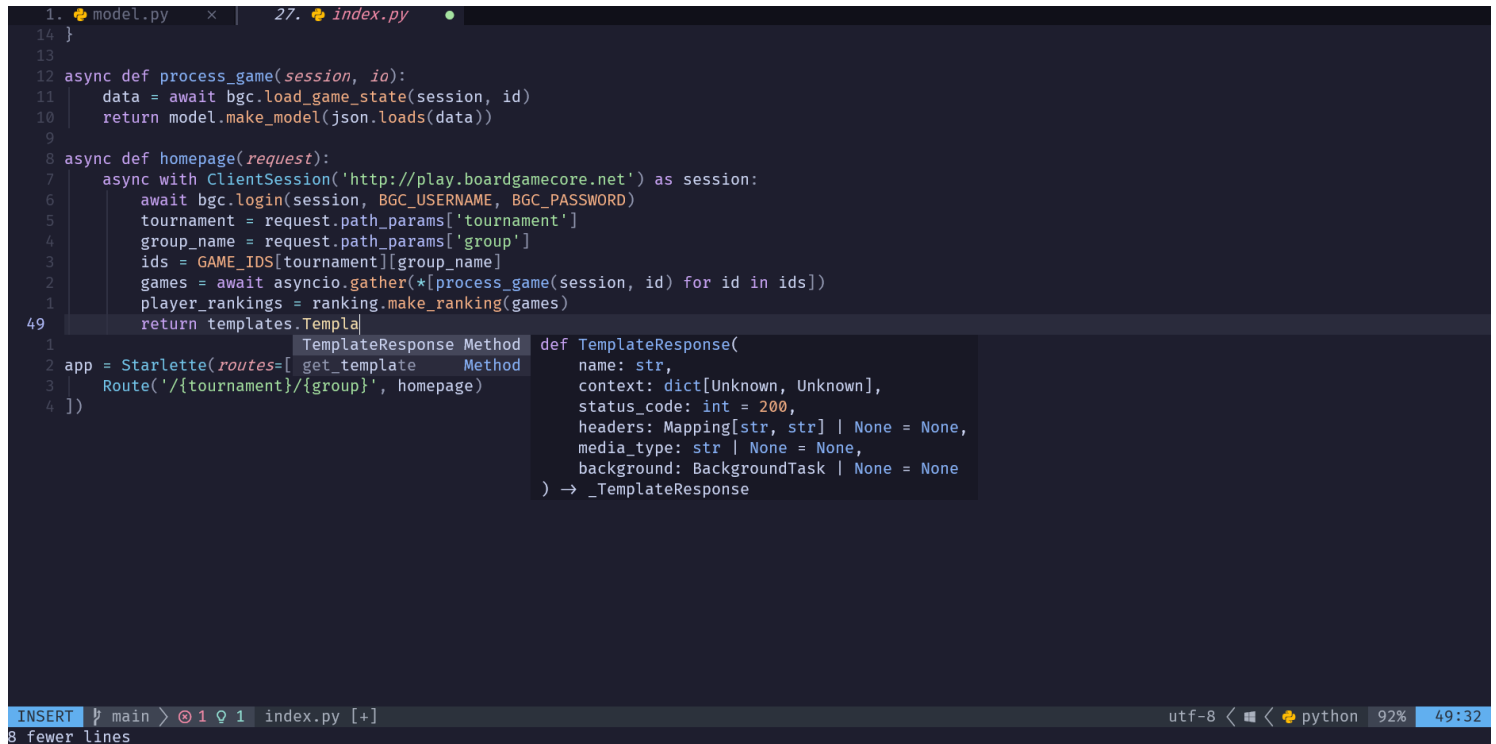


# Configuring NeoVim as a Python IDE

Siddharta Govindaraj : 12-16 minutes : 6/15/2023



```
1. model.py x | 27. index.py
14 }
13
12 async def process_game(session, id):
11     data = await bgc.load_game_state(session, id)
10     return model.make_model(json.loads(data))
9
8 async def homepage(request):
7     async with ClientSession('http://play.boardgamecore.net') as session:
6         await bgc.login(session, BGC_USERNAME, BGC_PASSWORD)
5         tournament = request.path_params['tournament']
4         group_name = request.path_params['group']
3         ids = GAME_IDS[tournament][group_name]
2         games = await asyncio.gather(*[process_game(session, id) for id in ids])
1         player_rankings = ranking.make_ranking(games)
49     return templates.TemplateResponse(Method)
1 app = Starlette(routes=[get_template(Method)
3     Route("/{tournament}/{group}', homepage)
4 ])
```

```
def TemplateResponse(
    name: str,
    context: dict[Unknown, Unknown],
    status_code: int = 200,
    headers: Mapping[str, str] | None = None,
    media_type: str | None = None,
    background: BackgroundTask | None = None
) -> _TemplateResponse
```

INSERT | main > 1 1 index.py [+]  
8 fewer lines

utf-8 < < python 92% 49:32

The article below has been republished from [/home/siddhi](#) with the permission of the author.

Apart from [VSCode](#) and [PyCharm](#), [NeoVim](#) (and Vim more generally) is probably the third most popular programming editor / IDE. One reason why developers like NeoVim is that it is very customisable. You can make it behave like a pure text editor, or customise it to a full blown IDE with debugging support and other features.

While other IDEs tend to be heavyweight and take up a lot of memory / CPU, you can configure NeoVim to have just the features that you want, so it loads really fast, while also fits to your development style.

All that comes with a cost: You have to learn and configure everything. You don't get a default starting experience.

In this article I am going to go through and [explain my configuration](#) step-by-step. I have a terrible memory, so this post will also serve as a guide when I inevitably need to look through this file in the future.

A couple of notes:

- I am using NeoVim version 0.9.0. Many of the features here require at least this version of NeoVim
- My setup is for windows. Most of the configuration is common for all operating systems, but I have some windows specific bits in there. I will mention these when I get to those parts
