Lifelines Python Extension Internals

David Cole Taylor

25 October 2021

Table of Contents

1 Lifelines Internals

1.1 Overview

This document concerns itself with the needs of those that wish to modify Lifelines – whether to extend the Python embedding or fix a bug in it – or to modify the core Lifelines.

If your goal is to use the Python embedding, please consult the document "Lifelines Python Extension". If your goal is to use the Lifelines Report Writing language, please consult the document "Lifelines Programming Subsystem and Report Generator".

Much of the Python internals knowledge needed is covered by the documents "Extending and Embedding Python" and "The Python/C API" which are part of the standard Python distribution. They were consulted heavily during the implementation of the Lifelines Python extension.

The other large knowledge base needed is knowledge of Lifelines internals. When I started the Python embedding project, there was a dearth of information available on Lifelines internals. This document is an attempt to partially address that absence.

While some of the information presented is specific to the Python embedding, most should be useful to anyone wishing to modify Lifelines. That said, the functions, variables, and macros documented are those that I encountered during the implementation of the Python embedding.

There are many many gaps here. If a knowledge or understanding of something was not needed as part of the Python embedding project, you will probably not find it here unless it was added at a later date. For example, there was no need to study the lexing and parsing of the Lifelines report writing language, so you won't find anything about that here. Nor, was there much need to study the disk layout of the Lifelines database – beyond high level functions for activities such as importing, exporting, and related activities.

This document is meant to supplement, not replace, the "Lifelines Developer Documentation" document.

If is my hope that this document will grow over time. Feedback, additions, and corrections are welcomed and encouraged.

1.2 Layout

There are a number of subsystems each in its own subdirectory:

- `arch`

  I do not know what this is.

- `btree`

  The code that implements the reading and the writing of the b-tree disk storage.

- `gedlib`

  Most of the GEDCOM related code lives here.

- `interp`

  Primarily, the code that implements the lexing, the parsing, and the user callable functions of the Lifelines Report writing language.

- `liflines`

  The code that deals with the screen (via `ncurses`), the various menus, the user interaction, and various routines that do not have a better location to live.
- `python`

  Support for the Python embedding.
- `stdlib`

  Not a subsystem, as such, but rather a collection of routines that are meant to be generically useful across subsystems. Some of the routines are implementations of functionality that is available on many systems, but not everywhere.
- `ui`

  User interface, obviously. Less obviously, the code here is used by `llexec` and not by `llines`. Some of the routines here are mere stubs of their more fleshed out cousins that live in `liflines`. Stubs that should be, in my opinion, fleshed out.
- `tools`

  Not a subsystem, but rather tools that can be helpful if you get into trouble – such as a corrupted database.
- `hdrs`

  Not a subsystem (obviously), but rather header files of macros and function declarations that are meant to be shared between subsystems.

1.3 Python Files

For the Python embedding, the emphasis is on RECORDs, not NODEs. With the exception of the header file `llpy-externs.h`, which lives in directory `src/hdrs`, all of the source files live in the directory `src/python`.

- `Makefile.am`

  Simple makefile saying which files belong to the Python extension and the name of the library that contains the compiled files.
- `Makefile.in`

  Boring. Generated by automake. **DO NOT EDIT.**
- `database.c`, `database.h`

  Contains database centric routines such as `firstindi`, `lastindi`, `firstfam`, and `lastfam`.
- `event.c`

  Event (NODE, not RECORD) and date specific routines are found here.
- `family.c`, `family.h`

  Most everythiing that deals with FAM RECORDs.
- `iter.c`, `iter.h`

  Iterators for individual, family, source, event, and other RECORDs are found here.
- `nodes.c`
- `other.c`

  Allocation, deallocation, and iteration of OTHR RECORDs.

- `person.c`, `person.h`

  Most everything that deals with INDI RECORDs. Might ought to be renamed to individual.[ch].

- `python-to-c.c`, `python-to-c.h`

  Contains PyInit_llines, llpy_init, and a handful of functions and variables that don't really fit anywhere else.

- `records.c`

  Functions which are RECORD type agnostic, for example the functions `key_to_record` and `keynum_to_record` are found here.

- `run.c`

  Functions for running the Python interpreter – whetther interactively or on a file are found here.

