Lifelines Python Extension Reference

David Cole Taylor

25 October 2021

Table of Contents

1 Overview

There are, at least, two potential audiences for information about the Python extension language –

- Those that want to modify the C code, whether to extend the module or to fix a bug.
- Those that want to use the module to write a report or to extend Lifelines in some fashion.

Those two audiences have vastly different needs and interests.

This document concerns itself with the **use** of the Python Lifelines module, called llines. If you wish to **modify** the C code that implements it, please consult the document "Lifelines Python Internals".

If you are already familiar with the Lifelines Report Writing language, you will have an easier time learning the Python Extension Language as there is a fair bit of similarity in the function names and their behavior.

Many of the function names are the same and take the same arguments in the same order as those in the Lifelines Report Writing language.

Three immediate differences –

- If the function operates on an instance of a particular Lifelines specific type, then instead of taking it as its first argument, it is an instance function of the type.
- The variables, if any, to be set are not passed to the function.
- All arguments have keyword names and as a result they can be given by name or by position. And – where reasonable – they are optional.

Example: if '`fam`' is a variable whose value is a family in the database, and you want the husband of the family, you might write:

```
husb = fam.husband()
```

If '`fam`' has no husband recorded, then `husband()` will return `None`.

After executing the above, `husb` is either '`None`' or an INDI record.

If `husb` is an individual (i.e., not `None`), then you could get `husb`'s name by typing:

```
husb.name()
```

or

```
husb.name(caps=False)
```

or

```
husb.name(True)
```

The first returns `husb`'s name as a string using the default capitalization for the last name. The second, returns it using the capitalization as found in the database, and the third puts it in all capital letters.

This is because '`name`' is a instance method for Individual Records. It takes one argument – `caps` – which is optional. You can let its value default (first example), give it explicitly by name (second example), or give it explicitly by position (third example).

In the function write ups that follow, the arguments are shown in positional order. The names shown are the keyword names. And if an argument is shown in []'s, then it is optional.

If the function is an instance method, the instance "type" is shown before the function name.

Also, some functions are documented in more than one place – for example, choosespouse is an instance method for individual records, so it is listed in the "Individual Records" section of the "RECORDS" chapter. But, it asks the user to choose the spouse, so it is in the "User Interaction Functions" chapter as well.

2 RECORDS

2.1 Individual Records

2.1.1 Name related

The following functions are instance methods.

`INDI` `name` ([*caps*]) → *STRING or* `None` [Function]
Returns the name found on the first '1 NAME' line of the record. Slashes are removed. If `caps` (optional) is `True` (default), the surname is made all capitals.

`INDI` `fullname` ([*upcase*],[*keep_order*],[*max_length*]) → *STRING or* `None` [Function]
Returns the name of a person in a variety of formats.

If `upcase` is `True` (default: `False`), the surname is shown in upper case, otherwise it is shown as in the record.

If `keep_order` is `True` (default: `True`), the parts are shown in the order as found in the record, otherwise the surname is given first, followed by a comma, followed by the other name parts.

Argument `max_length` (default: no maximum) specifies the maximum length that can be used to show the name. Zero means no maximum.

`INDI` `surname` (*void*) → *STRING or* `None` [Function]
Returns the surname as found in the first '1 NAME' line. Slashes are removed. If no NAME is found, `None` is returned.

`INDI` `givens` (*void*) → *STRING or* `None` [Function]
Returns the given names of the person in the same order and format as found in the first '1 NAME' line of the record. If no NAME is found, `None` is returned.

`INDI` `trimname` (*max_length*) → *STRING or* `None` [Function]
XXX should this be optional? If yes, what should be the default max_length?

`INDI` `title` (*void*) → *STRING or* `None` [Function]
Returns the first '1 TITL' line in the record or `None`.

`INDI` `soundex` (*void*) → *STRING* [Function]
Returns the SOUNDEX code of INDI.

`INDI` `pronoun` (*which*) [Function]

which	**Pronouns**	**Case**
0	He, She, It	Subjective
1	he, she, it	Subjective
2	His, Her, Its	Possessive Adjective
3	his, her, its	Possessive Adjective

4	him, her, it	Objective
5	his, hers, its	Possessive Pronoun
6	himself, herself, itself	Reflective

QUESTION: Should we allow / implement the it/its/itself column as a possible return value? If yes, when should we return one of those pronouns? My inclination is to go with tradition and return the masculine if the gender is unknown. Neuter / None / error all feel wrong...

Currently, if the SEX is neither 'M' nor 'F', the masculine choices are used.

2.1.2 Event NODE related

The following functions are instance methods.

`INDI birth` (*void*) → *EVENT or* `None` [Function]
Returns the first `BIRT` event of INDI; `None` if no event is found.

`INDI death` (*void*) → *EVENT or* `None` [Function]
Returns the first `DEAT` event of INDI; `None` if no event is found.

`INDI burial` (*void*) → *EVENT or* `None` [Function]
Returns the first burial event of INDI; `None` if no event is found.

2.1.3 Family related

The following functions are instance methods.

`INDI father` (*void*) → *INDI or* `None` [Function]
Returns the first father of INDI; `None` if no person in the role.

