OfflineIMAP Documentation

Release 7.0.14

OfflineIMAP contributors

Apr 03, 2017

Contents

License dco (dco)

Documented APIs

CHAPTER 1

offlineimap's API documentation

Within *offlineimap*, the classes *OfflineImap* provides the high-level functionality. The rest of the classes should usually not needed to be touched by the user. Email repositories are represented by a *offlineimap.repository.Base.BaseRepository* or derivatives (see offlineimap.repository for details). A folder within a repository is represented by a *offlineimap.folder.Base.BaseFolder* or any derivative from offlineimap.folder.

This page contains the main API overview of OfflineImap 7.0.14.

OfflineImap can be imported as:

```
from offlineimap import OfflineImap
```

CHAPTER 2

offlineimap – The OfflineImap module

class **OfflineImap**(*cmdline_opts = None*)

The main class that encapsulates the high level use of OfflineImap.

To invoke OfflineImap you would call it with:

```
oi = OfflineImap()
oi.run()
```

_OfflineImap__deletefolder(*options*)

_OfflineImap__dumpstacks(*context=1*, *sighandler_deep=2*)

Signal handler: dump a stack trace for each existing thread.

_OfflineImap__migratefmd5(*options*)

_OfflineImap__parse_cmd_options()

_OfflineImap__serverdiagnostics(*options*)

_OfflineImap__sync(*options*)

Invoke the correct single/multithread syncing

self.config is supposed to have been correctly initialized already.

_OfflineImap__sync_singlethreaded(*list_accounts*, *profiledir*)

Executed in singlethreaded mode only.

Parameters **accs** – A list of accounts that should be synced

_get_activeaccounts(*options*)

run()

Parse the commandline and invoke everything

CHAPTER 3

offlineimap.account

An *accounts.Account* connects two email repositories that are to be synced. It comes in two flavors, normal and syncable.

class Account (*config*, *name*)
Represents an account (ie. 2 repositories) to sync.

Most of the time you will actually want to use the derived *accounts.SyncableAccount* which contains all functions used for syncing an account.

Parameters

- **config** (offlineimap.CustomConfig.CustomConfigParser) – Representing the offlineimap configuration file.
- **name** – A (str) string denoting the name of the Account as configured.

class SyncableAccount (**args*, ***kwargs*)
A syncable email account connecting 2 repositories.

Derives from *accounts.Account* but contains the additional functions syncrunner(), sync(), syncfolders(), used for syncing.

In multi-threaded mode, one instance of this object is run per "account" thread.

ui = <module 'offlineimap.ui' from '/home/docs/checkouts/readthedocs.org/user_builds/offlineimap/checkouts/stable/offli
Contains the current offlineimap.ui, and can be used for logging etc.

get_abort_event ()
Checks if an abort signal had been sent.

If the 'skipsleep' config option for this account had been set, with *set_abort_event(config, 1)* it will get cleared in this function. Ie, we will only skip one sleep and not all.

Returns True, if the main thread had called set_abort_event() earlier, otherwise 'False'.

get_local_folder (*remotefolder*)
Return the corresponding local folder for a given remotefolder.

getconf(*option*, *default=<function CustomConfigDefault>*)
Retrieves string from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconf_xform(*option*, *xforms*, *default=<function CustomConfigDefault>*)
Retrieves string from the configuration transforming the result.

Arguments:

- option: option name whose value is to be retrieved;
- xforms: iterable that returns transform functions to be applied to the value of the option, both retrieved and default one;
- default: default value for string if no such option exists.

getconfboolean(*option*, *default=<function CustomConfigDefault>*)
Retrieves boolean value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconffloat(*option*, *default=<function CustomConfigDefault>*)
Retrieves floating-point value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconfint(*option*, *default=<function CustomConfigDefault>*)
Retrieves integer value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconflist(*option*, *separator_re*, *default=<function CustomConfigDefault>*)
Retrieves strings from the configuration and splits it into the list of strings.