- `set.c`, `set.h`

  Non-instance functions that take or return sets.

- `source.c`

  Allocation, deallocation, and iteration of SOUR RECORDs.

- `types.c`, `types.h`

  Rich compare functions for records and nodes. Hash function for records.

- `user.c`

  Those user interaction functions which are not instance methods can be found in here.

1.4 NODEs And RECORDs

The Lifelines Report Writing language primarily deals with NODES. By contrast, the Python Extension primarily deals with RECORDS.

Both NODEs and RECORDs are refernce counted. When creating a reference to either a NODE or a RECORD it is important to ensure that its reference count is incremented. And when destroying a reference, that it is decremented.

NOTE: The NODE reference counts **DO NOT** include references from other NODEs, such as the parent, sibling, and child links. Nor the link from a RECORD to its top NODE. The reference counts **DO** count the number of `PVALUE`s and `PyObject`s that point to the NODE.

Additionally, NODEs have a flags element. Currently, only one flag bit is defined – whether the NODE is a temporary NODE or not. A tempoary NODE is a NODE that is not part of the NODE tree of any RECORD.

1.5 Application Programmer Interfaces

The lists of functions, macros, and variables in the following chapters are **very** incomplete. They are largely the result of investigations made to facilitate adding Python as a scripting language. Additions and corrections are both encouraged and welcomed.

2 GEDLIB Subsystem

These entries are found in the `src/gedlib` directory.

2.1 NODE Functions

2.1.1 Tree Related NODE Functions

`nchild` (*node*) → *NODE* [Macro]
Given a NODE `node`, return its first child.
NOTE: This is "child" in the tree sense, **NOT** the genealogical sense.

`nsibling` (*node*) → *NODE* [Macro]
Given a NODE `node`, return its first sibling.
NOTE: This is "sibling" in the tree sense, **NOT** the genealogical sense.

`nparent` (*node*) → *NODE* [Macro]
Given a NODE `node`, return its parent.
NOTE: This is "parent" in the tree sense, **NOT** the genealogical sense.

`NODE find_tag` (*NODE node, CNSTRING str*) → [Function]
Given a NODE `node`, check its tag and that of each of its siblings until either the tag `str` is found or we run out of siblings. If we find it, we return a pointer to the node, otherwise we return NULL.

`NODE copy_nodes` (*NODE node, BOOLEAN kids, BOOLEANS sibs*) [Function]
A copy of `node` is made and returned. If `kids` is true, then the copy has copies of the children as well. If `sibs` is true, then the copy has copies of the siblings as well.

`NODE copy_node_subtree` (*NODE node*) [Function]
Returns a copy of the tree rooted at `node`.

`NODE create_temp_node` (*STRING xref, STRING tag, STRING value, NODE parent*) [Function]
Creates a new node with the specified `xref`, `tag`, `value`, and `parent`. Sets the `ND_TEMP` bit in the flags, and returns the new node.

`BOOLEAN equal_tree` (*NODE root1, NODE root2*) [Function]
Compares the tags and values of each node of the trees rooted at `root1` and `root2`. Returns true if they are all equal, false otherwise.

`void free_nodes` (*NODE node*) [Function]
Frees `node`, all of its descendants, all of its siblings, and all of their descendants. Does **NOT** check reference counts.

`BOOLEAN is_temp_node` (*NODE node*) [Function]
Tests the `ND_TEMP` bit of `node`'s flags. If set, it return true, otherwise it returns false.

`void set_temp_node` (*NODE node, BOOLEAN temp*) [Function]
If `temp` is true, sets `node`'s `ND_TEMP` flag. Otherwise clear the flag.

`INT length_nodes` (*NODE node*) [Function]
Returns the length of the sibling chain of `node`.

`NODE string_to_node` (*STRING str*) [Function]
Reads tree from `str`. Modifies `str` – adds NULs between lines. Might make other changes to `str`.

2.1.2 GEDCOM NODE Functions

`NODE fam_to_first_chil` (*NODE node*) → *top NODE of first child* [Function]
Given a NODE `node` for a family, returns the top NODE of the first child. If there are no children, NULL is returned.

`NODE fam_to_last_chil` (*NODE node*) → *top NODE of last child* [Function]
Given a NODE `node` for a family, returns the top NODE of the last child. If there are no children, NULL is returned.

