Inbuilt features

- NvChad is built upon its personal plugins and many general neovim plugins, below are the features that are provided by nvchad plugins **(our ui plugin, base46, extensions, nvterm, nvim-colorizer)**

Base46

- Base46 is NvChad's highlight performant theming plugin and has many ported themes (around 57+).

How it works?

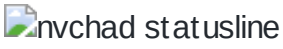
- Gets highlight groups
- Do some computations i.e check for overriden highlight groups, new highlight groups, theme overrides, custom user themes etc.
- Now base46 compiles all of that into bytecode.
- Integration files aren't loaded by default, for example highlight group for telescope, nvimtree etc are put into different files.
- highlight groups are lazyloaded i.e you load them when needed
- **example : dofile(vim.g.base46_cache .. "cmp")**
- In the below video you can see that the chadrc file's (user config) UI related options reload on the fly

Theme switcher

- A theme switcher with telescope.nvim which reloads theme on the fly using base46 plugin + plenary.nvim.

Statusline

- We have our own statusline module (our UI Plugin) which has 4 statusline styles



Tabufline

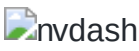
- NvChad's tabufline module (from UI Plugin) is a mix of tabline & bufferline.
- It manages buffers & tabs, buttons in it are clickable
- Each tab will have its own set of buffers stored and the tabufline will show those only.
- Think it like workspaces on Linux/Windows where windows stay in their own workspaces, but in vim buffers from all tabs will be shown in every tab!

Nvterm

- NvChad's terminal plugin to toggle and run commands in neovim terminal buffer
- Using it with our telescope picker (:Telescope terms) to unhide terminal buffers `leader + pt`.

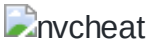
Dashboard

- Nvdash is NvChad's minimal dashboard module, It's very simple at this stage and will get more features in the future!
- Command to run it `Nvdash`, its disabled on startup, check the default_config.lua for its syntax and override it from chadrc.



NvCheatsheet

- Auto-generated mappings cheatsheet module which has a similar layout to that of CSS's masonry layout.
- It has 2 themes (grid & simple)



- command to toggle it : `NvCheatsheet` and mapping `leader + ch`

General neovim plugins

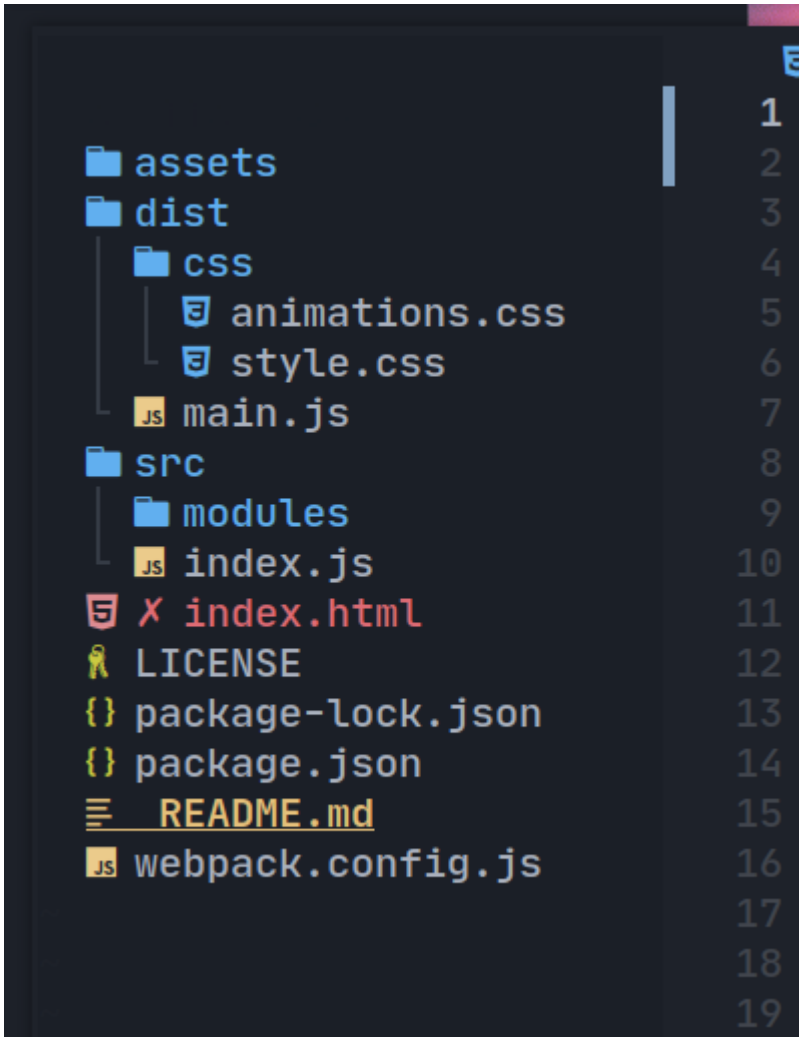
- These plugins aren't related to nvchad, we just tweak theme a bit and theme the UI related ones.

