

# Assignment 2: MiniVim

---

Author: Zhou Jiahao & Yang Zhenyu

Due: December 13th, **no delay, no delay, no delay.**

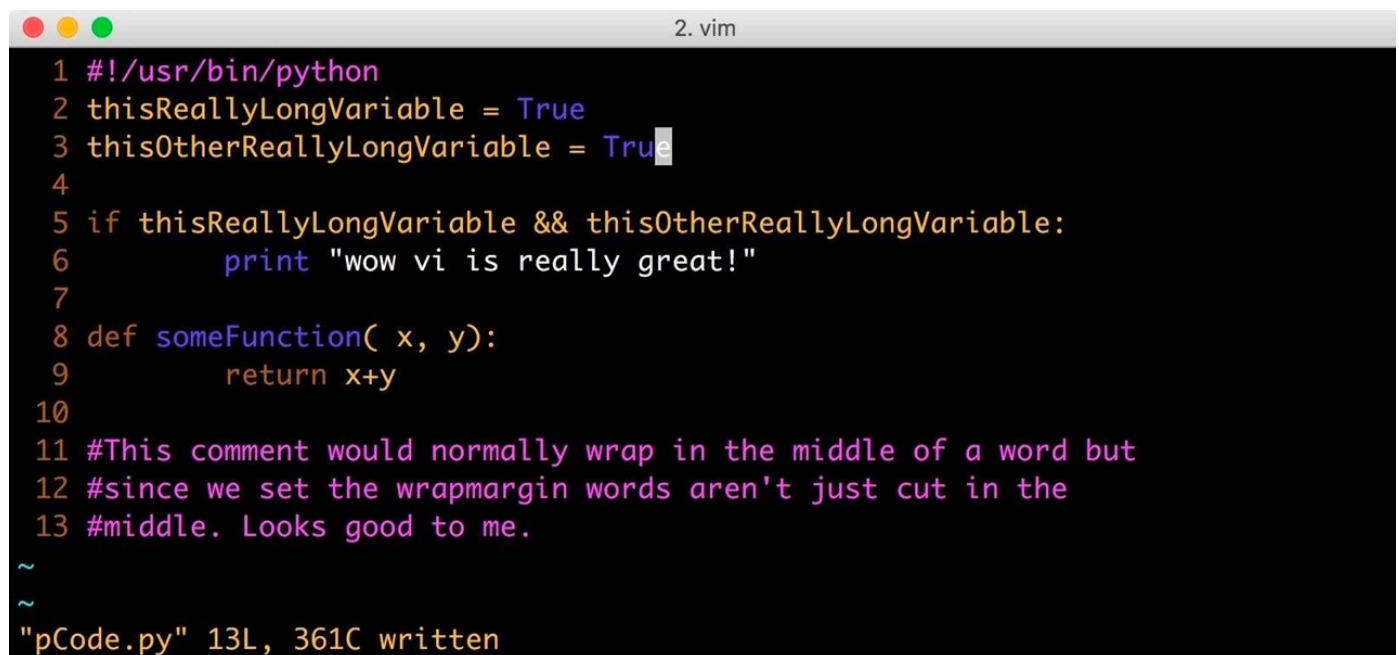
Please do not hesitate to contact us if you have any questions about this documents

You can get a Chinese version of this document by throw it into [deepl.com](https://deepl.com)

## Introduction

---

Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as "vi" with most UNIX systems and with Apple OS X.

A screenshot of a Vim editor window. The title bar at the top shows three colored window control buttons (red, yellow, green) on the left and the text "2. vim" on the right. The main editing area has a black background with syntax-highlighted Python code. The code includes variable assignments, an if-statement with a print statement, and a function definition. Line numbers 1 through 13 are visible on the left. At the bottom, a status line shows "~", "~", and "pCode.py" 13L, 361C written. The cursor is positioned at the end of line 3.

```
1 #!/usr/bin/python
2 thisReallyLongVariable = True
3 thisOtherReallyLongVariable = True
4
5 if thisReallyLongVariable && thisOtherReallyLongVariable:
6     print "wow vi is really great!"
7
8 def someFunction( x, y):
9     return x+y
10
11 #This comment would normally wrap in the middle of a word but
12 #since we set the wrapmargin words aren't just cut in the
13 #middle. Looks good to me.
~
~
"pCode.py" 13L, 361C written
```

By simply typing `vim <file_path>`, you can then open a file with vim. You can view text file in `Normal Mode`, insert characters in `Insert Mode` and input command in `Command Mode`. We will explain those modes in assignment details.