`NODE FAMS` (*NODE indi*) → *NODE* [Function]
Given an individual `indi`, return the tree pointed to by the first `FAMS` NODE. Returns NULL if there are none.

`NODE indi_to_famc` (*NODE indi*) → *NODE* [Function]
Returns the top NODE of the tree pointed to by the first `FAMC` node of `indi`.

`NODE indi_to_fath` (*NODE indi*) → *NODE* [Function]
Returns the top-node of the first `HUSB` node of the family represented by the first `FAMC` node of this individual.

Assumes that `indi` is the top-node of an INDI RECORD.

`NODE indi_to_moth` (*NODE*) → *NODE* [Function]
Returns the top-node of the first `HUSB` node of the family represented by the first `FAMC` node of this individual.

Assumes that `indi` is the top-node of an INDI RECORD.

`NODE indi_to_next_sib_old` (*NODE*) [Function]

`NODE indi_to_prev_sib_old` (*NODE*) [Function]

`NODE qkey_to_fam` (*CNSTRING key*) [Function]

`NODE qkey_to_indi` (*CNSTRING key*) [Function]

`void replace_indi` (*NODE cur_indi, NODE new_indi*) [Function]
Replaces a person, `cur_indi`, in database with a modified version, `new_indi`. Replaces all chilkdren nodes of `cur_indi` with children nodes of `new_indi`. Consumes `new_indi` (calls `free_node` on it).

`INT resolve_refn_links` (*NODE node*) [Function]
Check and resolve all links in node tree. This convets something like `"<1850.Census>"` to something like `"@S25@"`. Returns the number of unresolved links.

`BOOLEAN valid_indi_tree` (*NODE indi, STRING *pmsg, NODE orig*) [Function]

Validates the INDI tree having top-node `indi`. If `orig` is non-NULL, it is the original version of the individual. If problems are found, false is returned and `pmsg` is set to a message indicating what problem was found. Otherwise, true is returned.

`INT val_to_sex` (*NODE node*) [Function]

Converts `node`'s SEX value to one of the three integer constants `SEX_UNKNOWN`, `SEX_MALE`, or `SEX_FEMALE`. Assumes `node`'s tag is `SEX`.

2.1.3 INDISEQ NODE Functions

`INDISEQ fam_to_children` (*NODE fam*) [Function]

Create a sequence of `fam`'s children.

`INDISEQ fam_to_spouses` (*NODE fam*) [Function]

Creates a sequence of spouses of `fam`.

`INDISEQ indi_to_children` (*NODE indi*) [Function]

Creates a sequence of `indi`'s children. Filters out duplicates.

`INDISEQ indi_to_families` (*NODE indi, BOOLEAN fams*) [Function]

Creates a sequence of `indi`'s families. Values will be input person keynum (this is the ISPRN_FAMSEQ style). If `fams` is true, find spousal families, else find parental families.

`INDISEQ indi_to_spouses` (*NODE indi*) [Function]

Creates a sequence of `indi`'s spouses. Values will be family keynums. This sequence will be ISPRN_SPOUSESEQ style (whatever that is), meaning that marriage dates are included with names in print strings.

`void remove_indiseq` (*INDISEQ seq*) [Function]

Frees `seq` and all the memory used by it.

2.2 NODE to RECORD traversal

`RECORD node_to_record` (*NODE node*) → *RECORD* [Function]

Returns the RECORD that contains NODE `node`.

2.3 RECORD Functions

2.3.1 Information about a RECORD

`NODE nztop` (*RECORD record*) [Function]

Returns the top node of `record`.

`CNSTRING nzkey` (*RECORD record*) [Function]

Returns `record`'s key.

`INT nzkeynum` (*RECORD record*) [Function]

Returns `record`'s keynum.

`char nztype` (*RECORD record*) [Function]
Returns `record`'s type.

2.3.2 Raw RECORDS

`STRING retrieve_raw_record` (*CNSTRING key, INT *plen*) [Function]
Retrieve RECORD string from database. Returns the raw record as a string. `key` is the key of the desired record. And `plen` is used to return the length of the record that is being returned.

2.3.3 RECORD Reference Counts

`void release_record` (*RECORD record*) [Function]
Decrement reference count of `record` and freee the RECORD if the reference count reaches zero.

