Fiona Documentation

Release 1.8.6

Sean Gillies

Sep 03, 2019

CONTENTS

1 Fiona **1**
- 1.1 Usage . . . 1
- 1.2 Fiona CLI . . . 4
- 1.3 Installation . . . 4
- 1.4 Development and testing . . . 6
- 1.5 Changes . . . 6
- 1.6 Credits . . . 21

2 1 The Fiona User Manual **23**
- 2.1 1.1 Introduction . . . 23
- 2.2 1.2 Data Model . . . 25
- 2.3 1.3 Reading Vector Data . . . 26
- 2.4 1.4 Format Drivers, CRS, Bounds, and Schema . . . 28
- 2.5 1.5 Records . . . 31
- 2.6 1.6 Writing Vector Data . . . 33
- 2.7 1.7 Advanced Topics . . . 38
- 2.8 1.8 Fiona command line interface . . . 43
- 2.9 1.9 Final Notes . . . 43
- 2.10 1.10 References . . . 43

3 fiona **45**
- 3.1 fiona package . . . 45

4 Command Line Interface **63**
- 4.1 bounds . . . 63
- 4.2 calc . . . 64
- 4.3 cat . . . 64
- 4.4 collect . . . 64
- 4.5 distrib . . . 65
- 4.6 dump . . . 65
- 4.7 info . . . 65
- 4.8 load . . . 66
- 4.9 filter . . . 67
- 4.10 rm . . . 67
- 4.11 Coordinate Reference System Transformations . . . 68

5 Indices and tables **69**

Bibliography **71**

Python Module Index **73**

Index 75

CHAPTER

ONE

FIONA

Fiona reads and writes geographic data files and thereby helps Python programmers integrate geographic information systems with other computer systems. Fiona contains extension modules that link the Geospatial Data Abstraction Library (GDAL).

Fiona is designed to be simple and dependable. It focuses on reading and writing data in standard Python IO style and relies upon familiar Python types and protocols such as files, dictionaries, mappings, and iterators instead of classes specific to GDAL's OpenGIS Reference Implementation (OGR). Fiona can read and write real-world data using multi-layered GIS formats and zipped virtual file systems and integrates readily with other Python GIS packages such as pyproj, Rtree, and Shapely. Fiona is supported only on CPython versions 2.7 and 3.4+.

Why the name "Fiona"? Because Fiona is OGR's neat and nimble API for Python programmers.

For more details, see:

- Fiona home page
- Docs and manual
- Examples
- Main user discussion group
- Developers discussion group

1.1 Usage

1.1.1 Collections

Records are read from and written to `file`-like *Collection* objects returned from the `fiona.open()` function. Records are mappings modeled on the GeoJSON format. They don't have any spatial methods of their own, so if you want to do anything fancy with them you will probably need Shapely or something like it. Here is an example of using Fiona to read some records from one data file, change their geometry attributes, and write them to a new data file.

```
import fiona

# Open a file for reading. We'll call this the "source."
```

(continues on next page)

(continued from previous page)

```
with fiona.open('tests/data/coutwildrnp.shp') as src:

    # The file we'll write to, the "destination", must be initialized
    # with a coordinate system, a format driver name, and
    # a record schema.  We can get initial values from the open
    # collection's ``meta`` property and then modify them as
    # desired.

    meta = src.meta
    meta['schema']['geometry'] = 'Point'

    # Open an output file, using the same format driver and
    # coordinate reference system as the source. The ``meta``
    # mapping fills in the keyword parameters of fiona.open().

    with fiona.open('test_write.shp', 'w', **meta) as dst:

        # Process only the records intersecting a box.
        for f in src.filter(bbox=(-107.0, 37.0, -105.0, 39.0)):

            # Get a point on the boundary of the record's
            # geometry.

            f['geometry'] = {
                'type': 'Point',
                'coordinates': f['geometry']['coordinates'][0][0]}

            # Write the record out.

            dst.write(f)

# The destination's contents are flushed to disk and the file is
# closed when its ``with`` block ends. This effectively
# executes ``dst.flush(); dst.close()``.
```

1.1.2 Reading Multilayer data

Collections can also be made from single layers within multilayer files or directories of data. The target layer is specified by name or by its integer index within the file or directory. The `fiona.listlayers()` function provides an index ordered list of layer names.

```
for layername in fiona.listlayers('tests/data'):
    with fiona.open('tests/data', layer=layername) as src:
        print(layername, len(src))

# Output:
# (u'coutwildrnp', 67)
```

Layer can also be specified by index. In this case, `layer=0` and `layer='test_uk'` specify the same layer in the data file or directory.

```
for i, layername in enumerate(fiona.listlayers('tests/data')):
    with fiona.open('tests/data', layer=i) as src:
        print(i, layername, len(src))
```

(continues on next page)

(continued from previous page)

```
# Output:
# (0, u'coutwildrnp', 67)
```

1.1.3 Writing Multilayer data

Multilayer data can be written as well. Layers must be specified by name when writing.

```
with open('tests/data/cowildrnp.shp') as src:
    meta = src.meta
    f = next(src)

with fiona.open('/tmp/foo', 'w', layer='bar', **meta) as dst:
    dst.write(f)

print(fiona.listlayers('/tmp/foo'))

with fiona.open('/tmp/foo', layer='bar') as src:
    print(len(src))
    f = next(src)
    print(f['geometry']['type'])
    print(f['properties'])

    # Output:
    # [u'bar']
    # 1
    # Polygon
    # OrderedDict([(u'PERIMETER', 1.22107), (u'FEATURE2', None), (u'NAME', u'Mount␣
↪Naomi Wilderness'), (u'FEATURE1', u'Wilderness'), (u'URL', u'http://www.wilderness.
↪net/index.cfm?fuse=NWPS&sec=wildView&wname=Mount%20Naomi'), (u'AGBUR', u'FS'), (u
↪'AREA', 0.0179264), (u'STATE_FIPS', u'49'), (u'WILDRNP020', 332), (u'STATE', u'UT
↪')])
```

A view of the /tmp/foo directory will confirm the creation of the new files.

```
$ ls /tmp/foo
bar.cpg bar.dbf bar.prj bar.shp bar.shx
```

1.1.4 Collections from archives and virtual file systems

Zip and Tar archives can be treated as virtual filesystems and Collections can be made from paths and layers within them. In other words, Fiona lets you read and write zipped Shapefiles.

```
for i, layername in enumerate(
        fiona.listlayers('zip://tests/data/coutwildrnp.zip')):
    with fiona.open('zip://tests/data/coutwildrnp.zip', layer=i) as src:
        print(i, layername, len(src))

# Output:
# (0, u'coutwildrnp', 67)
```

Fiona can also read from more exotic file systems. For instance, a zipped shape file in S3 can be accessed like so:

```
with fiona.open('zip+s3://mapbox/rasterio/coutwildrnp.zip') as src:
    print(len(src))

# Output:
# 67
```

1.2 Fiona CLI

Fiona's command line interface, named "fio", is documented at docs/cli.rst. Its `fio info` pretty prints information about a data file.

```
$ fio info --indent 2 tests/data/coutwildrnp.shp
{
  "count": 67,
  "crs": "EPSG:4326",
  "driver": "ESRI Shapefile",
  "bounds": [
    -113.56424713134766,
    37.0689811706543,
    -104.97087097167969,
    41.99627685546875
  ],
  "schema": {
    "geometry": "Polygon",
    "properties": {
      "PERIMETER": "float:24.15",
      "FEATURE2": "str:80",
      "NAME": "str:80",
      "FEATURE1": "str:80",
      "URL": "str:101",
      "AGBUR": "str:80",
      "AREA": "float:24.15",
      "STATE_FIPS": "str:80",
      "WILDRNP020": "int:10",
      "STATE": "str:80"
    }
  }
}
```

1.3 Installation

Fiona requires Python versions 2.7 or 3.4+ and GDAL version 1.11-2.4. GDAL version 3 is not yet supported. To build from a source distribution you will need a C compiler and GDAL and Python development headers and libraries (libgdal1-dev for Debian/Ubuntu, gdal-dev for CentOS/Fedora).

To build from a repository copy, you will also need Cython to build C sources from the project's .pyx files. See the project's requirements-dev.txt file for guidance.

The Kyngchaos GDAL frameworks will satisfy the GDAL/OGR dependency for OS X, as will Homebrew's GDAL Formula (`brew install gdal`).

1.3.1 Python Requirements

Fiona depends on the modules `enum34`, `six`, `cligj`, `munch`, `argparse`, and `ordereddict` (the two latter modules are standard in Python 2.7+). Pip will fetch these requirements for you, but users installing Fiona from a Windows installer must get them separately.

1.3.2 Unix-like systems

Assuming you're using a virtualenv (if not, skip to the 4th command) and GDAL/OGR libraries, headers, and gdal-config program are installed to well known locations on your system via your system's package manager (`brew install gdal` using Homebrew on OS X), installation is this simple.

```
$ mkdir fiona_env
$ virtualenv fiona_env
$ source fiona_env/bin/activate
(fiona_env)$ pip install fiona
```

If gdal-config is not available or if GDAL/OGR headers and libs aren't installed to a well known location, you must set include dirs, library dirs, and libraries options via the setup.cfg file or setup command line as shown below (using `git`). You must also specify the version of the GDAL API on the command line using the `--gdalversion` argument (see example below) or with the `GDAL_VERSION` environment variable (e.g. `export GDAL_VERSION=2.1`).

```
(fiona_env)$ git clone git://github.com/Toblerity/Fiona.git
(fiona_env)$ cd Fiona
(fiona_env)$ python setup.py build_ext -I/path/to/gdal/include -L/path/to/gdal/lib -
↪lgdal install --gdalversion 2.1
```

Or specify that build options and GDAL API version should be provided by a particular gdal-config program.

```
(fiona_env)$ GDAL_CONFIG=/path/to/gdal-config pip install fiona
```

1.3.3 Windows

Binary installers are available at http://www.lfd.uci.edu/~gohlke/pythonlibs/#fiona and coming eventually to PyPI.

You can download a binary distribution of GDAL from here. You will also need to download the compiled libraries and headers (include files).

When building from source on Windows, it is important to know that setup.py cannot rely on gdal-config, which is only present on UNIX systems, to discover the locations of header files and libraries that Fiona needs to compile its C extensions. On Windows, these paths need to be provided by the user. You will need to find the include files and the library files for gdal and use setup.py as follows. You must also specify the version of the GDAL API on the command line using the `--gdalversion` argument (see example below) or with the `GDAL_VERSION` environment variable (e.g. `set GDAL_VERSION=2.1`).

```
$ python setup.py build_ext -I<path to gdal include files> -lgdal_i -L<path to gdal␣
↪library> install --gdalversion 2.1
```

Note: The GDAL DLL (`gdal111.dll` or similar) and gdal-data directory need to be in your Windows PATH otherwise Fiona will fail to work.

The Appveyor CI build uses the GISInternals GDAL binaries to build Fiona. This produces a binary wheel for successful builds, which includes GDAL and other dependencies, for users wanting to try an unstable development version. The Appveyor configuration file may be a useful example for users building from source on Windows.

1.4 Development and testing

Building from the source requires Cython. Tests require pytest. If the GDAL/OGR libraries, headers, and gdal-config program are installed to well known locations on your system (via your system's package manager), you can do this:

```
(fiona_env)$ git clone git://github.com/Toblerity/Fiona.git
(fiona_env)$ cd Fiona
(fiona_env)$ pip install cython
(fiona_env)$ pip install -e .[test]
(fiona_env)$ py.test
```

Or you can use the `pep-518-install` script:

```
(fiona_env)$ git clone git://github.com/Toblerity/Fiona.git
(fiona_env)$ cd Fiona
(fiona_env)$ ./pep-518-install
```

If you have a non-standard environment, you'll need to specify the include and lib dirs and GDAL library on the command line:

```
(fiona_env)$ python setup.py build_ext -I/path/to/gdal/include -L/path/to/gdal/lib -
↪lgdal --gdalversion 2 develop
(fiona_env)$ py.test
```

1.5 Changes

All issue numbers are relative to https://github.com/Toblerity/Fiona/issues.

1.5.1 1.8.6 (2019-03-18)

- The advertisement for JSON driver enablement in 1.8.5 was false (#176), but in this release they are ready for use.

1.5.2 1.8.5 (2019-03-15)

- GDAL seems to work best if GDAL_DATA is set as early as possible. Ideally it is set when building the library or in the environment before importing Fiona, but for wheels we patch GDAL_DATA into os.environ when fiona.env is imported. This resolves #731.
- A combination of bugs which allowed .cpg files to be overlooked has been fixed (#726).
- On entering a collection context (Collection.__enter__) a new anonymous GDAL environment is created if needed and entered. This makes *with fiona.open(...) as collection:* roughly equivalent to *with fiona.open(...) as collection, Env():*. This helps prevent bugs when Collections are created and then used later or in different scopes.
- Missing GDAL support for TopoJSON, GeoJSONSeq, and ESRIJSON has been enabled (#721).
- A regression in handling of polygons with M values (#724) has been fixed.
- Per-feature debug logging calls in OGRFeatureBuilder methods have been eliminated to improve feature writing performance (#718).
- Native support for datasets in Google Cloud Storage identified by "gs" resource names has been added (#709).

- Support has been added for triangle, polyhedral surface, and TIN geometry types (#679).
- Notes about using the MemoryFile and ZipMemoryFile classes has been added to the manual (#674).

1.5.3 1.8.4 (2018-12-10)

- 3D geometries can now be transformed with a specified precision (#523).
- A bug producing a spurious DriverSupportError for Shapefiles with a "time" field (#692) has been fixed.
- Patching of the GDAL_DATA environment variable was accidentally left in place in 1.8.3 and now has been removed.

1.5.4 1.8.3 (2018-11-30)

- The RASTERIO_ENV config environment marker this project picked up from Rasterio has been renamed to FIONA_ENV (#665).
- Options –gdal-data and –proj-data have been added to the fio-env command so that users of Rasterio wheels can get paths to set GDAL_DATA and PROJ_LIB environment variables.
- The unsuccessful attempt to make GDAL and PROJ support file discovery and configuration automatic within collection's crs and crs_wkt properties has been reverted. Users must execute such code inside a *with Env()* block or set the GDAL_DATA and PROJ_LIB environment variables needed by GDAL.

1.5.5 1.8.2 (2018-11-19)

Bug fixes:

- Raise FionaValueError when an iterator's __next__ is called and the session is found to be missing or inactive instead of passing a null pointer to OGR_L_GetNextFeature (#687).

1.5.6 1.8.1 (2018-11-15)

Bug fixes:

- Add checks around OSRGetAuthorityName and OSRGetAuthorityCode calls that will log problems with looking up these items.
- Opened data sources are now released before we raise exceptions in WritingSession.start (#676). This fixes an issue with locked files on Windows.
- We now ensure that an Env instance exists when getting the crs or crs_wkt properties of a Collection (#673, #690). Otherwise, required GDAL and PROJ data files included in Fiona wheels can not be found.
- GDAL and PROJ data search has been refactored to improve testability (#678).
- In the project's Cython code, void* pointers have been replaced with proper GDAL types (#672).
- Pervasive warning level log messages about ENCODING creation options (#668) have been eliminated.

1.5.7 1.8.0 (2018-10-31)

This is the final 1.8.0 release. Thanks, everyone!

Bug fixes:

- We cpdef Session.stop so that it has a C version that can be called safely from __dealloc__, fixing a PyPy issue (#659, #553).

1.5.8 1.8rc1 (2018-10-26)

There are no changes in 1.8rc1 other than more test standardization and the introduction of a temporary test_collection_legacy.py module to support the build of fully tested Python 2.7 macosx wheels on Travis-CI.

1.5.9 1.8b2 (2018-10-23)

Bug fixes:

- The ensure_env_with_credentials decorator will no longer clobber credentials of the outer environment. This fixes a bug reported to the Rasterio project and which also existed in Fiona.
- An unused import of the packaging module and the dependency have been removed (#653).
- The Env class logged to the 'rasterio' hierarchy instead of 'fiona'. This mistake has been corrected (#646).
- The Mapping abstract base class is imported from collections.abc when possible (#647).

Refactoring:

- Standardization of the tests on pytest functions and fixtures continues and is nearing completion (#648, #649, #650, #651, #652).

1.5.10 1.8b1 (2018-10-15)

Deprecations:

- Collection slicing has been deprecated and will be prohibited in a future version.

