dbfread Documentation

Release 2.0.7

Ole Martin Bjørndalen

January 29, 2017

Contents

Version 2.0.7

DBF is a file format used by databases such dBase, Visual FoxPro, and FoxBase+. This library reads DBF files and returns the data as native Python data types for further processing. It is primarily intended for batch jobs and one-off scripts.

```
>>> from dbfread import DBF
>>> for record in DBF('people.dbf'):
...     print(record)
OrderedDict([('NAME', 'Alice'), ('BIRTHDATE', datetime.date(1987, 3, 1))])
OrderedDict([('NAME', 'Bob'), ('BIRTHDATE', datetime.date(1980, 11, 12))])
```

CHAPTER 1

Source code

Latest stable release: https://github.com/olemb/dbfread/

Latest development version: https://github.com/olemb/dbfread/tree/develop/

CHAPTER 2

About This Document

This document is available at https://dbfread.readthedocs.io/

To build documentation locally:

```
python setup.py docs
```

This requires Sphinx. The resulting files can be found in `docs/_build/`.

CHAPTER 3

Contents

3.1 Changes

3.1.1 Release History

2.0.7 - 2016-11-24

- Sometimes numeric (N) and float fields (F) are padded with '*'. These are now stripped. (Reported by sgiangola and Matungos, issue #10.)
- added `char_decode_errors` option which lets you choose how to handle characters that can't be decoded. (Implemented by ZHU Enwei, pull request #16.)
- added `--char-decode-errors` option to `dbf2sqlite`.
- added `dbfread.version_info`.

2.0.6 - 2016-06-07

- Added support for long character (C) fields (up to 65535 bytes). (Requested by Eric Mertens and Marcelo Manzano.)
- Added support for Visual FoxPro varchar fields (V). (Thanks to Roman Kharin for reporting and bobintetley for providing a solution.)
- Bugfix (dbf2sqlite): some table or field names might actually collide with sql reserved words. (Fix by vthriller, pull request #15.)
- Documented how to convert records to Pandas data frames. (Thanks to Roman Yurchak for suggesting this.)

2.0.5 - 2015-11-30

- Bugfix: memo field parser used str instead of bytes. (Fix submitted independently by Sebastian Setzer (via email) and by Artem Vlasov, pull request #11.)
- Bugfix: some field parsers called self._get_memo() instead of self.get_memo(). (Fix by Yu Feng, pull request #9.)

2.0.4 - 2015-02-07

- DBF header and field headers are no longer read-only. For example you can now change field names by doing `table.fields[0].name = 'price'` or read from files where field sizes in the header don't match those in the actual records by doing `table.fields[0].length = 500`.
- fixed some examples that didn't work with Python 3.

2.0.3 - 2014-09-30

- added currency field (Y). (Patch by Stack-of-Pancakes.)

2.0.2 - 2014-09-29

- bugfix: a date with all zeroes in the DBF header resulted in 'ValueError: month must be in 1..12'. (Reported by Andrew Myers.) The `date` attribute is now set to `None` for any value that is not a valid date.

2.0.1 - 2014-09-19

- bugfix: didn't handle field names with garbage after b'0' terminator. (Patch by Cédric Krier.)
- now handles 0 (_NullFlags) fields that are more than 1 byte long. 0 fields are now returned as byte strings instead of integers. (Reported by Carlos Huga.)
- the type B field is a double precision floating point numbers in Visual FoxPro. The parser crashed when it tried to interpret this as a string containing a number. (Reported by Carlos Huga.)
- API changes: memo field parsers now return the memo data (typically a unicode string or bytes object) instead of returning the index. This makes it easier to implement new memo types or extend the existing ones since memo fields are no longer a special case.

