alot Documentation

Release 0.6

Patrick Totzke

Contents

Alot is a terminal-based mail user agent for the notmuch mail system. It features a modular and command prompt driven interface to provide a full MUA experience as an alternative to the Emacs mode shipped with notmuch.

CHAPTER 1

Installation

dependencies

Alot depends on recent versions of notmuch and urwid. Note that due to restrictions on argparse and subprocess, you need to run *'3.0'* > *python '2.7'* (see *faq*). A full list of dependencies is below:

- libmagic and python bindings, *5.04*
- configobj, *4.7.0*
- twisted, *10.2.0*:
- libnotmuch and it's python bindings, *0.13*
- urwid toolkit, *1.1.0*
- urwidtrees, *1.0*
- gpg

Note: urwidtrees was only recently detached from alot and is not widely available as a separate package. You can install it e.g., via pip directly from github:

```
pip install --user https://github.com/pazz/urwidtrees/archive/master.zip
```

On debian/ubuntu the rest are packaged as:

```
python-setuptools python-magic python-configobj python-twisted python-notmuch python-
→urwid python-gpg
```

On fedora/redhat these are packaged as:

```
python-setuptools python-magic python-configobj python-twisted python-notmuch python-
→urwid python-gpg
```

Alot uses mailcap to look up mime-handler for inline rendering and opening of attachments. For a full description of the maicap protocol consider the manpage `mailcap(5)` or **RFC 1524**. To avoid surprises you should at least have an inline renderer (copiousoutput) set up for *text/html*, i.e. have something like this in your `~/.mailcap`:

```
text/html;  w3m -dump -o document_charset=%{charset} '%s'; nametemplate=%s.html;
↪copiousoutput
```

get and install alot

You can use *pip* to install directly from GitHub:

```
$ pip install --user https://github.com/pazz/alot/archive/master.zip
```

Don't have pip installed? Just download and extract, then run:

```
python setup.py install --user
```

Make sure `~/.local/bin` is in your `PATH`. For system-wide installation omit the *–user* flag and call with the respective permissions.

generate manual and manpage

To generate the documentation you need sphinx, *1.07* installed. Go to `docs/` and do a:

```
make html
make man
```

to generate the user manual and a man page. Both will end up in their respective subfolders in `docs/build`.

Usage

Commandline invocation

Synopsis

alot [options ...] [subcommand]

Options

> **-r, --read-only**   open db in read only mode
>
> **-c, --config=FILENAME**   config file (default: ~/.config/alot/config)
>
> **-n, --notmuch-config=FILENAME** notmuch   config   (default: $NOTMUCH_CONFIG   or
> ~/.notmuch-config)
>
> **-C, --colour-mode=COLOUR**   terminal colour mode (default: 256). Must be 1, 16 or 256
>
> **-p, --mailindex-path=PATH**   path to notmuch index
>
> **-d, --debug-level=LEVEL**   debug log (default: info). Must be one of debug,info,warning or error
>
> **-l, --logfile=FILENAME**   logfile (default: /dev/null)
>
> **-v, --version**   Display version string and exit
>
> **-h, --help**   Display help and exit

Commands

alot can be invoked with an optional subcommand from the command line. Those have their own parameters (see e.g. *alot search –help*). The following commands are available.

search start in a search buffer using the querystring provided as parameter. See also `notmuch-search-terms(7)`.

compose compose a new message

bufferlist start with only a bufferlist buffer open

taglist start with only a taglist buffer open

pyshell start the interactive python shell inside alot

UNIX Signals

SIGUSR1 Refreshes the current buffer.

SIGINT Shuts down the user interface.

First Steps in the UI

The arrow keys, *page-up/down*, *j*, *k* and *Space* can be used to move the focus. *Escape* cancels prompts and *Enter* selects. Hit *:* at any time and type in commands to the prompt.

The interface shows one buffer at a time, you can use *Tab* and *Shift-Tab* to switch between them, close the current buffer with *d* and list them all with *;*.

The buffer type or *mode* (displayed at the bottom left) determines which prompt commands are available. Usage information on any command can be listed by typing *help YOURCOMMAND* to the prompt; The key bindings for the current mode are listed upon pressing *?*.

Commands

Alot interprets user input as command line strings given via its prompt or *bound to keys* in the config. Command lines are semi-colon separated command strings, each of which starts with a command name and possibly followed by arguments.

See the sections below for which commands are available in which (UI) mode. *global* commands are available independently of the mode.

Global Commands globally available commands

Commands in search mode commands available when showing thread search results

Commands in thread mode commands available while displaying a thread

Commands in envelope mode commands during message composition

Commands in bufferlist mode commands while listing active buffers

Commands in taglist mode commands while listing all tagstrings present in the notmuch database

Global Commands

The following commands are available globally

`bclose`
    close a buffer

> **optional arguments**
>
> > **—redraw** redraw current buffer after command has finished.
> >
> > **—force** never ask for confirmation.

bprevious
> focus previous buffer

search
> open a new search buffer. Search obeys the notmuch *search.exclude_tags* setting.
>
> **argument** search string
>
> **optional arguments**
>
> > **—sort** sort order. Valid choices are: 'oldest_first','newest_first','message_id','unsorted'.

repeat
> Repeats the command executed last time

prompt
> prompts for commandline and interprets it upon select
>
> **argument** initial content

help
> display help for a command. Use 'bindings' to display all keybings interpreted in current mode.'
>
> **argument** command or 'bindings'

buffer
> focus buffer with given index
>
> **argument** buffer index to focus

move
> move focus in current buffer
>
> **argument** up, down, [half]page up, [half]page down, first, last

shellescape
> run external command
>
> **argument** command line to execute
>
> **optional arguments**
>
> > **—spawn** run in terminal window.
> >
> > **—thread** run in separate thread.
> >
> > **—refocus** refocus current buffer after command has finished.

refresh
> refresh the current buffer

reload
> Reload all configuration files

pyshell
> open an interactive python shell for introspection

compose
> compose a new email
>
> **argument** None

optional arguments

> **—sender** sender.
>
> **—template** path to a template message file.
>
> **—subject** subject line.
>
> **—to** recipients.
>
> **—cc** copy to.
>
> **—bcc** blind copy to.
>
> **—attach** attach files.
>
> **—omit_signature** do not add signature.
>
> **—spawn** spawn editor in new terminal.

exit
> Shut down cleanly.
>
> The _prompt variable is for internal use only, it's used to control prompting to close without sending, and is used by the BufferCloseCommand if settings change after yielding to the UI.

flush
> flush write operations or retry until committed

bufferlist
> open a list of active buffers

call
> Executes python code
>
> **argument** python command string to call

bnext
> focus next buffer

taglist
> opens taglist buffer
>
> **optional arguments**
>
> > **—tags** tags to display.

Commands in *search* mode

The following commands are available in search mode

sort
> set sort order
>
> **argument** sort order. valid choices are: 'oldest_first','newest_first','message_id','unsorted'.

untag
> remove tags from all messages in the thread that match the query
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
> > **—no-flush** postpone a writeout to the index (Defaults to: 'True').
> >
> > **—all** retag all messages in search result.

move
> move focus in search buffer
>
> **argument** last

retag
> set tags of all messages in the thread that match the current query
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
>> **—no-flush** postpone a writeout to the index (Defaults to: 'True').
>>
>> **—all** retag all messages in search result.

refineprompt
> prompt to change this buffers querystring

tag
> add tags to all messages in the thread that match the current query
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
>> **—no-flush** postpone a writeout to the index (Defaults to: 'True').
>>
>> **—all** retag all messages in search result.

refine
> refine query
>
> **argument** search string
>
> **optional arguments**
>
>> **—sort** sort order. Valid choices are: 'oldest_first','newest_first','message_id','unsorted'.

retagprompt
> prompt to retag selected threads' tags

toggletags
> flip presence of tags on this thread. A tag is considered present if at least one message contained in this thread is tagged with it. In that case this command will remove the tag from every message in the thread.
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
>> **—no-flush** postpone a writeout to the index (Defaults to: 'True').

select
> open thread in a new buffer

Commands in *thread* mode

The following commands are available in thread mode

pipeto
> pipe message(s) to stdin of a shellcommand
>
> **argument** shellcommand to pipe to
>
> **optional arguments**

> **—all** pass all messages.
>
> **—format** output format. Valid choices are: 'raw','decoded','id','filepath' (Defaults to: 'raw').
>
> **—separately** call command once for each message.
>
> **—background** don't stop the interface.
>
> **—add_tags** add 'Tags' header to the message.
>
> **—shell** let the shell interpret the command.
>
> **—notify_stdout** display cmd's stdout as notification.
>
> **—field_key** mailcap field key for decoding (Defaults to: 'copiousoutput').

editnew
> edit message in as new
>
> **optional arguments**
>
> > **—spawn** open editor in new window.

move
> move focus in current buffer
>
> **argument** up, down, [half]page up, [half]page down, first, last, parent, first reply, last reply, next sibling, previous sibling, next, previous, next unfolded, previous unfolded, next NOTMUCH_QUERY, previous NOTMUCH_QUERY

untag
> remove tags from message(s)
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
> > **—all** tag all messages in thread.
> >
> > **—no-flush** postpone a writeout to the index (Defaults to: 'True').

toggleheaders
> display all headers
>
> **argument** query used to filter messages to affect

print
> print message(s)
>
> **optional arguments**
>
> > **—all** print all messages.
> >
> > **—raw** pass raw mail string.
> >
> > **—separately** call print command once for each message.
> >
> > **—add_tags** add 'Tags' header to the message.

bounce
> directly re-send selected message

togglesource
> display message source
>
> **argument** query used to filter messages to affect

retag
> set message(s) tags.
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
> > **—all** tag all messages in thread.
> >
> > **—no-flush** postpone a writeout to the index (Defaults to: 'True').

fold
> fold message(s)
>
> **argument** query used to filter messages to affect

tag
> add tags to message(s)
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
> > **—all** tag all messages in thread.
> >
> > **—no-flush** postpone a writeout to the index (Defaults to: 'True').

remove
> remove message(s) from the index
>
> **optional arguments**
>
> > **—all** remove whole thread.

unfold
> unfold message(s)
>
> **argument** query used to filter messages to affect

forward
> forward message
>
> **optional arguments**
>
> > **—attach** attach original mail.
> >
> > **—spawn** open editor in new window.

reply
> reply to message
>
> **optional arguments**
>
> > **—all** reply to all.
> >
> > **—list** reply to list.
> >
> > **—spawn** open editor in new window.

save
> save attachment(s)
>
> **argument** path to save to
>
> **optional arguments**
>
> > **—all** save all attachments.

toggletags
> flip presence of tags on message(s)
>
> **argument** comma separated list of tags
>
> **optional arguments**
>
>> **—all** tag all messages in thread.
>>
>> **—no-flush** postpone a writeout to the index (Defaults to: 'True').

select
> **select focussed element. The fired action depends on the focus:**
>
>> - if message summary, this toggles visibility of the message,
>> - if attachment line, this opens the attachment

Commands in *envelope* mode

The following commands are available in envelope mode

unencrypt
> remove request to encrypt message before sending

set
> set header value
>
> **positional arguments** 0: header to refine 1: value
>
> **optional arguments**
>
>> **—append** keep previous values.

encrypt
> request encryption of message before sendout
>
> **argument** keyid of the key to encrypt with
>
> **optional arguments**
>
>> **—trusted** only add trusted keys.

togglesign
> toggle sign status
>
> **argument** which key id to use

toggleheaders
> toggle display of all headers

edit
> edit mail
>
> **optional arguments**
>
>> **—spawn** spawn editor in new terminal.
>>
>> **—refocus** refocus envelope after editing (Defaults to: 'True').

retag
> set message tags.
>
> **argument** comma separated list of tags

tag
> add tags to message
>
> **argument** comma separated list of tags

send
> send mail

sign
> mark mail to be signed before sending
>
> **argument** which key id to use

untag
> remove tags from message
>
> **argument** comma separated list of tags

attach
> attach files to the mail
>
> **argument** file(s) to attach (accepts wildcads)

unattach
> remove attachments from current envelope
>
> **argument** which attached file to remove

rmencrypt
> do not encrypt to given recipient key
>
> **argument** keyid of the key to encrypt with

refine
> prompt to change the value of a header
>
> **argument** header to refine

toggleencrypt
> toggle if message should be encrypted before sendout
>
> **argument** keyid of the key to encrypt with
>
> **optional arguments**
>
>> **—trusted** only add trusted keys.

save
> save draft

unsign
> mark mail not to be signed before sending

toggletags
> flip presence of tags on message
>
> **argument** comma separated list of tags

unset
> remove header field
>
> **argument** header to refine

Commands in *bufferlist* mode

The following commands are available in bufferlist mode

close
> close focussed buffer

open
> focus selected buffer

Commands in *taglist* mode

The following commands are available in taglist mode

select
> search for messages with selected tag

Cryptography

Alot has built in support for constructing signed and/or encrypted mails according to PGP/MIME (RFC 3156, RFC 3156) via gnupg. It does however rely on a running *gpg-agent* to handle password entries.

Note: You need to have *gpg-agent* running to use GPG with alot!

gpg-agent will handle passphrase entry in a secure and configurable way, and it will cache your passphrase for some time so you don't have to enter it over and over again. For details on how to set this up we refer to gnupg's manual.

Signing outgoing emails

You can use the commands *sign*, *unsign* and *togglesign* in envelope mode to determine if you want this mail signed and if so, which key to use. To specify the key to use you may pass a hint string as argument to the *sign* or *togglesign* command. This hint would typically be a fingerprint or an email address associated (by gnupg) with a key.

Signing (and hence passwd entry) will be done at most once shortly before a mail is sent.

In case no key is specified, alot will leave the selection of a suitable key to gnupg so you can influence that by setting the *default-key* option in `~/.gnupg/gpg.conf` accordingly.

You can set the default to-sign bit and the key to use for each *account* individually using the options *sign_by_default* and *gpg_key*.

Encrypt outgoing emails

You can use the commands *encrypt*, *unencrypt* and and *toggleencrypt* and in envelope mode to ask alot to encrypt the mail before sending. The *encrypt* command accepts an optional hint string as argument to determine the key of the recipient.

You can set the default to-encrypt bit for each *account* individually using the option *encrypt_by_default*.

Note: If you want to access encrypt mail later it is useful to add yourself to the list of recipients when encrypting with gpg (not the recipients whom mail is actually send to). The simplest way to do this is to use the *encrypt-to* option

---

in the `~/.gnupg/gpg.conf`. But you might have to specify the correct encryption subkey otherwise gpg seems to throw an error.

The arrow keys, *page-up/down*, *j*, *k* and *Space* can be used to move the focus. *Escape* cancels prompts and *Enter* selects. Hit *:* at any time and type in commands to the prompt.

The interface shows one buffer at a time, you can use *Tab* and *Shift-Tab* to switch between them, close the current buffer with *d* and list them all with *;*.

The buffer type or *mode* (displayed at the bottom left) determines which prompt commands are available. Usage information on any command can be listed by typing *help YOURCOMMAND* to the prompt; The key bindings for the current mode are listed upon pressing *?*.

alot [options ...] [subcommand]

Cryptography

Alot has built in support for constructing signed and/or encrypted mails according to PGP/MIME ([RFC 3156](#), [RFC 3156](#)) via gnupg. It does however rely on a running *gpg-agent* to handle password entries.

Note: You need to have *gpg-agent* running to use GPG with alot!

gpg-agent will handle passphrase entry in a secure and configurable way, and it will cache your passphrase for some time so you don't have to enter it over and over again. For details on how to set this up we refer to [gnupg's manual](#).

Signing outgoing emails

You can use the commands *sign*, *unsign* and *togglesign* in envelope mode to determine if you want this mail signed and if so, which key to use. To specify the key to use you may pass a hint string as argument to the *sign* or *togglesign* command. This hint would typically be a fingerprint or an email address associated (by gnupg) with a key.

Signing (and hence passwd entry) will be done at most once shortly before a mail is sent.

In case no key is specified, alot will leave the selection of a suitable key to gnupg so you can influence that by setting the *default-key* option in `~/.gnupg/gpg.conf` accordingly.

You can set the default to-sign bit and the key to use for each *account* individually using the options *sign_by_default* and *gpg_key*.

Encrypt outgoing emails

You can use the commands *encrypt*, *unencrypt* and and *toggleencrypt* and in envelope mode to ask alot to encrypt the mail before sending. The *encrypt* command accepts an optional hint string as argument to determine the key of the recipient.

You can set the default to-encrypt bit for each *account* individually using the option *encrypt_by_default*.

Note: If you want to access encrypt mail later it is useful to add yourself to the list of recipients when encrypting with gpg (not the recipients whom mail is actually send to). The simplest way to do this is to use the *encrypt-to* option in the `~/.gnupg/gpg.conf`. But you might have to specify the correct encryption subkey otherwise gpg seems to throw an error.

Configuration

Alot reads a config file in "INI" syntax: It consists of key-value pairs that use "=" as separator and '#' is comment-prefixes. Sections and subsections are defined using square brackets.

The default location for the config file is `~/.config/alot/config`.

All configs are optional, but if you want to send mails you need to specify at least one *account* in your config.

