on the filesystem.

If the file(s) are within a repository, this will be an absolute path.

If the file(s) are outside of a repository, this will be a relative path based on the parent of the files.

This can take one of two forms:

1. A single string, if both the original and modified file have the same path.
2. A dictionary, if the path has changed (renaming, moving, or copying a file).

   The dictionary would contain the following keys:

   old (string – *required*):
   The path to the original file.

   new (string – *required*):
   The path to the modified file.

This is often the same value used in the `---` line (though without any special prefixes like Git's `a/`). It may contain spaces, and must be in the encoding format used for the section.

This **must not** contain revision information. That should be supplied in *revision*.

Listing 25: **Example:** Modified file within a Subversion repository

```
{
    "path": "/trunk/myproject/README"
}
```

Listing 26: **Example:** Renamed file within a Git repository

```
{
    "path": {
        "old": "/src/README",
        "new": "/src/README.txt"
    }
}
```

Listing 27: **Example:** Renamed local file

```
{
    "path": {
        "old": "lib/test.c",
        "new": "tests/test.c"
    }
}
```

`revision` (dictionary – *recommended*):

Revision information for the file. This contains the following sub-keys:

Revisions are dependent on the type of source code management system. They may be numeric IDs, SHA1 hashes, or any other indicator normally used for the system.

The revision identifies the file, not the commit. In many systems (such as Subversion), these may the same identifier. In others (such as Git), they're separate.

`old` (string – *recommended*):

The old revision of the file, before any modifications are made.

This is required if modifying or deleting a file. Otherwise, it can be `null` or omitted.

If provided, the patch data must be able to be applied to the file at this revision.

`new` (string – *recommended*):

The new revision of the file after the patch has been applied.

This is optional, as it may not always be useful information, depending on the type of source code management system. Most will have a value to provide.

If a value is available, it should be added if modifying or creating a file. Otherwise, it can be `null` or omitted.

Listing 28: **Example:** Numeric revisions

```
{
    "path": "/src/main.py",
    "revision": {
        "old": "41",
        "new": "42"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Listing 29: **Example:** SHA1 revisions

```
{
    "path": "/src/main.py",
    "revision": {
        "old": "4f416cce335e2cf872f521f54af4abe65af5188a",
        "new": "214e857ee0d65bb289c976cb4f9a444b71f749b3"
    }
}
```

Listing 30: **Example:** Sample SCM-specific revision strings

```
{
    "path": "/src/main.py",
    "revision": {
        "old": "change12945",
        "new": "change12968"
    }
}
```

Listing 31: **Example:** Only an old revision is available

```
{
    "path": "/src/main.py",
    "revision": {
        "old": "8179510"
    }
}
```

`stats` **(dictionary –** ***recommended*****):**
A dictionary of statistics on the file.

This can be useful information to provide to diff analytics tools to help quickly determine how much of a file has changed.

`lines changed` **(integer –** ***recommended*****):**
The total number of lines changed in the file.

`insertions` **(integer –** ***recommended*****):**
The total number of insertions (+ lines) in the *File Diff sections*.

`deletions` **(integer –** ***recommended*****):**
The total number of deletions (- lines) in the *File Diff sections*.

`total lines` **(integer –** ***optional*****):**
The total number of lines in the file.

`similarity` **(string –** ***optional*****):**
The similarity percent between the old and new files (i.e., how much of the file remains the same). How this is calculated depends on the source code management system. This can include decimal places.

Listing 32: **Example**

```
{
    "path": "/src/main.py",
    "stats": {
        "total lines": 315,
        "lines changed": 35,
        "insertions": 22,
        "deletions": 3,
        "similarity": "98.89%"
    }
}
```

`symlink target` (string or dictionary – *optional*):

The target for a symlink (if *type* is set to `symlink`). Target paths are absolute on the filesystem, or relative to the symlink.

If modifying an existing symlink, but changing it to point to a new path, this will be a dictionary containing the following subkeys:

`old` (string – *required*):

The old target path.

`new` (string – *required*):

The new target path.

If adding a symlink, this will be a string containing the target path, or a dictionary with a `new` key. A single string is preferred over a dictionary in this case.

If deleting a symlink, this will be a string containing the target path, or a dictionary with an `old` key. A single string is preferred over a dictionary in this case.

If modifying an existing symlink, but keeping the target path it points to, this will be a string containing the target path, or a dictionary with `old` and `new` keys set to the same path. A single string is preferred over a dictionary in this case.

Listing 33: **Example:** Creating a symlink.

```
{
    "op": "create",
    "path": "/test-data/images",
    "type": "symlink",
    "symlink target": "static/images"
}

{
    "op": "create",
    "path": "/test-data/images",
    "type": "symlink",
    "symlink target": {
        "new": "static/images"
    }
}
```

Listing 34: **Example:** Deleting a symlink.

```
{
    "op": "delete",
    "path": "/test-data/fonts",
    "type": "symlink",
    "symlink target": "static/fonts"
}

{
    "op": "delete",
    "path": "/test-data/fonts",
    "type": "symlink",
    "symlink target": {
        "old": "static/fonts"
    }
}
```

Listing 35: **Example:** Changing a symlink's target.

```
{
    "op": "modify",
    "path": "/test-data/fonts",
    "type": "symlink",
    "symlink target": {
        "old": "assets/fonts",
        "new": "static/fonts"
    }
}
```

Listing 36: **Example:** Renaming a symlink.

```
{
    "op": "modify",
    "path": {
        "old": "/test-data/fonts",
        "new": "/data/fonts"
    },
    "type": "symlink",
    "symlink target": "static/fonts"
}

{
    "op": "modify",
    "path": {
        "old": "/test-data/fonts",
        "new": "/data/fonts"
    },
    "type": "symlink",
    "symlink target": {
        "old": "static/fonts",
        "new": "static/fonts"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

`type` (string – *recommended*):

The type of entry designated by the path. This may help parsers to provide better error or output information, or to give patchers a better sense of the kinds of changes they should expect to make.

`directory`:

The entry represents changes to a directory.

This will most commonly be used to change permissions on a directory.

Listing 37: **Example**

```
{
    "path": "/src",
    "type": "directory",
    "unix file mode": {
        "old": "0100700",
        "new": "0100755"
    }
}
```

`file` (default):

The entry represents a file. This is the default in diffs.

Listing 38: **Example**

```
{
    "path": "/src/main.py",
    "type": "file"
}
```

`symlink`:

The entry represents a symbolic link.

This should not include changes to the contents of the file, but is likely to include *symlink target* metadata.

Listing 39: **Example**

```
{
    "op": "create",
    "path": "/test-data/images",
    "type": "symlink",
    "symlink target": "static/images"
}
```

Custom types can be used if needed by the source code management system, though it will be up to them to process those types of changes.

All custom types should be in the form of *`vendor:type`*. For example, `svn:properties`.

`unix file mode` (octal or dictionary – *optional*):

The UNIX file mode information for the file or directory.

If adding a new file or directory, this will be a string containing the file mode.

If modifying a file or directory, this will be a dictionary containing the following subkeys:

`old` (string – *required*):
The original file mode in Octal format for the file (e.g., `"100644"`). This should be provided if modifying or deleting the file.

`new` (string– *required*):
The new file mode in Octal format for the file. This should be provided if modifying or adding the file.

Listing 40: **Example:** Changing a file's type

```
{
    "path": "/src/main.py",
    "unix file mode":{
        "old": "0100644",
        "new": "0100755"
    }
}
```

Listing 41: **Example:** Adding a file with permissions.

```
{
    "op": "create",
    "path": "/src/run-tests.sh",
    "unix file mode": "0100755"
}
```

Changed File Diff Section

Type: *Content Section*

If the file was added, modified, or deleted, the file diff section must contain a representation of those changes.

This is designated by a `#...diff:` section.

This section supports traditional text-based diffs and binary diffs (following the format used for Git binary diffs). The `type` option for the section is used to specify the diff type (`text` or `binary`), and defaults to `text` if unspecified (see the *options*) below.

Diff sections **must** end in a newline, in the section's encoding.

Text Diffs

For text diffs, the section contains the content people are accustomed to from a Unified Diff. These are the `---` and `+++` lines with the diff hunks.

For compatibility purposes, this may also include any additional data normally provided in that Unified Diff. For example, an `Index:` line, or Git's `diff --git` or CVS's `RCS file:`. This allows a DiffX file to be used by tools like **`git apply`** without breaking.

DiffX parsers should always use the metadata section, if available, over old-fashioned metadata in the diff section when processing a DiffX file.

Binary Diffs

The diff section may also include binary diff data. This follows Git's binary patch support, and may optionally include the Git-specific lines (`diff --git`, `index` and `GIT binary patch`) for compatibility.

To flag a binary diff section, add a `type=binary` option to the `#...diff:` section.

Note: Determine if the Git approach is correct.

This is still a work-in-progress. Git's binary patch support may be ideal, or there may be a better approach.

Options

This supports the *common content section options*, along with:

`type` (string – *optional*):
Indicates the content type of the section.

Supported types are:

`binary`:
This is a binary file.

`text` (default):
This is a text file. This is standard for diffs.