- This is my setup. You may have different requirements and may not want the exact same setup that I do. That's the point of NeoVim. For example, I don't use a debugger, so I haven't configured it

here. Still, it is fairly easy to take this config as a base and add make the required changes.

There are some starter configurations available that give a good out-of-the-box development setup. You can try them out. Here are some of the popular ones:

- [LazyVim](#)
- [AstroVim](#)
- [Kickstart.vim](#)
- [LunarVim](#)

If you have never used NeoVim before, I'd suggest looking through some of those to get started with using NeoVim. You can start using NeoVim without messing around with the configuration.

Eventually, you will want a configuration that is customised to your preferred way of working. In this article, I'll go through my configuration step by step, explaining it for someone who is new to customising NeoVim.

You can find my configuration file on GitHub here: [my neovim configuration for python](#).

## Setup

The repository contains a file `init.lua`. This is the main configuration file that is needed. In addition, there are a few other files. To use this configuration, you would copy all these files into your nvim directory (in windows that is located at `C:\Users\<your user>\AppData\Local\nvim`)

The configuration is written in Lua programming language. You don't need to go and learn Lua as it's a simple language and most of the code you see here should be understandable even without knowing Lua. If you need to refer something, check out the [Programming in Lua reference](#).

Right, let us open up `init.lua`. It starts out with a set of comments.

The first comment is for setting up the `tabstop`, `shiftwidth` and `expandtab` settings for this file. These settings override the indentation settings for this particular file alone.

After that I have some comments to remind me what all external tools I need to install.

- I use Windows as my primary OS. [Windows Terminal](#) for the terminal and [pwsh](#) for the shell
- Some of the NeoVim plugins require C compilation. Windows does not have C compilers by default, so I install mingw toolchain distributed by [msys2](#)
- Next are [ripgrep](#) and [fd](#). These tools are used by plugins later in the configuration
- Finally, [win32yank](#) integrates NeoVim with the windows clipboard

There are many plugin managers for NeoVim. This following code block loads the [lazy.nvim](#) plugin manager.

## Options

The next section sets a bunch of Vim options.