Config options

The following lists all available config options with their type and default values. The type of an option is used to validate a given value. For instance, if the type says "boolean" you may only provide "True" or "False" as values in your config file, otherwise alot will complain on startup. Strings *may* be quoted but do not need to be.

ask_subject

> **Type** boolean
>
> **Default** True

attachment_prefix

> directory prefix for downloading attachments
>
> **Type** string
>
> **Default** "~"

auto_remove_unread

> automatically remove 'unread' tag when focussing messages in thread mode
>
> **Type** boolean
>
> **Default** True

auto_replyto_mailinglist

> Automatically switch to list reply mode if appropriate
>
> > **Type** boolean
> >
> > **Default** False

bounce_force_address

> Always use the accounts main address when constructing "Resent-From" headers for bounces. Set this to False to use the address string as received in the original message.
>
> > **Type** boolean
> >
> > **Default** False

bounce_force_realname

> Always use the proper realname when constructing "Resent-From" headers for bounces. Set this to False to use the realname string as received in the original message.
>
> > **Type** boolean
> >
> > **Default** True

bufferclose_focus_offset

> offset of next focused buffer if the current one gets closed
>
> > **Type** integer
> >
> > **Default** -1

bufferlist_statusbar

> Format of the status-bar in bufferlist mode. This is a pair of strings to be left and right aligned in the status-bar that may contain variables:
>
> > • *{buffer_no}*: index of this buffer in the global buffer list
> >
> > • *{total_messages}*: total numer of messages indexed by notmuch
> >
> > • *{pending_writes}*: number of pending write operations to the index
>
> > **Type** mixed_list
> >
> > **Default** [{buffer_no}: bufferlist], {input_queue} total messages: {total_messages}

bug_on_exit

> confirm exit
>
> > **Type** boolean
> >
> > **Default** False

colourmode

> number of colours to use
>
> > **Type** option, one of ['1', '16', '256']

> **Default** 256

complete_matching_abook_only

> in case more than one account has an address book: Set this to True to make tab completion for recipients during compose only look in the abook of the account matching the sender address
>
> **Type** boolean
>
> **Default** False

compose_ask_tags

> prompt for initial tags when compose
>
> **Type** boolean
>
> **Default** False

displayed_headers

> headers that get displayed by default
>
> **Type** string list
>
> **Default** From, To, Cc, Bcc, Subject

edit_headers_blacklist

> see *edit_headers_whitelist*
>
> **Type** string list
>
> **Default** Content-Type, MIME-Version, References, In-Reply-To

edit_headers_whitelist

> Which header fields should be editable in your editor used are those that match the whitelist and don't match the blacklist. in both cases '*' may be used to indicate all fields.
>
> **Type** string list
>
> **Default** *,

editor_cmd

> editor command if unset, alot will first try the `EDITOR` env variable, then `/usr/bin/editor`
>
> **Type** string
>
> **Default** None

editor_in_thread

> call editor in separate thread. In case your editor doesn't run in the same window as alot, setting true here will make alot non-blocking during edits
>
> **Type** boolean
>
> **Default** False

---

editor_spawn

> use *terminal_cmd* to spawn a new terminal for the editor? equivalent to always providing the –*spawn=yes* parameter to compose/edit commands
>
> **Type** boolean
>
> **Default** False

editor_writes_encoding

> file encoding used by your editor
>
> **Type** string
>
> **Default** "UTF-8"

envelope_headers_blacklist

> headers that are hidden in envelope buffers by default
>
> **Type** string list
>
> **Default** In-Reply-To, References

envelope_statusbar

> Format of the status-bar in envelope mode. This is a pair of strings to be left and right aligned in the status-bar. Apart from the global variables listed at *bufferlist_statusbar* these strings may contain variables:
>
> > •*{to}*: To-header of the envelope
>
> **Type** mixed_list
>
> **Default** [{buffer_no}: envelope], {input_queue} total messages: {total_messages}

flush_retry_timeout

> timeout in seconds after a failed attempt to writeout the database is repeated. Set to 0 for no retry.
>
> **Type** integer
>
> **Default** 5

followup_to

> When one of the recipients of an email is a subscribed mailing list, set the "Mail-Followup-To" header to the list of recipients without yourself
>
> **Type** boolean
>
> **Default** False

forward_force_address

> Always use the accounts main address when constructing "From" headers for forwards. Set this to False to use the address string as received in the original message.
>
> **Type** boolean

> **Default** False

forward_force_realname

> Always use the proper realname when constructing "From" headers for forwards. Set this to False to use the realname string as received in the original message.
>
> **Type** boolean
>
> **Default** True

forward_subject_prefix

> String prepended to subject header on forward only if original subject doesn't start with 'Fwd:' or this prefix
>
> **Type** string
>
> **Default** "Fwd: "

handle_mouse

> enable mouse support - mouse tracking will be handled by urwid
>
> ---
> **Note:** If this is set to True mouse events are passed from the terminal to urwid/alot. This means that normal text selection in alot will not be possible. Most terminal emulators will still allow you to select text when shift is pressed.
> ---
>
> **Type** boolean
>
> **Default** False

history_size

> The number of command line history entries to save
>
> ---
> **Note:** You can set this to -1 to save *all* entries to disk but the history file might get *very* long.
> ---
>
> **Type** integer
>
> **Default** 50

honor_followup_to

> When group-reply-ing to an email that has the "Mail-Followup-To" header set, use the content of this header as the new "To" header and leave the "Cc" header empty
>
> **Type** boolean
>
> **Default** False

hooksfile

> where to look up hooks
>
> **Type** string

3.1. Config options

> **Default** "~/.config/alot/hooks.py"

initial_command

> initial command when none is given as argument:
>
> **Type** string
>
> **Default** "search tag:inbox AND NOT tag:killed"

input_timeout

> timeout in (floating point) seconds until partial input is cleared
>
> **Type** float
>
> **Default** 1.0

mailinglists

> The list of addresses associated to the mailinglists you are subscribed to
>
> **Type** string list
>
> **Default** ,

msg_summary_hides_threadwide_tags

> In a thread buffer, hide from messages summaries tags that are commom to all messages in that thread.
>
> **Type** boolean
>
> **Default** True

notify_timeout

> time in secs to display status messages
>
> **Type** integer
>
> **Default** 2

periodic_hook_frequency

> The number of seconds to wait between calls to the loop_hook
>
> **Type** integer
>
> **Default** 300

prefer_plaintext

> prefer plaintext alternatives over html content in multipart/alternative
>
> **Type** boolean
>
> **Default** False

print_cmd

how to print messages: this specifies a shell command used for printing. threads/messages are piped to this command as plain text. muttprint/a2ps works nicely

Type string

Default None

prompt_suffix

Suffix of the prompt used when waiting for user input

Type string

Default ”:”

quit_on_last_bclose

shut down when the last buffer gets closed

Type boolean

Default False

quote_prefix

String prepended to line when quoting

Type string

Default "> "

reply_account_header_priority

The list of headers to match to determine sending account for a reply. Headers are searched in the order in which they are specified here, and the first header containing a match is used. If multiple accounts match in that header, the one defined first in the account block is used.

Type string list

Default From, To, Cc, Envelope-To, X-Envelope-To, Delivered-To

reply_force_address

Always use the accounts main address when constructing "From" headers for replies. Set this to False to use the address string as received in the original message.

Type boolean

Default False

reply_force_realname

Always use the proper realname when constructing "From" headers for replies. Set this to False to use the realname string as received in the original message.

Type boolean

Default True

reply_subject_prefix

String prepended to subject header on reply only if original subject doesn't start with 'Re:' or this prefix

Type string

Default "Re: "

search_statusbar

Format of the status-bar in search mode. This is a pair of strings to be left and right aligned in the status-bar. Apart from the global variables listed at *bufferlist_statusbar* these strings may contain variables:

- *{querystring}*: search string

- *{result_count}*: number of matching messages

- *{result_count_positive}*: 's' if result count is greater than 0.

Type mixed_list

Default [{buffer_no}: search] for "{querystring}", {input_queue} {result_count} of {total_messages} messages

search_threads_sort_order

default sort order of results in a search

Type option, one of ['oldest_first', 'newest_first', 'message_id', 'unsorted']

Default newest_first

show_statusbar

display status-bar at the bottom of the screen?

Type boolean

Default True

tabwidth

number of spaces used to replace tab characters

Type integer

Default 8

taglist_statusbar

Format of the status-bar in taglist mode. This is a pair of strings to be left and right aligned in the status-bar. These strings may contain variables listed at *bufferlist_statusbar* that will be substituted accordingly.

Type mixed_list

Default [{buffer_no}: taglist], {input_queue} total messages: {total_messages}

template_dir

templates directory that contains your message templates. It will be used if you give *compose – template* a filename without a path prefix.

Type string

Default "$XDG_CONFIG_HOME/alot/templates"

terminal_cmd

set terminal command used for spawning shell commands

Type string

Default "x-terminal-emulator -e"

theme

name of the theme to use

Type string

Default None

themes_dir

directory containing theme files

Type string

Default None

thread_authors_me

Word to replace own addresses with. Works in combination with *thread_authors_replace_me*

Type string

Default "Me"

thread_authors_order_by

When constructing the unique list of thread authors, order by date of author's first or latest message in thread

Type option, one of ['first_message', 'latest_message']

Default first_message

thread_authors_replace_me

Replace own email addresses with "me" in author lists Uses own addresses and aliases in all configured accounts.

Type boolean

Default True

thread_focus_linewise

Split message body linewise and allows to (move) the focus to each individual line. Setting this to False will result in one potentially big text widget for the whole message body.

> **Type** boolean
>
> **Default** True

thread_statusbar

> Format of the status-bar in thread mode. This is a pair of strings to be left and right aligned in the status-bar. Apart from the global variables listed at *bufferlist_statusbar* these strings may contain variables:
>
> > •*{tid}*: thread id
> >
> > •*{subject}*: subject line of the thread
> >
> > •*{authors}*: abbreviated authors string for this thread
> >
> > •*{message_count}*: number of contained messages
>
> **Type** mixed_list
>
> **Default** [{buffer_no}: thread] {subject}, {input_queue} total messages: {total_messages}

thread_subject

> What should be considered to be "the thread subject". Valid values are:
>
> > •'notmuch' (the default), will use the thread subject from notmuch, which depends on the selected sorting method
> >
> > •'oldest' will always use the subject of the oldest message in the thread as the thread subject
>
> **Type** option, one of ['oldest', 'notmuch']
>
> **Default** notmuch

timestamp_format

> timestamp format in strftime format syntax
>
> **Type** string
>
> **Default** None

user_agent

> value of the User-Agent header used for outgoing mails. setting this to the empty string will cause alot to omit the header all together. The string '{version}' will be replaced by the version string of the running instance.
>
> **Type** string
>
> **Default** "alot/{version}"

Notmuch options

The following lists the notmuch options that alot reads.

search.exclude_tags

> A list of tags that will be excluded from search results by default. Using an excluded tag in a query will override that exclusion.

Type semicolon separated list

Default empty list

Accounts

In order to be able to send mails, you have to define at least one account subsection in your config: There needs to be a section "accounts", and each subsection, indicated by double square brackets defines an account.

Here is an example configuration

```
[accounts]
    [[work]]
        realname = Bruce Wayne
        address = b.wayne@wayneenterprises.com
        alias_regexp = b.wayne\+.+@wayneenterprises.com
        gpg_key = D7D6C5AA
        sendmail_command = msmtp --account=wayne -t
        sent_box = maildir:///home/bruce/mail/work/Sent
        draft_box = maildir:///home/bruce/mail/work/Drafts

    [[secret]]
        realname = Batman
        address = batman@batcave.org
        aliases = batman@batmobile.org,
        sendmail_command = msmtp --account=batman -t
        signature = ~/.batman.vcf
        signature_as_attachment = True
```

> **Warning:** Sending mails is only supported via a sendmail shell command for now. If you want to use a sendmail command different from *sendmail -t*, specify it as *sendmail_command*.

The following entries are interpreted at the moment:

address

> your main email address
>
> **Type** string

realname

> used to format the (proposed) From-header in outgoing mails
>
> **Type** string

aliases

> used to clear your addresses/ match account when formatting replies
>
> **Type** string list
>
> **Default** ,

alias_regexp

a regex for catching further aliases (like + extensions).

> **Type** string
>
> **Default** None

sendmail_command

sendmail command. This is the shell command used to send out mails via the sendmail protocol

> **Type** string
>
> **Default** "sendmail -t"

sent_box

where to store outgoing mails, e.g. *maildir:///home/you/mail/Sent*. You can use mbox, maildir, mh, babyl and mmdf in the protocol part of the URL.

Note: If you want to add outgoing mails automatically to the notmuch index you must use maildir in a path within your notmuch database path.

> **Type** mail_container
>
> **Default** None

draft_box

where to store draft mails, e.g. *maildir:///home/you/mail/Drafts*. You can use mbox, maildir, mh, babyl and mmdf in the protocol part of the URL.

Note: You will most likely want drafts indexed by notmuch to be able to later access them within alot. This currently only works for maildir containers in a path below your notmuch database path.

> **Type** mail_container
>
> **Default** None

sent_tags

list of tags to automatically add to outgoing messages

> **Type** string list
>
> **Default** sent,

signature

path to signature file that gets attached to all outgoing mails from this account, optionally renamed to *signature_filename*.

> **Type** string
>
> **Default** None

signature_as_attachment

---

attach signature file if set to True, append its content (mimetype text) to the body text if set to False.

> **Type** boolean
>
> **Default** False

signature_filename

> signature file's name as it appears in outgoing mails if *signature_as_attachment* is set to True
>
> **Type** string
>
> **Default** None

sign_by_default

> Outgoing messages will be GPG signed by default if this is set to True.
>
> **Type** boolean
>
> **Default** False

encrypt_by_default

> Alot will try to GPG encrypt outgoing messages by default when this is set to *all* or *trusted*. If set to *all* the message will be encrypted for all recipients for who a key is available in the key ring. If set to *trusted* it will be encrypted to all recipients if a trusted key is available for all recipients (one where the user id for the key is signed with a trusted signature).
>
> ---
> **Note:** If the message will not be encrypted by default you can still use the *toggleencrypt*, *encrypt* and *unencrypt* commands to encrypt it.
>
> ---
>
> ---
> **Note:** The values *True* and *False* are interpreted as *all* and *none* respectively. They are kept for backwards compatibility to give users a change to migrate to the new option type. They might become deprecated in future versions.
>
> ---
>
> **Type** option, one of ['all', 'none', 'trusted', 'True', 'False', 'true', 'false', 'Yes', 'No', 'yes', 'no', '1', '0']
>
> **Default** none

gpg_key

> The GPG key ID you want to use with this account. If unset, alot will use your default key.
>
> **Type** string
>
> **Default** None

Contacts Completion

For each *account* you can define an address book by providing a subsection named *abook*. Crucially, this section needs an option *type* that specifies the type of the address book. The only types supported at the moment are "shellcommand" and "abook". Both respect the *ignorecase* option which defaults to *True* and results in case insensitive lookups.

---

shellcommand

Address books of this type use a shell command in combination with a regular expression to look up contacts.

The value of *command* will be called with the search prefix as only argument for lookups. Its output is searched for email-name pairs using the regular expression given as *regexp*, which must include named groups "email" and "name" to match the email address and realname parts respectively. See below for an example that uses abook

```
[accounts]
  [[youraccount]]
      # ...
      [[[abook]]]
          type = shellcommand
          command = abook --mutt-query
          regexp = '^(?P<email>[^@]+@[^\t]+)\t+(?P<name>[^\t]+)'
          ignorecase = True
```

See here for alternative lookup commands. The few others I have tested so far are:

goobook for cached google contacts lookups. Works with the above default regexp

```
command = goobook query
regexp = '^(?P<email>[^@]+@[^\t]+)\t+(?P<name>[^\t]+)'
```

nottoomuch-addresses completes contacts found in the notmuch index:

```
command = nottoomuch-addresses.sh
regexp = \"(?P<name>.+)\"\s*<(?P<email>.*.+?@.+?)>
```

notmuch-abook completes contacts found in database of notmuch-abook:

```
command = notmuch_abook.py lookup
regexp = ^((?P<name>[^(\\s+\<)]*)\s+<)?(?P<email>[^@]+?@[^>]+)>?$
```

notmuch address Since version *0.19*, notmuch itself offers a subcommand *address*, that returns email addresses found in the notmuch index. Combined with the *date:* syntax to query for mails within a certain timeframe, this allows to search for all recently used contacts:

```
command = 'notmuch address --format=json --output=recipients date:1Y.. AND␣
→from:my@address.org'
regexp = '\[?{"name": "(?P<name>.*)", "address": "(?P<email>.+)", "name-addr
→": ".*"}[,\]]?'
shellcommand_external_filtering = False
```

Don't hesitate to send me your custom *regexp* values to list them here.

abook

Address books of this type directly parse abooks contact files. You may specify a path using the "abook_contacts_file" option, which defaults to `~/.abook/addressbook`. To use the default path, simply do this:

```
[accounts]
[[youraccount]]
    # ...
    [[[abook]]]
        type = abook
```

Key Bindings

If you want to bind a command to a key you can do so by adding the pair to the *[bindings]* section. This will introduce a *global* binding, that works in all modes. To make a binding specific to a mode you have to add the pair under the subsection named like the mode. For instance, if you want to bind *T* to open a new search for threads tagged with 'todo', and be able to toggle this tag in search mode, you'd add this to your config

```
[bindings]
  T = search tag:todo

  [[search]]
  t = toggletags todo
```

Known modes are:

- envelope

- search

- thread

- taglist

- bufferlist

Have a look at the urwid User Input documentation on how key strings are formatted.

