notmuch Documentation

Release 0.21 rc2

notmuch contributors

October 19, 2015

Contents

The *notmuch* module provides an interface to the notmuch functionality, directly interfacing to a shared notmuch library. Within *notmuch*, the classes *Database*, *Query* provide most of the core functionality, returning *Threads*, *Messages* and *Tags*.

License This module is covered under the GNU GPL v3 (or later).

CHAPTER 1

Quickstart and examples

Notmuch can be imported as:

```
import notmuch
```

or:

```
from notmuch import Query, Database

db = Database('path', create=True)
msgs = Query(db, 'from:myself').search_messages()

for msg in msgs:
    print(msg)
```

CHAPTER 2

Interfacing with notmuch

The *notmuch* module provides most of the functionality that a user is likely to need.

Note: The underlying notmuch library is build on a hierarchical memory allocator called talloc. All objects derive from a top-level *Database* object.

This means that as soon as an object is deleted, all underlying derived objects such as Queries, Messages, Message, and Tags will be freed by the underlying library as well. Accessing these objects will then lead to segfaults and other unexpected behavior.

We implement reference counting, so that parent objects can be automatically freed when they are not needed anymore. For example:

```
db = Database('path',create=True)
msgs = Query(db,'from:myself').search_messages()
```

This returns *Messages* which internally contains a reference to its parent *Query* object. Otherwise the Query() would be immediately freed, taking our *msgs* down with it.

In this case, the above Query() object will be automatically freed whenever we delete all derived objects, ie in our case: *del(msgs)* would also delete the parent Query. It would not delete the parent Database() though, as that is still referenced from the variable *db* in which it is stored.

Pretty much the same is valid for all other objects in the hierarchy, such as *Query*, *Messages*, *Message*, and *Tags*.

CHAPTER 3

Status and Errors

Some methods return a status, indicating if an operation was successful and what the error was. Most of these status codes are expressed as a specific value, the *notmuch.STATUS*.

Note: Prior to version 0.12 the exception classes and the enumeration *notmuch.STATUS* were defined in *notmuch.globals*. They have since then been moved into *notmuch.errors*.

3.1 STATUS – Notmuch operation return value

class STATUS

STATUS is a class, whose attributes provide constants that serve as return indicators for notmuch functions. Currently the following ones are defined. For possible return values and specific meaning for each method, see the method description.

- SUCCESS
- OUT_OF_MEMORY
- READ_ONLY_DATABASE
- XAPIAN_EXCEPTION
- FILE_ERROR
- FILE_NOT_EMAIL
- DUPLICATE_MESSAGE_ID
- NULL_POINTER
- TAG_TOO_LONG
- UNBALANCED_FREEZE_THAW
- UNBALANCED_ATOMIC
- NOT_INITIALIZED

Invoke the class method *notmuch.STATUS.status2str* with a status value as argument to receive a human readable string

status2str(*status*)

Get a (unicode) string representation of a notmuch_status_t value.

STATUS.**status2str**(*status*)

Get a (unicode) string representation of a notmuch_status_t value.

3.2 NotmuchError – A Notmuch execution error

Whenever an error occurs, we throw a special Exception *NotmuchError*, or a more fine grained Exception which is derived from it. This means it is always safe to check for NotmuchErrors if you want to catch all errors. If you are interested in more fine grained exceptions, you can use those below.

exception NotmuchError(*status=None*, *message=None*)
Is initiated with a (notmuch.STATUS[, message=None]). It will not return an instance of the class NotmuchError, but a derived instance of a more specific Error Message, e.g. OutOfMemoryError. Each status but SUCCESS has a corresponding subclassed Exception.

The following exceptions are all directly derived from NotmuchError. Each of them corresponds to a specific *notmuch.STATUS* value. You can either check the `status` attribute of a NotmuchError to see if a specific error has occurred, or you can directly check for the following Exception types:

exception OutOfMemoryError(*message=None*)

exception ReadOnlyDatabaseError(*message=None*)

exception XapianError(*message=None*)

exception FileError(*message=None*)

exception FileNotEmailError(*message=None*)

exception DuplicateMessageIdError(*message=None*)

exception NullPointerError(*message=None*)

exception TagTooLongError(*message=None*)

exception UnbalancedFreezeThawError(*message=None*)

exception UnbalancedAtomicError(*message=None*)

exception NotInitializedError(*message=None*)
Derived from NotmuchError, this occurs if the underlying data structure (e.g. database is not initialized (yet) or an iterator has been exhausted. You can test for NotmuchError with .status = STATUS.NOT_INITIALIZED

CHAPTER 4

Database – The underlying notmuch database

class Database([*path=None*[, *create=False*[, *mode=MODE.READ_ONLY*]]])

The *Database* is the highest-level object that notmuch provides. It references a notmuch database, and can be opened in read-only or read-write mode. A *Query* can be derived from or be applied to a specific database to find messages. Also adding and removing messages to the database happens via this object. Modifications to the database are not atmic by default (see *begin_atomic()*) and once a database has been modified, all other database objects pointing to the same data-base will throw an *XapianError* as the underlying database has been modified. Close and reopen the database to continue working with it.