I have space set as my leader key. I used to use forward slash as the leader key for a long time. When I tried out Kickstart it set leader to space by default and I found that very convenient. I also disable space

to have no effect in normal and visual mode (in case I press space and dont follow it up with a key sequence)

Then come a set of options specific to configuring NVim terminal for pwsh

Normally, you have to come out of escape mode in the terminal by pressing `<Ctrl-\\><Ctrl-n>`, which is quite inconvenient. So I remap that to `Esc`. Similarly I remap `Ctrl-w` to directly switch out of the terminal split. The next few lines configure this

I also have a mapping to minimise the terminal split. I use this when I am running `pytest` in watch mode. It allows me to see the test success / failure state at the bottom of the screen. (For an example of this workflow, [check out this video](#))

## Plugins

Next comes all the plugin configuration. `lazy.nvim` allows us to create separate plugin files, which is a good idea for organising a complex configuration. For now, everything is configured in this one file.

## Theme

For years I used `solarized` dark mode for both Vim and NeoVim. Right now I'm using `catppuccin`. I also load `devicons` here as its used by many plugins.

Here is a look at the catppuccin theme

```
24     "GroupB": [106915, 106916, 106928, 106929, 106918, 106961, 106920],
23     "GroupC": [106912, 106966, 106957, 106967, 106968, 106969, 106963],
22     "GroupD": [106953, 106954, 106955, 106956, 106926, 106909, 106908],
21     "GroupE": [107694, 107722, 107709, 107695, 107710, 107696, 107697],
20     "GroupF": [107701, 107706, 107702, 107707, 107703, 107704, 107810],
19     "Final": [108119]
18   }
17 }
16
15 async def process_game(session, id):
14     data = await bgc.load_game_state(session, id)
13     return model.make_model(json.loads(data))
12
11 async def homepage(request):
10     async with ClientSession('http://play.boardgamecore.net') as session:
9         await bgc.login(session, BGC_USERNAME, BGC_PASSWORD)
8         tournament = request.path_params['tournament']
7         group_name = request.path_params['group']
6         ids = GAME_IDS[tournament][group_name]
5         games = await asyncio.gather(*[process_game(session, id) for id in ids])
4         player_rankings = ranking.make_ranking(games)
3         return templates.TemplateResponse(
2             'dashboard.html', {
1             'request': request,
52 ||         'group_name': group_name,
1             'games': games,
2             'ranking': player_rankings
3         },
4         headers={'Cache-Control': 'max-age=900, public'})
5     )
6
7 app = Starlette(routes=[
8     Route("/{tournament}/{group}", homepage)
9 ])
NORMAL | main | index.py | utf-8 | python | 85% | 52:1
```

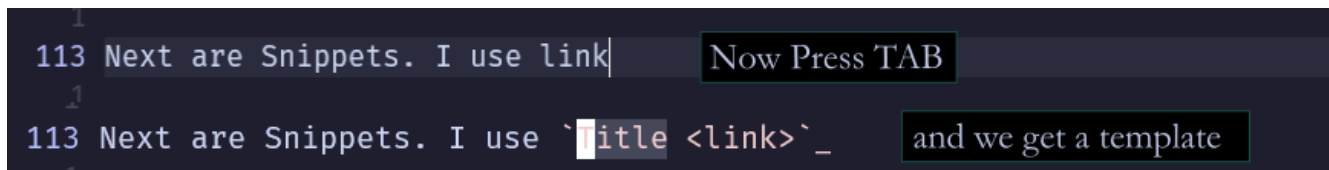
NeoVim catppuccin theme

## Snippets

Next are Snippets. I use `LuaSnip`

LuaSnip supports a few different formats: SnipMate, VSCode or pure Lua. I've gone with pure Lua snippets which I put in the `./snippets` folder.

I mainly use snippets for writing in [RST](#) format. Here is an example



LuaSnip in action

And here is the Lua snippet that I configured for that (s means add snippet, t stands for text node, i is an insert node where the user can edit the template)

