GEDCOM parser for Python Documentation

Release 0.4.4

Andy Salnikov

May 01, 2021

CONTENTS

Contents:

CHAPTER

ONE

GEDCOM PARSER FOR PYTHON

Implementation of the GEDCOM parser in Python

- Free software: MIT license
- Documentation: https://ged4py.readthedocs.io.

1.1 Features

- Parsing of GEDCOM files as defined by 5.5.1 version of GEDCOM standard
- Supported file encodings are UTF-8 (with or without BOM), ASCII or ANSEL
- Designed to parse large files efficiently
- Supports Python 3.6+

1.2 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER

TWO

INSTALLATION

2.1 Stable release

To install GEDCOM parser for Python, run this command in your terminal:

```
$ pip install ged4py
```

This is the preferred method to install GEDCOM parser for Python, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for GEDCOM parser for Python can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/andy-z/ged4py
```

Or download the tarball:

```
$ curl  -OL https://github.com/andy-z/ged4py/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER

THREE

USAGE

Currently *ged4py* supports parsing of existing GEDCOM files, there is no support for (re-)generating GEDCOM data. The main interface for parsing is *ged4py.parser.GedcomReader* class. To create parser instance one has to pass file with GEDCOM data as a single required parameter, this can be either file name of a Python file object. If file object is passed then the file has to be open in a binary mode and it has to support `seek()` and `tell()` methods. Example of instantiating a parser:

```
from ged4py import GedcomReader

path = "/path/to/file.gedcom"
with GedcomReader(path) as parser:
    # GedcomReader provides context support
    ...
```

or using in-memory buffer as a file (could be useful for testing):

```
import io
from ged4py import GedcomReader

data = b"..."                  # make some binary date here
with io.BytesIO(data) as file:
    parser = GedcomReader(file)
    ...
```

In most cases parser should be able to determine input file encoding from the file if data in the file follows GEDCOM specification. In other cases parser may need external help, if you know file encoding you can provide it as an argument to parser:

```
parser = GedcomReader(path, encoding="utf-8")
```

Any encoding supported by Python `codecs` module can be used as an argument. In addition, this package registers two additional encodings from the ansel package:

ansel	American National Standard for Extended Latin Alphabet Coded Character Set for Bibliographic Use (ANSEL)
gedcom	GEDCOM extensions for ANSEL

By default parser raises exception if it encounters errors while decoding data in a file. To override this behavior one can specify different error policy, following the same pattern as standard `codecs.decode()` method, e.g.:

```
parser = GedcomReader(path, encoding="utf-8", errors='replace')
```

Main mode of operation for parser is iterating over records in a file in sequential manner. GEDCOM records are organized in hierarchical structures, and ged4py parser facilitates access to these hierarchies by grouping records in tree-like structures. Instead of providing iterator over every record in a file parser iterates over top-level (level 0) records, and for each level-0 record it returns record structure which includes nested records below level 0.

The main method of the parser is the method *records0()* which returns iterator over all level-0 records. Method takes an optional argument for a tag name, without argument all level-0 records are returned by iterator (starting with "HEAD" and ending with "TRLR"). If tag is given then only the records with matching tag are returned:

```
with GedcomReader(path) as parser:
    # iterate over all INDI records
    for record in parser.records0("INDI"):
        ....
```

Records returned by iterator are instances of class *ged4py.model.Record* or one of its few sub-classes. Each record instance has a small set of attributes:

- `level` - record level, 0 for top-level records
- `xref_id` - record reference ID, may be `None`
- `tag` - record tag name
- `value` - record value, can be `None`, string, or value of some other type depending on record type
- `sub_records` - list of subordinate records, direct sub-records of this record, it is easier to access items in this list using methods described below.

If, for example, GEDCOM file contains sequence of records like this:

```
0 @ID12345@ INDI
1 NAME John /Smith/
1 BIRT
2 DATE 1 JAN 1901
2 PLAC Some place
1 FAMC @ID45623@
1 FAMS @ID7612@
```

then the record object returned from iterator will have these attributes:

- `level` is 0 (true for all records returned by *records0()*),
- `xref_id` - "@ID12345@",
- `tag` - "INDI",
- `value` - `None`,
- `sub_records` - list of *Record* instances corresponding to "NAME", "BIRT", "FAMC", and "FAMS" tags (but not "DATE" or "PLAC", records for these tags will be in `sub_records` of "BIRT" record).

Record class has few convenience methods:

- *sub_tags()* - return all direct subordinate records with a given tag name, list of records is returned, possibly empty.
- *sub_tag()* - return subordinate record with a given tag name (or tag "path"), if there is more than one record with matching tag then first one is returned, without match `None` is returned.
- *sub_tag_value()* - return value of subordinate record with a given tag name (or tag "path"), or `None` if record is not found or its value is `None`.

With the example records from above one can do `record.sub_tag("BIRT/DATE")` on level-0 record to retrieve a *Record* instance corresponding to level-2 "DATE" record, or alternatively use `record.sub_tag_value("BIRT/DATE")` to retrieve the `value` attribute of the same record.

There are few specialized sub-classes of *Record* each corresponding to specific record tag:

- NAME records generate *ged4py.model.NameRec* instances, this class knows how to split name representation into name components (first, last, maiden) and has attributes for accessing those.
- DATE records generate *ged4py.model.Date* instances, the `value` attribute of this class is converted into *ged4py.date.DateValue* instance.
- INDI records are represented by *ged4py.model.Individual* class.
- "pointer" records whose `value` has special GEDCOM <POINTER> syntax (`@xref_id@`) are represented by *ged4py.model.Pointer* class. This class has special property `ref` which returns referenced record. Methods *sub_tag()* and *sub_tag_value()* have keyword argument `follow` which can be set to `True` to allow automatic dereferencing of the pointer records.

CHAPTER

FOUR

GED4PY API

ged4py	Top-level package for GEDCOM parser for Python.

4.1 ged4py

Top-level package for GEDCOM parser for Python.

Most of the code in the s package is located in individual modules:

- *ged4py.parser* - defines *GedcomReader* class which is the main entry point for the whole package;
- *ged4py.model* - collection of classes constituting ged4py data model;
- *ged4py.calendar* - classes for working with calendar dates;
- *ged4py.date* - parsing and handling of GEDCOM dates;
- *ged4py.detail* - few modules for implementation details.

GedcomReader class can be imported directly from top-level package as:

```
from ged4py import GedcomReader
```

Modules

ged4py.calendar	Module for parsing and representing calendar dates in gedcom format.
ged4py.date	Module for parsing and representing dates in gedcom format.
ged4py.detail	
ged4py.model	Module containing Python in-memory model for GEDCOM data.
ged4py.parser	Module containing methods for parsing GEDCOM files.

4.1.1 ged4py.calendar

Module for parsing and representing calendar dates in gedcom format.

Classes

CalendarDate(year[, month, day, bc, original])	Interface for calendar date representation.
CalendarDateVisitor()	Interface for implementation of Visitor pattern for *CalendarDate* classes.
CalendarType(value)	Namespace for constants defining names of calendars.
FrenchDate(year[, month, day, bc, original])	Implementation of *CalendarDate* for French Republican calendar.
GregorianDate(year[, month, day, bc, ...])	Implementation of *CalendarDate* for Gregorian calendar.
HebrewDate(year[, month, day, bc, original])	Implementation of *CalendarDate* for Hebrew calendar.
JulianDate(year[, month, day, bc, original])	Implementation of *CalendarDate* for Julian calendar.

class ged4py.calendar.**CalendarType**(*value*)

Bases: enum.Enum

Namespace for constants defining names of calendars.

Note that it does not define constants for ROMAN calendar which is declared in GEDCOM standard as a placeholder for future definition, or UNKNOWN calendar which is not supported by this library.

The constants defined in this namespace are used for the values of the *CalendarDate.calendar* attribute. Each separate class implementing *CalendarDate* interface uses distinct value for that attribute, and this value can be used to deduce actual type of the *CalendarDate* instance.

GREGORIAN = 'GREGORIAN'

This is the value assigned to *GregorianDate.calendar* attribute.

JULIAN = 'JULIAN'

This is the value assigned to *JulianDate.calendar* attribute.

HEBREW = 'HEBREW'

This is the value assigned to *HebrewDate.calendar* attribute.

FRENCH_R = 'FRENCH R'

This is the value assigned to *FrenchDate.calendar* attribute.

class ged4py.calendar.**CalendarDate**(*year*, *month=None*, *day=None*, *bc=False*, *original=None*)

Bases: object

Interface for calendar date representation.