2.0.0 - 2014-08-12

- `dbfread.open()` and `dbfread.read()` are now deprecated and will be removed in 1.4. Since the `DBF` object is no longer a subclass of list, these functions instead return backward compatible `DeprecatedDBF` objects.
- records are now returned as ordered dictionaries. This makes it easier to iterate over fields in the same order that they appear in the file.
- now reads (at least some) DBT files.
- added support for 6 new field types.
- added `ignore_missing_memofile` argument. If `True` and the memo file is not found all memo fields will be returned as `None`.
- DBF now raises `DBFNotFound` and `MissingMemoFile`. These inherit from IOError, so old code should still work.
- added `InvalidValue`. This is currently not used by the library but can be useful for custom parsing.
- `FieldParser` is now available in the top scope.
- wrote documentation.
- switched to pytest for unit tests.

1.1.1 - 2014-08-03

- example and test data files were missing from the manifest.

1.1.0 - 2014-08-03

- the `DBF` object is no longer a subclass of list. Records are instead available in the `records` attribute, but the table can be iterated over like before. This change was made to make the API cleaner and easier to understand. `read()` is still included for backwards compatability, and returns an `OldStyleTable` object with the old behaviour.
- default character encoding is now `"ascii"`. This is a saner default than the previously used `"latin1"`, which would decode but could give the wrong characters.
- the DBF object can now be used as a context manager (using the "with" statement).

1.0.6 - 2014-08-02

- critical bugfix: each record contained only the last field. (Introduced in 1.0.5, making that version unusable.)
- improved performance of record reading a bit.

1.0.5 - 2014-08-01

This version is broken.

- more than doubled performance of record parsing.
- removed circular dependency between table and deleted record iterator.
- added `dbversion` attribute.
- added example `dbfinfo.py`.
- numeric field (N) parser now handles invalid data correctly.
- added more unit tests.

1.0.4 - 2014-07-27

- bugfix: crashed when record list was not terminated with b'x1a'. (Bug first apperad in 1.0.2 after a rewrite.)
- bugfix: memo fields with no value were returned as ''. They are now returned correctly as None.
- bugfix: field header terminaters were compared with strings.
- added example parserclass_debugstring.py.

1.0.3 - 2014-07-26

- reinstated hastily removed parserclass option.

1.0.2 - 2014-07-26

- added example record_objects.py.
- removed parserclass option to allow for internal changes. There is currently no (documented) way to add custom field types.

1.0.1 - 2014-07-26

- bugfix: deleted records were ignored when using open().
- memo file is now opened and closed by each iterator instead of staying open all the time.

1.0.0 - 2014-07-25

- records can now be streamed from the file, making it possible to read data files that are too large to fit in memory.
- documentation is more readable and complete.
- now installs correctly with easy_install.
- added "–encoding" option to dbf2sqlite which can be used to override character encoding.

0.1.0 - 2014-04-08

Initial release.

3.2 Installing dbfread

3.2.1 Requirements

Requires Python 3.2 or 2.7. `dbfread` is a pure Python module, so doesn't depend on any packages outside the standard library.

3.2.2 Installing

```
pip install dbfread
```

3.3 Introduction

This is a short introduction to the API. If you want to follow along you can find `people.dbf` in `examples/files/`.

3.3.1 Opening a DBF File

```
>>> from dbfread import DBF
>>> table = DBF('people.dbf')
```

This returns a `DBF` object. You can now iterate over records:

```
>>> for record in table:
...     print(record)
OrderedDict([('NAME', 'Alice'), ('BIRTHDATE', datetime.date(1987, 3, 1))])
OrderedDict([('NAME', 'Bob'), ('BIRTHDATE', datetime.date(1980, 11, 12))])
```

and count records:

```
>>> len(table)
2
```

Deleted records are available in `deleted`:

```
>>> for record in table.deleted:
...     print(record)
OrderedDict([('NAME', 'Deleted Guy'), ('BIRTHDATE', datetime.date(1979, 12, 22))])

>>> len(table.deleted)
1
```

You can also use the `with` statement:

```
with DBF('people.dbf') as table:
    ...
```

The DBF object doesn't keep any files open, so this is provided merely as a convenience.

3.3.2 Streaming or Loading Records

By default records are streamed directly off disk, which means only one record is in memory at a time.