Listing 42: **Example**

```
#...diff: type=binary
delta 729
...
delta 224
...
```

Example

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...diff: length=642
--- README
+++ README
@@ -7,7 +7,7 @@
...
#..file:
#...diff: length=12364, type=binary
delta 729
...
delta 224
...
```

6.2.4 Encoding Rules

Historically, diffs have lacked any encoding information. A diff generated on one computer could use an encoding for diff content or filenames that would make it difficult to parse or apply on another computer.

To address this, DiffX has explicit support for encodings.

DiffX files follow these simple rules:

1. DiffX files have no default encoding. Tools *should* always *set an explicit encoding* (`utf-8` is **strongly recommended**).

   If not specified, all content must be treated as 8-bit binary data, and tools should be careful when assuming the encoding of any content. This is to match behavior with existing *Unified Diff* files.

2. *Section headers* are *always* encoded as ASCII (no non-ASCII content is allowed in headers).
3. Sections inherit the encoding of their parent section, unless overridden with the *encoding option*.
4. Preambles and metadata in *content sections* are encoded using their section's encoding.
5. *Diff sections* **do not** inherit their parent section's encoding, for compatibility with standard diff behavior. Instead, diff content should always be treated as 8-bit binary data, unless an explicit *encoding option* is defined for the section.

Tip: DiffX parsers should prioritize content (such as filenames) in metadata sections over scraping content in *diff sections*, in order to avoid encoding issues.

6.2.5 Example DiffX Files

Diff of Local File

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...meta: format=json, length=82
{
    "path": {
        "new": "message2.py",
        "old": "message.py"
    }
}
#...diff: length=692
--- message.py        2021-07-02 13:20:12.285875444 -0700
+++ message2.py        2021-07-02 13:21:31.428383873 -0700
@@ -164,10 +164,10 @@
             not isinstance(headers, MultiValueDict)):
             # Instantiating a MultiValueDict from a dict does not ensure that
             # values are lists, so we have to ensure that ourselves.
-            headers = MultiValueDict(dict(
-                (key, [value])
-                for key, value in six.iteritems(headers)
-            ))
+            headers = MultiValueDict({
```

(continues on next page)

(continued from previous page)

```
+                key: [value]
+                for key, value in headers.items()
+            })

         if in_reply_to:
             headers['In-Reply-To'] = in_reply_to
```

Diff of File in a Repository

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...meta: format=json, length=176
{
    "path": "/src/message.py",
    "revision": {
        "new": "f814cf74766ba3e6d175254996072233ca18a690",
        "old": "9f6a412b3aee0a55808928b43f848202b4ee0f8d"
    }
}
#...diff: length=631
--- a/src/message.py
+++ b/src/message.py
@@ -164,10 +164,10 @@
             not isinstance(headers, MultiValueDict)):
             # Instantiating a MultiValueDict from a dict does not ensure that
             # values are lists, so we have to ensure that ourselves.
-            headers = MultiValueDict(dict(
-                (key, [value])
-                for key, value in six.iteritems(headers)
-            ))
+            headers = MultiValueDict({
+                key: [value]
+                for key, value in headers.items()
+            })

         if in_reply_to:
             headers['In-Reply-To'] = in_reply_to
```

Diff of Commit in a Repository

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..preamble: indent=4, length=319, mimetype=text/markdown
    Convert legacy header building code to Python 3.

    Header building for messages used old Python 2.6-era list comprehensions
    with tuples rather than modern dictionary comprehensions in order to build
    a message list. This change modernizes that, and swaps out six for a
```

(continues on next page)

(continued from previous page)

```
    3-friendly `.items()` call.
#..meta: format=json, length=270
{
    "author": "Christian Hammond <christian@example.com>",
    "committer": "Christian Hammond <christian@example.com>",
    "committer date": "2021-06-02T13:12:06-07:00",
    "date": "2021-06-01T19:26:31-07:00",
    "id": "a25e7b28af5e3184946068f432122c68c1a30b23"
}
#..file:
#...meta: format=json, length=176
{
    "path": "/src/message.py",
    "revision": {
        "new": "f814cf74766ba3e6d175254996072233ca18a690",
        "old": "9f6a412b3aee0a55808928b43f848202b4ee0f8d"
    }
}
#...diff: length=629
--- /src/message.py
+++ /src/message.py
@@ -164,10 +164,10 @@
             not isinstance(headers, MultiValueDict)):
             # Instantiating a MultiValueDict from a dict does not ensure that
             # values are lists, so we have to ensure that ourselves.
-            headers = MultiValueDict(dict(
-                (key, [value])
-                for key, value in six.iteritems(headers)
-            ))
+            headers = MultiValueDict({
+                key: [value]
+                for key, value in headers.items()
+            })

         if in_reply_to:
             headers['In-Reply-To'] = in_reply_to
```

Diff of Multiple Commits in a Repository

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..preamble: indent=4, length=338, mimetype=text/markdown
    Pass extra keyword arguments in create_diffset() to the DiffSet model.

    The `create_diffset()` unit test helper function took a fixed list of
    arguments, preventing unit tests from passing in any other arguments
    to the `DiffSet` constructor. This now passes any extra keyword arguments,
    future-proofing this a bit.
#..meta: format=json, length=270
{
```

(continues on next page)

(continued from previous page)

```
    "author": "Christian Hammond <christian@example.com>",
    "committer": "Christian Hammond <christian@example.com>",
    "committer date": "2021-06-02T13:12:06-07:00",
    "date": "2021-06-01T19:26:31-07:00",
    "id": "a25e7b28af5e3184946068f432122c68c1a30b23"
}
#..file:
#...meta: format=json, length=185
{
    "path": "/src/testing/testcase.py",
    "revision": {
        "new": "eed8df7f1400a95cdf5a87ddb947e7d9c5a19cef",
        "old": "c8839177d1a5605aa60abe69db95c84183f0eebe"
    }
}
#...diff: length=819
--- /src/testing/testcase.py
+++ /src/testing/testcase.py
@@ -498,7 +498,7 @@ class TestCase(FixturesCompilerMixin, DjbletsTestCase):
             **kwargs)

     def create_diffset(self, review_request=None, revision=1, repository=None,
-                       draft=False, name='diffset'):
+                       draft=False, name='diffset', **kwargs):
         """Creates a DiffSet for testing.

         The DiffSet defaults to revision 1. This can be overriden by the
@@ -513,7 +513,8 @@ class TestCase(FixturesCompilerMixin, DjbletsTestCase):
             name=name,
             revision=revision,
             repository=repository,
-            diffcompat=DiffCompatVersion.DEFAULT)
+            diffcompat=DiffCompatVersion.DEFAULT,
+            **kwargs)

         if review_request:
             if draft:
#.change:
#..preamble: indent=4, length=219, mimetype=text/markdown
    Set a diff description when creating a DiffSet in chunk generator tests.

    This makes use of the new `**kwargs` support in `create_diffset()` in
    a unit test to set a description of the diff, for testing.
#..meta: format=json, length=270
{
    "author": "Christian Hammond <christian@example.com>",
    "committer": "Christian Hammond <christian@example.com>",
    "committer date": "2021-06-02T19:13:08-07:00",
    "date": "2021-06-02T14:19:45-07:00",
    "id": "a25e7b28af5e3184946068f432122c68c1a30b23"
}
#..file:
```

(continues on next page)

(continued from previous page)

```
#...meta: format=json, length=211
{
    "path": "/src/diffviewer/tests/test_diff_chunk_generator.py",
    "revision": {
        "new": "a2ccb0cb48383472345d41a32afde39a7e6a72dd",
        "old": "1b7af7f97076effed5db722afe31c993e6adbc78"
    }
}
#...diff: length=662
--- a/src/diffviewer/tests/test_diff_chunk_generator.py
+++ b/src/diffviewer/tests/test_diff_chunk_generator.py
@@ -66,7 +66,8 @@ class DiffChunkGeneratorTests(SpyAgency, TestCase):
         super(DiffChunkGeneratorTests, self).setUp()

         self.repository = self.create_repository(tool_name='Test')
-        self.diffset = self.create_diffset(repository=self.repository)
+        self.diffset = self.create_diffset(repository=self.repository,
+                                           description=self.diff_description)
         self.filediff = self.create_filediff(diffset=self.diffset)
         self.generator = DiffChunkGenerator(None, self.filediff)
#..file:
#...meta: format=json, length=200
{
    "path": "/src/diffviewer/tests/test_diffutils.py",
    "revision": {
        "new": "0d4a0fb8d62b762a26e13591d06d93d79d61102f",
        "old": "be089b7197974703c83682088a068bef3422c6c2"
    }
}
#...diff: length=567
--- a/src/diffviewer/tests/test_diffutils.py
+++ b/src/diffviewer/tests/test_diffutils.py
@@ -258,7 +258,8 @@ class BaseFileDiffAncestorTests(SpyAgency, TestCase):
                     owner=Repository,
                     call_fake=lambda *args, **kwargs: True)

-        self.diffset = self.create_diffset(repository=self.repository)
+        self.diffset = self.create_diffset(repository=self.repository,
+                                           description='Test Diff')

         for i, diff in enumerate(self._COMMITS, 1):
             commit_id = 'r%d' % i
```

Wrapped Git Diff

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..preamble: length=352
commit 89a3a4ab76496079f3bb3073b3a04aacaa8bbee4
Author: Christian Hammond <christian@example.com>
Date:   Wed Jun 2 19:13:08 2021 -0700

    Set a diff description when creating a DiffSet in chunk generator tests.

    This makes use of the new `**kwargs` support in `create_diffset()` in
    a unit test to set a description of the diff, for testing.
#..meta: format=json, length=270
{
    "author": "Christian Hammond <christian@example.com>",
    "committer": "Christian Hammond <christian@example.com>",
    "committer date": "2021-06-02T19:13:08-07:00",
    "date": "2021-06-02T14:19:45-07:00",
    "id": "a25e7b28af5e3184946068f432122c68c1a30b23"
}
#..file:
#...meta: format=json, length=211
{
    "path": "/src/diffviewer/tests/test_diff_chunk_generator.py",
    "revision": {
        "new": "a2ccb0cb48383472345d41a32afde39a7e6a72dd",
        "old": "1b7af7f97076effed5db722afe31c993e6adbc78"
    }
}
#...diff: length=814
diff --git a/src/diffviewer/tests/test_diff_chunk_generator.py
index 1b7af7f97076effed5db722afe31c993e6adbc78..a2ccb0cb48383472345d41a32afde39a7e6a72dd
--- a/src/diffviewer/tests/test_diff_chunk_generator.py
+++ b/src/diffviewer/tests/test_diff_chunk_generator.py
@@ -66,7 +66,8 @@ class DiffChunkGeneratorTests(SpyAgency, TestCase):
         super(DiffChunkGeneratorTests, self).setUp()

         self.repository = self.create_repository(tool_name='Test')
-        self.diffset = self.create_diffset(repository=self.repository)
+        self.diffset = self.create_diffset(repository=self.repository,
+                                           description=self.diff_description)
         self.filediff = self.create_filediff(diffset=self.diffset)
         self.generator = DiffChunkGenerator(None, self.filediff)
```