Bug fixes:

- Rasterio CRS objects passed to transform module methods will be converted to dicts as needed (#590).
- Implicitly convert curve geometries to their linear approximations rather than failing (#617).
- Migrated unittest test cases in test_collection.py and test_layer.py to the use of the standard data_dir and path_coutwildrnp_shp fixtures (#616).
- Root logger configuration has been removed from all test scripts (#615).
- An AWS session is created for the CLI context Env only if explicitly requested, matching the behavior of Rasterio's CLI (#635).
- Dependency on attrs is made explicit.
- Other dependencies are pinned to known good versions in requirements files.
- Unused arguments have been removed from the Env constructor (#637).

Refactoring:

- A with_context_env decorator has been added and used to set up the GDAL environment for CLI commands. The command functions themselves are now simplified.

1.5.11 1.8a3 (2018-10-01)

Deprecations:

- The `fiona.drivers()` context manager is officially deprecated. All users should switch to `fiona.Env()`, which registers format drivers and manages GDAL configuration in a reversible manner.

Bug fixes:

- The Collection class now filters log messages about skipped fields to a maximum of one warning message per field (#627).
- The boto3 module is only imported when needed (#507, #629).
- Compatibility with Click 7.0 is achieved (#633).
- Use of %r instead of %s in a debug() call prevents UnicodeDecodeErrors (#620).

1.5.12 1.8a2 (2018-07-24)

New features:

- 64-bit integers are the now the default for int type fields (#562, #564).
- 'http', 's3', 'zip+http', and 'zip+s3' URI schemes for datasets are now supported (#425, #426).
- We've added a `MemoryFile` class which supports formatted in-memory feature collections (#501).
- Added support for GDAL 2.x boolean field sub-type (#531).
- A new `fio rm` command makes it possible to cleanly remove multi-file datasets (#538).
- The geometry type in a feature collection is more flexible. We can now specify not only a single geometry type, but a sequence of permissible types, or "Any" to permit any geometry type (#539).
- Support for GDAL 2.2+ null fields has been added (#554).
- The new `gdal_open_vector()` function of our internal API provides much improved error handling (#557).

Bug fixes:

- The bug involving OrderedDict import on Python 2.7 has been fixed (#533).
- An `AttributeError` raised when the `--bbox` option of fio-cat is used with more than one input file has been fixed (#543, #544).
- Obsolete and derelict fiona.tool module has been removed.
- Revert the change in 0a2bc7c that discards Z in geometry types when a collection's schema is reported (#541).
- Require six version 1.7 or higher (#550).
- A regression related to "zip+s3" URIs has been fixed.
- Debian's GDAL data locations are now searched by default (#583).

1.5.13 1.8a1 (2017-11-06)

New features:

- Each call of `writerecords()` involves one or more transactions of up to 20,000 features each. This improves performance when writing GeoPackage files as the previous transaction size was only 200 features (#476, #491).

Packaging:

- Fiona's Cython source files have been refactored so that there are no longer separate extension modules for GDAL 1.x and GDAL 2.x. Instead there is a base extension module based on GDAL 2.x and shim modules for installations that use GDAL 1.x.

1.5.14 1.7.11.post1 (2018-01-08)

- This post-release adds missing expat (and thereby GPX format) support to the included GDAL library (still version 2.2.2).

1.5.15 1.7.11 (2017-12-14)

- The `encoding` keyword argument for `fiona.open()`, which is intended to allow a caller to override a data source's own and possibly erroneous encoding, has not been working (#510, #512). The problem is that we weren't always setting GDAL open or config options before opening the data sources. This bug is resolved by a number of commits in the maint-1.7 branch and the fix is demonstrated in tests/test_encoding.py.
- An `--encoding` option has been added to fio-load to enable creation of encoded shapefiles with an accompanying .cpg file (#499, #517).

1.5.16 1.7.10.post1 (2017-10-30)

- A post-release has been made to fix a problem with macosx wheels uploaded to PyPI.

1.5.17 1.7.10 (2017-10-26)

Bug fixes:

- An extraneous printed line from the `rio cat --layers` validator has been removed (#478).

Packaging:

- Official OS X and Manylinux1 wheels (on PyPI) for this release will be compatible with Shapely 1.6.2 and Rasterio 1.0a10 wheels.

1.5.18 1.7.9.post1 (2017-08-21)

This release introduces no changes in the Fiona package. It upgrades GDAL from 2.2.0 to 2.2.1 in wheels that we publish to the Python Package Index.

1.5.19 1.7.9 (2017-08-17)

Bug fixes:

- Acquire the GIL for GDAL error callback functions to prevent crashes when GDAL errors occur when the GIL has been released by user code.
- Sync and flush layers when closing even when the number of features is not precisely known (#467).

1.5.20 1.7.8 (2017-06-20)

Bug fixes:

- Provide all arguments needed by CPLError based exceptions (#456).

1.5.21 1.7.7 (2017-06-05)

Bug fixes:

- Switch logger *warn()* (deprecated) calls to *warning()*.
- Replace all relative imports and cimports in Cython modules with absolute imports (#450).
- Avoid setting *PROJ_LIB* to a non-existent directory (#439).

1.5.22 1.7.6 (2017-04-26)

Bug fixes:

- Fall back to *share/proj* for PROJ_LIB (#440).
- Replace every call to *OSRDestroySpatialReference()* with *OSRRelease()*, fixing the GPKG driver crasher reported in #441 (#443).
- Add a *DriverIOError* derived from *IOError* to use for driver-specific errors such as the GeoJSON driver's refusal to overwrite existing files. Also we now ensure that when this error is raised by *fiona.open()* any created read or write session is deleted, this eliminates spurious exceptions on teardown of broken *Collection* objects (#437, #444).

1.5.23 1.7.5 (2017-03-20)

Bug fixes:

- Opening a data file in read (the default) mode with *fiona.open()* using the the *driver* or *drivers* keyword arguments (to specify certain format drivers) would sometimes cause a crash on Windows due to improperly terminated lists of strings (#428). The fix: Fiona's buggy *string_list()* has been replaced by GDAL's *CSLAddString()*.

1.5.24 1.7.4 (2017-02-20)

Bug fixes:

- OGR's EsriJSON detection fails when certain keys aren't found in the first 6000 bytes of data passed to *BytesCollection* (#422). A .json file extension is now explicitly given to the in-memory file behind *BytesCollection* when the *driver='GeoJSON'* keyword argument is given (#423).

1.5.25 1.7.3 (2017-02-14)

Roses are red. Tan is a pug. Software regression's the most embarrassing bug.

Bug fixes:

- Use __stdcall for GDAL error handling callback on Windows as in Rasterio.
- Turn on latent support for zip:// URLs in rio-cat and rio-info (#421).
- The 1.7.2 release broke support for zip files with absolute paths (#418). This regression has been fixed with tests to confirm.

1.5.26 1.7.2 (2017-01-27)

Future Deprecation:

- *Collection.__next__()* is buggy in that it can lead to duplication of features when used in combination with *Collection.filter()* or *Collection.__iter__()*. It will be removed in Fiona 2.0. Please check for usage of this deprecated feature by running your tests or programs with *PYTHONWARNINGS="always:::fiona"* or *-W"always:::fiona"* and switch from *next(collection)* to *next(iter(collection))* (#301).

Bug fix:

- Zipped streams of bytes can be accessed by *BytesCollection* (#318).

1.5.27 1.7.1.post1 (2016-12-23)

- New binary wheels using version 1.2.0 of sgillies/frs-wheel-builds. See https://github.com/sgillies/frs-wheel-builds/blob/master/CHANGES.txt.

1.5.28 1.7.1 (2016-11-16)

Bug Fixes:

- Prevent Fiona from stumbling over 'Z', 'M', and 'ZM' geometry types introduced in GDAL 2.1 (#384). Fiona 1.7.1 doesn't add explicit support for these types, they are coerced to geometry types 1-7 ('Point', 'LineString', etc.)
- Raise an *UnsupportedGeometryTypeError* when a bogus or unsupported geometry type is encountered in a new collection's schema or elsewhere (#340).
- Enable *–precision 0* for fio-cat (#370).
- Prevent datetime exceptions from unnecessarily stopping collection iteration by yielding *None* (#385)
- Replace log.warn calls with log.warning calls (#379).
- Print an error message if neither gdal-config or *–gdalversion* indicate a GDAL C API version when running *setup.py* (#364).
- Let dict-like subclasses through CRS type checks (#367).

1.5.29 1.7.0post2 (2016-06-15)

Packaging: define extension modules for 'clean' and 'config' targets (#363).

1.5.30 1.7.0post1 (2016-06-15)

Packaging: No files are copied for the 'clean' setup target (#361, #362).

1.5.31 1.7.0 (2016-06-14)

The C extension modules in this library can now be built and used with either a 1.x or 2.x release of the GDAL library. Big thanks to René Buffat for leading this effort.

Refactoring:

- The *ogrext1.pyx* and *ogrext2.pyx* files now use separate C APIs defined in *ogrext1.pxd* and *ogrex2.pxd*. The other extension modules have been refactored so that they do not depend on either of these modules and use subsets of the GDAL/OGR API compatible with both GDAL 1.x and 2.x (#359).

Packaging:

- Source distributions now contain two different sources for the *ogrext* extension module. The *ogrext1.c* file will be used with GDAL 1.x and the *ogrext2.c* file will be used with GDAL 2.x.

1.5.32 1.7b2 (2016-06-13)

- New feature: enhancement of the *–layer* option for fio-cat and fio-dump to allow separate layers of one or more multi-layer input files to be selected (#349).

1.5.33 1.7b1 (2016-06-10)

- New feature: support for GDAL version 2+ (#259).
- New feature: a new fio-calc CLI command (#273).
- New feature: *–layer* options for fio-info (#316) and fio-load (#299).
- New feature: a *–no-parse* option for fio-collect that lets a careful user avoid extra JSON serialization and deserialization (#306).
- Bug fix: *+wktext* is now preserved when serializing CRS from WKT to PROJ.4 dicts (#352).
- Bug fix: a small memory leak when opening a collection has been fixed (#337).
- Bug fix: internal unicode errors now result in a log message and a *UnicodeError* exception, not a *TypeError* (#356).

1.5.34 1.6.4 (2016-05-06)

- Raise ImportError if the active GDAL library version is >= 2.0 instead of failing unpredictably (#338, #341). Support for GDAL>=2.0 is coming in Fiona 1.7.

1.5.35 1.6.3.post1 (2016-03-27)

- No changes to the library in this post-release version, but there is a significant change to the distributions on PyPI: to help make Fiona more compatible with Shapely on OS X, the GDAL shared library included in the macosx (only) binary wheels now statically links the GEOS library. See https://github.com/sgillies/frs-wheel-builds/issues/5.

1.5.36 1.6.3 (2015-12-22)

- Daytime has been decreasing in the Northern Hemisphere, but is now increasing again as it should.
- Non-UTF strings were being passed into OGR functions in some situations and on Windows this would sometimes crash a Python process (#303). Fiona now raises errors derived from UnicodeError when field names or field values can't be encoded.

1.5.37 1.6.2 (2015-09-22)

- Providing only PROJ4 representations in the dataset meta property resulted in loss of CRS information when using the *fiona.open(..., **src.meta) as dst* pattern (#265). This bug has been addressed by adding a crs_wkt item to the' meta property and extending the *fiona.open()* and the collection constructor to look for and prioritize this keyword argument.

1.5.38 1.6.1 (2015-08-12)

- Bug fix: Fiona now deserializes JSON-encoded string properties provided by the OGR GeoJSON driver (#244, #245, #246).
- Bug fix: proj4 data was not copied properly into binary distributions due to a typo (#254).

Special thanks to WFMU DJ Liz Berg for the awesome playlist that's fueling my release sprint. Check it out at http://wfmu.org/playlists/shows/62083. You can't unhear Love Coffin.

1.5.39 1.6.0 (2015-07-21)

- Upgrade Cython requirement to 0.22 (#214).
- New BytesCollection class (#215).
- Add GDAL's OpenFileGDB driver to registered drivers (#221).
- Implement CLI commands as plugins (#228).
- Raise click.abort instead of calling sys.exit, preventing suprising exits (#236).

1.5.40 1.5.1 (2015-03-19)

- Restore test data to sdists by fixing MANIFEST.in (#216).

1.5.41 1.5.0 (2015-02-02)

- Finalize GeoJSON feature sequence options (#174).
- Fix for reading of datasets that don't support feature counting (#190).
- New test dataset (#188).
- Fix for encoding error (#191).
- Remove confusing warning (#195).
- Add data files for binary wheels (#196).
- Add control over drivers enabled when reading datasets (#203).

- Use cligj for CLI options involving GeoJSON (#204).
- Fix fio-info –bounds help (#206).

1.5.42 1.4.8 (2014-11-02)

- Add missing crs_wkt property as in Rasterio (#182).

1.5.43 1.4.7 (2014-10-28)

- Fix setting of CRS from EPSG codes (#149).

1.5.44 1.4.6 (2014-10-21)

- Handle 3D coordinates in bounds() #178.

1.5.45 1.4.5 (2014-10-18)

- Add –bbox option to fio-cat (#163).
- Skip geopackage tests if run from an sdist (#167).
- Add fio-bounds and fio-distrib.
- Restore fio-dump to working order.

1.5.46 1.4.4 (2014-10-13)

- Fix accidental requirement on GDAL 1.11 introduced in 1.4.3 (#164).

1.5.47 1.4.3 (2014-10-10)

- Add support for geopackage format (#160).
- Add -f and –format aliases for –driver in CLI (#162).
- Add –version option and env command to CLI.

1.5.48 1.4.2 (2014-10-03)

- –dst-crs and –src-crs options for fio cat and collect (#159).

1.5.49 1.4.1 (2014-09-30)

- Fix encoding bug in collection's __getitem__ (#153).

1.5.50 1.4.0 (2014-09-22)

- Add fio cat and fio collect commands (#150).
- Return of Python 2.6 compatibility (#148).
- Improved CRS support (#149).

1.5.51 1.3.0 (2014-09-17)

- Add single metadata item accessors to fio inf (#142).
- Move fio to setuptools entry point (#142).
- Add fio dump and load commands (#143).
- Remove fio translate command.

1.5.52 1.2.0 (2014-09-02)

- Always show property width and precision in schema (#123).
- Write datetime properties of features (#125).
- Reset spatial filtering in filter() (#129).
- Accept datetime.date objects as feature properties (#130).
- Add slicing to collection iterators (#132).
- Add geometry object masks to collection iterators (#136).
- Change source layout to match Shapely and Rasterio (#138).

1.5.53 1.1.6 (2014-07-23)

- Implement Collection __getitem__() (#112).
- Leave GDAL finalization to the DLL's destructor (#113).
- Add Collection keys(), values(), items(), __contains__() (#114).
- CRS bug fix (#116).
- Add fio CLI program.

1.5.54 1.1.5 (2014-05-21)

- Addition of cpl_errs context manager (#108).
- Check for NULLs with '==' test instead of 'is' (#109).
- Open auxiliary files with encoding='utf-8' in setup for Python 3 (#110).

1.5.55 1.1.4 (2014-04-03)

- Convert 'long' in schemas to 'int' (#101).
- Carefully map Python schema to the possibly munged internal schema (#105).
- Allow writing of features with geometry: None (#71).

1.5.56 1.1.3 (2014-03-23)

- Always register all GDAL and OGR drivers when entering the DriverManager context (#80, #92).
- Skip unsupported field types with a warning (#91).
- Allow OGR config options to be passed to fiona.drivers() (#90, #93).
- Add a bounds() function (#100).
- Turn on GPX driver.

1.5.57 1.1.2 (2014-02-14)

- Remove collection slice left in dumpgj (#88).

1.5.58 1.1.1 (2014-02-02)

- Add an interactive file inspector like the one in rasterio.
- CRS to_string bug fix (#83).

1.5.59 1.1 (2014-01-22)

- Use a context manager to manage drivers (#78), a backwards compatible but big change. Fiona is now compatible with rasterio and plays better with the osgeo package.

1.5.60 1.0.3 (2014-01-21)

- Fix serialization of +init projections (#69).

1.5.61 1.0.2 (2013-09-09)

- Smarter, better test setup (#65, #66, #67).
- Add type='Feature' to records read from a Collection (#68).
- Skip geometry validation when using GeoJSON driver (#61).
- Dumpgj file description reports record properties as a list (as in dict.items()) instead of a dict.

1.5.62 1.0.1 (2013-08-16)

- Allow ordering of written fields and preservation of field order when reading (#57).

1.5.63 1.0 (2013-07-30)

- Add prop_type() function.
- Allow UTF-8 encoded paths for Python 2 (#51). For Python 3, paths must always be str, never bytes.
- Remove encoding from collection.meta, it's a file creation option only.
- Support for linking GDAL frameworks (#54).