Telescope.nvim

- Telescope.nvim is a highly extendable fuzzy finder over lists. Built on the latest awesome features from neovim core. Telescope is centered around modularity, allowing for easy customization.
- Below are 2 styles of telescope in nvchad (bordered and borderless)

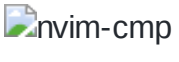
Nvim-tree.lua

- `nvim-tree.lua` is a file explorer tree for Neovim written in Lua.



Nvim-cmp

- [`nvim-cmp`](A completion plugin for neovim coded in Lua.) is a completion plugin for neovim coded in Lua.
- Below are some cmp styles in nvchad



- Note that thats just the cmp look in everblush theme, there are more 57 themes! You can hide cmp icons, cmpkind txt etc from the user config (chadrc) itself!

Auto-completion & LSP

- `nvim-lspconfig` is used along with cmp for completion and `luasnip` + `friendly-snippets` for snippet completion!

- `lazy.nvim` - A modern plugin manager for Neovim
- `whichkey.nvim` - Create key bindings that stick. WhichKey is a lua plugin for Neovim 0.5 that displays a popup with possible keybindings of the command you started typing.
- `nvim-colorizer.lua` - Fastest Neovim colorizer, hex colors, hsl codes and much more.
- `nvim-treesitter` - Nvim Treesitter configurations and abstraction layer, we use it for syntax highlighting & auto-indenting.
- `blankline` - Indent guides for Neovim i.e indentline plugin.
- `gitsigns.nvim` - Git integration for buffers
- `nvim-autopairs`
- `comment.nvim` - Commenting plugin
- `mason.nvim` - Portable package manager for Neovim that runs everywhere Neovim runs. Easily install and manage LSP servers, DAP servers, linters, and formatters.

import OS_Selector from "~/components/docpage/install.jsx";

Pre-requisites

- Neovim 0.9.0.
- Nerd Font as your terminal font.
  - Make sure the nerd font you set doesnt end with **Mono** to prevent small icons.
  - **Example :** JetbrainsMono Nerd Font and not ~~**JetbrainsMono Nerd Font Mono**~~
- Ripgrep is required for grep searching with Telescope **(OPTIONAL)**.
- GCC, Windows users must have `mingw` installed and set on path.
- Delete old neovim folder (check commands below)

Install

Update

To update NvChad run the following command :

- `NvChadUpdate`.

Uninstall

```
# Linux / Macos (unix)
rm -rf ~/.config/nvim
rm -rf ~/.local/share/nvim

# Windows
rd -r ~\AppData\Local\nvim
rd -r ~\AppData\Local\nvim-data
```

Luasnip

NvChad uses `luasnip` plugin for handling snippets, by default it uses `friendly-snippets` plugin which provides snippets for many languages .

- But you would want to extend or add your own snippets so read luasnip docs.

