Neovim Documentation

Neovim

Jan 26, 2023

Getting started

Implements support for Python plugins in Neovim. Also works as a library for connecting to and scripting Neovim processes through its msgpack-rpc API.

CHAPTER 1

Installation

The Neovim Python client supports Python 2.7, and 3.4 or later.

1.1 Using pip

You can install the package without being root by adding the `--user` flag:

```
pip2 install --user pynvim
pip3 install --user pynvim
```

Note: If you only use one of python2 or python3, it is enough to install that version.

If you follow Neovim HEAD, make sure to upgrade `pynvim` when you upgrade Neovim:

```
pip2 install --upgrade pynvim
pip3 install --upgrade pynvim
```

1.2 Install from source

Clone the repository somewhere on your disk and enter to the repository:

```
git clone https://github.com/neovim/pynvim.git
cd pynvim
```

Now you can install it on your system:

```
pip2 install .
pip3 install .
```

CHAPTER 2

Python Plugin API

Neovim has a new mechanism for defining plugins, as well as a number of extensions to the python API. The API extensions are accessible no matter if the traditional `:python` interface or the new mechanism is used, as discussed on *Remote (new-style) plugins*.

2.1 Nvim API methods: `vim.api`

Exposes Neovim API methods. For instance to call `nvim_strwidth`:

```
result = vim.api.strwidth("some text")
```

Note the initial `nvim_` is not included. Also, object methods can be called directly on their object:

```
buf = vim.current.buffer
length = buf.api.line_count()
```

calls `nvim_buf_line_count`. Alternatively msgpack requests can be invoked directly:

```
result = vim.request("nvim_strwith", "some text")
length = vim.request("nvim_buf_line_count", buf)
```

Both `vim.api` and `vim.request` can take an `async_=True` keyword argument to instead send a msgpack notification. Nvim will execute the API method the same way, but python will not wait for it to finish, so the return value is unavailable.

2.2 Vimscript functions: `vim.funcs`

Exposes vimscript functions (both builtin and global user defined functions) as a python namespace. For instance to set the value of a register:

```
vim.funcs.setreg('0', ["some", "text"], 'l')
```

These functions can also take the `async_=True` keyword argument, just like API methods.

2.3 Lua integration

Python plugins can define and invoke lua code in Nvim's in-process lua interpreter. This is especially useful in asynchronous contexts, where an async event handler can schedule a complex operation with many api calls to be executed by nvim without interleaved processing of user input or other event sources (unless requested).

The recommended usage is the following pattern. First use `vim.exec_lua(code)` to define a module with lua functions:

```
vim.exec_lua("""
   local a = vim.api
   local function add(a,b)
       return a+b
   end

   local function buffer_ticks()
      local ticks = {}
      for _, buf in ipairs(a.nvim_list_bufs()) do
          ticks[#ticks+1] = a.nvim_buf_get_changedtick(buf)
      end
      return ticks
   end

   _testplugin = {add=add, buffer_ticks=buffer_ticks}
""")
```

Alternatively, place the code in `/lua/testplugin.lua` under your plugin repo root, and use `vim.exec_lua("_testplugin = require('testplugin')")`. In both cases, replace `testplugin` with a unique string based on your plugin name.

Then, the module can be accessed as `vim.lua._testplugin`.

```
mod = vim.lua._testplugin
mod.add(2,3) # => 5
mod.buffer_ticks() # => list of ticks
```

These functions can also take the `async_=True` keyword argument, just like API methods.

It is also possible to pass arguments directly to a code block. Using `vim.exec_lua(code, args...)`, the arguments will be available in lua as `...`.

2.4 Async calls

The API is not thread-safe in general. However, `vim.async_call` allows a spawned thread to schedule code to be executed on the main thread. This method could also be called from `:python` or a synchronous request handler, to defer some execution that shouldn't block Neovim:

```
:python vim.async_call(myfunc, args...)
```

Note that this code will still block the plugin host if it does long-running computations. Intensive computations should be done in a separate thread (or process), and `vim.async_call` can be used to send results back to Neovim.