1.5.64 0.16.1 (2013-07-02)

- Add listlayers, open, prop_width to __init__py:__all__.
- Reset reading of OGR layer whenever we ask for a collection iterator (#49).

1.5.65 0.16 (2013-06-24)

- Add support for writing layers to multi-layer files.
- Add tests to reach 100% Python code coverage.

1.5.66 0.15 (2013-06-06)

- Get and set numeric field widths (#42).
- Add support for multi-layer data sources (#17).
- Add support for zip and tar virtual filesystems (#45).
- Add listlayers() function.
- Add GeoJSON to list of supported formats (#47).
- Allow selection of layers by index or name.

1.5.67 0.14 (2013-05-04)

- Add option to add JSON-LD in the dumpgj program.
- Compare values to six.string_types in Collection constructor.
- Add encoding to Collection.meta.
- Document dumpgj in README.

1.5.68 0.13 (2013-04-30)

- Python 2/3 compatibility in a single package. Pythons 2.6, 2.7, 3.3 now supported.

1.5.69 0.12.1 (2013-04-16)

- Fix messed up linking of README in sdist (#39).

1.5.70 0.12 (2013-04-15)

- Fix broken installation of extension modules (#35).
- Log CPL errors at their matching Python log levels.
- Use upper case for encoding names within OGR, lower case in Python.

1.5.71 0.11 (2013-04-14)

- Cythonize .pyx files (#34).
- Work with or around OGR's internal recoding of record data (#35).
- Fix bug in serialization of int/float PROJ.4 params.

1.5.72 0.10 (2013-03-23)

- Add function to get the width of str type properties.
- Handle validation and schema representation of 3D geometry types (#29).
- Return {'geometry': None} in the case of a NULL geometry (#31).

1.5.73 0.9.1 (2013-03-07)

- Silence the logger in ogrext.so (can be overridden).
- Allow user specification of record field encoding (like 'Windows-1252' for Natural Earth shapefiles) to help when OGR can't detect it.

1.5.74 0.9 (2013-03-06)

- Accessing file metadata (crs, schema, bounds) on never inspected closed files returns None without exceptions.
- Add a dict of supported_drivers and their supported modes.
- Raise ValueError for unsupported drivers and modes.
- Remove asserts from ogrext.pyx.
- Add validate_record method to collections.
- Add helpful coordinate system functions to fiona.crs.
- Promote use of fiona.open over fiona.collection.
- Handle Shapefile's mix of LineString/Polygon and multis (#18).
- Allow users to specify width of shapefile text fields (#20).

1.5.75 0.8 (2012-02-21)

- Replaced .opened attribute with .closed (product of collection() is always opened). Also a __del__() which will close a Collection, but still not to be depended upon.
- Added writerecords method.

- Added a record buffer and better counting of records in a collection.
- Manage one iterator per collection/session.
- Added a read-only bounds property.

1.5.76 0.7 (2012-01-29)

- Initial timezone-naive support for date, time, and datetime fields. Don't use these field types if you can avoid them. RFC 3339 datetimes in a string field are much better.

1.5.77 0.6.2 (2012-01-10)

- Diagnose and set the driver property of collection in read mode.
- Fail if collection paths are not to files. Multi-collection workspaces are a (maybe) TODO.

1.5.78 0.6.1 (2012-01-06)

- Handle the case of undefined crs for disk collections.

1.5.79 0.6 (2012-01-05)

- Support for collection coordinate reference systems based on Proj4.
- Redirect OGR warnings and errors to the Fiona log.
- Assert that pointers returned from the ograpi functions are not NULL before using.

1.5.80 0.5 (2011-12-19)

- Support for reading and writing collections of any geometry type.
- Feature and Geometry classes replaced by mappings (dicts).
- Removal of Workspace class.

1.5.81 0.2 (2011-09-16)

- Rename WorldMill to Fiona.

1.5.82 0.1.1 (2008-12-04)

- Support for features with no geometry.

1.6 Credits

Fiona is written by:

- Sean Gillies <sean.gillies@gmail.com>
- René Buffat <buffat@gmail.com>
- Joshua Arnott <josh@snorfalorpagus.net>
- Kevin Wurster <wursterk@gmail.com>
- Micah Cochran <micahcochran@users.noreply.github.com>
- Matthew Perry <perrygeo@gmail.com>
- Elliott Sales de Andrade <quantum.analyst@gmail.com>
- Kelsey Jordahl <kjordahl@enthought.com>
- Patrick Young <patrick.mckendree.young@gmail.com>
- Simon Norris <snorris@hillcrestgeo.ca>
- Hannes Gräuler <graeuler@geoplex.de>
- Johan Van de Wauw <johan.vandewauw@gmail.com>
- Jacob Wasserman <jwasserman@gmail.com>
- Michael Weisman <mweisman@gmail.com>
- Ryan Grout <rgrout@continuum.io>
- Bas Couwenberg <sebastic@xs4all.nl>
- Brendan Ward <bcward@consbio.org>
- Hannes <kannes@users.noreply.github.com>
- Michele Citterio <michele@citterio.net>
- Miro Hrončok <miro@hroncok.cz>
- Sid Kapur <sid-kap@users.noreply.github.com>
- Tim Tröndle <tim.troendle@usys.ethz.ch>
- fredj <frederic.junod@camptocamp.com>
- qinfeng <guo.qinfeng+github@gmail.com>
- Ariel Nunez <ingenieroariel@gmail.com>
- Ariki <Ariki@users.noreply.github.com>
- Brandon Liu <bdon@bdon.org>
- Chris Mutel <cmutel@gmail.com>
- Denis Rykov <rykovd@gmail.com>
- Efrén <chefren@users.noreply.github.com>
- Egor Fedorov <egor.fedorov@emlid.com>
- Even Rouault <even.rouault@mines-paris.org>
- Filipe Fernandes <ocefpaf@gmail.com>

- Géraud <galak75@users.noreply.github.com>
- Hannes Gräuler <hgraeule@uos.de>
- Jesse Crocker <jesse@gaiagps.com>
- Juan Luis Cano Rodríguez <Juanlu001@users.noreply.github.com>
- Ludovic Delauné <ludotux@gmail.com>
- Martijn Visser <mgvisser@gmail.com>
- Matthew Perry <perrygeo@users.noreply.github.com>
- Michael Weisman <michael@urbanmapping.com>
- Oliver Tonnhofer <olt@bogosoft.com>
- Stefano Costa <steko@iosa.it>
- Stephane Poss <stephposs@gmail.com>
- dimlev <dimlev@gmail.com>
- wilsaj <wilson.andrew.j+github@gmail.com>

The GeoPandas project (Joris Van den Bossche et al.) has been a major driver for new features in 1.8.0.

Fiona would not be possible without the great work of Frank Warmerdam and other GDAL/OGR developers.

Some portions of this work were supported by a grant (for Pleiades) from the U.S. National Endowment for the Humanities (http://www.neh.gov).

CHAPTER

TWO

1 THE FIONA USER MANUAL

Author Sean Gillies, <sean.gillies@gmail.com>

Version 1.8.6

Date Sep 03, 2019

Abstract Fiona is OGR's neat, nimble, no-nonsense API. This document explains how to use the Fiona package for reading and writing geospatial data files. Python 3 is used in examples. See the README for installation and quick start instructions.

2.1 1.1 Introduction

Geographic information systems (GIS) help us plan, react to, and understand changes in our physical, political, economic, and cultural landscapes. A generation ago, GIS was something done only by major institutions like nations and cities, but it's become ubiquitous today thanks to accurate and inexpensive global positioning systems, commoditization of satellite imagery, and open source software.

The kinds of data in GIS are roughly divided into *rasters* representing continuous scalar fields (land surface temperature or elevation, for example) and *vectors* representing discrete entities like roads and administrative boundaries. Fiona is concerned exclusively with the latter. It is a Python wrapper for vector data access functions from the OGR library. A very simple wrapper for minimalists. It reads data records from files as GeoJSON-like mappings and writes the same kind of mappings as records back to files. That's it. There are no layers, no cursors, no geometric operations, no transformations between coordinate systems, no remote method calls; all these concerns are left to other Python packages such as `Shapely` and `pyproj` and Python language protocols. Why? To eliminate unnecessary complication. Fiona aims to be simple to understand and use, with no gotchas.

Please understand this: Fiona is designed to excel in a certain range of tasks and is less optimal in others. Fiona trades memory and speed for simplicity and reliability. Where OGR's Python bindings (for example) use C pointers, Fiona copies vector data from the data source to Python objects. These are simpler and safer to use, but more memory intensive. Fiona's performance is relatively more slow if you only need access to a single record field – and of course if you just want to reproject or filter data files, nothing beats the **`ogr2ogr`** program – but Fiona's performance is much better than OGR's Python bindings if you want *all* fields and coordinates of a record. The copying is a constraint, but it simplifies programs. With Fiona, you don't have to track references to C objects to avoid crashes, and you can work with vector data using familiar Python mapping accessors. Less worry, less time spent reading API documentation.

2.1.1 1.1.1 Rules of Thumb

In what cases would you benefit from using Fiona?

- If the features of interest are from or destined for a file in a non-text format like ESRI Shapefiles, Mapinfo TAB files, etc.
- If you're more interested in the values of many feature properties than in a single property's value.
- If you're more interested in all the coordinate values of a feature's geometry than in a single value.
- If your processing system is distributed or not contained to a single process.

In what cases would you not benefit from using Fiona?

- If your data is in or destined for a JSON document you should use Python's `json` or `simplejson` modules.
- If your data is in a RDBMS like PostGIS, use a Python DB package or ORM like `SQLAlchemy` or `GeoAlchemy`. Maybe you're using `GeoDjango` already. If so, carry on.
- If your data is served via HTTP from CouchDB or CartoDB, etc, use an HTTP package (`httplib2`, `Requests`, etc) or the provider's Python API.
- If you can use **`ogr2ogr`**, do so.

2.1.2 1.1.2 Example

The first example of using Fiona is this: copying records from one file to another, adding two attributes and making sure that all polygons are facing "up". Orientation of polygons is significant in some applications, extruded polygons in Google Earth for one. No other library (like `Shapely`) is needed here, which keeps it uncomplicated. There's a `test_uk` file in the Fiona repository for use in this and other examples.

```
import datetime
import logging
import sys

import fiona

logging.basicConfig(stream=sys.stderr, level=logging.INFO)

def signed_area(coords):
    """Return the signed area enclosed by a ring using the linear time
    algorithm at http://www.cgafaq.info/wiki/Polygon_Area. A value >= 0
    indicates a counter-clockwise oriented ring.
    """
    xs, ys = map(list, zip(*coords))
    xs.append(xs[1])
    ys.append(ys[1])
    return sum(xs[i]*(ys[i+1]-ys[i-1]) for i in range(1, len(coords)))/2.0

with fiona.open('docs/data/test_uk.shp', 'r') as source:

    # Copy the source schema and add two new properties.
    sink_schema = source.schema
    sink_schema['properties']['s_area'] = 'float'
    sink_schema['properties']['timestamp'] = 'datetime'

    # Create a sink for processed features with the same format and
    # coordinate reference system as the source.
    with fiona.open(
            'oriented-ccw.shp', 'w',
            crs=source.crs,
            driver=source.driver,
```

(continues on next page)

(continued from previous page)

```
            schema=sink_schema,
            ) as sink:

        for f in source:

            try:

                # If any feature's polygon is facing "down" (has rings
                # wound clockwise), its rings will be reordered to flip
                # it "up".
                g = f['geometry']
                assert g['type'] == "Polygon"
                rings = g['coordinates']
                sa = sum(signed_area(r) for r in rings)
                if sa < 0.0:
                    rings = [r[::-1] for r in rings]
                    g['coordinates'] = rings
                    f['geometry'] = g

                # Add the signed area of the polygon and a timestamp
                # to the feature properties map.
                f['properties'].update(
                    s_area=sa,
                    timestamp=datetime.datetime.now().isoformat() )

                sink.write(f)

            except Exception, e:
                logging.exception("Error processing feature %s:", f['id'])

        # The sink file is written to disk and closed when its block ends.
```

2.2 1.2 Data Model

Discrete geographic features are usually represented in geographic information systems by *records*. The characteristics of records and their semantic implications are well known [Kent1978]. Among those most significant for geographic data: records have a single type, all records of that type have the same fields, and a record's fields concern a single geographic feature. Different systems model records in different ways, but the various models have enough in common that programmers have been able to create useful abstract data models. The OGR model is one. Its primary entities are *Data Sources*, *Layers*, and *Features*. Features have not fields, but attributes and a *Geometry*. An OGR Layer contains Features of a single type ("roads" or "wells", for example). The GeoJSON model is a bit more simple, keeping Features and substituting *Feature Collections* for OGR Data Sources and Layers. The term "Feature" is thus overloaded in GIS modeling, denoting entities in both our conceptual and data models.

Various formats for record files exist. The *ESRI Shapefile* [ESRI1998] has been, at least in the United States, the most significant of these up to about 2005 and remains popular today. It is a binary format. The shape fields are stored in one .shp file and the other fields in another .dbf file. The GeoJSON [GeoJSON] format, from 2008, proposed a human readable text format in which geometry and other attribute fields are encoded together using *Javascript Object Notation* [JSON]. In GeoJSON, there's a uniformity of data access. Attributes of features are accessed in the same manner as attributes of a feature collection. Coordinates of a geometry are accessed in the same manner as features of a collection.

The GeoJSON format turns out to be a good model for a Python API. JSON objects and Python dictionaries are semantically and syntactically similar. Replacing object-oriented Layer and Feature APIs with interfaces based on

Python mappings provides a uniformity of access to data and reduces the amount of time spent reading documentation. A Python programmer knows how to use a mapping, so why not treat features as dictionaries? Use of existing Python idioms is one of Fiona's major design principles.

TL;DR

Fiona subscribes to the conventional record model of data, but provides GeoJSON-like access to the data via Python file-like and mapping protocols.

2.3 1.3 Reading Vector Data

Reading a GIS vector file begins by opening it in mode `'r'` using Fiona's *open()* function. It returns an opened *Collection* object.

```
>>> import fiona
>>> c = fiona.open('docs/data/test_uk.shp', 'r')
>>> c
<open Collection 'docs/data/test_uk.shp:test_uk', mode 'r' at 0x...>
>>> c.closed
False
```

API Change

fiona.collection() is deprecated, but aliased to *fiona.open()* in version 0.9.

Mode `'r'` is the default and will be omitted in following examples.

Fiona's *Collection* is like a Python `file`, but is iterable for records rather than lines.

```
>>> next(c)
{'geometry': {'type': 'Polygon', 'coordinates': ...
>>> len(list(c))
48
```

Note that `list()` iterates over the entire collection, effectively emptying it as with a Python `file`.

```
>>> next(c)
Traceback (most recent call last):
...
StopIteration
>>> len(list(c))
0
```

Seeking the beginning of the file is not supported. You must reopen the collection to get back to the beginning.

```
>>> c = fiona.open('docs/data/test_uk.shp')
>>> len(list(c))
48
```

File Encoding

The format drivers will attempt to detect the encoding of your data, but may fail. In my experience GDAL 1.7.2 (for example) doesn't detect that the encoding of the Natural Earth dataset is Windows-1252. In this case,

the proper encoding can be specified explicitly by using the `encoding` keyword parameter of *fiona.open()*: `encoding='Windows-1252'`.

New in version 0.9.1.

2.3.1 1.3.1 Collection indexing

Features of a collection may also be accessed by index.

```
>>> import pprint
>>> with fiona.open('docs/data/test_uk.shp') as src:
...     pprint.pprint(src[1])
...
{'geometry': {'coordinates': [[(-4.663611, 51.158333),
                               (-4.669168, 51.159439),
                               (-4.673334, 51.161385),
                               (-4.674445, 51.165276),
                               (-4.67139, 51.185272),
                               (-4.669445, 51.193054),
                               (-4.665556, 51.195),
                               (-4.65889, 51.195),
                               (-4.656389, 51.192215),
                               (-4.646389, 51.164444),
                               (-4.646945, 51.160828),
                               (-4.651668, 51.159439),
                               (-4.663611, 51.158333)]],
              'type': 'Polygon'},
 'id': '1',
 'properties': OrderedDict([(u'CAT', 232.0), (u'FIPS_CNTRY', u'UK'), (u'CNTRY_NAME', u
↪'United Kingdom'), (u'AREA', 244820.0), (u'POP_CNTRY', 60270708.0)]),
 'type': 'Feature'}
```

Note that these indices are controlled by GDAL, and do not always follow Python conventions. They can start from 0, 1 (e.g. geopackages), or even other values, and have no guarantee of contiguity. Negative indices will only function correctly if indices start from 0 and are contiguous.