Arguments:

- option: option name whose value is to be retrieved;
- separator_re: regular expression for separator to be used for split operation;
- default: default return value if no such option exists.

serverdiagnostics()
Output diagnostics for all involved repositories.

set_abort_event(*config*, *signum*)
Set skip sleep/abort event for all accounts.

If we want to skip a current (or the next) sleep, or if we want to abort an autorefresh loop, the main thread can use set_abort_event() to send the corresponding signal. Signum = 1 implies that we want all accounts

to abort or skip the current or next sleep phase. Signum = 2 will end the autorefresh loop, ie all accounts will return after they finished a sync. signum=3 means, abort NOW, e.g. on SIGINT or SIGTERM.

This is a class method, it will send the signal to all accounts.

syncrunner()
The target for both single and multi-threaded modes.

OfflineImapError – A Notmuch execution error

exception OfflineImapError(*reason*, *severity*, *errcode=None*)
An Error during offlineimap synchronization

Parameters

- **reason** – Human readable string suitable for logging
- **severity** (*OfflineImapError.ERROR value*) – denoting which operations should be aborted. E.g. a ERROR.MESSAGE can occur on a faulty message, but a ERROR.REPO occurs when the server is offline.
- **errcode** – optional number denoting a predefined error situation (which let's us exit with a predefined exit value). So far, no errcodes have been defined yet.

This exception inherits directly from `Exception` and is raised on errors during the offlineimap execution. It has an attribute *severity* that denotes the severity level of the error.

class ERROR
Severity level of an Exception

•**MESSAGE**: Abort the current message, but continue with folder

•**FOLDER_RETRY**: Error syncing folder, but do retry

•**FOLDER**: Abort folder sync, but continue with next folder

•**REPO**: Abort repository sync, continue with next account

•**CRITICAL**: Immediately exit offlineimap

CHAPTER 4

offlineimap.globals – module with global variables

Module *offlineimap.globals* provides the read-only storage for the global variables.

All exported module attributes can be set manually, but this practice is highly discouraged and shouldn't be used. However, attributes of all stored variables can only be read, write access to them is denied.

Currently, we have only `options` attribute that holds command-line options as returned by OptionParser. The value of `options` must be set by `set_options()` prior to its first use.

options
You can access the values of stored options using the usual syntax, offlineimap.globals.options.<option-name>

set_options(*source*)
Sets the source for options variable.

CHAPTER 5

offlineimap.repository – Email repositories

A derivative of class *Base.BaseRepository* represents an email repository depending on the type of storage, possible options are:

- *IMAPRepository*,
- *MappedIMAPRepository*
- *GmailRepository*,
- *MaildirRepository*, or
- *LocalStatusRepository*.

Which class you need depends on your account configuration. The helper class *offlineimap.repository.Repository* is an *autoloader*, that returns the correct class depending on your configuration. So when you want to instanciate a new offlineimap.repository, you will mostly do it through this class.

class Repository (*account*, *reqtype*)

Abstract class that returns the correct Repository type instance based on 'account' and 'reqtype', e.g. a class:*ImapRepository* instance.

Load the correct Repository type and return that. The __init__ of the corresponding Repository class will be executed instead of this stub

Parameters

- **account** – Account
- **regtype** – 'remote', 'local', or 'status'

CHAPTER 6

offlineimap.repository.Base.BaseRepository – Representation of a mail repository

class BaseRepository(*reposname*, *account*)

accountname
Account name as string

connect()
Establish a connection to the remote, if necessary. This exists so that IMAP connections can all be established up front, gathering passwords as needed. It was added in order to support the error recovery – we need to connect first outside of the error trap in order to validate the password, and that's the point of this function.

deletefolder(*foldername*)

dropconnections()

forgetfolders()
Forgets the cached list of folders, if any. Useful to run after a sync run.

getaccount()

getconf(*option*, *default=<function CustomConfigDefault>*)
Retrieves string from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconf_xform(*option*, *xforms*, *default=<function CustomConfigDefault>*)
Retrieves string from the configuration transforming the result.