If have enough memory, you can load the records into a list by passing `load=True`. This allows for random access:

```
>>> table = DBF('people.dbf', load=True)
>>> print(table.records[1]['NAME'])
Bob
>>> print(table.records[0]['NAME'])
Alice
```

Deleted records are also loaded into a list in `table.deleted`.

Alternatively, you can load the records later by calling `table.load()`. This is useful when you want to look at the header before you commit to loading anything. For example, you can make a function which returns a list of tables in a directory and load only the ones you need.

If you just want a list of records and you don't care about the other table attributes you can do:

```
>>> records = list(DBF('people.dbf'))
```

You can unload records again with `table.unload()`.

If the table is not loaded, the `records` and `deleted` attributes return `RecordIterator` objects.

Loading or iterating over records will open the DBF and memo file once for each iteration. This means the `DBF` object doesn't hold any files open, only the `RecordIterator` object does.

3.3.3 Character Encodings

All text fields and memos (except binary ones) will be returned as unicode strings.

dbfread will try to detect the character encoding (code page) used in the file by looking at the `language_driver` byte. If this fails it reverts to ASCII. You can override this by passing `encoding='my-encoding'`. The encoding is available in the `encoding` attribute.

There may still be characters that won't decode. You can choose how to handle these by passing the `char_decode_errors` option. This is passed straight to `bytes.decode`. See `pydoc bytes.decode` for more.

3.3.4 Memo Files

If there is at least one memo field in the file dbfread will look for the corresponding memo file. For `buildings.dbf` this would be `buildings.fpt` (for Visual FoxPro) or `buildings.dbt` (for other databases).

Since the Windows file system is case preserving, the file names may end up mixed case. For example, you could have:

```
Buildings.dbf BUILDINGS.DBT
```

This creates problems in Linux, where file names are case sensitive. dbfread gets around this by ignoring case in file names. You can turn this off by passing `ignorecase=False`.

If the memo file is missing you will get a `MissingMemoFile` exception. If you still want the rest of the data you can pass `ignore_missing_memofile=True`. All memo field values will now be returned as `None`, as would be the case if there was no memo.

dbfread has full support for Visual FoxPro (`.FPT`) and dBase III (`.DBT`) memo files. It reads dBase IV (also `.DBT`) memo files, but only if they use the default block size of 512 bytes. (This will be fixed if I can find more files to study.)

3.3.5 Record Factories

If you don't want records returned as `collections.OrderedDict` you can use the `recfactory` argument to provide your own record factory.

A record factory is a function that takes a list of `(name, value)` pairs and returns a record. You can do whatever you like with this data. Here's a function that creates a record object with fields as attributes:

```
class Record(object):
    def __init__(self, items):
        for (name, value) in items:
            setattr(self, name, value)

for record in DBF('people.dbf', recfactory=Record, lowernames=True):
    print(record.name, record.birthdate)
```

If you pass `recfactory=None` you will get the original `(name, value)` list. (This is a shortcut for `recfactory=lambda items: items`.)

3.3.6 Custom Field Types

If the included message types are not enough you can add your own by subclassing `FieldParser`. As a silly example, here how you can read text (`C`) fields in reverse:

```
from dbfread import DBF, FieldParser

class MyFieldParser(FieldParser):
    def parseC(self, field, data):
        # Return strings reversed.
        return data.rstrip(' 0').decode()[::-1]

for record in DBF('files/people.dbf', parserclass=MyFieldParser):
    print(record['NAME'])
```

and here's how you can return invalid values as `InvalidValue` instead of raising `ValueError`:

```
from dbfread import DBF, FieldParser, InvalidValue

class MyFieldParser(FieldParser):
    def parse(self, field, data):
        try:
            return FieldParser.parse(self, field, data)
        except ValueError:
            return InvalidValue(data)

table = DBF('invalid_value.dbf', parserclass=MyFieldParser):
for i, record in enumerate(table):
    for name, value in record.items():
        if isinstance(value, InvalidValue):
            print('records[{}][{!r}] == {!r}'.format(i, name, value))
```