2.4 NAME Related

`CNSTRING getsxsurname` (*CNSTRING*) [Function]
Returns surname for use by soundex routines. Uses a static buffer. And returns `____` if the first non-white space character of the surname is not a letter.

`CNSTRING getasurname` (*CNSTRING*) [Function]
Returns a surname. Uses a static buffer.

`CNSTRING givens` (*CNSTRING name*) [Function]
Returns the given names of `name`. Uses a static buffer.

`CNSTRING trad_soundex` (*CNSTRING*) [Function]
Computes and returns the traditional soundex for the given surname.

`STRING manip_name` (*STRING name, SURCAPTYPE captype, SURORDER surorder, INT len*) [Function]
Manipulates the `name` according to the arguments.

Argument `captype` says whether to return the surname in all caps or as found in `name`.

Argument `surorder` says whether the preserve the order in `name` or to reorder as surname comma rest-of-name.

Argument `len` says the maximum length for the returned result. Regardless of the value of `len`, the surname will not be truncated and at least the first initial will be included in the result.

`STRING trim_name` (*STRING name, INT len*) [Function]
Trim GEDCOM name, `name` to less than or equal to given length, `len`, but no shorter than first initial and surname.

2.5 DataBase Functions and Variables

2.5.1 Database Traversal Functions

`INT xref_{next|prev}{i|f|s|e|x}` (*INT keynum*) → *INT* [Function]
Given { an individual | a family | a source | an event | an other } RECORD's keynum, return the keynum of the { next | previous } { individual | family | source | event | other } RECORD in keynum order.

Keynums are never negative. Zero is reserved for NOT FOUND / DOES NOT EXIST.

`INT xref_{next|prev}` (*char ntype, int keynum*) → *INT* [Function]
`ntype` is one of `'I'`, `'F'`, `'S'`, `'E'`, or `'X'`. This just calls the appropriate xref_{next|prev}{i|f|s|e|x} function and returns the result.

`INT xref_{first|last}{i|f|s|e|x}` (*void*) → *INT* [Function]
Return the keynum of the { first | last } {individual | family | source | event | other } in the database.

2.6 Database Variables

`readonly` [Variable]
When set, the current database was opened in read-only mode and changes are not premitted.

2.7 Keys and Keynums

`RECORD qkey_to_irecord` (*CNSTRING key*) [Function]
`RECORD qkey_to_frecord` (*CNSTRING key*) [Function]
`RECORD qkey_to_srecord` (*CNSTRING key*) [Function]
`RECORD qkey_to_erecord` (*CNSTRING key*) [Function]
`RECORD qkey_to_orecord` (*CNSTRING key*) [Function]
Looks for a RECORD of the given type having the specified key. On success, returns an addref'd record. On failure, returns NULL.

`RECORD keynum_to_frecord` (*int keynum*) [Function]
`RECORD keynum_to_irecord` (*int keynum*) [Function]
`RECORD keynum_to_record` (*char ntype, int keynum*) [Function]
Looks for a RECORD of the specified type having the specifed keynum. If found, returns the RECORD. Otherwise returns NULL.

`RECORD key_to_irecord` (*CNSTRING*) [Function]

2.8 Date, Place, and Event Functions

`INT date_get_year` (*GDATEVAL gdv*) [Function]

`STRING date_get_year_string` (*GDATEVAL gdv*) [Function]

`STRING event_to_date` (*NODE node, BOOLEAN shrt*) [Function]
Convert an event to a date, where `node` is the event node and the short form is used if `shrt` is true.

`GDATEVAL extract_date` (*STRING str*) [Function]
Extracts date from the free format string `str`, returns a parsed date in a newly allocated GDATEVAL.

`void free_gdateval` (*GDATEVAL gdb*) [Function]
Free a GDATEVAL and all the memory allocated to it.

`STRING shorten_date` (*STRING date*) [Function]
Returns a short form of `date`. Uses a static buffer.

`STRING event_to_plac` (*NODE node, BOOLEAN shrt*) [Function]
Convert an event to a place, where `node` is the event node and the short form is used if `shrt` is true.