Some methods accept an `async_` keyword argument: `vim.eval`, `vim.command`, `vim.request` as well as the `vim.funcs`, `vim.api` and ``vim.lua` wrappers. When `async_=True` is passed the client will not wait for Neovim to complete the request (which also means that the return value is unavailable).

CHAPTER 3

Remote (new-style) plugins

Neovim allows Python 3 plugins to be defined by placing python files or packages in `rplugin/python3/` (in a `runtimepath` folder). Python 2 rplugins are also supported and placed in `rplugin/python/`, but are considered deprecated. Further added library features will only be available on Python 3. Rplugins follow the structure of this example:

```
import pynvim

@pynvim.plugin
class TestPlugin(object):

    def __init__(self, nvim):
        self.nvim = nvim

    @pynvim.function('TestFunction', sync=True)
    def testfunction(self, args):
        return 3

    @pynvim.command('TestCommand', nargs='*', range='')
    def testcommand(self, args, range):
        self.nvim.current.line = ('Command with args: {}, range: {}'
                                  .format(args, range))

    @pynvim.autocmd('BufEnter', pattern='*.py', eval='expand("<afile>")', sync=True)
    def on_bufenter(self, filename):
        self.nvim.out_write('testplugin is in ' + filename + '\n')
```

If `sync=True` is supplied Neovim will wait for the handler to finish (this is required for function return values), but by default handlers are executed asynchronously.

Normally async handlers (`sync=False`, the default) are blocked while a synchronous handler is running. This ensures that async handlers can call requests without Neovim confusing these requests with requests from a synchronous handler. To execute an asynchronous handler even when other handlers are running, add `allow_nested=True` to the decorator. This handler must then not make synchronous Neovim requests, but it can make asynchronous requests, i.e. passing `async_=True`.

Note: Plugin objects are constructed the first time any request of the class is invoked. Any error in `__init__` will be reported as an error from this first request. A well-behaved rplugin will not start executing until its functionality is requested by the user. Initialize the plugin when user invokes a command, or use an appropriate autocommand, e.g. FileType if it makes sense to automatically start the plugin for a given filetype. Plugins must not invoke API methods (or really do anything with non-trivial side-effects) in global module scope, as the module might be loaded as part of executing *UpdateRemotePlugins*.

You need to run `:UpdateRemotePlugins` in Neovim for changes in the specifications to have effect. For details see `:help remote-plugin` in Neovim.

For local plugin development, it's a good idea to use an isolated vimrc:

```
cat vimrc
let &runtimepath.=','.escape(expand('<sfile>:p:h'), '\,')
```

That appends the current directory to the Nvim runtime path so Nvim can find your plugin. You can now invoke Neovim:

```
nvim -u ./vimrc
```

Then run `:UpdateRemotePlugins` and your plugin should be activated.

In case you run into some issues, you can list your loaded plugins from inside Neovim by running `:scriptnames` like so.:

```
:scriptnames
1: ~/path/to/your/plugin-git-repo/vimrc
2: /usr/share/nvim/runtime/filetype.vim
...
25: /usr/share/nvim/runtime/plugin/zipPlugin.vim
26: ~/path/to/your/plugin-git-repo/plugin/lucid.vim
```

You can also inspect the `&runtimepath` like this:

```
:set runtimepath
runtimepath=~/.config/nvim,/etc/xdg/nvim,~/.local/share/nvim/site,...,
,~/g/path/to/your/plugin-git-repo

" Or alternatively
:echo &rtp
```

CHAPTER 4

Plugin Decorators

Plugin decorators.

4.1 Plugin

pynvim.plugin.**plugin**(*cls*)
Tag a class as a plugin.

This decorator is required to make the class methods discoverable by the plugin_load method of the host.