`INDI mother` (*void*) → *INDI or* `None` [Function]
Returns the first mother of INDI; `None` if no person in the role.

`INDI nextsib` (*void*) → *INDI or* `None` [Function]
Returns the next (younger) sibling of INDI; `None` if no person in the role.

`INDI prevsib` (*void*) → *INDI or* `None` [Function]
Returns the previous (older) sibling of INDI; `None` if no person in the role.

`INDI nspouses` (*void*) → *INTEGER* [Function]
Returns the number of spouses of INDI.

`INDI nfamilies` (*void*) → *INTEGER* [Function]
Returns the number of families of INDI.

`INDI parents` (*void*) → *FAM or* `None` [Function]
Returns the first FAM in which INDI is a child.

`INDI spouses` (*void*) → *SET* [Function]
Returns the set of spouses of INDI.

`INDI children` (*void*) → *SET* [Function]
Returns the set of children of INDI.

`INDI choosechild` (*void*) → *INDI or* `None` [Function]
Selects and returns child of person through user interface. Returns `None` if INDI has no children or if the user cancels.

`INDI choosespouse` (*void*) → *INDI or* `None` [Function]
Selects and returns spouse of person through user interface. Returns `None` if INDI has no spouses or if the user cancels.

`INDI choosefam` (*void*) [Function]
Selects and returns a family that INDI is in. Returns `None` if INDI is not in any families or if the user cancels.

The following function is part of the module but is not an instance method:

`spouseset` (*SET*) → *SET* [Function]
Returns the set of INDIs that are spouses of the input INDIs.

2.1.4 Individual related

The following functions are instance methods.

`INDI sex` (*void*) → "*M*", "*F*", "*U*" [Function]
Returns the sex of INDI ("M", "F", or "U")

XXX Currently does not distinguish between a value of "U" (unknown) and no sex specified – perhaps it should? XXX

`INDI male` (*void*) → *BOOLEAN* [Function]
Returns `True` if male, `False` otherwise.

`INDI female` (*void*) → *BOOLEAN* [Function]
Returns `True` if female, `False` otherwise.

2.1.5 Database related

The following functions are instance methods.

`INDI key` ([*strip_prefix*]) → *STRING.* [Function]
Returns the database key of the record. If `strip_prefix` is `True` (default: `False`), the non-numeric prefix is stripped.

`INDI nextindi` (*void*) → *INDI or* `None` [Function]
Returns the next INDI in the database (in key order). Returns `None` when the end is reached.

`INDI previndi` (*void*) → *INDI or* `None` [Function]
Returns the previous INDI in the database (in key order). Returns `None` when the beginning is reached.

The following functions are part of the module but are not instance methods:

`firstindi` (*void*) → *INDI or* `None` [Function]
Returns the first individual in the database (in key order). Returns `None` if there are no INDIs in the database.

In the future this MIGHT become a database instance function.

`lastindi` (*void*) → *INDI or* `None` [Function]
Returns the last individual in the database (in key order). Returns `None` if there are no INDIs in the database.

In the future this MIGHT become a database instance function.

`individuals` (*void*) → *Iterator* [Function]
Returns an iterator that when called produces all the INDI records in the database in key order.

2.2 Family Records

2.2.1 Event related

The following functions are instance methods.

`FAM marriage` (*void*) → *EVENT or* `None` [Function]
Returns first marriage event of FAM. `None` if no event is found.

2.2.2 Person related

`FAM husband` (*void*) → *INDI or* `None` [Function]
Returns the first HUSB of the family. `None` if there are none.

`FAM wife` (*void*) → *INDI or* `None` [Function]
Returns the first WIFE of the family. `None` if there are none.

`FAM nchildren` (*void*) → *INTEGER* [Function]
Returns the number of children in the family.

`FAM firstchild` (*void*) → *INDI or* `None` [Function]
Returns the first child of FAM; `None` if there are no children.

`FAM lastchild` (*void*) → *INDI or* `None` [Function]
Returns the last child of FAM; `None` if there are no children.

`FAM children` (*void*) → *SET of INDIs* [Function]
Returns the set of children in the family.

`FAM spouses` (*void*) → *SET of INDIs* [Function]
Returns the set of spouses in the family.

`FAM choosechild` (*void*) → *INDI or* `None` [Function]
Figures out FAM's collection of children and asks the user to choose one. Returns `None` if FAM has no children or if the user cancelled the operation.

`FAM choosespouse` (*void*) → *INDI or* `None` [Function]
Selects and returns spouse of family through user interface. Returns `None` if FAM has no spouses or if the user cancels.

2.2.3 Database related

The following functions are instance methods.

`FAM key` ([*strip_prefix*]) → *STRING* [Function]
Returns the database key of the record. If `strip_prefix` is `True` (default: `False`), then the non-numeric prefix is stripped.

`FAM nextfam` (*void*) → *FAM or* `None` [Function]
Returns the next family (in key order) in the database. Returns `None` if the end has been reached.

`FAM prevfam` (*void*) → *FAM or* `None` [Function]
Returns the previous family (in key order) in the database. Returns `None` if the beginning has been reached.

The following functions are part of the module but are not instance methods:

`firstfam` (*void*) → *FAM or* `None` [Function]
Returns the first family in the database (in key order). Returns `None` if there are no FAMs in the database.