Arguments:

- option: option name whose value is to be retrieved;

- xforms: iterable that returns transform functions to be applied to the value of the option, both retrieved and default one;
- default: default value for string if no such option exists.

getconfboolean(*option*, *default=<function CustomConfigDefault>*)
Retrieves boolean value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconffloat(*option*, *default=<function CustomConfigDefault>*)
Retrieves floating-point value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconfig()

getconfint(*option*, *default=<function CustomConfigDefault>*)
Retrieves integer value from the configuration.

Arguments:

- option: option name whose value is to be retrieved;
- default: default return value if no such option exists.

getconflist(*option*, *separator_re*, *default=<function CustomConfigDefault>*)
Retrieves strings from the configuration and splits it into the list of strings.

Arguments:

- option: option name whose value is to be retrieved;
- separator_re: regular expression for separator to be used for split operation;
- default: default return value if no such option exists.

getfolder(*foldername*)

getfolders()
Returns a list of ALL folders on this server.

getkeywordmap()

getlocaleval()

getlocalroot()
Local root folder for storing messages. Will not be set for remote repositories.

getmapdir()

getname()

getsection()

getsep()

getuiddir()

holdordropconnections()

makefolder(*foldername*)
Create a new folder.

readonly
Is the repository readonly?

restore_atime()
Sets folders' atime back to their values after a sync

Controlled by the 'restoreatime' config parameter (default False), applies only to local Maildir mailboxes and does nothing on all other repository types.

should_create_folders()
Is folder creation enabled on this repository?

It is disabled by either setting the whole repository 'readonly' or by using the 'createfolders' setting.

should_sync_folder(*fname*)
Should this folder be synced?

startkeepalive()
The default implementation will do nothing.

stopkeepalive()
Stop keep alive, but don't bother waiting for the threads to terminate.

sync_folder_structure(*local_repo*, *status_repo*)
Sync the folders structure.

It does NOT sync the contents of those folders. nametrans rules in both directions will be honored

Configuring nametrans on BOTH repositories could lead to infinite folder creation cycles.

class IMAPRepository(*reposname*, *account*)

class MappedIMAPRepository(*reposname*, *account*)

class GmailRepository(*reposname*, *account*)
Gmail IMAP repository.

This class just has default settings for GMail's IMAP service. So you can do 'type = Gmail' instead of 'type = IMAP' and skip specifying the hostname, port etc. See http://mail.google.com/support/bin/answer.py?answer=78799&topic=12814 for the values we use.

Initialize a GmailRepository object.

class MaildirRepository(*reposname*, *account*)
Initialize a MaildirRepository object. Takes a path name to the directory holding all the Maildir directories.

class LocalStatusRepository(*reposname*, *account*)

CHAPTER 7

offlineimap.folder – Basic representation of a local or remote Mail folder

class BaseFolder (*name*, *repository*)

Parameters

- **name** – Path & name of folder minus root or reference
- **repository** – Repository() in which the folder is.

accountname
Account name as string

addmessageflags (*uid*, *flags*)
Adds the specified flags to the message's flag set.

If a given flag is already present, it will not be duplicated.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

Parameters **flags** – A set() of flags

addmessageheader (*content*, *linebreak*, *headername*, *headervalue*)
Adds new header to the provided message.

WARNING: This function is a bit tricky, and modifying it in the wrong way, may easily lead to data-loss.

Arguments: - content: message content, headers and body as a single string - linebreak: string that carries line ending - headername: name of the header to add - headervalue: value of the header to add

Note: The following documentation will not get displayed correctly after being processed by Sphinx. View the source of this method to read it.

This has to deal with strange corner cases where the header is missing or empty. Here are illustrations for all the cases, showing where the header gets inserted and what the end result is. In each illustration, '+' means the added contents. Note that these examples assume LF for linebreak, not CRLF, so '

' denotes a linebreak and '

' corresponds to the transition between header and body. However if the linebreak parameter is set to '

' then you would have to substitute ' ' for

'

' in the below examples.