You can take a brief lesson on vim on this [website](#). This may take you twenty minutes or so, but covers some of the basic operations of vim, and you can also run vim in your own environment to actually try it out.

Our assignment requirements are not as complex, but will allow you to implement a text editor that has basic functionality (with some specified extensions) and can actually be used, **starting from scratch**.

That means we won't give you any existing files, you need to complete the whole job starting from your own new directory. The **guidelines** needed to complete the project are at the back of this document, and you can always find a TA to answer questions.

## Assignment

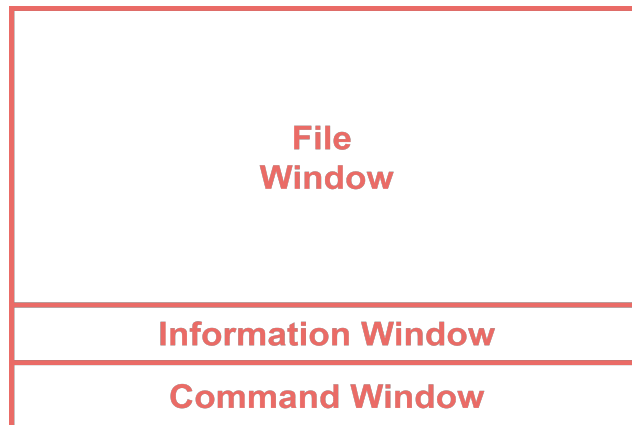
---

# Basic

## 1. Base Terminal User Interface.

Your TUI should look like this.

- File Window displays the opened file, and you can edit file in this window.
- Information Window displays some file-related information. Here's the minimum information you should display: `Mode`, `Filename`, `Cursor line and column`.
- Command Window displays the commands user input. Note that you should print a `:` at the beginning of command window when enter command mode.



## 2. Multiple modes.

- Normal mode:

In normal mode, we can move the cursor and browse the opened file by  $\uparrow\downarrow\leftarrow\rightarrow$ . Note that:

- If the number of file lines exceeds the maximum number of lines in file window, you should display only part of the file and scroll the file when the cursor reaches the bottom of the window.
- Your cursor should NOT exceed the end of a line and the end of the file.

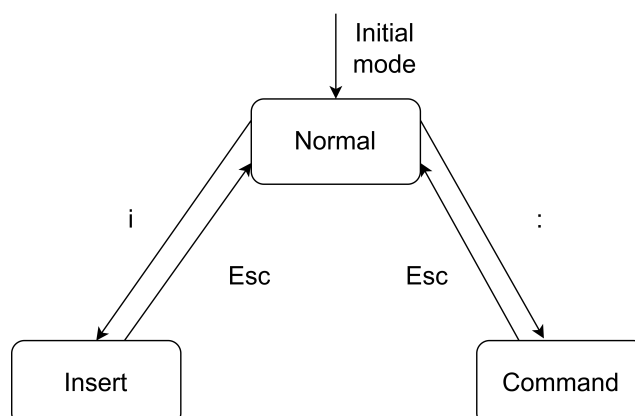
- Insert mode:

In insert mode, we can edit file by moving cursor. Edit file include append, modify and deletion (just like vim).

- Command mode:

In command mode, we can execute some command.

Mode swithing graph is shown below:



### 3. Command line arguments.

User can open your minivim in terminal by command

```
minivim [options] <filename>
```

This command will open a file specified by `<filename>` in your minivim. If the file doesn't exist, create it.

You should support several optional arguments in command line:

- `-t`: open file in truncation mode. You should truncate the file from the beginning.
- `-R`: open a file in read-only mode.

To start your MiniVim by `minivim` rather than `PATH/TO/MiniVim`, you may need to add your executable file to `/bin` or export your executable file to `$PATH`. See more details in [Export to PATH](#).

### 4. Commands.

You should support several base commands:

- `:w`: Save the file.
- `:q`: Quit. Warn user if the file is changed and is unsaved.
- `:q!`: Force quit (i.e. Do not save the file and force quit.) If the file does not exist and was created by MiniVim, then this operation will also undo the creation.
- `:wq`: Save then quit.