In the future this MIGHT become a database instance function.

`lastfam` (*void*) → *FAM or* `None` [Function]
Returns the last family in the database (in key order). Returns `None` if there are no FAMs in the database.

In the future this MIGHT become a database instance function.

`families` (*void*) → *Iterator* [Function]
Returns an iterator that when called produces all the FAM records in the database in key order.

2.3 Event Records

When exchanging GEDCOM files with other researchers, please be aware that event records are a Lifelines extension. They are not part of standard GEDCOM.

The following function is an instance method.

`EVEN key` ([*strip_prefix*]) → *STRING* [Function]
Returns the database key of the record. If `strip_prefix` is `True` (default: `False`), then the non-numeric prefix is stripped.

The following function is part of the module but is not an instance method:

`events` (*void*) → *Iterator* [Function]
Returns an iterator that when called produces all EVEN records in the database in key order.

2.4 Source Records

The following function is an instance method.

`SOUR key` ([*strip_prefix*]) → *STRING* [Function]
Returns the database key of the record. If `strip_prefix` is `True` (default: `False`), the non-numeric prefix is stripped.

The following function is part of the module but is not an instance method:

`sources` (*void*) → *Iterator* [Function]
Returns an iterator that when called produces all the SOUR records in the database in key order.

2.5 Other Records

The following function is an instance method.

`OTHR key` ([*strip_prefix*]) → *STRING* [Function]
Returns the database key of the record. If `strip_prefix` is `True` (default: `False`), then the non-numeric prefix is stripped.

The following function is part of the module but is not an instance method:

`others` (*void*) → *Iterator* [Function]
Returns an iterator that when called produces all OTHR records in the database in key order.

2.6 Person Set Functions

The functions in this section operate on sets of INDI Records. They are not instance methods.

Note further, that while they are documented as taking a SET as input, the input need only be an iterable Python collection of INDIs – for example, it could be a list or a tuple. The output is a SET,

`parentset` (*SET*) → *SET* [Function]
Returns the set of parents of the input INDIs.

`spouseset` (*SET*) → *SET* [Function]
Returns the set of all spouses of the INDIs in the input set.

`siblingset` (*SET*) → *SET* [Function]
Returns a set consisting of all the siblings of every INDI in the input set.

`ancestorset` (*SET*) → *SET* [Function]
Returns the set of INDI ancestors of every INDI in the input set.

`descendantset` (*SET*) → *SET* [Function]
Returns the set of descendants of the input set.

`childset` (*SET*) → *SET* [Function]
Returns the set of INDIs that are children of the input INDIs.

2.7 Record Agnostic

The following functions are instance functions that apply to all types of RECORDs.

`INDI top_node` (*void*) → *NODE* [Function]
`FAM top_node` (*void*) → *NODE* [Function]
`EVEN top_node` (*void*) → *NODE* [Function]
`SOUR top_node` (*void*) → *NODE* [Function]
`OTHR top_node` (*void*) → *NODE* [Function]
Returns the NODE that is the top of the NODE tree associated with the RECORD. Works for each kind of RECORD.

`key_to_record` (*key*, [*type*]) → *RECORD or None* [Function]
Looks up and returns the database entry having the specified key. If the database key supplied is complete, then type is optional. If `strip_prefix` was specified when retrieving the key, then type is required. Argument `type`, if supplied, must be GEDCOM tag that is part of the RECORD's top node. That is, when supplied, the type must be one of the strings "FAM", "INDI", "SOUR", "EVEN", "REPO", "SUBM", "SNOTE", or "OBJE".

`keynum_to_record` (*keynum, type*) → *RECORD or None* [Function]
Looks up and returns the database entry having the specified key. Since the key is incomplete (it is just the numeric portion), the type is required. For the list of permitted values for `type`, see the function `key_to_record`, above.

3 NODES

3.1 GEDCOM Node Functions

The following functions are instance methods.

`NODE xref` (*void*) → *STRING or* `None` [Function]
Returns cross references index of NODE.

XXX Currently returns a string. Should it return the record pointed to, instead? XXX

`NODE tag` (*void*) → *STRING* [Function]
Returns NODE's tag.

`NODE value` (*void*) → *STRING or* `None` [Function]
Returns NODE's value. XXX check: does it return `None` if no value? or empty string? XXX

`NODE parent_node` (*void*) → *NODE or None* [Function]
Returns NODE's parent node. Returns None if already at the top of the tree.

`NODE child_node` (*void*) → *NODE or None* [Function]
Returns NODE's first child node. Returns None if NODE has no children.

`NODE sibling_node` (*void*) → *NODE* [Function]
Returns NODE's next sibling. Returns None if there are no more siblings.

`NODE copy_node_tree` (*void*) → *NODE* [Function]
Returns a copy of the node structure.

`NODE level` (*void*) → *INTEGER* [Function]
Returns NODE's level.

NOTE: If NODE is not part of a RECORD (e.g., part of a disconnected NODE tree), then level is just the distance from the top of the NODE tree.

`NODE add_node` ([*parent*],[*prev*]) → *SELF* [Function]
Adds NODE into the GEDCOM tree with specified parent and specified previous sibling. Returns the modified NODE. Node `prev` MUST be a child of `parent`. IF `prev` is None, then NODE becomes the first child of parent.