•Case 1: No '

', leading ' '

+X-Flying-Pig-Header: i am here

This is the body

next line

•Case 2: '

' at position 0

+X-Flying-Pig-Header: i am here

This is the body

next line

•Case 3: No '

', no leading ' '

+X-Flying-Pig-Header: i am here

•

This is the body

next line

•Case 4: '

' at non-zero position

Subject: Something wrong with OI

From: some@person.at +

X-Flying-Pig-Header: i am here

This is the body

next line

addmessagelabels(*uid*, *labels*)
Adds the specified labels to the message's labels set. If a given label is already present, it will not be duplicated.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

Parameters **labels** – A set() of labels

addmessagesflags(*uidlist*, *flags*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

addmessageslabels(*uidlist*, *labels*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

cachemessagelist()
Cache the list of messages.

Reads the message list from disk or network and stores it in memory for later use. This list will not be re-read from disk or memory unless this function is called again.

change_message_uid(*uid*, *new_uid*)
Change the message from existing uid to new_uid.

If the backend supports it (IMAP does not).

Parameters **new_uid** – (optional) If given, the old UID will be changed to a new UID. This allows backends efficient renaming of messages if the UID has changed.

check_uidvalidity()
Tests if the cached UIDVALIDITY match the real current one

If required it saves the UIDVALIDITY value. In this case the function is not threadsafe. So don't attempt to call it from concurrent threads.

Returns Boolean indicating the match. Returns True in case it implicitely saved the UIDVALIDITY.

combine_flags_and_keywords(*uid*, *dstfolder*)
Combine the message's flags and keywords using the mapping for the destination folder.

copymessageto(*uid*, *dstfolder*, *statusfolder*, *register=1*)
Copies a message from self to dst if needed, updating the status

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

Parameters

- **uid** – uid of the message to be copied.
- **dstfolder** – A BaseFolder-derived instance
- **statusfolder** – A LocalStatusFolder instance
- **register** – whether we should register a new thread."

Returns Nothing on success, or raises an Exception.

deletemessage(*uid*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

deletemessageflags(*uid*, *flags*)
Removes each flag given from the message's flag set.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

If a given flag is already removed, no action will be taken for that flag.

deletemessageheaders(*content*, *header_list*)
Deletes headers in the given list from the message content.

Arguments: - content: message itself - header_list: list of headers to be deleted or just the header name

We expect our message to have ‘

‘ as line endings.

deletemessagelabels(*uid*, *labels*)
Removes each label given from the message's label set.

If a given label is already removed, no action will be taken for that label.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

deletemessages(*uidlist*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

deletemessagesflags(*uidlist*, *flags*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

deletemessageslabels(*uidlist*, *labels*)
Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

dofsync()

dropmessagelistcache()
Empty everythings we know about messages.

get_min_uid_file()

get_saveduidvalidity()
Return the previously cached UIDVALIDITY value

Returns UIDVALIDITY as (long) number or None, if None had been saved yet.

get_uidvalidity()
Retrieve the current connections UIDVALIDITY value

This function needs to be implemented by each Backend :returns: UIDVALIDITY as a (long) number.

getexplainedname()
Name that shows both real and nametrans-mangled values.

getfolderbasename()
Return base file name of file to store Status/UID info in.

getfullname()

getinstancelimitnamespace()
For threading folders, returns the instancelimitname for InstanceLimitedThreads.

getmaxage()
Return maxage.

maxage is allowed to be either an integer or a date of the form YYYY-mm-dd. This returns a time_struct.

getmaxsize()

getmessage(*uid*)
Returns the content of the specified message.

getmessagecount()
Gets the number of messages.

getmessageflags(*uid*)
Returns the flags for the specified message.

getmessageheader(*content*, *name*)
Return the value of the first occurence of the given header.

Header name is case-insensitive.

Arguments: - contents: message itself - name: name of the header to be searched

Returns: header value or None if no such header was found.

getmessageheaderlist(*content*, *name*)
Return a list of values for the given header.

Arguments: - contents: message itself - name: name of the header to be searched

Returns: list of header values or empty list if no such header was found.

getmessagekeywords(*uid*)
Returns the keywords for the specified message.

getmessagelabels(*uid*)
Returns the labels for the specified message.

getmessagelist()
Gets the current message list.