## Extension

- Word Completion. MiniVim gives some options for auto-completion when user has entered the prefix. You may need an extra window to display the alternative words and let user to select words by numbers or  $\leftarrow \rightarrow$  (just like your `shu ru fa`. I really do not know what it is in English TAT ). You can get the english word library named `words_alpha.txt` in canvas.
- Search and Substitution. Minivim supports searching for a word in the full file and substitute it to another word. User may use command `:sub "stone" "gold"` to substitute stone with gold.
- Line Number and Jump. MiniVim supports displaying line number at the beginning of a line and jumping to a specific line by command `:jmp LineNumber`. You may display the specific line on the top.
- Shortcut Key. MiniVim supports several shortcut key in **Normal Mode**:
  - `dd`: delete the entire line that the cursor is currently on.
  - `o`: Move the cursor to the beginning of the line.
  - `$`: Move the cursor to the end of the line.
  - `w`: Move forward one word (delimited by a white space or a new line).
  - `b`: Move backward one word (delimited by a white space or a new line).
- Command History. MiniVim supports browse command previously entered in **Command Mode** by  $\uparrow \downarrow$ .

## Environment

---

# WSL2 Install

Warning: everything in this section is tailored for Windows. If you are an exception, please ask for help.

## Install *Windows Subsystem for Linux*

We are sorry to announce you that due to some technical limitations, this project is to be done in *Linux*. You may have heard of Linux: It's an operating system just like Windows and MacOS. Of course it would be impractical to require all of you guys to install a new system directly on your desktop, rather we recommend doing this through WSL. I'd like to say beforehand that all the steps below are simply suggestions, and you are free to achieve the same goal using anything you prefer.

WSL stands for *Windows Subsystem for Linux*. For now just view it as a virtual machine run in Windows, though in fact there are some subtle (and important) differences between them. To install WSL (or most of other tools you need in the future), the best way is to go directly to the official website and read the [guide](#). In short, the easy way:

1. right-click the Windows icon
2. click Powershell(root) or something similar
3. click yes if the system asks
4. type `wsl --install` in the opened window and hit enter
5. wait.

there is a hard, but more customizable way to install, but let's omit it for now. The tutorial is available on the website above. Now WSL2 and Ubuntu (One of the various Linux-es) should have been successfully installed.

To boot WSL, locate it in the start menu, or type `wsl` and hit enter in Powershell. You may have to wait for a few minutes the first time you launch it. Then, do as the system tells you to create a account and login.

An aside: you can install *Windows terminal* to gain an more pleasing experience when using Powershell and Ubuntu.

If everything goes smoothly, you are now in a working Linux system. It's very likely you have no idea how to use it; however, to this end not much is needed. If you're interested, find a good book and read a few chapters. Here I'll just explain something basic.

## Install Build Tools (like compiler)

In Linux, you interact with system mainly by using commands in a shell. Firstly, as you have done in Windows via CLion, a compiler for C++ is essential. Type `sudo apt update` and hit enter. You are required to enter your password (created by yourself a few minutes ago). Then a lot of texts rolled down the screen: that's what it should be like. Then type `sudo apt install build-essential` to install the most basic components for developing programs. During the process, enter "y" every time the system asks.

A little bit of explanation: `sudo` means you run the following command as a root user: you have the permission to do this. This is why you are asked to provide the password. `apt` is a software, or program, designed for managing your software. The `update` you feed it means you want it to update its *sources* (see later). The `install` you feed it means you want it to install the following software package `build-essential` for you. build-essential contains a bunch of software packages including gcc, g++, make and so on.

```
# get build-essential
sudo apt update
sudo apt install build-essential
```

## Change APT Source (if needed)

By the way, as we all know, in China and some other places the internet is sometimes strange. From my own experience `apt` worked quite well by default, but there do exist circumstances it just can't download something. There are many ways to solve this: you can use a proxy, but to be honest, I managed to do that in my home but failed here in SJTU. Another approach is to change the *sources*.

Briefly, there is a list of URLs as "sources" for `apt` to fetch something like a list. The list contains the names of software available, their descriptions, where to download it, what other software it depends on and many other things. We can replace the default "sources" with new ones which, made a little change: the download URLs it provides work well in China. Then how to do this? This needs a little bit more work, and a good guidance can be found [here](#).