## Language Server Protocol

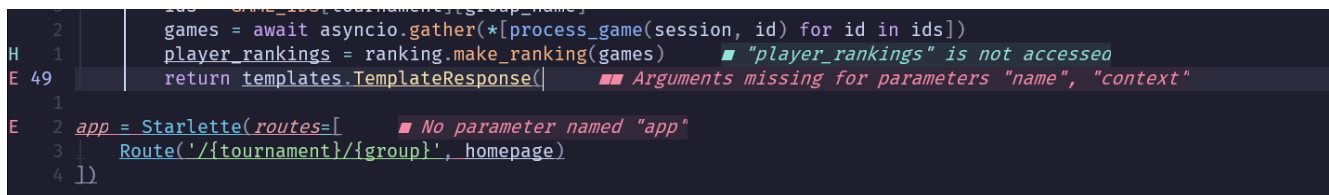
Now comes the big one: Language Server Protocol (LSP). The LSP server will analyse our code. NeoVim will communicate with the LSP server to get autocomplete suggestions and code diagnostics. I am using [pyright](#) for the LSP server. This open source LSP server is developed by Microsoft. A closed source derivative is used in VS Code.

This part is a little complicated.

First, we are installing [Mason](#). Mason allows us to install / uninstall / manage all our LSP servers from within NeoVim. So we don't need to install pyright separately on the terminal. Next we install [nvim-lspconfig](#) and its Mason interface [mason-lspconfig](#).

After that we load `cmp_nvim_lsp`. This is our autocomplete plugin (more about it in the next section). We ask it what capabilities it requires from the LSP server. Then we tell Mason to make sure pyright is installed, and it gets started with the required list of capabilities.

Once set up, you will get LSP diagnostics in NeoVim



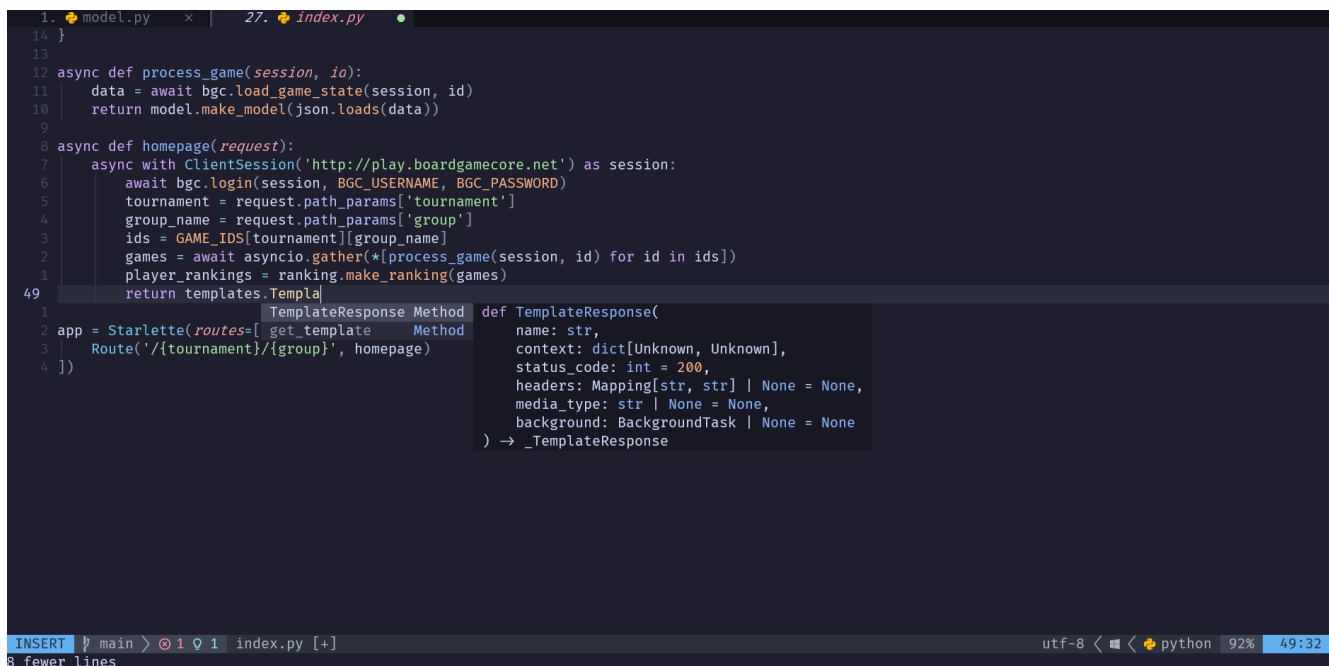
LSP Diagnostics

## Autocomplete

Well, if you thought the LSP configuration was complicated, wait till we get to autocomplete. I am using [hrsh7th/ncim-cmp](#) as the autocomplete plugin. This plugin can autocomplete from many different sources. I have it configured to autocomplete from LSP (via [cmp-nvim-lsp](#)) as well as LuaSnip snippets (via [cmp\\_luasnip](#)).