Parameters

year [int] Calendar year number. If bc parameter is True then this year is before "epoch" of that calendar.

month [str] Name of the month. Optional, but if day is given then month cannot be None.

day [int] Day in a month, optional.

bc [bool] True if year has "B.C."

original [str] Original string representation of this date as it was specified in GEDCOM file, could be None.

Notes

This class defines attributes and methods that are common for all calendars defined in GEDCOM (though the meaning and representation can be different in different calendars). In GEDCOM date consists of year, month, and day; day and month are optional (either day or day+month), year must be present. Day is a number, month is month name in a given calendar. Year is a number optionally followed by B.C. or /NUMBER (latter is defined for Gregorian calendar only).

Implementation for different calendars are provided by subclasses which can implement additional attributes or methods. All subclasses need to implement *key()* method to support ordering of the dates from different calendars. There are presently four implementations defined in this module:

- *GregorianDate* for "GREGORIAN" calendar
- *JulianDate* for "JULIAN" calendar
- *HebrewDate* for "HEBREW" calendar
- *FrenchDate* for "FRENCH R" calendar

To implement type-specific code on client side one can use one of these approaches:

- dispatch based on the value of *calendar* attribute, it has one of the values defined in *CalendarType* enum, the value maps uniquely to an implementation class;
- dispatch based on the type of the instance using isinstance method to check the type (e.g. isinstance(date, GregorianDate));
- double dispatch (visitor pattern) by implementing *CalendarDateVisitor* interface.

Attributes

calendar Calendar used for this date, one of the *CalendarType* enums

year_str Calendar year in string representation, this can include dual year and/or B.C.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
months()	Ordered list of month names (in GEDCOM format) defined in calendar.
parse(datestr)	Parse <DATE> string and make *CalendarDate* from it.

year
Calendar year number (int)

month
Month name or None (str)

day
Day number or None (int)

bc

Flag which is `True` if year has a "B.C" suffix (`bool`).

original
Original string representation of this date as it was specified in GEDCOM file, could be `None` (`str`).

month_num
Integer month number (1-based) or `None` if month name is not given or unknown (`int`).

abstract classmethod months()
Ordered list of month names (in GEDCOM format) defined in calendar.

abstract key()
Return ordering key for this instance.

Returned key is a tuple with two numbers (jd, flag). `jd` is the Julian Day number as floating point, `flag` is an integer flag. If month or day is not known then last month or last day should be returned in its place (in corresponding calendar, and converted to JD) and `flag` should be set to 1. If date and month are known then flag should be set to 0.

property year_str
Calendar year in string representation, this can include dual year and/or B.C. suffix (`str`)

abstract property calendar
Calendar used for this date, one of the *CalendarType* enums (*CalendarType*)

abstract accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*CalendarDateVisitor*] Visitor instance.

Returns

value [`object`] Value returned from a visitor method.

classmethod parse(*datestr*)
Parse `<DATE>` string and make *CalendarDate* from it.

Parameters

datestr [`str`] String with GEDCOM date.

Returns

date [*CalendarDate*] Date instance.

Raises

ValueError Raised if parsing fails.

class ged4py.calendar.**FrenchDate**(*year*, *month=None*, *day=None*, *bc=False*, *original=None*)
Bases: *ged4py.calendar.CalendarDate*

Implementation of *CalendarDate* for French Republican calendar.

All parameters have the same meaning as in *CalendarDate* class.

Attributes

calendar Calendar used for this date, one of the *CalendarType* enums

year_str Calendar year in string representation, this can include dual year and/or B.C.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
months()	Ordered list of month names (in GEDCOM format) defined in calendar.
parse(datestr)	Parse <DATE> string and make *CalendarDate* from it.

classmethod months()
Ordered list of month names (in GEDCOM format) defined in calendar.

key()
Return ordering key for this instance.

property calendar
Calendar used for this date, one of the *CalendarType* enums (*CalendarType*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*CalendarDateVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.calendar.**GregorianDate**(*year*, *month=None*, *day=None*, *bc=False*, *original=None*, *dual_year=None*)
Bases: *ged4py.calendar.CalendarDate*

Implementation of *CalendarDate* for Gregorian calendar.

Parameter dual_year (and corresponding attribute) is used for dual year. Other parameters have the same meaning as in *CalendarDate* class.

Parameters

dual_year [int, optional] Dual year number or None. Actual year should be given, not just two last digits.

Notes

In GEDCOM Gregorian calendar dates are allowed to specify year in the form YEAR1/YEAR2 (a.k.a.) dual-dating. Second number is used to specify year as if calendar year starts in January, while the first number is used for actual calendar year which starts at different date. Note that GEDCOM specifies that dual year uses just two last digits in the dual year number, though some implementations use 4 digits. This class expects actual year number (e.g. as if it was specified as "1699/1700").

Attributes

calendar Calendar used for this date, one of the *CalendarType* enums

year_str Calendar year in string representation, this can include dual year and/or B.C.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
months()	Ordered list of month names (in GEDCOM format) defined in calendar.
parse(datestr)	Parse <DATE> string and make *CalendarDate* from it.

dual_year
If not None then this number represent year in a calendar with year starting on January 1st (int).

classmethod months()
Ordered list of month names (in GEDCOM format) defined in calendar.

property calendar
Calendar used for this date, one of the *CalendarType* enums (*CalendarType*)

key()
Return ordering key for this instance.

property year_str
Calendar year in string representation, this can include dual year and/or B.C. suffix (str)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*CalendarDateVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.calendar.**HebrewDate**(*year*, *month=None*, *day=None*, *bc=False*, *original=None*)
Bases: *ged4py.calendar.CalendarDate*

Implementation of *CalendarDate* for Hebrew calendar.

All parameters have the same meaning as in *CalendarDate* class.

Attributes

calendar Calendar used for this date, one of the *CalendarType* enums

year_str Calendar year in string representation, this can include dual year and/or B.C.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
months()	Ordered list of month names (in GEDCOM format) defined in calendar.
parse(datestr)	Parse <DATE> string and make *CalendarDate* from it.

classmethod months()
Ordered list of month names (in GEDCOM format) defined in calendar.

key()
Return ordering key for this instance.

property calendar
Calendar used for this date, one of the *CalendarType* enums (*CalendarType*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*CalendarDateVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.calendar.**JulianDate**(*year*, *month=None*, *day=None*, *bc=False*, *original=None*)
Bases: *ged4py.calendar.CalendarDate*

Implementation of *CalendarDate* for Julian calendar.

All parameters have the same meaning as in *CalendarDate* class.

Attributes

calendar Calendar used for this date, one of the *CalendarType* enums

year_str Calendar year in string representation, this can include dual year and/or B.C.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
months()	Ordered list of month names (in GEDCOM format) defined in calendar.
parse(datestr)	Parse <DATE> string and make *CalendarDate* from it.

classmethod months()
Ordered list of month names (in GEDCOM format) defined in calendar.

key()

Return ordering key for this instance.

property calendar
Calendar used for this date, one of the *CalendarType* enums (*CalendarType*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*CalendarDateVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.calendar.**CalendarDateVisitor**
Bases: object

Interface for implementation of Visitor pattern for *CalendarDate* classes.

One can easily extend behavior of the *CalendarDate* class hierarchy without modifying classes themselves. Clients need to implement new behavior by sub-classing *CalendarDateVisitor* and calling *CalendarDate.accept()* method, e.g.:

```
class FormatterVisitor(CalendarDateVisitor):

    def visitGregorian(self, date):
        return "Gregorian date:" + str(date)

    # and so on for each date type

visitor = FormatterVisitor()

date = CalendarDate.parse(date_string)
formatted = date.accept(visitor)
```

Methods

visitFrench(date)	Visit an instance of *FrenchDate* type.
visitGregorian(date)	Visit an instance of *GregorianDate* type.
visitHebrew(date)	Visit an instance of *HebrewDate* type.
visitJulian(date)	Visit an instance of *JulianDate* type.

abstract visitGregorian(*date*)
Visit an instance of *GregorianDate* type.

Parameters

date [*GregorianDate*] Date instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *CalendarDate.accept()* method.

abstract visitJulian(*date*)

Visit an instance of *JulianDate* type.

Parameters

date [*JulianDate*] Date instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *CalendarDate.accept()* method.

abstract visitHebrew(*date*)
Visit an instance of *HebrewDate* type.

Parameters

date [*HebrewDate*] Date instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *CalendarDate.accept()* method.

abstract visitFrench(*date*)
Visit an instance of *FrenchDate* type.

Parameters

date [*FrenchDate*] Date instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *CalendarDate.accept()* method.

4.1.2 ged4py.date

Module for parsing and representing dates in gedcom format.