Globals

These are the globals you can use to include the path of your snippets. Put them in **custom/init.lua**.

```
-- vscode format i.e json files
vim.g.vscode_snippets_path = "your snippets path"

-- snipmate format
vim.g.snipmate_snippets_path = "your snippets path"

-- lua format
vim.g.lua_snippets_path = vim.fn.stdpath "config" .. "/lua/custom/lua_snippets'
```

> The above code is an example in which we first get the path of nvim config and then add

our custom snippets dir

Setup lsp server

Before starting, it is strongly recommended that you walk through the LSP configuration: `lspconfig` repository.

Then check server_configurations.md to make sure your language's LSP server is present there.

- **custom/plugins.lua**

```
-- In order to modify the `lspconfig` configuration:
{
  "neovim/nvim-lspconfig",
   config = function()
      require "plugins.configs.lspconfig"
      require "custom.configs.lspconfig"
   end,
},
```

- **custom/configs/lspconfig.lua**

```
local on_attach = require("plugins.configs.lspconfig").on_attach
local capabilities = require("plugins.configs.lspconfig").capabilities

local lspconfig = require "lspconfig"
local servers = { "html", "cssls", "clangd"}

for _, lsp in ipairs(servers) do
  lspconfig[lsp].setup {
    on_attach = on_attach,
    capabilities = capabilities,
  }
end

-- Without the loop, you would have to manually set up each LSP
--
-- lspconfig.html.setup {
--   on_attach = on_attach,
--   capabilities = capabilities,
-- }
--
-- lspconfig.cssls.setup {
```

```
--   on_attach = on_attach,
--   capabilities = capabilities,
-- }
```

Mason.nvim

The `mason.nvim` plugin is used to install LSP servers, formatters, linters, and debug adapters. It's better to list all your required packages in your Mason override config, so they automatically install when running `MasonInstallAll` command.

You can find the exact name of the LSP packages using `:Mason`, that will open a window.

```
{
  "williamboman/mason.nvim",
  opts = {
     ensure_installed = {
       "lua-language-server",
       "html-lsp",
       "prettier",
       "stylua"
     },
   },
 }
```

> Once the binaries are installed, you will have to configure them to properly work with LSP, null-ls, nvim-dap etc. It depends on what you installed. **NvChad does not provide any language configuration aside from lua**.

Statusline & tabufline

We use our own plugin for `statusline` and `tabufline`. The default config is (keep in mind that every plugin's default config is just a table):

```
M.ui = {
  -- ...other options

  statusline = {
    theme = "default", -- default/vscode/vscode_colored/minimal

    -- default/round/block/arrow (separators work only for "default" statusline
    -- round and block will work for the minimal theme only)
    separator_style = "default",
```

```
    overriden_modules = nil,
  },

  tabufline = {
    lazyload = true,
    overriden_modules = nil,
  },

  -- ...other options
}
```

Override `statusline` modules

It is also possible to override the plugin's configuration:

```
M.ui = {
  statusline = {
    overriden_modules = function()
      local st_modules = require "nvchad_ui.statusline.default"
      -- this is just default table of statusline modules

      return {
        mode = function()
          return st_modules.mode() .. " bruh "
          -- or just return "" to hide this module
        end,
      }
    end,
  },
}
```

It is recommended to check the list of modules in our `statusline` modules file. In the above code, you can see that we want to print "bruh" next to the mode module, in the statusline. In order to add highlight group to your text, do:

```
"%#BruhHl#" .. " bruh " -- the highlight group here is BruhHl
```

Override `tabufline` modules

The configuration for overriding `tabufline` is the same as in `statusline`:

```
M.ui = {
  tabufline = {
```

```
      overriden_modules = function()
        local modules = require "nvchad_ui.tabufline.modules"

        return {
          buttons = function()
            return modules.buttons() .. " my button "
          end,
          -- or buttons = "" , this will hide the buttons
        }
      end,
  },
}
```

Again, check the list of modules in our tabufline modules file.

Overview

NvChad uses lazy.nvim for plugins management. Basically, NvChad expects a user plugin table, which then gets merged with the default plugins table. You can find the default table in: `lua/plugins/init.lua`.