Most of this configuration is setting up the keys for autocomplete. I use TAB to trigger autocomplete. Since TAB is also a normal key used for indentation and moving between sections of a LuaSnip snippet, we need to write some code so it triggers correctly. Essentially, if you press TAB immediately after any text, the plugin will pop up the autocomplete menu. If the menu is already open, then it will cycle through the items in the autocomplete menu. Otherwise it will fall back to the default behaviour. Same for Shift-TAB in reverse order.

This is what it looks like in action



```
1. model.py x | 27. index.py
14 }
13
12 async def process_game(session, id):
11     data = await bgc.load_game_state(session, id)
10     return model.make_model(json.loads(data))
9
8 async def homepage(request):
7     async with ClientSession('http://play.boardgamecore.net') as session:
6         await bgc.login(session, BGC_USERNAME, BGC_PASSWORD)
5         tournament = request.path_params['tournament']
4         group_name = request.path_params['group']
3         ids = GAME_IDS[tournament][group_name]
2         games = await asyncio.gather(*[process_game(session, id) for id in ids])
1         player_rankings = ranking.make_ranking(games)
49     return templates.TemplateResponse(Method)
1
2 app = Starlette(routes=[get_template(Method)
3     Route('/{tournament}/{group}', homepage)
4 ])
def TemplateResponse(
    name: str,
    context: dict[Unknown, Unknown],
    status_code: int = 200,
    headers: Mapping[str, str] | None = None,
    media_type: str | None = None,
    background: BackgroundTask | None = None
) -> _TemplateResponse
```

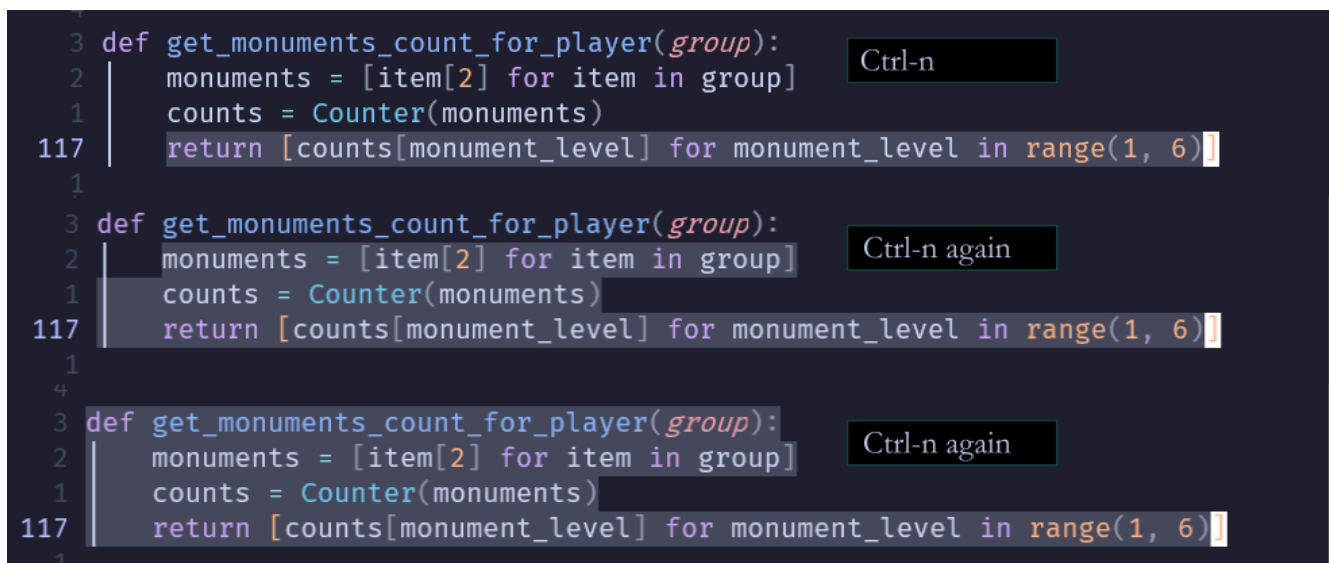
LSP Autocomplete

## Treesitter

We configure [Treesitter](#) next. Treesitter is a fast, incremental code parser. It can parse the code as we type and give the results to NeoVim. NeoVim mainly uses it for syntax highlighting, though there are a few other use cases as well. The syntax highlighting is superior to the usual regex based highlighting. Treesitter can differentiate between a local and global variable for example, even though both have the same regex.

