University of Victoria

Faculty of Engineering

Spring 2024

SENG 265

Term Portfolio Project (TPP)

LexPh

Software Engineering

Abstract

The following notebook documents the learning process for SENG 265.

Journal

Week 1, January 8th, 2024

- Recently performed a clean install, restructured the filesystem to utilize cloud storage, and created new restore points.
- Redownloaded the VS Code IDE.
- Downloaded the latest gcc compiler.
- Installed Jupyter Notebook.

Week 2, January 15th, 2024

- Explored HTML and CSS for title page styling.
- Reviewed LinkedIn and Github logo guidelines, downloaded logos, and linked profile and repo on the title page.
- Connected to the Uvic reference platform for assignment compilation.
    - Added batch files for sending and retrieving file commands, as well as one to connect to the reference platform (R.P).
        - Connect to R.P.: `./lab.bat`
        - Send file to R.P.: `./send.bat %filename%`
        - Retrieve file from R.P.: `./retrieve.bat %filename%`
- Reviewed and started a quick sheet for bash.

Week 3, January 22nd, 2024

- Updated bash into a table first using HTML but preferred the structure more in markdown. Used ChatGPT to do the restructuring by providing a markdown format reference and the data in an HTML table.
- Created a Vim quicksheet in a similar markdown format.
- Refreshed knowledge on inserting aliased links.
- Practiced bash commands on the reference platform.
- Explored Cyber Physical Systems. Came across a very interesting company called "Zipline" that has implemented a logistics company for small package deliveries using drones. First implemented in Rwanda as an urgent medical supply carrier; it has incorporated a special propeller design that allows these drones to fly nearly silently to deliver urgent packages to people in need, all within a 150-mile radius. Awesome stuff!
- Interested in researching more into synchronized drone operations. Instead of using a single drone to try to accomplish a task, one could have a network of drones that combine to accomplish said task. This ties into a previous self-directed co-op where I was designing a proof of concept for a drone system that harvests conifer tree cones.

Week 4, January 29th, 2024

- Updated the bash cheatsheet to include scripting variables, conditionals, and loops.
- Updated the git cheatsheet.

Week 5, February 5th, 2024

- Created an outline for Assignment 1.
- Reviewed C concepts and notes from a previous C course.

- Reviewed the git workflow and set up a local git repo.
- Completed each part of the assignment and pushed to the remote repo.
- Finished a feature that puts higher constraints on buffer size - splits up larger input lines and saves split tokens for following calls.
- Tested and merged the feature back into the main branch of the remote repo.

Week 6, February 12th, 2024

- Created a new local repo in accordance with new submission requirements.
- Tested Assignment 1, submitted, and cloned to verify success.
- Created a LaTeX symbols cheatsheet.
- Started reviewing Python utilizing python.org. I have used Python sparingly in the past, mostly for manipulating CSV files in Excel and querying data, specifically using the pandas library. As I usually code in C++ and have taken a course in C in the past, the on-ramping for Assignment 1 was minimal. I will need to do a lot more review for Assignment 2 that utilizes Python.
- Removed LinkedIn and Github links and logos due to incorrect PDF formatting. Will try to implement for the second part of the portfolio.
- Formatted the reference page.

Week 7, February 12th, 2024

- Created a new local repo in accordance with new submission requirements.
- Tested Assignment 1, submitted, and cloned to verify success.
- Created a LaTeX symbols cheatsheet.
- Started reviewing Python utilizing python.org. I have used Python sparingly in the past, mostly for manipulating CSV files in Excel and querying data, specifically using the pandas library. As I usually code in C++ and have taken a course in C in the past, the on-ramping for Assignment 1 was minimal. I will need to do a lot more review for Assignment 2 that utilizes Python.
- Removed LinkedIn and Github links and logos due to incorrect PDF formatting. Will try to implement for the second part of the portfolio.
- Formatted the reference page.

Week 8, February 26th, 2024

- Reviewed the pandas library for use in Assignment 2.
- Implemented mini programs to practice Python comprehensions and collections - mostly worked with sets and dictionaries.
- Reviewed PEP 8 guidelines, utilized ChatGPT for validating PEP 8 guidelines being followed with practice programs.
- Completed the Assignment 2 outline, and a command-line parse function that utilizes a comprehension to create a dictionary of inputs.
- Researched the unittest module to create an additional test file for testing individual function outputs.