Classes

DateValue(key)	Representation of the <DATE_VALUE>, can be exact date, range, period, etc.
DateValueAbout(date)	Implementation of *DateValue* interface for ABT date.
DateValueAfter(date)	Implementation of *DateValue* interface for AFT date.
DateValueBefore(date)	Implementation of *DateValue* interface for BEF date.
DateValueCalculated(date)	Implementation of *DateValue* interface for CAL date.
DateValueEstimated(date)	Implementation of *DateValue* interface for EST date.
DateValueFrom(date)	Implementation of *DateValue* interface for FROM date.
DateValueInterpreted(date, phrase)	Implementation of *DateValue* interface for INT date.
DateValuePeriod(date1, date2)	Implementation of *DateValue* interface for FROM .
DateValuePhrase(phrase)	Implementation of *DateValue* interface for phrase-date.
DateValueRange(date1, date2)	Implementation of *DateValue* interface for BET .

continues on next page

Table 10 – continued from previous page

DateValueSimple(date)	Implementation of *DateValue* interface for simple single-value DATE.
DateValueTo(date)	Implementation of *DateValue* interface for TO date.
DateValueTypes(value)	Namespace for constants defining types of date values.
DateValueVisitor()	Interface for implementation of Visitor pattern for *DateValue* classes.

class ged4py.date.**DateValueTypes**(*value*)

Bases: enum.Enum

Namespace for constants defining types of date values.

The constants defined in this namespace are used for the values of the *DateValue.kind* attribute. Each separate class implementing *DateValue* interface uses distinct value for that attribute, and this value can be used to deduce actual type of the date *DateValue* instance.

SIMPLE = 'SIMPLE'
Date value consists of a single CalendarDate, corresponding implementation class is *DateValueSimple*.

FROM = 'FROM'
Period of dates starting at specified date, end date is unknown, corresponding implementation class is *DateValueFrom*

TO = 'TO'
Period of dates ending at specified date, start date is unknown, corresponding implementation class is *DateValueTo*.

PERIOD = 'PERIOD'
Period of dates starting at one date and ending at another, corresponding implementation class is *DateValuePeriod*.

BEFORE = 'BEFORE'
Date value for an event known to happen before given date, corresponding implementation class is *DateValueBefore*.

AFTER = 'AFTER'
Date value for an event known to happen after given date, corresponding implementation class is *DateValueAfter*.

RANGE = 'RANGE'
Date value for an event known to happen between given dates, corresponding implementation class is *DateValueRange*.

ABOUT = 'ABOUT'
Date value for an event known to happen at approximate date, corresponding implementation class is *DateValueAbout*.

CALCULATED = 'CALCULATED'
Date value for an event calculated from other known information, corresponding implementation class is *DateValueCalculated*.

ESTIMATED = 'ESTIMATED'
Date value for an event estimated from other known information, corresponding implementation class is *DateValueEstimated*.

INTERPRETED = 'INTERPRETED'
Date value for an event interpreted from a specified phrase, corresponding implementation class is *DateValueInterpreted*.

PHRASE = 'PHRASE'
Date value for an event is a phrase, corresponding implementation class is *DateValuePhrase*.

class ged4py.date.**DateValue**(*key*)
Bases: object

Representation of the <DATE_VALUE>, can be exact date, range, period, etc.

Parameters

key [object] Object that is used for ordering, usually it is a pair of *CalendarDate* instances but can be None.

Notes

DateValue is an abstract base class, for each separate kind of GEDCOM date there is a separate concrete class. Class method *parse* is used to parse a date string and return an instance of corresponding sub-class of DateValue type.

There are presently 12 concrete classes implementing this interface (e.g. *DateValueSimple*, *DateValueRange*, etc.) Different types have somewhat different set of attributes, to implement type-specific code on client side one can use one of these approaches:

- dispatch based on the value of *kind* attribute, it has one of the values defined in *DateValueTypes* namespace, and that value maps uniquely to a corresponding sub-class of *DateValue*;
- dispatch based on the type of the instance using isinstance method to check the type (e.g. isinstance(date, DateValueRange));
- double dispatch (visitor pattern) by implementing *DateValueVisitor* interface.

Attributes

kind The type of GEDCOM date, one of the *DateValueTypes* enums (*DateValueTypes*).

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

classmethod parse(*datestr*)
Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

Parameters

datestr [str] String with GEDCOM date, range, period, etc.

Returns

date_value [*DateValue*] Object representing the date value.

abstract property kind
The type of GEDCOM date, one of the *DateValueTypes* enums (*DateValueTypes*).

key()
Return ordering key for this instance.

If this instance has a range of dates associated with it then this method returns the range as pair of dates. If this instance has a single date associated with it then this method returns pair which includes the date twice. For other dates (PHRASE is the only instance without date) it returns a a pair of fixed but arbitrary dates in the future.

Returns

key [tuple [*CalendarDate*]] Key used for ordering.

abstract accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueAbout**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for ABT date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueAbout class this is always *DateValueTypes.ABOUT*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind

For DateValueAbout class this is always *DateValueTypes.ABOUT*.

property date

Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueAfter**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for AFT date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueAfter class this is always *DateValueTypes.AFTER*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind

For DateValueAfter class this is always *DateValueTypes.AFTER*.

property date

Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueBefore**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for BEF date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueBefore class this is always *DateValueTypes.BEFORE*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueBefore class this is always *DateValueTypes.BEFORE*.

property date
Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueCalculated**(*date*)
Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for CAL date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueCalculated class this is always *DateValueTypes.CALCULATED*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueCalculated class this is always *DateValueTypes.CALCULATED*.

property date
Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueEstimated**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for EST date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueEstimated class this is always *DateValueTypes.ESTIMATED*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind

For DateValueEstimated class this is always *DateValueTypes.ESTIMATED*.

property date

Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueFrom**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for FROM date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueFrom class this is always *DateValueTypes.FROM*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueFrom class this is always *DateValueTypes.FROM*.

property date
Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueInterpreted**(*date*, *phrase*)
Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for INT date.

Parameters

date [*CalendarDate*] Corresponding date.

phrase [str] Phrase string associated with this date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueInterpreted class this is always *DateValueTypes.INTERPRETED*.

phrase Phrase associated with this date (str)

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueInterpreted class this is always *DateValueTypes.INTERPRETED*.

property date
Calendar date corresponding to this instance (*CalendarDate*)

property phrase

Phrase associated with this date (str)

accept (*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValuePeriod**(*date1*, *date2*)
Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for FROM . . . TO date.

Parameters

date1 [*CalendarDate*] FROM date.

date2 [*CalendarDate*] TO date.

Attributes

date1 First Calendar date corresponding to this instance (*CalendarDate*)

date2 Second Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValuePeriod class this is always *DateValueTypes.PERIOD*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValuePeriod class this is always *DateValueTypes.PERIOD*.

property date1
First Calendar date corresponding to this instance (*CalendarDate*)

property date2
Second Calendar date corresponding to this instance (*CalendarDate*)

accept (*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValuePhrase**(*phrase*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for phrase-date.

Parameters

phrase [str] Phrase string associated with this date.

Attributes

kind For DateValuePhrase class this is always *DateValueTypes.PHRASE*.

phrase Phrase associated with this date (str)

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind

For DateValuePhrase class this is always *DateValueTypes.PHRASE*.

property phrase

Phrase associated with this date (str)

accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueRange**(*date1*, *date2*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for BET ... AND ... date.

Parameters

date1 [*CalendarDate*] First date.

date2 [*CalendarDate*] Second date.

Attributes

date1 First Calendar date corresponding to this instance (*CalendarDate*)

date2 Second Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueRange class this is always *DateValueTypes.RANGE*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueRange class this is always *DateValueTypes.RANGE*.

property date1
First Calendar date corresponding to this instance (*CalendarDate*)

property date2
Second Calendar date corresponding to this instance (*CalendarDate*)

accept (*visitor*)
Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueSimple**(*date*)
Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for simple single-value DATE.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueSimple class this is always *DateValueTypes.SIMPLE*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind
For DateValueSimple class this is always *DateValueTypes.SIMPLE*.

property date
Calendar date corresponding to this instance (*CalendarDate*)

accept (*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueTo**(*date*)

Bases: *ged4py.date.DateValue*

Implementation of *DateValue* interface for TO date.

Parameters

date [*CalendarDate*] Corresponding date.

Attributes

date Calendar date corresponding to this instance (*CalendarDate*)

kind For DateValueTo class this is always *DateValueTypes.TO*.

Methods

accept(visitor)	Implementation of visitor pattern.
key()	Return ordering key for this instance.
parse(datestr)	Parse string <DATE_VALUE> string and make *DateValue* instance out of it.

property kind

For DateValueTo class this is always *DateValueTypes.TO*.

property date

Calendar date corresponding to this instance (*CalendarDate*)

accept(*visitor*)

Implementation of visitor pattern.