I have treesitter configured for a bunch of languages.

In addition, I have enabled incremental selection on Treesitter. Pressing `Ctrl-n` will select the innermost syntactical piece of code based on the cursor location. Press `Ctrl-n` again and it expands the selection to the next scope in the parse tree. Take a look



```
3 def get_monuments_count_for_player(group):
2 |     monuments = [item[2] for item in group]
1 |     counts = Counter(monuments)
117 |     return [counts[monument_level] for monument_level in range(1, 6)]

3 def get_monuments_count_for_player(group):
2 |     monuments = [item[2] for item in group]
1 |     counts = Counter(monuments)
117 |     return [counts[monument_level] for monument_level in range(1, 6)]

3 def get_monuments_count_for_player(group):
2 |     monuments = [item[2] for item in group]
1 |     counts = Counter(monuments)
117 |     return [counts[monument_level] for monument_level in range(1, 6)]
```

Incremental selection via Treesitter

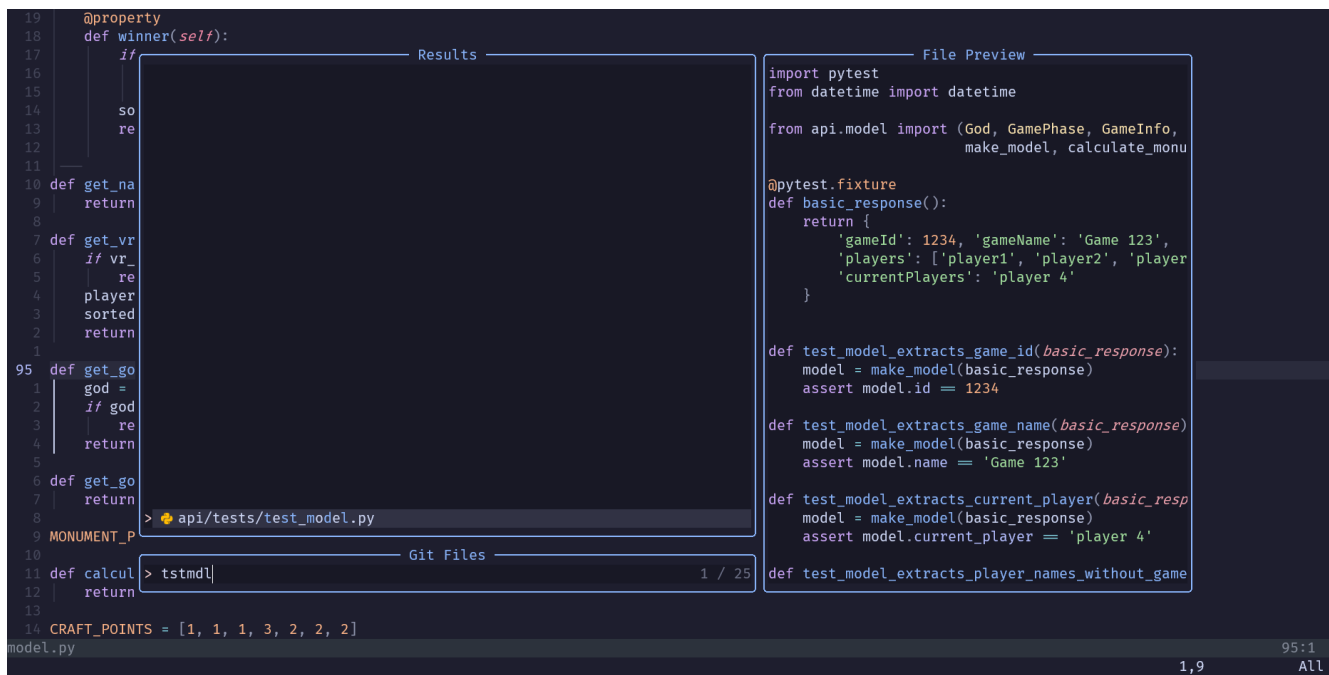
Here the cursor starts on the `return` word. Had the cursor been inside the list comprehension, then it would have selected the list comprehension first, then the whole line and so on. This is such a great feature.

