- To start Neovim, run `nvim` (not `neovim`).
- If something broke after upgrading, check Following-HEAD.
- `:checkhealth` detects and resolves many of the problems in this FAQ. Try it!

General Questions

Where should I put my config (`vimrc`)?

See `:help config`.

You can copy your existing vimrc, or symlink to it. `:help nvim-from-vim`

How stable is the development (pre-release) version?

The unstable (pre-release) version of Neovim ("HEAD", i.e. the `master` branch) is used to aggressively stage new features and changes. It's usually stable, but will occasionally break your workflow. We depend on HEAD users to report "blind spots" that were not caught by automated tests.

Use the stable (release) version for a more predictable experience.

Can I use Ruby-based Vim plugins (e.g. LustyExplorer)?

Yes, starting with Neovim 0.1.5 PR #4980 the legacy Vim `if_ruby` interface is supported.

Can I use Lua-based Vim plugins (e.g. neocomplete)?

No. Starting with Neovim 0.2 PR #4411 Lua is built-in, but the legacy Vim `if_lua` interface is not supported.

How can I use "true color" in the terminal?

Add this to your `init.vim`:

```
set termguicolors
```

- See this gist for more information.
- See this thread for guidance on how to check your system capabilities.

Nvim shows weird symbols (`�[2 q`) when changing modes

This is a bug in your terminal emulator. It happens because Nvim sends cursor-shape termcodes by default, if the terminal appears to be xterm-compatible (`TERM=xterm-256color`).

To workaround the issue, you can:

- Use a different terminal emulator
- Disable `guicursor` in your Nvim config:

```
:set guicursor=
" Workaround some broken plugins which set guicursor indiscriminately.
:autocmd OptionSet guicursor noautocmd set guicursor=
```

See also `:help $TERM` for recommended values of `$TERM`.

How to change cursor *shape* in the terminal?

- For Nvim 0.1.7 or older: see the note about `NVIM_TUI_ENABLE_CURSOR_SHAPE` in `man nvim`.
- For Nvim 0.2 or newer: cursor styling is controlled by the `guicursor` option.
  - To *disable* cursor-styling, set `guicursor` to empty:

    ```
    :set guicursor=
    " Workaround some broken plugins which set guicursor indiscriminately.
    :autocmd OptionSet guicursor noautocmd set guicursor=
    ```

  - If you want a non-blinking cursor, use `blinkon0`. See `:help 'guicursor'`.
  - `guicursor` is enabled by default, unless Nvim thinks your terminal doesn't support it. If you're sure that your terminal supports cursor-shaping, set `guicursor` in your init.vim, as described in `:help 'guicursor'`.
- The Vim terminal options `t_SI` and `t_EI` are ignored, like all other `t_XX` options.
- Old versions of libvte (gnome-terminal, roxterm, terminator, ...) do not support cursor style control codes. #2537

How to change cursor *color* in the terminal?

Cursor styling (shape, color, behavior) is controlled by `guicursor`, even in the terminal. Cursor *color* (as opposed to shape) ***only*** works if `termguicolors` is set.

`:help 'guicursor'` gives an example, but here's a more complicated example which sets different colors in insert-mode and normal-mode:

```
:set termguicolors
:hi Cursor guifg=green guibg=green
:hi Cursor2 guifg=red guibg=red
:set guicursor=n-v-c:block-Cursor/lCursor,i-ci-ve:ver25-Cursor2/lCursor2,r-
cr:hor20,o:hor50
```

Cursor style isn't restored after exiting or suspending and resuming Nvim

Terminals do not provide a way to query the cursor style. Use autocommands to manage the cursor style:

```
au VimEnter,VimResume * set guicursor=n-v-c:block,i-ci-ve:ver25,r-cr:hor20,o:hor50
  \,a:blinkwait700-blinkoff400-blinkon250-Cursor/lCursor
  \,sm:block-blinkwait175-blinkoff150-blinkon175

au VimLeave,VimSuspend * set guicursor=a:block-blinkon0
```

Cursor shape doesn't change in tmux

tmux decides that, not Nvim. See :help tui-cursor-shape for a fix.