Database objects implement the context manager protocol so you can use the `with` statement to ensure that the database is properly closed. See *close()* for more information.

Note: Any function in this class can and will throw an *NotInitializedError* if the database was not intitialized properly.

If *path* is *None*, we will try to read a users notmuch configuration and use his configured database. The location of the configuration file can be specified through the environment variable *NOTMUCH_CONFIG*, falling back to the default *~/.notmuch-config*.

If *create* is *True*, the database will always be created in *MODE*.READ_WRITE mode. Default mode for opening is READ_ONLY.

Parameters

- **path** (*str* or *None*) – Directory to open/create the database in (see above for behavior if *None*)
- **create** (*bool*) – Pass *False* to open an existing, *True* to create a new database.
- **mode** (*MODE*) – Mode to open a database in. Is always *MODE*.READ_WRITE when creating a new one.

Raises *NotmuchError* or derived exception in case of failure.

create(*path*)

Creates a new notmuch database

This function is used by __init__() and usually does not need to be called directly. It wraps the underlying *notmuch_database_create* function and creates a new notmuch database at *path*. It will always return a database in *MODE* .READ_WRITE mode as creating an empty database for reading only does not make a great deal of sense.

Parameters **path** (*str*) – A directory in which we should create the database.

Raises *NotmuchError* in case of any failure (possibly after printing an error message on stderr).

open(*path*, *status=MODE.READ_ONLY*)
Opens an existing database

This function is used by __init__() and usually does not need to be called directly. It wraps the underlying *notmuch_database_open* function.

Parameters **status** (*MODE*) – Open the database in read-only or read-write mode

Raises Raises *NotmuchError* in case of any failure (possibly after printing an error message on stderr).

close()
Closes the notmuch database.

> **Warning:** This function closes the notmuch database. From that point on every method invoked on any object ever derived from the closed database may cease to function and raise a NotmuchError.

get_path()
Returns the file path of an open database

get_version()
Returns the database format version

Returns The database version as positive integer

needs_upgrade()
Does this database need to be upgraded before writing to it?

If this function returns *True* then no functions that modify the database (*add_message()*, *Message.add_tag()*, *Directory.set_mtime()*, etc.) will work unless *upgrade()* is called successfully first.

Returns *True* or *False*

upgrade()
Upgrades the current database

After opening a database in read-write mode, the client should check if an upgrade is needed (notmuch_database_needs_upgrade) and if so, upgrade with this function before making any modifications.

NOT IMPLEMENTED: The optional progress_notify callback can be used by the caller to provide progress indication to the user. If non-NULL it will be called periodically with 'progress' as a floating-point value in the range of [0.0..1.0] indicating the progress made so far in the upgrade process.

TODO catch exceptions, document return values and etc...

begin_atomic()
Begin an atomic database operation

Any modifications performed between a successful *begin_atomic()* and a *end_atomic()* will be applied to the database atomically. Note that, unlike a typical database transaction, this only ensures atomicity, not durability; neither begin nor end necessarily flush modifications to disk.

Returns *STATUS*.SUCCESS or raises

Raises *NotmuchError*: *STATUS*.XAPIAN_EXCEPTION Xapian exception occurred; atomic section not entered.

Added in notmuch 0.9

end_atomic()
Indicate the end of an atomic database operation

See *begin_atomic()* for details.

Returns *STATUS*.SUCCESS or raises

Raises

***NotmuchError*:**

***STATUS*.XAPIAN_EXCEPTION** A Xapian exception occurred; atomic section not ended.

***STATUS*.UNBALANCED_ATOMIC:** end_atomic has been called more times than begin_atomic.

Added in notmuch 0.9

get_directory(*path*)

Returns a *Directory* of path,

Parameters **path** – An unicode string containing the path relative to the path of database (see *get_path()*), or else should be an absolute path with initial components that match the path of 'database'.

Returns *Directory* or raises an exception.

Raises *FileError* if path is not relative database or absolute with initial components same as database.

add_message(*filename*, *sync_maildir_flags=False*)

Adds a new message to the database

Parameters

- **filename** – should be a path relative to the path of the open database (see *get_path()*), or else should be an absolute filename with initial components that match the path of the database.

  The file should be a single mail message (not a multi-message mbox) that is expected to remain at its current location, since the notmuch database will reference the filename, and will not copy the entire contents of the file.

- **sync_maildir_flags** – If the message contains Maildir flags, we will -depending on the notmuch configuration- sync those tags to initial notmuch tags, if set to *True*. It is *False* by default to remain consistent with the libnotmuch API. You might want to look into the underlying method *Message.maildir_flags_to_tags()*.