`STRING event_to_string` (*NODE node, RFMT rfmt*) [Function]
Convert an event to a string. Finds `DATE` and `PLAC` nodes within `node`'s tree and prints a string representation of them. `rfmt` is reformatting info (may be NULL).

`STRING shorten_plac` (*STRING plac*) [Function]
Return short form of `plac`. Returns modified input string or value from `placabbr` table.

2.9 Miscellaneous Functions

2.9.1 Lifelines Options

`INT getlloptint` (*CNSTRING optname, INT defval*) → *INT* [Function]
Looks up the integer option `optname`. If found, its value is returned, otherwise `defval` is returned.

`STRING getlloptstr` (*CNSTRING optname, STRING defval*) → *STRING* [Function]
Looks up the string option `optname`. If found, its value is returned, otherwise `defval` is returned.

2.9.2 String Functions

`STRING rmvat` (*CNSTRING str*) [Function]
Remove `@`'s from both ends of `str`. Returns a static buffer.

`STRING name_string` (*STRING name*) [Function]
Removes slashes from a GEDCOM name. Returns a static buffer.

2.10 Variables and Constants

2.10.1 Messages and Other Strings

To facilitate translation, many – but not all – of the strings printed by Lifelines were collected into one file – `gedlib/messages.c`. Some of these strings represent warning and error messages, some are menu choices, and others are words and phrases, such as "born", "died", and "married" that are used in various contexts.

All of the "messages" (there are over 600 of them) found in `gedlib/messages.c`, such as:

`qSaskstr` [Message]

`qSchoostrttl` [Message]

`qSifonei` [Message]

`qSnotonei` [Message]

have variable names that start with `qS`.

2.10.2 Miscellaneous Variables

`BOOLEAN uu8` [Variable]

Flag indicating if internal codeset is UTF-8 or not.

3 Interpreter Subsystem

These entries are found in the `src/interp` directory.

3.1 User Interface Wrappers

The functions in this section are used by the reporting writing functions to wrap the user interaction so as to be able to give a more accurate timing of how long a script takes to run. There is still some "computation" within the wrapper, but it is reduced.

`rptui_ask_for_fam` (*STRING s1, STRING s2*) → *RECORD* [Function]
Wrapper around `ask_for_fam`.

`rptui_ask_for_indi` (*STRING ttl, ASK1Q ask1*) → *RECORD* [Function]
Wrapper around `ask_for_indi`.

`rptui_ask_for_int` (*STRING ttl, INT *prtn*) → *BOOLEAN* [Function]
Wrapper around `ask_for_int`.

3.2 Miscellaneous Interpreter Functions

`void dolock_node_in_cache` (*NODE node, BOOLEAN lock*) [Function]
Lock or unlock node in cache – if possible.

4 Lifelines Program

These entries are found in the `src/liflines` directory.

4.1 User Interaction

`RECORD ask_for_fam` (*STRING pttl, STRING sttl*) [Function]
Ask user to identify family by spouses. String `pttl` is a prompt to identify a spouse in the family. String `sttl` is a prompt to identify a child in the family.

`RECORD ask_for_indi` (*STRING ttl, ASK1Q ask1*) [Function]
Ask user to identify sequence and select a person. String `ttl` is a title for the question. And `ask1q` says whether to present the list if only one individual matches.

`BOOLEAN ask_for_int` (*STRING ttl, INT *prtn*) [Function]
Ask user to provide an integer. `ttl` is the title prompt. Value is returned in `prtn`. Return value is true if a value is returned. And false if the user canceled the operation.

`BOOLEAN ask_for_string` (*CNSTRING ttl, CNSTRING prompt, STRING buffer, INT buflen*) [Function]
Ask the user for a string. The title of the question (first line) is `ttl` and the prompt of the question (second line) is `prompt`. The response is placed in `buffer` and, if necessary, truncated to `buflen`. The return value indicates whether the user provided an answer (true) or canceled (false).

`RECORD choose_from_indiseq` (*INDISEQ seq, ASK1Q ask1, STRING titl1, STRING titln*) [Function]
Format sequence and have user choose from it (any type). This handles bad pointers. Here, `seq` is the sequence from which to choose, `ask1` is whether to prompt if only one element is in the sequence, `titl1` is the title if sequence only has one element, and `titln` is the title if sequence has multiple elements.