See #3165 for discussion.

Cursor flicker in tmux?

If cursor `_` appears and disappears very quickly when opening nvim without a document under tmux, and you set `ctermbg` in `EndOfBuffer` and `Normal`, try setting these to `NONE`:

```
hi EndOfBuffer ctermbg=NONE ctermfg=200 cterm=NONE
hi Normal ctermbg=NONE ctermfg=200 cterm=NONE
```

Is Windows supported?

Yes, starting with the `0.2` release. See the Install page. However, since none of the current maintainers are active Windows users and we have to rely on corresponding support by all our dependencies, this is purely on a reasonable-effort basis and cannot cover all possible platform variants (Windows 10/11, MingW/WSL, scoop/chocolatey, ...) If you rely on Windows support, please consider helping out.

How to use the Windows clipboard from WSL?

To use the Windows clipboard from within WSL, `win32yank.exe` has to be on our `$PATH`.

If Neovim is installed on both Windows and within the WSL distribution, the `win32yank.exe` binary provided by the Neovim Windows installation can be symlinked to a directory included in our `$PATH` so it can be found by Neovim on WSL. Replace `$NEOVIM_WIN_DIR` with the path to our Neovim Windows installation, e.g. `/mnt/c/Program Files/Neovim`. The command can then be symlinked using:

```
sudo ln -s "$NEOVIM_WIN_DIR/bin/win32yank.exe" "/usr/local/bin/win32yank.exe"
```

On some versions of Windows, WSL is unable to execute symbolic links to Windows executables (microsoft/WSL#3999). In that case, use one of the methods in #12113 (comment).

If Neovim is only installed within our WSL distribution, we can just install `win32yank.exe` manually:

```
curl -sLo/tmp/win32yank.zip
https://github.com/equalsraf/win32yank/releases/download/v0.0.4/win32yank-x64.zip
unzip -p /tmp/win32yank.zip win32yank.exe > /tmp/win32yank.exe
chmod +x /tmp/win32yank.exe
sudo mv /tmp/win32yank.exe /usr/local/bin/
```

In either case, don't forget to set Neovim's clipboard to `unnamedplus` using `set clipboard=unnamedplus` to make Neovim use the system's (i.e Window's) clipboard by default.

If it does not work, you can debug it using `:checkhealth` to identify any issues, e.g., there could be xclip installed and taking precedent over win32yank.

See also #6227 for more information.

What happened to --remote and friends?

The code for that family of command-line arguments was removed. It may eventually be reimplemented using the Neovim API, but until then `neovim-remote` can be used instead.

See #1750 for more information.

Runtime issues

Copying to X11 primary selection with the mouse doesn't work

`clipboard=autoselect` is not implemented yet. You may find this *partial* workaround to be useful:

```
vnoremap <LeftRelease> "*ygv
```

Note that this is only a partial workaround. It doesn't work for double-click (word selection) nor triple-click (line selection). But it's better than nothing.

My CTRL-H mapping doesn't work

This was fixed in Neovim **0.2**. If you are running Neovim **0.1.7 or older**, adjust your terminal's "kbs" (key_backspace) terminfo entry:

```
infocmp $TERM | sed 's/kbs=^[hH]/kbs=\\177/' > $TERM.ti
tic $TERM.ti
```

(Feel free to delete the temporary `*.ti` file created after running the above commands).

`<Home>` or some other "special" key doesn't work

Make sure `$TERM` is set correctly.

- For screen or tmux, `TERM` should be `screen-256color` (*Not* `xterm-256color`)
- In other cases if "256" does not appear in the string it's probably wrong. Try `TERM=xterm-256color`.

`:!` and `system()` do weird things with interactive processes

Interactive commands are supported by `:terminal` in Neovim. But `:!` and `system()` do not support interactive commands, primarily because Neovim UIs use stdio for msgpack communication, but also for performance, reliability, and consistency across platforms (see `:help gui-pty`).

See also #1496 and #8217.

Python support isn't working

Run `:checkhealth` in Nvim for automatic diagnosis.