Returns

On success, we return

1. a *Message* object that can be used for things such as adding tags to the just-added message.
2. one of the following *STATUS* values:

   ***STATUS*.SUCCESS** Message successfully added to database.

   ***STATUS*.DUPLICATE_MESSAGE_ID** Message has the same message ID as another message already in the database. The new filename was successfully added to the list of the filenames for the existing message.

Return type 2-tuple(*Message*, *STATUS*)

Raises Raises a *NotmuchError* with the following meaning. If such an exception occurs, nothing was added to the database.

***STATUS*.FILE_ERROR** An error occurred trying to open the file, (such as permission denied, or file not found, etc.).

***STATUS*.FILE_NOT_EMAIL** The contents of filename don't look like an email message.

***STATUS*.READ_ONLY_DATABASE** Database was opened in read-only mode so no message can be added.

remove_message (*filename*)

Removes a message (filename) from the given notmuch database

Note that only this particular filename association is removed from the database. If the same message (as determined by the message ID) is still available via other filenames, then the message will persist in the database for those filenames. When the last filename is removed for a particular message, the database content for that message will be entirely removed.

Returns

A *STATUS* value with the following meaning:

***STATUS*.SUCCESS** The last filename was removed and the message was removed from the database.

***STATUS*.DUPLICATE_MESSAGE_ID** This filename was removed but the message persists in the database with at least one other filename.

Raises Raises a *NotmuchError* with the following meaning. If such an exception occurs, nothing was removed from the database.

***STATUS*.READ_ONLY_DATABASE** Database was opened in read-only mode so no message can be removed.

find_message (*msgid*)

Returns a *Message* as identified by its message ID

Wraps the underlying *notmuch_database_find_message* function.

Parameters **msgid** (*unicode or str*) – The message ID

Returns *Message* or *None* if no message is found.

Raises

OutOfMemoryError If an Out-of-memory occured while constructing the message.

XapianError In case of a Xapian Exception. These exceptions include "Database modified" situations, e.g. when the notmuch database has been modified by another program in the meantime. In this case, you should close and reopen the database and retry.

NotInitializedError **if** the database was not intitialized.

find_message_by_filename (*filename*)

Find a message with the given filename

Returns If the database contains a message with the given filename, then a class:*Message:* is returned. This function returns None if no message is found with the given filename.

Raises *OutOfMemoryError* if an Out-of-memory occured while constructing the message.

Raises *XapianError* in case of a Xapian Exception. These exceptions include "Database modified" situations, e.g. when the notmuch database has been modified by another program in the meantime. In this case, you should close and reopen the database and retry.

Raises *NotInitializedError* if the database was not intitialized.

Added in notmuch 0.9

get_all_tags()
Returns *Tags* with a list of all tags found in the database

Returns *Tags*

Execption *NotmuchError* with *STATUS*.NULL_POINTER on error

create_query(*querystring*)
Returns a *Query* derived from this database

This is a shorthand method for doing:

```
# short version
# Automatically frees the Database() when 'q' is deleted

q  = Database(dbpath).create_query('from:"Biene Maja"')

# long version, which is functionally equivalent but will keep the
# Database in the 'db' variable around after we delete 'q':

db = Database(dbpath)
q  = Query(db,'from:"Biene Maja"')
```

This function is a python extension and not in the underlying C API.

MODE
Defines constants that are used as the mode in which to open a database.

MODE.READ_ONLY Open the database in read-only mode

MODE.READ_WRITE Open the database in read-write mode

CHAPTER 5

Query – A search query

class Query (*db*, *querystr*)

Represents a search query on an opened *Database*.

A query selects and filters a subset of messages from the notmuch database we derive from.

Query provides an instance attribute *sort*, which contains the sort order (if specified via *set_sort()*) or *None*.

Any function in this class may throw an *NotInitializedError* in case the underlying query object was not set up correctly.

Note: Do remember that as soon as we tear down this object, all underlying derived objects such as threads, messages, tags etc will be freed by the underlying library as well. Accessing these objects will lead to segfaults and other unexpected behavior. See above for more details.

Parameters

- **db** (*Database*) – An open database which we derive the Query from.
- **querystr** (*utf-8 encoded str or unicode*) – The query string for the message.

create (*db*, *querystr*)

Creates a new query derived from a Database

This function is utilized by __init__() and usually does not need to be called directly.

Parameters

- **db** (*Database*) – Database to create the query from.
- **querystr** (*utf-8 encoded str or unicode*) – The query string

Raises

NullPointerError **if the query creation failed** (e.g. too little memory).

NotInitializedError **if the underlying db was not** intitialized.

SORT

Defines constants that are used as the mode in which to open a database.

SORT.OLDEST_FIRST Sort by message date, oldest first.

SORT.NEWEST_FIRST Sort by message date, newest first.

SORT.MESSAGE_ID Sort by email message ID.