## Use Clion to Code in WSL

But, how to write code? No no we don't use `vim` because there is some way to access WSL using CLion, and it provides more than merely manipulating files. To this end install CMake (see below) in advance. As always, read [this](#) is a good idea. In short,

1. create a new project. The project should be located in the file system of Ubuntu. A good place is somewhere in `/home/your-user-name/`.
2. in `CMakeList.txt`, change `cmake_minimum_required` to 3.16
3. go to Settings / Build, Execution, Deployment / Toolchains
4. click "+" and choose WSL
5. Everything is automated. wait then click OK
6. go to Settings / Build, Execution, Deployment / CMake
7. change toolchain to "WSL" (the one you just created)
8. click OK. if everything goes fine the debug window below shouldn't report any errors.

IDEs used for this assignment are not limited, you can still use whatever you're already familiar with.

**Vscode** is a good alternative for its comprehensive support for WSL.

## Ncurses Install

Use this command to install ncurses in WSL2.

```
sudo apt install libncurses5-dev libncursesw5-dev
```

## CMake Install

Use this command to install & verify cmake in WSL2.

```
sudo apt install cmake
cmake --version
```

# Guideline

## Keyboard

MiniVim is a keyboard-only editor, so all operations will be done with keys. To achieve this, we need to know how to represent keys from the keyboard first.

As we all know, it's easy to get character 'a' in the alphabet by typing 'a'. However, there are many chars on the keyboard (such as `Esc`, `Backspace`) that are not in the alphabet. What is used to represent them in the computer?

[ASCII](#) stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as `a` or `Esc` or an action of some sort. For example, number `97` represents char `a` and `27` represents `Esc`. You can find the corresponding characters in the ASCII table.

## Ncurses

Ncurses (new curses) is a programming library providing an application programming interface (API) that allows the programmer to write text-based user interfaces (TUI) in a terminal-independent manner. It is a toolkit for developing "GUI-like" application software that runs under a terminal emulator.

To use ncurses library, you should include header file by `#include <ncurses.h>` and link ncurses library by `g++ -o xxx xxx.cpp -lncurses` when compiling.

To use ncurses library in CMake project, you can refer to example in CMake subsection of this document.

## Window

Window is a square area in your terminal. In this project, the whole terminal is simply divided into 3 windows.

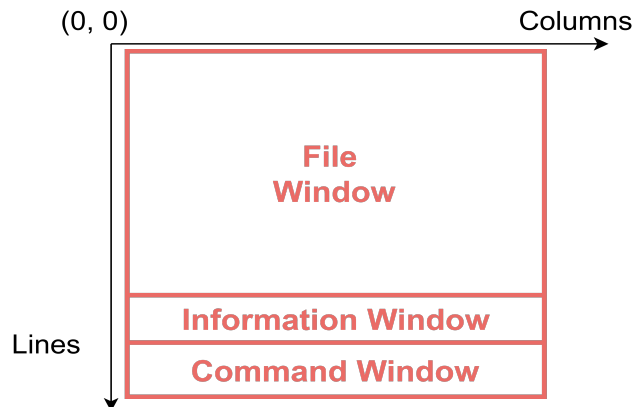


File window displays opened file, and user can edit file in this window. Information window displays some file informations such as filename, edit mode and so on. Command window displays the command input by user in command mode.

At the very beginning, you need to call `initscr()` to create the initial window. A customize window can be created by calling

```
WINDOW *win = newwin(lines, cols, startx, starty);
```

By the way, the coordinates of windows is shown below:



And we can get the begin position by `getbegyx(win, y, x)` and max position by `getmaxyx(win, y, x)`.

Next, we want to input and output something in the window. `wprintw` and `getch` can help us. We can print "Hello world in 2022" to the windows we just created by

```
wprintw(win, "Hello world in %d", 2022);  
wrefresh(win);
```

And we can clear the window by `wclear(win)`.

Note that after we have printed some characters, we usually have to refresh the window by calling `wrefresh`.

To distinguish the different windows more clearly, you can call `wbkgd` to set the frontend and backend colors of the window. Here is a code snippet:

```
int color_pair_num = 0;  
start_color();  
// init color pair  
init_pair(COLOR_PAIR(color_pair_num), COLOR_WHITE, COLOR_CYAN);  
// set window color  
wbkgd(win, color_pair_num);
```

In the end, remember to call `endwin()` to destroy all windows before the program exits.