You must call cachemessagelist() before calling this function!

getmessagemtime(*uid*)
Returns the message modification time of the specified message.

getmessagetime(*uid*)
Return the received time for the specified message.

getmessageuidlist()
Gets a list of UIDs.

You may have to call cachemessagelist() before calling this function!

getname()
Returns name

getrepository()
Returns the repository object that this folder is within.

getroot()
Returns the root of the folder, in a folder-specific fashion.

getsep()
Returns the separator for this folder type.

getstartdate()
Retrieve the value of the configuration option startdate

getvisiblename()
The nametrans-transposed name of the folder's name.

ismessagelistempty()
Is the list of messages empty.

msglist_item_initializer(*uid*)
Returns value for empty messagelist element with given UID.

This function must initialize all fields of messagelist item and must be called every time when one creates new messagelist entry to ensure that all fields that must be present are present.

quickchanged(*statusfolder*)
Runs quick check for folder changes and returns changed status: True – changed, False – not changed.

Parameters **statusfolder** – keeps track of the last known folder state.

retrieve_min_uid()

save_min_uid(*min_uid*)

save_uidvalidity()
Save the UIDVALIDITY value of the folder to the cache

This function is not threadsafe, so don't attempt to call it from concurrent threads.

savemessage(*uid*, *content*, *flags*, *rtime*)
Writes a new message, with the specified uid.

If the uid is < 0: The backend should assign a new uid and return it. In case it cannot assign a new uid, it returns the negative uid passed in WITHOUT saving the message.

If the backend CAN assign a new uid, but cannot find out what this UID is (as is the case with some IMAP servers), it returns 0 but DOES save the message.

IMAP backend should be the only one that can assign a new uid.

If the uid is > 0, the backend should set the uid to this, if it can. If it cannot set the uid to that, it will save it anyway. It will return the uid assigned in any case.

Note that savemessage() does not check against dryrun settings, so you need to ensure that savemessage is never called in a dryrun mode.

savemessageflags(*uid*, *flags*)
Sets the specified message's flags to the given set.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

savemessagelabels(*uid*, *labels*, *ignorelabels=set()*, *mtime=0*)
Sets the specified message's labels to the given set.

Note that this function does not check against dryrun settings, so you need to ensure that it is never called in a dryrun mode.

storesmessages()
Should be true for any backend that actually saves message bodies. (Almost all of them). False for the LocalStatus backend. Saves us from having to slurp up messages just for localstatus purposes.

suggeststhreads()
Returns True if this folder suggests using threads for actions.

Only IMAP returns True. This method must honor any CLI or configuration option.

sync_this
Should this folder be synced or is it e.g. filtered out?

syncmessagesto(*dstfolder*, *statusfolder*)
Syncs messages in this folder to the destination dstfolder.

This is the high level entry for syncing messages in one direction. Syncsteps are:

Pass1: Copy locally existing messages Copy messages in self, but not statusfolder to dstfolder if not already in dstfolder. dstfolder might assign a new UID (e.g. if uploading to IMAP). Update statusfolder.

Pass2: Remove locally deleted messages Get all UIDS in statusfolder but not self. These are messages that were deleted in 'self'. Delete those from dstfolder and statusfolder.

After this pass, the message lists should be identical wrt the uids present (except for potential negative uids that couldn't be placed anywhere).

Pass3: Synchronize flag changes Compare flag mismatches in self with those in statusfolder. If msg has a valid UID and exists on dstfolder (has not e.g. been deleted there), sync the flag change to both dstfolder and statusfolder.

Pass4: Synchronize label changes (Gmail only) Compares label mismatches in self with those in statusfolder. If msg has a valid UID and exists on dstfolder, syncs the labels to both dstfolder and statusfolder.

Parameters

- **dstfolder** – Folderinstance to sync the msgs to.
- **statusfolder** – LocalStatus instance to sync against.

uidexists(*uid*)
Returns True if uid exists.

waitforthread()
Implements method that waits for thread to be usable. Should be implemented only for folders that suggest threads.