SORT.UNSORTED Do not apply a special sort order (returns results in document id order).

set_sort(*sort*)
Set the sort order future results will be delivered in

Parameters **sort** – Sort order (see *Query.SORT*)

sort
Instance attribute *sort* contains the sort order (see *Query.SORT*) if explicitely specified via *set_sort()*. By default it is set to *None*.

exclude_tag(*tagname*)
Add a tag that will be excluded from the query results by default.

This exclusion will be overridden if this tag appears explicitly in the query.

Parameters **tagname** – Name of the tag to be excluded

search_threads()
Execute a query for threads

Execute a query for threads, returning a *Threads* iterator. The returned threads are owned by the query and as such, will only be valid until the Query is deleted.

The method sets *Message.FLAG*.MATCH for those messages that match the query. The method *Message.get_flag()* allows us to get the value of this flag.

Returns *Threads*

Raises *NullPointerError* if search_threads failed

search_messages()
Filter messages according to the query and return *Messages* in the defined sort order

Returns *Messages*

Raises *NullPointerError* if search_messages failed

count_messages()
This function performs a search and returns Xapian's best guess as to the number of matching messages.

Returns the estimated number of messages matching this query

Return type int

count_threads()
This function performs a search and returns the number of unique thread IDs in the matching messages. This is the same as number of threads matching a search.

Note that this is a significantly heavier operation than meth:*Query.count_messages*.

Returns the number of threads returned by this query

Return type int

CHAPTER 6

Messages – A bunch of messages

class Messages (*msgs_p*, *parent=None*)

Represents a list of notmuch messages

This object provides an iterator over a list of notmuch messages (Technically, it provides a wrapper for the underlying *notmuch_messages_t* structure). Do note that the underlying library only provides a one-time iterator (it cannot reset the iterator to the start). Thus iterating over the function will "exhaust" the list of messages, and a subsequent iteration attempt will raise a *NotInitializedError*. If you need to re-iterate over a list of messages you will need to retrieve a new *Messages* object or cache your *Message*s in a list via:

```
msglist = list(msgs)
```

You can store and reuse the single *Message* objects as often as you want as long as you keep the parent *Messages* object around. (Due to hierarchical memory allocation, all derived *Message* objects will be invalid when we delete the parent *Messages* object, even if it was already exhausted.) So this works:

```
db    = Database()
msgs = Query(db,'').search_messages() #get a Messages() object
msglist = list(msgs)

# msgs is "exhausted" now and msgs.next() will raise an exception.
# However it will be kept alive until all retrieved Message()
# objects are also deleted. If you do e.g. an explicit del(msgs)
# here, the following lines would fail.

# You can reiterate over *msglist* however as often as you want.
# It is simply a list with :class:`Message`s.

print (msglist[0].get_filename())
print (msglist[1].get_filename())
print (msglist[0].get_message_id())
```

As *Message* implements both __hash__() and __cmp__(), it is possible to make sets out of *Messages* and use set arithmetic (this happens in python and will of course be *much* slower than redoing a proper query with the appropriate filters:

```
s1, s2 = set(msgs1), set(msgs2)
s.union(s2)
s1 -= s2
...
```

Be careful when using set arithmetic between message sets derived from different Databases (ie the same database reopened after messages have changed). If messages have added or removed associated files in the

meantime, it is possible that the same message would be considered as a different object (as it points to a different file).

Parameters

- **msgs_p** (ctypes.c_void_p) – A pointer to an underlying *notmuch_messages_t* structure. These are not publically exposed, so a user will almost never instantiate a *Messages* object herself. They are usually handed back as a result, e.g. in *Query.search_messages()*. *msgs_p* must be valid, we will raise an *NullPointerError* if it is *None*.
- **parent** – The parent object (ie *Query*) these tags are derived from. It saves a reference to it, so we can automatically delete the db object once all derived objects are dead.

TODO Make the iterator work more than once and cache the tags in the Python object.(?)

collect_tags()

Return the unique *Tags* in the contained messages

Returns *Tags*

Exceptions *NotInitializedError* if not init'ed

Note: *collect_tags()* will iterate over the messages and therefore will not allow further iterations.

__len__()

Warning: *__len__()* was removed in version 0.6 as it exhausted the iterator and broke list(Messages()). Use the *Query.count_messages()* function or use *len(list(msgs))*.

CHAPTER 7

Message – A single message

class Message(*msg_p*, *parent=None*)

Represents a single Email message

Technically, this wraps the underlying *notmuch_message_t* structure. A user will usually not create these objects themselves but get them as search results.

As it implements `__cmp__()`, it is possible to compare two *Message*s using *if msg1 == msg2: ....*

Parameters

- **msg_p** – A pointer to an internal notmuch_message_t Structure. If it is *None*, we will raise an *NullPointerError*.
- **parent** – A 'parent' object is passed which this message is derived from. We save a reference to it, so we can automatically delete the parent object once all derived objects are dead.

get_message_id()

Returns the message ID

Returns String with a message ID

Raises *NotInitializedError* if the message is not initialized.

get_thread_id()

Returns the thread ID

The returned string belongs to 'message' will only be valid for as long as the message is valid.