Lazy loading

We lazy load almost 95% of the plugins, so we expect and recommend you to lazy load the plugins as well, as its efficient in reducing startuptime.

- We don't want users making NvChad slow just because they didn't lazy load plugins they've added.
- Please read the lazy.nvim plugin specs docs to know what options are available for lazyloading etc.
- Try your best to lazy-load a plugin!

Manage custom plugins

All NvChad default plugins will have `lazy = true` set. Therefore, if you want a plugin to be enabled on startup, change it to `lazy = false`.

It is recommended that you avoid saving any files in the `custom/plugins/*` directory.

> Our system utilizes the import feature provided by `lazy.nvim`, which imports all files in a directory and expects each file to return plugin tables. This behavior is undesirable for our purposes, so it is recommendeed to create a single file

> named **custom/plugins.lua**. This file will be imported by `lazy.nvim`, and no other files in the directory will be processed.

- **custom/chadrc.lua** ```lua M.plugins = "custom.plugins" ```
- **custom/plugins.lua** ```lua local plugins = {
{ "elkowar/yuck.vim" , lazy = false }, -- load a plugin at startup

-- You can use any plugin specification from lazy.nvim { "Pocco81/TrueZen.nvim", cmd = { "TZAtaraxis", "TZMinimalist" }, config = function() require "custom.configs.truezen" -- just an example path end, },

-- this opts will extend the default opts { "nvim-treesitter/nvim-treesitter", opts = { ensure_installed = {"html", "css", "bash"}, }, },

-- if you load some function or module within your opt, wrap it with a function { "nvim-telescope/telescope.nvim", opts = { defaults = { mappings = { i = { [""] = function(...) require("telescope.actions").close(...) end, }, }, }, }, },

{ "folke/which-key.nvim", enabled = false, }

}

return plugins

```
# Nvim-treesitter

We use [Nvim-treesitter](https://github.com/nvim-treesitter/nvim-treesitter) pl

## Install parsers

The TSInstall command is used to install treesitter parsers i.e `TSInstall <par

- Example :

```lua
TSInstall lua html
```

But this may be tedious when you have so many parsers to install and you'd have to repeat this step if you're re-installing nvchad with your old custom settings.

Custom config

- So now we'll just override the default config and add our own parser names to it.
```

- For knowing correct parser names, do check nvim-treesitter docs
- `custom/plugins.lua`

```
{
    "nvim-treesitter/nvim-treesitter",
    opts = {
      ensure_installed = {
        -- defaults
        "vim",
        "lua",

        -- web dev
        "html",
        "css",
        "javascript",
        "typescript",
        "tsx",
        "json",
        -- "vue", "svelte",

       -- low level
        "c",
        "zig"
      },
    },
  },
```

Override default highlight groups

- Make sure you use a valid highlight group.
- Check your theme colors in the base46 theme dir
- To know which highlight groups are available, check the base46 integrations dir
- Also if you just press tab key in hl_override, a list of highlight groups will show up via the completion menu.

When modifying the custom highlight groups in your theme file, such as "onedark.lua", it is important to note that only the variables from "base_30" can be used for this purpose.

Although hex colors can also be used in the "fg/bg" field, it is recommended to utilize the variable names (e.g. "blue", "darker_black", "one_bg", etc.) from your theme file as they will provide a better aesthetic. This way, there is no need to manually write the hex colors.

```
M.ui = {
   hl_override = {
      Pmenu = { bg = "white" },
      -- Pmenu = { bg = "#ffffff" }, this works too

      MyHighlightGroup = { -- custom highlights are also allowed
         fg = "red",
         bg = "darker_black"
      }
   },
}
```

In order to add custom highlights, its the same as above, just use `hl_add`.

Customize themes

If you just want to customize an already existing theme, you can change the following configuration:

```
M.ui = {
   changed_themes = {
      onedark = {
         base_16 = {
            base00 = "#mycol",
         },
         base_30 = {
            red = "#mycol",
            white = "#mycol",
         },
      },

      nord = {
         -- and so on!
      },
   },
}
```

Local themes

> WARNING: Do this at your own risk because you might not be able to make nice nvchad themes like siduck.