Default bindings

User-defined bindings are combined with the default bindings listed below.

```
up = move up
down = move down
page up = move page up
page down = move page down
mouse press 4 = move up
mouse press 5 = move down
j = move down
k = move up
'g g' = move first
G = move last
' ' = move page down
'ctrl d' = move halfpage down
'ctrl u' = move halfpage up
@ = refresh
? = help bindings
I = search tag:inbox AND NOT tag:killed
'#' = taglist
shift tab = bprevious
U = search tag:unread
tab = bnext
\ = prompt 'search '
d = bclose
$ = flush
m = compose
o = prompt 'search '
q = exit
';' = bufferlist
```

```
':' = prompt
. = repeat

[bufferlist]
    x = close
    enter = open

[search]
    enter = select
    a = toggletags inbox
    & = toggletags killed
    ! = toggletags flagged
    s = toggletags unread
    l = retagprompt
    O = refineprompt
    | = refineprompt

[envelope]
    a = prompt 'attach ~/'
    y = send
    P = save
    s = 'refine Subject'
    f = prompt 'set From '
    t = 'refine To'
    b = 'refine Bcc'
    c = 'refine Cc'
    S = togglesign
    enter = edit
    'g f' = togglesource

[taglist]
    enter = select

[thread]
    enter = select
    C = fold *
    E = unfold *
    c = fold
    e = unfold
    < = fold
    > = unfold
    'g f' = togglesource
    H = toggleheaders
    P = print --all --separately --add_tags
    S = save --all
    g = reply --all
    f = forward
    p = print --add_tags
    n = editnew
    b= bounce
    s = save
    r = reply
    | = prompt 'pipeto '

    'g j' = move next sibling
    'g k' = move previous sibling
    'g h' = move parent
    'g l' = move first reply
```

```
    ' ' = move next
```

In prompts the following hardcoded bindings are available.

Key	Function
Ctrl-f/b	Moves the curser one character to the right/left
Alt-f/b Shift-right/left	Moves the cursor one word to the right/left
Ctrl-a/e	Moves the curser to the beginning/end of the line
Ctrl-d	Deletes the character under the cursor
Alt-d	Deletes everything from the cursor to the end of the current or next word
Alt-Delete/Backspace Ctrl-w	Deletes everything from the cursor to the beginning of the current or previous word
Ctrl-k	Deletes everything from the cursor to the end of the line
Ctrl-u	Deletes everything from the cursor to the beginning of the line

Overwriting defaults

To disable a global binding you can redefine it in your config to point to an empty command string. For example, to add a new global binding for key *a*, which is bound to *toggletags inbox* in search mode by default, you can remap it as follows.

```
[bindings]
  a = NEW GLOBAL COMMAND

  [[search]]
    a =
```

If you omit the last two lines, *a* will still be bound to the default binding in search mode.

Hooks

Hooks are python callables that live in a module specified by *hooksfile* in the config. Per default this points to ~/.config/alot/hooks.py.

Pre/Post Command Hooks

For every *COMMAND* in mode *MODE*, the callables pre_MODE_COMMAND() and post_MODE_COMMAND() – if defined – will be called before and after the command is applied respectively. In addition callables pre_global_COMMAND() and post_global_COMMAND() can be used. They will be called if no specific hook function for a mode is defined. The signature for the pre-*send* hook in envelope mode for example looks like this:

pre_envelope_send(*ui=None, dbm=None, cmd=None*)

> **Parameters**
>
> - **ui** (`alot.ui.UI`) – the main user interface
> - **dbm** (`alot.db.manager.DBManager`) – a database manager
> - **cmd** (`alot.commands.Command`) – the Command instance that is being called

Consider this pre-hook for the exit command, that logs a personalized goodbye message:

```python
import logging
from alot.settings import settings
def pre_global_exit(**kwargs):
    accounts = settings.get_accounts()
    if accounts:
        logging.info('goodbye, %s!' % accounts[0].realname)
    else:
        logging.info('goodbye!')
```

Other Hooks

Apart from command pre- and posthooks, the following hooks will be interpreted:

reply_prefix(*realname*, *address*, *timestamp*[, *ui= None*, *dbm=None*])
> Is used to reformat the first indented line in a reply message. This defaults to 'Quoting %s (%s)n' % (realname, timestamp)' unless this hook is defined

>> **Parameters**

>>> - **realname** (*str*) – name or the original sender
>>> - **address** (*str*) – address of the sender
>>> - **timestamp** (*datetime.datetime*) – value of the Date header of the replied message

>> **Return type** string

forward_prefix(*realname*, *address*, *timestamp*[, *ui= None*, *dbm=None*])
> Is used to reformat the first indented line in a inline forwarded message. This defaults to 'Forwarded message from %s (%s)n' % (realname, timestamp)' if this hook is undefined

>> **Parameters**

>>> - **realname** (*str*) – name or the original sender
>>> - **address** (*str*) – address of the sender
>>> - **timestamp** (*datetime.datetime*) – value of the Date header of the replied message

>> **Return type** string

pre_edit_translate(*text*[, *ui= None*, *dbm=None*])
> Used to manipulate a message's text *before* the editor is called. The text might also contain some header lines, depending on the settings *edit_headers_whitelist* and *edit_header_blacklist*.

>> **Parameters** **text** (*str*) – text representation of mail as displayed in the interface and as sent to the editor

>> **Return type** str

post_edit_translate(*text*[, *ui= None*, *dbm=None*])
> used to manipulate a message's text *after* the editor is called, also see *pre_edit_translate*

>> **Parameters** **text** (*str*) – text representation of mail as displayed in the interface and as sent to the editor

>> **Return type** str

text_quote(*message*)
> used to transform a message into a quoted one

>> **Parameters** **message** (*str*) – message to be quoted

> > **Return type** str

timestamp_format(*timestamp*)
> represents given timestamp as string

> > **Parameters** **timestamp** (*datetime*) – timestamp to represent

> > **Return type** str

touch_external_cmdlist(*cmd*, *shell=shell*, *spawn=spawn*, *thread=thread*)
> used to change external commands according to given flags shortly before they are called.

> > **Parameters**

> > > - **cmd** (`list of str`) – command to be called
> > > - **shell** (*bool*) – is this to be interpreted by the shell?
> > > - **spawn** (*bool*) – should be spawned in new terminal/environment
> > > - **threads** – should be called in new thread

> > **Returns** triple of amended command list, shell and thread flags

> > **Return type** list of str, bool, bool

reply_subject(*subject*)
> used to reformat the subject header on reply

> > **Parameters** **subject** (*str*) – subject to reformat

> > **Return type** str

forward_subject(*subject*)
> used to reformat the subject header on forward

> > **Parameters** **subject** (*str*) – subject to reformat

> > **Return type** str

pre_buffer_open(*ui= None*, *dbm=None*, *buf=buf*)
> run before a new buffer is opened

> > **Parameters** **buf** (`alot.buffer.Buffer`) – buffer to open

post_buffer_open(*ui=None*, *dbm=None*, *buf=buf*)
> run after a new buffer is opened

> > **Parameters** **buf** (`alot.buffer.Buffer`) – buffer to open

pre_buffer_close(*ui=None*, *dbm=None*, *buf=buf*)
> run before a buffer is closed

> > **Parameters** **buf** (`alot.buffer.Buffer`) – buffer to open

post_buffer_close(*ui=None*, *dbm=None*, *buf=buf*, *success=success*)
> run after a buffer is closed

> > **Parameters**

> > > - **buf** (`alot.buffer.Buffer`) – buffer to open
> > > - **success** (`boolean`) – true if successfully closed buffer

pre_buffer_focus(*ui=None*, *dbm=None*, *buf=buf*)
> run before a buffer is focused

> > **Parameters** **buf** (`alot.buffer.Buffer`) – buffer to open

post_buffer_focus(*ui=None*, *dbm=None*, *buf=buf*, *success=success*)
    run after a buffer is focused

> ### Parameters
>
> - **buf** (`alot.buffer.Buffer`) – buffer to open
>
> - **success** (`boolean`) – true if successfully focused buffer

exit()
    run just before the program exits

sanitize_attachment_filename(*filename=None*, *prefix=''*, *suffix=''*)
    returns *prefix* and *suffix* for a sanitized filename to use while opening an attachment. The *prefix* and *suffix* are used to open a file named *prefix* + *XXXXXX* + *suffix* in a temporary directory.

> ### Parameters
>
> - **filename** (`str or None`) – filename provided in the email (can be None)
>
> - **prefix** (`str`) – prefix string as found on mailcap
>
> - **suffix** (`str`) – suffix string as found on mailcap
>
> **Returns** tuple of *prefix* and *suffix*
>
> **Return type** (str, str)

loop_hook(*ui=None*)
    Run on a period controlled by *_periodic_hook_frequency*

> **Parameters** **ui** (`alot.ui.UI`) – the main user interface

Theming

Alot can be run in 1, 16 or 256 colour mode. The requested mode is determined by the command-line parameter *-C* or read from option *colourmode* config value. The default is 256, which scales down depending on how many colours your terminal supports.

Most parts of the user interface can be individually coloured to your liking. To make it easier to switch between or share different such themes, they are defined in separate files (see below for the exact format). To specify the theme to use, set the *theme* config option to the name of a theme-file. A file by that name will be looked up in the path given by the *themes_dir* config setting which defaults to `~/.config/alot/themes/`. If the themes_dir is not present then the contents of $XDG_DATA_DIRS/alot/themes will be tried in order. This defaults to `/usr/local/share/alot/themes` and `/usr/share/alot/themes`, in that order. These locations are meant to be used by distro packages to put themes in.

Theme Files

contain a section for each *MODE* plus "help" for the bindings-help overlay and "global" for globally used themables like footer, prompt etc. Each such section defines colour *attributes* for the parts that can be themed. The names of the themables should be self-explanatory. Have a look at the default theme file at `alot/defaults/default.theme` and the config spec `alot/defaults/default.theme` for the exact format.

Colour Attributes

Attributes are *sextuples* of urwid Attribute strings that specify foreground and background for mono, 16 and 256-colour modes respectively. For mono-mode only the flags *blink*, *standup*, *underline* and *bold* are available, 16c mode supports these in combination with the colour names:

```
brown      dark red      dark magenta    dark blue     dark cyan     dark green
yellow     light red     light magenta   light blue    light cyan    light green
black      dark gray     light gray      white
```

In high-colour mode, you may use the above plus grayscales *g0* to *g100* and colour codes given as *#* followed by three hex values. See here and here for more details on the interpreted values. A colour picker that makes choosing colours easy can be found in `alot/extra/colour_picker.py`.

As an example, check the setting below that makes the footer line appear as underlined bold red text on a bright green background:

```
[[global]]
  #name     mono fg      mono bg    16c fg                                 16c bg          256c fg
→              256c bg
  #          |           |        |                                      |               |
→                        |
  #          v           v        v                                      v               v
→              v
  footer = 'bold,underline', '', 'light red, bold, underline', 'light green', 'light
→red, bold, underline', '#8f6'
```

Highlighting Thread lines in Search Mode

The subsection '[[threadline]]' of the '[search]' section in *Theme Files* determines how to present a thread: here, *attributes* 'normal' and 'focus' provide fallback/spacer themes and 'parts' is a (string) list of displayed subwidgets. Possible part strings are:

- authors
- content
- date
- mailcount
- subject
- tags

For every listed part there must be a subsection with the same name, defining

normal *attribute* used for this part if unfocussed

focus *attribute* used for this part if focussed

width tuple indicating the width of the part. This is either *('fit', min, max)* to force the widget to be at least *min* and at most *max* characters wide, or *('weight', n)* which makes it share remaining space with other 'weight' parts.

alignment how to place the content string if the widget space is larger. This must be one of 'right', 'left' or 'center'.

To highlight some thread lines (use different attributes than the defaults found in the '[[threadline]]' section), one can define sections with prefix 'threadline'. Each one of those can redefine any part of the structure outlined above, the rest defaults to values defined in '[[threadline]]'.

The section used to theme a particular thread is the first one (in file-order) that matches the criteria defined by its 'query' and 'tagged_with' values:

- If 'query' is defined, the thread must match that querystring.

- If 'tagged_with' is defined, is value (string list) must be a subset of the accumulated tags of all messages in the thread.

Note: that 'tagged_with = A,B' is different from 'query = "is:A AND is:B"': the latter will match only if the thread contains a single message that is both tagged with A and B.

Moreover, note that if both query and tagged_with is undefined, this section will always match and thus overwrite the defaults.

The example below shows how to highlight unread threads: The date-part will be bold red if the thread has unread messages and flagged messages and just bold if the thread has unread but no flagged messages:

```
[search]
    # default threadline
    [[threadline]]
        normal = 'default','default','default','default','#6d6','default'
        focus = 'standout','default','light gray','dark gray','white','#68a'
        parts = date,mailcount,tags,authors,subject
        [[[date]]]
            normal = 'default','default','light gray','default','g58','default'
            focus = 'standout','default','light gray','dark gray','g89','#68a'
            width = 'fit',10,10
        # ...

    # highlight threads containing unread and flagged messages
    [[threadline-flagged-unread]]
        tagged_with = 'unread','flagged'
        [[[date]]]
            normal = 'default','default','light red,bold','default','light red,bold',
↪'default'

    # highlight threads containing unread messages
    [[threadline-unread]]
        query = 'is:unread'
        [[[date]]]
            normal = 'default','default','light gray,bold','default','g58,bold',
↪'default'
```

Custom Tagstring Formatting

One can specify how a particular tagstring is displayed throughout the interface. To use this feature, add a section *[tags]* to you alot config (not the theme file) and for each tag you want to customize, add a subsection named after the tag. Such a subsection may define

normal *attribute* used if unfocussed

focus *attribute* used if focussed

translated fixed string representation for this tag. The tag can be hidden from view, if the key *translated* is set to '', the empty string.

translation a pair of strings that define a regular substitution to compute the string representation on the fly using *re.sub*. This only really makes sense if one uses a regular expression to match more than one tagstring (see below).

The following will make alot display the "todo" tag as "TODO" in white on red.

```
[tags]
  [[todo]]
    normal = '','', 'white','light red', 'white','#d66'
    translated = TODO
```

Utf-8 symbols are welcome here, see e.g. http://panmental.de/symbols/info.htm for some fancy symbols. I personally display my maildir flags like this:

```
[tags]

  [[flagged]]
    translated =
    normal = '','','light red','','light red',''
    focus = '','','light red','','light red',''

  [[unread]]
    translated =

  [[replied]]
    translated =

  [[encrypted]]
    translated =
```

You may use regular expressions in the tagstring subsections to theme multiple tagstrings at once (first match wins). If you do so, you can use the *translation* option to specify a string substitution that will rename a matching tagstring. *translation* takes a comma separated *pair* of strings that will be fed to re.sub(). For instance, to theme all your nmbug tagstrings and especially colour tag *notmuch::bug* red, do the following:

```
[[notmuch::bug]]
  translated = 'nm:bug'
  normal = "", "", "light red, bold", "light blue", "light red, bold", "#88d"

[[notmuch::.*]]
  translation = 'notmuch::(.*)','nm:\1'
  normal = "", "", "white", "light blue", "#fff", "#88d"
```

API and Development

Overview

The main component is *alot.ui.UI*, which provides methods for user input and notifications, sets up the widget tree and maintains the list of active buffers. When you start up alot, `init.py` initializes logging, parses settings and commandline args and instantiates the *UI* instance of that gets passes around later. From its constructor this instance starts the `urwid mainloop` that takes over.

Apart from the central *UI*, there are two other "managers" responsible for core functionalities, also set up in `init.py`:

- *ui.dbman*: a `DBManager` to access the email database and
- `alot.settings.settings`: a `SettingsManager` oo access user settings

Every user action, triggered either by key bindings or via the command prompt, is given as commandline string that gets *translated* to a *Command* object which is then *applied*. Different actions are defined as a subclasses of *Command*, which live in `alot/commands/MODE.py`, where MODE is the name of the mode (*Buffer* type) they are used in.

Contributing

Getting Involved

Development is coordinated almost entirely via the projects github page especially the issue tracker. We also have an irc channel on freenode: #alot

Bugs and Feature Requests

Filing a Bug

Use the issue tracker on github to file bugs.

Before filing a bug please be sure that you've checked the following:

- That there is not already a bug filed
    - If there is a bug filed, you can follow that bug
    - Please refrain from adding "me too" comments unless the bug has been quiet for a while
- That you are using the most recent version of alot. If you are using an old version please update to the latest version before filing a bug

Once you've checked the above, please file a bug using the issue template.

Requesting a Feature

Feature requests are also filed via the issue tracker.

Before filing a feature request be sure to check the following:

- That the feature has not already been added to master or in a newer version
- That the feature has not already been requested. There is a feature tag for feature requests in the issue tracker, which is a good place to start.
- That the feature is in scope of the project

    Some examples of features that are not in scope:

    - Contact management, fetching email, sending email, indexing email. Alot relies on external services for all of these features. Integration with a new external service is okay.
- If the feature is requested already, please use the thumbs up emoji to add your support for the feature. Please do not add a "me too" comment.

Once you're sure that the feature isn't implemented or already requested, add a request. You will be given a template to fill out, but don't, that template is for bugs. Please be as thorough as possible when describing the feature, please explain what it does, how it should work, and how you plan to use the feature; if applicable.

Contributing

Before you Contribute

Alot is licensed under the GNU GPLv3+, and all code contributions will be covered by this license. You will retain your copyright on any code you write.

By contributing you have agreed that you have the right to contribute the code you have written, that the code you are contributing is owned by you or licensed with a compatible license allowing it to be added to alot, and that your employer or educational institution either does not or cannot claim the copyright on any code you write, or that they have given you a waiver to contribute to alot.