This function will not return *None* since Notmuch ensures that every message belongs to a single thread.

Returns String with a thread ID

Raises *NotInitializedError* if the message is not initialized.

get_replies()

Gets all direct replies to this message as *Messages* iterator

---

Note: This call only makes sense if 'message' was ultimately obtained from a *Thread* object, (such as by coming directly from the result of calling *Thread.get_toplevel_messages()* or by any number of subsequent calls to *get_replies()*). If this message was obtained through some non-thread means, (such as by a call to *Query.search_messages()*), then this function will return an empty Messages iterator.

Returns *Messages*.

Raises *NotInitializedError* if the message is not initialized.

get_filename()
Returns the file path of the message file

Returns Absolute file path & name of the message file

Raises *NotInitializedError* if the message is not initialized.

get_filenames()
Get all filenames for the email corresponding to 'message'

Returns a Filenames() generator with all absolute filepaths for messages recorded to have the same Message-ID. These files must not necessarily have identical content.

FLAG

FLAG.MATCH This flag is automatically set by a Query.search_threads on those messages that match the query. This allows us to distinguish matches from the rest of the messages in that thread.

get_flag(*flag*)
Checks whether a specific flag is set for this message

The method *Query.search_threads()* sets *Message.FLAG.MATCH* for those messages that match the query. This method allows us to get the value of this flag.

Parameters **flag** – One of the *Message.FLAG* values (currently only *Message.FLAG.MATCH*

Returns An unsigned int (0/1), indicating whether the flag is set.

Raises *NotInitializedError* if the message is not initialized.

set_flag(*flag*, *value*)
Sets/Unsets a specific flag for this message

Parameters

- **flag** – One of the *Message.FLAG* values (currently only *Message.FLAG.MATCH*
- **value** – A bool indicating whether to set or unset the flag.

Raises *NotInitializedError* if the message is not initialized.

get_date()
Returns time_t of the message date

For the original textual representation of the Date header from the message call notmuch_message_get_header() with a header value of "date".

Returns A time_t timestamp.

Return type c_unit64

Raises *NotInitializedError* if the message is not initialized.

get_header(*header*)
Get the value of the specified header.

The value will be read from the actual message file, not from the notmuch database. The header name is case insensitive.

Returns an empty string ("") if the message does not contain a header line matching 'header'.

Parameters **header** (*str*) – The name of the header to be retrieved. It is not case-sensitive.

Returns The header value as string

Raises *NotInitializedError* if the message is not initialized

Raises *NullPointerError* if any error occured

get_tags()
Returns the message tags

Returns A *Tags* iterator.

Raises *NotInitializedError* if the message is not initialized

Raises *NullPointerError* if any error occured

maildir_flags_to_tags()
Synchronize file Maildir flags to notmuch tags

Flag Action if present —- ————— ‘D’ Adds the “draft” tag to the message ‘F’ Adds the “flagged” tag to the message ‘P’ Adds the “passed” tag to the message ‘R’ Adds the “replied” tag to the message ‘S’ Removes the “unread” tag from the message

For each flag that is not present, the opposite action (add/remove) is performed for the corresponding tags. If there are multiple filenames associated with this message, the flag is considered present if it appears in one or more filenames. (That is, the flags from the multiple filenames are combined with the logical OR operator.)

As a convenience, you can set the sync_maildir_flags parameter in *Database.add_message()* to implicitly call this.

Returns a *STATUS*. In short, you want to see notmuch.STATUS.SUCCESS here. See there for details.

tags_to_maildir_flags()
Synchronize notmuch tags to file Maildir flags

‘D’ if the message has the “draft” tag ‘F’ if the message has the “flagged” tag ‘P’ if the message has the “passed” tag ‘R’ if the message has the “replied” tag ‘S’ if the message does not have the “unread” tag

Any existing flags unmentioned in the list above will be preserved in the renaming.

Also, if this filename is in a directory named “new”, rename it to be within the neighboring directory named “cur”.

Do note that calling this method while a message is frozen might not work yet, as the modified tags have not been committed yet to the database.

Returns a *STATUS* value. In short, you want to see notmuch.STATUS.SUCCESS here. See there for details.

remove_tag(*tag*, *sync_maildir_flags=False*)
Removes a tag from the given message

If the message has no such tag, this is a non-operation and will report success anyway.

Parameters

- **tag** – String with a ‘tag’ to be removed.
- **sync_maildir_flags** – If notmuch configuration is set to do this, add maildir flags corresponding to notmuch tags. See underlying method *tags_to_maildir_flags()*. Use False if you want to add/remove many tags on a message without having to physically rename the file every time. Do note, that this will do nothing when a message is frozen, as tag changes will not be committed to the database yet.

Returns STATUS.SUCCESS if the tag was successfully removed or if the message had no such tag. Raises an exception otherwise.