4.2 Command

pynvim.plugin.**command**(*name*, *nargs=0*, *complete=None*, *range=None*, *count=None*, *bang=False*, *register=False*, *sync=False*, *allow_nested=False*, *eval=None*)
Tag a function or plugin method as a Nvim command handler.

4.3 Autocmd

pynvim.plugin.**autocmd**(*name*, *pattern='*'*, *sync=False*, *allow_nested=False*, *eval=None*)
Tag a function or plugin method as a Nvim autocommand handler.

4.4 Function

pynvim.plugin.**function**(*name*, *range=False*, *sync=False*, *allow_nested=False*, *eval=None*)
Tag a function or plugin method as a Nvim function handler.

CHAPTER 5

Nvim Class

An instance of this class is used by remote plugins.

class pynvim.api.**Nvim**(*session*, *channel_id*, *metadata*, *types*, *decode=False*, *err_cb=None*)
Class that represents a remote Nvim instance.

This class is main entry point to Nvim remote API, it is a wrapper around Session instances.

The constructor of this class must not be called directly. Instead, the *from_session* class method should be used to create the first instance from a raw *Session* instance.

Subsequent instances for the same session can be created by calling the *with_decode* instance method to change the decoding behavior or *SubClass.from_nvim(nvim)* where *SubClass* is a subclass of *Nvim*, which is useful for having multiple *Nvim* objects that behave differently without one affecting the other.

When this library is used on python3.4+, asyncio event loop is guaranteed to be used. It is available as the "loop" attribute of this class. Note that asyncio callbacks cannot make blocking requests, which includes accessing state-dependent attributes. They should instead schedule another callback using nvim.async_call, which will not have this restriction.

async_call(*fn*, **args*, ***kwargs*)
Schedule *fn* to be called by the event loop soon.

This function is thread-safe, and is the only way code not on the main thread could interact with nvim api objects.

This function can also be called in a synchronous event handler, just before it returns, to defer execution that shouldn't block neovim.

call(*name*, **args*, ***kwargs*)
Call a vimscript function.

chdir(*dir_path*)
Run os.chdir, then all appropriate vim stuff.

close()
Close the nvim session and release its resources.

command(*string*, ***kwargs*)
Execute a single ex command.

command_output(*string*)
Execute a single ex command and return the output.

err_write(*msg*, ***kwargs*)
Print *msg* as an error message.

The message is buffered (won't display) until linefeed ("n").

eval(*string*, ***kwargs*)
Evaluate a vimscript expression.

exec_lua(*code*, **args*, ***kwargs*)
Execute lua code.

Additional parameters are available as ... inside the lua chunk. Only statements are executed. To evaluate an expression, prefix it with *return*: *return my_function(...)*

There is a shorthand syntax to call lua functions with arguments:

nvim.lua.func(1,2) nvim.lua.mymod.myfunction(data, async_=True)

is equivalent to

nvim.exec_lua("return func(...)", 1, 2) nvim.exec_lua("mymod.myfunction(...)", data, async_=True)

Note that with *async_=True* there is no return value.

feedkeys(*keys*, *options=''*, *escape_csi=True*)
Push *keys* to Nvim user input buffer.

Options can be a string with the following character flags: - 'm': Remap keys. This is default. - 'n': Do not remap keys. - 't': Handle keys as if typed; otherwise they are handled as if coming

from a mapping. This matters for undo, opening folds, etc.

foreach_rtp(*cb*)
Invoke *cb* for each path in 'runtimepath'.

Call the given callable for each path in 'runtimepath' until either callable returns something but None, the exception is raised or there are no longer paths. If stopped in case callable returned non-None, vim.foreach_rtp function returns the value returned by callable.

classmethod from_nvim(*nvim*)
Create a new Nvim instance from an existing instance.

classmethod from_session(*session*)
Create a new Nvim instance for a Session instance.

This method must be called to create the first Nvim instance, since it queries Nvim metadata for type information and sets a SessionHook for creating specialized objects from Nvim remote handles.

input(*bytes*)
Push *bytes* to Nvim low level input buffer.