2.3.2 1.3.2 Closing Files

A *Collection* involves external resources. There's no guarantee that these will be released unless you explicitly *close()* the object or use a `with` statement. When a *Collection* is a context guard, it is closed no matter what happens within the block.

```
>>> try:
...     with fiona.open('docs/data/test_uk.shp') as c:
...         print(len(list(c)))
...         assert True is False
... except:
...     print(c.closed)
...     raise
...
48
True
Traceback (most recent call last):
  ...
AssertionError
```

An exception is raised in the `with` block above, but as you can see from the print statement in the `except` clause `c.__exit__()` (and thereby `c.close()`) has been called.

Important: Always call *close()* or use `with` and you'll never stumble over tied-up external resources, locked files, etc.

2.4 1.4 Format Drivers, CRS, Bounds, and Schema

In addition to attributes like those of `file` (`name`, `mode`, `closed`), a *Collection* has a read-only *driver* attribute which names the **OGR** *format driver* used to open the vector file.

```
>>> c = fiona.open('docs/data/test_uk.shp')
>>> c.driver
'ESRI Shapefile'
```

The *coordinate reference system* (CRS) of the collection's vector data is accessed via a read-only *crs* attribute.

```
>>> c.crs
{'no_defs': True, 'ellps': 'WGS84', 'datum': 'WGS84', 'proj': 'longlat'}
```

The CRS is represented by a mapping of **PROJ.4** parameters.

The *fiona.crs* module provides 3 functions to assist with these mappings. *to_string()* converts mappings to PROJ.4 strings:

```
>>> from fiona.crs import to_string
>>> print(to_string(c.crs))
+datum=WGS84 +ellps=WGS84 +no_defs +proj=longlat
```

from_string() does the inverse.

```
>>> from fiona.crs import from_string
>>> from_string("+datum=WGS84 +ellps=WGS84 +no_defs +proj=longlat")
{'no_defs': True, 'ellps': 'WGS84', 'datum': 'WGS84', 'proj': 'longlat'}
```

from_epsg() is a shortcut to CRS mappings from EPSG codes.

```
>>> from fiona.crs import from_epsg
>>> from_epsg(3857)
{'init': 'epsg:3857', 'no_defs': True}
```

No Validation

Both *from_epsg()* and *from_string()* simply restructure data, they do not ensure that the resulting mapping is a pre-defined or otherwise valid CRS in any way.

The number of records in the collection's file can be obtained via Python's built in `len()` function.

```
>>> len(c)
48
```

The *minimum bounding rectangle* (MBR) or *bounds* of the collection's records is obtained via a read-only *bounds* attribute.

```
>>> c.bounds
(-8.621389, 49.911659, 1.749444, 60.844444)
```

Finally, the schema of its record type (a vector file has a single type of record, remember) is accessed via a read-only *schema* attribute. It has 'geometry' and 'properties' items. The former is a string and the latter is an ordered dict with items having the same order as the fields in the data file.

```
>>> import pprint
>>> pprint.pprint(c.schema)
{'geometry': 'Polygon',
 'properties': {'CAT': 'float:16',
                'FIPS_CNTRY': 'str',
                'CNTRY_NAME': 'str',
                'AREA': 'float:15.2',
                'POP_CNTRY': 'float:15.2'}}
```

2.4.1 1.4.1 Keeping Schemas Simple

Fiona takes a less is more approach to record types and schemas. Data about record types is structured as closely to data about records as can be done. Modulo a record's 'id' key, the keys of a schema mapping are the same as the keys of the collection's record mappings.

```
>>> rec = next(c)
>>> set(rec.keys()) - set(c.schema.keys())
{'id'}
>>> set(rec['properties'].keys()) == set(c.schema['properties'].keys())
True
```

The values of the schema mapping are either additional mappings or field type names like 'Polygon', 'float', and 'str'. The corresponding Python types can be found in a dictionary named `fiona.FIELD_TYPES_MAP`.

```
>>> pprint.pprint(fiona.FIELD_TYPES_MAP)
{'date': <class 'fiona.rfc3339.FionaDateType'>,
 'datetime': <class 'fiona.rfc3339.FionaDateTimeType'>,
 'float': <class 'float'>,
 'int': <class 'int'>,
 'str': <class 'str'>,
 'time': <class 'fiona.rfc3339.FionaTimeType'>}
```

2.4.2 1.4.2 Field Types

In a nutshell, the types and their names are as near to what you'd expect in Python (or Javascript) as possible. The 'str' vs 'unicode' muddle is a fact of life in Python < 3.0. Fiona records have Unicode strings, but their field type name is 'str' (looking forward to Python 3).

```
>>> type(rec['properties']['CNTRY_NAME'])
<class 'str'>
>>> c.schema['properties']['CNTRY_NAME']
'str'
>>> fiona.FIELD_TYPES_MAP[c.schema['properties']['CNTRY_NAME']]
<class 'str'>
```

String type fields may also indicate their maximum width. A value of 'str:25' indicates that all values will be no longer than 25 characters. If this value is used in the schema of a file opened for writing, values of that property will be truncated at 25 characters. The default width is 80 chars, which means 'str' and 'str:80' are more or less equivalent.

Fiona provides a function to get the width of a property.

```
>>> from fiona import prop_width
>>> prop_width('str:25')
25
>>> prop_width('str')
80
```

Another function gets the proper Python type of a property.

```
>>> from fiona import prop_type
>>> prop_type('int')
<type 'int'>
>>> prop_type('float')
<type 'float'>
>>> prop_type('str:25')
<class 'str'>
```

The example above is for Python 3. With Python 2, the type of 'str' properties is 'unicode'.

```
>>> prop_type('str:25')
<class 'unicode'>
```

2.4.3 1.4.3 Geometry Types

Fiona supports the geometry types in GeoJSON and their 3D variants. This means that the value of a schema's geometry item will be one of the following:

- Point
- LineString
- Polygon
- MultiPoint
- MultiLineString
- MultiPolygon
- GeometryCollection
- 3D Point
- 3D LineString
- 3D Polygon
- 3D MultiPoint
- 3D MultiLineString
- 3D MultiPolygon
- 3D GeometryCollection

The last seven of these, the 3D types, apply only to collection schema. The geometry types of features are always one of the first seven. A '3D Point' collection, for example, always has features with geometry type 'Point'. The coordinates of those geometries will be (x, y, z) tuples.

Note that one of the most common vector data formats, Esri's Shapefile, has no 'MultiLineString' or 'MultiPolygon' schema geometries. However, a Shapefile that indicates 'Polygon' in its schema may yield either 'Polygon' or 'MultiPolygon' features.

2.5 1.5 Records

A record you get from a collection is a Python `dict` structured exactly like a GeoJSON Feature. Fiona records are self-describing; the names of its fields are contained within the data structure and the values in the fields are typed properly for the type of record. Numeric field values are instances of type `int` and `float`, for example, not strings.

```
>>> pprint.pprint(rec)
{'geometry': {'coordinates': [[(-4.663611, 51.158333),
                               (-4.669168, 51.159439),
                               (-4.673334, 51.161385),
                               (-4.674445, 51.165276),
                               (-4.67139, 51.185272),
                               (-4.669445, 51.193054),
                               (-4.665556, 51.195),
                               (-4.65889, 51.195),
                               (-4.656389, 51.192215),
                               (-4.646389, 51.164444),
                               (-4.646945, 51.160828),
                               (-4.651668, 51.159439),
                               (-4.663611, 51.158333)]],
              'type': 'Polygon'},
 'id': '1',
 'properties': {'CAT': 232.0,
                'FIPS_CNTRY': 'UK',
                'CNTRY_NAME': 'United Kingdom',
                'AREA': 244820.0,
                'POP_CNTRY': 60270708.0}}
```

The record data has no references to the *Collection* from which it originates or to any other external resource. It's entirely independent and safe to use in any way. Closing the collection does not affect the record at all.

```
>>> c.close()
>>> rec['id']
'1'
```

2.5.1 1.5.1 Record Id

A record has an `id` key. As in the GeoJSON specification, its corresponding value is a string unique within the data file.

```
>>> c = fiona.open('docs/data/test_uk.shp')
>>> rec = next(c)
>>> rec['id']
'0'
```

OGR Details

In the **OGR** model, feature ids are long integers. Fiona record ids are therefore usually string representations of integer record indexes.

2.5.2 1.5.2 Record Properties

A record has a `properties` key. Its corresponding value is a mapping: an ordered dict to be precise. The keys of the properties mapping are the same as the keys of the properties mapping in the schema of the collection the record comes from (see above).

```
>>> pprint.pprint(rec['properties'])
{'CAT': 232.0,
 'FIPS_CNTRY': 'UK',
 'CNTRY_NAME': 'United Kingdom',
 'AREA': 244820.0,
 'POP_CNTRY': 60270708.0}
```

2.5.3 1.5.3 Record Geometry

A record has a `geometry` key. Its corresponding value is a mapping with `type` and `coordinates` keys.

```
>>> pprint.pprint(rec['geometry'])
{'coordinates': [[(0.899167, 51.357216),
                  (0.885278, 51.35833),
                  (0.7875, 51.369438),
                  (0.781111, 51.370552),
                  (0.766111, 51.375832),
                  (0.759444, 51.380829),
                  (0.745278, 51.39444),
                  (0.740833, 51.400276),
                  (0.735, 51.408333),
                  (0.740556, 51.429718),
                  (0.748889, 51.443604),
                  (0.760278, 51.444717),
                  (0.791111, 51.439995),
                  (0.892222, 51.421387),
                  (0.904167, 51.418884),
                  (0.908889, 51.416939),
                  (0.930555, 51.398888),
                  (0.936667, 51.393608),
                  (0.943889, 51.384995),
                  (0.9475, 51.378609),
                  (0.947778, 51.374718),
                  (0.946944, 51.371109),
                  (0.9425, 51.369164),
                  (0.904722, 51.358055),
                  (0.899167, 51.357216)]],
 'type': 'Polygon'}
```

Since the coordinates are just tuples, or lists of tuples, or lists of lists of tuples, the `type` tells you how to interpret them.

Type	Coordinates
Point	A single (x, y) tuple
LineString	A list of (x, y) tuple vertices
Polygon	A list of rings (each a list of (x, y) tuples)
MultiPoint	A list of points (each a single (x, y) tuple)
MultiLineString	A list of lines (each a list of (x, y) tuples)
MultiPolygon	A list of polygons (see above)

Fiona, like the GeoJSON format, has both Northern Hemisphere "North is up" and Cartesian "X-Y" biases. The values within a tuple that denoted as `(x, y)` above are either (longitude E of the prime meridian, latitude N of the equator) or, for other projected coordinate systems, (easting, northing).

Long-Lat, not Lat-Long

Even though most of us say "lat, long" out loud, Fiona's `x,y` is always easting, northing, which means `(long, lat)`. Longitude first and latitude second, consistent with the GeoJSON format specification.

2.5.4 1.5.4 Point Set Theory and Simple Features

In a proper, well-scrubbed vector data file the geometry mappings explained above are representations of geometric objects made up of *point sets*. The following

```
{'type': 'LineString', 'coordinates': [(0.0, 0.0), (0.0, 1.0)]}
```

represents not just two points, but the set of infinitely many points along the line of length 1.0 from `(0.0, 0.0)` to `(0.0, 1.0)`. In the application of point set theory commonly called *Simple Features Access* [SFA] two geometric objects are equal if their point sets are equal whether they are equal in the Python sense or not. If you have Shapely (which implements Simple Features Access) installed, you can see this in by verifying the following.

```
>>> from shapely.geometry import shape
>>> l1 = shape(
...     {'type': 'LineString', 'coordinates': [(0, 0), (2, 2)]})
>>> l2 = shape(
...     {'type': 'LineString', 'coordinates': [(0, 0), (1, 1), (2, 2)]})
>>> l1 == l2
False
>>> l1.equals(l2)
True
```

Dirty data

Some files may contain vectors that are *invalid* from a simple features standpoint due to accident (inadequate quality control on the producer's end), intention ("dirty" vectors saved to a file for special treatment) or discrepancies of the numeric precision models (Fiona can't handle fixed precision models yet). Fiona doesn't sniff for or attempt to clean dirty data, so make sure you're getting yours from a clean source.

2.6 1.6 Writing Vector Data

A vector file can be opened for writing in mode `'a'` (append) or mode `'w'` (write).

Note

The in situ "update" mode of **OGR** is quite format dependent and is therefore not supported by Fiona.

2.6.1 1.6.1 Appending Data to Existing Files

Let's start with the simplest if not most common use case, adding new records to an existing file. The file is copied before modification and a suitable record extracted in the example below.

```
>>> with fiona.open('docs/data/test_uk.shp') as c:
...     rec = next(c)
>>> rec['id'] = '-1'
>>> rec['properties']['CNTRY_NAME'] = 'Gondor'
>>> import os
>>> os.system("cp docs/data/test_uk.* /tmp")
0
```

The coordinate reference system. format, and schema of the file are already defined, so it's opened with just two arguments as for reading, but in `'a'` mode. The new record is written to the end of the file using the *write()* method. Accordingly, the length of the file grows from 48 to 49.

```
>>> with fiona.open('/tmp/test_uk.shp', 'a') as c:
...     print(len(c))
...     c.write(rec)
...     print(len(c))
...
48
49
```

The record you write must match the file's schema (because a file contains one type of record, remember). You'll get a `ValueError` if it doesn't.

```
>>> with fiona.open('/tmp/test_uk.shp', 'a') as c:
...     c.write({'properties': {'foo': 'bar'}})
...
Traceback (most recent call last):
  ...
ValueError: Record data not match collection schema
```

Now, what about record ids? The id of a record written to a file is ignored and replaced by the next value appropriate for the file. If you read the file just appended to above,

```
>>> with fiona.open('/tmp/test_uk.shp', 'a') as c:
...     records = list(c)
>>> records[-1]['id']
'48'
>>> records[-1]['properties']['CNTRY_NAME']
'Gondor'
```

You'll see that the id of `'-1'` which the record had when written is replaced by `'48'`.

The *write()* method writes a single record to the collection's file. Its sibling *writerecords()* writes a sequence (or iterator) of records.

```
>>> with fiona.open('/tmp/test_uk.shp', 'a') as c:
...     c.writerecords([rec, rec, rec])
...     print(len(c))
...
52
```

Duplication

Fiona's collections do not guard against duplication. The code above will write 3 duplicate records to the file, and they will be given unique sequential ids.

Buffering

Fiona's output is buffered. The records passed to `write()` and `writerecords()` are flushed to disk when the collection is closed. You may also call `flush()` periodically to write the buffer contents to disk.

2.6.2 1.6.2 Creating files of the same structure

Writing a new file is more complex than appending to an existing file because the file CRS, format, and schema have not yet been defined and must be done so by the programmer. Still, it's not very complicated. A schema is just a mapping, as described above. A CRS is also just a mapping, and the possible formats are enumerated in the `fiona.supported_drivers` list.

Review the parameters of our demo file.

```
>>> with fiona.open('docs/data/test_uk.shp') as source:
...     source_driver = source.driver
...     source_crs = source.crs
...     source_schema = source.schema
...
>>> source_driver
'ESRI Shapefile'
>>> source_crs
{'no_defs': True, 'ellps': 'WGS84', 'datum': 'WGS84', 'proj': 'longlat'}
>>> pprint.pprint(source_schema)
{'geometry': 'Polygon',
 'properties': {'CAT': 'float:16',
                'FIPS_CNTRY': 'str',
                'CNTRY_NAME': 'str',
                'AREA': 'float:15.2',
                'POP_CNTRY': 'float:15.2'}}
```

We can create a new file using them.

```
>>> with fiona.open(
...         '/tmp/foo.shp',
...         'w',
...         driver=source_driver,
...         crs=source_crs,
...         schema=source_schema) as c:
...     print(len(c))
...     c.write(rec)
...     print(len(c))
```

(continues on next page)

(continued from previous page)

```
...
0
1
>>> c.closed
True
>>> len(c)
1
```

Because the properties of the source schema are ordered and are passed in the same order to the write-mode collection, the written file's fields have the same order as those of the source file.

```
$ ogrinfo /tmp/foo.shp foo -so
INFO: Open of `/tmp/foo.shp'
      using driver `ESRI Shapefile' successful.

Layer name: foo
Geometry: 3D Polygon
Feature Count: 1
Extent: (0.735000, 51.357216) - (0.947778, 51.444717)
Layer SRS WKT:
GEOGCS["GCS_WGS_1984",
    DATUM["WGS_1984",
        SPHEROID["WGS_84",6378137,298.257223563]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.017453292519943295]]
CAT: Real (16.0)
FIPS_CNTRY: String (80.0)
CNTRY_NAME: String (80.0)
AREA: Real (15.2)
POP_CNTRY: Real (15.2)
```

The *meta* attribute makes duplication of a file's meta properties even easier.

```
>>> source = fiona.open('docs/data/test_uk.shp')
>>> sink = fiona.open('/tmp/foo.shp', 'w', **source.meta)
```

2.6.3 1.6.3 Writing new files from scratch

To write a new file from scratch we have to define our own specific driver, crs and schema.