Raises *NullPointerError* if the *tag* argument is NULL

Raises *TagTooLongError* if the length of *tag* exceeds Message.NOTMUCH_TAG_MAX)

Raises *ReadOnlyDatabaseError* if the database was opened in read-only mode so message cannot be modified

Raises *NotInitializedError* if message has not been initialized

add_tag(*tag*, *sync_maildir_flags=False*)

Adds a tag to the given message

Adds a tag to the current message. The maximal tag length is defined in the notmuch library and is currently 200 bytes.

Parameters

- **tag** – String with a 'tag' to be added.
- **sync_maildir_flags** – If notmuch configuration is set to do this, add maildir flags corresponding to notmuch tags. See underlying method *tags_to_maildir_flags()*. Use False if you want to add/remove many tags on a message without having to physically rename the file every time. Do note, that this will do nothing when a message is frozen, as tag changes will not be committed to the database yet.

Returns STATUS.SUCCESS if the tag was successfully added. Raises an exception otherwise.

Raises *NullPointerError* if the *tag* argument is NULL

Raises *TagTooLongError* if the length of *tag* exceeds Message.NOTMUCH_TAG_MAX)

Raises *ReadOnlyDatabaseError* if the database was opened in read-only mode so message cannot be modified

Raises *NotInitializedError* if message has not been initialized

remove_all_tags(*sync_maildir_flags=False*)

Removes all tags from the given message.

See *freeze()* for an example showing how to safely replace tag values.

Parameters **sync_maildir_flags** – If notmuch configuration is set to do this, add maildir flags corresponding to notmuch tags. See *tags_to_maildir_flags()*. Use False if you want to add/remove many tags on a message without having to physically rename the file every time. Do note, that this will do nothing when a message is frozen, as tag changes will not be committed to the database yet.

Returns STATUS.SUCCESS if the tags were successfully removed. Raises an exception otherwise.

Raises *ReadOnlyDatabaseError* if the database was opened in read-only mode so message cannot be modified

Raises *NotInitializedError* if message has not been initialized

freeze()

Freezes the current state of 'message' within the database

This means that changes to the message state, (via *add_tag()*, *remove_tag()*, and *remove_all_tags()*), will not be committed to the database until the message is *thaw()* ed.

Multiple calls to freeze/thaw are valid and these calls will "stack". That is there must be as many calls to thaw as to freeze before a message is actually thawed.

The ability to do freeze/thaw allows for safe transactions to change tag values. For example, explicitly setting a message to have a given set of tags might look like this:

```
msg.freeze()
msg.remove_all_tags(False)
for tag in new_tags:
    msg.add_tag(tag, False)
msg.thaw()
msg.tags_to_maildir_flags()
```

With freeze/thaw used like this, the message in the database is guaranteed to have either the full set of original tag values, or the full set of new tag values, but nothing in between.

Imagine the example above without freeze/thaw and the operation somehow getting interrupted. This could result in the message being left with no tags if the interruption happened after *remove_all_tags()* but before *add_tag()*.

Returns STATUS.SUCCESS if the message was successfully frozen. Raises an exception otherwise.

Raises *ReadOnlyDatabaseError* if the database was opened in read-only mode so message cannot be modified

Raises *NotInitializedError* if message has not been initialized

thaw()

Thaws the current 'message'

Thaw the current 'message', synchronizing any changes that may have occurred while 'message' was frozen into the notmuch database.

See *freeze()* for an example of how to use this function to safely provide tag changes.

Multiple calls to freeze/thaw are valid and these calls with "stack". That is there must be as many calls to thaw as to freeze before a message is actually thawed.

Returns STATUS.SUCCESS if the message was successfully frozen. Raises an exception otherwise.

Raises *UnbalancedFreezeThawError* if an attempt was made to thaw an unfrozen message. That is, there have been an unbalanced number of calls to *freeze()* and *thaw()*.

Raises *NotInitializedError* if message has not been initialized

__str__()

CHAPTER 8

Tags – Notmuch tags

class Tags (*tags_p*, *parent=None*)

Represents a list of notmuch tags

This object provides an iterator over a list of notmuch tags (which are unicode instances).

Do note that the underlying library only provides a one-time iterator (it cannot reset the iterator to the start). Thus iterating over the function will "exhaust" the list of tags, and a subsequent iteration attempt will raise a *NotInitializedError*. Also note, that any function that uses iteration (nearly all) will also exhaust the tags. So both:

```
for tag in tags: print tag
```

as well as:

```
number_of_tags = len(tags)
```

and even a simple:

```
#str() iterates over all tags to construct a space separated list
print(str(tags))
```

will "exhaust" the Tags. If you need to re-iterate over a list of tags you will need to retrieve a new *Tags* object.

Parameters

- **tags_p** (ctypes.c_void_p) – A pointer to an underlying *notmuch_tags_t* structure. These are not publically exposed, so a user will almost never instantiate a *Tags* object herself. They are usually handed back as a result, e.g. in *Database.get_all_tags()*. *tags_p* must be valid, we will raise an *NullPointerError* if it is *None*.
- **parent** – The parent object (ie *Database* or *Message* these tags are derived from, and saves a reference to it, so we can automatically delete the db object once all derived objects are dead.