Unlike *feedkeys()*, this uses the lowest level input buffer and the call is not deferred. It returns the number of bytes actually written(which can be less than what was requested if the buffer is full).

list_runtime_paths()
Return a list of paths contained in the 'runtimepath' option.

new_highlight_source()
Return new src_id for use with Buffer.add_highlight.

next_message()
Block until a message(request or notification) is available.

If any messages were previously enqueued, return the first in queue. If not, run the event loop until one is received.

out_write(*msg*, ***kwargs*)
Print *msg* as a normal message.

The message is buffered (won't display) until linefeed ("n").

quit(*quit_command='qa!'*)
Send a quit command to Nvim.

By default, the quit command is 'qa!' which will make Nvim quit without saving anything.

replace_termcodes(*string*, *from_part=False*, *do_lt=True*, *special=True*)
Replace any terminal code strings by byte sequences.

The returned sequences are Nvim's internal representation of keys, for example:

<esc> -> 'x1b' <cr> -> 'r' <c-l> -> 'x0c' <up> -> 'x80ku'

The returned sequences can be used as input to *feedkeys*.

request(*name*, **args*, ***kwargs*)
Send an API request or notification to nvim.

It is rarely needed to call this function directly, as most API functions have python wrapper functions. The *api* object can be also be used to call API functions as methods:

vim.api.err_write('ERRORn', async_=True) vim.current.buffer.api.get_mark('.')

is equivalent to

vim.request('nvim_err_write', 'ERRORn', async_=True) vim.request('nvim_buf_get_mark', vim.current.buffer, '.')

Normally a blocking request will be sent. If the *async_* flag is present and True, a asynchronous notification is sent instead. This will never block, and the return value or error is ignored.

run_loop(*request_cb*, *notification_cb*, *setup_cb=None*, *err_cb=None*)
Run the event loop to receive requests and notifications from Nvim.

This should not be called from a plugin running in the host, which already runs the loop and dispatches events to plugins.

stop_loop()
Stop the event loop being started with *run_loop*.

strwidth(*string*)
Return the number of display cells *string* occupies.

Tab is counted as one cell.

subscribe(*event*)
Subscribe to a Nvim event.

ui_attach(*width*, *height*, *rgb=None*, ***kwargs*)
Register as a remote UI.

After this method is called, the client will receive redraw notifications.

ui_detach()
Unregister as a remote UI.

ui_try_resize(*width*, *height*)
Notify nvim that the client window has resized.

If possible, nvim will send a redraw request to resize.

unsubscribe(*event*)
Unsubscribe to a Nvim event.

with_decode(*decode=True*)
Initialize a new Nvim instance.

CHAPTER 6

Buffer Class

class pynvim.api.**Buffer**(*session*, *code_data*)

A remote Nvim buffer.

add_highlight(*hl_group*, *line*, *col_start=0*, *col_end=-1*, *src_id=-1*, *async_=None*, ***kwargs*)

Add a highlight to the buffer.

append(*lines*, *index=-1*)

Append a string or list of lines to the buffer.

clear_highlight(*src_id*, *line_start=0*, *line_end=-1*, *async_=None*, ***kwargs*)

Clear highlights from the buffer.

mark(*name*)

Return (row, col) tuple for a named mark.

name

Get the buffer name.

number

Get the buffer number.

range(*start*, *end*)

Return a *Range* object, which represents part of the Buffer.

update_highlights(*src_id*, *hls*, *clear_start=0*, *clear_end=-1*, *clear=False*, *async_=True*)

Add or update highlights in batch to avoid unnecessary redraws.

A *src_id* must have been allocated prior to use of this function. Use for instance *nvim.new_highlight_source()* to get a src_id for your plugin.

hls should be a list of highlight items. Each item should be a list or tuple on the form *("GroupName", linenr, col_start, col_end)* or *("GroupName", linenr)* to highlight an entire line.