CHAPTER 8

offlineimap.ui – A flexible logging system

OfflineImap has various ui systems, that can be selected. They offer various functionalities. They must implement all functions that the offlineimap.ui.UIBase offers. Early on, the ui must be set using *getglobalui()*

ui.**setglobalui**(*newui*)
Set the global ui object to be used for logging.

ui.**getglobalui**()
Return the current ui object.

Base UI plugin

class UIBase(*config*, *loglevel=20*)

acct(*account*)
Output that we start syncing an account (and start counting).

acctdone(*account*)
Output that we finished syncing an account (in which time).

connecting(*reposname*, *hostname*, *port*)
Log 'Establishing connection to'.

copyingmessage(*uid*, *num*, *num_to_copy*, *src*, *destfolder*)
Output a log line stating which message we copy.

error(*exc*, *exc_traceback=None*, *msg=None*)
Log a message at severity level ERROR.

Log Exception 'exc' to error log, possibly prepended by a preceding error "msg", detailing at what point the error occurred.

In debug mode, we also output the full traceback that occurred if one has been passed in via sys.info()[2].

Also save the Exception to a stack that can be output at the end of the sync run when offlineiamp exits. It is recommended to always pass in exceptions if possible, so we can give the user the best debugging info.

We are always pushing tracebacks to the exception queue to make them to be output at the end of the run to allow users pass sensible diagnostics to the developers or to solve problems by themselves.

One example of such a call might be:

ui.error(exc, sys.exc_info()[2], msg="While syncing Folder %s in " "repo %s")

getnicename(*object*)
Return the type of a repository or Folder as string.

(IMAP, Gmail, Maildir, etc...)

getthreadaccount(*thr=None*)
Get Account() for a thread (current if None)

If no account has been registered with this thread, return 'None'.

ignorecopyingmessage(*uid*, *src*, *destfolder*)
Output a log line stating which message is ignored.

info(*msg*)
Display a message.

init_banner()
Called when the UI starts. Must be called before any other UI call except isusable(). Displays the copyright banner. This is where the UI should do its setup – TK, for instance, would create the application window here.

isusable()
Returns true if this UI object is usable in the current environment. For instance, an X GUI would return true if it's being run in X with a valid DISPLAY setting, and false otherwise.

makefolder(*repo*, *foldername*)
Called when a folder is created.

registerthread(*account*)
Register current thread as being associated with an account name.

savemessage(*debugtype*, *uid*, *flags*, *folder*)
Output a log line stating that we save a msg.

serverdiagnostics(*repository*, *type*)
Connect to repository and output useful information for debugging.

setlogfile(*logfile*)
Create file handler which logs to file.

setup_consolehandler()
Backend specific console handler.

Sets up things and adds them to self.logger. :returns: The logging.Handler() for console output

setup_sysloghandler()
Backend specific syslog handler.

skippingfolder(*folder*)
Called when a folder sync operation is started.

sleep(*sleepsecs*, *account*)
This function does not actually output anything, but handles the overall sleep, dealing with updates as necessary. It will, however, call sleeping() which DOES output something.

Returns 0/False if timeout expired, 1/2/True if there is a request to cancel the timer.

sleeping(*sleepsecs*, *remainingsecs*)
Sleep for sleepsecs, display remainingsecs to go.

Does nothing if sleepsecs <= 0. Display a message on the screen if we pass a full minute.

This implementation in UIBase does not support this, but some implementations return 0 for successful sleep and 1 for an 'abort', ie a request to sync immediately.

syncfolders(*src_repo*, *dst_repo*)
Log 'Copying folder structure...'.

syncingfolder(*srcrepos*, *srcfolder*, *destrepos*, *destfolder*)
Called when a folder sync operation is started.

terminate(*exitstatus=0*, *errortitle=None*, *errormsg=None*)
Called to terminate the application.

threadException(*thread*)
Called when a thread has terminated with an exception. The argument is the ExitNotifyThread that has so terminated.

threadExited(*thread*)
Called when a thread has exited normally.

Many UIs will just ignore this.

unregisterthread(*thr*)
Unregister a thread as being associated with an account name.

License This module is covered under the GNU GPL v2 (or later).

Python Module Index

o

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

O

Q

R

S

T

U

W