TODO Make the iterator optionally work more than once by cache the tags in the Python object(?)

__len__()

> **Warning:** *__len__()* was removed in version 0.6 as it exhausted the iterator and broke list(Tags()). Use `len(list(msgs))()` instead if you need to know the number of tags.

__str__()

CHAPTER 9

Threads – Threads iterator

class Threads (*threads_p*, *parent=None*)

Represents a list of notmuch threads

This object provides an iterator over a list of notmuch threads (Technically, it provides a wrapper for the underlying *notmuch_threads_t* structure). Do note that the underlying library only provides a one-time iterator (it cannot reset the iterator to the start). Thus iterating over the function will "exhaust" the list of threads, and a subsequent iteration attempt will raise a *NotInitializedError*. Also note, that any function that uses iteration will also exhaust the messages. So both:

```
for thread in threads: print thread
```

as well as:

```
number_of_msgs = len(threads)
```

will "exhaust" the threads. If you need to re-iterate over a list of messages you will need to retrieve a new *Threads* object.

Things are not as bad as it seems though, you can store and reuse the single Thread objects as often as you want as long as you keep the parent Threads object around. (Recall that due to hierarchical memory allocation, all derived Threads objects will be invalid when we delete the parent Threads() object, even if it was already "exhausted".) So this works:

```
db    = Database()
threads = Query(db,'').search_threads() #get a Threads() object
threadlist = []
for thread in threads:
   threadlist.append(thread)

# threads is "exhausted" now and even len(threads) will raise an
# exception.
# However it will be kept around until all retrieved Thread() objects are
# also deleted. If you did e.g. an explicit del(threads) here, the
# following lines would fail.

# You can reiterate over *threadlist* however as often as you want.
# It is simply a list with Thread objects.

print (threadlist[0].get_thread_id())
print (threadlist[1].get_thread_id())
print (threadlist[0].get_total_messages())
```

Parameters

- **threads_p** (ctypes.c_void_p) – A pointer to an underlying *notmuch_threads_t* structure. These are not publically exposed, so a user will almost never instantiate a *Threads* object herself. They are usually handed back as a result, e.g. in *Query.search_threads()*. *threads_p* must be valid, we will raise an *NullPointerError* if it is *None*.
- **parent** – The parent object (ie *Query*) these tags are derived from. It saves a reference to it, so we can automatically delete the db object once all derived objects are dead.

TODO Make the iterator work more than once and cache the tags in the Python object.(?)

__len__()

len(*Threads*) returns the number of contained Threads

Note: As this iterates over the threads, we will not be able to iterate over them again! So this will fail:

```
#THIS FAILS
threads = Database().create_query('').search_threads()
if len(threads) > 0:                #this 'exhausts' threads
    # next line raises :exc:`NotInitializedError`!!!
    for thread in threads: print thread
```

__str__()

CHAPTER 10

Thread – A single thread

class Thread(*thread_p*, *parent=None*)

Represents a single message thread.

Parameters

- **thread_p** – A pointer to an internal notmuch_thread_t Structure. These are not publically exposed, so a user will almost never instantiate a *Thread* object herself. They are usually handed back as a result, e.g. when iterating through *Threads*. *thread_p* must be valid, we will raise an *NullPointerError* if it is *None*.
- **parent** – A 'parent' object is passed which this message is derived from. We save a reference to it, so we can automatically delete the parent object once all derived objects are dead.

get_thread_id()

Get the thread ID of 'thread'

The returned string belongs to 'thread' and will only be valid for as long as the thread is valid.

Returns String with a message ID

Raises *NotInitializedError* if the thread is not initialized.

get_total_messages()

Get the total number of messages in 'thread'

Returns The number of all messages in the database belonging to this thread. Contrast with *get_matched_messages()*.

Raises *NotInitializedError* if the thread is not initialized.

get_toplevel_messages()

Returns a *Messages* iterator for the top-level messages in 'thread'

This iterator will not necessarily iterate over all of the messages in the thread. It will only iterate over the messages in the thread which are not replies to other messages in the thread.

Returns *Messages*

Raises *NotInitializedError* if query is not initialized

Raises *NullPointerError* if search_messages failed

get_matched_messages()

Returns the number of messages in 'thread' that matched the query

Returns The number of all messages belonging to this thread that matched the `Query`'from which this thread was created. Contrast with :meth:`get_total_messages`.

Raises *NotInitializedError* if the thread is not initialized.

get_authors()
Returns the authors of 'thread'

The returned string is a comma-separated list of the names of the authors of mail messages in the query results that belong to this thread.

The returned string belongs to 'thread' and will only be valid for as long as this Thread() is not deleted.

get_subject()
Returns the Subject of 'thread'

The returned string belongs to 'thread' and will only be valid for as long as this Thread() is not deleted.

get_oldest_date()
Returns time_t of the oldest message date

Returns A time_t timestamp.