## Cursor

Cursor is an indicator that shows you where you are in editor. We can move the cursor to  $(line, col)$  in the window by calling

```
wmove(win, line, col);
```

If you want to move to (*line*, *col*) and print 'Hello, world!', just call `mvwprintw(win, line, col, "Hello, world!");` to finish these actions in one line.

If you want to insert the characters before cursor, you may need `winsch(win, char);`. There is also a combination of move and insert `mvwinsch(win, y, x, char);`.

If you want to delete the characters under cursor, `wdelch` and `mvwdelch` may help.

To delete and insert line, use `wdeleteln` and `winsertln`.

In fact, there are lots of functions you can use to reduce the workload, you can check the documentation of [ncurses](#) or just google it.

## Some Setup Function

- `raw()`: Normally the terminal driver buffers the characters a user types until a new line or carriage return is encountered. But most programs require that the characters be available as soon as the user types them. The above function is used to disable line buffering.
- `echo()` and `noecho()`: These functions control the echoing of characters typed by the user to the terminal. `noecho()` switches off echoing.
- `keypad(win, bool)`: It enables the reading of function keys like F1, F2, arrow keys etc. Almost every interactive program enables this, as arrow keys are a major part of any User Interface.

## Simple Demo

You can find a **ncurses demo** named `ncurses_demo.cpp` in canvas.

## More Informations

Get more informations in [Ncurses Tutorial](#) and [Ncurses API](#).

## Command Line Arguments

MiniVim is a command-line program that accepts arguments passed from the command line by the user at startup. At its simplest, the `hello.txt` in `vim hello.txt` is a command line argument.

In C/C++, command line arguments are passed into the program via two arguments to the `main` function:

```
int main(int argc, char *argv[]) {  
    ...  
}
```

`argc` indicates the number of arguments; `argv` is an array of strings of length `argc`, which stores all the arguments passed in.

for example, using `./bin/minivim --test -cmdline arguments` to run the minivim program, its `main` function will receive the following arguments:

```
argc = 4  
argv = [ "./bin/minivim", "--test", "-cmdline", "arguments"]
```



## Manually Parsing

Once you understand the basics, you can accept **simple** command-line arguments yourself (like, in this project, using `argv[1]` as filename). But if the accepted parameters start to get complicated, parsing them yourself becomes cumbersome and error-prone.

## GNU `getopt` Function

C/C++ introduces the `getopt` function in the header `<unistd.h>` to help programmers with the relatively tedious task of parsing. This is a convenient option if you don't want to parse manually and have to add some arguments in a simple format.

Here are some tutorials and examples on `getopt`:

- <https://cloud.tencent.com/developer/article/1176216>
- <https://en.wikipedia.org/wiki/Getopt>

## Using Third-party Library

It is more convenient and encouraged to use any other **third-party library** to complete more complicated command-line arguments parsing, but this requires that you have the ability to import external libraries into your project, which you can try if you have more spare time.

Popular Modern C++ Argument Parsers:

- <https://github.com/adishavit/argh> (C++11 required)
- <https://github.com/muellan/clipp> (C++11 required)
- <https://github.com/p-ranav/argparse> (C++17 required)

You can choose the most suitable implementation yourself, as long as the required functionality is completed. :)

## Project Layout

For a code project that is not a single file, you will of course need to consider how to plan the directory layout: A sensible and elegant project layout can greatly enhance your programming experience.

Although the volume of this assignment is still relatively small, we still recommend a sound structure to get you familiar with how to organize a CMake project from scratch.

We take `CMake` as an example build-system, but you can use whatever you like: `Makefile`, `XMake`...

## Example

```
project/
├── .gitignore
├── README.md
├── CMakeLists.txt
├── bin/
│   └── <compiled executable files>
├── build/
│   └── <cmake generated build files>
```

```

├── include/
│   └── *.h
├── lib/
│   └── <precompiled lib files>
├── src/
│   └── *.cpp
...