## Telescope

The last of the big configurations is [Telescope](#). Telescope is a fuzzy finder that can search for different things. You can search for files in the current project, switch between open buffers, search text within all the files, symbols in the document – almost anything really. Check the docs for a full list of capabilities.

The two most common that I use are `<leader>sf` to fuzzy search files in the project, and `<leader><space>` to swap between buffers

Here is a screenshot using Telescope to fuzzy find a file in the project



Type a few characters to fuzzy find a file in the project

## Linting & Formatting

I use the [null-ls](#) to lint and reformat the file everytime I save it. I use [ruff](#) as the linter and [Black](#) for formatting Python code.

`null-ls` supports both, as well as many others.

## Terminal

Nothing much to say about this really. `toggleterm.nvim` will open the terminal when I press `Ctrl-s`. Most of the interesting terminal configuration happened earlier in the config. `toggleterm` is nice because it allows you to have multiple terminals open. Show / hide them all or show / hide a single one. Useful when you want to run a server on one terminal, hide it and open another terminal for running commands.

An interesting feature is you can select some code and send it to run on a terminal. So if you have a Python REPL session open in a terminal, you can select code in the buffer and have it run in the REPL.

## Other Editor Plugins

The rest of the file is standard editor configuration plugins. My configuration is

- [lualine](#) for the status line
- [bufferline](#) to display a list of open buffers
- [mini.pairs](#) for auto-closing quotes, brackets, etc
- [mini.surround](#) to surround text with a character, or remove/replace a surrounding
- [indent-blankline](#) to show indentation guides (super useful for Python coding)

One nice thing about `bufferline` is that it allows us to pin buffers, and close all unpinned buffers. Often I find myself working on one or two files, but I temporarily need to open many other files. You can pin the buffers you are working on and just close everything else when you are done.

You can also hook it up to LSP diagnostics, so it will show an icon if the file has any warnings or errors.

## Other Stuff

I added an autocommand to highlight text when it is yanked, so you know what was yanked. This is just copied over from Kickstart.

Kickstart also has a convenience remapping for using navigation keys with lines that are wrapped (I use arrow keys for navigation in NeoVim)

Last, there are some useful LSP related keymaps. These should be self explanatory. Note that `pyright` does not support code actions. But I put a keymap for it anyway 😊

## Summary

Thats a walkthrough of the entire config.

It covers all the basics and should be a good starting point for anyone wanting to setup their NeoVim from scratch for Python coding.

By the end of it all, you should have a fast, lightweight editor that can do everything that the heavier IDEs do.

So whats next from here? Well you might want to add more plugins or swap some of my plugins with other alternatives. Here are some things that I don't have yet

- Plugin to comment / uncomment code. I just don't do it all that often
- Support for Python virtual environments. I just activate the virtual environment on the terminal before opening nvim and that works fine for me
- Debugging support. I don't do much debugging so I left it out. But if you need it, Mason has support for adding debuggers via Debug Adapter Protocol (DAP)
- File Explorer: I don't include a file explorer plugin as I tend to use Telescope's fuzzy find. If you want one, [nvim-tree](#) is quite popular

Apart from that, there are many other plugins so you can customise NeoVim to your preferred way of working. Check out [This Week in NeoVim](#) to see whats the latest and greatest.

Happy Editing 📖

### **Did you like this article?**

If you liked this article, [consider subscribing to this site](#). Subscribing is free.

Why subscribe? Here are three reasons:

1. You will get every new article as an email in your inbox, so you never miss an article
2. You will be able to comment on all the posts, ask questions, etc
3. Once in a while, I will be posting conference talk slides, longer form articles (such as this one), and other content as subscriber-only