`NODE copy_node_tree` (*void*) → *NODE* [Function]
Returns a copy of the tree from the current node on down.

`NODE nodeiter` (*type*, [*tag*]) → *Iterator* [Function]
Returns an iterator that performs the specified iteration upon the `NODE` tree.

Argument `type` is either `ITER_CHILDREN` or `ITER_TRAVERSE`. If `type` is `ITER_CHILDREN`, then we iterate through the immediate children of the node. If `type` is `ITER_TRAVERSE`, then we iterate through all the nodes of the tree, depth first, parent before child.

Optional argument `tag` is a tag to search for. Only nodes with a matching tag will be returned. If `tag` is omitted, is `None`, or is the empty string, then all nodes will be returned.

The following functions are not instance methods.

`create_node` (*tag*, [*value*]) → *NODE* [Function]
Creates a node having the specified tag `tag` and value `value`. Both `tag` and (if specified) `value` are strings. The created node is returned. NOTE: This CANNOT be used to create a reference, use `create_reference` for that.

3.2 Event Nodes and Date Nodes

The following functions are instance methods.

`NODE date` (*void*) → *STRING or* `None` [Function]
Returns a string containing the value of the first DATE line of the event. If there is no DATE line, returns `None`.

`NODE place` (*void*) → *STRING or* `None` [Function]
Returns a string containing the value of the first PLAC line of the event. If there is no PLAC line, returns `None`.

`NODE year` (*void*) → *STRING or* `None` [Function]
Returns a string containing the first three or four digit number in the value of the first DATE line. This number is assumed to be the year.

`NODE long` (*void*) → *STRING or* `None` [Function]
Returns the verbatim values of the DATE and PLAC lines in an event, concatenated together and separated by a comma.

`NODE short` (*void*) → *STRING or* `None` [Function]
Abbreviates information from the first DATE and PLAC lines, concatenates the shortened information together with a comma separator, and returns it. The abbreviated date is its year. The abbreviated pace is the last component in the value, further abbreviated if the component has an entry in the place abbreviation table.

`NODE stddate` (*void*) → *STRING* [Function]
Takes the given NODE `event`, finds the date, breaks it apart, formats it according to the previously specified formats, and returns the resulting string.

NOTE: There is a non-instance version that takes a string.

The following functions affect how dates are presented. They are not instance methods.

`dayformat` ([*format*]) → *INT.* [Function]
Sets the day format for stddate calls. Returns the previous format.

-1	Do not change format, just return existing format.
0	leave space before single digit days.
1	Use leading 0 before single digit days.
2	No space or leading 0 before single digit days.

`monthformat` ([*format*]) → *INT*. [Function]

Sets the month format for stddate calls. Returns the previous format.

-1	Do not change format, just return existing format.
0	Number with space before single digit months.
1	Number with leading zero before single digit months.
2	Number with no space for zero before single digit months.
3	Upper case abbreviation (eg.g. JAN, FEB) (localized).
4	Capitalized abbreviation (e.g., Jan, Feb) (localized).
5	Upper case full word (e.g., JANUARY, FEBRUARY) (localized).
6	Capitalized full word (e.g., January, February) (localized).
7	Lower case abbreviations (e.g., jan, feb) (localized).
8	Lower case full word (e.g., january, february) (localized).
9	Upper case abbreviation in English per GEDCOM (e.g., JAN, FEB).
10	Lower case roman letter (e.g., i, ii).
11	Upper case roman letter (e.g., I, II).

`yearformat` ([*format*]) → *INT*. [Function]

Sets the year format for stddate calls. Returns the previous format.

-1	Do not change format, just return existing format.
0	Use leading spaces before years with less than four digits.
1	Use leading 0 before years with less than four digits.
2	No space or leading 0 before years.

`eraformat` ([*format*]) → *INT*. [Function]

Sets the era format for stddate calls. Returns the previous format.

-1	Do not change format, just return existing format.
0	No AD/BC markers.
1	Trailing B.C. if appropriate.
2	Trailing A.D. or B.C..
11	Trailing BC if appropriate.
12	Trailing AD or BC.
21	Trailing B.C.E. if appropriate.
22	Trailing C.E. or B.C.E..
31	Trailing BC if appropriate.
32	Trailing CE or BCE.

`dateformat` ([*format*]) → *INT*. [Function]

Sets the date format for stddate calls. Returns the previous format.

-1	Do not change format, just return existing format.
0	d0 mo yr
1	mo da, yr
2	mo/da/yr
3	da/mo/yr
4	mo-da-yr
5	da-mo-yr
6	modayr
7	damoyr
8	yr mo da
9	yr/mo/da
10	yr-mo-da
11	yrmoda
12	yr (Year only omitting all else.)
13	da/mo yr
14	As in GEDCOM.

`complexformat` (*INT format*) → *INT* [Function]

The value `format` must be either `-1` or in the range `[3,8]`. The value `-1` just returns the current value without changing it. Values in the range `[3,8]` get set and return the previous value. The value of `format` says what to do with words such as "about", "between", and "estimated" that accompany the date. Format values:

Word Length	**Upper Case**	**Title Case**	**Lower Case**
abbreviation	3	4	7
full words	5	6	8

`stddate` (*STRING date*) → *STRING* [Function]

Takes the given date, breaks it apart, formats it according to the previously specified formats, and returns the resulting string.