- Default themes can be found in our `base46` repository.

Here is the default structure for NvChad themes:

```
-- place the file in /custom/themes/<theme-name>.lua
-- for example: custom/themes/siduck.lua

local M = {}

M.base_30 = {
   -- 30 colors based on base_16
}

M.base_16 = {
   -- base16 colors
}

M.type = "dark" -- light / dark

return M
```

Finally, add your theme in chadrc.

```
M.ui = {
   theme = "siduck",
}
```

How does NvChad work?

Understanding the basics

Before getting into the this topic, first you should understand the `vim.tbl_deep_extend` function which is used for merging tables and their values recursively.

- The function `vim.tbl_deep_extend` is normally used to merge 2 tables, and its syntax looks like this:

```
-- table 1
local person = {
    name = "joe",
    age = 19,
}
```

```
-- table 2
local someone = {
    name = "siduck",
}

-- "force" will overwrite equal values from the someone table over the person t
local result = vim.tbl_deep_extend("force", person, someone)

-- result :
{
    name = "siduck", -- as you can see, name has been overwritten
    age = 19,
}
```

Its usage can even be used in more complex tables. As said, it works recursively, which means that it will apply the same behaviour for nested table values:

```
local person = {
    name = "joe",
    age = 19,
    skills = {"python", "java", "c++"}

    distros_used = {
        ubuntu = "5 years",
        arch = "10 minutes",
        manjaro = "10 years",
    }
}

local someone = {
    name = "siduck",
    skills = {"js", "lua"},

    distros_used = {
       ubuntu = "1 month",
       artix = "2 years"
    }
}

local result = vim.tbl_deep_extend("force", person, someone)
```

The resulting table will have merged each property from the tables, and the same for the `skills` and `distros_used` values:

```
{
   name = "siduck",
   age = 19

   skills = {"js", "lua"},

   distros_used = {
       ubuntu = "1 month",
       arch = "10 minutes",
       manjaro = "10 years",
       artix = "2 years"
   }
}

-- tbl_deep_extend function merges values recursively, but if there's an array

-- Example: the first table has {"python", "java", "c++"} and the second table

-- Now you might be wondering that it should merge it like this: { "python", "j

-- But no! thats wrong, the result will be only {"js","lua"}
```

To sum up, `tbl_deep_extend` merges dictonary tables recursively (i.e tables which have `key/value` pair but not lists).

Config Structure

```
├── init.lua ( main init.lua )
│
├── lua
│   │
│   ├── core
│   │   ├── default_config.lua
│   │   ├── mappings.lua
│   │   ├── utils.lua
│   │   └── init.lua
│   │
│   ├── plugins
│   │    ├── init.lua
│   │    └── configs
│   │        ├── cmp.lua
│   │        └── other configs
│   │
│   │   USER CONFIG
│   │
```

```
|   ├── custom *
|   |   ├── chadrc.lua
|   |   ├── init.lua
|   |   ├── more files, dirs
```

- **`init.lua`** - runs whole config
- **`core/default_config`** - returns a table of default options in NvChad.
- **`core/mappings`** - default mappings
- **`core/init`** - default globals, nvim options, commands, autocmds
- **`core/utils`** - helpful functions

Custom config

There are 2 important files in **custom** dir which extend NvChad:

- **`custom/chadrc.lua`** meant to override that table in `default_config.lua` file
- **`custom/init.lua`** runs in the main `init.lua`, its meant to have vim options, globals, autocmds, commands etc.

From now on, whenever we talk about paths, keep in mind that they're relative to the `lua` folder in your `nvim` config (by default it should be `~/.config/nvim/`).

- It is not recommended to make changes outside the `custom` dir, because NvChad config is a repo and it **gitignores** only the custom dir, it uses `git pull` to update the config.
- Any other file outside the `custom` dir will be treated as a change by `git`, meaning that NvChad will not be able to fast-forward the pull.