Each concrete sub-class will implement this method by dispatching the call to corresponding visitor method.

Parameters

visitor [*DateValueVisitor*] Visitor instance.

Returns

value [object] Value returned from a visitor method.

class ged4py.date.**DateValueVisitor**

Bases: object

Interface for implementation of Visitor pattern for *DateValue* classes.

One can easily extend behavior of the *DateValue* class hierarchy without modifying classes themselves. Clients need to implement new behavior by sub-classing DateValueVisitor and calling *DateValue.accept* method, e.g.:

```
class FormatterVisitor(DateValueVisitor):

    def visitSimple(self, date):
        return "Simple date: " + str(date.date)

    # and so on for each date type

visitor = FormatterVisitor()

date = DateValue.parse(date_string)
formatted = date.accept(visitor)
```

Methods

visitAbout(date)	Visit an instance of *DateValueAbout* type.
visitAfter(date)	Visit an instance of *DateValueAfter* type.
visitBefore(date)	Visit an instance of *DateValueBefore* type.
visitCalculated(date)	Visit an instance of *DateValueCalculated* type.
visitEstimated(date)	Visit an instance of *DateValueEstimated* type.
visitFrom(date)	Visit an instance of *DateValueFrom* type.
visitInterpreted(date)	Visit an instance of *DateValueInterpreted* type.
visitPeriod(date)	Visit an instance of *DateValuePeriod* type.
visitPhrase(date)	Visit an instance of *DateValuePhrase* type.
visitRange(date)	Visit an instance of *DateValueRange* type.
visitSimple(date)	Visit an instance of *DateValueSimple* type.
visitTo(date)	Visit an instance of *DateValueTo* type.

abstract visitSimple(*date*)

Visit an instance of *DateValueSimple* type.

Parameters

date [*DateValueSimple*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitPeriod(*date*)

Visit an instance of *DateValuePeriod* type.

Parameters

date [*DateValuePeriod*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitFrom(*date*)

Visit an instance of *DateValueFrom* type.

Parameters

date [*DateValueFrom*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitTo(*date*)
Visit an instance of *DateValueTo* type.

Parameters

date [*DateValueTo*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitRange(*date*)
Visit an instance of *DateValueRange* type.

Parameters

date [*DateValueRange*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitBefore(*date*)
Visit an instance of *DateValueBefore* type.

Parameters

date [*DateValueBefore*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitAfter(*date*)
Visit an instance of *DateValueAfter* type.

Parameters

date [*DateValueAfter*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitAbout(*date*)
Visit an instance of *DateValueAbout* type.

Parameters

date [*DateValueAbout*] Date value instance.

Returns

value [`object`] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitCalculated(*date*)
Visit an instance of *DateValueCalculated* type.

Parameters

date [*DateValueCalculated*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitEstimated(*date*)
Visit an instance of *DateValueEstimated* type.

Parameters

date [*DateValueEstimated*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitInterpreted(*date*)
Visit an instance of *DateValueInterpreted* type.

Parameters

date [*DateValueInterpreted*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

abstract visitPhrase(*date*)
Visit an instance of *DateValuePhrase* type.

Parameters

date [*DateValuePhrase*] Date value instance.

Returns

value [object] Implementation of this method can return anything, value will be returned from *DateValue.accept()* method.

4.1.3 ged4py.detail

Modules

ged4py.detail.io	Internal module for I/O related methods.
ged4py.detail.name	Internal module for parsing names in gedcom format.

ged4py.detail.io

Internal module for I/O related methods.

Functions

check_bom(file)	Determines file codec from from its BOM record.
guess_lineno(file)	Guess current line number in a file.

Classes

BinaryFileCR(raw)	Binary file with support of CR line terminators.

ged4py.detail.io.**check_bom**(*file*)

Determines file codec from from its BOM record.

If file starts with BOM record encoded with UTF-8 or UTF-16(BE/LE) then corresponding encoding name is returned, otherwise None is returned. In both cases file current position is set to after-BOM bytes. The file must be open in binary mode and positioned at offset 0.

ged4py.detail.io.**guess_lineno**(*file*)

Guess current line number in a file.

Guessing is done in a very crude way - scanning file from beginning until current offset and counting newlines. Only meant to be used in exceptional cases - generating line number for error message.

class ged4py.detail.io.**BinaryFileCR**(*raw*)

Bases: _io.BufferedReader

Binary file with support of CR line terminators.

I need a binary file object with readline() method which supports all possible line terminators (LF, CR-LF, CR). Standard binary files have readline that only stops at LF (and hence CR-LF). This class adds a workaround for readline method to understand CR-delimited files.

Attributes

closed

mode

name

raw

Methods

close	Flush and close the IO object.
detach	Disconnect this buffer from its underlying raw stream and return it.
fileno	Returns underlying file descriptor if one exists.
flush	Flush write buffers, if applicable.
isatty	Return whether this is an 'interactive' stream.

continues on next page

Table 28 – continued from previous page

read([size])	Read and return up to n bytes.
read1([size])	Read and return up to n bytes, with at most one read() call to the underlying raw stream.
readable	Return whether object was opened for reading.
readline([limit])	Read and return a line from the stream.
readlines([hint])	Return a list of lines from the stream.
seek(target[, whence])	Change stream position.
seekable	Return whether object supports random access.
tell	Return current stream position.
truncate([pos])	Truncate file to size bytes.
writable(/)	Return whether object was opened for writing.
write	Write the given buffer to the IO stream.
writelines(lines, /)	Write a list of lines to stream.

peek	
readinto	
readinto1	

CR = b'\r'

LF = b'\n'

readline(*limit=- 1*)

Read and return a line from the stream.

If size is specified, at most size bytes will be read.

The line terminator is always b'n' for binary files; for text files, the newlines argument to open can be used to select the line terminator(s) recognized.

ged4py.detail.name

Internal module for parsing names in gedcom format.

Functions

parse_name_altree(record)	Parse NAME structure assuming ALTREE dialect.
parse_name_ancestris(record)	Parse NAME structure assuming ANCESTRIS dialect.
parse_name_myher(record)	Parse NAME structure assuming MYHERITAGE dialect.
split_name(name)	Extracts pieces of name from full name string.

ged4py.detail.name.**split_name**(*name*)

Extracts pieces of name from full name string.

Parameters

name [str] Full name string.

Returns

name [tuple] 3-tuple (given1, surname, given2), surname or given will be empty strings if they are not present in full string.

Notes

Full name can have one of these formats:

```
<NAME_TEXT> |
/<NAME_TEXT>/ |
<NAME_TEXT> /<NAME_TEXT>/ |
/<NAME_TEXT>/ <NAME_TEXT> |
<NAME_TEXT> /<NAME_TEXT>/ <NAME_TEXT>
```

<NAME_TEXT> can include almost anything excluding commas, numbers, special characters (though some test files use numbers for the names). Text between slashes is considered a surname, outside slashes - given name.

This method splits full name into pieces at slashes, e.g.:

```
"First /Last/" -> ("First", "Last", "")
"/Last/ First" -> ("", "Last", "First")
"First /Last/ Jr." -> ("First", "Last", "Jr.")
"First Jr." -> ("First Jr.", "", "")
```

ged4py.detail.name.**parse_name_altree**(*record*)

Parse NAME structure assuming ALTREE dialect.

Parameters

record [*ged4py.model.Record*] NAME record.

Returns

parsed_name [tuple] Tuple with 3 or 4 elements, first three elements of tuple are the same as returned from *split_name* method, fourth element (if present) denotes maiden name.

Notes

In ALTREE dialect maiden name (if present) is saved as SURN sub-record and is also appended to family name in parens. Given name is saved in GIVN sub-record. Few examples:

No maiden name:

```
1 NAME John /Smith/
2 GIVN John
```

With maiden name:

```
1 NAME Jane /Smith (Ivanova)/
2 GIVN Jane
2 SURN Ivanova
```

No maiden name:

```
1 NAME Mers /Daimler (-Benz)/
2 GIVN Mers
```

Because family name can also contain parens it's not enough to parse family name and guess maiden name from it, we also have to check for SURN record.

ALTREE also replaces empty names with question mark, we undo that too.

ged4py.detail.name.**parse_name_myher**(*record*)
Parse NAME structure assuming MYHERITAGE dialect.

Parameters

record [*ged4py.model.Record*] NAME record.

Returns

parsed_name [tuple] Tuple with 3 or 4 elements, first three elements of tuple are the same as returned from *split_name* method, fourth element (if present) denotes maiden name.