Week 9, March 4th, 2024

- Completed the Assignment 2 function that processes the table's data using pandas dataframes.
- Updated file line parsing to handle lines that exceed the max size of the read-in buffer. It verifies that a split token has occurred and reconstructs the split token on the next call to the file.
- Completed the 'write_to_file' function which does final table formatting before outputting to a `.csv` output file.
- Reviewed dynamic memory allocation in C.
- Reviewed and practiced linked list implementations.
- Reviewed struct and typedef for implementing object-oriented design.

Week 10, March 11th, 2024

- Verified the feature that successfully parses command-line arguments and handles split tokens, merged the feature into the main branch in the repo.
- Started Assignment 3, formed the assignment outline, noted further questions to get clarification on.
- For Assignment 3, created a `song_t` struct that holds a song record in each dynamically allocated list node.
- Implemented the argument parsing function for Assignment 3.
- Adjusted the song record structure, made each list node hold each song record variable instead of a pointer to a `song_t` struct. This allows for easier deallocation and reduces levels of dot accessing to get at variables - allowing for better readability.
- Completed the Assignment 3 `fill_record` function. Spent a considerable amount of time recreating a similar implementation that I had previously done for Assignment 1. Finally, decided to cut my losses and instead made a few small changes to the Assignment 1 implementation that worked perfectly - a valuable lesson learned on time management and the additional time needed on starting an implementation from scratch.
- Reviewed operator precedence.
- Researched tacit and implicit knowledge. It is very important as a software engineer to be able to form relationships with clientele and other domain experts to match the intended design; to ask the right questions and be able to explain the possible solutions as the software specialist.

Week 11, March 18th, 2024

- Researched function pointers for Assignment 3; implemented a `compare` function pointer and a helper function `get_compare` to assign a compare function to a function pointer based on relative inputs. Allows for singly-purposed functions, making it more modular, easier to maintain, and improved flexibility.
- Completed data processing and file output for Assignment 3. Tested and validated the repo for final submission.
- Researched Python Object-Oriented Design, and software engineering principles.
- Created small class programs to implement Object-Oriented Design, focusing on high cohesion and low coupling, encapsulation and abstraction, as well as some polymorphism.

Week 12, March 25th, 2024

- Created an outline for Assignment 4, roughed out the object-oriented design structure from the provided source files. Formed questions to get clarification on.
- For Assignment 4, added additional functionality for HTML component classes. Each HTML component inherits from the base class `HtmlComponent`, which encapsulates related members and methods.
- For Assignment 4, added a recursive private method in the base class for updating indentation levels when a component is added to another parent component. For example, it allows for adding nested `<div>` tags for separating HTML components and automatically adjusts the indentation.
- For Assignment 4, added a `tag_attributes` member in the base class `HtmlComponent` to allow additional functionality in HTML tags.
- For Assignment 4, consolidated SVG-specific shape classes into a base class that holds similar members and methods.
- For Assignment 4, converted the rigid render method of the shape class to loop through a dictionary of SVG shape tag attributes, allowing for a non-fixed length number of attributes inputted for each SVG generated shape.

Week 13, April 1st, 2024

- For Assignment 4, added a `PyArtConfig` class that sets ranges for art configuration that will be used to generate random shapes. Created default class variables which get assigned to ranges if the user did not define them.
- Researched random and pseudo-random number generation in Python.

Personal Insights

One of the most helpful things I've learned so far is the implementation of LaTeX code and HTML tags to achieve very useful typeset notes. I had previously incorporated a piece of software called Obsidian for organizing ideas and quick notes but had not gone to a full implementation of school notes. Previously, I lacked the ability to insert and resize pictures as well as quickly typeset formulas and mathematical symbols, but that has now changed. With this new structure in place, I've been able to typeset all my course notes and have them streamed on all devices through cloud storage, giving me access to study and make adjustments anywhere; school, home, or on the bus - I always have my notes.

Another personal insight I've had was the speed at which natural language processing tools, such as ChatGPT, allow you to arrive at an answer. It's an advanced Google search of sorts that you can quickly query a large amount of data on a subject. It's the ultimate assistant to maintain a dialogue and come to an answer a lot quicker than previous methods of scouring pages of Google search links or Stack Overflow pages.