5 STDLIB Subsystem

These entries are found in the `src/stdlib` directory.

`STRING get_lifelines_version` (*INT maxlen*) [Function]
: Returns Lifelines version string using a static buffer. Truncates result to smaller of buffer size or `maxlen`.

`FILE* fopenpath` (*CNSTRING name, STRING mode, STRING path, STRING ext, INT utf8, STRING *pfname*) [Function]
: Search the directories in `path` for `name`, using extension `ext` if necessary. The file is opened with mode `mode`. The pathname opened is returned in `pfname`.

  WARNING: If the file is not found and iis being opened in a mode other than read-only and `name` is not absolute nor `./something`, then it will be created in the first directory of `path` with `ext` appended.

6 Python Subsystem

These entries are found in the `src/python` directory.

6.1 Initialization Functions

`void llpy_database_init` (*void*) [Function]
`void llpy_event_init` (*void*) [Function]
`void llpy_iter_init` (*void*) [Function]
`void llpy_nodes_init` (*void*) [Function]
`void llpy_person_init` (*void*) [Function]
`void llpy_records_init` (*void*) [Function]
`void llpy_set_init` (*void*) [Function]
`void llpy_user_init` (*void*) [Function]

These are the initialization routines for the respective files.

6.2 Dellocation Functions

`void llpy_event_dealloc` (*PyObject *self*) [Function]

`void llpy_family_dealloc` (*PyObject *self*) [Function]

`void llpy_individual_dealloc` (*PyObject *self*) [Function]

`void llpy_iter_dealloc` (*PyObject *self*) [Function]

`void llpy_node_dealloc` (*PyObject *self*) [Function]

`void llpy_nodeiter_dealloc` (*PyObject *self*) [Function]

`void llpy_other_dealloc` (*PyObject *self*) [Function]

`void llpy_source_dealloc` (*PyObject *self*) [Function]

6.3 Iterator Functions

`LLINES_PY_ITER* llpy_events` (*PyObject *self, PyObject *args*) [Function]

Implementation of the `llines` module function `events`. Returns an iterator, an instance of `llines.Iter`, for EVEN RECORDS.

`LLINES_PY_ITER* llpy_families` (*PyObject *self, PyObject *args*) [Function]

Implementation of the `llines` module function `families`. Returns an iterator, an instance of `llines.Iter`, for FAM RECORDS.

`LLINES_PY_ITER* llpy_individuals` (*PyObject *self, PyObject *args*) [Function]

Implementation of the `llines` module function `individuals`. Returns an iterator, an instance of `llines.Iter`, for INDI RECORDS.

`LLINES_PY_ITER* llpy_others` (*PyObject *self, PyObject *args*) [Function]

Implementation of the `llines` module function `others`. Returns an iterator, an instance of `llines.Iter`, for OTHR RECORDS.

`LLINES_PY_ITER* llpy_sources` (*PyObject *self, PyObject *args*) [Function]
Implementation of the `llines` module function `sources`. Returns an iterator, an instance of `llines.Iter`, for SOUR RECORDS.

`LLINES_PY_ITER* llpy_iter_iter` (*PyObject *self*) [Function]
This is the `tp_iter` function for the `llines.Iter` type. When invoked it simply returns its argument.

`PyObject* llpy_iter_iternext` (*PyObject *self*) [Function]
This is the `tp_iternext` function for the `llines.Iter` type. If the iterator was previously exhaused, it sets exception and returns NULL. Otherwise it calls `xref_next` to get the keynum of the next RECORD of the specified type. If the iterator is exhausted, it returns NULL, otherwise it looks up the RECORD and returns a pointer to a Python Object that points to the RECORD.

`PyObject* llpy_nodeiter` (*PyObject *self, PyObject *args, PyObject *kw*) [Function]
Implementation of the `llines` module function `nodeiter`.

`PyObject* llpy_nodeiter_iter` (*PyObject *self*) [Function]
This is the `tp_iter` function for the `llines.NodeIter` type. When invoked it simply returns its argument.

`PyObject* llpy_nodeiter_iternext` (*PyObject *self*) [Function]
Iterates on a node tree in accordance with the arguments supplied when the iterator was created.

6.4 Types

Types can be approached from two perspectives – the C perspective and the Python perspective.