What to Contribute

- Bug fixes are always welcome

- Tests for bugs. If you can replicate a bug but don't want to or can't fix it we'll still take a unit test. Please decorate the test as an expected failure if it is failing.

- New features. Please be aware that we won't take every feature proposed, especially those that expand the scope of the project.

- Documentation. Including typos, spelling and grammar, wrong type and parameter annotations, or documentation that has gotten out of sync with the code.

Sending a Pull Request

The best way to send new code is with a pull request using the github interface.

- Follow **PEP 8**. This means in particular a maximum linewidth of *79* and no trailing white spaces. If in doubt, use an Automatic tool ([0], [1], [2]) to verify your code.

- Document! Needless to say, we want readable and well documented code. Moreover,

    - use sphinx directives to document the parameters and return values of your methods so that we maintain up-to-date API docs.

    - If you implemented a new feature, update the user manual in */docs/source/usage* accordingly.

    - If you implement a new feature or fix a major bug, add it to the NEWS file

- If you close an issue, but sure to use a "fixes" tag to automatically close the issue.

- Alot is currently python 2.7 only, but transitioning to python 3.x, please do not use constructs that do not map to python 3

- Make sure you don't regress any unit tests. They are implemented with the builtin unittest module, and can be run with *python setup.py test*, or with your favorite test runner such as pytest or nose.

- If you are fixing a bug or adding a new features, please include unit tests.

- Your patch will be automatically checked in our CI system, which will build docs and run unit tests. If any of these fail merging will be blocked until they are fixed.

Email Database

The python bindings to libnotmuch define `notmuch.Thread` and `notmuch.Message`, which unfortunately are very fragile. Alot defines the wrapper classes *alot.db.Thread* and *alot.db.Message* that use an *manager. DBManager* instance to transparently provide persistent objects.

alot.db.Message moreover contains convenience methods to extract information about the message like reformated header values, a summary, decoded and interpreted body text and a list of *Attachments*.

The central *UI* instance carries around a *DBManager* object that is used for any lookups or modifications of the email base. *DBManager* can directly look up *Thread* and *Message* objects and is able to postpone/cache/retry writing operations in case the Xapian index is locked by another process.

Database Manager

class `alot.db.manager.`**`DBManager`**(*path=None*, *ro=False*)
    Keeps track of your index parameters, maintains a write-queue and lets you look up threads and messages directly to the persistent wrapper classes.

        Parameters

- **path** (`str`) – absolute path to the notmuch index
- **ro** (`bool`) – open the index in read-only mode

add_message(*path*, *tags=None*, *afterwards=None*)
 Adds a file to the notmuch index.

   Parameters

   - **path** (`str`) – path to the file
   - **tags** (`list of str`) – tagstrings to add
   - **afterwards** (`callable or None`) – callback to trigger after adding

async(*cbl*, *fun*)
 return a pair (pipe, process) so that the process writes *fun(a)* to the pipe for each element *a* in the iterable returned by the callable *cbl*.

   Parameters

   - **cbl** (`callable`) – a function returning something iterable
   - **fun** (`callable`) – an unary translation function

   Return type (`multiprocessing.Pipe`, `multiprocessing.Process`)

count_messages(*querystring*)
 returns number of messages that match *querystring*

count_threads(*querystring*)
 returns number of threads that match *querystring*

flush()
 write out all queued write-commands in order, each one in a separate `atomic` transaction.

   If this fails the current action is rolled back, stays in the write queue and an exception is raised. You are responsible to retry flushing at a later time if you want to ensure that the cached changes are applied to the database.

   Exception `DatabaseROError` if db is opened read-only

   Exception `DatabaseLockedError` if db is locked

get_all_tags()
 returns all tagsstrings used in the database :rtype: list of str

get_message(*mid*)
 returns `Message` with given message id (str)

get_thread(*tid*)
 returns `Thread` with given thread id (str)

get_threads(*querystring*, *sort='newest_first'*, *exclude_tags=None*)
 asynchronously look up thread ids matching *querystring*.

   Parameters

   - **querystring** (`str.`) – The query string to use for the lookup
   - **sort** – Sort order. one of ['oldest_first', 'newest_first', 'message_id', 'unsorted']
   - **exclude_tags** (`list of str`) – Tags to exclude by default unless included in the search

   Returns a pipe together with the process that asynchronously writes to it.

> **Return type** (multiprocessing.Pipe, `multiprocessing.Process`)

kill_search_processes()
>   terminate all search processes that originate from this managers `get_threads()`.

query(*querystring*)
>   creates `notmuch.Query` objects on demand

>   **Parameters** **querystring** – The query string to use for the lookup

>   **Returns** `notmuch.Query` – the query object.

remove_message(*message*, *afterwards=None*)
>   Remove a message from the notmuch index

>   **Parameters**

>   - **message** (*[Message](#)*) – message to remove

>   - **afterwards** (*[callable or None](#)*) – callback to trigger after removing

tag(*querystring*, *tags*, *afterwards=None*, *remove_rest=False*)
>   add tags to messages matching *querystring*. This appends a tag operation to the write queue and raises
>   *[DatabaseROError](#)* if in read only mode.

>   **Parameters**

>   - **querystring** (*[str](#)*) – notmuch search string

>   - **tags** (*list of str*) – a list of tags to be added

>   - **afterwards** (*[callable](#)*) – callback that gets called after successful application of this
>   tagging operation

>   - **remove_rest** (*[bool](#)*) – remove tags from matching messages before tagging

>   **Exception** *[DatabaseROError](#)*

>   ---

>   **Note:** This only adds the requested operation to the write queue. You need to call `DBManager.`
>   `flush()` to actually write out.

>   ---

untag(*querystring*, *tags*, *afterwards=None*)
>   removes tags from messages that match *querystring*. This appends an untag operation to the write queue
>   and raises *[DatabaseROError](#)* if in read only mode.

>   **Parameters**

>   - **querystring** (*[str](#)*) – notmuch search string

>   - **tags** (*list of str*) – a list of tags to be added

>   - **afterwards** (*[callable](#)*) – callback that gets called after successful application of this
>   tagging operation

>   **Exception** *[DatabaseROError](#)*

>   ---

>   **Note:** This only adds the requested operation to the write queue. You need to call `DBManager.`
>   `flush()` to actually write out.

>   ---

Errors

class `alot.db.errors.`**`DatabaseError`**

class `alot.db.errors.`**`DatabaseROError`**
   cannot write to read-only database

class `alot.db.errors.`**`DatabaseLockedError`**
   cannot write to locked index

class `alot.db.errors.`**`NonexistantObjectError`**
   requested thread or message does not exist in the index

Wrapper

class `alot.db.`**`Thread`**(*dbman*, *thread*)
   A wrapper around a notmuch mailthread (`notmuch.database.Thread`) that ensures persistence of the
   thread: It can be safely read multiple times, its manipulation is done via a `alot.db.DBManager` and it can
   directly provide contained messages as `Message`.

   > **Parameters**
   >
   > - **dbman** (`DBManager`) – db manager that is used for further lookups
   >
   > - **thread** (`notmuch.database.Thread`) – the wrapped thread

   `add_tags`(*tags*, *afterwards=None*, *remove_rest=False*)
   add *tags* to all messages in this thread

   Note: This only adds the requested operation to this objects `DBManager`'s write queue. You need to
   call `DBManager.flush` to actually write out.

   > **Parameters**
   >
   > - **tags** (`list of str`) – a list of tags to be added
   >
   > - **afterwards** (*callable*) – callback that gets called after successful application of this
   >   tagging operation
   >
   > - **remove_rest** (*bool*) – remove all other tags

   `get_authors`()
   returns a list of authors (name, addr) of the messages. The authors are ordered by msg date and unique (by
   name/addr).

   > **Return type** list of (str, str)

   `get_authors_string`(*own_addrs=None*, *replace_own=None*)
   returns a string of comma-separated authors Depending on settings, it will substitute "me" for author name
   if address is user's own.

   > **Parameters**
   >
   > - **own_addrs** (`list of str`) – list of own email addresses to replace
   >
   > - **replace_own** (*bool*) – whether or not to actually do replacement

   > **Return type** str

---

get_messages()
>   returns all messages in this thread as dict mapping all contained messages to their direct responses.

>>   **Return type** dict mapping `Message` to a list of `Message`.

get_newest_date()
>   returns date header of newest message in this thread as `datetime`

get_oldest_date()
>   returns date header of oldest message in this thread as `datetime`

get_replies_to(*msg*)
>   returns all replies to the given message contained in this thread.

>>   **Parameters** **msg** (`Message`) – parent message to look up

>>   **Returns** list of `Message` or *None*

get_subject()
>   returns subject string

get_tags(*intersection=False*)
>   returns tagsstrings attached to this thread

>>   **Parameters** **intersection** (`bool`) – return tags present in all contained messages instead of in at least one (union)

>>   **Return type** set of str

get_thread_id()
>   returns id of this thread

get_toplevel_messages()
>   returns all toplevel messages contained in this thread. This are all the messages without a parent message (identified by 'in-reply-to' or 'references' header.

>>   **Return type** list of `Message`

get_total_messages()
>   returns number of contained messages

matches(*query*)
>   Check if this thread matches the given notmuch query.

>>   **Parameters** **query** (`string`) – The query to check against

>>   **Returns** True if this thread matches the given query, False otherwise

>>   **Return type** bool

refresh(*thread=None*)
>   refresh thread metadata from the index

remove_tags(*tags*, *afterwards=None*)
>   remove *tags* (list of str) from all messages in this thread

---

> **Note:** This only adds the requested operation to this objects `DBManager's` write queue. You need to call `DBManager.flush` to actually write out.

>>   **Parameters**

>>   • **tags** (*list of str*) – a list of tags to be added

- **afterwards** (*callable*) – callback that gets called after successful application of this tagging operation

class alot.db.**Message**(*dbman*, *msg*, *thread=None*)

a persistent notmuch message object. It it uses a DBManager for cached manipulation and lazy lookups.

> **Parameters**
>
> - **dbman** (*alot.db.DBManager*) – db manager that is used for further lookups
> - **msg** (*notmuch.database.Message*) – the wrapped message
> - **thread** (*Thread* or *None*) – this messages thread (will be looked up later if *None*)

accumulate_body()

returns bodystring extracted from this mail

add_tags(*tags*, *afterwards=None*, *remove_rest=False*)

adds tags to message

Note: This only adds the requested operation to this objects DBManager's write queue. You need to call flush() to write out.

> **Parameters**
>
> - **tags** (*list of str*) – a list of tags to be added
> - **afterwards** (*callable*) – callback that gets called after successful application of this tagging operation
> - **remove_rest** (*bool*) – remove all other tags

get_attachments()

returns messages attachments

Derived from the leaves of the email mime tree that and are not part of **RFC 2015** syntax for encrypted/signed mails and either have *Content-Disposition attachment* or have *Content-Disposition inline* but specify a filename (as parameter to *Content-Disposition*).

> **Return type** list of Attachment

get_author()

returns realname and address of this messages author

> **Return type** (str,str)

get_date()

returns Date header value as datetime

get_datestring()

returns reformated datestring for this message.

It uses SettingsManager.represent_datetime() to represent this messages *Date* header

> **Return type** str

get_email()

returns email.Message for this message

get_filename()

returns absolute path of message files location

---

get_message_id()
    returns messages id (str)

get_message_parts()
    yield all body parts of this message

get_replies()
    returns replies to this message as list of Message

get_tags()
    returns tags attached to this message as list of strings

get_thread()
    returns the *Thread* this msg belongs to

get_thread_id()
    returns id (str) of the thread this message belongs to

has_replies()
    returns true if this message has at least one reply

matches(*querystring*)
    tests if this messages is in the resultset for *querystring*

remove_tags(*tags*, *afterwards=None*)
    remove tags from message

Note: This only adds the requested operation to this objects DBManager's write queue. You need to call flush() to actually out.

Parameters

- **tags** (*list of str*) – a list of tags to be added
- **afterwards** (*callable*) – callback that gets called after successful application of this tagging operation

Other Structures

class alot.db.attachment.**Attachment**(*emailpart*)
    represents a mail attachment

        Parameters emailpart (email.message.Message) – a non-multipart email that is the attachment

get_content_type()
    mime type of the attachment part

get_data()
    return data blob from wrapped file

get_filename()
    return name of attached file. If the content-disposition header contains no file name, this returns *None*

get_mime_representation()
    returns mime part that constitutes this attachment

get_size()
    returns attachments size in bytes

---

save (*path*)
> save the attachment to disk. Uses `get_filename()` in case path is a directory

write (*fhandle*)
> writes content to a given filehandle

class `alot.db.envelope.`**`Envelope`** (*template=None*, *bodytext=None*, *headers=None*, *attach-ments=None*, *sign=False*, *sign_key=None*, *encrypt=False*, *tags=None*)

a message that is not yet sent and still editable. It holds references to unencoded! body text and mail headers among other things. Envelope implements the python container API for easy access of header values. So *e['To']*, *e['To'] = 'foo@bar.baz'* and 'e.get_all('To')' would work for an envelope *e.*.

> **Parameters**
>
> - **template** (*str*) – if not None, the envelope will be initialised by `parsing` this string before setting any other values given to this constructor.
>
> - **bodytext** (*str*) – text used as body part
>
> - **headers** (*dict (str -> [unicode])*) – unencoded header values
>
> - **attachments** (list of *Attachment*) – file attachments to include
>
> - **tags** (*list of str*) – tags to add after successful sendout and saving this msg

add (*key*, *value*)
> add header value

attach (*attachment*, *filename=None*, *ctype=None*)
> attach a file
>
> > **Parameters**
> >
> > - **attachment** (*Attachment* or str) – File to attach, given as *Attachment* object or path to a file.
> >
> > - **filename** – filename to use in content-disposition. Will be ignored if *path* matches multiple files
> >
> > - **ctype** (*str*) – force content-type to be used for this attachment

construct_mail ()
> compiles the information contained in this envelope into a `email.Message`.

get (*key*, *fallback=None*)
> secure getter for header values that allows specifying a *fallback* return string (defaults to None). This returns the first matching value and doesn't raise KeyErrors

get_all (*key*, *fallback=None*)
> returns all header values for given key

parse_template (*tmp*, *reset=False*, *only_body=False*)
> parses a template or user edited string to fills this envelope.
>
> > **Parameters**
> >
> > - **tmp** (*str*) – the string to parse.
> >
> > - **reset** (*bool*) – remove previous envelope content

attachments = None
> list of *Attachments*

body = None
> mail body as unicode string

> **headers = None**
>> dict containing the mail headers (a list of strings for each header key)
>
> **tags = []**
>> tags to add after successful sendout
>
> **tmpfile = None**
>> template text for initial content

Utilities

alot.db.utils.**add_signature_headers**(*mail*, *sigs*, *error_msg*)
> Add pseudo headers to the mail indicating whether the signature verification was successful.
>
>> **Parameters**
>>
>>> - **mail** – `email.message.Message` the message to entitle
>>>
>>> - **sigs** – list of `gpg.results.Signature`
>>>
>>> - **error_msg** – *str* containing an error message, the empty string indicating no error

alot.db.utils.**decode_header**(*header*, *normalize=False*)
> decode a header value to a unicode string
>
> values are usually a mixture of different substrings encoded in quoted printable using different encodings. This turns it into a single unicode string
>
>> **Parameters**
>>
>>> - **header** (`str`) – the header value
>>>
>>> - **normalize** (`bool`) – replace trailing spaces after newlines
>>
>> **Return type** unicode

alot.db.utils.**encode_header**(*key*, *value*)
> encodes a unicode string as a valid header value
>
>> **Parameters**
>>
>>> - **key** (`str`) – the header field this value will be stored in
>>>
>>> - **value** (`unicode`) – the value to be encoded

alot.db.utils.**extract_body**(*mail*, *types=None*, *field_key='copiousoutput'*)
> returns a body text string for given mail. If types is *None*, *text/** is used: The exact preferred type is specified by the prefer_plaintext config option which defaults to text/html.
>
>> **Parameters**
>>
>>> - **mail** (`email.Message`) – the mail to use
>>>
>>> - **types** (`list of str`) – mime content types to use for body string

alot.db.utils.**extract_headers**(*mail*, *headers=None*)
> returns subset of this messages headers as human-readable format: all header values are decoded, the resulting string has one line "KEY: VALUE" for each requested header present in the mail.
>
>> **Parameters**
>>
>>> - **mail** (`email.Message`) – the mail to use
>>>
>>> - **headers** (`list of str`) – headers to extract

alot.db.utils.**get_params**(*mail*, *failobj=None*, *header='content-type'*, *unquote=True*)
>   Get Content-Type parameters as dict.
>
>   RFC 2045 specifies that parameter names are case-insensitive, so we normalize them here.
>
>>   **Parameters**
>>
>>   - **mail** – `email.message.Message`
>>
>>   - **failobj** – object to return if no such header is found
>>
>>   - **header** – the header to search for parameters, default
>>
>>   - **unquote** – unquote the values
>>
>>   **Returns** a *dict* containing the parameters

alot.db.utils.**message_from_file**(*handle*)
>   Reads a mail from the given file-like object and returns an email object, very much like email.message_from_file. In addition to that OpenPGP encrypted data is detected and decrypted. If this succeeds, any mime messages found in the recovered plaintext message are added to the returned message object.
>
>>   **Parameters handle** – a file-like object
>>
>>   **Returns** `email.message.Message` possibly augmented with decrypted data

alot.db.utils.**message_from_string**(*s*)
>   Reads a mail from the given string. This is the equivalent of `email.message_from_string()` which does nothing but to wrap the given string in a StringIO object and to call `email.message_from_file()`.
>
>   Please refer to the documentation of *message_from_file()* for details.