Return type c_unit64

Raises *NotInitializedError* if the message is not initialized.

get_newest_date()
Returns time_t of the newest message date

Returns A time_t timestamp.

Return type c_unit64

Raises *NotInitializedError* if the message is not initialized.

get_tags()
Returns the message tags

In the Notmuch database, tags are stored on individual messages, not on threads. So the tags returned here will be all tags of the messages which matched the search and which belong to this thread.

The *Tags* object is owned by the thread and as such, will only be valid for as long as this *Thread* is valid (e.g. until the query from which it derived is explicitely deleted).

Returns A *Tags* iterator.

Raises *NotInitializedError* if query is not initialized

Raises *NullPointerError* if search_messages failed

__str__() <==> *str(x)*

CHAPTER 11

Files and directories

11.1 `Filenames` – An iterator over filenames

class `Filenames` (*files_p*, *parent*)

Represents a list of filenames as returned by notmuch

Objects of this class implement the iterator protocol.

Note: The underlying library only provides a one-time iterator (it cannot reset the iterator to the start). Thus iterating over the function will "exhaust" the list of tags, and a subsequent iteration attempt will raise a *NotInitializedError*. Also note, that any function that uses iteration (nearly all) will also exhaust the tags. So both:

```
for name in filenames: print name
```

as well as:

```
number_of_names = len(names)
```

and even a simple:

```
#str() iterates over all tags to construct a space separated list
print(str(filenames))
```

will "exhaust" the Filenames. However, you can use *Message.get_filenames()* repeatedly to get fresh Filenames objects to perform various actions on filenames.

Parameters

- **`files_p`** (ctypes.c_void_p) – A pointer to an underlying *notmuch_tags_t* structure. These are not publically exposed, so a user will almost never instantiate a *Tags* object herself. They are usually handed back as a result, e.g. in *Database.get_all_tags()*. *tags_p* must be valid, we will raise an *NullPointerError* if it is *None*.
- **`parent`** – The parent object (ie *Message* these filenames are derived from, and saves a reference to it, so we can automatically delete the db object once all derived objects are dead.

`__len__`()

len(*Filenames*) returns the number of contained files

Note: This method exhausts the iterator object, so you will not be able to iterate over them again.

11.2 Directoy – A directory entry in the database

class Directory(*path*, *dir_p*, *parent*)

Represents a directory entry in the notmuch directory

Modifying attributes of this object will modify the database, not the real directory attributes.

The Directory object is usually derived from another object e.g. via *Database.get_directory()*, and will automatically be become invalid whenever that parent is deleted. You should therefore initialized this object handing it a reference to the parent, preventing the parent from automatically being garbage collected.

Parameters

- **path** – The absolute path of the directory object.
- **dir_p** – The pointer to an internal notmuch_directory_t object.
- **parent** – The object this Directory is derived from (usually a *Database*). We do not directly use this, but store a reference to it as long as this Directory object lives. This keeps the parent object alive.

get_child_files()

Gets a Filenames iterator listing all the filenames of messages in the database within the given directory.

The returned filenames will be the basename-entries only (not complete paths.

get_child_directories()

Gets a *Filenames* iterator listing all the filenames of sub-directories in the database within the given directory

The returned filenames will be the basename-entries only (not complete paths.

get_mtime()

Gets the mtime value of this directory in the database

Retrieves a previously stored mtime for this directory.

Parameters **mtime** – A (time_t) timestamp

Raises *NotmuchError*:

***STATUS*.NOT_INITIALIZED** The directory has not been initialized

set_mtime(*mtime*)

Sets the mtime value of this directory in the database

The intention is for the caller to use the mtime to allow efficient identification of new messages to be added to the database. The recommended usage is as follows:

•Read the mtime of a directory from the filesystem

•Call *Database.add_message()* for all mail files in the directory

•Call notmuch_directory_set_mtime with the mtime read from the filesystem. Then, when wanting to check for updates to the directory in the future, the client can call *get_mtime()* and know that it only needs to add files if the mtime of the directory and files are newer than the stored timestamp.

Note: *get_mtime()* function does not allow the caller to distinguish a timestamp of 0 from a non-existent timestamp. So don't store a timestamp of 0 unless you are comfortable with that.

Parameters **mtime** – A (time_t) timestamp

Raises *XapianError* a Xapian exception occurred, mtime not stored

Raises *ReadOnlyDatabaseError* the database was opened in read-only mode so directory mtime cannot be modified

Raises *NotInitializedError* the directory object has not been initialized

mtime
Property that allows getting and setting of the Directory *mtime* (read-write)

See *get_mtime()* and *set_mtime()* for usage and possible exceptions.

path
Returns the absolute path of this Directory (read-only)

CHAPTER 12

Indices and tables

- genindex
- search

n

Symbols

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

T

U

X