NOTE: There is an instance version that takes a NODE.

`complexpic` (*int which*, [*STRING format*]) → *BOOLEAN* [Function]

Parameter `which` must be one of the constants listed in the `which` column of table below which shows the default lower case strings for both abbreviation and full word formats. The upper case and title case versions are the same except for the capitalization.

If a format string is specified and is to be used due to the text present in the date, then it is used as specified – it is not abbreviated, nor is its capitalization altered.

To return to the use of the default behavior, either omit `format` or supply either a value of `None` or an empty string.

which	Abbreviated	Full words
DATE_COMPLEX_ABT	abt %1	about %1
DATE_COMPLEX_EST	est %1	estimated %1
DATE_COMPLEX_CAL	cal %1	calculted %1
DATE_COMPLEX_BEF	bef %1	before %1
DATE_COMPLEX_AFT	aft %1	after %1
DATE_COMPLEX_BET_AND	bet %1 and %2	between %1 and %2
DATE_COMPLEX_FROM	fr %1	from %1
DATE_COMPLEX_TO	to %1	to %1
DATE_COMPLEX_FROM_TO	fr %1 to %2	from %1 to %2

For now it returns True on success. This might change in the future to return the previous custom complex pic.

3.3 Value Extraction Functions

`NODE extractdate` [Function]

XXX not yet implemented XXX

`extractdatestr` [Function]

XXX not yet implemented XXX

4 User Interaction Functions

The following functions are instance methods.

`INDI choosechild` (*void*) → *INDI* [Function]
`FAM choosechild` (*void*) → *INDI* [Function]
Select child of individual or family through user interface.

`INDI choosefam` (*void*) → *FAM* [Function]
Select, through user interface, one of the families individual is in as a spouse.

`INDI choosespouse` (*void*) → *INDI* [Function]
Select spouse of Individual.

The following functions are not instance methods.

`getindi` ([*prompt*]) → *INDI* [Function]
Identify individual through user interface. Returns INDI (or `None` if user cancels).

`getindiset` ([*prompt*]) → *SET* [Function]
Identify set of persons through user interface. Returns SET (or `None` if user cancels).

XXX not yet implemented XXX

`getfam` (*void*) → *FAM* [Function]
Identify family through user interface. Returns family (or `None` if user cancels).

`getint` ([*prompt*]) → *INTEGER* [Function]
Get integer through user interface.

`getstr` ([*prompt*]) → *STRING* [Function]
Get a string through the user interface.

`chooseindi` (*SET*) → *INDI* [Function]
Select individual from a set of persons.

XXX not yet implemented XXX

`choosesubset` (*SET*) → *SET* [Function]
Select a subset of persons from a set of persons.

XXX not yet implemented XXX

`menuchoose` (*LIST*,[*prompt*]) → *INT* [Function]
Select from a list of options.

XXX not yet implemented XXX

5 Iterators

5.1 RECORD Iterators

Each RECORD type – Individuals, Families, Sources, Events, Others – has a function that returns an iterator for that type.

N.B.: There are two types of events –

- RECORD EVENTs – which are a Lifelines extension.
- NODE EVENTs – such as births and marriages.

We are concerned here with RECORDs.

NOTE: The use of an iterator for XXXX (XXXX = even, fam, indi, othr, or sour), is conceptually equivalent to first calling firstXXXX and then calling nextXXXX repeatedly until exhaustion.

For example,

```
for person in llines.individuals():
  do_something_with(person)
```

would assign the variable `person` each individual in the database, in turn, and then call `do_something_with` with `person` as its argument. The RECORD iterators are

`individuals` (*void*) [Function]
`families` (*void*) [Function]
`sources` (*void*) [Function]
`events` (*void*) [Function]
`others` (*void*) [Function]

QUESTIONS:

Should we support "small" iterators? Iterators for things like:

- all spouses of an individual
- all families of an individual
- all children of an individual
- all children of a family
- all parents of a family

Does Python support mapping functions? And if yes, how does that interact with iterators?

5.2 NODE Iterators

It is also possible to iterate over some NODEs. There is nothing to distinguish a "family" NODE from an "individual" NODE, other than which tree it is in. Additionally, there are "temp" NODEs which are NODEs that are not associated with any RECORD.

To get an iterator for a NODE, call instance function `nodeiter`. Its signature is:

`NODE noteiter` (*type*, [*tag*]) → *Iterator* [Function]
Here `type` is one of `ITER_CHILDREN` or `ITER_TRAVERSE` and `tag` is a tag to search for. See [nodeiter], page 10, for details.

6 Miscellaneous Functions

The following functions are not instance methods.

`version` (*void*) → *STRING* [Function]

Returns the version of Lifelines.