Notes

In MYHERITAGE dialect married name (if present) is saved as _MARNM sub-record. Maiden name is stored in SURN record. Few examples:

No maiden name:

```
1 NAME John /Smith/
2 GIVN John
2 SURN Smith
```

With maiden name:

```
1 NAME Jane /Ivanova/
2 GIVN Jane
2 SURN Ivanova
2 _MARNM Smith
```

No maiden name:

```
1 NAME Mers /Daimler (-Benz)/
2 GIVN Mers
2 SURN Daimler (-Benz)
```

ged4py.detail.name.**parse_name_ancestris**(*record*)
Parse NAME structure assuming ANCESTRIS dialect.

As far as I can tell Ancestris does not have any standard convention for representing maiden or married names. Best we can do in this situation is to use NAME record value and ignore any other fields.

Parameters

record [*ged4py.model.Record*] NAME record.

Returns

parsed_name [tuple] Tuple with 3 or 4 elements, first three elements of tuple are the same as returned from *split_name* method, fourth element (if present) denotes maiden name.

4.1.4 ged4py.model

Module containing Python in-memory model for GEDCOM data.

Functions

make_record(level, xref_id, tag, value, ...)	Create *Record* instance based on parameters.

Classes

Date()	Sub-class of *Record* representing the DATE record.
Dialect(value)	Even though the structure of GEDCOM file is more or less fixed, interpretation of some data may vary depending on which application produced GEDCOM file.
Individual()	Sub-class of *Record* representing the INDI record.
Name(names, dialect)	Class representing "summary" of person names.
NameOrder(value)	Names/Individuals can be ordered differently, e.g.
NameRec()	Sub-class of *Record* representing the NAME record.
Pointer(parser)	Sub-class of *Record* representing a pointer to a record in a GEDCOM file.
Record()	Class representing a parsed GEDCOM record in a generic format.

ged4py.model.**make_record**(*level*, *xref_id*, *tag*, *value*, *sub_records*, *offset*, *dialect*, *parser=None*) → *ged4py.model.Record*

Create *Record* instance based on parameters.

Parameters

level [int] Record level number.

xref_id [str] Record reference ID, possibly empty.

tag [str] Tag name.

value [str] Record value, possibly empty. Value can be None, bytes, or string object, if it is bytes then it should be decoded into strings before calling freeze(), this is normally done by the parser which knows about encodings.

sub_records [list [*Record*]] Initial list of subordinate records, possibly empty. List can be updated later.

offset [int] Record location in a file.

dialect [Dialect] One of Dialect enums.

parser [*GedcomReader*] Parser instance, only needed for pointer records.

Returns

record [*Record*] Instance of *Record* (or one of its subclasses).

Notes

This is the factory method for record instances, it can create different types of record based on tag of value:

- if value has a pointer form (`@ref_id@`) then *Pointer* instance is created
- if tag is "INDI" then *Individual* instance is created
- if tag is "NAME" then *NameRec* instance is created
- if tag is "DATE" then *Date* instance is created
- otherwise *Record* instance is created

Returned record is not complete, it could be updated by parser. When parser finishes updates it calls *Record.freeze()* method to finalize record construction.

class ged4py.model.**Record**

Bases: `object`

Class representing a parsed GEDCOM record in a generic format.

This is the main element of the data model, it represents records in GEDCOM files. Each GEDCOM records consists of small number of items:

- level number, integer;
- optional reference ID, string in format `@identifier@`;
- tag name, short string;
- optional record value, arbitrary string, for pointer records the record value is the reference ID of some other record.

For many record types GEDCOM specifies subordinate (nested) records with incremental level number.

Record class defines an interface that makes it easier to navigate this complex hierarchy of subordinate and referenced records:

- `sub_records` attribute contains the list of all immediate subordinate records of this record.
- *sub_tag* method find subordinate record given its tag, it can do it recursively if tag name contains multiple levels separated by slashes, and it can navigate through the pointer records transparently if `follow` argument is `True`.
- *sub_tag_value* is a convenience method that finds a subordinate record (via *sub_tag* call) but returns value of the record instead of record itself. This simplifies handling of missing tags.
- *sub_tags* returns the list of immediate subordinate records (no recursion). It is useful when multiple sub-records with the same tag can exist.

There are few sub-classes of the `Record` class providing additional methods or facilities for specific tag types.

In general it is impossible to define what constitutes value or identity of GEDCOM record, so comparison of the records does not make sense. Similarly hashing operation cannot be used on Record instances, and the class is explicitly marked as non-hashable.

Client code usually does not need to create instances of this class directly, *make_record()* should be used instead. If you create an instance of this class (or its subclass) then you are responsible for filling its attributes.

Attributes

level [`int`] Record level number

xref_id [`str`] Record reference ID, possibly empty.

tag [`str`] Tag name

value [object] Record value, possibly None, for many record types value is a string or None, some subclasses can define different type of record value.

sub_records [list [*Record*]] List of subordinate records, possibly empty.

offset [int] Record location in a file.

dialect: `Dialect` GEDCOM source dialect, one of the Dialect enums.

Methods

freeze()	Method called by parser when updates to this record finish.
sub_tag(path[, follow])	Finds and returns sub-record with given tag name.
sub_tag_value(path[, follow])	Returns value of a direct sub-record.
sub_tags(*tags[, follow])	Returns a list of sub-records matching any tag name.

freeze() → *ged4py.model.Record*

Method called by parser when updates to this record finish.

Some sub-classes will override this method to implement conversion of record data to different representation.

Returns

self [*Record*] Finalized record instance.

sub_tag(*path*, *follow=True*) → Optional[*ged4py.model.Record*]

Finds and returns sub-record with given tag name.

Path can be a simple tag name, in which case the first direct sub-record of this record with the matching tag is returned. Path can also consist of several tags separated by slashes, in that case sub-records are searched recursively.

If follow is True then pointer records are resolved and pointed record is used instead of pointer record, this also works for all intermediate records in a path.

Parameters

path [str] One or more tag names separated by slashes.

follow [bool] If True then resolve pointers.

Returns

record [*Record*] Subordinate record or None if sub-record with a given tag does not exist.

sub_tag_value(*path*, *follow=True*) → Any

Returns value of a direct sub-record.

Works as *sub_tag()* but returns value of a sub-record instead of sub-record itself.

Parameters

path [str] One or more tag names separated by slashes.

follow [bool] If True then resolve pointers.

Returns

value [object] Subordinate record value or None if sub-record with a given tag does not exist.

sub_tags(**tags: str, follow: bool = True*) → List[*ged4py.model.Record*]
Returns a list of sub-records matching any tag name.

If no positional arguments are provided then all direct sub-records of this record are returned, pointers are resolved if `follow` is True. If one or more positional arguments are given then this method returns all sub-records, direct or nested, that match any of the given tags.

If `follow` is True then pointer records are resolved and pointed record is used instead of pointer record, this also works for all intermediate records in a path.

Parameters

***tags** [`str`] Each positional argument is one or more tag names separated by slashes.

follow [`bool`, optional] If True then resolve pointers.

Returns

records [`list` [*Record*]] List of records, possibly empty.

class ged4py.model.**Pointer**(*parser*)
Bases: *ged4py.model.Record*

Sub-class of *Record* representing a pointer to a record in a GEDCOM file.

This class wraps a GEDCOM pointer value and adds a `ref` property which retrieves pointed object. Instance of this class will be used in place of the GEDCOM pointers in the objects created by parser.

Parameters

parser [*ged4py.parser.GedcomReader*] Instance of parser class.

Attributes

value [`str`] Value of the GEDCOM pointer (e.g. "@I1234@")

ref [*Record*] Referenced GEDCOM record.

Methods

freeze()	Method called by parser when updates to this record finish.
sub_tag(path[, follow])	Finds and returns sub-record with given tag name.
sub_tag_value(path[, follow])	Returns value of a direct sub-record.
sub_tags(*tags[, follow])	Returns a list of sub-records matching any tag name.

property ref

class ged4py.model.**NameRec**
Bases: *ged4py.model.Record*

Sub-class of *Record* representing the NAME record.

This class adds an additional method for determining type of the name. It also redefines the type of the `value` attribute, it's type is tuple. Value tuple can contain 3 or 4 elements, if there are 4 elements then last element is a maiden name. Second element of a tuple is surname, first and third elements are pieces of the given name (this is determined entirely by how name is represented in GEDCOM file). Any of the elements can be empty string. If NAME record value is empty in GEDCOM file then all three fields of the tuple will be empty strings. Few examples:

```
("John", "Smith", "")
("Mary Joan", "Smith", "", "Ivanova")     # maiden name
("", "Ivanov", "Ivan Ivanovich")
("John", "Smith", "Jr.")
("", "", "")                              # empty NAME record
```

Client code usually does not need to create instances of this class directly, *make_record()* should be used instead.