User Interface

Alot sets up a widget tree and a `mainloop` in the constructor of *alot.ui.UI*. The visible area is a `urwid.Frame`, where the footer is used as a status line and the body part displays the currently active *alot.buffers.Buffer*.

To be able to bind keystrokes and translate them to *Commands*, keypresses are *not* propagated down the widget tree as is customary in urwid. Instead, the root widget given to urwids mainloop is a custom wrapper (`alot.ui.Inputwrap`) that interprets key presses. A dedicated `SendKeypressCommand` can be used to trigger key presses to the wrapped root widget and thereby accessing standard urwid behaviour.

In order to keep the interface non-blocking and react to events like terminal size changes, alot makes use of twisted's deferred - a framework that makes it easy to deal with callbacks. Many commands in alot make use of inline callbacks, which allow you to treat deferred-returning functions almost like syncronous functions. Consider the following example of a function that prompts for some input and acts on it:

```python
from twisted.internet import defer


@defer.inlineCallbacks
def greet(ui):  # ui is instance of alot.ui.UI
    name = yield ui.prompt('pls enter your name')
    ui.notify('your name is: ' + name)
```

UI - the main component

class alot.ui.**UI**(*dbman*, *initialcmdline*)
>   This class integrates all components of alot and offers methods for user interaction like *prompt()*, *notify()*

etc. It handles the urwid widget tree and mainloop (we use twisted) and is responsible for opening, closing and focussing buffers.

> **Parameters**
>
> - **dbman** – DBManager
> - **initialcmdline** (*str*) – commandline applied after setting up interface
> - **colourmode** (*int in [1,16,256]*) – determines which theme to chose

apply_command (*cmd*)
> applies a command
>
> This calls the pre and post hooks attached to the command, as well as `cmd.apply()`.
>
> > **Parameters cmd** (*Command*) – an applicable command

apply_commandline (*cmdline*)
> interprets a command line string
>
> i.e., splits it into separate command strings, instanciates *Commands* accordingly and applies then in sequence.
>
> > **Parameters cmdline** (*str*) – command line to interpret

buffer_close (*buf*, *redraw=True*)
> closes given *Buffer*.
>
> This it removes it from the bufferlist and calls its cleanup() method.

buffer_focus (*buf*, *redraw=True*)
> focus given *Buffer*.

buffer_open (*buf*)
> register and focus new *Buffer*.

build_statusbar ()
> construct and return statusbar widget

choice (*message*, *choices=None*, *select=None*, *cancel=None*, *msg_position='above'*)
> prompt user to make a choice.
>
> > **Parameters**
> >
> > - **message** (*unicode*) – string to display before list of choices
> > - **choices** (*dict: keymap->choice (both str)*) – dict of possible choices
> > - **select** (*str*) – choice to return if enter/return is hit. Ignored if set to *None*.
> > - **cancel** (*str*) – choice to return if escape is hit. Ignored if set to *None*.
> > - **msg_position** (*str*) – determines if *message* is above or left of the prompt. Must be *above* or *left*.
> >
> > **Return type** twisted.defer.Deferred

cleanup ()
> Do the final clean up before shutting down.

clear_notify (*messages*)
> Clears notification popups. Call this to ged rid of messages that don't time out.
>
> > **Parameters messages** – The popups to remove. This should be exactly what *notify()* returned when creating the popup

static **exit**()
> shuts down user interface without cleaning up. Use a *alot.commands.globals.ExitCommand* for a clean shutdown.

get_buffers_of_type(*t*)
> returns currently open buffers for a given subclass of *Buffer*.
>
> > **Parameters t** (`alot.buffers.Buffer`) – Buffer class
> >
> > **Return type** list

get_deep_focus(*startfrom=None*)
> return the bottom most focussed widget of the widget tree

handle_signal(*signum*, *frame*)
> handles UNIX signals
>
> This function currently just handles SIGUSR1. It could be extended to handle more
>
> > **Parameters**
> >
> > - **signum** – The signal number (see man 7 signal)
> >
> > - **frame** – The execution frame (https://docs.python.org/2/reference/datamodel.html#frame-objects)

notify(*message*, *priority='normal'*, *timeout=0*, *block=False*)
> opens notification popup.
>
> > **Parameters**
> >
> > - **message** (`str`) – message to print
> >
> > - **priority** (`str`) – priority string, used to format the popup: currently, 'normal' and 'error' are defined. If you use 'X' here, the attribute 'global_notify_X' is used to format the popup.
> >
> > - **timeout** (`int`) – seconds until message disappears. Defaults to the value of 'notify_timeout' in the general config section. A negative value means never time out.
> >
> > - **block** (`bool`) – this notification blocks until a keypress is made
> >
> > **Returns** an urwid widget (this notification) that can be handed to *clear_notify()* for removal

prompt(*prefix*, *text=u''*, *completer=None*, *tab=0*, *history=None*)
> prompt for text input. This returns a `Deferred` that calls back with the input string.
>
> > **Parameters**
> >
> > - **prefix** (`str`) – text to print before the input field
> >
> > - **text** (`str`) – initial content of the input field
> >
> > - **completer** (`alot.completion.Completer()`) – completion object to use
> >
> > - **tab** (`int`) – number of tabs to press initially (to select completion results)
> >
> > - **history** (`list of str`) – history to be used for up/down keys
> >
> > **Return type** `twisted.defer.Deferred`

show_as_root_until_keypress(*w*, *key*, *afterwards=None*)
> Replaces root widget by given `urwid.Widget` and makes the UI ignore all further commands apart from cursor movement. If later on *key* is pressed, the old root widget is reset, callable *afterwards* is called and normal behaviour is resumed.

update(*redraw=True*)
>   redraw interface

buffers = None
>   list of active buffers

commandprompthistory = None
>   history of the command line prompt

current_buffer = None
>   points to currently active *Buffer*

db_was_locked = None
>   flag used to prevent multiple 'index locked' notifications

dbman = None
>   Database Manager (*DBManager*)

input_queue = None
>   stores partial keyboard input

last_commandline = None
>   saves the last executed commandline

mode = None
>   interface mode identifier - type of current buffer

recipienthistory = None
>   history of the recipients prompt

senderhistory = None
>   history of the sender prompt

Buffers

A buffer defines a view to your data. It knows how to render itself, to interpret keypresses and is visible in the "body" part of the widget frame. Different modes are defined by subclasses of the following base class.

class `alot.buffers.`**Buffer**(*ui*, *widget*)
>   Abstract base class for buffers.

>   **cleanup**()
>   >   called before buffer is closed

>   **get_info**()
>   >   return dict of meta infos about this buffer. This can be requested to be displayed in the statusbar.

>   **rebuild**()
>   >   tells the buffer to (re)construct its visible content.

Available modes are:

Mode	Buffer Subclass
search	*SearchBuffer*
thread	*ThreadBuffer*
bufferlist	*BufferlistBuffer*
taglist	*TagListBuffer*
envelope	*EnvelopeBuffer*

class `alot.buffers.`**BufferlistBuffer**(*ui*, *filtfun=<function <lambda>>*)
>   lists all active buffers

focus_first()
> Focus the first line in the buffer list.

get_selected_buffer()
> returns currently selected *Buffer* element from list

index_of(*b*)
> returns the index of *Buffer* *b* in the global list of active buffers.

class alot.buffers.**EnvelopeBuffer**(*ui*, *envelope*)
> message composition mode

toggle_all_headers()
> toggles visibility of all envelope headers

class alot.buffers.**SearchBuffer**(*ui*, *initialquery=''*, *sort_order=None*)
> shows a result list of threads for a query

get_selected_thread()
> returns currently selected *Thread*

get_selected_threadline()
> returns curently focussed alot.widgets.ThreadlineWidget from the result list.

kill_filler_process()
> terminates the process that fills this buffers PipeWalker.

class alot.buffers.**ThreadBuffer**(*ui*, *thread*)
> displays a thread as a tree of messages

> **Parameters**
>> - **ui** (*UI*) – main UI
>>
>> - **thread** (*Thread*) – thread to display

collapse(*msgpos*)
> collapse message at given position

collapse_all()
> collapse all messages in thread

expand(*msgpos*)
> expand message at given position

expand_all()
> expand all messages in thread

focus_first()
> set focus to first message of thread

focus_first_reply()
> move focus to first reply to currently focussed message

focus_last_reply()
> move focus to last reply to currently focussed message

focus_next()
> focus next message in depth first order

focus_next_matching(*querystring*)
> focus next matching message in depth first order

focus_next_sibling()
> focus next sibling of currently focussed message in thread tree

focus_next_unfolded()
  focus next unfolded message in depth first order

focus_parent()
  move focus to parent of currently focussed message

focus_prev()
  focus previous message in depth first order

focus_prev_matching(*querystring*)
  focus previous matching message in depth first order

focus_prev_sibling()
  focus previous sibling of currently focussed message in thread tree

focus_prev_unfolded()
  focus previous unfolded message in depth first order

focus_property(*prop*, *direction*)
  does a walk in the given direction and focuses the first message tree that matches the given property

focus_selected_message()
  focus the summary line of currently focussed message

get_focus()
  Get the focus from the underlying body widget.

get_messagetree_positions()
  returns a Generator to walk through all positions of `MessageTree` in the `ThreadTree` of this buffer.

get_selected_message()
  returns focussed `Message`

get_selected_message_position()
  returns position of focussed message in the thread tree

get_selected_messagetree()
  returns currently focussed `MessageTree`

get_selected_mid()
  returns Message ID of focussed message

get_selected_thread()
  returns the displayed *Thread*

messagetree_at_position(*pos*)
  get `MessageTree` for given position

messagetrees()
  returns a Generator of all `MessageTree` in the `ThreadTree` of this buffer.

refresh()
  refresh and flushe caches of Thread tree

set_focus(*pos*)
  Set the focus in the underlying body widget.

unfold_matching(*querystring*, *focus_first=True*)
  expand all messages that match a given querystring.

  Parameters

  - **querystring** (*str*) – query to match

  - **focus_first** (*bool*) – set the focus to the first matching message

---

class `alot.buffers.`**`TagListBuffer`**(*ui*, *alltags=None*, *filtfun=<function <lambda>>*)
> lists all tagstrings present in the notmuch database

> **`focus_first`**()
> > Focus the first line in the tag list.

> **`get_selected_tag`**()
> > returns selected tagstring

Widgets

What follows is a list of the non-standard urwid widgets used in alot. Some of them respect *user settings*, themes in particular.

utils

Utility Widgets not specific to alot

class `alot.widgets.utils.`**`AttrFlipWidget`**(*w*, *maps*, *init_map='normal'*)
> An AttrMap that can remember attributes to set

globals

This contains alot-specific `urwid.Widget` used in more than one mode.

class `alot.widgets.globals.`**`AttachmentWidget`**(*attachment*, *selectable=True*)
> one-line summary of an `Attachment`.

class `alot.widgets.globals.`**`CompleteEdit`**(*completer*, *on_exit*, *on_error=None*, *edit_text=u''*, *history=None*, ***kwargs*)
> This is a vamped-up `urwid.Edit` widget that allows for tab-completion using `Completer` objects

> These widgets are meant to be used as user input prompts and hence react to 'return' key presses by calling a 'on_exit' callback that processes the current text value.

> **The interpretation of some keypresses is hard-wired:**

> > **enter** calls 'on_exit' callback with current value

> > **esc/ctrl g** calls 'on_exit' with value *None*, which can be interpreted as cancelation

> > **tab** calls the completer and tabs forward in the result list

> > **shift tab** tabs backward in the result list

> > **up/down** move in the local input history

> > **ctrl f/b** moves curser one character to the right/left

> > **meta f/b shift right/left** moves the cursor one word to the right/left

> > **ctrl a/e** moves curser to the beginning/end of the input

> > **ctrl d** deletes the character under the cursor

> > **meta d** deletes everything from the cursor to the end of the next word

> > **meta delete/backspace ctrl w** deletes everything from the cursor to the beginning of the current word

> > **ctrl k** deletes everything from the cursor to the end of the input

ctrl u deletes everything from the cursor to the beginning of the input

> **Parameters**
> - **completer** (`alot.completion.Completer`) – completer to use
> - **on_exit** (`callable`) – "enter"-callback that interprets the input (str)
> - **on_error** (`callback`) – callback that handles `alot.errors.CompletionErrors`
> - **edit_text** (`str`) – initial text
> - **history** (`list or str`) – initial command history

class `alot.widgets.globals.`**HeadersList**(*headerslist*, *key_attr*, *value_attr*, *gaps_attr=None*)
   renders a pile of header values as key/value list

> **Parameters**
> - **headerslist** (`list of (str, str)`) – list of key/value pairs to display
> - **key_attr** (`urwid.AttrSpec`) – theming attribute to use for keys
> - **value_attr** (`urwid.AttrSpec`) – theming attribute to use for values
> - **gaps_attr** (`urwid.AttrSpec`) – theming attribute to wrap lines in

class `alot.widgets.globals.`**TagWidget**(*tag*, *fallback_normal=None*, *fallback_focus=None*)
   text widget that renders a tagstring.

   It looks up the string it displays in the *tags* section of the config as well as custom theme settings for its tag.

   Atributes that should be considered publicly readable:

   > **attr tag** the notmuch tag
   >
   > **type tag** str

bufferlist

Widgets specific to Bufferlist mode

class `alot.widgets.bufferlist.`**BufferlineWidget**(*buffer*)
   selectable text widget that represents a *Buffer* in the *BufferlistBuffer*.

search

Widgets specific to search mode

class `alot.widgets.search.`**ThreadlineWidget**(*tid*, *dbman*)
   selectable line widget that represents a *Thread* in the *SearchBuffer*.

thread

Widgets specific to thread mode

class `alot.widgets.thread.`**`DictList`**(*content*, *key_attr*, *value_attr*, *gaps_attr=None*)
    `SimpleTree` that displays key-value pairs.

    The structure will obey the Tree API but will not actually be a tree but a flat list: It contains one top-level node (displaying the k/v pair in Columns) per pair. That is, the root will be the first pair, its sibblings will be the other pairs and first|last_child will always be None.

        Parameters

- **`headerslist`** (*list of (str, str)*) – list of key/value pairs to display
- **`key_attr`** (*urwid.AttrSpec*) – theming attribute to use for keys
- **`value_attr`** (*urwid.AttrSpec*) – theming attribute to use for values
- **`gaps_attr`** (*urwid.AttrSpec*) – theming attribute to wrap lines in

class `alot.widgets.thread.`**`FocusableText`**(*txt*, *att*, *att_focus*)
    Selectable Text used for nodes in our example

class `alot.widgets.thread.`**`MessageSummaryWidget`**(*message*, *even=True*)
    one line summary of a `Message`.

        Parameters

- **`message`** (`alot.db.Message`) – a message
- **`even`** (*bool*) – even entry in a pile of messages? Used for theming.

class `alot.widgets.thread.`**`MessageTree`**(*message*, *odd=True*)
    Tree that displays contents of a single `alot.db.Message`.

    Its root node is a `MessageSummaryWidget`, and its child nodes reflect the messages content (parts for headers/attachments etc).

    Collapsing this message corresponds to showing the summary only.

        Parameters

- **`message`** (`alot.db.Message`) – Message to display
- **`odd`** (*bool*) – theme summary widget as if this is an odd line (in the message-pile)

    `collapse_if_matches`(*querystring*)
        collapse (and show summary only) if the `alot.db.Message` matches given *querystring*

class `alot.widgets.thread.`**`ThreadTree`**(*thread*)
    Tree that parses a given `alot.db.Thread` into a tree of `MessageTrees` that display this threads individual messages. As MessageTreess are *not* urwid widgets themself this is to be used in combination with `NestedTree` only.

Completion

`alot.ui.UI.prompt()` allows tab completion using a `Completer` object handed as 'completer' parameter. `alot.completion` defines several subclasses for different occasions like completing email addresses from an `AddressBook`, notmuch tagstrings. Some of these actually build on top of each other; the `QueryCompleter` for example uses a `TagsCompleter` internally to allow tagstring completion after "is:" or "tag:" keywords when typing a notmuch querystring.

All these classes overide the method `complete()`, which for a given string and cursor position in that string returns a list of tuples *(completed_string, new_cursor_position)* that are taken to be the completed values. Note that *completed_string* does not need to have the original string as prefix. `complete()` may rise `alot.errors.CompletionError` exceptions.

class `alot.completion.`**`AbooksCompleter`**(*abooks*, *addressesonly=False*)

completes a contact from given address books

> **Parameters**
>
> - **abooks** (list of `AddresBook`) – used to look up email addresses
> - **addressesonly** ([`bool`](#)) – only insert address, not the realname of the contact

class `alot.completion.`**`AccountCompleter`**(***kwargs*)

completes users' own mailaddresses

class `alot.completion.`**`ArgparseOptionCompleter`**(*parser*)

completes option parameters for a given argparse.Parser

> **Parameters** **parser** (*argparse.ArgumentParser*) – the option parser we look up parameter and choices from

class `alot.completion.`**`CommandCompleter`**(*dbman*, *mode*, *currentbuffer=None*)

completes one command consisting of command name and parameters

> **Parameters**
>
> - **dbman** (`DBManager`) – used to look up avaliable tagstrings
> - **mode** ([`str`](#)) – mode identifier
> - **currentbuffer** ([`Buffer`](#)) – currently active buffer. If defined, this will be used to dynamically extract possible completion strings

class `alot.completion.`**`CommandLineCompleter`**(*dbman*, *mode*, *currentbuffer=None*)

completes command lines: semicolon separated command strings

> **Parameters**
>
> - **dbman** (`DBManager`) – used to look up avaliable tagstrings
> - **mode** ([`str`](#)) – mode identifier
> - **currentbuffer** ([`Buffer`](#)) – currently active buffer. If defined, this will be used to dynamically extract possible completion strings

> **static** **`get_context`**(*line*, *pos*)
>
> computes start and end position of substring of line that is the command string under given position

class `alot.completion.`**`CommandNameCompleter`**(*mode*)

completes command names

> **Parameters** **mode** ([`str`](#)) – mode identifier

class `alot.completion.`**`Completer`**

base class for completers

> **`complete`**(*original*, *pos*)
>
> returns a list of completions and cursor positions for the string original from position pos on.
>
> > **Parameters**
> >
> > - **original** ([`str`](#)) – the string to complete
> > - **pos** ([`int`](#)) – starting position to complete from
> >
> > **Returns** pairs of completed string and cursor position in the new string
> >
> > **Return type** list of ([str](#), [int](#))
> >
> > **Raises** `CompletionError`

relevant_part(*original*, *pos*, *sep=' '*)
> calculates the subword in a *sep*-splitted list of substrings of *original* that *pos* is ia.n

class `alot.completion.`**ContactsCompleter**(*abooks*, *addressesonly=False*)
> completes contacts from given address books

> > **Parameters**
> >
> > * **abooks** (list of `AddresBook`) – used to look up email addresses
> >
> > * **addressesonly** (`bool`) – only insert address, not the realname of the contact

class `alot.completion.`**CryptoKeyCompleter**(*private=False*)
> completion for gpg keys

> > **Parameters** **private** (`bool`) – return private keys

class `alot.completion.`**MultipleSelectionCompleter**(*completer*, *separator=', '*)
> Meta-Completer that turns any Completer into one that deals with a list of completion strings using the wrapped Completer. This allows for example to easily construct a completer for comma separated recipient-lists using a *ContactsCompleter*.

> > **Parameters**
> >
> > * **completer** (`Completer`) – completer to use for individual substrings
> >
> > * **separator** (`str`) – separator used to split the completion string into substrings to be fed to *completer*.

relevant_part(*original*, *pos*)
> calculates the subword of *original* that *pos* is in

class `alot.completion.`**PathCompleter**
> completion for paths

class `alot.completion.`**QueryCompleter**(*dbman*)
> completion for a notmuch query string

> > **Parameters** **dbman** (`DBManager`) – used to look up avaliable tagstrings

class `alot.completion.`**StringlistCompleter**(*resultlist*,   *ignorecase=True*,   *match_anywhere=False*)
> completer for a fixed list of strings

> > **Parameters**
> >
> > * **resultlist** (`list of str`) – strings used for completion
> >
> > * **liberal** (`bool`) – match case insensitive and not prefix-only

class `alot.completion.`**TagCompleter**(*dbman*)
> complete a tagstring

> > **Parameters** **dbman** (`DBManager`) – used to look up avaliable tagstrings

class `alot.completion.`**TagsCompleter**(*dbman*)
> completion for a comma separated list of tagstrings

> > **Parameters** **dbman** (`DBManager`) – used to look up avaliable tagstrings

User Settings

Alot sets up a *SettingsManager* to access user settings defined in different places uniformly. There are four types of user settings:

what?	location	accessible via
alot config	`~/.config/alot/config` or given by command option *-c*.	*SettingsManager. get()*
hooks – user provided python code	`~/.config/alot/hooks.py` or as given by the *hooksfile* config value	*SettingsManager. get_hook()*
notmuch config	`~/.notmuch-config` or given by *$NOTMUCH_CONFIG* or given by command option *-n*	*SettingsManager. get_notmuch_setting()*
mailcap – defines shellcommands to handle mime types	`~/.mailcap` (`/etc/mailcap`)	*SettingsManager. mailcap_find_match()*

Settings Manager

class `alot.settings.manager.`**`SettingsManager`**(*alot_rc=None*, *notmuch_rc=None*)
Organizes user settings

> **Parameters**
>
> - **`alot_rc`** (*str*) – path to alot's config file
> - **`notmuch_rc`** (*str*) – path to notmuch's config file

`get`(*key*, *fallback=None*)
look up global config values from alot's config

> **Parameters**
>
> - **`key`** (*str*) – key to look up
> - **`fallback`** (*str*) – fallback returned if key is not present
>
> **Returns** config value with type as specified in the spec-file

`get_account_by_address`(*address*, *return_default=False*)
returns `Account` for a given email address (str)

> **Parameters**
>
> - **`address`** (*str*) – address to look up. A realname part will be ignored.
> - **`return_default`** (*bool*) – If True and no address can be found, then the default account wil be returned
>
> **Return type** `Account`
>
> **Raises** *NoMatchingAccount* – If no account can be found. Thsi includes if return_default is True and there are no accounts defined.

`get_accounts`()
returns known accounts

> **Return type** list of `Account`

`get_addressbooks`(*order=None*, *append_remaining=True*)
returns list of all defined `AddressBook` objects

get_addresses()
>   returns addresses of known accounts including all their aliases

get_hook(*key*)
>   return hook (*callable*) identified by *key*

get_keybinding(*mode*, *key*)
>   look up keybinding from *MODE-maps* sections

>   >   **Parameters**

>   >   >   • **mode** ([*str*](#)) – mode identifier

>   >   >   • **key** ([*str*](#)) – urwid-style key identifier

>   >   **Returns** a command line to be applied upon keypress

>   >   **Return type** str

get_keybindings(*mode*)
>   look up keybindings from *MODE-maps* sections

>   >   **Parameters** **mode** ([*str*](#)) – mode identifier

>   >   **Returns** dictionaries of key-cmd for global and specific mode

>   >   **Return type** 2-tuple of dicts

get_main_addresses()
>   returns addresses of known accounts without its aliases

get_notmuch_setting(*section*, *key*, *fallback=None*)
>   look up config values from notmuch's config

>   >   **Parameters**

>   >   >   • **section** ([*str*](#)) – key is in

>   >   >   • **key** ([*str*](#)) – key to look up

>   >   >   • **fallback** ([*str*](#)) – fallback returned if key is not present

>   >   **Returns** config value with type as specified in the spec-file

get_tagstring_representation(*tag*, *onebelow_normal=None*, *onebelow_focus=None*)
>   looks up user's preferred way to represent a given tagstring.

>   >   **Parameters**

>   >   >   • **tag** ([*str*](#)) – tagstring

>   >   >   • **onebelow_normal** ([*urwid.AttrSpec*](#)) – attribute that shines through if unfocussed

>   >   >   • **onebelow_focus** ([*urwid.AttrSpec*](#)) – attribute that shines through if focussed

>   If *onebelow_normal* or *onebelow_focus* is given these attributes will be used as fallbacks for fg/bg values '' and 'default'.

>   **This returns a dictionary mapping**

>   >   **normal** to urwid.AttrSpec used if unfocussed

>   >   **focussed** to urwid.AttrSpec used if focussed

>   >   **translated** to an alternative string representation

get_theming_attribute(*mode*, *name*, *part=None*)
>   looks up theming attribute

Parameters

- **mode** (`str`) – ui-mode (e.g. *search*,'thread'...)

- **name** (`str`) – identifier of the atttribute

Return type urwid.AttrSpec

get_threadline_theming(*thread*)

looks up theming info a threadline displaying a given thread. This wraps around *get_threadline_theming()*, filling in the current colour mode.

Parameters thread(`alot.db.thread.Thread`) – thread to theme

mailcap_find_match(**args*, ***kwargs*)

Propagates `mailcap.find_match()` but caches the mailcap (first argument)

read_config()

parse alot's config file from path

read_notmuch_config()

parse notmuch's config file from path

reload()

Reload All configurations.

This first resets all configs to default (in case an overwritten binding is removed from the user config), then reloads the notmuch config, and finally reads the alot config.

Implementation Detail: this is the same code called by the constructor to set bindings at alot startup.

represent_datetime(*d*)

turns a given datetime obj into a unicode string representation. This will:

1. look if a fixed 'timestamp_format' is given in the config

2. check if a 'timestamp_format' hook is defined

3. use *pretty_datetime()* as fallback

set(*key*, *value*)

setter for global config values

Parameters

- **key** (`str`) – config option identifies

- **value** (depends on the specfile `alot.rc.spec`) – option to set

Errors

exception `alot.settings.errors.`**ConfigError**

could not parse user config

exception `alot.settings.errors.`**NoMatchingAccount**

No account matching requirements found.

Utils

`alot.settings.utils.`**read_config**(*configpath=None*, *specpath=None*, *checks=None*)

get a (validated) config object for given config file path.

Parameters

- **configpath** (`str or list(str)`) – path to config-file or a list of lines as its content

- **specpath** (`str`) – path to spec-file

- **checks** (`dict str->callable,`) – custom checks to use for validator. see validate docs

> **Raises** `ConfigError`

> **Return type** *configobj.ConfigObj*

alot.settings.utils.**resolve_att**(*a*, *fallback*)
  replace '' and 'default' by fallback values

Themes

class alot.settings.theme.**Theme**(*path*)
  Colour theme

> **Parameters path** (`str`) – path to theme file

> **Raises** `ConfigError`

get_attribute(*colourmode*, *mode*, *name*, *part=None*)
  returns requested attribute

> **Parameters**
>
> - **mode** (`str`) – ui-mode (e.g. *search*,'thread'...)
>
> - **name** (`str`) – of the atttribute
>
> - **colourmode** (`int`) – colour mode; in [1, 16, 256]

> **Return type** urwid.AttrSpec

get_threadline_theming(*thread*, *colourmode*)
  look up how to display a Threadline wiidget in search mode for a given thread.

> **Parameters**
>
> - **thread** (`alot.db.thread.Thread`) – Thread to theme Threadline for
>
> - **colourmode** (`int`) – colourmode to use, one of 1,16,256.

> **This will return a dict mapping**
>
> > **normal** to *urwid.AttrSpec*,
> >
> > **focus** to *urwid.AttrSpec*,
> >
> > **parts** to a list of strings indentifying subwidgets to be displayed in this order.
>
> **Moreover, for every part listed this will map 'part' to a dict mapping**
>
> > **normal** to *urwid.AttrSpec*,
> >
> > **focus** to *urwid.AttrSpec*,
> >
> > **width** to a tuple indicating the width of the subpart. This is either *('fit', min, max)* to force the widget to be at least *min* and at most *max* characters wide, or *('weight', n)* which makes it share remaining space with other 'weight' parts.
> >
> > **alignment** where to place the content if shorter than the widget. This is either 'right', 'left' or 'center'.

Accounts

class `alot.account.``**Account**`(*address=None*, *aliases=None*, *alias_regexp=None*, *realname=None*, *gpg_key=None*, *signature=None*, *signature_filename=None*, *signature_as_attachment=False*, *sent_box=None*, *sent_tags=None*, *draft_box=None*, *draft_tags=None*, *abook=None*, *sign_by_default=False*, *encrypt_by_default=u'none'*, ***_*)

Datastructure that represents an email account. It manages this account's settings, can send and store mails to maildirs (drafts/send).

Note: This is an abstract class that leaves *send_mail()* unspecified. See *SendmailAccount* for a subclass that uses a sendmail command to send out mails.

`get_addresses`()
    return all email addresses connected to this account, in order of their importance

`send_mail`(*mail*)
    sends given mail

> **Parameters** **`mail`** (`email.message.Message` or string) – the mail to send
>
> **Returns** a *Deferred* that errs back with a class:*SendingMailFailed*, containing a reason string if an error occured.

`store_draft_mail`(*mail*)
    stores mail (`email.message.Message` or str) as draft if `draft_box` is set.

static `store_mail`(*mbx*, *mail*)
    stores given mail in mailbox. If mailbox is maildir, set the S-flag and return path to newly added mail. Oherwise this will return *None*.

> **Parameters**
>
> - **`mbx`** (`mailbox.Mailbox`) – mailbox to use
>
> - **`mail`** (`email.message.Message` or str) – the mail to store
>
> **Returns** absolute path of mail-file for Maildir or None if mail was successfully stored
>
> **Return type** str or None
>
> **Raises** StoreMailError

`store_sent_mail`(*mail*)
    stores mail (`email.message.Message` or str) in send-store if `sent_box` is set.

`abook` = None
    addressbook (`addressbook.AddressBook`) managing this accounts contacts

`address` = None
    this accounts main email address

`alias_regexp` = []
    regex matching alternative addresses

`aliases` = []
    list of alternative addresses

`gpg_key` = None
    gpg fingerprint for this account's private key

---

realname = None
  real name used to format from-headers

signature = None
  signature to append to outgoing mails

signature_as_attachment = None
  attach signature file instead of appending its content to body text

signature_filename = None
  filename of signature file in attachment

class alot.account.**SendmailAccount**(*cmd*, ***kwargs*)
  Account that pipes a message to a *sendmail* shell command for sending

  > **Parameters cmd** (*str*) – sendmail command to use for this account

send_mail(*mail*)
  Pipe the given mail to the configured sendmail command. Display a short message on success or a notification on error. :param mail: the mail to send out :type mail: email.message.Message or string :returns: the deferred that calls the sendmail command :rtype: *twisted.internet.defer.Deferred*

Addressbooks

class alot.addressbook.**AddressBook**(*ignorecase=True*)
  can look up email addresses and realnames for contacts.

  Note: This is an abstract class that leaves *get_contacts()* unspecified. See AbookAddressBook and ExternalAddressbook for implementations.

  ---

  get_contacts()
    list all contacts tuples in this abook as (name, email) tuples

  lookup(*query=''*)
    looks up all contacts where name or address match query

class alot.addressbook.abook.**AbookAddressBook**(*path='~/.abook/addressbook'*, ***kwargs*)
  AddressBook that parses abook's config/database files

  > **Parameters path** (*str*) – path to abook addressbook file

class alot.addressbook.external.**ExternalAddressbook**(*commandline*, *regex*, *reflags=0*, *external_filtering=True*, ***kwargs*)

  AddressBook that parses a shell command's output

  Parameters

  - **commandline** (*str*) – commandline
  - **regex** (*str*) – regular expression used to match contacts in *commands* output to stdout. Must define subparts named "email" and "name".
  - **reflags** (*str*) – flags to use with regular expression. Use the constants defined in re here (*re.IGNORECASE* etc.)
  - **external_filtering** (*bool*) – if True the command is fired with the given search string as parameter and the result is not filtered further. If set to False, the command is fired without additional parameters and the result list is filtered according to the search string.

Utils

alot.helper.**RFC3156_canonicalize**(*text*)
> Canonicalizes plain text (MIME-encoded usually) according to RFC3156.
>
> This function works as follows (in that order):
>
> > 1. Convert all line endings to \r\n (DOS line endings).
> >
> > 2. Encode all occurences of "From " at the beginning of a line to "From=20" in order to prevent other mail programs to replace this with "> From" (to avoid MBox conflicts) and thus invalidate the signature.
> >
> > **Parameters** **text** – text to canonicalize (already encoded as quoted-printable)
> >
> > **Return type** str

alot.helper.**call_cmd**(*cmdlist*, *stdin=None*)
> get a shell commands output, error message and return value and immediately return.
>
> > **Warning:** This returns with the first screen content for interactive commands.
>
> > **Parameters**
> >
> > - **cmdlist** (*list of str*) – shellcommand to call, already splitted into a list accepted by subprocess.Popen()
> > - **stdin** (*str*) – string to pipe to the process
> >
> > **Returns** triple of stdout, stderr, return value of the shell command
> >
> > **Return type** str, str, int

alot.helper.**call_cmd_async**(*cmdlist*, *stdin=None*, *env=None*)
> get a shell commands output, error message and return value as a deferred.
>
> > **Parameters** **stdin** (*str*) – string to pipe to the process
> >
> > **Returns** deferred that calls back with triple of stdout, stderr and return value of the shell command
> >
> > **Return type** *twisted.internet.defer.Deferred*

alot.helper.**email_as_string**(*mail*)
> Converts the given message to a string, without mangling "From" lines (like as_string() does).
>
> > **Parameters** **mail** – email to convert to string
> >
> > **Return type** str

alot.helper.**guess_encoding**(*blob*)
> uses file magic to determine the encoding of the given data blob.
>
> > **Parameters** **blob** (*data*) – file content as read by file.read()
> >
> > **Returns** encoding
> >
> > **Return type** str

alot.helper.**guess_mimetype**(*blob*)
> uses file magic to determine the mime-type of the given data blob.
>
> > **Parameters** **blob** (*data*) – file content as read by file.read()

> **Returns** mime-type, falls back to 'application/octet-stream'
>
> **Return type** str

alot.helper.**humanize_size**(*size*)

> Create a nice human readable representation of the given number (understood as bytes) using the "KiB" and "MiB" suffixes to indicate kibibytes and mebibytes. A kibibyte is defined as 1024 bytes (as opposed to a kilobyte which is 1000 bytes) and a mibibyte is 1024**2 bytes (as opposed to a megabyte which is 1000**2 bytes).
>
> > **Parameters** **size** (*int*) – the number to convert
> >
> > **Returns** the human readable representation of size
> >
> > **Return type** str

alot.helper.**libmagic_version_at_least**(*version*)

> checks if the libmagic library installed is more recent than a given version.
>
> > **Parameters** **version** – minimum version expected in the form XYY (i.e. 5.14 -> 514) with XYY >= 513

alot.helper.**mailto_to_envelope**(*mailto_str*)

> Interpret mailto-string into a *alot.db.envelope.Envelope*

alot.helper.**mimewrap**(*path*, *filename=None*, *ctype=None*)

> Take the contents of the given path and wrap them into an email MIME part according to the content type. The content type is auto detected from the actual file contents and the file name if it is not given.
>
> > **Parameters**
> >
> > - **path** (*str*) – the path to the file contents
> > - **filename** (*str or None*) – the file name to use in the generated MIME part
> > - **ctype** (*str or None*) – the content type of the file contents in path
> >
> > **Returns** the message MIME part storing the data from path
> >
> > **Return type** subclasses of email.mime.base.MIMEBase

alot.helper.**parse_mailcap_nametemplate**(*tmplate='%s'*)

> this returns a prefix and suffix to be used in the tempfile module for a given mailcap nametemplate string

alot.helper.**parse_mailto**(*mailto_str*)

> Interpret mailto-string
>
> > **Parameters** **mailto_str** (*str*) – the string to interpret. Must conform to :rfc:2368.
> >
> > **Returns** the header fields and the body found in the mailto link as a tuple of length two
> >
> > **Return type** tuple(dict(str->list(str)), str)

alot.helper.**pretty_datetime**(*d*)

> translates datetime *d* to a "sup-style" human readable string.

```
>>> now = datetime.now()
>>> now.strftime('%c')
'Sat 31 Mar 2012 14:47:26 '
>>> pretty_datetime(now)
u'just now'
>>> pretty_datetime(now - timedelta(minutes=1))
u'1min ago'
>>> pretty_datetime(now - timedelta(hours=5))
u'5h ago'
```

```
>>> pretty_datetime(now - timedelta(hours=12))
u'02:54am'
>>> pretty_datetime(now - timedelta(days=1))
u'yest 02pm'
>>> pretty_datetime(now - timedelta(days=2))
u'Thu 02pm'
>>> pretty_datetime(now - timedelta(days=7))
u'Mar 24'
>>> pretty_datetime(now - timedelta(days=356))
u'Apr 2011'
```

alot.helper.**shell_quote**(*text*)

> Escape the given text for passing it to the shell for interpretation. The resulting string will be parsed into one "word" (in the sense used in the shell documentation, see sh(1)) by the shell.
>
> > **Parameters** **text** (*str*) – the text to quote
> >
> > **Returns** the quoted text
> >
> > **Return type** str

alot.helper.**shorten**(*string*, *maxlen*)

> shortens string if longer than maxlen, appending ellipsis

alot.helper.**shorten_author_string**(*authors_string*, *maxlength*)

> Parse a list of authors concatenated as a text string (comma separated) and smartly adjust them to maxlength.
>
> 1) If the complete list of sender names does not fit in maxlength, it tries to shorten names by using only the first part of each.
>
> 2) If the list is still too long, hide authors according to the following priority:
>
> > • First author is always shown (if too long is shorten with ellipsis)
> >
> > • If possible, last author is also shown (if too long, uses ellipsis)
> >
> > • If there are more than 2 authors in the thread, show the maximum of them. More recent senders have higher priority.
> >
> > • If it is finally necessary to hide any author, an ellipsis between first and next authors is added.

alot.helper.**split_commandline**(*s*, *comments=False*, *posix=True*)

> splits semi-colon separated commandlines

alot.helper.**split_commandstring**(*cmdstring*)

> split command string into a list of strings to pass on to subprocess.Popen and the like. This simply calls shlex.split but works also with unicode bytestrings.

alot.helper.**string_decode**(*string*, *enc='ascii'*)

> safely decodes string to unicode bytestring, respecting *enc* as a hint.
>
> > **Parameters**
> >
> > > • **string** (*str or unicode*) – the string to decode
> > >
> > > • **enc** (*str*) – a hint what encoding is used in string ('ascii', 'utf-8', ...)
> >
> > **Returns** the unicode decoded input string
> >
> > **Return type** unicode

alot.helper.**string_sanitize**(*string*, *tab_width=8*)

> strips, and replaces non-printable characters

Parameters **tab_width** (int or *None*) – number of spaces to replace tabs with. Read from *glob-als.tabwidth* setting if *None*

```
>>> string_sanitize(' foo\rbar ', 8)
' foobar '
>>> string_sanitize('foo\tbar', 8)
'foo     bar'
>>> string_sanitize('foo\t\tbar', 8)
'foo             bar'
```

Commands

User actions are represented by *Command* objects that can then be triggered by *alot.ui.UI.apply_command()*. Command-line strings given by the user via the prompt or key bindings can be translated to *Command* objects using *alot.commands.commandfactory()*. Specific actions are defined as subclasses of *Command* and can be registered to a global command pool using the *registerCommand* decorator.

Note: that the return value of *commandfactory()* depends on the current *mode* the user interface is in. The mode identifier is a string that is uniquely defined by the currently focuses *Buffer*.

Note: The names of the commands available to the user in any given mode do not correspond one-to-one to these subclasses. You can register a Command multiple times under different names, with different forced constructor parameters and so on. See for instance the definition of BufferFocusCommand in 'commands/globals.py':

```
@registerCommand(MODE, 'bprevious', forced={'offset': -1},
                 help='focus previous buffer')
@registerCommand(MODE, 'bnext', forced={'offset': +1},
                 help='focus next buffer')
class BufferFocusCommand(Command):
    def __init__(self, buffer=None, offset=0, **kwargs):
    ...
```

class alot.commands.**Command**
    base class for commands

    apply(*caller*)
        code that gets executed when this command is applied

class alot.commands.**CommandParseError**
    could not parse commandline string

class alot.commands.**CommandArgumentParser**(*args*, **kwargs*)
    ArgumentParser that raises *CommandParseError* instead of printing to *sys.stderr*

alot.commands.**commandfactory**(*cmdline*, *mode='global'*)
    parses *cmdline* and constructs a *Command*.

        Parameters

            • **cmdline** (*str*) – command line to interpret

            • **mode** (*str*) – mode identifier

alot.commands.**lookup_command**(*cmdname*, *mode*)

> returns commandclass, argparser and forced parameters used to construct a command for *cmdname* when called in *mode*.

> > **Parameters**
> >
> > > - **cmdname** (`str`) – name of the command to look up
> > >
> > > - **mode** (`str`) – mode identifier
> >
> > **Return type** (`Command`, `ArgumentParser`, dict(str->dict))

alot.commands.**lookup_parser**(*cmdname*, *mode*)

> returns the `CommandArgumentParser` used to construct a command for *cmdname* when called in *mode*.

class alot.commands.**registerCommand**(*mode*, *name*, *help=None*, *usage=None*, *forced=None*, *arguments=None*)

> Decorator used to register a `Command` as handler for command *name* in *mode* so that it can be looked up later using `lookup_command()`.
>
> Consider this example that shows how a `Command` class definition is decorated to register it as handler for 'save' in mode 'thread' and add boolean and string arguments:

```
.. code-block::
```

> > **@registerCommand('thread', 'save', arguments=[** (['–all'], {'action': 'store_true', 'help':'save all'}), (['path'], {'nargs':'?', 'help':'path to save to'})], help='save attachment(s)')
> >
> > **class SaveAttachmentCommand(Command):** pass
>
> > **Parameters**
> >
> > > - **mode** (`str`) – mode identifier
> > >
> > > - **name** (`str`) – command name to register as
> > >
> > > - **help** (`str`) – help string summarizing what this command does
> > >
> > > - **usage** (`str`) – overides the auto generated usage string
> > >
> > > - **forced** (`dict (str->str)`) – keyword parameter used for commands constructor
> > >
> > > - **arguments** (`list of (list of str, dict (str->str))`) – list of arguments given as pairs (args, kwargs) accepted by `argparse.ArgumentParser.add_argument()`.

Globals

class alot.commands.globals.**BufferCloseCommand**(*buffer=None*, *force=False*, *redraw=True*, ***kwargs*)

> close a buffer
>
> > **Parameters**
> >
> > > - **buffer** (*alot.buffers.Buffer*) – the buffer to close or None for current
> > >
> > > - **force** (`bool`) – force buffer close

class alot.commands.globals.**BufferFocusCommand**(*buffer=None*, *index=None*, *offset=0*, ***kwargs*)

> focus a `Buffer`
>
> > **Parameters**

- **buffer** (*alot.buffers.Buffer*) – the buffer to focus or None

- **index** (`int`) – index (in bufferlist) of the buffer to focus.

- **offset** (`int`) – position of the buffer to focus relative to the currently focussed one. This is used only if *buffer* is set to *None*

class `alot.commands.globals.`**`CallCommand`**(*command*, ***kwargs*)

  Executes python code

      Parameters command (`str`) – python command string to call

class `alot.commands.globals.`**`ComposeCommand`**(*envelope=None*, *headers=None*, *template=None*, *sender=u''*, *subject=u''*, *to=None*, *cc=None*, *bcc=None*, *attach=None*, *omit_signature=False*, *spawn=None*, *rest=None*, *encrypt=False*, ***kwargs*)

  compose a new email

      Parameters

- **envelope** (`Envelope`) – use existing envelope

- **headers** (`dict (str->str)`) – forced header values

- **template** (`str`) – name of template to parse into the envelope after creation. This should be the name of a file in your template_dir

- **sender** (`str`) – From-header value

- **subject** (`str`) – Subject-header value

- **to** (`str`) – To-header value

- **cc** (`str`) – Cc-header value

- **bcc** (`str`) – Bcc-header value

- **attach** (`str`) – Path to files to be attached (globable)

- **omit_signature** (`bool`) – do not attach/append signature

- **spawn** (`bool`) – force spawning of editor in a new terminal

- **rest** (`list(str)`) – remaining parameters. These can start with 'mailto' in which case it is interpreted as mailto string. Otherwise it will be interpreted as recipients (to) header

- **encrypt** (`bool`) – if the email should be encrypted

class `alot.commands.globals.`**`EditCommand`**(*path*, *spawn=None*, *thread=None*, ***kwargs*)

  edit a file

      Parameters

- **path** (`str`) – path to the file to be edited

- **spawn** (`bool`) – force running edtor in a new terminal

- **thread** (`bool`) – run asynchronously, don't block alot

class `alot.commands.globals.`**`ExitCommand`**(*_prompt=True*, ***kwargs*)

  Shut down cleanly.

  The _prompt variable is for internal use only, it's used to control prompting to close without sending, and is used by the BufferCloseCommand if settings change after yielding to the UI.

class `alot.commands.globals.`**`ExternalCommand`**(*cmd*, *stdin=None*, *shell=False*, *spawn=False*, *refocus=True*, *thread=False*, *on_success=None*, ***kwargs*)

> run external command
>
> > **Parameters**
> >
> > - **cmd** (`list or str`) – the command to call
> > - **stdin** (`file or str`) – input to pipe to the process
> > - **spawn** (`bool`) – run command in a new terminal
> > - **shell** (`bool`) – let shell interpret command string
> > - **thread** (`bool`) – run asynchronously, don't block alot
> > - **refocus** (`bool`) – refocus calling buffer after cmd termination
> > - **on_success** (`callable`) – code to execute after command successfully exited

class `alot.commands.globals.`**`FlushCommand`**(*callback=None*, *silent=False*, ***kwargs*)

> flush write operations or retry until committed
>
> > **Parameters callback** (`callable`) – function to call after successful writeout

class `alot.commands.globals.`**`HelpCommand`**(*commandname=''*, ***kwargs*)

> display help for a command. Use 'bindings' to display all keybings interpreted in current mode.'
>
> > **Parameters commandname** (`str`) – command to document

class `alot.commands.globals.`**`MoveCommand`**(*movement=None*, ***kwargs*)

> move in widget

class `alot.commands.globals.`**`OpenBufferlistCommand`**(*filtfun=<function <lambda>>*, ***kwargs*)

> open a list of active buffers
>
> > **Parameters filtfun** (`callable (str->bool)`) – filter to apply to displayed list

class `alot.commands.globals.`**`PromptCommand`**(*startwith=''*, ***kwargs*)

> prompts for commandline and interprets it upon select
>
> > **Parameters startwith** (`str`) – initial content of the prompt widget

class `alot.commands.globals.`**`PythonShellCommand`**

> open an interactive python shell for introspection

class `alot.commands.globals.`**`RefreshCommand`**

> refresh the current buffer

class `alot.commands.globals.`**`ReloadCommand`**

> Reload configuration.

class `alot.commands.globals.`**`RepeatCommand`**(***kwargs*)

> Repeats the command executed last time

class `alot.commands.globals.`**`SearchCommand`**(*query*, *sort=None*, ***kwargs*)

> open a new search buffer. Search obeys the notmuch *search.exclude_tags* setting.
>
> > **Parameters**
> >
> > - **query** (`str`) – notmuch querystring
> > - **sort** (`str`) – how to order results. Must be one of 'oldest_first', 'newest_first', 'message_id' or 'unsorted'.

class alot.commands.globals.**TagListCommand**(*filtfun=<function   <lambda>>,   tags=None,   **kwargs*)

opens taglist buffer

> **Parameters filtfun** (`callable (str->bool)`) – filter to apply to displayed list

Envelope

class alot.commands.envelope.**AttachCommand**(*path, **kwargs*)

attach files to the mail

> **Parameters path** (`str`) – files to attach (globable string)

class alot.commands.envelope.**EditCommand**(*envelope=None,   spawn=None,   refocus=True,   **kwargs*)

edit mail

> **Parameters**
>
> - **envelope** (`Envelope`) – email to edit
> - **spawn** (`bool`) – force spawning of editor in a new terminal
> - **refocus** – m

class alot.commands.envelope.**RefineCommand**(*key='', **kwargs*)

prompt to change the value of a header

> **Parameters key** (`str`) – key of the header to change

class alot.commands.envelope.**SaveCommand**

save draft

class alot.commands.envelope.**SendCommand**(*mail=None, envelope=None, **kwargs*)

send mail

> **Parameters**
>
> - **mail** – email to send
> - **envelope** (`alot.db.envelope.envelope`) – envelope to use to construct the outgoing mail. This will be ignored in case the mail parameter is set.

class alot.commands.envelope.**SetCommand**(*key, value, append=False, **kwargs*)

set header value

> **Parameters**
>
> - **key** (`str`) – key of the header to change
> - **value** (`str`) – new value

class alot.commands.envelope.**SignCommand**(*action=None, keyid=None, **kwargs*)

toggle signing this email

> **Parameters**
>
> - **action** (`str`) – whether to sign/unsign/toggle
> - **keyid** (`str`) – which key id to use

class alot.commands.envelope.**TagCommand**(*tags=u'', action='add', **kwargs*)

manipulate message tags

> **Parameters**

- **tags** (`unicode`) – comma separated list of tagstrings to set

- **action** (`str`) – adds tags if 'add', removes them if 'remove', adds tags and removes all other if 'set' or toggle individually if 'toggle'

class alot.commands.envelope.**ToggleHeaderCommand**
    toggle display of all headers

class alot.commands.envelope.**UnattachCommand**(*hint=None*, ***kwargs*)
    remove attachments from current envelope

        Parameters hint (`str`) – which attached file to remove

class alot.commands.envelope.**UnsetCommand**(*key*, ***kwargs*)
    remove header field

        Parameters key (`str`) – key of the header to remove

Bufferlist

class alot.commands.bufferlist.**BufferCloseCommand**
    close focussed buffer

class alot.commands.bufferlist.**BufferFocusCommand**
    focus selected buffer

Search

class alot.commands.search.**OpenThreadCommand**(*thread=None*, ***kwargs*)
    open thread in a new buffer

        Parameters thread ([`Thread`]) – thread to open (Uses focussed thread if unset)

class alot.commands.search.**RefineCommand**(*query=None*, *sort=None*, ***kwargs*)
    refine the querystring of this buffer

        Parameters query (`list of str`) – new querystring given as list of strings as returned by argparse

class alot.commands.search.**RefinePromptCommand**
    prompt to change this buffers querystring

class alot.commands.search.**RetagPromptCommand**
    prompt to retag selected threads' tags

class alot.commands.search.**TagCommand**(*tags=u''*, *action='add'*, *allmessages=False*, *flush=True*, ***kwargs*)
    manipulate message tags

        Parameters

- **tags** (`str`) – comma separated list of tagstrings to set

- **action** (`str`) – adds tags if 'add', removes them if 'remove', adds tags and removes all other if 'set' or toggle individually if 'toggle'

- **allmessages** (`bool`) – tag all messages in search result

- **flush** (`bool`) – imediately write out to the index

Taglist

class `alot.commands.taglist.`**`TaglistSelectCommand`**
  search for messages with selected tag

Thread

class `alot.commands.thread.`**`BounceMailCommand`**(*message=None*, ***kwargs*)
  directly re-send selected message

> **Parameters `message`** (*alot.db.message.Message*) – message to bounce (defaults to selected message)

class `alot.commands.thread.`**`ChangeDisplaymodeCommand`**(*query=None*, *visible=None*, *raw=None*, *all_headers=None*, ***kwargs*)
  fold or unfold messages

> **Parameters**
>
> - **query** (*str*) – notmuch query string used to filter messages to affect
> - **visible** (*True, False, 'toggle' or None*) – unfold if *True*, fold if *False*, ignore if *None*
> - **raw** (*True, False, 'toggle' or None*) – display raw message text.
> - **all_headers** (*True, False, 'toggle' or None*) – show all headers (only visible if not in raw mode)

class `alot.commands.thread.`**`EditNewCommand`**(*message=None*, *spawn=None*, ***kwargs*)
  edit message in as new

> **Parameters**
>
> - **message** (*alot.db.message.Message*) – message to reply to (defaults to selected message)
> - **spawn** (*bool*) – force spawning of editor in a new terminal

class `alot.commands.thread.`**`ForwardCommand`**(*message=None*, *attach=True*, *spawn=None*, ***kwargs*)
  forward message

> **Parameters**
>
> - **message** (*alot.db.message.Message*) – message to forward (defaults to selected message)
> - **attach** (*bool*) – attach original mail instead of inline quoting its body
> - **spawn** (*bool*) – force spawning of editor in a new terminal

class `alot.commands.thread.`**`OpenAttachmentCommand`**(*attachment*, ***kwargs*)
  displays an attachment according to mailcap

> **Parameters `attachment`** (*Attachment*) – attachment to open

class `alot.commands.thread.`**`PipeCommand`**(*cmd*, *all=False*, *separately=False*, *background=False*, *shell=False*, *notify_stdout=False*, *format='raw'*, *add_tags=False*, *noop_msg='no command specified'*, *confirm_msg=''*, *done_msg=None*, *field_key='copiousoutput'*, ***kwargs*)
  pipe message(s) to stdin of a shellcommand

> **Parameters**

- **cmd** (`str or list of str`) – shellcommand to open
- **all** (`bool`) – pipe all, not only selected message
- **separately** (`bool`) – call command once per message
- **background** (`bool`) – do not suspend the interface
- **notify_stdout** (`bool`) – display command's stdout as notification message
- **shell** – let the shell interpret the command
- **add_tags** (`bool`) – add 'Tags' header to the message
- **noop_msg** (`str`) – error notification to show if *cmd* is empty
- **confirm_msg** (`str`) – confirmation question to ask (continues directly if unset)
- **done_msg** (`str`) – notification message to show upon success
- **field_key** (`str`) – malcap field key for decoding

class `alot.commands.thread.`**PrintCommand**(*all=False*,   *separately=False*,   *raw=False*,
add_tags=False, ***kwargs*)

 print message(s)

 Parameters

- **all** (`bool`) – print all, not only selected messages
- **separately** (`bool`) – call print command once per message
- **raw** (`bool`) – pipe raw message string to print command
- **add_tags** (`bool`) – add 'Tags' header to the message

class `alot.commands.thread.`**RemoveCommand**(*all=False*, ***kwargs*)

 remove message(s) from the index

 Parameters all (`bool`) – remove all messages from thread, not just selected one

class `alot.commands.thread.`**ReplyCommand**(*message=None*,   *all=False*,   *listreply=None*,
spawn=None, ***kwargs*)

 reply to message

 Parameters

- **message** (*alot.db.message.Message*) – message to reply to (defaults to selected message)
- **all** (`bool`) – group reply; copies recipients from Bcc/Cc/To to the reply
- **listreply** (`bool`) – reply to list; autodetect if unset and enabled in config
- **spawn** (`bool`) – force spawning of editor in a new terminal

 static **clear_my_address**(*my_addresses*, *value*)

  return recipient header without the addresses in my_addresses

  Parameters

- **my_addresses** (`list(str)`) – a list of my email addresses (no real name part)
- **value** (`list(str)`) – a list of recipient or sender strings (with or without real names as taken from email headers)

  Returns a new, potentially shortend list

  Return type list(str)

static **ensure_unique_address**(*recipients*)
> clean up a list of name,address pairs so that no address appears multiple times.

class alot.commands.thread.**SaveAttachmentCommand**(*all=False*, *path=None*, ***kwargs*)
> save attachment(s)

> **Parameters**

> - **all** (*bool*) – save all, not only selected attachment
> - **path** (*str*) – path to write to. if *all* is set, this must be a directory.

class alot.commands.thread.**TagCommand**(*tags=u''*, *action='add'*, *all=False*, *flush=True*, ***kwargs*)
> manipulate message tags

> **Parameters**

> - **tags** (*str*) – comma separated list of tagstrings to set
> - **action** (*str*) – adds tags if 'add', removes them if 'remove', adds tags and removes all other if 'set' or toggle individually if 'toggle'
> - **all** (*bool*) – tag all messages in thread
> - **flush** (*bool*) – imediately write out to the index

class alot.commands.thread.**ThreadSelectCommand**
> select focussed element. The fired action depends on the focus: - if message summary, this toggles visibility of the message, - if attachment line, this opens the attachment

alot.commands.thread.**determine_sender**(*mail*, *action='reply'*)
> Inspect a given mail to reply/forward/bounce and find the most appropriate account to act from and construct a suitable From-Header to use.

> **Parameters**

> - **mail** (*email.message.Message*) – the email to inspect
> - **action** (*str*) – intended use case: one of "reply", "forward" or "bounce"

Crypto

alot.crypto.**RFC3156_micalg_from_algo**(*hash_algo*)
> Converts a GPGME hash algorithm name to one conforming to RFC3156.

> GPGME returns hash algorithm names such as "SHA256", but RFC3156 says that programs need to use names such as "pgp-sha256" instead.

> **Parameters** **hash_algo** (*str*) – GPGME hash_algo

> **Returns** the lowercase name of of the algorithm with "pgp-" prepended

> **Return type** str

alot.crypto.**check_uid_validity**(*key*, *email*)
> Check that a the email belongs to the given key. Also check the trust level of this connection. Only if the trust level is high enough (>=4) the email is assumed to belong to the key.

> **Parameters**

> - **key** (*gpg.gpgme._gpgme_key*) – the GPG key to which the email should belong
> - **email** (*str*) – the email address that should belong to the key

> **Returns** whether the key can be assumed to belong to the given email
>
> **Return type** [bool](#)

alot.crypto.**decrypt_verify**(*encrypted*)

> Decrypts the given ciphertext string and returns both the signatures (if any) and the plaintext.
>
> > **Parameters encrypted** ([str](#)) – the mail to decrypt
> >
> > **Returns** the signatures and decrypted plaintext data
> >
> > **Return type** [tuple](#)[list[gpg.resuit.Signature], [str](#)]
> >
> > **Raises** GPGProblem if the decryption fails

alot.crypto.**detached_signature_for**(*plaintext_str*, *keys*)

> Signs the given plaintext string and returns the detached signature.
>
> A detached signature in GPG speak is a separate blob of data containing a signature for the specified plaintext.
>
> > **Parameters**
> >
> > - **plaintext_str** ([str](#)) – text to sign
> > - **keys** (*list[gpg.gpgme._gpgme_key]*) – list of one or more key to sign with.
> >
> > **Returns** A list of signature and the signed blob of data
> >
> > **Return type** [tuple](#)[list[gpg.results.NewSignature], [str](#)]

alot.crypto.**encrypt**(*plaintext_str*, *keys=None*)

> Encrypt data and return the encrypted form.
>
> > **Parameters**
> >
> > - **plaintext_str** ([str](#)) – the mail to encrypt
> > - **key** (*list[gpg.gpgme.gpgme_key_t] or* [None](#)) – optionally, a list of keys to encrypt with
> >
> > **Returns** encrypted mail
> >
> > **Return type** [str](#)

alot.crypto.**get_key**(*keyid*, *validate=False*, *encrypt=False*, *sign=False*, *signed_only=False*)

> Gets a key from the keyring by filtering for the specified keyid, but only if the given keyid is specific enough (if it matches multiple keys, an exception will be thrown).
>
> If validate is True also make sure that returned key is not invalid, revoked or expired. In addition if encrypt or sign is True also validate that key is valid for that action. For example only keys with private key can sign. If signed_only is True make sure that the user id can be trusted to belong to the key (is signed). This last check will only work if the keyid is part of the user id associated with the key, not if it is part of the key fingerprint.
>
> > **Parameters**
> >
> > - **keyid** ([str](#)) – filter term for the keyring (usually a key ID)
> > - **validate** ([bool](#)) – validate that returned keyid is valid
> > - **encrypt** ([bool](#)) – when validating confirm that returned key can encrypt
> > - **sign** ([bool](#)) – when validating confirm that returned key can sign
> > - **signed_only** ([bool](#)) – only return keys whose uid is signed (trusted to belong to the key)
> >
> > **Returns** A gpg key matching the given parameters

> **Return type** gpg.gpgme._gpgme_key
>
> **Raises**
>
> - **GPGProblem** – if the keyid is ambiguous
>
> - **GPGProblem** – if there is no key that matches the parameters
>
> - **GPGProblem** – if a key is found, but signed_only is true and the key is unused

alot.crypto.**list_keys**(*hint=None*, *private=False*)

> Returns a generator of all keys containing the fingerprint, or all keys if hint is None.
>
> The generator may raise exceptions of :class:gpg.errors.GPGMEError, and it is the caller's responsibility to handle them.
>
> **Parameters**
>
> - **hint** (*str or None*) – Part of a fingerprint to usee to search
>
> - **private** (*bool*) – Whether to return public keys or secret keys
>
> **Returns** A generator that yields keys.
>
> **Return type** Generator[gpg.gpgme.gpgme_key_t, None, None]

alot.crypto.**validate_key**(*key*, *sign=False*, *encrypt=False*)

> Assert that a key is valide and optionally that it can be used for signing or encrypting. Raise GPGProblem otherwise.
>
> **Parameters**
>
> - **key** (*gpg.gpgme._gpgme_key*) – the GPG key to check
>
> - **sign** (*bool*) – whether the key should be able to sign
>
> - **encrypt** (*bool*) – whether the key should be able to encrypt
>
> **Raises**
>
> - **GPGProblem** – If the key is revoked, expired, or invalid
>
> - **GPGProblem** – If encrypt is true and the key cannot be used to encrypt
>
> - **GPGProblem** – If sign is true and th key cannot be used to encrypt

alot.crypto.**verify_detached**(*message*, *signature*)

> Verifies whether the message is authentic by checking the signature.
>
> **Parameters**
>
> - **message** (*str*) – The message to be verified, in canonical form.
>
> - **signature** (*str*) – the OpenPGP signature to verify
>
> **Returns** a list of signatures
>
> **Return type** list[gpg.results.Signature]
>
> **Raises** GPGProblem if the verification fails

FAQ

1. Why reinvent the wheel? Why not extend an existing MUA to work nicely with notmuch?

   alot makes use of existing solutions where possible: It does not fetch, send or edit mails; it lets notmuch handle your mailindex and uses a toolkit to render its display. You are responsible for automatic initial tagging.

   This said, there are few CLI MUAs that could be easily and naturally adapted to using notmuch. Rebuilding an interface from scratch using friendly and extensible tools seemed easier and more promising.

   Update: see mutt-kz for a fork of mutt..

2. What's with the snotty name?

   It's not meant to be presumptuous. I like the dichotomy; I like to picture the look on someone's face who reads the `User-Agent` header "notmuch/alot"; I like cookies; I like this comic strip.

3. I want feature X!

   Me too! Feel free to file a new or comment on existing issues if you don't want/have the time/know how to implement it yourself. Be verbose as to how it should look or work when it's finished and give it some thought how you think we should implement it. We'll discuss it from there.

4. Why are the default key bindings so counter-intuitive?

   Be aware that the bindings for all modes are *fully configurable*. That said, I choose the bindings to be natural for me. I use vim and pentadactyl a lot. However, I'd be interested in discussing the defaults. If you think your bindings are more intuitive or better suited as defaults for some reason, don't hesitate to send me your config. The same holds for the theme settings you use. Tell me. Let's improve the defaults.

5. Help! I don't see *text/html* content!

   better: How do I properly set up an inline renderer for *text/html*? Try w3m and put the following into your `~/.mailcap`:

   ```
   text/html;  w3m -dump -o document_charset=%{charset} '%s'; nametemplate=
   →%s.html; copiousoutput
   ```

Most text based browsers have a dump mode that can be used here.

6. Why are you doing $THIS not $THAT way?

   Lazyness and Ignorance: In most cases I simply did not or still don't know a better solution. I try to outsource as much as I can to well established libraries and be it only to avoid having to read rfc's. But there are lots of tasks I implemented myself, possibly overlooking a ready made and available solution. Twisted is such a feature-rich but gray area in my mind for example. If you think you know how to improve the current implementation let me know!

   The few exceptions to above stated rule are the following:

   * The modules cmd and cmd2, that handle all sorts of convenience around command objects hate urwid: They are painfully strongly coupled to user in/output via stdin and out.

   * *notmuch reply* is not used to format reply messages because 1. it is not offered by notmuch's library but is a feature of the CLI. This means we would have to call the notmuch binary, something that is avoided where possible. 2. As there is no *notmuch forward* equivalent, this (very similar) functionality would have to be re-implemented anyway.

7. Why doesn't alot run on python3?

   We're on it. Check out the py3k milestone

Manpage

Synopsis

alot [options ...] [subcommand]

Description

Alot is a terminal-based mail user agent for the notmuch mail system. It features a modular and command prompt driven interface to provide a full MUA experience as an alternative to the Emacs mode shipped with notmuch.

Options

-r, --read-only   open db in read only mode

-c, --config=FILENAME   config file (default: ~/.config/alot/config)

-n, --notmuch-config=FILENAME notmuch   config   (default: $NOTMUCH_CONFIG   or ~/.notmuch-config)

-C, --colour-mode=COLOUR   terminal colour mode (default: 256). Must be 1, 16 or 256

-p, --mailindex-path=PATH   path to notmuch index

-d, --debug-level=LEVEL   debug log (default: info). Must be one of debug,info,warning or error

-l, --logfile=FILENAME   logfile (default: /dev/null)

-v, --version   Display version string and exit

-h, --help   Display help and exit

Commands

search start in a search buffer using the querystring provided as parameter. See also `notmuch-search-terms(7)`.

compose compose a new message

bufferlist start with only a bufferlist buffer open

taglist start with only a taglist buffer open

pyshell start the interactive python shell inside alot

Usage

The arrow keys, *page-up/down*, *j*, *k* and *Space* can be used to move the focus. *Escape* cancels prompts and *Enter* selects. Hit *:* at any time and type in commands to the prompt.

The interface shows one buffer at a time, you can use *Tab* and *Shift-Tab* to switch between them, close the current buffer with *d* and list them all with *;*.

The buffer type or *mode* (displayed at the bottom left) determines which prompt commands are available. Usage information on any command can be listed by typing *help YOURCOMMAND* to the prompt; The key bindings for the current mode are listed upon pressing *?*.

UNIX Signals

SIGUSR1 Refreshes the current buffer.

SIGINT Shuts down the user interface.

See Also

`notmuch(1)`

Alot is a terminal-based mail user agent for the notmuch mail system. It features a modular and command prompt driven interface to provide a full MUA experience as an alternative to the Emacs mode shipped with notmuch.

a

Index

A

abook (alot.account.Account attribute), 67
AbookAddressBook (class in alot.addressbook.abook), 68
AbooksCompleter (class in alot.completion), 60
Account (class in alot.account), 67
AccountCompleter (class in alot.completion), 61
accumulate_body() (alot.db.Message method), 48
add() (alot.db.envelope.Envelope method), 50
add_message() (alot.db.manager.DBManager method), 44
add_signature_headers() (in module alot.db.utils), 51
add_tags() (alot.db.Message method), 48
add_tags() (alot.db.Thread method), 46
address (alot.account.Account attribute), 67
AddressBook (class in alot.addressbook), 68
alias_regexp (alot.account.Account attribute), 67
aliases (alot.account.Account attribute), 67
alot (module), 41
alot.account (module), 67
alot.addressbook (module), 68
alot.addressbook.abook (module), 68
alot.addressbook.external (module), 68
alot.buffers (module), 55
alot.commands (module), 72
alot.commands.bufferlist (module), 77
alot.commands.envelope (module), 76
alot.commands.globals (module), 73
alot.commands.search (module), 77
alot.commands.taglist (module), 78
alot.commands.thread (module), 78
alot.completion (module), 60
alot.crypto (module), 80
alot.db (module), 43
alot.db.errors (module), 46
alot.db.utils (module), 51
alot.helper (module), 69
alot.settings.errors (module), 65
alot.settings.manager (module), 63

alot.settings.utils (module), 65
alot.ui (module), 52
alot.utils (module), 72
alot.widgets.bufferlist (module), 59
alot.widgets.globals (module), 58
alot.widgets.search (module), 59
alot.widgets.thread (module), 59
alot.widgets.utils (module), 58
apply() (alot.commands.Command method), 72
apply_command() (alot.ui.UI method), 53
apply_commandline() (alot.ui.UI method), 53
ArgparseOptionCompleter (class in alot.completion), 61
async() (alot.db.manager.DBManager method), 44
attach() (alot.db.envelope.Envelope method), 50
AttachCommand (class in alot.commands.envelope), 76
Attachment (class in alot.db.attachment), 49
attachments (alot.db.envelope.Envelope attribute), 50
AttachmentWidget (class in alot.widgets.globals), 58
AttrFlipWidget (class in alot.widgets.utils), 58

B

body (alot.db.envelope.Envelope attribute), 50
BounceMailCommand (class in alot.commands.thread), 78
Buffer (class in alot.buffers), 55
buffer_close() (alot.ui.UI method), 53
buffer_focus() (alot.ui.UI method), 53
buffer_open() (alot.ui.UI method), 53
BufferCloseCommand (class in alot.commands.bufferlist), 77
BufferCloseCommand (class in alot.commands.globals), 73
BufferFocusCommand (class in alot.commands.bufferlist), 77
BufferFocusCommand (class in alot.commands.globals), 73
BufferlineWidget (class in alot.widgets.bufferlist), 59
BufferlistBuffer (class in alot.buffers), 55
buffers (alot.ui.UI attribute), 55
build_statusbar() (alot.ui.UI method), 53

C

D

E

F

V

W