Other hints:

- The python `neovim` module was renamed to `pynvim`. See Following-HEAD.
- If you're using pyenv or virtualenv for the `pynvim` module, you must set `g:python_host_prog` and/or `g:python3_host_prog` to the virtualenv's interpreter path.
- Read `:help provider-python`.
- Be sure you have the **latest version** of the `pynvim` Python module:

  ```
  python -m pip install setuptools
  python -m pip install --upgrade pynvim
  python2 -m pip install --upgrade pynvim
  python3 -m pip install --upgrade pynvim
  ```

- Try with `nvim -u NORC` to make sure your config (`init.vim`) isn't causing a problem. If you get `E117: Unknown function`, that means Neovim can't find its runtime.

`:checkhealth` reports `E5009: Invalid $VIMRUNTIME`

This means `health#check()` couldn't load, which suggests that `$VIMRUNTIME` or `&runtimepath` is broken.

- `$VIMRUNTIME` must point to Neovim's runtime files, not Vim's.
- The `$VIMRUNTIME` directory contents should be readable by the current user.
- Verify that `:echo &runtimepath` contains the $VIMRUNTIME path.
- Check the output of:

```
:call health#check()
:verbose func health#check
```

Neovim can't find its runtime

This is the case if `:help nvim` shows `E149: Sorry, no help for nvim`.

Make sure that `$VIM` and `$VIMRUNTIME` point to Neovim's (as opposed to Vim's) runtime by checking `:echo $VIM` and `:echo $VIMRUNTIME`. This should give something like `/usr/share/nvim` resp. `/usr/share/nvim/runtime`.

Also make sure that you don't accidentally overwrite your runtimepath (`:set runtimepath?`), which includes the above `$VIMRUNTIME` by default (see `:help 'runtimepath'`).

`E518: Unknown option: [option]`

Some very old/unnecessary options have been removed from Neovim. See `:help nvim-features-removed` for the complete list.

Neovim is slow

Use a fast terminal emulator

- kitty
- alacritty

Use an optimized build

`:checkhealth nvim` should report one of these "build types":

```
Build type: RelWithDebInfo
Build type: MinSizeRel
Build type: Release
```

If it reports `Build type: Debug` and you're building Neovim from source, see Building Neovim#optimized-builds.

Colors aren't displayed correctly

Ensure that `$TERM` is set correctly. See :help $TERM for recommended values.

From a shell, run `TERM=xterm-256color nvim`. If colors are displayed correctly, then export that value of `TERM` in your user profile (usually `~/.profile`):

```
export TERM=xterm-256color
```

If you're using `tmux`, instead add this to your `tmux.conf`:

```
set -g default-terminal "tmux-256color"
```

For GNU `screen`, configure your `.screenrc`:

```
term screen-256color
```

Note: Neovim ignores `t_Co` and other terminal codes.

Neovim can't read UTF-8 characters

Run the following from the command line:

```
locale | grep -E '(LANG|LC_CTYPE|LC_ALL)=(.*\.)?(UTF|utf)-?8'
```

If there's no results, then you might not be using a UTF-8 locale. See the following issues: #1601 #1858 #2386

`ESC` in tmux or GNU Screen is delayed

This is a common problem in `tmux` / `screen` (see also tmux/#131). The corresponding timeout needs to be tweaked to a low value (10-20ms).

`.tmux.conf`:

```
set -g escape-time 10
# Or for tmux >= 2.6
set -sg escape-time 10
```

`.screenrc`:

```
maptimeout 10
```

"Why doesn't this happen in Vim?"

It *does* happen (try `vim -N -u NONE`), but *if you hit a key quickly after ESC* then Vim interprets the ESC as ESC instead of ALT (META). You won't notice the delay unless you closely observe the cursor. The tradeoff is that Vim won't understand ALT (META) key-chords, so for example `nnoremap <M-a>` won't work. ALT (META) key-chords always work in Nvim. See also `:help xterm-cursor-keys` in Vim.

Nvim 0.3 mimics the Vim behavior while still fully supporting ALT mappings. See `:help i_ALT`.