Attributes

type Name type as defined in TYPE record.

Methods

freeze()	Method called by parser when updates to this record finish.
sub_tag(path[, follow])	Finds and returns sub-record with given tag name.
sub_tag_value(path[, follow])	Returns value of a direct sub-record.
sub_tags(*tags[, follow])	Returns a list of sub-records matching any tag name.

freeze()
Method called by parser when updates to this record finish.

Returns

self [*NameRec*] Finalized record instance.

property type
Name type as defined in TYPE record. None if TYPE record is missing, otherwise string, e.g. "aka", "birth", "immigrant", "maiden", "married" (or anything else).

class ged4py.model.**Name**(*names*, *dialect*)
Bases: object

Class representing "summary" of person names.

Parameters

names [list [*NameRec*]] List of NAME records (*NameRec* instances).

dialect [Dialect] One of Dialect enums.

Notes

Person in GEDCOM can have multiple NAME records, e.g. "aka" name, "maiden" name, etc. This class provides simple interface for selecting "best" name from all existing names. The algorithm for choosing best options is:

- If there are no NAME records then it makes an empty name (with all empty components)
- If there is only one NAME record then it is used for person name.
- If there are multiple NAME records then the first record without TYPE sub-record is used, or if all records have TYPE sub-records then first NAME record is used.

Attributes

first First name is the first part of a given name (drops middle name)

given Given name could include both first and middle name (`str`)

maiden Maiden last name, can be `None` (`str`)

surname Person surname (`str`)

Methods

format()	Format name for output.
order(order)	Return name order key.

property surname
Person surname (`str`)

property given
Given name could include both first and middle name (`str`)

property first
First name is the first part of a given name (drops middle name)

property maiden
Maiden last name, can be `None` (`str`)

order(*order*)
Return name order key.

Returns tuple with two strings that can be compared to other such tuple obtained from different name. Note that if you want locale-dependent ordering then you need to compare strings using locale-aware method (e.g. `locale.strxfrm`).

Parameters

order [`NameOrder`] One of the `NameOrder` enums.

Returns

order [`tuple` [`str`]] Tuple of two strings.

format()
Format name for output.

There is no single correct way to represent name, values returned from this method are only useful in limited context, e.g. for logging.

Returns

name [`str`] Formatted name representation.

class ged4py.model.**Date**
Bases: *ged4py.model.Record*

Sub-class of *Record* representing the DATE record.

After *freeze()* method is called by parser the `value` attribute contains instance of *ged4py.date.DateValue* class.

Methods

freeze()	Method called by parser when updates to this record finish.
sub_tag(path[, follow])	Finds and returns sub-record with given tag name.
sub_tag_value(path[, follow])	Returns value of a direct sub-record.
sub_tags(*tags[, follow])	Returns a list of sub-records matching any tag name.

freeze()
Method called by parser when updates to this record finish.

Returns

self [*Date*] Finalized record instance.

class ged4py.model.**Individual**
Bases: *ged4py.model.Record*

Sub-class of *Record* representing the INDI record.

INDI record represents a single person in GEDCOM. This class defines few methods that are useful shortcuts for accessing person information, such as navigation to parent records, name, etc.

Client code usually does not need to create instances of this class directly, *make_record()* should be used instead.

Attributes

father Parent of this individual (*Individual* or None)

mother Parent of this individual (*Individual* or None)

name Person name (*Name*).

sex Person sex, one of "M", "F", or "U" for unknown (str).

Methods

freeze()	Method called by parser when updates to this record finish.
sub_tag(path[, follow])	Finds and returns sub-record with given tag name.
sub_tag_value(path[, follow])	Returns value of a direct sub-record.
sub_tags(*tags[, follow])	Returns a list of sub-records matching any tag name.

property name
Person name (*Name*).

property sex
Person sex, one of "M", "F", or "U" for unknown (str).

property mother
Parent of this individual (*Individual* or None)

property father
Parent of this individual (*Individual* or None)

4.1.5 ged4py.parser

Module containing methods for parsing GEDCOM files.

Functions

guess_codec(file[, errors, require_char, warn])	Look at file contents and guess its correct encoding.

Classes

GedcomLine(level, xref_id, tag, value, offset)	Class representing single line in a GEDCOM file.
GedcomReader(file[, encoding, errors, ...])	Main interface for reading GEDCOM files.

Exceptions

CodecError	Class for exceptions raised for codec-related errors.
IntegrityError	Class for exceptions raised for structural errors, e.g.
ParserError	Class for exceptions raised for parsing errors.

class ged4py.parser.**GedcomReader**(*file*, *encoding=None*, *errors='strict'*, *require_char=False*)

Bases: object

Main interface for reading GEDCOM files.

Parameters

file File name or file object open in binary mode, file must be seekable.

encoding [str, optional] If None (default) then file is analyzed using *guess_codec()* method to determine correct codec. Otherwise file is open using specified codec.

errors [str, optional] Controls error handling behavior during string decoding, accepts same values as standard codecs.decode method.

require_char [bool, optional] If True then exception is thrown if CHAR record is not found in a header, if False and CHAR is not in the header then codec determined from BOM or "gedcom" is used.

Notes

Instance of this class is used to read and parse single GEDCOM file. Records in GEDCOM file are transformed into instances of types defined in *ged4py.model* module, either *ged4py.model.Record* class or one of its sub-classes. Main method of access to the data in the file is by iterating over level-0 records, optionally restricted by the tag name. The method which does this is *GedcomReader.records0()*. Most commonly the code which reads GEDCOM file at the top-level loop will look like this:

```
with GedcomReader(path) as parser:
    # iterate over each INDI record in a file
    for record in parser.records0("INDI"):
        # do something with the record or navigate to other linked records
```

Attributes

dialect File dialect as one of `ged4py.model.Dialect` enums.

header Header record (*ged4py.model.Record*).

index0 List of level=0 record positions and tag names (`list[(int, str)]`).

xref0 Dictionary which maps xref_id to level=0 record position and tag name (`dict[str, (int, str)]`).

Methods

GedcomLines(offset)	Generator method for *gedcom lines*.
read_record(offset)	Read next complete record from a file starting at given position.
records0([tag])	Iterator over level=0 records with given tag.

property index0
List of level=0 record positions and tag names (`list[(int, str)]`).

property xref0
Dictionary which maps xref_id to level=0 record position and tag name (`dict[str, (int, str)]`).

property header
Header record (*ged4py.model.Record*).

property dialect
File dialect as one of `ged4py.model.Dialect` enums.

GedcomLines(*offset*)
Generator method for *gedcom lines*.

Parameters

offset [`int`] Position in the file to start reading.

Yields

line [*GedcomLine*] An object representing one line of GEDCOM file.

Raises

ParserError Raised if lines have incorrect syntax.

Notes

GEDCOM line grammar is defined in Chapter 1 of GEDCOM standard, it consists of the level number, optional reference ID, tag name, and optional value separated by spaces. Chaper 1 is pure grammar level, it does not assign any semantics to tags or levels. Consequently this method does not perform any operations on the lines other than returning the lines in their order in file.

This method iterates over all lines in input file and converts each line into *GedcomLine* class. It is an implementation detail used by other methods, most clients will not need to use this method.

records0(*tag=None*)
Iterator over level=0 records with given tag.

This is the main method of this class. Clients access data in GEDCOM files by iterating over level=0 records and then navigating to sub-records using the methods of the *Record* class.

Parameters

tag [str, optional] If tag is None (default) then return all level=0 records, otherwise return level=0 records with the given tag.

Yields

record [*Record*] Instances of *Record* or its subclasses.

read_record(*offset*)

Read next complete record from a file starting at given position.

Reads the record at given position and all its sub-records. Stops reading at EOF or next record with the same or higher (smaller) level number. File position after return from this method is not specified, re-position file if you want to read other records.

This is mostly for internal use, regular clients don't need to use it.

Parameters

offset [int] Position in the file to start reading.

Returns

record [*Record* or None] model.Record instance or None if offset points past EOF.

Raises

ParserError Raised if offsets does not point to the beginning of a record or for any parsing errors.

exception ged4py.parser.**ParserError**

Bases: Exception

Class for exceptions raised for parsing errors.

exception ged4py.parser.**CodecError**

Bases: *ged4py.parser.ParserError*

Class for exceptions raised for codec-related errors.

exception ged4py.parser.**IntegrityError**

Bases: Exception

Class for exceptions raised for structural errors, e.g. when record level nesting is inconsistent.

ged4py.parser.**guess_codec**(*file*, *errors='strict'*, *require_char=False*, *warn=True*)

Look at file contents and guess its correct encoding.