To ensure the order of the attribute fields is predictable, in both the schema and the actual manifestation as feature attributes, we will use ordered dictionaries.

```
>>> from collections import OrderedDict
```

Consider the following record, structured in accordance to the Python geo protocol, representing the Eiffel Tower using a point geometry with UTM coordinates in zone 31N.

```
>>> eiffel_tower =  {
...   'geometry': {
...     'type': 'Point',
...     'coordinates': (448252, 5411935)
...   },
...   'properties': OrderedDict([
...     ('name', 'Eiffel Tower'),
```

(continues on next page)

(continued from previous page)

```
...      ('height', 300.01),
...      ('view', 'scenic'),
...      ('year', 1889)
...    ])
... }
```

A corresponding scheme could be:

```
>>> landmarks_schema = {
...   'geometry': 'Point',
...   'properties': OrderedDict([
...      ('name', 'str'),
...      ('height', 'float'),
...      ('view', 'str'),
...      ('year', 'int')
...    ])
... }
```

The coordinate reference system of these landmark coordinates is ETRS89 / UTM zone 31N which is referenced in the EPSG database as EPSG:25831.

```
>>> from fiona.crs import from_epsg
>>> landmarks_crs = from_epsg(25831)
```

An appropriate driver could be:

```
>>> output_driver = "GeoJSON"
```

Having specified schema, crs and driver, we are ready to open a file for writing our record:

```
>>> with fiona.open(
...         '/tmp/foo.geojson',
...         'w',
...         driver=output_driver,
...         crs=landmarks_crs,
...         schema=landmarks_schema) as c:
...     c.write(eiffel_tower)
...

>>> import pprint
>>> with fiona.open('/tmp/foo.geojson') as source:
...   for record in source:
...     pprint.pprint(record)
{'geometry': {'coordinates': (448252.0, 5411935.0), 'type': 'Point'},
 'id': '0',
 'properties': OrderedDict([('name', 'Eiffel Tower'),
                            ('height', 300.01),
                            ('view', 'scenic'),
                            ('year', 1889)]),
 'type': 'Feature'}
```

1.6.3.1 Ordering Record Fields

Beginning with Fiona 1.0.1, the 'properties' item of *fiona.open()*'s 'schema' keyword argument may be an ordered dict or a list of (key, value) pairs, specifying an ordering that carries into written files. If an ordinary dict is given, the ordering is determined by the output of that dict's `items()` method.

For example, since

```
>>> {'bar': 'int', 'foo': 'str'}.keys()
['foo', 'bar']
```

a schema of `{'properties': {'bar': 'int', 'foo': 'str'}}` will produce a shapefile where the first field is 'foo' and the second field is 'bar'. If you want 'bar' to be the first field, you must use a list of property items

```
c = fiona.open(
    '/tmp/file.shp',
    'w',
    schema={'properties': [('bar', 'int'), ('foo', 'str')], ...},
    ... )
```

or an ordered dict.

```
from collections import OrderedDict

schema_props = OrderedDict([('bar', 'int'), ('foo', 'str')])

c = fiona.open(
    '/tmp/file.shp',
    'w',
    schema={'properties': schema_props, ...},
    ... )
```

2.6.4 1.6.4 Coordinates and Geometry Types

If you write 3D coordinates, ones having (x, y, z) tuples, to a 2D file ('Point' schema geometry, for example) the z values will be lost.

If you write 2D coordinates, ones having only (x, y) tuples, to a 3D file ('3D Point' schema geometry, for example) a default z value of 0 will be provided.

2.7 1.7 Advanced Topics

2.7.1 1.7.1 OGR configuration options

GDAL/OGR has a large number of features that are controlled by global or thread-local configuration options. Fiona allows you to configure these options using a context manager, `fiona.Env`. This class's constructor takes GDAL/OGR configuration options as keyword arguments. To see debugging information from GDAL/OGR, for example, you may do the following.

```
import logging

import fiona


logging.basicConfig(level=logging.DEBUG)

with fiona.Env(CPL_DEBUG=True):
    fiona.open('tests/data/coutwildrnp.shp')
```

The following extra messages will appear in the Python logger's output.:

```
DEBUG:fiona._env:CPLE_None in GNM: GNMRegisterAllInternal
DEBUG:fiona._env:CPLE_None in GNM: RegisterGNMFile
DEBUG:fiona._env:CPLE_None in GNM: RegisterGNMdatabase
DEBUG:fiona._env:CPLE_None in GNM: GNMRegisterAllInternal
DEBUG:fiona._env:CPLE_None in GNM: RegisterGNMFile
DEBUG:fiona._env:CPLE_None in GNM: RegisterGNMdatabase
DEBUG:fiona._env:CPLE_None in GDAL: GDALOpen(tests/data/coutwildrnp.shp,␣
↪this=0x1683930) succeeds as ESRI Shapefile.
```

If you call `fiona.open()` with no surrounding `Env` environment, one will be created for you.

When your program exits the environent's with block the configuration reverts to its previous state.

2.7.2 1.7.2 Cloud storage credentials

One of the most important uses of `fiona.Env` is to set credentials for accessing data stored in AWS S3 or another cloud storage system.

```
from fiona.session import AWSSession
import fiona

with fiona.Env(
    session=AWSSession(
        aws_access_key_id="key",
        aws_secret_access_key="secret",
    )
):
    fiona.open("zip+s3://example-bucket/example.zip")
```

The AWSSession class is currently the only credential session manager in Fiona. The source code has an example of how classes for other cloud storage providers may be implemented. AWSSession relies upon boto3 and botocore, which will be installed as extra dependencies of Fiona if you run `pip install fiona[s3]`.

If you call `fiona.open()` with no surrounding `Env` and pass a path to an S3 object, a session will be created for you using code equivalent to the following code.

```
import boto3

from fiona.session import AWSSession
import fiona

with fiona.Env(session=AWSSession(boto3.Session())):
    fiona.open('zip+s3://fiona-testing/coutwildrnp.zip')
```

2.7.3 1.7.3 Slicing and masking iterators

With some vector data formats a spatial index accompanies the data file, allowing efficient bounding box searches. A collection's *`items()`* method returns an iterator over pairs of FIDs and records that intersect a given `(minx, miny, maxx, maxy)` bounding box or geometry object. The collection's own coordinate reference system (see below) is used to interpret the box's values. If you want a list of the iterator's items, pass it to Python's builtin `list()` as shown below.

```
>>> c = fiona.open('docs/data/test_uk.shp')
>>> hits = list(c.items(bbox=(-5.0, 55.0, 0.0, 60.0)))
>>> len(hits)
7
```

The iterator method takes the same `stop` or `start, stop[, step]` slicing arguments as `itertools.islice()`. To get just the first two items from that iterator, pass a stop index.

```
>>> hits = c.items(2, bbox=(-5.0, 55.0, 0.0, 60.0))
>>> len(list(hits))
2
```

To get the third through fifth items from that iterator, pass start and stop indexes.

```
>>> hits = c.items(2, 5, bbox=(-5.0, 55.0, 0.0, 60.0))
>>> len(list(hits))
3
```

To filter features by property values, use Python's builtin `filter()` and `lambda` or your own filter function that takes a single feature record and returns `True` or `False`.

```
>>> def pass_positive_area(rec):
...     return rec['properties'].get('AREA', 0.0) > 0.0
...
>>> c = fiona.open('docs/data/test_uk.shp')
>>> hits = filter(pass_positive_area, c)
>>> len(list(hits))
48
```

2.7.4 1.7.4 Reading Multilayer data

Up to this point, only simple datasets with one thematic layer or feature type per file have been shown and the venerable Esri Shapefile has been the primary example. Other GIS data formats can encode multiple layers or feature types within a single file or directory. Esri's File Geodatabase is one example of such a format. A more useful example, for the purpose of this manual, is a directory comprising multiple shapefiles. The following three shell commands will create just such a two layered data source from the test data distributed with Fiona.

```
$ mkdir /tmp/data
$ ogr2ogr /tmp/data/ docs/data/test_uk.shp test_uk -nln foo
$ ogr2ogr /tmp/data/ docs/data/test_uk.shp test_uk -nln bar
```

The layers of a data source can be listed using *`fiona.listlayers()`*. In the shapefile format case, layer names match base names of the files.

```
>>> fiona.listlayers('/tmp/data')
['bar', 'foo']
```

Unlike OGR, Fiona has no classes representing layers or data sources. To access the features of a layer, open a collection using the path to the data source and specify the layer by name using the *layer* keyword.

```
>>> import pprint
>>> datasrc_path = '/tmp/data'
>>> for name in fiona.listlayers(datasrc_path):
...     with fiona.open(datasrc_path, layer=name) as c:
...         pprint.pprint(c.schema)
```

(continues on next page)

(continued from previous page)

```
...
{'geometry': 'Polygon',
 'properties': {'CAT': 'float:16',
                'FIPS_CNTRY': 'str',
                'CNTRY_NAME': 'str',
                'AREA': 'float:15.2',
                'POP_CNTRY': 'float:15.2'}}
{'geometry': 'Polygon',
 'properties': {'CAT': 'float:16',
                'FIPS_CNTRY': 'str',
                'CNTRY_NAME': 'str',
                'AREA': 'float:15.2',
                'POP_CNTRY': 'float:15.2'}}
```

Layers may also be specified by their index.

```
>>> for i, name in enumerate(fiona.listlayers(datasrc_path)):
...     with fiona.open(datasrc_path, layer=i) as c:
...         print(len(c))
...
48
48
```

If no layer is specified, *fiona.open()* returns an open collection using the first layer.

```
>>> with fiona.open(datasrc_path) as c:
...     c.name == fiona.listlayers(datasrc_path)[0]
...
True
```

The most general way to open a shapefile for reading, using all of the parameters of *fiona.open()*, is to treat it as a data source with a named layer.

```
>>> fiona.open('docs/data/test_uk.shp', 'r', layer='test_uk')
```

In practice, it is fine to rely on the implicit first layer and default `'r'` mode and open a shapefile like this:

```
>>> fiona.open('docs/data/test_uk.shp')
```

2.7.5 1.7.5 Writing Multilayer data

To write an entirely new layer to a multilayer data source, simply provide a unique name to the *layer* keyword argument.

```
>>> 'wah' not in fiona.listlayers(datasrc_path)
True
>>> with fiona.open(datasrc_path, layer='bar') as c:
...     with fiona.open(datasrc_path, 'w', layer='wah', **c.meta) as d:
...         d.write(next(c))
...
>>> fiona.listlayers(datasrc_path)
['bar', 'foo', 'wah']
```

In `'w'` mode, existing layers will be overwritten if specified, just as normal files are overwritten by Python's `open()` function.

```
>>> 'wah' in fiona.listlayers(datasrc_path)
True
>>> with fiona.open(datasrc_path, layer='bar') as c:
...     with fiona.open(datasrc_path, 'w', layer='wah', **c.meta) as d:
...         # Overwrites the existing layer named 'wah'!
```

2.7.6 1.7.6 Virtual filesystems

Zip and Tar archives can be treated as virtual filesystems and collections can be made from paths and layers within them. In other words, Fiona lets you read zipped shapefiles. For example, make a Zip archive from the shapefile distributed with Fiona.

```
$ zip /tmp/zed.zip docs/data/test_uk.*
adding: docs/data/test_uk.shp (deflated 48%)
adding: docs/data/test_uk.shx (deflated 37%)
adding: docs/data/test_uk.dbf (deflated 98%)
adding: docs/data/test_uk.prj (deflated 15%)
```

The *vfs* keyword parameter for *fiona.listlayers()* and *fiona.open()* may be an Apache Commons VFS style string beginning with "zip://" or "tar://" and followed by an absolute or relative path to the archive file. When this parameter is used, the first argument to must be an absolute path within that archive. The layers in that Zip archive are:

```
>>> import fiona
>>> fiona.listlayers('/docs/data', vfs='zip:///tmp/zed.zip')
['test_uk']
```

The single shapefile may also be accessed like so:

```
>>> with fiona.open(
...         '/docs/data/test_uk.shp',
...         vfs='zip:///tmp/zed.zip') as c:
...     print(len(c))
...
48
```

2.7.7 1.7.7 MemoryFile and ZipMemoryFile

fiona.io.MemoryFile and *fiona.io.ZipMemoryFile* allow formatted feature collections, even zipped feature collections, to be read or written in memory, with no filesystem access required. For example, you may have a zipped shapefile in a stream of bytes coming from a web upload or download.

```
>>> data = open('tests/data/coutwildrnp.zip', 'rb').read()
>>> len(data)
154006
>>> data[:20]
b'PK\x03\x04\x14\x00\x00\x00\x00\x00\xaa~VM\xech\xae\x1e\xec\xab'
```

The feature collection in this stream of bytes can be accessed by wrapping it in an instance of ZipMemoryFile.

```
>>> from fiona.io import ZipMemoryFile
>>> with ZipMemoryFile(data) as zip:
...     with zip.open('coutwildrnp.shp') as collection:
```

(continues on next page)

(continued from previous page)

```
...         print(len(collection))
...         print(collection.schema)
...
67
{'properties': OrderedDict([('PERIMETER', 'float:24.15'), ('FEATURE2', 'str:80'), (
↪'NAME', 'str:80'), ('FEATURE1', 'str:80'), ('URL', 'str:101'), ('AGBUR', 'str:80'),␣
↪('AREA', 'float:24.15'), ('STATE_FIPS', 'str:80'), ('WILDRNP020', 'int:10'), ('STATE
↪', 'str:80')]), 'geometry': 'Polygon'}
```

New in 1.8.0

2.8 1.8 Fiona command line interface

Fiona comes with a command line interface called "fio". See the CLI Documentation for detailed usage instructions.

2.9 1.9 Final Notes

This manual is a work in progress and will grow and improve with Fiona. Questions and suggestions are very welcome. Please feel free to use the issue tracker or email the author directly.

Do see the README for installation instructions and information about supported versions of Python and other software dependencies.

Fiona would not be possible without the contributions of other developers, especially Frank Warmerdam and Even Rouault, the developers of GDAL/OGR; and Mike Weisman, who saved Fiona from neglect and obscurity.

2.10 1.10 References

CHAPTER

THREE

FIONA

3.1 fiona package

3.1.1 Subpackages

fiona.fio package

Submodules

fiona.fio.bounds module

$ fio bounds

fiona.fio.calc module

fiona.fio.cat module

$ fio cat

fiona.fio.collect module

$ fio collect

fiona.fio.distrib module

$ fio distrib

fiona.fio.dump module

$ fio dump

fiona.fio.env module

$ fio env

fiona.fio.filter module

$ fio filter

fiona.fio.helpers module

Helper objects needed by multiple CLI commands.

fiona.fio.helpers.**eval_feature_expression**(*feature*, *expression*)

fiona.fio.helpers.**id_record**(*rec*)
Converts a record's id to a blank node id and returns the record.

fiona.fio.helpers.**make_ld_context**(*context_items*)
Returns a JSON-LD Context object.

See http://json-ld.org/spec/latest/json-ld.

fiona.fio.helpers.**nullable**(*val*, *cast*)

fiona.fio.helpers.**obj_gen**(*lines*)
Return a generator of JSON objects loaded from lines.

fiona.fio.info module

$ fio info

fiona.fio.insp module

$ fio insp

fiona.fio.load module

$ fio load

fiona.fio.ls module

$ fiona ls

fiona.fio.main module

Main click group for the CLI. Needs to be isolated for entry-point loading.

fiona.fio.main.**configure_logging**(*verbosity*)

fiona.fio.options module

Common commandline options for *fio*

fiona.fio.options.**cb_layer**(*ctx*, *param*, *value*)
Let –layer be a name or index.

fiona.fio.options.**cb_multilayer**(*ctx*, *param*, *value*)
Transform layer options from strings ("1:a,1:b", "2:a,2:c,2:z") to { '1': ['a', 'b'], '2': ['a', 'c', 'z'] }

fiona.fio.options.**validate_multilayer_file_index**(*files*, *layerdict*)
Ensure file indexes provided in the –layer option are valid

fiona.fio.rm module

Module contents

Fiona's command line interface

fiona.fio.**with_context_env**(*f*)
Pops the Fiona Env from the passed context and executes the wrapped func in the context of that obj.