7 What is not here

7.1 Functions definitely NOT being implemented

- Arithmetic and Logic Functions
- List Functions
- Table Functions
- String Functions
- Deprecated Functions

  The functions mentioned in the aforementioned sections in their entirety will NOT be implemented.
- Trigonometric and Spherical Calculations

  The functions `sin`, `cos`, `tan`, `arcsin`, `arccos`, and `arctan` will not be implemented.
- Output Mode Functions

  The functions `test`, `nl`, `sp`, `qt`, and `print` will not be implemented. The others are yet to be decided.
- Person Set Functions and GEDCOM Extraction

  The functions `indiset`, `addtoset`, `deletefromset`, `length`, `union`, `intersect`, `difference`, `uniqueset`, and `inset` will not be implemented. Python provides all of their functionality and more, natively.

  The function `valuesort` will not be implemented as we do not collect the information it needs.

  The functions `namesort` and `keysort` will not be implemented as their functionality – and more – can be done with a Python one-liner.

  For example, to sort by names, you could do:

  ```
  sorted_indis =
    sorted(unsorted_indis,
           key=lambda indi: indi.fullname(keep_order=False))
  ```

  And to sort by keys you could do:

  ```
  sorted_indis = sorted(unsorted_indis, key=lambda indi: indi.key())
  ```
- Miscellaneous Functions

  The functions `system` and `heapused` will not be implemented.

  The function `program` will not be implemented as the functionality is available from Python via the variable `__file__`. If you wish to omit the directory part, you can use `os.path.basename(__file__)`.
- Value Extraction Functions

  The functions `extractplaces` and `extractnames` will not be implemented.

  The output of the function `extractplaces` can be obtained by simply calling `re.split` with a pattern of `'\s*,\s*'`.

  The function `extractnames` is slightly more involved, but still quite simple. Call `partition` twice with a separator of `'/'`. The first time to separate the names before

the surname from the rest. The second time to separate the surname from the parts after the surname. If you then wish the individual pieces separated, call `split` with the default separator.

7.2 Functions that MIGHT be implemented

- Trigonometric and Spherical Calculations

  The functions `dms2deg`, `deg2dms`, and `spdist` might be implemented.

- Output Mode Functions

  Some of these functions might be implemented. Uncertain.

7.3 Functions that WILL eventually be implemented

Some variant of all the currently unimplemented functions in the section "Event and Data Functions" of the Lifelines Report Writing manual" will eventually be implemented. I have held off on implementing them as I do not like the interface. However, I have been unable to come up with a better interface and the functionality is needed.

8 Debug Functions

NOTHING IN THIS CHAPTER HAS BEEN IMPLEMENTED

I am still trying to figure out the right interface and the right options. Input is both welcome and encouraged.

`getdebug` (*void*) → *Returns a collection of keyword value pairs. Data type not yet decided.* [Function]

`setdebug` (*keyword=value*[,*keyword=value*]*) [Function]

To assist in debugging, various options can be turned on or off. For each keyword, there are two permitted values – `True` and `False`. `True` turns the option on. `False` turns it off. Not specifying it leaves it alone.

Keyword	Description
`entry_exit`	Report function entry, exit, information about arguments, and return values.
`arguments`	Display information on function arguments.
`exceptions`	Display information on exceptions generated within the Lifelines module.
`reference_counts`	Report reference count activity.

9 Building Lifelines with an embedded Python interpreter

9.1 MacOS and Windows

While Lifelines runs on GNU/Linux, MacOS, and Windows, and Python also runs on all three, I have only built and tested Lifelines with Python on GNU/Linux.

It should (famous last words), build and work on both Windows and MacOS. But, neither has been tested. Feedback would be appreciated.

There is a lot of truth in the software adage "If it has not been tested, then it does not work". So, while I am not aware of any problems with building and using it on Windows or MacOS, neither would surprise me.

Feedback on success and/or failure building on MacOS and/or Windows would be appreciated. Feedback on success and/or failure with using it on Windows or MacOS would also be appreciated.

9.2 Configuring

I added the option `--with-python` to the top-level `configure.ac`. As with most `--with` options, there are four possible values for the option.

- `auto`

  The default. If it finds the necessary headers and libraries, it enables Python, otherwise it disables Python.

- `no`

  Does not search for the Python headers and libraries. Python is disabled.

- `yes`

  Searches for the Python headers and libraries. If it finds what it needs, Python is enabled. Otherwise, `configure` fails with a fatal error.

- `/path/to/a/specific/python`

  Is like `enable-python=yes`, but uses the specified Python.

Development used the Python 3.9.6 documentation. Initial builds were against Python 3.8.4 (it is what was installed on my system). Later builds were done against Python 3.9.7. Python 3.10 has been released.

I do not know what is the minimum required Python version. Certainly Python 3 is required. And Python 3.8.4 is sufficient. Beyond that, I do not know.

Success and failure stories with older versions would be useful to further refine what is the minimum required version.

If you wish to build Lifelines with Python, you will need to have the `libpython3-dev` or equivalent package installed. If you encounter other dependencies beyond those required to build Lifelines without Python, I would appreciate hearing about them.

10 Invoking The Python Extension

QUESTIONS: Should the Python interpreter be shutdown and restarted between scripts? What portions of Lifeline's initialization should be run before each script?

10.1 llines

There are two ways to invoke a Python Report, one is via the command line. But, instead of saying `-x reportname.ll`, you should instead say `-p reportname.py`. The other way is via the `p` option on the main menu.