Wrapped CVS Diff

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...meta: format=json, length=94
{
    "path": "/readme",
    "revision": {
        "new": "1.2",
        "old": "1.1"
    }
}
#...diff: length=320
Index: readme
===================================================================
RCS file: /cvsroot/readme,v
retrieving version 1.1
retrieving version 1.2
diff -u -p -r1.1 -r1.2
--- readme    26 Jan 2016 16:29:12 -0000        1.1
+++ readme    31 Jan 2016 11:54:32 -0000        1.2
@@ -1 +1,3 @@
 Hello there
+
+Oh hi!
```

Wrapped Subversion Property Diff

```
#diffx: encoding=utf-8, version=1.0
#.change:
#..file:
#...meta: format=json, length=269
{
    "path": "/readme",
    "revision": {
        "old": "123"
    },
    "svn": {
        "properties": {
            "myproperty": {
                "new": "new value",
                "old": "old value"
            }
        }
    },
    "type": "svn:properties"
}
#...diff: length=266
Index: readme
===================================================================
```

(continues on next page)

(continued from previous page)

```
--- (revision 123)
+++ (working copy)
Property changes on: .
___________________________________________________________________
Modified: myproperty
## -1 +1 ##
-old value
+new value
```

6.3 pydiffx

pydiffx is a Python implementation of the *DiffX specification*.

DiffX is a proposed specification for a structured version of *Unified Diffs* that contains metadata, standardized parsing, multi-commit diffs, and binary diffs, in a format compatible with existing diff parsers. *Learn more about DiffX*.

This module is a reference implementation designed to make it easy to read and write DiffX files in any Python application.

6.3.1 Compatibility

- Python 2.7
- Python 3.6
- Python 3.7
- Python 3.8
- Python 3.9
- Python 3.10
- Python 3.11

6.3.2 Installation

pydiffx can be installed on Python 2.7 and 3.6+ using **pip**:

```
pip install -U pydiffx
```

6.3.3 Using pydiffx

DiffX files can be managed through one of two sets of interfaces:

- A streaming reader (*pydiffx.reader.DiffXReader*) and writer (*pydiffx.writer.DiffXWriter*) for progressively working with DiffX files of any size.
- A DiffX Object Model (*pydiffx.dom.objects.DiffX*) for treating DiffX files as a mutable data structure.

To get familiar with these interfaces, follow our tutorials:

- *Writing DiffX Files using DiffXWriter*

Tutorials

Writing DiffX Files using DiffXWriter

pydiffx.writer.DiffXWriter is a low-level class for incrementally writing DiffX files to a stream (such as a file, an HTTP response, or as input to another process).

When using this writer, the caller is responsible for ensuring that all necessary metadata or other content is correct and present. Errors cannot be caught up-front, and any failures may cause a failure to write mid-stream.

Step 1. Create the Writer

To start, construct an instance of *pydiffx.writer.DiffXWriter* and point it to an open byte stream. This will immediately write the main DiffX header to the stream.

Important: Make sure you're writing to a byte stream! If the stream expects Unicode content, you will encounter failures when writing.

```
from pydiffx import DiffXWriter

with open('outfile.diff', 'wb') as fp:
    writer = DiffXWriter(fp)

    ...
```

Once you've set up the writer, you can optionally add a *preamble* and/or *metadata* section (in that order), followed by your first (required) *change section*.

Step 2. Write a Preamble (Optional)

Preamble sections are free-form text that describe an overall set of changes. The main DiffX section (which you're writing right now) can have a preamble that describes the entirety of all changes made in the entire DiffX file

---

Tip: This would be a good spot for a merge commit message or a review request or pull request description.

The preamble can be written using *writer.write_preamble()*:

```
writer.write_preamble(
    'Here is a summary of the set of changes in this DiffX file.\n'
    '\n'
    'And here would be the multi-line description!')
```

Preamble text is considered to be plain text by default. If this instead represents Markdown-formatted text, you'll want to specify that using the `mimetype` parameter, like so:

```
from pydiffx import PreambleMimeType

...
```

(continues on next page)

(continued from previous page)

```
writer.write_preamble(
    'This is some Markdown text.\n'
    '\n'
    'You can tell because of the **bold** and the '
    '[links](https://example.com).',
    mimetype=PreambleMimeType.MARKDOWN)
```

A few additional things to note:

1. The preamble will be encoded using UTF-8 (assuming a different encoding was set up when creating the writer).
2. the written text will be indented 4 spaces (which avoids issues with user-provided preamble text conflicting with other parts of the DiffX file).
3. The line endings are going to be consistent throughout the text, as either UNIX (LF – `\n`) or DOS (CRLF – `\r\n`) line endings.

All of these can be overridden when writing by using the optional parameters to *DiffXWriter.write_preamble*.

Step 3. Write Metadata (Optional)

Metadata sections contain information in JSON form that parsers can use to determine, for instance, where a diff would apply, or which repository a diff pertains to. See the *main metadata section documentation* for the kind of information you would put here.

The metadata can be written using *writer.write_meta*:

```
writer.write_meta({
    'stats': {
        'changes': 1,
        'files': 2,
        'insertions': 27,
        'deletions': 5,
    }
})
```

While any metadata can go in here, we **strongly recommend** putting anything specific to your tool or revision control system under a key that's unique to your tool. For example, custom Git data might be under a `git` key.

Step 4. Begin a New Change

DiffX files must have at least one *Change section*. These contain an optional preamble and/or metadata, and one or more modified files.

To start writing a new Change section:

```
writer.new_change()
```

Note: If representing multiple commits, you're going to end up calling this once per commit, but only after you've finished writing all the *File sections* under this change.

To write a change's preamble or metadata, just use the same functions shown above, and they'll be part of this new section.

See the information on *Change Preamble Sections* and *Change Metadata Sections* for what should go here.

Step 5. Begin a New File

You can now start writing *File sections*, one per file in the change.

To start writing a new File section:

```
writer.new_file()
```

File sections require a *File Metadata section*, which must contain information identifying the file being changed. They *do not* contain a preamble section.

Step 6. Write a File's Diff (Optional)

If there are changes made to the contents of the file, you'll need to write a *File Diff section*.

This will contain a byte string of the diff content, which may be a plain *Unified Diff*, or it may wrap a full diff variant, such as a Git-style diff.

To write the diff:

```
writer.write_diff(
    b'--- src/main.py\t2021-07-13 16:40:05.442067927 -0800\n'
    b'+++ src/main.py\n2021-07-17 22:22:27.834102484 -0800\n'
    b'@@ -120,6 +120,6 @@\n'
    b'     verbosity = options["verbosity"]\n'
    b'\n'
    b'     if verbosity > 0:\n'
    b'-        print("Starting the build...")\n'
    b'+        logging.info("Starting the build...")\n'
    b'\n'
    b'     start_build(**options)\n'
    b'\n'
)
```

Or, if we're dealing with a Git-style diff, it might look like:

```
writer.write_diff(
    b'diff --git a/src/main.py b/src/main.py\n'
    b'index aba891f..cc52f7 100644\n'
    b'--- a/src/main.py\n'
    b'+++ b/src/main.py\n'
    b'@@ -120,6 +120,6 @@\n'
    b'     verbosity = options["verbosity"]\n'
    b'\n'
    b'     if verbosity > 0:\n'
    b'-        print("Starting the build...")\n'
    b'+        logging.info("Starting the build...")\n'
    b'\n'
    b'     start_build(**options)\n'
    b'\n'
)
```

Note: The DiffX specification does not define the format of these diffs.

It is completely okay to wrap another diff variant in here, and necessary if you need an existing parser to extract variant-specific information from the file.

There are some **really** useful options you can provide to help parsers better understand and process this diff:

- Pass `encoding=...` if you know the encoding of the file.

  This will help DiffX-compatible tools process the file contents correctly, normalizing it for the local filesystem or the contents coming from a repository.

  This is **strongly recommended**, and one of the major benefits to representing changes as a DiffX file.

- Pass `line_endings=` if you know for sure that this file is intended to use UNIX (LF – `\n`) or DOS (CRLF – `\r\n`) line endings.

  This is **strongly recommended**, and will help parsers process the file if there's a mixture of line endings. This is a real-world problem, as some source code repositories contain, for example, `\r\n` as a line ending but `\n` as a regular character in the file.

  You can use either *`LineEndings.UNIX`* or *`LineEndings.DOS`* as values.

Step 7: Rinse and Repeat

You've now written a file! Bet that feels good.

You can now go back to *Step 5. Begin a New File* to write a new file in the Change section, or go back to *Step 4. Begin a New Change* to write a new change full of files.

Once you're done, close the stream. Your DiffX file was written!

Putting It All Together

Let's look at an example tying together everything we've learned:

```
from pydiffx import DiffXWriter, LineEndings, PreambleMimeType

with open('outfile.diff', 'wb') as fp:
    writer = DiffXWriter(fp)
    writer.write_preamble(
        '89e6c98d92887913cadf06b2adb97f26cde4849b'

        'This file makes a bunch of changes over a couple of commits.\n'
        '\n'
        'And we are using **Markdown** to describe it.',
        mimetype=PreambleMimeType.MARKDOWN)
    writer.write_meta({
        'stats': {
            'changes': 1,
            'files': 2,
            'insertions': 3,
            'deletions': 2,
        }
```

(continues on next page)

(continued from previous page)

```
})

writer.new_change()
writer.write_preamble('Something very enlightening about commit #1.')
writer.write_meta({
    'author': 'Christian Hammond <christian@example.com>',
    'id': 'a25e7b28af5e3184946068f432122c68c1a30b23',
    'date': '2021-07-17T19:26:31-07:00',
    'stats': {
        'files': 2,
        'insertions': 2,
        'deletions': 2,
    },
})

writer.new_file()
writer.write_meta({
    'path': 'src/main.py',
    'revision': 'revision': {
        'old': '3f786850e38755 0fdab836ed7e6dc881de23001b',
        'new': '89e6c98d92887913cadf06b2adb97f26cde4849b',
    },
    'stats': {
        'lines': 1,
        'insertions': 1,
        'deletions': 1,
    },
})
writer.write_diff(
    b'--- src/main.py\n'
    b'+++ src/main.py\n'
    b'@@ -120,6 +120,6 @@\n'
    b'     verbosity = options["verbosity"]\n'
    b'\n'
    b'     if verbosity > 0:\n'
    b'-        print("Starting the build...")\n'
    b'+        logging.info("Starting the build...")\n'
    b'\n'
    b'     start_build(**options)\n'
    b'\n',
    encoding='utf-8',
    line_endings=LineEndings.UNIX)

# And so on...
writer.new_file()
writer.write_meta(...)
writer.write_diff(...)

writer.new_change()
writer.write_preamble(...)
writer.write_meta(...)
```

(continues on next page)

(continued from previous page)

```
    writer.new_file()
    writer.write_meta(...)
    writer.write_diff(...)
```

That's not so bad, right? Sure beats a bunch of `print` statements.

Now that you know how to write a DiffX file, you can begin integrating *pydiffx* into your codebase. *We'll be happy to list you as a DiffX user!*

6.3.4 Documentation

Module and Class References

pydiffx	An implementation of DiffX, an extensible, structured Unified Diff format.
pydiffx.dom	The DiffX Object Model and high-level reader/writer.
pydiffx.dom.objects	The DiffX Object Model.
pydiffx.dom.reader	Reader for parsing a DiffX file into DOM objects.
pydiffx.dom.writer	Writer for generating a DiffX file from DOM objects.
pydiffx.errors	Common errors for parsing and generating diffs.
pydiffx.options	Constants and utilities for options.
pydiffx.reader	A streaming reader for DiffX files.
pydiffx.sections	Section-related definitions.
pydiffx.utils	
pydiffx.utils.text	Utilities for processing text.
pydiffx.utils.unified_diffs	Utilities for parsing Unified Diffs.
pydiffx.writer	A streaming writer for DiffX files.

pydiffx

An implementation of DiffX, an extensible, structured Unified Diff format.

pydiffx makes it easy to work with DiffX files. This is a proposed standard for a new diff format that can contain structured metadata, formal parsing rules, multiple commits and binary files, while remaining backwards-compatible with existing Unified Diff parsers.

This module is the starting point for using pydiffx, and contains several convenience imports:

DiffX	Representation of a DiffX file.
DiffType	Types available for a diff.
LineEndings	Line ending types available for a content section.
MetaFormat	Formats available for a meta section.
PreambleMimeType	Mimetypes available for a preamble section.
DiffXReader	A streaming reader for DiffX files.
DiffXWriter	A streaming writer for DiffX files.

pydiffx.dom

The DiffX Object Model and high-level reader/writer.

The DiffX Object Model makes it easy to create or process DiffX files. They can be constructed up-front and then written out, or loaded from data into a series of objects for processing.

This module provides convenience imports for the DiffX Object Model:

DiffX	Representation of a DiffX file.

Consumers will want to start with the documentation for *DiffX*.

pydiffx.dom.objects

The DiffX Object Model.

This is a set of classes that comprise the DiffX Object Model. These consistent of container and content section classes, each with type-safe properties used to manage content and options for the DiffX file.

The only object that should be created manually is *DiffX*. The others are created automatically or when calling *DiffX.add_change()* or *DiffXChangeSection.add_file()*.

Classes

BaseDiffXContainerSection([parent_section])	Base class for container sections.
BaseDiffXContentSection(**kwargs)	Base class for content sections.
BaseDiffXSection([parent_section])	Base class for a DiffX section.
DiffX([parent_section])	Representation of a DiffX file.
DiffXChangeSection([parent_section])	A change section within a DiffX file.
DiffXFileDiffSection(**kwargs)	A diff content section.
DiffXFileSection([parent_section])	A file section within a change section.
DiffXMetaSection(**kwargs)	A metadata section.
DiffXPreambleSection(**kwargs)	A preamble section.

class pydiffx.dom.objects.**BaseDiffXSection**(*parent_section=None*, ***attrs*)

Bases: object

Base class for a DiffX section.

This manages option storage and controls the initialization process for the subclass.

options

The options set for this section. This can be manipulated directly without any type checking, but it's recommended that consumers go through the dedicated class-level attributes.

Type
dict

section_id

The ID of this section. This corresponds to a value in *Section*.

Type
unicode

section_name = None

The name of the section.

This must be provided by subclasses.

Type
unicode

default_options = {}

Default options to set for the section.

These will be written to *options* when constructing the section if not otherwise provided by the caller.

Type
dict

__init__(*parent_section=None*, ***attrs*)

Initialize the section.

Parameters

- **parent_section** (*BaseDiffXContainerSection*, *optional*) – The parent container section.
- ****attrs** (dict) – Attributes to set for the section.

  This may consist of attributes representing options or content subsections.

  Any invalid option will raise a *DiffXUnknownOptionError*.

options

section_id

__eq__(*other*)

Return whether this section is equal to another section.

Parameters
other (*BaseDiffXSection*) – The section to compare to.

Returns
`True` if the two sections are equal. `False` if they are not.

Return type
bool

__repr__()

Return a string representation of this section.

Returns
The string representation.

Return type
unicode

__hash__ = None

class pydiffx.dom.objects.**BaseDiffXContainerSection**(*parent_section=None*, ***attrs*)

Bases: *BaseDiffXSection*

Base class for container sections.

Container sections contain additional container and/or content sections. They're also responsible for setting options on the content sections.

Subclasses must explicitly set *subsections*.

subsections

The list of subsections in this section.

Type
list of *BaseDiffXSection*

__eq__(*other*)

Return whether this section is equal to another section.

Parameters
other (*BaseDiffXSection*) – The section to compare to.

Returns
True if the two sections are equal. False if they are not.

Return type
bool

__iter__()

Iterate through the immediate subsections.

Yields
BaseDiffXSection – A subsection of this section.

subsections

__hash__ = None

class pydiffx.dom.objects.**BaseDiffXContentSection**(***kwargs*)

Bases: *BaseDiffXSection*

Base class for content sections.

Content sections contain data in some form, indicated by *data_type*.

They cannot have subsections of their own.

Consumers will generally not need to access content sections directly. Instead, they'll set data or options through the parent container class's type-safe attributes, or during construction of the parent section.

data_type = None

The type of data allowed for this section.

Type
type

default_value = None

Default value for the section.

Type
object

__init__(***kwargs*)

Initialize the section.

Parameters
****kwargs** (dict) – Keyword arguments to pass to the parent. See the documentation for details.

property content

The content of this section.

The type will be that of *data_type*.

__eq__(*other*)

Return whether this section is equal to another section.

Parameters
other (*BaseDiffXSection*) – The section to compare to.

Returns
`True` if the two sections are equal. `False` if they are not.

Return type
bool

__hash__ = None

class pydiffx.dom.objects.**DiffX**(*parent_section=None*, ***attrs*)

Bases: ContainerOptionsMixin, MetaOptionsMixin, PreambleOptionsMixin, *BaseDiffXContainerSection*

Representation of a DiffX file.

This represents a DiffX file as a hierarchical series of objects and attributes. It can be used to construct a new DiffX file piece-by-piece before writing it out to a file or stream, or to read in an existing DiffX file for processing or manipulation.

Consumers will start by working directly with a *DiffX* instance.

When constructing one, they'll need to add at least one change by using *add_change()*, and at least one file to that change.

When reading one, they can read the preamble, metadata, or list of files using the provided attributes.

encoding

The default encoding for all preamble and metadata sections in the DiffX file. This may be `None`, in which case an encoding cannot be assumed.

Changing this will not affect the in-memory representation of any data, but it will affect how it's written.

Type
unicode

meta

Global metadata for the entire DiffX file.

Type
dict

meta_encoding

Encoding used when reading/writing the metadata in the file.

See *DiffXMetaSection.encoding*.

Type
unicode

meta_section

The actual metadata section. This will generally not be accessed directly.

Type
DiffXMetaSection

preamble

The preamble content describing the entire series of changes in the DiffX file.

Type
unicode

preamble_encoding

Encoding used when reading/writing the preamble content in the file.

See *DiffXPreambleSection.encoding*.

Type
unicode

preamble_indent

Indentation applied to each line of preamble content.

See *DiffXPreambleSection.indent*.

Type
int

preamble_line_endings

The type of line endings used in the preamble content.

See *DiffXPreambleSection.line_endings*.

Type
unicode

preamble_mimetype

The mimetype representing the format of the preamble content.

See *DiffXPreambleSection.mimetype*.

Type
unicode

preamble_section

The actual preamble section. This will generally not be accessed directly.

Type
DiffXPreambleSection

default_options = {'encoding': 'utf-8', 'version': '1.0'}

Default options to set for the section.

These will be written to `options` when constructing the section if not otherwise provided by the caller.

Type
dict

version

The version of the DiffX file.

Only supported versions can be set.

Type
unicode

classmethod from_bytes(*data*)

Construct an instance from a DiffX file stored in a byte string.

Parameters
data (bytes) – The DiffX file contents to parse.

Returns
The resulting DiffX instance.

Return type
DiffX

Raises
pydiffx.errors.DiffXParseError – The DiffX contents could not be parsed. Details will be in the error message.

classmethod from_stream(*stream*)

Construct an instance from a DiffX file read from a stream.

This will close the stream after it's been read.

Parameters
data (file or io.IOBase) – The stream to read from.

Returns
The resulting DiffX instance.

Return type
DiffX

Raises
pydiffx.errors.DiffXParseError – The DiffX contents could not be parsed. Details will be in the error message.

property subsections

A list of the preamble, meta, and change subsections.

Type
list of *BaseDiffXSection*

add_change(***attrs*)

Add a new change section.

Parameters
****attrs** (dict) – Attributes to set on the section. This may consist of any attributes listed on *DiffXChangeSection*.

Returns
The newly-added change section.

Return type
DiffXChangeSection

Raises
pydiffx.errors.DiffXUnknownOptionError – One or more attribute names are invalid.

generate_stats()

Generate statistics for the DiffX metadata.

This will gather statistics on the number of changes, files, insertions, deletions, and total lines changed.

This should only be run once the diff is complete, before writing it.

to_bytes()

Write and return the DiffX file contents.

Returns
The DiffX file contents.

Return type
bytes

Raises
pydiffx.errors.BaseDiffXError – There was an error generating the content.

changes

meta_section

preamble_section

class pydiffx.dom.objects.**DiffXChangeSection**(*parent_section=None*, ***attrs*)

Bases: ContainerOptionsMixin, MetaOptionsMixin, PreambleOptionsMixin, *BaseDiffXContainerSection*

A change section within a DiffX file.

A change represents a set of changes to files, possibly backed by a commit.

Changes can be added through *DiffX.add_change()*.

encoding

The default encoding for preamble and metadata sections anywhere under this section.

This may be None, in which case the parent *DiffX.encoding* value will be used.

Changing this will not affect the in-memory representation of any data, but it will affect how it's written.

Type
unicode

meta

Metadata for the change.

Type
dict

meta_encoding

Encoding used when reading/writing the metadata in the file.

See *DiffXMetaSection.encoding*.

Type
unicode

meta_section

The actual metadata section. This will generally not be accessed directly.

Type
DiffXMetaSection

preamble

The preamble content describing the change.

Type
unicode

preamble_encoding

Encoding used when reading/writing the preamble content in the file.

See *DiffXPreambleSection.encoding*.

Type
unicode

preamble_indent

Indentation applied to each line of preamble content.

See *DiffXPreambleSection.indent*.

Type
int

preamble_line_endings

The type of line endings used in the preamble content.

See *DiffXPreambleSection.line_endings*.

Type
unicode

preamble_mimetype

The mimetype representing the format of the preamble content.

See *DiffXPreambleSection.mimetype*.

Type
unicode

preamble_section

The actual preamble section. This will generally not be accessed directly.

Type
DiffXPreambleSection

section_name = 'change'

The name of the section.

This must be provided by subclasses.

Type
unicode

property subsections

A list of the preamble, meta, and file subsections.

Type
list of *BaseDiffXSection*

add_file(***attrs*)

Add a new file section.

Parameters
****attrs** (dict) – Attributes to set on the section. This may consist of any attributes listed on *DiffXFileSection*.

Returns
The newly-added change section.

Return type
DiffXFileSection

Raises
pydiffx.errors.DiffXUnknownOptionError – One or more attribute names are invalid.

generate_stats()

Generate statistics for the change section's metadata.

This will gather statistics on the number of files, insertions, deletions, and total lines changed.

This should only be run once the change is complete. Normally, callers will want to call *DiffX.generate_stats()* instead.

meta_section

preamble_section

files

class pydiffx.dom.objects.**DiffXFileSection**(*parent_section=None*, ***attrs*)

Bases: ContainerOptionsMixin, DiffOptionsMixin, MetaOptionsMixin, *BaseDiffXContainerSection*

A file section within a change section.

A file represents a change to a particular file. This may be a change to the file contents, or just to the metadata of the file.

Metadata must always provide sufficient information for identifying and working with the file without having to parse the embedded diff.

Files can be added through *DiffXChangeSection.add_file()*.

diff

The file's Unified Diff contents.

This may be a plain Unified Diff, or it may be a vendor-specific variant (such as a Git-style diff).

Type
bytes

diff_encoding

The encoding of the diff content.

See *DiffXFileDiffSection.encoding*.

Type
unicode

diff_line_endings

The identifier for the type of line endings (DOS or UNIX) separating each line of the diff content.

See *DiffXFileDiffSection.line_endings*.

Type
unicode

diff_section

The actual diff section. This will generally not be accessed directly.

Type
DiffXFileDiffSection

diff_type

The type of the diff (text or binary).

See *DiffXFileDiffSection.type*.

Type
unicode

encoding

The default encoding for this section.

This is sort of redundant with *meta_encoding*, as the metadata is the only content section affected by this encoding. However, it's here for consistency and future expansion.

This may be `None`, in which case the parent `DiffXChange.encoding` value will be used.

Changing this will not affect the in-memory representation of any data, but it will affect how it's written.

Type
unicode

meta

Metadata for the file.

Type
dict

meta_encoding

The encoding used when reading/writing the metadata content in the file.

See *DiffXMetaSection.encoding*.

Type
unicode

meta_section

The actual metadata section. This will generally not be accessed directly.

Type
DiffXMetaSection

section_name = 'file'

The name of the section.

This must be provided by subclasses.

Type
unicode

generate_stats()

Generate statistics for the file section's metadata.

This will gather statistics on the number of insertions, deletions, and total lines changed.

Note that if the content in *diff* has a parse error, the data may be incorrect.

This should only be run once the change is complete. Normally, callers will want to call *DiffX.generate_stats()* instead.

diff_section

meta_section

class pydiffx.dom.objects.**DiffXPreambleSection**(***kwargs*)

Bases: *BaseDiffXContentSection*

A preamble section.

The contents and options for this section will generally be accessed through the parent section's attributes.

section_name = 'preamble'

The name of the section.

This must be provided by subclasses.

Type
unicode

data_type

alias of str

encoding

The encoding used when reading/writing the preamble content.

Changing this will not affect the in-memory representation of the preamble, but it will affect how it's written.

If None, the encoding option of the section is used instead.

Type
unicode

indent

The indentation applied to each line of the preamble content.

This will be added to the beginning of each encoded line of the preamble when reading/writing the preamble content in the file.

Changing this will not affect the in-memory representation of the preamble, but it will affect how it's written.

If None, no indentation will be applied.

It's recommended to use an indentation of 4, to ensure preamble content does not impact parsing.

Type
int

line_endings

The type of line endings used in the preamble content.

Valid values are defined in *LineEndings*.

This should be explicitly set if the type of line endings are known, as a hint to parsers.

If None, parsers will need to carefully handle newline detection based on their needs.

Type
unicode

mimetype

The mimetype representing the format of the preamble content.

This can help consumers render the preamble content the way it was meant to be seen.

Valid values are defined in *PreambleMimeType*.

If None, the preamble content is assumed to be plain text.

Type
unicode

class pydiffx.dom.objects.**DiffXMetaSection**(***kwargs*)

Bases: *BaseDiffXContentSection*

A metadata section.

The contents and options for this section will generally be accessed through the parent section's attributes.

default_options = {'format': 'json'}

Default options to set for the section.

These will be written to `options` when constructing the section if not otherwise provided by the caller.

Type
dict

section_name = 'meta'

The name of the section.

This must be provided by subclasses.

Type
unicode

data_type

alias of `dict`

default_value = {}

Default value for the section.

Type
object

encoding

The encoding used when reading/writing the metadata content in the file.

Changing this will not affect the in-memory representation of the metadata, but it will affect how it's written.

If `None`, the section's `encoding` will be used instead.

Type
unicode

format

The metadata format used when reading/writing the content in the file.

This is available for future expansion. For now, it will always be *JSON*.

Type
unicode

class pydiffx.dom.objects.**DiffXFileDiffSection**(***kwargs*)

Bases: *BaseDiffXContentSection*

A diff content section.

The contents and options for this section will generally be accessed through the parent section's attributes.

section_name = 'diff'

The name of the section.

This must be provided by subclasses.

Type
unicode

data_type

alias of bytes

encoding

The encoding of the diff content.

This _does **not**_ inherit from any other section's encoding. It must be explicitly provided for an encoding to be set.

It's recommended that diff generators set this if they know the encoding of the file being changed.

If None, no encoding can be assumed.

Type
unicode

line_endings

The type of line endings used in the diff content.

Valid values are defined in *LineEndings*.

This should be explicitly set if the type of line endings are known, as a hint to parsers. Diffs may legitimately contain newline characters of an alternate type that are not intended to be interpreted as newlines. This hint can help avoid issues parsing those diffs.

If None, parsers will need to carefully handle newline detection based on their needs.

Type
unicode

type

The type of the diff (text or binary).

Valid values are defined in *DiffType*.

Type
unicode

pydiffx.dom.reader

Reader for parsing a DiffX file into DOM objects.

Classes

DiffXDOMReader(diffx_cls)	A reader for parsing a DiffX file into DOM objects.

class pydiffx.dom.reader.**DiffXDOMReader**(*diffx_cls*)

Bases: object

A reader for parsing a DiffX file into DOM objects.

This will construct a *DiffX* from an input byte stream, such as a file, HTTP response, or memory-backed stream.

Often, you won't use this directly. Instead, you'll call *DiffXFile.from_stream()* or *DiffXFile.from_bytes()*.

If constructing manually, one instance can be reused for multiple streams.

diffx_cls

The *DiffX* class or subclass to create when parsing.

Type
type

reader_cls

The class to instantiate for reading from a stream.

Subclasses can set this if they need to use a more specialized reader.

Type
type

alias of *DiffXReader*

__init__(*diffx_cls*)

Initialize the reader.

Parameters
diffx_cls (type) – The *DiffX* class or subclass to create when parsing.

parse(*stream*)

Parse a stream and construct the DOM objects.

The stream will be closed after reading.

Parameters
stream (file or io.IOBase) – The byte stream containing a valid DiffX file.

Returns
The resulting DiffX instance.

Return type
pydiffx.dom.objects.DiffX

Raises
pydiffx.errors.DiffXParseError – The DiffX contents could not be parsed. Details will be in the error message.

pydiffx.dom.writer

Writer for generating a DiffX file from DOM objects.

Classes

DiffXDOMWriter()	A writer for generating a DiffX file from DOM objects.

class pydiffx.dom.writer.**DiffXDOMWriter**

Bases: object

A writer for generating a DiffX file from DOM objects.

This will write a *DiffX* object tree to a byte stream, such as a file, HTTP response, or memory-backed stream.

If constructing manually, one instance can be reused for multiple DiffX objects.

writer_cls

The class to instantiate for writing to a stream.

Subclasses can set this if they need to use a more specialized writer.

Type
type

alias of *DiffXWriter*

write_stream(*diffx*, *stream*)

Write a DiffX object to a stream.

Parameters

- **diffx** (*pydiffx.dom.objects.DiffX*) – The DiffX object to write.
- **stream** (file or io.IOBase) – The byte stream to write to.

Raises

pydiffx.errors.BaseDiffXError – The DiffX contents could not be written. Details will be in the error message.

pydiffx.errors

Common errors for parsing and generating diffs.

Exceptions

BaseDiffXError	Base class for all DiffX errors.
DiffXContentError	An error with content for a section.
DiffXOptionValueChoiceError(option, value, ...)	An error with the choice for a value for an option.
DiffXOptionValueError	An error with a value for an option.
DiffXParseError(msg, linenum[, column])	An error when parsing a DiffX file.
DiffXSectionOrderError	An error with the order of a section within the DiffX file.
DiffXUnknownOptionError	An option name is unknown for a given section.
MalformedHunkError(line, line_num[, msg])	Error with the contents of a hunk in a patch.

exception pydiffx.errors.**BaseDiffXError**

Bases: Exception

Base class for all DiffX errors.

exception pydiffx.errors.**DiffXParseError**(*msg*, *linenum*, *column=None*)

Bases: *BaseDiffXError*

An error when parsing a DiffX file.

Parse errors contain information on the line (and sometimes the column) causing parsing to fail, along with an error message.

column

The 0-based column number where the parse error occurred. This may be None for some parse errors.

Type
int

linenum

The 0-based line number where the parse error occurred.

Type
int

__init__(*msg*, *linenum*, *column=None*)

Initialize the error.

Parameters

- **msg** (unicode) – An error message explaining why the file could not be parsed.
- **linenum** (int) – The 0-based line number where the parse error occurred.
- **column** (int, *optional*) – The 0-based column number where the parse error occurred.

exception pydiffx.errors.**DiffXSectionOrderError**

Bases: *BaseDiffXError*

An error with the order of a section within the DiffX file.

exception pydiffx.errors.**DiffXContentError**

Bases: *BaseDiffXError*

An error with content for a section.

exception pydiffx.errors.**DiffXUnknownOptionError**

Bases: *BaseDiffXError*

An option name is unknown for a given section.

exception pydiffx.errors.**DiffXOptionValueError**

Bases: *BaseDiffXError*

An error with a value for an option.

exception pydiffx.errors.**DiffXOptionValueChoiceError**(*option*, *value*, *choices*)

Bases: *DiffXOptionValueError*

An error with the choice for a value for an option.

__init__(*option*, *value*, *choices*)

Initialize the error.

Parameters

- **option** (unicode) – The name of the option.
- **value** (object) – The value that was chosen.
- **choices** (list of unicode) – The list of values considered valid.

exception pydiffx.errors.**MalformedHunkError**(*line*, *line_num*, *msg=None*)

Bases: Exception

Error with the contents of a hunk in a patch.

line

The contents of the line triggering the error.

Type
bytes

line_num

The 1-based line number where the error occurred.

Type
int

__init__(*line*, *line_num*, *msg=None*)

Initialize the error.

Parameters

- **line** (bytes) – The contents of the line triggering the error.
- **line_num** (int) – The 1-based line number where the error occurred.
- **msg** (unicode, *optional*) – An optional error message to display instead of the default message. This may contain `line` and `line_num` format strings (built for `%`-based formatting).