By default existing highlights are preserved. Specify a line range with clear_start and clear_end to replace highlights in this range. As a shorthand, use clear=True to clear the entire buffer before adding the new highlights.

valid
Return True if the buffer still exists.

CHAPTER 7

Window Class

class pynvim.api.**Window**(*session*, *code_data*)

A remote Nvim window.

buffer

Get the *Buffer* currently being displayed by the window.

col

0-indexed, on-screen window position(col) in display cells.

cursor

Get the (row, col) tuple with the current cursor position.

height

Get the window height in rows.

number

Get the window number.

row

0-indexed, on-screen window position(row) in display cells.

tabpage

Get the *Tabpage* that contains the window.

valid

Return True if the window still exists.

width

Get the window width in rows.

CHAPTER 8

Tabpage Class

class pynvim.api.**Tabpage**(**args*)

A remote Nvim tabpage.

number

Get the tabpage number.

valid

Return True if the tabpage still exists.

window

Get the *Window* currently focused on the tabpage.

CHAPTER 9

Development

If you change the code, you need to run:

```
pip2 install .
pip3 install .
```

for the changes to have effect. Alternatively you could execute Neovim with the `$PYTHONPATH` environment variable:

```
PYTHONPATH=/path/to/pynvim nvim
```

But note this is not completely reliable, as installed packages can appear before `$PYTHONPATH` in the python search path.

You need to rerun this command if you have changed the code, in order for Neovim to use it for the plugin host.

To run the tests execute:

```
python -m pytest
```

This will run the tests in an embedded instance of Neovim, with the current directory added to `sys.path`.

If you want to test a different version than `nvim` in `$PATH` use:

```
NVIM_CHILD_ARGV='["/path/to/nvim", "-u", "NONE", "--embed", "--headless"]' pytest
```

Alternatively, if you want to see the state of nvim, you could use:

```
export NVIM_LISTEN_ADDRESS=/tmp/nvimtest
xterm -e "nvim -u NONE"&
python -m pytest
```

But note you need to restart Neovim every time you run the tests! Substitute your favorite terminal emulator for `xterm`.

9.1 Troubleshooting

You can run the plugin host in Neovim with logging enabled to debug errors:

```
NVIM_PYTHON_LOG_FILE=logfile NVIM_PYTHON_LOG_LEVEL=DEBUG nvim
```

As more than one Python host process might be started, the log filenames take the pattern `logfile_pyX_KIND` where `X` is the major python version (2 or 3) and `KIND` is either "rplugin" or "script" (for the `:python[3]` script interface).

If the host cannot start at all, the error could be found in `~/.nvimlog` if `nvim` was compiled with logging.

9.2 Usage through the Python REPL

A number of different transports are supported, but the simplest way to get started is with the python REPL. First, start Neovim with a known address (or use the `$NVIM_LISTEN_ADDRESS` of a running instance):

```
NVIM_LISTEN_ADDRESS=/tmp/nvim nvim
```

In another terminal, connect a python REPL to Neovim (note that the API is similar to the one exposed by the python-vim bridge):

```
>>> from pynvim import attach
# Create a python API session attached to unix domain socket created above:
>>> nvim = attach('socket', path='/tmp/nvim')
# Now do some work.
>>> buffer = nvim.current.buffer # Get the current buffer
>>> buffer[0] = 'replace first line'
>>> buffer[:] = ['replace whole buffer']
>>> nvim.command('vsplit')
>>> nvim.windows[1].width = 10
>>> nvim.vars['global_var'] = [1, 2, 3]
>>> nvim.eval('g:global_var')
[1, 2, 3]
```

You can embed Neovim into your python application instead of binding to a running neovim instance:

```
>>> from pynvim import attach
>>> nvim = attach('child', argv=["/bin/env", "nvim", "--embed", "--headless"])
```

The tests can be consulted for more examples.

Python Module Index

p

Index

T

U

V

W