```

Above is a simple C++ project layout, let's breakdown:

- `.gitignore`: Ignore non-project files like `.idea/`, `bin/`, `build/` and so on.
- `README.md`: Self-introduction of your project.
- `CMakeLists.txt`: Tell CMake how to build your project & compile executable file.
- `bin/`: Save your compiled executable file (e.g., `minivim`).
- `build/`: Hold the compile scripts and half-compile files generated by CMake.
- `include/`: Header files of your own project or imported third-party project.
- `lib/`: Lib files (precompiled code) from imported third-party project.
- `src/`: Your own `*.cpp` source code.

Not all parts of it are necessary, e.g. the `lib` directory is not needed if no third-party libraries are introduced; you can also add directory yourself, like `scripts`, `docs`, `tests` ...

See [this website](#) or google for more introduction.

## Build Above Project Using CMake

For complex projects, it is annoying and unnecessary to execute `g++ ...` commands manually every time you compile. CMake can generate compilation scripts based on the `CMakeLists.txt` and conveniently invoke them.

For simple projects with ncurses libraries, a `CMakeLists.txt` example is as follows:

```

# specify the minimum version of CMake that is supported
cmake_minimum_required(VERSION 3.10)

# include a project name
project(cmake-example)

# set C++ Version & executable output path
set(CMAKE_CXX_STANDARD 17)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# make your compiler aware of header directory
include_directories(${PROJECT_SOURCE_DIR}/include)

# find & include ncurses library
find_package(Curses REQUIRED)
include_directories(${CURSES_INCLUDE_DIR})

# create a variable which includes needed source files

```

```

set(MY_SOURCES
    ${PROJECT_SOURCE_DIR}/src/hello.cpp
    ${PROJECT_SOURCE_DIR}/src/world.cpp
    ${PROJECT_SOURCE_DIR}/src/main.cpp
)

# specify an executable,
# build from the specified source files
add_executable(example-bin ${MY_SOURCES})

# link ncurses library with your executable
target_link_libraries(example-bin ${CURSES_LIBRARY})

```

To configure & compile this project, simply:

```

# configure project in specified cmake directory: build/
cmake -B build
# compile target in build directory
cmake --build build --target example-bin

```

For more example & usage, see [troy50/cmake-examples](https://github.com/troy50/cmake-examples), ranging from basic hello-world project to complicated features, with detailed explanation. [on-demand, Recommended]

## Word Completion

In order to efficiently predict target words based on its prefixes, we need to use trie-tree. The algorithm for trie-tree is not the focus of our assignment, so you can put the implementation of the tire-tree in this link (any other impl is acceptable) into your project:

<https://github.com/kephir4eg/trie>

Since `auto_ptr` used in this repo was deprecated in C++17, you can change it to `unique_ptr` if your Cpp standard is 17 or later.

## Usage

To install this library, simply copy `trie/src/trie.h` to your `{project}/include/`

To create & initialize & fill the `trie_set`:

```

#include <trie.h>
#include <iostream>

using Lexicon = trie::trie_map<char, trie::SetCounter>;

int main() {
    Lexicon lexicon;

    lexicon.insert("where");
    lexicon.insert("have");
}

```

```

lexicon.insert("you");
lexicon.insert("been");

std::cout << lexicon.contains("where") << " ";
std::cout << lexicon.contains("have") << " ";
std::cout << lexicon.contains("you") << " ";
std::cout << lexicon.contains("been") << " ";
std::cout << std::endl;
}

```

To iterate words with specified prefix:

```

#include <trie.h>
#include <iostream>
#include <vector>

using Lexicon = trie::trie_map<char, trie::SetCounter>;
using WordList = std::vector<std::string>;

int main() {
    Lexicon lexicon;
    WordList words = { "what", "wakes", "you", "and", "when", "why" };

    for (const auto &word : words) {
        lexicon.insert(word);
    }

    for (auto it = lexicon.find_prefix("wh"); it != lexicon.end(); ++it)
        std::cout << it.key() << ' ';
    std::cout << std::endl;
}

```

The output is `what when why`, as expected. With this library, you can predict words by its prefix easily.

## Grade

- Basic: total 55
  - Base terminal user interface: 10
  - Multiple modes & Text Edit: 25
  - In-editor Commands: 5
  - Command line arguments: 5
  - File persistence: 10
- Extensions: up to 25
  - Word Completion: 15
  - Search and Substitution: 10
  - Line Number and Jump: 5
  - Shortcut Key: 5

- Command History: 5
- Coding Conventions & Project Layout: 10
- Q & A: 10