Lastly, and most intriguing for my future aspirations in robotics, is the development of the cyber-physical systems(CPS) space. The ability to have smart systems that can act as a network of nodes that can dynamically take in data and adjust the network to perform tasks is extraordinary. Examples of networks of drones that can create amazing visual patterns in the

sky can also one day lead to the incorporation of advanced pattern recognition and task delegation to accomplish dynamically changing tasks out in the world. The amazing abilities that some of the newest multimodal assembly line robotics that are performing in controlled environments could one day be harnessed in dynamic environments with the help of smart systems.

Incorporating ChatGPT & Variants

Part 1
I tested OpenAI's ChatGPT and found that the platform delivered results quite quickly. However, with its free version, you only had access to version 3.5, and couldn't upload images. The results were often quite good for queries with large data samples, but it struggled with more niche tasks. Microsoft Bing's Copilot, which is built into their browser, provides access to the ChatGPT version 4 variant as well as the ability to upload images. However, I found that the response speed was significantly slower than OpenAI's website.

In general, these platforms are great for studying and querying lots of data from various sources. You can ask to have things rephrased, request visual examples, test code snippets, and get lists of external sources for verification or reference. Verification was a must, as there were numerous times when confidently incorrect answers were given. But, it allows you to start a dialogue and usually, with added input on why you thought its answer was wrong, it could arrive at the correct one.

I didn't directly use it to improve my software development workflow, but I used it a handful of times to restructure markdown tables. Well-structured, thought-out input was needed for large tasks. Creating markdown tables from a list of inputs and various constraints tended to take longer to perfect than to get information on basic syntax and structure and then manually input it. Or, provided it with a pattern and a list of data to populate that pattern - that tended to work more often. For example, the reference page was created doing that; querying copilot for the html/CSS syntax basics, forming a template for an IEEE citation, then providing Copilot with my finished template as well as a list of urls/headers to populate the template and repeat the pattern.

I found out that you can get a plug-in for VS Code for ChatGPT integration - that will be my next area to investigate. As I didn't opt for some of the paid versions, I haven't been able to access some of the more advanced features and implementations that various groups are offering. I surmise that this would be more in tune with improving developer workflow. For now, I will use it as a learning tool and an advanced Google search.

Part 2
Overall, I have found generative AI to be an invaluable tool for learning. I have used it extensively to clarify concepts, produce test cases, and to engage in immediate back-and-forth dialogues to better understand a concept. Although it can sometimes output incorrect information with the wrong prompt, if you continually rephrase and draw connections in other ways, it can find the information to back up your thought process, or offer resources that indicate where your thought process was incorrect. I consistently used it to verify overall code structure as I practiced with problems, giving me instant feedback on where I could improve the design; either to follow standard guidelines like PEP-8 or to improve overall Object-Oriented Design

(OOD). It was very helpful to be able to ask about overarching design concepts, possible benefits, and pitfalls that I may not have considered. I did not end up utilizing any of the generative AI plugins for some of the IDEs, but at this time, as I'm still learning concepts, it's beneficial to minimize the use of some of the larger automation tools. However, in the future, this will play a large role in improving the efficiency of a developer and software engineer - a skill that will be mandatory to stay relevant in the job market. Going forward, I am curious about how many employers are starting to implement generative AI in their software development, and if so, how much access they are giving these large platforms to their proprietary software. What implementations are companies using to mitigate possible data leaks or security vulnerabilities? Questions I look forward to posing to future hiring managers.

Markdown

A lightweight markup language for creating text documents that are formatted in a visually appealing, structured way.