Click's pass_context decorator must precede this decorator, or else there will be no context in the wrapper args.

3.1.2 Submodules

3.1.3 fiona.collection module

class fiona.collection.**BytesCollection**(*bytesbuf*, ***kwds*)
Bases: *fiona.collection.Collection*

BytesCollection takes a buffer of bytes and maps that to a virtual file that can then be opened by fiona.

close()
Removes the virtual file associated with the class.

class fiona.collection.**Collection**(*path*, *mode='r'*, *driver=None*, *schema=None*, *crs=None*, *encoding=None*, *layer=None*, *vsi=None*, *archive=None*, *enabled_drivers=None*, *crs_wkt=None*, *ignore_fields=None*, *ignore_geometry=False*, ***kwargs*)
Bases: object

A file-like interface to features of a vector dataset

Python text file objects are iterators over lines of a file. Fiona Collections are similar iterators (not lists!) over features represented as GeoJSON-like mappings.

property bounds
Returns (minx, miny, maxx, maxy).

close()
In append or write mode, flushes data to disk, then ends access.

property closed
`False` if data can be accessed, otherwise `True`.

property crs
Returns a Proj4 string.

property crs_wkt
Returns a WKT string.

property driver
Returns the name of the proper OGR driver.

filter(**args*, ***kwds*)
Returns an iterator over records, but filtered by a test for spatial intersection with the provided `bbox`, a (minx, miny, maxx, maxy) tuple or a geometry `mask`.

Positional arguments `stop` or `start, stop[, step]` allows iteration to skip over items or stop at a specific item.

flush()
Flush the buffer.

get(*item*)

guard_driver_mode()

items(**args*, ***kwds*)
Returns an iterator over FID, record pairs, optionally filtered by a test for spatial intersection with the provided `bbox`, a (minx, miny, maxx, maxy) tuple or a geometry `mask`.

Positional arguments `stop` or `start, stop[, step]` allows iteration to skip over items or stop at a specific item.

keys(**args*, ***kwds*)
Returns an iterator over FIDs, optionally filtered by a test for spatial intersection with the provided `bbox`, a (minx, miny, maxx, maxy) tuple or a geometry `mask`.

Positional arguments `stop` or `start, stop[, step]` allows iteration to skip over items or stop at a specific item.

property meta
Returns a mapping with the driver, schema, crs, and additional properties.

next()
Returns next record from iterator.

property profile
Returns a mapping with the driver, schema, crs, and additional properties.

property schema
Returns a mapping describing the data schema.

The mapping has 'geometry' and 'properties' items. The former is a string such as 'Point' and the latter is an ordered mapping that follows the order of fields in the data file.

validate_record(*record*)
Compares the record to the collection's schema.

Returns `True` if the record matches, else `False`.

validate_record_geometry(*record*)
Compares the record's geometry to the collection's schema.

Returns `True` if the record matches, else `False`.

values(**args*, ***kwds*)
Returns an iterator over records, but filtered by a test for spatial intersection with the provided `bbox`, a (minx, miny, maxx, maxy) tuple or a geometry `mask`.

Positional arguments `stop` or `start, stop[, step]` allows iteration to skip over items or stop at a specific item.

write(*record*)
Stages a record for writing to disk.

writerecords(*records*)
Stages multiple records for writing to disk.

`fiona.collection.`**get_filetype**(*bytesbuf*)
Detect compression type of bytesbuf.

ZIP only. TODO: add others relevant to GDAL/OGR.

3.1.4 fiona.compat module

`fiona.compat.`**strencode**(*instr*, *encoding='utf-8'*)

3.1.5 fiona.crs module

Coordinate reference systems and functions

PROJ.4 is the law of this land: http://proj.osgeo.org/. But whereas PROJ.4 coordinate reference systems are described by strings of parameters such as

+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs

here we use mappings:

{'proj': 'longlat', 'ellps': 'WGS84', 'datum': 'WGS84', 'no_defs': True}

`fiona.crs.`**from_epsg**(*code*)
Given an integer code, returns an EPSG-like mapping.

Note: the input code is not validated against an EPSG database.

`fiona.crs.`**from_string**(*prjs*)
Turn a PROJ.4 string into a mapping of parameters.

Bare parameters like "+no_defs" are given a value of `True`. All keys are checked against the `all_proj_keys` list.

`fiona.crs.`**to_string**(*crs*)
Turn a parameter mapping into a more conventional PROJ.4 string.

Mapping keys are tested against the `all_proj_keys` list. Values of `True` are omitted, leaving the key bare: {'no_defs': True} -> "+no_defs" and items where the value is otherwise not a str, int, or float are omitted.

3.1.6 fiona.drvsupport module

3.1.7 fiona.env module

Fiona's GDAL/AWS environment

class `fiona.env.`**Env**(*session=None*, ***options*)
Bases: `object`

Abstraction for GDAL and AWS configuration

The GDAL library is stateful: it has a registry of format drivers, an error stack, and dozens of configuration options.

Fiona's approach to working with GDAL is to wrap all the state up using a Python context manager (see PEP 343, https://www.python.org/dev/peps/pep-0343/). When the context is entered GDAL drivers are registered, error handlers are configured, and configuration options are set. When the context is exited, drivers are removed from the registry and other configurations are removed.

Example:

with fiona.Env(GDAL_CACHEMAX=512) as env: # All drivers are registered, GDAL's raster block cache # size is set to 512MB. # Commence processing... ... # End of processing.

At this point, configuration options are set to their # previous (possible unset) values.

A boto3 session or boto3 session constructor arguments *aws_access_key_id*, *aws_secret_access_key*, *aws_session_token* may be passed to Env's constructor. In the latter case, a session will be created as soon as needed. AWS credentials are configured for GDAL as needed.

credentialize()
Get credentials and configure GDAL

Note well: this method is a no-op if the GDAL environment already has credentials, unless session is not None.

None

classmethod default_options()
Default configuration options

None

dict

drivers()
Return a mapping of registered drivers.

classmethod from_defaults(*session=None*, ***options*)
Create an environment with default config options

session [optional] A Session object.

****options** [optional] A mapping of GDAL configuration options, e.g., *CPL_DEBUG=True, CHECK_WITH_INVERT_PROJ=False*.

Env

The items in kwargs will be overlaid on the default values.

property is_credentialized
Test for existence of cloud credentials

bool

class fiona.env.**GDALVersion**(*major=0*, *minor=0*)
Bases: object

Convenience class for obtaining GDAL major and minor version components and comparing between versions. This is highly simplistic and assumes a very normal numbering scheme for versions and ignores everything except the major and minor components.

at_least(*other*)

major

minor

classmethod parse(*input*)
Parses input tuple or string to GDALVersion. If input is a GDALVersion instance, it is returned.

input: tuple of (major, minor), string, or instance of GDALVersion

GDALVersion instance

classmethod runtime()
Return GDALVersion of current GDAL runtime

class fiona.env.**NullContextManager**
Bases: object

class fiona.env.**ThreadEnv**
Bases: _thread._local

fiona.env.**defenv**(***options*)
Create a default environment if necessary.

fiona.env.**delenv**()
Delete options in the existing environment.

fiona.env.**ensure_env**(*f*)
A decorator that ensures an env exists before a function calls any GDAL C functions.

f [function] A function.

A function wrapper.

If there is already an existing environment, the wrapper does nothing and immediately calls f with the given arguments.

fiona.env.**ensure_env_with_credentials**(*f*)
Ensures a config environment exists and has credentials.

f [function] A function.

A function wrapper.

The function wrapper checks the first argument of f and credentializes the environment if the first argument is a URI with scheme "s3".

If there is already an existing environment, the wrapper does nothing and immediately calls f with the given arguments.

fiona.env.**env_ctx_if_needed**()
Return an Env if one does not exist

Env or a do-nothing context manager

fiona.env.**getenv**()
Get a mapping of current options.

fiona.env.**hascreds**()

fiona.env.**hasenv**()

fiona.env.**require_gdal_version**(*version*, *param=None*, *values=None*, *is_max_version=False*, *reason=''*)
A decorator that ensures the called function or parameters are supported by the runtime version of GDAL. Raises GDALVersionError if conditions are not met.

Examples:

@require_gdal_version('2.2') def some_func():

calling *some_func* with a runtime version of GDAL that is < 2.2 raises a GDALVersionErorr.

@require_gdal_version('2.2', param='foo') def some_func(foo='bar'):

calling *some_func* with parameter *foo* of any value on GDAL < 2.2 raises a GDALVersionError.

@require_gdal_version('2.2', param='foo', values=('bar',)) def some_func(foo=None):

calling *some_func* with parameter *foo* and value *bar* on GDAL < 2.2 raises a GDALVersionError.

version: tuple, string, or GDALVersion param: string (optional, default: None)

If *values* are absent, then all use of this parameter with a value other than default value requires at least GDAL *version*.

values: tuple, list, or set (optional, default: None) contains values that require at least GDAL *version*. *param* is required for *values*.

is_max_version: bool (optional, default: False) if *True* indicates that the version provided is the maximum version allowed, instead of requiring at least that version.

reason: string (optional: default: '') custom error message presented to user in addition to message about GDAL version. Use this to provide an explanation of what changed if necessary context to the user.

wrapped function

`fiona.env.`**`setenv`**`(**options)`
Set options in the existing environment.

3.1.8 fiona.errors module

exception `fiona.errors.`**`CRSError`**
Bases: *fiona.errors.FionaValueError*

When a crs mapping has neither init or proj items.

exception `fiona.errors.`**`DataIOError`**
Bases: `OSError`

IO errors involving driver registration or availability.

exception `fiona.errors.`**`DatasetDeleteError`**
Bases: `OSError`

Failure to delete a dataset

exception `fiona.errors.`**`DriverError`**
Bases: *fiona.errors.FionaValueError*

Encapsulates unsupported driver and driver mode errors.

exception `fiona.errors.`**`DriverIOError`**
Bases: `OSError`

A format specific driver error.

exception `fiona.errors.`**`DriverSupportError`**
Bases: *fiona.errors.DriverIOError*

Driver does not support schema

exception `fiona.errors.`**`EnvError`**
Bases: *fiona.errors.FionaError*

Environment Errors

exception fiona.errors.**FieldNameEncodeError**
Bases: UnicodeEncodeError

Failure to encode a field name.

exception fiona.errors.**FionaDeprecationWarning**
Bases: UserWarning

A warning about deprecation of Fiona features

exception fiona.errors.**FionaError**
Bases: Exception

Base Fiona error

exception fiona.errors.**FionaValueError**
Bases: ValueError

Fiona-specific value errors

exception fiona.errors.**GDALVersionError**
Bases: *fiona.errors.FionaError*

Raised if the runtime version of GDAL does not meet the required version of GDAL.

exception fiona.errors.**GeometryTypeValidationError**
Bases: *fiona.errors.FionaValueError*

Tried to write a geometry type not specified in the schema

exception fiona.errors.**SchemaError**
Bases: *fiona.errors.FionaValueError*

When a schema mapping has no properties or no geometry.

exception fiona.errors.**TransactionError**
Bases: RuntimeError

Failure relating to GDAL transactions

exception fiona.errors.**UnsupportedGeometryTypeError**
Bases: KeyError

When a OGR geometry type isn't supported by Fiona.

3.1.9 fiona.inspector module

fiona.inspector.**main**(*srcfile*)

3.1.10 fiona.io module

Classes capable of reading and writing collections

class fiona.io.**MemoryFile**(*file_or_bytes=None*, *filename=None*, *ext=''*)
Bases: *fiona.ogrext.MemoryFileBase*

A BytesIO-like object, backed by an in-memory file.

This allows formatted files to be read and written without I/O.

A MemoryFile created with initial bytes becomes immutable. A MemoryFile created without initial bytes may be written to using either file-like or dataset interfaces.

open(*driver=None*, *schema=None*, *crs=None*, *encoding=None*, *layer=None*, *vfs=None*, *enabled_drivers=None*, *crs_wkt=None*, ***kwargs*)
Open the file and return a Fiona collection object.

If data has already been written, the file is opened in 'r' mode. Otherwise, the file is opened in 'w' mode.

Note well that there is no *path* parameter: a *MemoryFile* contains a single dataset and there is no need to specify a path.

Other parameters are optional and have the same semantics as the parameters of *fiona.open()*.

class fiona.io.**ZipMemoryFile**(*file_or_bytes=None*)
Bases: *fiona.io.MemoryFile*

A read-only BytesIO-like object backed by an in-memory zip file.

This allows a zip file containing formatted files to be read without I/O.

open(*path*, *driver=None*, *encoding=None*, *layer=None*, *enabled_drivers=None*, ***kwargs*)
Open a dataset within the zipped stream.

path [str] Path to a dataset in the zip file, relative to the root of the archive.

A Fiona collection object

3.1.11 fiona.logutils module

Logging helper classes.

class fiona.logutils.**FieldSkipLogFilter**(*name=''*)
Bases: logging.Filter

Filter field skip log messges.

At most, one message per field skipped per loop will be passed.

filter(*record*)
Pass record if not seen.

class fiona.logutils.**LogFiltering**(*logger*, *filter*)
Bases: object

3.1.12 fiona.ogrext module

class fiona.ogrext.**FeatureBuilder**
Bases: object

Build Fiona features from OGR feature pointers.

No OGR objects are allocated by this function and the feature argument is not destroyed.

class fiona.ogrext.**ItemsIterator**
Bases: *fiona.ogrext.Iterator*

class fiona.ogrext.**Iterator**
Bases: object

Provides iterated access to feature data.

class fiona.ogrext.**KeysIterator**
Bases: *fiona.ogrext.Iterator*

class fiona.ogrext.**MemoryFileBase**
Bases: object

Base for a BytesIO-like class backed by an in-memory file.

close()
Close MemoryFile and release allocated memory.

exists()
Test if the in-memory file exists.

bool True if the in-memory file exists.

read()
Read size bytes from MemoryFile.

seek()
Seek to position in MemoryFile.

tell()
Tell current position in MemoryFile.

write()
Write data bytes to MemoryFile

class fiona.ogrext.**OGRFeatureBuilder**
Bases: object

Builds an OGR Feature from a Fiona feature mapping.

Allocates one OGR Feature which should be destroyed by the caller. Borrows a layer definition from the collection.

class fiona.ogrext.**Session**
Bases: object

get()
Provides access to feature data by FID.

Supports Collection.__contains__().

get_crs()
Get the layer's CRS

CRS

get_crs_wkt()

get_driver()

get_extent()

get_feature()
Provides access to feature data by FID.

Supports Collection.__contains__().

get_fileencoding()
DEPRECATED

get_length()

get_schema()

has_feature()
Provides access to feature data by FID.

Supports Collection.__contains__().

isactive()

start()

stop()

class fiona.ogrext.**WritingSession**
Bases: *fiona.ogrext.Session*

start()

sync()
Syncs OGR to disk.

writerecs()
Writes buffered records to OGR.

fiona.ogrext.**buffer_to_virtual_file**()
Maps a bytes buffer to a virtual file.

ext is empty or begins with a period and contains at most one period.

fiona.ogrext.**featureRT**()

fiona.ogrext.**remove_virtual_file**()

3.1.13 fiona.path module

Dataset paths, identifiers, and filenames

class fiona.path.**ParsedPath**(*path*, *archive*, *scheme*)
Bases: *fiona.path.Path*

Result of parsing a dataset URI/Path

path [str] Parsed path. Includes the hostname and query string in the case of a URI.

archive [str] Parsed archive path.

scheme [str] URI scheme such as "https" or "zip+s3".

archive

classmethod from_uri(*uri*)

property is_local
Test if the path is a local URI

property is_remote
Test if the path is a remote, network URI

property name
The parsed path's original URI

path

scheme

class fiona.path.**Path**
Bases: object

Base class for dataset paths

class fiona.path.**UnparsedPath**(*path*)
Bases: *fiona.path.Path*

Encapsulates legacy GDAL filenames

path [str] The legacy GDAL filename.

property name
The unparsed path's original path

path

fiona.path.**parse_path**(*path*)
Parse a dataset's identifier or path into its parts

path [str or path-like object] The path to be parsed.