Themes

You can see all the themes with the following keymap: `<leader> + th`.

> The `leader` key is the `space` in NvChad.

Mappings

If you want to know all the keymaps, you can run the following comands:

- `NvCheatsheet`
- `Telescope keymaps`

Null-ls.nvim

It is recommended that you install `null-ls` to manage formatting & linting. Here's a possible install configuration for `null-ls`:

```
{
  "neovim/nvim-lspconfig",

   dependencies = {
     "jose-elias-alvarez/null-ls.nvim",
     config = function()
       require "custom.configs.null-ls"
     end,
   },

   config = function()
      require "plugins.configs.lspconfig"
      require "custom.configs.lspconfig"
   end,
}
```

- Dependencies are loaded after the original plugin (`lspconfig` in NvChad's case). - `null-ls` is loaded after `lspconfig` as `lspconfig` is lazy-loaded.

Configuration

Make sure to check `null-ls` builtins to get exact names for formatters, linters etc.

Here's an example configuration for `null-ls`, following the NvChad file directory structure:

```
-- custom/configs/null-ls.lua

local null_ls = require "null-ls"

local formatting = null_ls.builtins.formatting
local lint = null_ls.builtins.diagnostics

local sources = {
   formatting.prettier,
   formatting.stylua,
```

```
   lint.shellcheck,
}

null_ls.setup {
   debug = true,
   sources = sources,
}
```

- Check null-ls docs for adding format on save. Other things to take into account when configuring `null-ls` for NvChad:
- This shortcut is defined for code formatting: `<leader> + fm`.
- The linter, formatter or debugger that you will use in `null-ls` configuration, has to be downloaded via `mason` (that ensure_installed opt) or system wide.
- Make sure the LSP servers for the filetypes are active for the relevant `null-ls` formatter and/or linter to work.

Overview

The mapping configuration uses the nvim name shorcuts as:

- `<C>` -> `Ctrl`
- `<leader>` -> `Space`
- `<A>` -> `alt`
- `<S>` -> `shift`
- The default mappings are defined in `core.mappings` (`core/mappings.lua).
- Alternatively, you can use `NvCheatsheet` or `Telescope keymaps` to list all mappings.

Mapping format

In order to list custom shortcuts in NvCheatsheet, make sure to use the following format

```
-- opts is an optional parameter
["keys"] = {"action", "description", opts = {}},

["<C-n>"] = {"<cmd> NvimTreeToggle <CR>", "Toggle nvimtree"},
["<leader>ff"] = {"<cmd> Telescope <CR>", "Telescope"},
```

```
-- opts can have the props: buffer, silent, noremap, nowait and so on.
-- All standard key binding opts are supported.
[";"] = { ":", "enter cmdline", opts = { nowait = true } },

-- For a more complex keymap
["<leader>tt"] = {
  function()
     require("base46").toggle_transparency()
  end,
  "toggle transparency",
},
```

Add new mappings

- In order to add or customize the mappings, make sure that you follow the expected file structure for NvChad.
- The default mappings are loaded from `core.mappings`, and it is recommended that you place your mappings inside `custom.mappings` file.
- Remember that the mappings **must** have a vim mode: `n` (for normal), `v` (for visual), `i` (for insert) and so on.
- **custom/chadrc.lua**

```
M.mappings = require "custom.mappings"
```

- **custom/mappings.lua**

```
local M = {}

-- In order to disable a default keymap, use
M.disabled = {
  n = {
      ["<leader>h"] = "",
      ["<C-a>"] = ""
  }
}

-- Your custom mappings
M.abc = {
  n = {
     ["<C-n>"] = {"<cmd> Telescope <CR>", "Telescope"},
     ["<C-s>"] = {":Telescope Files <CR>", "Telescope Files"}
```

```
  }

  i = {
     ["jk"] = { "<ESC>", "escape insert mode" , opts = { nowait = true }},
    -- ...
  }
}

return M
```

> Mappings will be automatically loaded. You don't need to load them manually!