File must be open in binary mode and positioned at offset 0. If BOM record is present then it is assumed to be UTF-8 or UTF-16 encoded file. GEDCOM header is searched for CHAR record and encoding name is extracted from it, if BOM record is present then CHAR record must match BOM-defined encoding.

Parameters

file File object, must be open in binary mode.

errors [str, optional] Controls error handling behavior during string decoding, accepts same values as standard codecs.decode method.

require_char [bool, optional] If True then exception is thrown if CHAR record is not found in a header, if False and CHAR is not in the header then codec determined from BOM or "gedcom" is returned.

warn [bool, optional] If True (default) then generate error/warning messages for illegal encodings.

Returns

codec_name [str] The name of the codec in this file.

bom_size [int] Size of the BOM record, 0 if no BOM record.

Raises

CodecError Raised if codec name in file is unknown or when codec name in file contradicts codec determined from BOM.

UnicodeDecodeError Raised if codec fails to decode input lines and errors is set to "strict" (default).

class ged4py.parser.**GedcomLine**(*level: int*, *xref_id: Optional[str]*, *tag: str*, *value: bytes*, *offset: int*)

Bases: tuple

Class representing single line in a GEDCOM file.

Note: Mostly for internal use by parser, most clients do not need to know about this class.

Attributes

level [int] Alias for field number 0

xref_id [str, possibly empty or None] Alias for field number 1

tag [str, required, non-empty] Alias for field number 2

value [bytes, possibly empty or None] Alias for field number 3

offset [int] Alias for field number 4

Methods

count(value, /)	Return number of occurrences of value.
index(value[, start, stop])	Return first index of value.

property level
Record level number (int)

property xref_id
Reference for this record (str or None)

property tag
Tag name (str)

property value
Record value (bytes)

property offset
Record offset in a file (int)

CHAPTER

FIVE

TECHNICAL INFORMATION

5.1 Character encoding

GEDCOM originally provided very little support for non-Latin alphabets. To support Latin-based characters beyond ASCII set GEDCOM used ANSEL 8-bit encoding which added a bunch of diacritical marks (modifiers) and few commonly used non-ASCI characters. Support for non-Latin characters was added in latter version of GEDCOM standard, version 5.3 added wording for UNICODE support (mostly broken) and draft 5.5.1 improved situation by declaring UTF-8 encoding as supported UNICODE encoding. Several systems producing GEDCOM output today seem to have converged on UTF-8.

The encoding of GEDCOM file is determined by the content of the file itself, in particular by the `CHAR` record in the header (which is a required record), e.g.:

```
0 HEAD
  1 SOUR PAF
    2 VERS 2.1
  1 DEST ANSTFILE
  1 CHAR ANSEL
```

GEDCOM standard seems to imply that character set specified in `CHAR` record applies to everything after that record and until `TRLR` record (last record in file). My interpretation of that statement is that all header records before and including `CHAR` should be encoded with default ANSEL encoding. This may be a source of incompatibilities, I can imagine that software encoding its output in e.g. UTF-8 can decide to encode all header records in the the same UTF-8 which can cause errors if decoded using ANSEL.

Additional source of concerns is the BOM record that some applications (or many on Windows) tend to add to files encoded with UTF-8 (or UTF-16). Presence of BOM usually implies that the whole content of the file should be decoded using UTF-8/-16. This contradicts assumption that initial part of GEDCOM header is encoded in ANSEL.

Ged4py tries to make a best guess as to how it should decode input data, and it uses simple algorithm to determine that:

- if file starts with BOM record then ged4py reads the whole file using UTF-8 or UTF-16 encoding, if the `CHAR` record specifies something other than UTF-8/-16 the exception is raised;
- otherwise if file starts with regular "0" and " " ASCII characters the header is read using ANSEL encoding until `CHAR` record is met, after that reading switches to the encoding specified in that record;
- decoding errors are handled according to the mode specified when opening GEDCOM file, it can be one of standard error handling schemes defined in `codecs` module. This scheme applies to to both header (before `CHAR` record) and regular content.

See also Tamura Jones' excellent article summarizing many varieties of illegal encodings that may be present in GEDCOM files.

5.2 Name representation

GEDCOM `NAME` record defines a structured format for representing names but applications are not required to fill that structural information and can instead present name as a value part or `NAME` record in a "custom of culture" representation. Only requirement for that representation is that surname should be delimited by slash characters, e.g.:

```
0 @I1@ INDI
  1 NAME John /Smith/            -- given name and surname
0 @I2@ INDI
  1 NAME Joanne                  -- without surname
0 @I3@ INDI
  1 NAME //  .       -- surname and given name
0 @I4@ INDI
  1 NAME Sir John /Ivanoff/ Jr.  -- with prefix/suffix
```

Potentially individual can have more than one NAME record which can be distinguished by TYPE record which can be arbitrary string, GEDCOM does not define standard or allowed types. Types could be use for example to specify maiden name or names in previous marriages, e.g.:

```
0 @I1@ INDI
  1 NAME  //
  1 NAME Jeanne /d'Arc/
    2 TYPE maiden
```

Couple of application that I know of do not use TYPE records for maiden name representation instead they chose different ways to encode names. Here is how individual applications encode names.

5.2.1 Agelong Tree (Genery)

Agelong Tree produces single NAME record per individual, I don't think it is possible to make it to create more than one NAME record. Given name and and surname are encoded as value in the NAME record, and given name also appears in GIVN sub-record:

```
1 NAME Given Name /Surname/
  2 GIVN Given Name
```

If person has a maiden name then it is encoded as additional surname enclosed in parentheses, also SURN sub-record specifies maiden name:

```
1 NAME Given Name /Surname (Maiden)/
  2 GIVN Given Name
  2 SURN Maiden
```

Additionally Agelong tends to represent missing parts of names in GEDCOM file with question mark (?).

Agelong can also store name suffix and prefix, they are not included into NAME record value but stored as NPFX and NSFX sub-records:

```
1 NAME Given Name /Surname/
  2 NPFX Dr.
  2 GIVN Given Name
  2 NSFX Jr.
```

5.2.2 MyHeritage

MyHeritage Family Tree Builder can generate more than one NAME record but I could not find a way to specify TYPE of the created NAME records, likely all NAME records are created without TYPE which is not too useful.

Given name and and surname are encoded as value in the NAME record and they also appear in GIVN and SURN sub-records:

```
1 NAME Given Name /Surname/
  2 GIVN Given Name
  2 SURN Surname
```

If name of the person after marriage is different from birth/maiden name (apparently in MyHeritage this can only happen for female individuals) then married name is stored in a non-standard sub-record with `_MARNM` tag:

```
1 NAME Given Name /Maiden/
  2 GIVN Given Name
  2 SURN Maiden
  2 _MARNM Married
```

MyHeritage can also store name suffix and prefix, and also nickname in corresponding sub-records, they are not rendered in NAME record value:

```
1 NAME Given Name /Surname/
  2 NPFX Dr.
  2 GIVN Given Name
  2 SURN Surname
  2 NSFX Jr.
  2 NICK Professore
```

MyHeritage can also store few name pieces in NAME sub-records using non-standard tags such as `_AKA`, `_RNAME` (for religious name), `_FORMERNAME`, etc.

5.2.3 ged4py behavior

ged4py tries to determine individual name pieces from all info in GEDCOM records. Because interpretation of the information depends on the application which produced GEDCOM file ged4py also has to determine the application name. Application name (a.k.a. GEDCOM "dialect") is determined from file header and is stored in a `dialect` property of *GedcomReader* class (one of the DIALECT_* constants defined in *ged4py.model* module). In general naming of individuals can be overly complicated, ged4py tries to build a simpler model of person naming by determining four pieces of each individual's name:

- given name, in some cultures it can include middle (or father) name
- first name, ged4py just uses first word (before space) of given name
- last name, for married females who changed their name in marriage ged4py assumes this to be a married name
- maiden name, only applies to married females who changed their name in marriage

Here is the algorithm that ged4py uses for extracting these pieces:

- for Agelong dialect:
  - only NAME record value is used, sub-records are ignored
  - maiden name is determined from parenthesized portion of surname
  - last name is everything except maiden name in surname

  - given name is value without surname, collects everything before and after slashes in NAME value

- for MyHeritage dialect:
  - if `_MARNM` sub-record is present then it is used as last name and everything between slashes in NAME value is used as maiden name
  - otherwise everything between slashes is used as last name, maiden name is empty
  - given name is NAME value without slashes and stuff between slashes
- for other cases ("default" dialect):
  - if there is NAME record with TYPE sub-record equal 'maiden' then use surname from that record value as maiden name
  - if there is more than one NAME record choose one without TYPE sub-record as "primary" name, or use first NAME record; last name comes from primary NAME value between slashes, first name is the rest of value.