`ESC` in GNU Screen is lost when mouse mode is enabled

This happens because of a bug in screen: in mouse mode, screen assumes that `ESC` is part of a mouse sequence and will wait an unlimited time for the rest of the sequence, regardless of `maptimeout`. Until it's fixed in screen, there's no known workaround for this other than double-pressing escape, which causes a single escape to be passed through to Nvim.

Calling `inputlist()`, `echomsg`, ... in filetype plugins and `autocmd` does not work

#10008, #10116, #12288, # vim/vim#4379. This is because Nvim sets `shortmess+=F` by default. Vim behaves the same way with `set shortmes+=F`. There are plans to improve this, but meanwhile as a workaround, use `set`

`shortmess-=F` or use `unsilent` as follows.

```
unsilent let var = inputlist(['1. item1', '2. item2'])
autocmd BufNewFile * unsilent echomsg 'The autocmd has been fired.'
```

`g:clipboard` settings are not used.

If the clipboard provider is already loaded, you will need to reload it after configuration. Use the following configuration.

```
let g:clipboard = { 'name' : … }
if exists('g:loaded_clipboard_provider')
  unlet g:loaded_clipboard_provider
  runtime autoload/provider/clipboard.vim
endif
```

Or, if you want automatic reloading when assigning to `g:clipboard`, set `init.vim` as follows.

```
function! s:clipboard_changed(...) abort
  if exists('g:loaded_clipboard_provider')
    unlet g:loaded_clipboard_provider
  endif
  runtime autoload/provider/clipboard.vim
endfunction

if !exists('s:loaded")
  call dictwatcheradd(g:, 'clipboard', function('s:clipboard_changed'))
endif
let s:loaded = v:true
```

Installation issues

Generating helptags failed

If re-installation fails with `Generating helptags failed`, try removing the previously installed runtime directory (if `CMAKE_INSTALL_PREFIX` is not set during building, the default is `/usr/local/share/nvim`):

```
# rm -r /usr/local/share/nvim
```

Build issues

General build issues

Run `make distclean && make` to rule out a stale build environment causing the failure.

Proxy issues #2482

If your machine is behind a network proxy and you see this error:

```
Error: Failed installing dependency: https://rocks.moonscript.org/penlight-1.3.2-2.rockspec
Error fetching file: Failed downloading http://stevedonovan.github.io/files/penlight-1.3.2-core.zip
```

this can be fixed by setting the `https_proxy` environment variable (for cURL).

Settings in `local.mk` don't take effect

CMake caches build settings, so you might need to run `rm -r build && make` after modifying `local.mk`.

CMake errors

`configure_file Problem configuring file`

This is probably a permissions issue, which can happen if you run `make` as the root user, then later run an unprivileged `make`. To fix this, run `rm -rf build` and try again.

`A suitable Lua interpreter was not found.`

This can be caused by a local LuaRocks installation. Try unsetting the `LUA_PATH` and `LUA_CPATH` environment variables (via `unset`) before building.

Lua packages

The Lua packages required by the build process should be automatically installed by LuaRocks (invoked by CMake automatically). If that fails, it could mean:

- The LuaRocks servers are down.
- Your network is down.
- `unzip` isn't found. In that case LuaRocks will report something like this: `Warning: Failed searching manifest: Failed loading manifest: Failed extracting manifest file`.
- The `$CDPATH` environment variable is interfering with the build, so it should be unset prior to running `make`.

To try a different LuaRocks mirror, create the file `.deps/usr/etc/luarocks/config-5.1.lua` with these contents:

```
rocks_servers={
 "http://luarocks.giga.puc-rio.br/"
}
```

Then run `make cmake`.