Note: As of yet the program `llines` has not been modified. If the `p` option is unavailable in the main menu, some other letter will be chosen.

10.2 llexec

Like `llines`, there are two ways to invoke the Python extension from `llexec`. One is the same as for `llines` – namely, `-p reportname.py`. However, the other also involves the comand line, specify `-P` with no arguments. The program `llexec` will process the other command line options and then invoke the interactive Python interpreter with the Lifelines extensions linked in. Like the Lifelines report language, the Python interpreter has access to the full database.

This is especially useful for speeding up the test-edit cycle. It is also useful for small tests, for resolving documentation ambiguities and errors, and for getting your feet wet. It also allows you to run your script – as that functionality is part of the core Python interpreter! At the interactive interpreter prompt, just type:

```
exec(open("/path/to/script.py").read())
```

and it will open the script, read it, and execute the contents.

To play with it, you might try something like:

```
import llines
fam1=llines.getfam()
print(fam1)
husb1=fam1.husband()
print(husb1)
name1=husb1.name()
print(name1)
```

Appendix A Functions Being Considered

There is a good chance that NONE of these will be in the first release. Some of these might **never** be part of a release. Others might be part of a release but with a different interface than that described here.

These are, at some level, thoughts about possible future enhancements. They are not in any sense a commitment.

A.1 RECORD or NODE Related

The following functions are instance methods.

`INDI add_family` (*family, role,* [*position*]) → *INDI* [Function]
`FAM add_individual` (*individual, role,* [*position*]) → *FAM* [Function]
Adds the indicated family to the individual (for `add_family`) or the indicated family to the individual (for `add_individual`).

Role must be one of "CHIL", "HUSB", or "WIFE". Creates the back link (between "FAMC" and "CHIL", "FAMS" and "HUSB" or "WIFE") as well. For "CHIL", `position` says where in the family's children to put the child. If omitted, the new child is added to the end. For "HUSB" and "WIFE", `position` says where in the individual to place the new family link. If omitted, the new family is added to the end.

QUESTION: What should be done about individuals with two or more "FAMC" links (e.g., someone who was adopted and both families are known)? Should there be two `position` arguments – one each for the family and the individual?

`NODE create_record` (*type*) → *RECORD* [Function]
Takes the current node and makes a record of the specified type with the current node as the top node. The created record is returned.

NOTE: The current node must be a temp node and have no parent (i.e., it must be the top of its node tree) and have no siblings. There would probably be other restrictions, yet to be determined, as well.

`RECORD sync` (*void*) → *BOOLEAN* [Function]
Takes the current record and, if it has changed since it was last written, writes it out to the database. On success, it returns True.

The following functions are not instance methods.

`create_reference` (*tag, RECORD*) → *NODE* [Function]
Creates and returns a NODE having the specified `tag` that references `RECORD`.

NOTE: To protect database integrity, tags "CHIL", "HUSB", "WIFE", "FAMS", and "FAMC" cannot have references created by this function. Instead, see the functions `add_family` and `add_individual`.

`find_record` (*search, type*) → *Collection of some flavor* [Function]
Searches records of type `type` for `search`. For "INDI" records, would search the `NAME`. For "FAM" records, would probably search the `NAME` fields of the spouses or

potentially the children. For other record types, would need to figure out what it would search.

Need to have a way to search by `REFN`. And possibly other values such as keynums or keys or dates or locations.

Might be nice to have a way to search for all records and/or nodes that reference a particular record.

Definitely needs more thought as to what it should look like – if it is to ever get implemented.

Probably the best answer is to not implement it in C and instead let people implement whatever they want in Python.

`find_references` (*record, options*) → *Collection of some flavor* [Function]
Given a `record`, return the collection of { NODEs | RECORDs } that reference that record. Whether it is `NODEs` or `RECORDs` might be determined by `options`. What sort of "collection" also needs to be decided.

A.2 Other GEDCOM Records

There are a bunch of GEDCOM records that Lifelines either does not support or does not support that well. Some of these are because they were added long after Lifelines was written. Others because the program's author changed jobs.

QUESTIONS:

What support should there be for records of types

- REPO
- SOUR
- SUBM
- OBJE
- SNOTE

Iterators? Anything else?

A.3 Database Related

Unless explicitly stated otherwise, none of the following change which database is the **current** database. Unless stated otherwise, functions described that do not take a database argument act upon data within the current database.

One of the nebulous future goals is to allow multiple concurrent databases to be in memory at once. What that should look like is unknown. How to resolve issues such as how to { close | drop | unmount | whatever } a database is unknown.

`database_open` (*pathname*, [*access*], [*cache_sizes*], [*locking*]) → *DataBase* [Function]
Attempts to open `pathname` as a LifeLines database. If successful, the database object is returned.

Does not change current database.

QUESTIONS: Do we want to support closing a database? If yes, how do we invalidate any existing Python objects that represent records or nodes within the database? Or do we adopt the attitude that they are still valid but the database is now read-only? Something else? Similar questions for PVALUEs and the existing report writing language.

Opening and closing a database feels like the wrong paradigm. But, what is the right paradigm?

`create_database` (*pathname*) → *DataBase* [Function]

Creates a new empty database at the specified path. Returns the newly created empty database.

Does not change current database.