For a C perspective, there are four types defined by the Python extension:

- LLINES_PY_NODE

  This contains two Lifelines specific fields – `lln_node`, a pointer to a NODE and, when known, `lln_type` records the type of the RECORD that contains the NODE. When the record type is unknown, which is often, the `lln_type` field is initialized to zero. The `lln_type` field might be eliminated in the future.

- LLINES_PY_RECORD

  Like LLINES_PY_NODE, this contains two Lifelines specific fields – `llr_record` and `llr_type`. The field `llr_record` points to a RECORD. And the field `llr_type` contains the type of the RECORD. The type is **always** known.

- LLINES_PY_ITER

  This contains two Lifelines specific fields – `li_type` and `li_current`. The `li_type` records the type of RECORD being iterated over. And the `li_current` records the keynum of the most recently returned RECORD. If no iteration has occurred, the value is 0. If it has been exhaused, the value is -1.

- LLINES_PY_NODEITER
  This is used for iterating over NODES. And there is more than one type of NODE iteration. As a result, this type is more complex and has five Lifelines specific fields:
  - `ni_top_node`
    This is the top of the NODE tree being iterated over.
  - `ni_cur_node`
    This is the most currently returned NODE. If iteration has not started, this is NULL.
  - `ni_tag`
    When iterating, if we are looking for a specific tag, this is that tag. Otherwise this is NULL. When it is non-NULL, only nodes having a matching tag will be returned.
  - `ni_type`
    This is the type of NODE iteration to perform: There is child iteration, where we iterate over the immediate children of the top node. And there is whole tree iteration, where we iterate over all the nodes in the tree. When doing whole tree iteration, we do depth first, parent before child.
  - `ni_level`
    This records how far we are from the top node of the tree being iterated over. For whole tree iteration it is part of the return value. Before we start it will be 0. After exhaustion it will be -1.

From a Python perspective, the types are:

- `llines.Database`
- `llines.Event`
- `llines.Family`
- `llines.Individual`
- `llines.Iter`
- `llines.NodeIter`
- `llines.Node`
- `llines.Other`
- `llines.Source`

6.5 Variables

The following arrays define instance methods of their datatype:

`PyMethodDef Lifelines_Event_Methods` [Variable]

`PyMethodDef Lifelines_Family_Methods` [Variable]

`PyMethodDef Lifelines_Iter_Methods` [Variable]

`PyMethodDef Lifelines_Other_Methods` [Variable]

`PyMethodDef Lifelines_Person_Methods` [Variable]

`PyMethodDef Lifelines_Source_Methods` [Variable]

`PyMethodDef Liflines_Node_Methods` [Variable]

The following arrays define additional `llines` module functions:

`PyMethodDef Lifelines_Database_Functions` [Variable]

`PyMethodDef Lifelines_Date_Functions` [Variable]

`PyMethodDef Liflines_Node_Functions` [Variable]

`PyMethodDef Lifelines_Person_Functions` [Variable]

`PyMethodDef Lifelines_Records_Functions` [Variable]

`PyMethodDef Lifelines_Set_Functions` [Variable]

`PyMethodDef Lifelines_User_Functions` [Variable]

The following variables define the `llines` module types that are added to Python by the Lifelines extension:

`PyTypeObject llines_database_type` [Variable]
Definition of Python type `llines.Database`.

`PyTypeObject llines_event_type` [Variable]
Definition of Python type `llines.Event`.

`PyTypeObject llines_family_type` [Variable]
Definition of Python type `llines.Family`.

`PyTypeObject llines_individual_type` [Variable]
Definition of Python type `llines.Individual`.

`PyTypeObject llines_iter_type` [Variable]
Definition of Python type `llines.Iter`.

`PyTypeObject llines_node_type` [Variable]
Definition of Python type `llines.Node`.

`PyTypeObject llines_nodeiter_type` [Variable]
Definition of Python type `llines.NodeIter`.

`PyTypeObject llines_other_type` [Variable]
Definition of Python type `llines.Other`.

`PyTypeObject llines_record_type` [Variable]
Definition of Python type `llines.Record`.

`PyTypeObject llines_source_type` [Variable]
Definition of Python type `llines.Source`.

Function, Variable, and Type Index

M

N

Q

R

S

T

U

V

X