CHAPTER

SIX

EXAMPLES

This page collects several simple code examples which use *ged4py*.

6.1 Example 1

Trivial example of opening the file, iterating over INDI records (which produces *Individual* instances) and printing basic information for each person. *format()* method is used to produce printable representation of a name, though this is only one of possible ways to format names. Method *sub_tag_value()* is used to access the values of subordinate tags of the record, it can follow many levels of tags.

```
import sys
from ged4py.parser import GedcomReader

# open GEDCOM file
with GedcomReader(sys.argv[1]) as parser:
    # iterate over each INDI record in a file
    for i, indi in enumerate(parser.records0("INDI")):
        # Print a name (one of many possible representations)
        print(f"{i}: {indi.name.format()}")

        father = indi.father
        if father:
            print(f"    father: {father.name.format()}")

        mother = indi.mother
        if mother:
            print(f"    mother: {mother.name.format()}")

        # Get _value_ of the BIRT/DATE tag
        birth_date = indi.sub_tag_value("BIRT/DATE")
        if birth_date:
            print(f"    birth date: {birth_date}")

        # Get _value_ of the BIRT/PLAC tag
        birth_place = indi.sub_tag_value("BIRT/PLAC")
        if birth_place:
            print(f"    birth place: {birth_place}")
```

6.2 Example 2

This example iterates over FAM records in the file which represent family structure. FAM records do not have special record type so they produce generic *Record* instances. This example shows the use of *sub_tag()* method which can dereference pointer records contained in FAM records to retrieve corresponding INDI records.

```
import sys
from ged4py.parser import GedcomReader

with GedcomReader(sys.argv[1]) as parser:
    # iterate over each FAM record in a file
    for i, fam in enumerate(parser.records0("FAM")):

        print(f"{i}:")

        # Get records for spouses, FAM record contains pointers to INDI
        # records but sub_tag knows how to follow the pointers and return
        # the referenced records instead.
        husband, wife = fam.sub_tag("HUSB"), fam.sub_tag("WIFE")
        if husband:
            print(f"    husband: {husband.name.format()}")
        if wife:
            print(f"    wife: {wife.name.format()}")

        # Get _value_ of the MARR/DATE tag
        marr_date = fam.sub_tag_value("MARR/DATE")
        if marr_date:
            print(f"    marriage date: {marr_date}")

        # access all CHIL records, sub_tags method returns list (possibly empty)
        children = fam.sub_tags("CHIL")
        for child in children:
            # print name and date of birth
            print(f"    child: {child.name.format()}")
            birth_date = child.sub_tag_value("BIRT/DATE")
            if birth_date:
                print(f"        birth date: {birth_date}")
```

6.3 Example 3

This example shows how to specialize date formatting. Date representation in different calendars is a very complicated topic and `ged4py` cannot solve it in any general way. Instead it gives clients an option to specialize date handling in whatever way clients prefer. This is done by implementing *DateValueVisitor* interface and passing a visitor instance to *ged4py.date.DateValue.accept()* method. For completeness one also has to implement *CalendarDateVisitor* to format or do anything else to the instances of *CalendarDate*, this is not shown in the example.

```
import sys
from ged4py.parser import GedcomReader
from ged4py.date import DateValueVisitor


class DateFormatter(DateValueVisitor):
    """Visitor class that produces string representation of dates.
```

(continues on next page)

(continued from previous page)

```
    """
    def visitSimple(self, date):
        return f"{date.date}"

    def visitPeriod(self, date):
        return f"from {date.date1} to {date.date2}"

    def visitFrom(self, date):
        return f"from {date.date}"

    def visitTo(self, date):
        return f"to {date.date}"

    def visitRange(self, date):
        return f"between {date.date1} and {date.date2}"

    def visitBefore(self, date):
        return f"before {date.date}"

    def visitAfter(self, date):
        return f"after {date.date}"

    def visitAbout(self, date):
        return f"about {date.date}"

    def visitCalculated(self, date):
        return f"calculated {date.date}"

    def visitEstimated(self, date):
        return f"estimated {date.date}"

    def visitInterpreted(self, date):
        return f"interpreted {date.date} ({date.phrase})"

    def visitPhrase(self, date):
        return f"({date.phrase})"


format_visitor = DateFormatter()

with GedcomReader(sys.argv[1]) as parser:
    # iterate over each INDI record in a file
    for i, indi in enumerate(parser.records0("INDI")):
        print(f"{i}: {indi.name.format()}")

        # get all possible event types and print their dates,
        # full list of events is longer, this is only an example
        events = indi.sub_tags("BIRT", "CHR", "DEAT", "BURI", "ADOP", "EVEN")
        for event in events:
            date = event.sub_tag_value("DATE")
            # Some event types like generic EVEN can define TYPE tag
            event_type = event.sub_tag_value("TYPE")
            # pass a visitor to format the date
            if date:
                date_str = date.accept(format_visitor)
            else:
                date_str = "N/A"
```

(continues on next page)

(continued from previous page)

```
            print(f"    event: {event.tag} date: {date_str} type: {event_type}")
```

CHAPTER

SEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/andy-z/ged4py/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

GEDCOM parser for Python could always use more documentation, whether as part of the official GEDCOM parser for Python docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/andy-z/ged4py/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up *ged4py* for local development.

1. Fork the *ged4py* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/ged4py.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv ged4py
$ cd ged4py/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 ged4py tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6+. Check https://travis-ci.org/andy-z/ged4py/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ python -m unittest tests.test_ged4py
```

CHAPTER

EIGHT

CREDITS

8.1 Development Lead

- Andy Salnikov <ged4py@py-dev.com>

8.2 Contributors

- David Haney (@haney)

CHAPTER

NINE

HISTORY

9.1 0.4.4 (2021-05-01)

Add Python3.9 to tox, github test, and classifiers.

9.2 0.4.3 (2021-04-30)

- Extend behavior of `Record.sub_tags()` method.

9.3 0.4.2 (2021-04-09)

- Fix crash in `sub_tag()` with broken files

9.4 0.4.1 (2021-04-08)

- Improve handling of invalid dates

9.5 0.4.0 (2020-10-09)

- Python3 goodies, use enum classes for enums

9.6 0.3.2 (2020-10-04)

- Use numpydoc style for docstrings, add extension to Sphinx
- Drop Python2 compatibility code

9.7 0.3.1 (2020-09-28)

- Use github actions instead of Travis CI

9.8 0.3.0 (2020-09-28)

- Drop Python2 support
- Python3 supported versions are 3.6 - 3.8

9.9 0.2.4 (2020-08-30)

- Extend dialect detection for new genery.com SOUR format

9.10 0.2.3 (2020-08-29)

- Disable hashing for Record types
- Add hash method for DateValue and CalendarDate classes
- Improve ordering of DateValue instances

9.11 0.2.2 (2020-08-16)

- Fix parsing of DATE records with leading blanks

9.12 0.2.1 (2020-08-15)

- Extend documentation with examples
- Extend docstrings for few classes

9.13 0.2.0 (2020-07-05)

- Improve support for GEDCOM date types

9.14 0.1.13 (2020-04-15)

- Add support for MacOS line breaks (single CR character)

9.15 0.1.12 (2020-03-01)

- Add support for a bunch of illegal encodings (thanks @Tuisto59 for report).

9.16 0.1.11 (2019-01-06)

- Improve support for ANSEL encoded documents that use combining characters.

9.17 0.1.10 (2018-10-17)

- Add protection for empty DATE fields.

9.18 0.1.9 (2018-05-17)

- Improve exception messages, convert bytes to string

9.19 0.1.8 (2018-05-16)

- Add simple integrity checks to parser

9.20 0.1.7 (2018-04-23)

- Fix for DateValue comparison, few small improvements

9.21 0.1.6 (2018-04-02)

- Improve handling of non-standard dates, any date string that cannot be parsed according to GEDCOM syntax is assumed to be a "Date phrase"

9.22 0.1.5 (2018-03-25)

- Fix for exception due to empty NAME record

9.23 0.1.4 (2018-01-31)

- Improve name parsing for ALTREE dialect

9.24 0.1.3 (2018-01-16)

- improve Py3 compatibility

9.25 0.1.2 (2017-11-26)

- Get rid of name formatting options, too complicated for this package.
- Describe name parsing for different dialects.

9.26 0.1.1 (2017-11-20)

- Fix for missing modules.

9.27 0.1.0 (2017-07-17)

- First release on PyPI.

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX