alot Documentation

Release 0.3.6

Patrick Totzke

December 10, 2015

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*
- PyGPGME *0.2*

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 pygpgme
```

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`.

CHAPTER 2

Usage

2.1 Commandline invocation

```
alot [-r] [-c CONFIGFILE] [-n NOTMUCHCONFIGFILE] [-C {1,16,256}] [-p DB_PATH]
     [-d {debug,info,warning,error}] [-l LOGFILE] [--version] [--help]
     [command]
```

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)

--version Display version string and exit

--help Display help and exit

Subommands

search start in a search buffer using the querystring provided as parameter. See also the SEARCH SYNTAX section of notmuch(1) and the output of *alot search –help*.

compose compose a new message See the output of *alot compose –help* for more info on parameters.

2.2 First Steps

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 *?*.

2.3 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

2.3.1 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

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

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

pyshell
open an interactive python shell for introspection

compose
compose a new email

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

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

2.3.2 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

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

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

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

2.3.3 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.

editnew
edit message in as new

optional arguments

—spawn open editor in new window.

move
move focus in current buffer

argument up, down, page up, page down, first, last

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.

—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

2.3.4 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

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').

send
send mail

sign
mark mail to be signed before sending

argument which key id to use

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

save
save draft

unsign
mark mail not to be signed before sending

unset
remove header field

argument header to refine

2.3.5 Commands in *bufferlist* mode

The following commands are available in bufferlist mode

close
close focussed buffer

open
focus selected buffer

2.3.6 Commands in *taglist* mode

The following commands are available in taglist mode

select
search for messages with selected tag

2.4 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.

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.

2.4.1 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

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

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

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

pyshell
open an interactive python shell for introspection

compose
compose a new email

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

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

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

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

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

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.

editnew
edit message in as new

optional arguments

—spawn open editor in new window.

move
move focus in current buffer

argument up, down, page up, page down, first, last

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.

—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

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').

send
send mail

sign
mark mail to be signed before sending

argument which key id to use

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

save
save draft

unsign
mark mail not to be signed before sending

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

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 [-r] [-c CONFIGFILE] [-n NOTMUCHCONFIGFILE] [-C {1,16,256}] [-p DB_PATH]
     [-d {debug,info,warning,error}] [-l LOGFILE] [--version] [--help]
     [command]
```

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)

--version Display version string and exit

--help Display help and exit

Subommands

search start in a search buffer using the querystring provided as parameter. See also the SEARCH SYNTAX section of notmuch(1) and the output of *alot search –help*.

compose compose a new message See the output of *alot compose –help* for more info on parameters.

2.4.2 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.

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.

CHAPTER 3

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.

3.1 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

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

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: "

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

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 ,

notify_timeout

time in secs to display status messages

Type integer

Default 2

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_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_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_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}

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}"

3.2 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
        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 ,

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

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

3.3 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 --output=recipients date:1Y.. AND from:my@address.org"
        regexp = (\"?(?P<name>.+)\"?)?\s*<(?P<email>.*@.+?)>
        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
```

3.4 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.

3.4.1 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
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

3.4.2 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.

3.5 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:

```
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 (*bodytext*[, *ui= None*, *dbm=None*])
used to manipulate a messages bodytext *before* the editor is called.

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

Return type str

post_edit_translate (*bodytext*[, *ui= None*, *dbm=None*])
used to manipulate a messages bodytext *after* the editor is called

Parameters **bodytext** (*str*) – text representation of mail body 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 **bodytext** – 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

3.6 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/`.

3.6.1 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.

3.6.2 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
  #         |                 |   |                               |               |
  #         v                 v   v                               v               v
  footer = 'bold,underline', '', 'light red, bold, underline', 'light green', 'light red, bold, unde
```

3.6.3 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:

- date
- mailcount
- tags
- authors
- subject

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'
```

3.6.4 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"
```

CHAPTER 4

API and Development

4.1 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.

4.2 Contributing

Development is coordinated entirely via the projects github page especially the issue tracker.

You can send patches to notmuch's mailing list but pull requests on github are preferred. Here are a few more things you should know and check before you send pull requests:

- 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.
  - Make sure your patch doesn't break the API docs. The build service at readthedocs.org is fragile when it comes to new import statements in our code.
  - If you implemented a new feature, update the user manual in `/docs/user` accordingly.

4.3 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.

4.3.1 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=[]*, *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'*)
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']

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.

4.3.2 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

4.3.3 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 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_headers_string(*headers*)
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 **headers** (*list of str*) – headers to extract

get_message_id()
returns messages id (str)

get_message_parts()
returns a list of 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

4.3.4 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*, *attachments=[]*, *sign=False*, *sign_key=None*, *encrypt=False*, *tags=[]*)
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=[]*)
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

4.3.5 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 gpgme.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*)
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=[]*, *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.

4.4 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:

```
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)
```

4.4.1 `UI` - the main component

4.4.2 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.

Available modes are:

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

4.4.3 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 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.

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

4.4.4 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.

4.5 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	~/.notmuchrc or given by command option *-n*	*SettingsManager.get_notmuch_setting*
mailcap – defines shellcommands to handle mime types	~/.mailcap (/etc/mailcap)	*SettingsManager.mailcap_find_match*

4.5.1 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*)
returns `Account` for a given email address (str)

Parameters **address** (*string*) – address to look up

Return type `Account` or None

get_accounts()
returns known accounts

Return type list of `Account`

get_addressbooks(*order=[]*, *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(*path*)
parse alot's config file from path

read_notmuch_config(*path*)
parse notmuch's config file from path

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

4.5.2 Errors

exception `alot.settings.errors.`**ConfigError**
could not parse user config

4.5.3 Utils

alot.settings.utils.**read_config**(*configpath=None*, *specpath=None*, *checks={}*)
get a (validated) config object for given config file path.

Parameters

- **configpath** (*str*) – path to config-file
- **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

4.5.4 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'.

4.5.5 Accounts

class `alot.account.`**Account**(*address=None, aliases=None, realname=None, gpg_key=None, signature=None, signature_filename=None, signature_as_attachment=False, sent_box=None, sent_tags=['sent'], draft_box=None, draft_tags=['draft'], abook=None, sign_by_default=False, **rest*)

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.

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

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

4.5.6 Addressbooks

4.6 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. Ensure the text ends with a newline (\r\n).
3. 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*)

```
>>> humanize_size(1)
'1'
>>> humanize_size(123)
'123'
>>> humanize_size(1234)
'1K'
>>> humanize_size(1234 * 1024)
'1.2M'
>>> humanize_size(1234 * 1024 * 1024)
'1234.0M'
```

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.**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** – the string to interpret. Must conform to :rfc:2368.

Returns pair headers,body. headers is a dict mapping str to lists of (str, body) is a str.

Return type (dict(str–>[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.**safely_get**(*clb*, *E*, *on_error=''*)
returns result of `clb()` and falls back to *on_error* in case exception *E* is raised.

Parameters

- **clb** (*callable*) – function to evaluate
- **E** (*Exception*) – exception to catch
- **on_error** (*str*) – default string returned when exception is caught

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

```
    >>> print(shell_quote("hello"))
    'hello'
    >>> print(shell_quote("hello'there"))
    'hello'"'"'there'
```

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.

```
    >>> authors = u'King Kong, Mucho Muchacho, Jaime Huerta, Flash Gordon'
    >>> print shorten_author_string(authors, 60)
    King Kong, Mucho Muchacho, Jaime Huerta, Flash Gordon
    >>> print shorten_author_string(authors, 40)
```

```
    King, Mucho, Jaime, Flash
    >>> print shorten_author_string(authors, 20)
    King, ..., Jai..., Flash
    >>> print shorten_author_string(authors, 10)
    King, ...
    >>> print shorten_author_string(authors, 2)
    K...
    >>> print shorten_author_string(authors, 1)
    K
```

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.

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 *globals.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'
```

alot.helper.**tag_cmp**(*a*, *b*)
Sorting tags using this function puts all tags of length 1 at the beginning. This groups all tags mapped to unicode characters.

4.7 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

```
    >>> cmd = alot.commands.commandfactory('save --all /foo', mode='thread')
    >>> cmd
    <alot.commands.thread.SaveAttachmentCommand object at 0x272cf10
    >>> cmd.all
    True
    >>> cmd.path
    u'/foo'
```

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))

```
    >>> (cmd, parser, kwargs) = lookup_command('save', 'thread')
    >>> cmd
    <class 'alot.commands.thread.SaveAttachmentCommand'>
```

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={}*, *arguments=[]*)
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:

```
    @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()`.

4.7.1 Globals

4.7.2 Envelope

4.7.3 Bufferlist

4.7.4 Search

4.7.5 Taglist

4.7.6 Thread

4.8 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** – GPGME hash_algo

Return type str

`alot.crypto.`**`decrypt_verify`**(*encrypted*)

Decrypts the given ciphertext string and returns both the signatures (if any) and the plaintext.

Parameters **encrypted** – the mail to decrypt

Returns a tuple (sigs, plaintext) with sigs being a list of a `gpgme.Signature` and plaintext is a *str* holding the decrypted mail

Raises `GPGProblem` if the decryption fails

`alot.crypto.`**`detached_signature_for`**(*plaintext_str*, *key=None*)

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** – text to sign
- **key** – gpgme_key_t object representing the key to use

Return type tuple of gpgme.NewSignature array and str

alot.crypto.**encrypt**(*plaintext_str*, *keys=None*)
Encrypts the given plaintext string and returns a PGP/MIME compatible string

Parameters

- **plaintext_str** – the mail to encrypt
- **key** – gpgme_key_t object representing the key to use

Return type a string holding the encrypted mail

alot.crypto.**get_key**(*keyid*, *validate=False*, *encrypt=False*, *sign=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.

Parameters

- **keyid** – filter term for the keyring (usually a key ID)
- **validate** – validate that returned keyid is valid
- **encrypt** – when validating confirm that returned key can encrypt
- **sign** – when validating confirm that returned key can sign

Return type gpgme.Key

alot.crypto.**hash_key**(*key*)
Returns a hash of the given key. This is a workaround for https://bugs.launchpad.net/pygpgme/+bug/1089865 and can be removed if the missing feature is added to pygpgme

Parameters **key** – the key we want a hash of

Return type a has of the key as string

alot.crypto.**list_keys**(*hint=None*, *private=False*)
Returns a list of all keys containing keyid.

Parameters

- **keyid** – The part we search for
- **private** – Whether secret keys are listed

Return type list

alot.crypto.**verify_detached**(*message*, *signature*)
Verifies whether the message is authentic by checking the signature.

Parameters

- **message** – the message as *str*
- **signature** – a *str* containing an OpenPGP signature

Returns a list of `gpgme.Signature`

Raises `GPGProblem` if the verification fails

CHAPTER 5

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; copiousoutp
```

   Most text based browsers have a dump mode that can be used here.

6. Why are you $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:

- CLI option parsing is done using twisted.usage.Options, and not (as e.g. in-app command parsing) via argparse. The reason is that argparse does not yet offer optional subcommands.
- 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?

   Because it builds on libraries that don't (yet):

   - configobj
   - twisted

   Alot itself can be converted to py3k syntax automatically using 2to3 and I will push those changes as soon as the libs are ready.

CHAPTER 6

Manpage

6.1 Synopsis

```
alot [-r] [-c CONFIGFILE] [-n NOTMUCHCONFIGFILE] [-C {1,16,256}] [-p DB_PATH]
     [-d {debug,info,warning,error}] [-l LOGFILE] [--version] [--help]
     [command]
```

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)

--version Display version string and exit

--help Display help and exit

Subommands

search start in a search buffer using the querystring provided as parameter. See also the SEARCH SYNTAX section of notmuch(1) and the output of *alot search –help*.

compose compose a new message See the output of *alot compose –help* for more info on parameters.

6.2 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.

6.3 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 *?*.

6.4 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

A

B

C

D

E

F

G

H

K

L

M