Manually load mappings

Even though it is not required, you can manually load your mappings

```
M.some_plugin_name = {
  plugin = true, -- Important

  n = {
     ["<C-n>"] = {"<cmd> Telescope <CR>", "Telescope"}
  }
}

-- Now to load it
require("core.utils").load_mappings("someplugin")
```

Comments

```
-- comment
print("Hi") -- comment

--[[
 multi-line
 comment
]]
```

Variables

```
-- Different types
```

```
local x = 10 -- number
local name = "sid" -- string
local isAlive = true -- boolean
local a = nil --no value or invalid value

-- increment in numbers
local n = 1
n = n + 1
print(n) -- 2

-- strings
-- Concatenate strings
local phrase = "I am"
local name = "Sid"

print(phrase .. " " .. name) -- I am Sid
print("I am " .. "Sid")
```

Comparison Operators

```
 == equality
 < less than
 > greater than
 <= less than or equal to
 >= greater than or equal to
 ~= inequality
```

Conditional Statements

```
-- Number comparisons
local age = 10

if age > 18 then
  print("over 18") -- this will not be executed
end

-- elseif and else
age = 20

if age > 18 then
  print("over 18")
elseif age == 18 then
  print("18 huh")
else
```

```
  print("kiddo")
end

-- Boolean comparison
local isAlive = true

if isAlive then
    print("Be grateful!")
end

-- String comparisons
local name = "sid"

if name ~= "sid" then
  print("not sid")
end
```

Combining Statements

```
local age = 22

if age == 10 and x > 0 then -- both should be true
  print("kiddo!")
elseif x == 18 or x > 18 then -- 1 or more are true
  print("over 18")
end

-- result: over 18
```

Invert Value

You can also invert a value with the **not** keyword:

```
local isAlive = true

if not isAlive then
  print(" ye ded!")
end
```

Functions

```
local function print_num(a)
  print(a)
```

```
end

or

local print_num = function(a)
  print(a)
end

print_num(5) -- prints 5

-- multiple parameters
function sum(a, b)
  return a + b
end
```

Scope

Variables have different scopes. Once the end of the scope is reached, the values in that scope are no longer accessible.

```
function foo()
  local n = 10
end

print(n) -- nil , n isn't accessible outside foo()
```

Loops

Different ways to make a loop:

While

```
local i = 1

while i <= 3 do
   print("hi")
   i = i + 1
end
```

For

```
for i = 1, 3 do
   print("hi")
end
-- Both print "hi" 3 times
```

Tables

- Tables can be used to store complex data.
- Types of tables: arrays (lists) and dicts (key,value)

Arrays

- Items within these can be accessed by "index".

```
local colors = { "red", "green", "blue" }

print(colors[1]) -- red

-- Different ways to loop through lists
-- #colors is the length of the table, #tablename is the syntax

for i = 1, #colors do
  print(colors[i])
end

-- ipairs
for index, value in ipairs(colors) do
   print(colors[index])
   -- or
   print(value)
end

-- If you dont use index or value here then you can replace it with _
for _, value in ipairs(colors) do
   print(value)
end
```

Dictionaries

- These contain keys and values:

```
local info = {
   name = "sid",
```

```
   age = 20,
   isAlive = true
}

-- both print sid
print(info["name"])
print(info.name)

-- Loop by pairs
for key, value in pairs(info) do
   print(key .. " " .. tostring(value))
end

-- prints name sid, age 20 etc
```

Nested Tables

```
-- Nested lists
local data = {
    { "sid", 20 },
    { "tim", 90 },
}

for i = 1, #data do
  print(data[i][1] .. " is " .. data[i][2] .. " years old")
end

-- Nested dictionaries
local data = {
    sid = { age = 20 },
    tim = { age = 90 },
}
```

Modules

Import code from other files

```
require("path")

-- for example in ~/.config/nvim/lua , all dirs and files are accessable via re
-- Do know that all files in that lua folder are in path!
-- ~/.config/nvim/lua/custom
-- ~/.config/nvim/lua/custom/init.lua

 require "custom"
```

```
-- both do the same thing
```