`DataBase import` (*gedcom_file*, [*keys* = *keys_keep*|*keys_new*|*keys_auto*]) → *Something* [Function]

Imports the specified GEDCOM file. The integer argument `keys`, if specified, must have one of three values.

If `keys` is `keys_new`, then existing keys are ignored and records have new keys.

If `keys` is `keys_keep`, then if existing keys are compatible and there are no conflicts, existing keys are kept. Otherwise an error occurs

If `keys` is `keys_auto` (default), then if existing keys are compabible and there are no conflicts, existing keys are kept. Otherwise, new keys are used.

Does not change the current database.

QUESTIONS:

Should we support doing this on a non-empty database – possibly as a first step of "merging" two GEDCOM files.

Other possibilities include merging databases... instead of merging GEDCOM files. Something to think about.

What should it return? Possibilities include the current database, some statistics on the import, some statistics on the new database, success or failure indication, something else? Needs thought.

`DataBase export` (*pathname*) → *Something* [Function]

QUESTIONS:

What should this return? Success / failure? Statistics? Something else?

Should we take a file descriptor instead of a pathname? That would be more Pythonic, but how to go between a Python file object and a stdio FILE object?

Should there be an argument saying what to export? Or should that be a different function?

If there is an argument for what to export – whether this or a separate function – what to do about references to records outside the set? VOID pointers? Closure?

`DataBase merge` (*database*) → *DataBase* [Function]

Merge the specified database into our database. The records and nodes of the database merged in become part of this database.

QUESTION: Is old database closed? Discarded? Something else?

Probably cannot support his until whole database layer is reworked and btree layer is eliminated.

`DataBase statistics` (*void*) → *tuple* [Function]

Tentative interface: Returns a tuple of tuples of two elements. Each two element tuple is of the form

```
(string integer)
```

where the `string` is one of the known record types and the integer is how many records of that type are present in the database.

Another possibility is to return a dictionary, the strings, as described above, are the keys, the integers are the values.

XXX Think more about what this interface should be. XXX

`current_database` (*void*) → *DataBase* [Function]

Returns an object that represents the current database.

Long term goal: allow multiple concurrent databases in use.

`DataBase set_current` (*void*) → *DataBase* [Function]

Makes this database the current database. Returns the previous database. Returns None if there is no previous database.

`DataBase pathname` (*void*) → *STRING* [Function]

Returns the pathname of the database.

`DataBase cache_sizes` (*void*) → *tuple* [Function]

Returns a tuple of tuples of two elements. For each two element tuple, the first element is a string representing the cache's name, and the second element is an integer representing its size.

QUESTION: Should cache sizes be database specific? Or at they shared? Should the caches even exist? Long term, I'm leaning towards eliminating them altogether.

Appendix B Possible Future Directions

B.1 User Interface

I would like for there to be a better separation between functionality and user interface. Functions which interact with the user, whether by asking a question, printing a message, or drawing something on the screen being totally separated from those that provide the underlying functionality. Different files in different directories.

I would like for there to be a variant of `llines` – possibly `llexec` or possibly a new program that has a fully functional non-curses command line interface.

I would also like there to be a GUI variant .

B.2 Disk Storage

I would like the disk storage to be a GEDCOM file. Or, if not strictly GEDCOM, then – at least – textual and hierarchical like GEDCOM. That is, it might – like the current Lifelines – allow extensions (e.g., event records) that make it not necessarily strictly GEDCOM conforming. But, structured like GEDCOM and if you didn't use the extensions, then it would be GEDCOM.

Other options include formats such as XML and JSON. If such a format was chosen as the native format, then GEDCOM would remain an import and export option.

B.3 Memory Usage

Computer memory has in the past two decades become significantly cheapter. And significantly larger. Many machines ship with more memory than the maximum supported decades ago. So...

Keep the whole database in memory all the time.

If the entire database is in memory, some of the fields within the NODE and RECORD structures (e.g., for starters, the cache pointers) can be eliminated making the structures smaller, reducing the memory cost of keeping the whole database in memory.

B.4 Multiple Databases

Allow multiple databases to be present in memory simultaneously – for example, a research database and a confirmed (need a better word) database. Or my database and a sister-in-laws database. Or a complete database and a shareable (i.e., private information removed) database. Or...

B.5 Additional Record Types

Better – not sure what that really means here – support for additional record types – e.g., Sources, Repositories, Shared Notes, Multimedia, and Submitters.

Possibly allowing the user to define their own record types. Not sure what this would look like...

B.6 Support for Non-GEDCOM Formats

Possibly support some non GEDCOM format(s) such as XML or JSON in addition to GEDCOM. Especially if there are any "popular" (whatever that means here) programs with a non-proprietary non-GEDCOM textual format.

B.7 Support for Export Restrictions

Not sure if the GEDCOM `RESN` tag is adequate here.

Some way to select when exporting data whether to export everything, all the GEDCOM conforming (i.e., no extensions) data, or a subset based on RECORD, EVENT, and/or NODE markers such as the `RESN` tag.

This idea needs more thought.

B.8 configure

The configure files and make files used by Lifelines could stand **significant** improvement. Dependencies are NOT properly tracked.

Function Index

L

M

N

O

P

Q

S

T

U

V

W

X

Y