This will print:

```
records[0][u'BIRTHDATE'] == InvalidValue(b'NotAYear')
```

3.4 Moving data to SQL, CSV, Pandas etc.

3.4.1 CSV

This uses the standard library `csv` module:

```
"""Export to CSV."""
import sys
import csv
from dbfread import DBF

table = DBF('files/people.dbf')
writer = csv.writer(sys.stdout)

writer.writerow(table.field_names)
for record in table:
    writer.writerow(list(record.values()))
```

The output is:

```
NAME,BIRTHDATE
Alice,1987-03-01
Bob,1980-11-12
```

3.4.2 Pandas Data Frames

```
"""
Load content of a DBF file into a Pandas data frame.

The iter() is required because Pandas doesn't detect that the DBF
object is iterable.
"""
from dbfread import DBF
from pandas import DataFrame

dbf = DBF('files/people.dbf')
frame = DataFrame(iter(dbf))

print(frame)
```

This will print:

```
    BIRTHDATE   NAME
0  1987-03-01  Alice
1  1980-11-12    Bob
```

The `iter()` is required. Without it Pandas will not realize that it can iterate over the table.

Pandas will create a new list internally before converting the records to data frames. This means they will all be loaded into memory. There seems to be no way around this at the moment.

3.4.3 dataset (SQL)

The dataset package makes it easy to move data to a modern database. Here's how you can insert the `people` table into an SQLite database:

```
"""
Convert a DBF file to an SQLite table.

Requires dataset: https://dataset.readthedocs.io/
"""
import dataset
from dbfread import DBF

# Change to "dataset.connect('people.sqlite')" if you want a file.
db = dataset.connect('sqlite:///:memory:')
table = db['people']

for record in DBF('files/people.dbf', lowernames=True):
    table.insert(record)

# Select and print a record just to show that it worked.
print(table.find_one(name='Alice'))
```

(This also creates the schema.)

3.4.4 dbf2sqlite

You can use the included example program `dbf2sqlite` to insert tables into an SQLite database:

```
dbf2sqlite -o example.sqlite table1.dbf table2.dbf
```

This will create one table for each DBF file. You can also omit the `-o example.sqlite` option to have the SQL printed directly to stdout.

If you get character encoding errors you can pass `--encoding` to override the encoding, for example:

```
dbf2sqlite --encoding=latin1 ...
```

3.5 DBF Objects

3.5.1 Arguments

filename The DBF file to open.

The file name is case insensitive, which means `DBF('PEOPLE.DBF')` will open the file `people.dbf`. If there is a memo file, it too will be looked for in a case insensitive manner, so `DBF('PEOPLE.DBF')` would find the memo file `people.FPT`.

`DBFNotFound` will be raised if the file is not found, and `MissingMemoFile` if the memo file is missing.

load=False By default records will streamed directly from disk. If you pass `load=True` they will instead be loaded into lists and made available as the `records` and `deleted` attributes.

You can load and unload records at any time with the `load()` and `unload()` methods.

encoding=None Specify character encoding to use.

By default dbfread will try to guess character encoding from the `language_driver` byte. If this fails it falls back on ASCII.

char_decode_errors='strict'

The error handling scheme to use for the handling of decoding errors. This is passed as the `errors` option to the `bytes.decode()` method. From the documentation of that method:

"The default is 'strict' meaning that decoding errors raise a UnicodeDecodeError. Other possible values are 'ignore' and 'replace' as well as any other name registered with codecs.register_error that can handle UnicodeDecodeErrors."

lowernames=False Field names are typically uppercase. If you pass `True` all field names will be converted to lowercase.

recfactory=collections.OrderedDict

Takes a function that will be used to produce new records. The function will be called with a list of `(name, value)` pairs.

If you pass `recfactory=None` you will get the original `(name, value)` list.

ignorecase=True Windows uses a case preserving file system which means `people.dbf` and `PEOPLE.DBF` are the same file. This causes problems in for example Linux where case is significant. To get around this dbfread ignores case in file names. You can turn this off by passing `ignorecase=False`.

parserclass=FieldParser The parser to use when parsing field values. You can use this to add new field types or do custom parsing by subclassing `dbfread.FieldParser`. (See Field Types.)

ignore_missing_memofile=False If you don't have the memo field you can pass `ignore_missing_memofile=True`. All memo fields will then be returned as `None`, so you at least get the rest of the data.

raw=False Returns all data values as byte strings. This can be used for debugging or for doing your own decoding.

3.5.2 Methods

load() Load records into memory. This loads both records and deleted records. The `records` and `deleted` attributes will now be lists of records.

unload() Unload records from memory. The `records` and `deleted` attributes will now be instances of `RecordIterator`, which streams records from disk.

3.5.3 Attributes

records If the table is loaded this is a list of records. If not, it's a `RecordIterator` object. In either case, iterating over it or calling `len()` on it will give the same results.

deleted If the table is loaded this is a list of deleted records. If not, it's a `RecordIterator` object. In either case, iterating over it or calling `len()` on it will give the same results.

loaded `True` if records are loaded into memory.

dbversion The name of the program that created the database (based on the `dbversion` byte in the header). Example: `"FoxBASE+/Dbase III plus, no memory"`.

name Name of the table. This is the lowercased stem of the filename, for example the file `/home/me/SHOES.dbf` will have the name `shoes`.

date Date when the file was last updated (as `datetime.date`) or `None` if the date was all zeroes or invalid.

field_names A list of field names in the order they appear in the file. This can for example be used to produce the header line in a CSV file.

encoding Character encoding used in the file. This is determined by the `language_driver` byte in the header, and can be overriden with the `encoding` keyword argument.

ignorecase, lowernames, recfactory, parserclass, raw These are set to the values of the same keyword arguments.

filename File name of the DBF file.

memofilename File name of the memo file, or `None` if there is no memo file.

header The file header. This is only intended for internal use, but is exposed for debugging purposes. Example:

```
DBFHeader(dbversion=3, year=114, month=8, day=2, numrecords=3,
headerlen=97, recordlen=25, reserved1=0, incomplete_transaction=0,
encryption_flag=0, free_record_thread=0, reserved2=0, reserved3=0,
mdx_flag=0, language_driver=0, reserved4=0)
```

fields A list of field headers from the file. Example of a field:

```
DBFField(name='NAME', type='C', address=1, length=16, decimal_count=0,
reserved1=0, workarea_id=0, reserved2=0, reserved3=0, set_fields_flag=0,
reserved4=b'\x00\x00\x00\x00\x00\x00\x00', index_field_flag=0)
```

Only the `name`, `type` and `length` attributes are used.

3.6 Field Types

3.6.1 Supported Field Types

:	Field type	Converted to
+	autoincrement	int
@	time	datetime.datetime
0	flags	byte string (int before 2.0)
B	double	float (Visual FoxPro)
B	binary memo	byte string (other versions)
C	text	unicode string
D	date	datetime.date or None
F	float	float
G	OLE object	byte string
I	integer	int
L	logical	True, False or None
M	memo	unicode string (memo), byte string (picture or object) or None
N	numeric	int, float or None
O	double	float (floats are doubles in Python)
P	picture	byte string
T	time	datetime.datetime
V	varchar	unicode string
Y	currency	decimal.Decimal

Text values ('C') can be up to 65535 bytes long. DBF was originally limited to 255 bytes but some vendors have reused the `decimal_count` field to get another byte for field length.

The 'B' field type is used to store double precision (64 bit) floats in Visual FoxPro databases and binary memos in other versions. `dbfread` will look at the database version to parse and return the correct data type.

The '0' field type is used for '_NullFlags' in Visual FoxPro. It was mistakenly though to always be one byte long and was interpreted as an integer. From 2.0.1 on it is returned as a byte string.

The 'V' field is an alternative character field used by Visual FoxPro. The binary version of this field is not yet supported. (See https://msdn.microsoft.com/en-us/library/st4a0s68%28VS.80%29.aspx for more.)

3.6.2 Adding Custom Field Types

You can add new field types by subclassing `FieldParser`. For example:

```
"""
Add custom field parsing by subclassing FieldParser.
"""

from dbfread import DBF, FieldParser

class CustomFieldParser(FieldParser):
    def parseC(self, field, data):
        # Return strings reversed.
        return data.rstrip(b' 0').decode()[::-1]

for record in DBF('files/people.dbf', parserclass=CustomFieldParser):
    print(record['NAME'])
```

The `FieldParser` object has the following attributes:

self.table A reference to the `DBF` objects. This can be used to get the headers to find `dbversion` and other things.

self.encoding The character encoding. (A a shortcut for `self.table.encoding` to speed things up a bit.)

self.char_decode_errors Error handling scheme to use while decoding. (A shortcut for `self.table.char_decode_errors`.)

self.dbversion The database version as an integer. (A shortcut for `self.table.header.dbversion`.)

self.get_memo(index) Returns a memo from the memo file using the index stored in the field data.

This returns a byte string (`bytes`) which you can then decode.

For Visual FoxPro (`.FPT`) files it will return `TextMemo`, `PictureMemo` and `ObjectMemo` objects depending on the type of memo. These are all subclasses of `bytes` so the type is only used to annotate the memo type without breaking code elsewhere. The full class tree:

```
bytes
  VFPMemo
    TextMemo
    BinaryMemo
      PictureMemo
      ObjectMemo
```

These are all found in `dbfread.memo`.

self.decode_text(text)

This will decode the text using the correct encoding and the user supplied `char_decode_errors` option.

3.6.3 Special Characters in Field Type Names

For a field type like '+' (autoincrement) the method would be named `parse+()`. Since this is not allowed in Python you can instead use its ASCII value in hexadecimal. For example, the '+' parser is called `parse3F()`.

You can name your method with:

```
>>> 'parse' + format(ord('?'), 'x').upper()
'parse3F'
```

Just replace `'?'` with your field type.

3.6.4 InvalidValue

The field parser will normally raise `ValueError` when invalid values are encountered. If instead you want them returned as raw data you can do this:

```
"""
A field parser that returns invalid values as InvalidValue objects
instead of raising ValueError.
"""
from dbfread import DBF, FieldParser, InvalidValue

class MyFieldParser(FieldParser):
    def parse(self, field, data):
        try:
            return FieldParser.parse(self, field, data)
        except ValueError:
            return InvalidValue(data)
```

```
table = DBF('files/invalid_value.dbf', parserclass=MyFieldParser)
for i, record in enumerate(table):
    for name, value in record.items():
        if isinstance(value, InvalidValue):
            print('records[{}][{!r}] == {!r}'.format(i, name, value))
```

`InvalidValue` is a subclass of `bytes`, and allows you to tell invalid data apart from valid data that happens to be byte strings. You can test for this with:

```
isinstance(value, InvalidData)
```

You can also tell from the `repr()` string:

```
>>> value
InvalidData(b'not a number')
```

3.7 API Changes

`dbfread.open()` and `dbfread.read()` are deprecated as of version `2.0`, and will be removed in `2.2`.

The `DBF` class is no longer a subclass of `list`. This makes the API a lot cleaner and easier to understand, but old code that relied on this behaviour will be broken. Iteration and record counting works the same as before. Other list operations can be rewritten using the `record` attribute. For example:

```
table = dbfread.read('people.dbf')
print(table[1])
```

can be rewritten as:

```
table = DBF('people.dbf', load=True)
print(table.records[1])
```

`open()` and `read()` both return `DeprecatedDBF`, which is a subclass of `DBF` and `list` and thus backward compatible.

3.8 Resources

3.8.1 DBF file format documentation

- Xbase File Format Description by Erik Bachmann
- Data File Header Structure for the dBASE Version 7 Table File
- Wikipedia article about dBase
- DBF Field Types and Specifications
- DBase File Structure
- dBase IV limitations
- DBF Table File Structure (Microsoft Developer Network)

3.9 License

3.10 Acknowledgements

The code page table is based on the one in Ethan Furman's dbf package.