Anaconda error

Error message: `anaconda3/bin/x86_64-conda_cos6-linux-gnu-cc: not found`

Solution: `conda install gxx_linux-64` or `conda deactivate`

Develop

Plugins

- nvimdev.nvim: Neomake integration with Lua/C + Neovim codebase
- helpful.vim: get the version of Vim/Neovim where a feature was introduced.
- Neomake: Async linting
- deoplete.nvim: auto-completion
  - deoplete-clang2: clang2 completion support for deoplete.nvim
- coc.nvim: auto-completion, code navigation
- nvim-cmp: auto-completion

Tools

- hererocks (very similar to Python's `virtualenv`) is useful for installing Luarocks, LuaJIT, and Lua:

```
curl -LO
https://raw.githubusercontent.com/luarocks/hererocks/latest/hererocks.py
chmod u+x hererocks.py
# Install LuaJit and LuaRocks 3.0 to the "myenv" directory.
./hererocks.py myenv --luajit latest -r3.0
```

- croissant is a Lua REPL

Debug

Backtrace (Linux)

Core dumps are disabled by default on Ubuntu, CentOS and others. To enable core dumps:

```
ulimit -c unlimited
```

On systemd-based systems getting a backtrace is as easy as:

```
coredumpctl -1 gdb
```

It's an optional tool, so you may need to install it:

```
sudo apt install systemd-coredump
```

The **full backtrace** is most useful, send us the `bt.txt` file:

```
2>&1 coredumpctl -1 gdb | tee -a bt.txt
thread apply all bt full
```

On older systems a `core` file will appear in the current directory. To get a backtrace from the `core` file:

```
gdb build/bin/nvim core 2>&1 | tee backtrace.txt
thread apply all bt full
```

Backtrace (macOS / OSX)

If `nvim` crashes, you can see the backtrace in Console.app (under "Crash Reports" or "User Diagnostic Reports" for older macOS versions).

```
open -a Console
```

You may also want to enable core dumps on macOS. To do this, first make sure the `/cores/` directory exists and is writable:

```
sudo mkdir /cores
sudo chown root:admin /cores
sudo chmod 1775 /cores
```

Then set the core size limit to `unlimited`:

```
ulimit -c unlimited
```

Note that this is done per shell process. If you want to make this the default for all shells, add the above line to your shell's init file (e.g. `~/.bashrc` or similar).

You can then open the core file in `lldb`:

```
lldb -c /cores/core.12345
```

Apple's documentation archive has some other useful information, but note that some of the things on this page are out of date (such as enabling core dumps with `/etc/launchd.conf`).

Using `gdb` to step through functional tests

Use `TEST_TAG` to run tests matching busted tags (of the form `#foo` e.g. `it("test #foo ...", ...)`):

```
GDB=1 TEST_TAG=foo make functionaltest
```

Then, in another terminal:

```
gdb build/bin/nvim
target remote localhost:7777
```

- See also test/functional/helpers.lua.

Using `lldb` to step through unit tests

```
lldb .deps/usr/bin/luajit -- .deps/usr/bin/busted --lpath="./build/?.lua" test/unit/
```

Using `gdb`

To attach to a running `nvim` process with a pid of 1234:

```
gdb -tui -p 1234 build/bin/nvim
```

The `gdb` interactive prompt will appear. At any time you can:

- `break foo` to set a breakpoint on the `foo()` function
- `n` to **step over** the next statement

    - `<Enter>` to repeat the last command
- `s` to **step into** the next statement
- `c` to **continue**
- `finish` to **step out** of the current function
- `p zub` to print the value of `zub`
- `bt` to see a **backtrace** (callstack) from the current location
- `CTRL-x CTRL-a` or `tui enable` to **show a TUI view of the source file** in the current debugging context. This can be extremely useful as it avoids the need for a gdb "frontend".
    - `<up>` and `<down>` to scroll the source file view

gdb "reverse debugging"

- `set record full insn-number-max unlimited`
- `continue` for a bit (at least until `main()` is executed
- `record`
- provoke the bug, then use `revert-next`, `reverse-step`, etc. to rewind the debugger

Using `gdbserver`

You may want to connect multiple `gdb` clients to the same running `nvim` process, or you may want to connect to a remote `nvim` process with a local `gdb`. Using `gdbserver`, you can attach to a single process and control it from multiple `gdb` clients.