__eq__(*other*)

Return whether this exception equals another.

Parameters
other (object) – The object to compare to.

Returns
`True` if the objectws are equal. `False` if they are not.

Return type
bool

__hash__ = None

pydiffx.options

Constants and utilities for options.

Classes

DiffType()	Types available for a diff.
LineEndings()	Line ending types available for a content section.
MetaFormat()	Formats available for a meta section.
PreambleMimeType()	Mimetypes available for a preamble section.
SpecVersion()	Supported specification versions.

class pydiffx.options.**DiffType**

Bases: object

Types available for a diff.

These may be used in a diff section's `diff_type` option.

TEXT = 'text'

Text-based diffs.

BINARY = 'binary'

Binary diffs.

VALID_VALUES = {'binary', 'text'}

A set of values allowed for the diff_type option.

class pydiffx.options.**LineEndings**

Bases: object

Line ending types available for a content section.

These may be used in a content section's `line_endings` option.

DOS = 'dos'

DOS (CRLF) line endings.

UNIX = 'unix'

UNIX (LF) line endings.

VALID_VALUES = {'dos', 'unix'}

A set of values allowed for the line_endings option.

class pydiffx.options.**MetaFormat**

Bases: object

Formats available for a meta section.

These may be used in a meta section's `format` option.

JSON = 'json'

JSON metadata.

VALID_VALUES = {'json'}

A set of values allowed for the format option.

class pydiffx.options.**PreambleMimeType**

Bases: object

Mimetypes available for a preamble section.

These may be used in a preamble section's `mimetype` option.

PLAIN = 'text/plain'

Plain text.

MARKDOWN = 'text/markdown'

Markdown text.

VALID_VALUES = {'text/markdown', 'text/plain'}

A set of values allowed for the mimetype option.

class pydiffx.options.**SpecVersion**

Bases: object

Supported specification versions.

These may be used as the DiffX `version` option.

DEFAULT_VERSION = '1.0'

The default version to write.

VALID_VALUES = {'1.0'}

A set of values allowed for the version option.

pydiffx.reader

A streaming reader for DiffX files.

Classes

DiffXReader(fp)	A streaming reader for DiffX files.

class pydiffx.reader.**DiffXReader**(*fp*)

Bases: object

A streaming reader for DiffX files.

This is a low-level interface for reading a DiffX file from an existing stream, such as an opened file handle or a web server response.

Consumers can iterate through each section of the DiffX file, reading sections one-by-one and processing them. This can be used to process the metadata on-the-fly without retaining the entirety of the file in memory, or to convert it into another data structure.

See *iter_sections()* for details on the information returned during iteration.

__init__(*fp*)

Initialize the reader.

Parameters
fp (file or io.IOBase) – The file pointer/stream to read from. This must be opened in binary (bytes) mode.

__iter__()

Iterate through all sections of a DiffX file.

This is a convenience wrapper around *iter_sections()*. See that method for details.

Yields
dict – Information on the section.

Raises
pydiffx.errors.DiffXParseError – The file or a section was unable to be parsed. Information will be provided in the message and the instance's attributes.

iter_sections()

Iterate through all sections of a DiffX file.

Each section and subsection will be parsed individually, returning the following data on each new section:

level (**int**):
The 0-based section level (corresponding to the number of . level indicator characters in the section ID).

line (**int**):
The 0-based line number where the section starts.

options (**dict**):
A dictionary of options found for the section.

section (**unicode**):
The ID of the section. This corresponds to one of:

- *MAIN*
- *MAIN_PREAMBLE*
- *MAIN_META*
- *CHANGE*
- *CHANGE_PREAMBLE*
- *CHANGE_META*
- *FILE*
- *FILE_META*
- *FILE_DIFF*

type (**unicode**):
The type of section (the ID in the file following the `.` level indicator characters).

Preamble sections will also contain:

text (**unicode**):
The decoded text content of the preamble.

Metadata sections will also contain:

metadata (**dict**):
A dictionary containing all metadata for the section.

Diff sections will also contain:

diff (**bytes** or **unicode**):
The diff content. If an encoding is specified, this will be decoded to a Unicode string. Otherwise, it will be a byte string. Callers must check for this.

Note: If any given section fails to parse, an error will be raised and parsing will stop.

Yields
`dict` – Information on the section.

Raises
pydiffx.errors.DiffXParseError – The file or a section was unable to be parsed. Information will be provided in the message and the instance's attributes.

pydiffx.sections

Section-related definitions.

This is mostly useful internally for diff generation and parsing.

Module Attributes

PREAMBLE_SECTIONS	A set of all preamble sections.
META_SECTIONS	A set of all meta sections.
CONTENT_SECTIONS	A set of all content sections.
VALID_SECTION_STATES	A mapping of section IDs to sections that may appear next in the file.

Classes

Section()	Valid section IDs in a DiffX file.

class pydiffx.sections.**Section**

Bases: object

Valid section IDs in a DiffX file.

MAIN = 'diffx'

The ID of the main DiffX section.

MAIN_PREAMBLE = '.preamble'

The ID of the main DiffX preamble section.

MAIN_META = '.meta'

The ID of the main DiffX metadata section.

CHANGE = '.change'

The ID of a change section.

CHANGE_PREAMBLE = '..preamble'

The ID of a change's preamble section.

CHANGE_META = '..meta'

The ID of a change's metadata section.

FILE = '..file'

The ID of a file section.

FILE_META = '...meta'

The ID of a file's metadata section.

FILE_DIFF = '...diff'

The ID of a file's diff section.

pydiffx.sections.**PREAMBLE_SECTIONS = {'..preamble', '.preamble'}**

A set of all preamble sections.

pydiffx.sections.**META_SECTIONS = {'...meta', '..meta', '.meta'}**

A set of all meta sections.

pydiffx.sections.**CONTENT_SECTIONS = {'...diff', '...meta', '..meta', '..preamble', '.meta', '.preamble'}**

A set of all content sections.

pydiffx.sections.**VALID_SECTION_STATES = {'...diff': {'..file', '.change'}, '...meta': {'...diff', '..file', '.change'}, '..file': {'...meta'}, '..meta': {'..file', '.change'}, '..preamble': {'..file', '..meta'}, '.change': {'..file', '..meta', '..preamble'}, '.meta': {'.change'}, '.preamble': {'.change', '.meta'}, 'diffx': {'.change', '.meta', '.preamble'}}**

A mapping of section IDs to sections that may appear next in the file.

pydiffx.utils

pydiffx.utils.text

Utilities for processing text.

Module Attributes

NEWLINE_FORMATS	A mapping of newline format types to character sequences.
BOMS	A mapping of encodings to possible BOM markers.

Functions

get_newline_for_type(line_endings[, encoding])	Return the newline for a given type of line endings.
guess_line_endings(text[, encoding])	Return the line endings that appear to be used for text.
split_lines(data, newline[, keep_ends])	Split data along newline boundaries.
strip_bom(data, encoding)	Strip a BOM from the beginning of a string.

pydiffx.utils.text.**NEWLINE_FORMATS = {'dos': '\r\n', 'unix': '\n'}**

A mapping of newline format types to character sequences.

This contains only formats that are allowed in the `line_endings=` option in DiffX content sections.

Type
dict

pydiffx.utils.text.**BOMS = {'utf-16': (b'\xfe\xff', b'\xff\xfe'), 'utf-16-be': (b'\xfe\xff',), 'utf-16-le': (b'\xff\xfe',), 'utf-32': (b'\x00\x00\xfe\xff', b'\xff\xfe\x00\x00'), 'utf-32-be': (b'\x00\x00\xfe\xff',), 'utf-32-le': (b'\xff\xfe\x00\x00',), 'utf-8': (b'\xef\xbb\xbf',)}**

A mapping of encodings to possible BOM markers.

pydiffx.utils.text.**split_lines**(*data*, *newline*, *keep_ends=False*)

Split data along newline boundaries.

This differs from `str.splitlines()` in that it will split across a specific newline boundary, rather than against any sequence of newline characters.

Parameters

- **data** (bytes) – The data to split.
- **newline** (bytes) – The newline character(s) used to split the data into lines.

- **keep_ends** (bool, *optional*) – Whether to keep the line endings in the resulting lines.

Returns
The split list of lines.

Return type
list of bytes

pydiffx.utils.text.**get_newline_for_type**(*line_endings*, *encoding=None*)

Return the newline for a given type of line endings.

The resulting newline characters will be encoded into the given encoding, if specified, or as plain ASCII if not specified.

If a BOM is present in the result, it will be stripped.

Parameters

- **line_endings** (unicode) – The type of line endings. This will be of of *LineEndings.DOS* or *LineEndings.UNIX*.
- **encoding** (unicode, *optional*) – The encoding to use for the resulting newline. If None, "ascii" will be used.

Returns
The resulting encoded newline characters.

Return type
bytes

Raises

- **LookupError** – encoding was not a valid encoding type.
- **ValueError** – line_endings was not a valid type of line endings.

pydiffx.utils.text.**guess_line_endings**(*text*, *encoding=None*)

Return the line endings that appear to be used for text.

This will check the first line of content and see if it appears to be DOS or UNIX line endings.

If there are no newlines, UNIX line endings are assumed.

Parameters

- **text** (bytes or unicode) – The text to guess line endings from.
- **encoding** (unicode, *optional*) – The encoding of the text, if it's a byte string.

Returns

A 2-tuple of:

1. The guessed line endings type (as a line_endings= option value).
2. The line ending characters (in the same string type as text).

Return type
tuple

pydiffx.utils.text.**strip_bom**(*data*, *encoding*)

Strip a BOM from the beginning of a string.

If the encoding is one that contains a BOM, and any version (such as Big Endian or Little Endian) of the BOM are present, they'll be stripped.

Parameters

- **data** (bytes) – The byte string to strip a BOM from.
- **encoding** (unicode) – The encoding of the byte string.

Returns
The string, without any BOM markers.

Return type
bytes

pydiffx.utils.unified_diffs

Utilities for parsing Unified Diffs.

Functions

get_unified_diff_hunks(lines[, ignore_garbage])	Return information on each hunk in a Unified Diff.

pydiffx.utils.unified_diffs.**get_unified_diff_hunks**(*lines*, *ignore_garbage=False*)

Return information on each hunk in a Unified Diff.

This will iterate through each hunk, generating information on each hunk. Parsing will continue until something other than a hunk is found (unless passing `ignore_garbage=True`).

Parameters

- **lines** (list of bytes) – The list of lines in the diff. This should generally be the result of using *split_lines()*.
- **ignore_garbage** (bool, *optional*) – Whether to ignore garbage lines found outside of a hunk.

  If `True`, all lines will be processed for hunk data.

  If `False` (the default), reading will stop once something other than a hunk is found.

Returns

A dictionary containing the results. This will have the following keys:

hunks (**list of** **dict**)**:**
The list of hunks. Each dictionary will contain:

context (**bytes**)**:**
Optional context shown after the `@@` header. This may be `None`.

lines_of_context_pre (**int**)**:**
The number of lines of context before the first changed line in the hunk.

lines_of_context_post (**int**)**:**
The number of lines of context after the last changed line in the hunk.

modified (**dict**)**:**
Information on the modified side of the hunk. This will contain the following keys:

start_line (**int**)**:**
The 0-based line number in the original file where the hunk begins. This will be the line number of the first line shown in the hunk, which may include lines of context.

num_lines (int):
The number of lines shown for the original side of the hunk in the diff, including any lines of context, unchanged lines, or changed lines.

first_changed_line (int):
The 0-based line number in the original file where the first change in the hunk (a - line) occurs. This will always be after any lines of context.

last_changed_line (int):
The 0-based line number in the original file where the last change in the hunk (a - line) occurs. This will always be before any lines of context.

num_lines_changed (int):
The number of lines that were changed in original side of the hunk (the number of - lines).

orig (dict):
Information on the original side of the hunk. This will contain the following keys:

start_line (int):
The 0-based line number in the modified file where the hunk begins. This will be the line number of the first line shown in the hunk, which may include lines of context.