ParsedPath or UnparsedPath

When legacy GDAL filenames are encountered, they will be returned in a UnparsedPath.

fiona.path.**vsi_path**(*path*)
Convert a parsed path to a GDAL VSI path

path [Path] A ParsedPath or UnparsedPath object.

str

3.1.14 fiona.rfc3339 module

class fiona.rfc3339.**FionaDateTimeType**
Bases: str

Dates and times.

class fiona.rfc3339.**FionaDateType**
Bases: str

Dates without time.

class fiona.rfc3339.**FionaTimeType**
Bases: str

Times without dates.

class fiona.rfc3339.**group_accessor**(*m*)
Bases: object

group(*i*)

fiona.rfc3339.**parse_date**(*text*)
Given a RFC 3339 date, returns a tz-naive datetime tuple

fiona.rfc3339.**parse_datetime**(*text*)
Given a RFC 3339 datetime, returns a tz-naive datetime tuple

fiona.rfc3339.**parse_time**(*text*)
Given a RFC 3339 time, returns a tz-naive datetime tuple

3.1.15 fiona.schema module

fiona.schema.**normalize_field_type**()
Normalize free form field types to an element of FIELD_TYPES

ftype [str] A type:width format like 'int:9' or 'str:255'

str An element from FIELD_TYPES

3.1.16 fiona.session module

Abstraction for sessions in various clouds.

class fiona.session.**AWSSession**(*session=None*, *aws_unsigned=False*, *aws_access_key_id=None*, *aws_secret_access_key=None*, *aws_session_token=None*, *region_name=None*, *profile_name=None*, *requester_pays=False*)

Bases: *fiona.session.Session*

Configures access to secured resources stored in AWS S3.

property credentials
The session credentials as a dict

get_credential_options()
Get credentials as GDAL configuration options

dict

class fiona.session.**DummySession**(**args*, ***kwargs*)

Bases: *fiona.session.Session*

A dummy session.

credentials [dict] The session credentials.

get_credential_options()
Get credentials as GDAL configuration options

dict

class fiona.session.**GSSession**(*google_application_credentials=None*)

Bases: *fiona.session.Session*

Configures access to secured resources stored in Google Cloud Storage

property credentials
The session credentials as a dict

get_credential_options()
Get credentials as GDAL configuration options

dict

classmethod hascreds(*config*)
Determine if the given configuration has proper credentials

cls [class] A Session class.

config [dict] GDAL configuration as a dict.

bool

class fiona.session.**Session**

Bases: object

Base for classes that configure access to secured resources.

credentials [dict] Keys and values for session credentials.

This class is not intended to be instantiated.

static from_foreign_session(*session*, *cls=None*)
Create a session object matching the foreign *session*.

session [obj] A foreign session object.

cls [Session class, optional] The class to return.

Session

static from_path(*path*, **args*, ***kwargs*)
Create a session object suited to the data at *path*.

path [str] A dataset path or identifier.

args [sequence] Positional arguments for the foreign session constructor.

kwargs [dict] Keyword arguments for the foreign session constructor.

Session

get_credential_options()
Get credentials as GDAL configuration options

dict

3.1.17 fiona.transform module

Coordinate and geometry warping and reprojection

fiona.transform.**transform**(*src_crs*, *dst_crs*, *xs*, *ys*)
Transform coordinates from one reference system to another.

src_crs: str or dict A string like 'EPSG:4326' or a dict of proj4 parameters like {'proj': 'lcc', 'lat_0': 18.0, 'lat_1': 18.0, 'lon_0': -77.0} representing the coordinate reference system on the "source" or "from" side of the transformation.

dst_crs: str or dict A string or dict representing the coordinate reference system on the "destination" or "to" side of the transformation.

xs: sequence of float A list or tuple of x coordinate values. Must have the same length as the `ys` parameter.

ys: sequence of float A list or tuple of y coordinate values. Must have the same length as the `xs` parameter.

xp, yp: list of float A pair of transformed coordinate sequences. The elements of `xp` and `yp` correspond exactly to the elements of the `xs` and `ys` input parameters.

```
>>> transform('EPSG:4326', 'EPSG:26953', [-105.0], [40.0])
([957097.0952383667], [378940.8419189212])
```

fiona.transform.**transform_geom**(*src_crs*, *dst_crs*, *geom*, *antimeridian_cutting=False*, *antimeridian_offset=10.0*, *precision=-1*)
Transform a geometry obj from one reference system to another.

src_crs: str or dict A string like 'EPSG:4326' or a dict of proj4 parameters like {'proj': 'lcc', 'lat_0': 18.0, 'lat_1': 18.0, 'lon_0': -77.0} representing the coordinate reference system on the "source" or "from" side of the transformation.

dst_crs: str or dict A string or dict representing the coordinate reference system on the "destination" or "to" side of the transformation.

geom: obj A GeoJSON-like geometry object with 'type' and 'coordinates' members.

antimeridian_cutting: bool, optional `True` to cut output geometries in two at the antimeridian, the default is ‘‘False‘.

antimeridian_offset: float, optional A distance in decimal degrees from the antimeridian, outside of which geometries will not be cut.

precision: int, optional Optional rounding precision of output coordinates, in number of decimal places.

obj A new GeoJSON-like geometry with transformed coordinates. Note that if the output is at the antimeridian, it may be cut and of a different geometry `type` than the input, e.g., a polygon input may result in multi-polygon output.

```
>>> transform_geom(
...     'EPSG:4326', 'EPSG:26953',
...     {'type': 'Point', 'coordinates': [-105.0, 40.0]})
{'type': 'Point', 'coordinates': (957097.0952383667, 378940.8419189212)}
```

3.1.18 fiona.vfs module

Implementation of Apache VFS schemes and URLs.

`fiona.vfs.`**`is_remote`**(*scheme*)

`fiona.vfs.`**`parse_paths`**(*uri*, *vfs=None*)
Parse a URI or Apache VFS URL into its parts

Returns: tuple (path, scheme, archive)

`fiona.vfs.`**`valid_vsi`**(*vsi*)
Ensures all parts of our vsi path are valid schemes.

`fiona.vfs.`**`vsi_path`**(*path*, *vsi=None*, *archive=None*)

3.1.19 Module contents

Fiona is OGR’s neat, nimble, no-nonsense API.

Fiona provides a minimal, uncomplicated Python interface to the open source GIS community’s most trusted geodata access library and integrates readily with other Python GIS packages such as pyproj, Rtree and Shapely.

How minimal? Fiona can read features as mappings from shapefiles or other GIS vector formats and write mappings as features to files using the same formats. That’s all. There aren’t any feature or geometry classes. Features and their geometries are just data.

A Fiona feature is a Python mapping inspired by the GeoJSON format. It has *id*, ‘geometry‘, and *properties* keys. The value of *id* is a string identifier unique within the feature’s parent collection. The *geometry* is another mapping with *type* and *coordinates* keys. The *properties* of a feature is another mapping corresponding to its attribute table. For example:

{‘id’: ‘1’, ‘geometry’: {‘type’: ‘Point’, ‘coordinates’: (0.0, 0.0)}, ‘properties’: {‘label’: u’Null Island’} }

is a Fiona feature with a point geometry and one property.

Features are read and written using objects returned by the `collection` function. These `Collection` objects are a lot like Python `file` objects. A `Collection` opened in reading mode serves as an iterator over features. One opened in a writing mode provides a `write` method.

Usage

Here's an example of reading a select few polygon features from a shapefile and for each, picking off the first vertex of the exterior ring of the polygon and using that as the point geometry for a new feature writing to a "points.shp" file.

```
>>> import fiona
>>> with fiona.open('docs/data/test_uk.shp', 'r') as inp:
...     output_schema = inp.schema.copy()
...     output_schema['geometry'] = 'Point'
...     with collection(
...             "points.shp", "w",
...             crs=inp.crs,
...             driver="ESRI Shapefile",
...             schema=output_schema
...             ) as out:
...         for f in inp.filter(
...                 bbox=(-5.0, 55.0, 0.0, 60.0)
...                 ):
...             value = f['geometry']['coordinates'][0][0]
...             f['geometry'] = {
...                 'type': 'Point', 'coordinates': value}
...             out.write(f)
```

Because Fiona collections are context managers, they are closed and (in writing modes) flush contents to disk when their `with` blocks end.

`fiona.`**`bounds`**(*ob*)
Returns a (minx, miny, maxx, maxy) bounding box.

The `ob` may be a feature record or geometry.

`fiona.`**`listlayers`**(*path*, *vfs=None*)
Returns a list of layer names in their index order.

The required `path` argument may be an absolute or relative file or directory path.

A virtual filesystem can be specified. The `vfs` parameter may be an Apache Commons VFS style string beginning with "zip://" or "tar://"". In this case, the `path` must be an absolute path within that container.

`fiona.`**`open`**(*fp*, *mode='r'*, *driver=None*, *schema=None*, *crs=None*, *encoding=None*, *layer=None*, *vfs=None*, *enabled_drivers=None*, *crs_wkt=None*, ***kwargs*)
Open a collection for read, append, or write

In write mode, a driver name such as "ESRI Shapefile" or "GPX" (see OGR docs or `ogr2ogr --help` on the command line) and a schema mapping such as:

{'geometry': 'Point',

'properties': [('class', 'int'), ('label', 'str'), ('value', 'float')]}

must be provided. If a particular ordering of properties ("fields" in GIS parlance) in the written file is desired, a list of (key, value) pairs as above or an ordered dict is required. If no ordering is needed, a standard dict will suffice.

A coordinate reference system for collections in write mode can be defined by the `crs` parameter. It takes Proj4 style mappings like

{'proj': 'longlat', 'ellps': 'WGS84', 'datum': 'WGS84', 'no_defs': True}

short hand strings like

EPSG:4326

or WKT representations of coordinate reference systems.

The drivers used by Fiona will try to detect the encoding of data files. If they fail, you may provide the proper `encoding`, such as 'Windows-1252' for the Natural Earth datasets.

When the provided path is to a file containing multiple named layers of data, a layer can be singled out by `layer`.

The drivers enabled for opening datasets may be restricted to those listed in the `enabled_drivers` parameter. This and the `driver` parameter afford much control over opening of files.

Trying only the GeoJSON driver when opening to read, the # following raises `DataIOError`: fiona.open('example.shp', driver='GeoJSON')

Trying first the GeoJSON driver, then the Shapefile driver, # the following succeeds: fiona.open(

'example.shp', enabled_drivers=['GeoJSON', 'ESRI Shapefile'])

fp [URI (str or pathlib.Path), or file-like object] A dataset resource identifier or file object.

mode [str] One of 'r', to read (the default); 'a', to append; or 'w', to write.

driver [str] In 'w' mode a format driver name is required. In 'r' or 'a' mode this parameter has no effect.

schema [dict] Required in 'w' mode, has no effect in 'r' or 'a' mode.

crs [str or dict] Required in 'w' mode, has no effect in 'r' or 'a' mode.

encoding [str] Name of the encoding used to encode or decode the dataset.

layer [int or str] The integer index or name of a layer in a multi-layer dataset.

vfs [str] This is a deprecated parameter. A URI scheme such as "zip://" should be used instead.

enabled_drivers [list] An optional list of driver names to used when opening a collection.

crs_wkt [str] An optional WKT representation of a coordinate reference system.

kwargs [mapping] Other driver-specific parameters that will be interpreted by the OGR library as layer creation or opening options.

Collection

`fiona.`**`prop_type`**`(`*text*`)`

Returns a schema property's proper Python type.

Example:

```
>>> prop_type('int')
<class 'int'>
>>> prop_type('str:25')
<class 'str'>
```

`fiona.`**`prop_width`**`(`*val*`)`

Returns the width of a str type property.

Undefined for non-str properties. Example:

```
>>> prop_width('str:25')
25
>>> prop_width('str')
80
```

CHAPTER

FOUR

COMMAND LINE INTERFACE

Fiona's new command line interface is a program named "fio".

```
Usage: fio [OPTIONS] COMMAND [ARGS]...

  Fiona command line interface.

Options:
  -v, --verbose     Increase verbosity.
  -q, --quiet       Decrease verbosity.
  --version         Show the version and exit.
  --gdal-version    Show the version and exit.
  --python-version  Show the version and exit.
  --help            Show this message and exit.

Commands:
  bounds   Print the extent of GeoJSON objects
  calc     Calculate GeoJSON property by Python expression
  cat      Concatenate and print the features of datasets
  collect  Collect a sequence of features.
  distrib  Distribute features from a collection.
  dump     Dump a dataset to GeoJSON.
  env      Print information about the fio environment.
  filter   Filter GeoJSON features by python expression.
  info     Print information about a dataset.
  insp     Open a dataset and start an interpreter.
  load     Load GeoJSON to a dataset in another format.
  ls       List layers in a datasource.
  rm       Remove a datasource or an individual layer.
```

It is developed using the `click` package and is new in 1.1.6.

4.1 bounds

New in 1.4.5.

Fio-bounds reads LF or RS-delimited GeoJSON texts, either features or collections, from stdin and prints their bounds with or without other data to stdout.

With no options, it works like this:

```
$ fio cat docs/data/test_uk.shp | head -n 1 \
> | fio bounds
[0.735, 51.357216, 0.947778, 51.444717]
```

Using `--with-id` gives you

```
$ fio cat docs/data/test_uk.shp | head -n 1 \
> | fio bounds --with-id
{"id": "0", "bbox": [0.735, 51.357216, 0.947778, 51.444717]}
```

4.2 calc

New in 1.7b1

The calc command creates a new property on GeoJSON features using the specified expression.

The expression is evaluated in a restricted namespace containing 4 functions (*sum*, *pow*, *min*, *max*), the *math* module, the shapely *shape* function, type conversions (*bool*, *int*, *str*, *len*, *float*), and an object *f* representing the feature to be evaluated. This *f* object allows access in javascript-style dot notation for convenience.

The expression will be evaluated for each feature and its return value will be added to the properties as the specified property_name. Existing properties will not be overwritten by default (an *Exception* is raised).

```
$ fio cat data.shp | fio calc sumAB  "f.properties.A + f.properties.B"
```

4.3 cat

The cat command concatenates the features of one or more datasets and prints them as a JSON text sequence of features. In other words: GeoJSON feature objects, possibly pretty printed, optionally separated by ASCII RS (x1e) chars using *–rs*.

The output of `fio cat` can be piped to `fio load` to create new concatenated datasets.

```
$ fio cat docs/data/test_uk.shp docs/data/test_uk.shp \
> | fio load /tmp/double.shp --driver Shapefile
$ fio info /tmp/double.shp --count
96
$ fio info docs/data/test_uk.shp --count
48
```

New in 1.4.0.

4.4 collect

The collect command takes a JSON text sequence of GeoJSON feature objects, such as the output of `fio cat` and writes a GeoJSON feature collection.

```
$ fio cat docs/data/test_uk.shp docs/data/test_uk.shp \
> | fio collect > /tmp/collected.json
$ fio info /tmp/collected.json --count
96
```

New in 1.4.0.

4.5 distrib

The inverse of fio-collect, fio-distrib takes a GeoJSON feature collection and writes a JSON text sequence of GeoJSON feature objects.

```
$ fio info --count tests/data/coutwildrnp.shp
67
$ fio cat tests/data/coutwildrnp.shp | fio collect | fio distrib | wc -l
67
```

New in 1.4.0.