Open a terminal and start `gdbserver` attached to `nvim` like this:

```
gdbserver :6666 build/bin/nvim 2> gdbserver.log
```

`gdbserver` is now listening on port 6666. You then need to attach to this debugging session in another terminal:

```
gdb build/bin/nvim
```

Once you've entered `gdb`, you need to attach to the remote session:

```
target remote localhost:6666
```

In case gdbserver puts the TUI as a background process, the TUI can become unable to read input from pty (and receives SIGTTIN signal) and/or output data (SIGTTOU signal). To force the TUI as the foreground process, you can add

```
signal (SIGTTOU, SIG_IGN);
if (!tcsetpgrp(data->input.in_fd, getpid())) {
    perror("tcsetpgrp failed");
}
```

to tui.c:terminfo_start.

Using `gdbserver` in `tmux`

Consider using a custom makefile to quickly start debugging sessions using the `gdbserver` method mentioned above. This example `local.mk` will create the debugging session when you type `make debug`.

```
.PHONY: dbg-start dbg-attach debug build

build:
    @$(MAKE) nvim

dbg-start: build
    @tmux new-window -n 'dbg-neovim' 'gdbserver :6666 ./build/bin/nvim -D'

dbg-attach:
    @tmux new-window -n 'dbg-cgdb' 'cgdb -x gdb_start.sh ./build/bin/nvim'

debug: dbg-start dbg-attach
```

Here `gdb_start.sh` includes `gdb` commands to be called when the debugger starts. It needs to attach to the server started by the `dbg-start` rule. For example:

```
target remote localhost:6666
br main
```

Log file location

Nvim's low-level logs are written to `~/.local/share/nvim/log` (usually; see `:help $NVIM_LOG_FILE`). Debug builds write INFO-level messages to this log file. You can specify the location with the `$NVIM_LOG_FILE` environment variable. Non-debug builds only log ERROR-level messages.

Design

Why not use JSON for RPC?

- JSON cannot easily/efficiently handle binary data
- JSON specification is ambiguous: http://seriot.ch/parsing_json.php

Why embed Lua instead of X?

- Lua is a very small language, ideal for embedding. The biggest advantage of Python/Ruby/etc is their huge collection of libraries, but that isn't relevant for Nvim, where Nvim *is* the "batteries included" library: introducing another stdlib would be redundant.
- Lua 5.1 is a *complete* language: the syntax is frozen. This is great for backwards compatibility.
- **Nvim also uses Lua *internally* as an alternative to C.** Extra performance is useful there, as opposed to a slow language like Python.
- LuaJIT is one of the fastest runtimes on the planet. It is at least 10x faster than Python.
- Python/JS cost more than Lua in terms of size and portability, and there are already numerous Python/JS-based editors. So Python/JS would make Nvim bigger and less portable, in exchange for a non-differentiating feature.

See also:

- Why Lua
- Redis and scripting
- The Design of Lua
- Luajit performance, performance guide, luajit not yet implemented

- Discussion of JavaScript vs Lua
- Discussion Python embedding

Why Lua 5.1 instead of Lua 5.3+?

Lua 5.1 is a different language than 5.3. The Lua org makes breaking changes with every new version, so even if we switched (not upgraded, but *switched*) to 5.3 we gain nothing when they create the next new language in 5.4, 5.5, etc. And we would lose LuaJit, which is far more valuable than Lua 5.3+.

Lua 5.1 is a complete language. To "upgrade" it, add libraries, not syntax. Nvim itself already is a pretty good "stdlib" for Lua, and we will continue to grow and enhance it. Changing the rules of Lua gains nothing in this context.

Will Neovim translate VimL to Lua, instead of executing VimL directly?

Update (2016): PR #243 implements the VimL-to-Lua translator. But it is blocked by technical concerns. Much of the work in that PR was re-used/re-purposed (viz. `typval_T` / `vim_to_object` refactor, `eval.c` refactor).

Are plugin authors encouraged to port their plugins from Vimscript to Lua? Do you plan on supporting Vimscript indefinitely? (#1152)

We don't anticipate any reason to deprecate Vimscript, which is a valuable DSL for text-editing tasks. Maintaining Vimscript compatibility is less costly than a mass migration of existing Vim plugins.

Porting from Vimscript to Lua just for the heck of it gains nothing. Neovim is emphatically a *fork of Vim* in order to leverage the work already spent on thousands of Vim plugins, while enabling *new* types of plugins and integrations.