num_lines (int):
The number of lines shown for the modified side of the hunk in the diff, including any lines of context, unchanged lines, or changed lines.

first_changed_line (int):
The 0-based line number in the modified file where the first change in the hunk (a + line) occurs. This will always be after any lines of context.

last_changed_line (int):
The 0-based line number in the modified file where the last change in the hunk (a + line) occurs. This will always be before any lines of context.

num_lines_changed (int):
The number of lines that were changed in modified side of the hunk (the number of + lines).

num_processed_lines (int):
The number of lines read in the diff to produce these results. Callers can use this to start parsing the rest of a diff after these lines.

total_deletes (int):
The total number of deleted lines found.

total_inserts (int):
The total number of inserted lines found.

Return type
dict

Raises
pydiffx.errors.MalformedHunkError – A line was found within a hunk that was not valid and could not be parsed, or a hunk was terminated prematurely.

pydiffx.writer

A streaming writer for DiffX files.

Classes

DiffXWriter(fp[, encoding, version])	A streaming writer for DiffX files.

class pydiffx.writer.**DiffXWriter**(*fp*, *encoding='utf-8'*, *version='1.0'*)

Bases: object

A streaming writer for DiffX files.

This is a low-level interface for writing a DiffX file to an existing stream, such as an opened file handle or an in-progress web server response.

Consumers can incrementally write change, file, metadata, preamble, and diff contents to the stream without keeping it all in memory up-front. Consumers are responsible for including any necessary metadata for each section.

VERSION = '1.0'

The supported version of the DiffX specification.

DEFAULT_PREAMBLE_INDENT = 4

Default indentation to apply to preamble sections.

DEFAULT_ENCODING = 'utf-8'

Default encoding to use for the DiffX file.

__init__(*fp*, *encoding='utf-8'*, *version='1.0'*)

Initialize the writer.

Parameters

- **fp** (file or io.IOBase) – The file pointer/stream to write to. This must be opened in binary (bytes) mode.
- **encoding** (unicode, *optional*) – The default encoding for content in the file. This will generally be left as the default of "utf-8".
- **version** (unicode, *optional*) – The version of the DiffX file to write.

  This must currently be 1.0.

new_change(*encoding=None*)

Write a new change section to the stream.

Parameters

encoding (unicode, *optional*) – The encoding to use for the section. Defaults to the main DiffX file encoding.

Raises

pydiffx.errors.DiffXSectionOrderError – This was called at the wrong point in diff generation.

new_file(*encoding=None*)

Write a new file section to the stream.

new_change() must have been called at least once before this is called.

Parameters

encoding (unicode, *optional*) – The encoding to use for the section. Defaults to the parent change section's encoding.

Raises

pydiffx.errors.DiffXSectionOrderError – This was called at the wrong point in diff generation.

write_preamble(*text*, *encoding=None*, *indent=4*, *line_endings=None*, *mimetype=None*)

Write a new preamble section for a change or a file.

If called as the first operation on a new stream, this will write a top-level DiffX preamble.

If called immediately after a call to *new_change()*, this will write a change preamble.

This cannot be called at any other time.

This must be called before *write_meta()* in the section.

Parameters

- **text** (unicode) – The text to write.
- **encoding** (unicode, *optional*) – The encoding to use for the section. Defaults to the parent change section's encoding.
- **indent** (int, *optional*) – The optional indentation level for the text. This defaults to 4 spaces.

  This is used to ensure preamble text cannot interfere with the parsing of any DiffX or diff content.
- **line_endings** (unicode, *optional*) – The line endings used for the preamble. This can be "dos" or "unix".

  If not provided, a value will be computed based on content, and then inserted into the header.
- **mimetype** (unicode, *optional*) – The optional mimetype for the file contents. If not provided, this will be plain text.

  Supported values are `text/plain` or `text/markdown`.

Raises

- *pydiffx.errors.DiffXContentError* – The content was empty or was an invalid type.
- *pydiffx.errors.DiffXOptionValueError* – An option value was invalid.
- *pydiffx.errors.DiffXSectionOrderError* – This was called at the wrong point in diff generation.

write_meta(*metadata*, *encoding=None*, *meta_format='json'*)

Write a new meta section for DiffX, a change, or a file.

If called before *new_change()*, this will write a top-level DiffX meta section.

If called after *new_change()* but before *new_file()*, this will write a change meta section.

If called after *new_file()*, this will write a file meta section.

This cannot be called before *write_preamble()* in the section, or after *write_diff()* in file sections.

Parameters

- **metadata** (dict) – The metadata to write.

- **encoding** (unicode, *optional*) – The encoding to use for the section. Defaults to the parent change section's encoding.
- **meta_format** (unicode, *optional*) – The format for this metadata section.

  Valid values are in *MetaFormat*.

Raises

- *pydiffx.errors.DiffXContentError* – The metadata was empty or was an invalid type.
- *pydiffx.errors.DiffXOptionValueError* – An option value was invalid.
- *pydiffx.errors.DiffXSectionOrderError* – This was called at the wrong point in diff generation.

write_diff(*content*, *diff_type=None*, *encoding=None*, *line_endings=None*)

Write a new diff section for a file.

This must be called after *new_file()*, and must be after the *write_meta()* call.

Parameters

- **content** (bytes) – The diff content to write.
- **diff_type** (unicode, *optional*) – The type of diff to write. This must be one of DIFF_TYPE_TEXT or DIFF_TYPE_BINARY.
- **encoding** (unicode, *optional*) – The encoding to use for the section. This does not inherit from previous sections.
- **line_endings** (unicode, *optional*) – The line endings used for the diff. This can be "dos" or "unix".

  If not provided, a value will be computed based on content, and then inserted into the header.

Raises

- *pydiffx.errors.DiffXContentError* – The diff was an invalid type.
- *pydiffx.errors.DiffXOptionValueError* – An option value was invalid.
- *pydiffx.errors.DiffXSectionOrderError* – This was called at the wrong point in diff generation.

Release Notes

1.x Releases

pydiffx 1.1.0 Release Notes

Release date: September 18, 2022

Compatibility

- Added explicit support for Python 3.10 and 3.11.

Bug Fixes

- Fixed parsing Unified Diff hunks with "No newline at end of file" markers in *pydiffx.utils.unified_diffs.get_unified_diff_hunks()*.

  This also applies when generating stats for metadata sections.
- Generating stats on empty diffs no longer results in errors.

Contributors

- Christian Hammond
- David Trowbridge
- Jordan Van Den Bruel

pydiffx 1.0.1 Release Notes

Release date: August 4, 2021

Bug Fixes

- Fixed writing out binary diff content when using *pydiffx.dom.objects.DiffX*.
- Diff statistics are no longer generated for binary diff content when using *pydiffx.dom.objects.DiffX*.

Contributors

- Christian Hammond
- David Trowbridge

pydiffx 1.0 Release Notes

Release date: August 1, 2021

Initial Release

This is the first release of pydiffx. It's compliant with the *DiffX 1.0 specification* as of August 1, 2021, and features the following interfaces:

- `pydiffx.dom.objects.DiffX` - The DiffX Object Model
- `pydiffx.reader.DiffXReader` - A streaming reader
- `pydiffx.writer.DiffXWriter` - A streaming writer

pydiffx is production-ready, and being used today in Review Board. We're also planning official DiffX implementations in additional languages.

Contributors

- Christian Hammond
- David Trowbridge

6.4 Frequently Asked Questions

6.4.1 How important is this, really?

If you're developing a code review tool or patcher or something that makes use of diff files, you've probably had to deal with all the *subtle things that can go wrong in a diff*.

If you're an end user working solely in Git, or in Subversion, or something similar, you probably don't directly care. That being said, sometimes users hit really funky problems that end up being due to command line options or environmental problems mixed with the lack of information in a diff (no knowledge of whether whitespace was being ignored, or the line endings being used, or the text encoding). If tools had this information, they could be smarter, and you wouldn't have to worry about as many things going wrong.

A structured, parsable format can only help.

6.4.2 Why not move to JSON or some other format for diffs?

Unified Diffs are a pretty decent format in many regards. Practically any tool that understands diffs knows how to parse them, and they're very forgiving in that they don't mind having unknown content outside of the range of changes.

If we used an alternative format, it's likely nobody would ever use it. Creating an incompatible format doesn't provide any real benefit, and would fragment the development world and many current workflows.

By building on top of Unified Diffs, we get to keep compatibility with existing tools, without having to rewrite the world. Everybody wins.

6.4.3 Why not use Git Bundles, or similar?

Git's bundles format is really just a way of taking part of a Git tree and transporting it. You still need to have parent commits available on a clone. You can't upload it to some service and expect it'll be able to work with it.

It's also a Git format, not something Mercurial, Subversion, etc. can make use of. It's not an alternative to diffs.

6.4.4 How does DiffX retain backwards-compatibility?

Unified Diffs aren't at all strict about the content that exists outside of a file header and a set of changed lines. This means you can add basically anything before and after these parts of the diff. DiffX takes advantage of this by adding identifiable markers that a parser can look for.

It also knows how to ignore any special data that may be specific to a Git diff, Subversion diff, etc., preferring the DiffX data instead.

However, when you feed this back into something expecting an old-fashioned Git diff or similar, that parser will ignore all the DiffX content that it doesn't understand, and instead read in the legacy information.

This only happens if you generate a DiffX that contains the legacy information, of course. DiffX files don't have to include these. It's really up to the tool generating the diff.

So basically, we keep all the older content that non-DiffX tools look for, and DiffX-capable tools will just ignore that content in favor of the new content.

6.4.5 What do DiffX files offer that Unified Diffs don't?

Many things:

- A consistent way to parse, generate, and update diffs
- Multiple commits represented in one file
- Binary diffs
- Structured metadata in a standard format
- Per-file text encoding indicators
- Standard metadata for representing moved files, renames, attribute changes, and more.

6.4.6 What supports DiffX today?

DiffX is still in a specification and prototype phase. We are adding support in Review Board and RBTools.

If you're looking to add support as well, please let us know and we'll add you to a list.

6.5 Glossary

CR
Carriage Return
A Carriage Return character (\n), generally used as part of a *CRLF* line ending.

CRLF
Carriage Return, Line Feed
A Carriage Return character followed by a Line Feed (\r\n), generally used as a line ending on DOS/Windows-based systems.

LF
Line Feed
A Line Feed character (\n), generally used as a line ending on UNIX-based systems, or as part of a *CRLF* line ending.

Unified Diff
Unified Diffs
A more-or-less standard way of representing changes to one or more text files. The standard part is the way it represents changes to lines, like:

```
@@ -1 +1,3 @@
 Hello there
+
+Oh hi!
```

The rest of the format has no standardization. There are some general standard-ish markers that tools like GNU Patch understand, but there's a *lot* of variety here, so they're hard to parse. For instance:

```
--- readme    26 Jan 2016 16:29:12 -0000        1.1
+++ readme    31 Jan 2016 11:54:32 -0000        1.2
```

```
--- readme    (revision 123)
+++ readme    (working copy)
```

```
--- a/readme
+++ b/readme
```

This is one of the problems being solved by DiffX.

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W