4.6 dump

The dump command reads a vector dataset and writes a GeoJSON feature collection to stdout. Its output can be piped to `fio load` (see below).

```
$ fio dump docs/data/test_uk.shp --indent 2 --precision 2 | head
{
  "features": [
    {
      "geometry": {
        "coordinates": [
          [
            [
              0.9,
              51.36
            ],
```

You can optionally dump out JSON text sequences using `--x-json-seq`. Since version 1.4.0, `fio cat` is the better tool for generating sequences.

```
$ fio dump docs/data/test_uk.shp --precision 2 --x-json-seq | head -n 2
{"geometry": {"coordinates": [[[0.9, 51.36], [0.89, 51.36], [0.79, 51.37], [0.78, 51.
↪37], [0.77, 51.38], [0.76, 51.38], [0.75, 51.39], [0.74, 51.4], [0.73, 51.41], [0.
↪74, 51.43], [0.75, 51.44], [0.76, 51.44], [0.79, 51.44], [0.89, 51.42], [0.9, 51.
↪42], [0.91, 51.42], [0.93, 51.4], [0.94, 51.39], [0.94, 51.38], [0.95, 51.38], [0.
↪95, 51.37], [0.95, 51.37], [0.94, 51.37], [0.9, 51.36], [0.9, 51.36]]], "type":
↪"Polygon"}, "id": "0", "properties": {"AREA": 244820.0, "CAT": 232.0, "CNTRY_NAME":
↪"United Kingdom", "FIPS_CNTRY": "UK", "POP_CNTRY": 60270708.0}, "type": "Feature"}
{"geometry": {"coordinates": [[[-4.66, 51.16], [-4.67, 51.16], [-4.67, 51.16], [-4.67,
↪ 51.17], [-4.67, 51.19], [-4.67, 51.19], [-4.67, 51.2], [-4.66, 51.2], [-4.66, 51.
↪19], [-4.65, 51.16], [-4.65, 51.16], [-4.65, 51.16], [-4.66, 51.16]]], "type":
↪"Polygon"}, "id": "1", "properties": {"AREA": 244820.0, "CAT": 232.0, "CNTRY_NAME":
↪"United Kingdom", "FIPS_CNTRY": "UK", "POP_CNTRY": 60270708.0}, "type": "Feature"}
```

4.7 info

The info command prints information about a dataset as a JSON object.

```
$ fio info docs/data/test_uk.shp --indent 2
{
```

(continues on next page)

(continued from previous page)

```
  "count": 48,
  "crs": "+datum=WGS84 +no_defs +proj=longlat",
  "driver": "ESRI Shapefile",
  "bounds": [
    -8.621389,
    49.911659,
    1.749444,
    60.844444
  ],
  "schema": {
    "geometry": "Polygon",
    "properties": {
      "CAT": "float:16",
      "FIPS_CNTRY": "str:80",
      "CNTRY_NAME": "str:80",
      "AREA": "float:15.2",
      "POP_CNTRY": "float:15.2"
    }
  }
}
```

You can process this JSON using, e.g., underscore-cli.

```
$ fio info docs/data/test_uk.shp | underscore extract count
48
```

You can also optionally get single info items as plain text (not JSON) strings

```
$ fio info docs/data/test_uk.shp --count
48
$ fio info docs/data/test_uk.shp --bounds
-8.621389 49.911659 1.749444 60.844444
```

4.8 load

The load command reads GeoJSON features from stdin and writes them to a vector dataset using another format.

```
$ fio dump docs/data/test_uk.shp \
> | fio load /tmp/test.shp --driver Shapefile
```

This command also supports GeoJSON text sequences. RS-separated sequences will be detected. If you want to load LF-separated sequences, you must specfiy `--x-json-seq`.

```
$ fio cat docs/data/test_uk.shp | fio load /tmp/foo.shp --driver Shapefile
$ fio info /tmp/foo.shp --indent 2
{
  "count": 48,
  "crs": "+datum=WGS84 +no_defs +proj=longlat",
  "driver": "ESRI Shapefile",
  "bounds": [
    -8.621389,
    49.911659,
    1.749444,
    60.844444
```

(continues on next page)

(continued from previous page)

```
  ],
  "schema": {
    "geometry": "Polygon",
    "properties": {
      "AREA": "float:24.15",
      "CNTRY_NAME": "str:80",
      "POP_CNTRY": "float:24.15",
      "FIPS_CNTRY": "str:80",
      "CAT": "float:24.15"
    }
  }
}
```

The underscore-cli process command is another way of turning a GeoJSON feature collection into a feature sequence.

```
$ fio dump docs/data/test_uk.shp \
> | underscore process \
> 'each(data.features,function(o){console.log(JSON.stringify(o))})' \
> | fio load /tmp/test-seq.shp --x-json-seq --driver Shapefile
```

4.9 filter

The filter command reads GeoJSON features from stdin and writes the feature to stdout *if* the provided expression evalutates to *True* for that feature.

The python expression is evaluated in a restricted namespace containing 3 functions (*sum*, *min*, *max*), the *math* module, the shapely *shape* function, and an object *f* representing the feature to be evaluated. This *f* object allows access in javascript-style dot notation for convenience.

If the expression evaluates to a "truthy" value, the feature is printed verbatim. Otherwise, the feature is excluded from the output.

```
$ fio cat data.shp \
> | fio filter "f.properties.area > 1000.0" \
> | fio collect > large_polygons.geojson
```

Would create a geojson file with only those features from *data.shp* where the area was over a given threshold.

4.10 rm

The `fio rm` command deletes an entire datasource or a single layer in a multi-layer datasource. If the datasource is composed of multiple files (e.g. an ESRI Shapefile) all of the files will be removed.

```
$ fio rm countries.shp
$ fio rm --layer forests land_cover.gpkg
```

New in 1.8.0.

4.11 Coordinate Reference System Transformations

The `fio cat` command can optionally transform feature geometries to a new coordinate reference system specified with `--dst_crs`. The `fio collect` command can optionally transform from a coordinate reference system specified with `--src_crs` to the default WGS84 GeoJSON CRS. Like collect, `fio load` can accept non-WGS84 features, but as it can write files in formats other than GeoJSON, you can optionally specify a `--dst_crs`. For example, the WGS84 features read from docs/data/test_uk.shp,

```
$ fio cat docs/data/test_uk.shp --dst_crs EPSG:3857 \
> | fio collect --src_crs EPSG:3857 > /tmp/foo.json
```

make a detour through EPSG:3857 (Web Mercator) and are transformed back to WGS84 by fio cat. The following,

```
$ fio cat docs/data/test_uk.shp --dst_crs EPSG:3857 \
> | fio load --src_crs EPSG:3857 --dst_crs EPSG:4326 --driver Shapefile \
> /tmp/foo.shp
```

does the same thing, but for ESRI Shapefile output.

New in 1.4.2.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Kent1978] William Kent, Data and Reality, North Holland, 1978.

[ESRI1998] ESRI Shapefile Technical Description. July 1998. http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[GeoJSON] http://geojson.org

[JSON] http://www.ietf.org/rfc/rfc4627

[SFA] http://en.wikipedia.org/wiki/Simple_feature_access

PYTHON MODULE INDEX

f

fiona, 60
fiona.collection, 47
fiona.compat, 49
fiona.crs, 49
fiona.drvsupport, 49
fiona.env, 49
fiona.errors, 52
fiona.fio, 47
fiona.fio.bounds, 45
fiona.fio.calc, 45
fiona.fio.cat, 45
fiona.fio.collect, 45
fiona.fio.distrib, 45
fiona.fio.dump, 45
fiona.fio.env, 45
fiona.fio.filter, 46
fiona.fio.helpers, 46
fiona.fio.info, 46
fiona.fio.insp, 46
fiona.fio.load, 46
fiona.fio.ls, 46
fiona.fio.main, 46
fiona.fio.options, 47
fiona.fio.rm, 47
fiona.inspector, 53
fiona.io, 53
fiona.logutils, 54
fiona.ogrext, 54
fiona.path, 56
fiona.rfc3339, 57
fiona.schema, 57
fiona.session, 58
fiona.transform, 59
fiona.vfs, 60

INDEX

A

archive (*fiona.path.ParsedPath attribute*), 56
at_least() (*fiona.env.GDALVersion method*), 50
AWSSession (*class in fiona.session*), 58

B

bounds() (*fiona.collection.Collection property*), 47
bounds() (*in module fiona*), 61
buffer_to_virtual_file() (*in module fiona.ogrext*), 56
BytesCollection (*class in fiona.collection*), 47

C

cb_layer() (*in module fiona.fio.options*), 47
cb_multilayer() (*in module fiona.fio.options*), 47
close() (*fiona.collection.BytesCollection method*), 47
close() (*fiona.collection.Collection method*), 47
close() (*fiona.ogrext.MemoryFileBase method*), 55
closed() (*fiona.collection.Collection property*), 47
Collection (*class in fiona.collection*), 47
configure_logging() (*in module fiona.fio.main*), 46
credentialize() (*fiona.env.Env method*), 50
credentials() (*fiona.session.AWSSession property*), 58
credentials() (*fiona.session.GSSession property*), 58
crs() (*fiona.collection.Collection property*), 47
crs_wkt() (*fiona.collection.Collection property*), 47
CRSError, 52

D

DataIOError, 52
DatasetDeleteError, 52
default_options() (*fiona.env.Env class method*), 50
defenv() (*in module fiona.env*), 51
delenv() (*in module fiona.env*), 51
driver() (*fiona.collection.Collection property*), 48
DriverError, 52
DriverIOError, 52
drivers() (*fiona.env.Env method*), 50
DriverSupportError, 52
DummySession (*class in fiona.session*), 58

E

ensure_env() (*in module fiona.env*), 51
ensure_env_with_credentials() (*in module fiona.env*), 51
Env (*class in fiona.env*), 49
env_ctx_if_needed() (*in module fiona.env*), 51
EnvError, 52
eval_feature_expression() (*in module fiona.fio.helpers*), 46
exists() (*fiona.ogrext.MemoryFileBase method*), 55

F

FeatureBuilder (*class in fiona.ogrext*), 54
featureRT() (*in module fiona.ogrext*), 56
FieldNameEncodeError, 52
FieldSkipLogFilter (*class in fiona.logutils*), 54
filter() (*fiona.collection.Collection method*), 48
filter() (*fiona.logutils.FieldSkipLogFilter method*), 54
fiona (*module*), 60
fiona.collection (*module*), 47
fiona.compat (*module*), 49
fiona.crs (*module*), 49
fiona.drvsupport (*module*), 49
fiona.env (*module*), 49
fiona.errors (*module*), 52
fiona.fio (*module*), 47
fiona.fio.bounds (*module*), 45
fiona.fio.calc (*module*), 45
fiona.fio.cat (*module*), 45
fiona.fio.collect (*module*), 45
fiona.fio.distrib (*module*), 45
fiona.fio.dump (*module*), 45
fiona.fio.env (*module*), 45
fiona.fio.filter (*module*), 46
fiona.fio.helpers (*module*), 46
fiona.fio.info (*module*), 46
fiona.fio.insp (*module*), 46
fiona.fio.load (*module*), 46

fiona.fio.ls (*module*), 46
fiona.fio.main (*module*), 46
fiona.fio.options (*module*), 47
fiona.fio.rm (*module*), 47
fiona.inspector (*module*), 53
fiona.io (*module*), 53
fiona.logutils (*module*), 54
fiona.ogrext (*module*), 54
fiona.path (*module*), 56
fiona.rfc3339 (*module*), 57
fiona.schema (*module*), 57
fiona.session (*module*), 58
fiona.transform (*module*), 59
fiona.vfs (*module*), 60
FionaDateTimeType (*class in fiona.rfc3339*), 57
FionaDateType (*class in fiona.rfc3339*), 57
FionaDeprecationWarning, 53
FionaError, 53
FionaTimeType (*class in fiona.rfc3339*), 57
FionaValueError, 53
flush() (*fiona.collection.Collection method*), 48
from_defaults() (*fiona.env.Env class method*), 50
from_epsg() (*in module fiona.crs*), 49
from_foreign_session() (*fiona.session.Session static method*), 58
from_path() (*fiona.session.Session static method*), 59
from_string() (*in module fiona.crs*), 49
from_uri() (*fiona.path.ParsedPath class method*), 56

G

GDALVersion (*class in fiona.env*), 50
GDALVersionError, 53
GeometryTypeValidationError, 53
get() (*fiona.collection.Collection method*), 48
get() (*fiona.ogrext.Session method*), 55
get_credential_options() (*fiona.session.AWSSession method*), 58
get_credential_options() (*fiona.session.DummySession method*), 58
get_credential_options() (*fiona.session.GSSession method*), 58
get_credential_options() (*fiona.session.Session method*), 59
get_crs() (*fiona.ogrext.Session method*), 55
get_crs_wkt() (*fiona.ogrext.Session method*), 55
get_driver() (*fiona.ogrext.Session method*), 55
get_extent() (*fiona.ogrext.Session method*), 55
get_feature() (*fiona.ogrext.Session method*), 55
get_fileencoding() (*fiona.ogrext.Session method*), 55
get_filetype() (*in module fiona.collection*), 49
get_length() (*fiona.ogrext.Session method*), 55
get_schema() (*fiona.ogrext.Session method*), 55
getenv() (*in module fiona.env*), 51
group() (*fiona.rfc3339.group_accessor method*), 57
group_accessor (*class in fiona.rfc3339*), 57
GSSession (*class in fiona.session*), 58
guard_driver_mode() (*fiona.collection.Collection method*), 48

H

has_feature() (*fiona.ogrext.Session method*), 55
hascreds() (*fiona.session.GSSession class method*), 58
hascreds() (*in module fiona.env*), 51
hasenv() (*in module fiona.env*), 51

I

id_record() (*in module fiona.fio.helpers*), 46
is_credentialized() (*fiona.env.Env property*), 50
is_local() (*fiona.path.ParsedPath property*), 56
is_remote() (*fiona.path.ParsedPath property*), 56
is_remote() (*in module fiona.vfs*), 60
isactive() (*fiona.ogrext.Session method*), 56
items() (*fiona.collection.Collection method*), 48
ItemsIterator (*class in fiona.ogrext*), 54
Iterator (*class in fiona.ogrext*), 54

K

keys() (*fiona.collection.Collection method*), 48
KeysIterator (*class in fiona.ogrext*), 54

L

listlayers() (*in module fiona*), 61
LogFiltering (*class in fiona.logutils*), 54

M

main() (*in module fiona.inspector*), 53
major (*fiona.env.GDALVersion attribute*), 50
make_ld_context() (*in module fiona.fio.helpers*), 46
MemoryFile (*class in fiona.io*), 53
MemoryFileBase (*class in fiona.ogrext*), 54
meta() (*fiona.collection.Collection property*), 48
minor (*fiona.env.GDALVersion attribute*), 50

N

name() (*fiona.path.ParsedPath property*), 56
name() (*fiona.path.UnparsedPath property*), 57
next() (*fiona.collection.Collection method*), 48
normalize_field_type() (*in module fiona.schema*), 57
nullable() (*in module fiona.fio.helpers*), 46
NullContextManager (*class in fiona.env*), 51

O

obj_gen() (*in module fiona.fio.helpers*), 46

OGRFeatureBuilder (*class in fiona.ogrext*), 55
open() (*fiona.io.MemoryFile method*), 53
open() (*fiona.io.ZipMemoryFile method*), 54
open() (*in module fiona*), 61

P

parse() (*fiona.env.GDALVersion class method*), 50
parse_date() (*in module fiona.rfc3339*), 57
parse_datetime() (*in module fiona.rfc3339*), 57
parse_path() (*in module fiona.path*), 57
parse_paths() (*in module fiona.vfs*), 60
parse_time() (*in module fiona.rfc3339*), 57
ParsedPath (*class in fiona.path*), 56
Path (*class in fiona.path*), 56
path (*fiona.path.ParsedPath attribute*), 56
path (*fiona.path.UnparsedPath attribute*), 57
profile() (*fiona.collection.Collection property*), 48
prop_type() (*in module fiona*), 62
prop_width() (*in module fiona*), 62

R

read() (*fiona.ogrext.MemoryFileBase method*), 55
remove_virtual_file() (*in module fiona.ogrext*), 56
require_gdal_version() (*in module fiona.env*), 51
runtime() (*fiona.env.GDALVersion class method*), 51

S

schema() (*fiona.collection.Collection property*), 48
SchemaError, 53
scheme (*fiona.path.ParsedPath attribute*), 56
seek() (*fiona.ogrext.MemoryFileBase method*), 55
Session (*class in fiona.ogrext*), 55
Session (*class in fiona.session*), 58
setenv() (*in module fiona.env*), 52
start() (*fiona.ogrext.Session method*), 56
start() (*fiona.ogrext.WritingSession method*), 56
stop() (*fiona.ogrext.Session method*), 56
strencode() (*in module fiona.compat*), 49
sync() (*fiona.ogrext.WritingSession method*), 56

T

tell() (*fiona.ogrext.MemoryFileBase method*), 55
ThreadEnv (*class in fiona.env*), 51
to_string() (*in module fiona.crs*), 49
TransactionError, 53
transform() (*in module fiona.transform*), 59
transform_geom() (*in module fiona.transform*), 59

U

UnparsedPath (*class in fiona.path*), 56
UnsupportedGeometryTypeError, 53

V

valid_vsi() (*in module fiona.vfs*), 60
validate_multilayer_file_index() (*in module fiona.fio.options*), 47
validate_record() (*fiona.collection.Collection method*), 48
validate_record_geometry() (*fiona.collection.Collection method*), 48
values() (*fiona.collection.Collection method*), 48
vsi_path() (*in module fiona.path*), 57
vsi_path() (*in module fiona.vfs*), 60

W

with_context_env() (*in module fiona.fio*), 47
write() (*fiona.collection.Collection method*), 49
write() (*fiona.ogrext.MemoryFileBase method*), 55
writerecords() (*fiona.collection.Collection method*), 49
writerecs() (*fiona.ogrext.WritingSession method*), 56
WritingSession (*class in fiona.ogrext*), 56

Z

ZipMemoryFile (*class in fiona.io*), 54