Basic Syntax [1]

Element	Markdown Syntax

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`> blockquote`

Ordered List	`1. First item`
	`2. Second item`
	`3. Third item`

Unordered List	`- First item`
	`- Second item`
	`- Third item`

Code	`` `code` ``
Horizontal Rule	`---`
Link	`[title](https://www.example.com)`
Image	`![alt text](image.jpg)`

Extended Syntax [1]

Element	Markdown Syntax

Table	`| Syntax | Description |`
	`| ---------- | ---------- |`
	`| Header | Title |`
	`| Paragraph | Text |`

Fenced Code Block	```` ```
	{
	  "firstName": "John",
	  "lastName": "Smith",
	  "age": 25
	}
	``` ````

Footnote	`Here's a sentence with a footnote. [^1]`
	
	`[^1]: This is the footnote.`

Heading ID	`### My Great Heading {#custom-id}`
Definition List	`term` `: definition`
Strikethrough	`~~The world is flat.~~`

Task List	`- [x] Write the press release`
	`- [ ] Update the website`
	`- [ ] Contact the media`

Emoji (see also Copying and Pasting Emoji)	`That is so funny! :joy:`
Highlight	`I need to highlight these ==very important words==.`
Subscript	`H~2~O`
Superscript	`X^2^`

Resources

https://www.markdownguide.org (https://www.markdownguide.org)

LaTex Markup

A markup language for typesetting documents. It is used extensively in the academic and technical fields to create documents that accurately display formulas, non-standard characters, and images.

Quick Sheet

Category	Symbol	LaTeX Code	Description	Symbol	LaT
Logical Connectives	$\neg$	`\neg`	Negation	$\wedge$	
	$\vee$	`\lor`	Disjunction	$\rightarrow$	`\righ`
	$\leftrightarrow$	`\leftrightarrow`	Bi-implication	$\Leftrightarrow$	`\Leftrigh`
	$\Rightarrow$	`\Rightarrow`	Logically implies	$\Leftarrow$	`\Le`
Set Theory Operations	$\in$	`\in`	Element of	$\subseteq$	`\su`
	$\subset$	`\subset`	Subset of	$\cup$	
	$\cap$	`\cap`	Intersection	$\setminus$	`\se`
Universal Set Symbols	$\varnothing$	`\emptyset`	Empty set	$\mathbb{N}$	`\mat`
	$\mathbb{Z}$	`\mathbb{Z}`	Integers	$\mathbb{Q}$	`\mat`
	$\mathbb{R}$	`\mathbb{R}`	Real numbers	$\mathbb{C}$	`\mat`
Common Greek Letters	α	`\alpha`	Alpha	β	
	γ	`\gamma`	Gamma	δ	
	ϵ	`\epsilon`	Epsilon	ζ	
	η	`\eta`	Eta	θ	
	ι	`\iota`	Iota	κ	
	λ	`\lambda`	Lambda	μ	
	ν	`\nu`	Nu	ξ	
	π	`\pi`	Pi	ρ	
	σ	`\sigma`	Sigma	τ	
	υ	`\upsilon`	Upsilon	ϕ	
	χ	`\chi`	Chi	ψ	
	ω	`\omega`	Omega		

Category	Symbol	LaTeX Code	Description	Symbol	LaT
Basic Mathematical Operators	$\times$	`\times`	Multiplication	$\div$	
	$\pm$	`\pm`	Plus/minus	$\mp$	
	$\sqrt{x}$	`\sqrt{x}`	Square root of x	x^y	
Famous Mathematical Equation Symbols	$e^{i\pi} + 1 = 0$	`e^{i\pi}+1=0`	Euler's identity	$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$	`\sum_{n=1}^{\infty}\ {n^2}=\frac{\pi`
	$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$	`\int_{-\infty}^{\infty}e^{-x^2}dx=\sqrt{\pi}`	Gaussian integral		
Other Common LaTeX Commands	$\frac{a}{b}$	`\frac{a}{b}`	Fraction	$\lim_{x \to a} f(x)$	`\lim_{x\to`
	$\sum_{i=1}^{n} a_i$	`\sum_{i=1}^{n}a_i`	Summation	$\prod_{i=1}^{n} a_i$	`\prod_{i=1}^`

Resources

HTML and CSS

HTML(HyperText Markup Language) and CSS(Cascading Style Sheets) are two pf the main programming languages for structuring and style webpages.

Quick Sheet

Tag Type	Tag	Description
HTML	`<!DOCTYPE html>`	Defines the document type and version of HTML.
	`<html>`	Encloses the entire HTML document.
	`<head>`	Contains meta-information about the document.
	`<title>`	Specifies a title for the document.
	`<body>`	Contains the document's body content.
	`<h1>` to `<h6>`	Defines HTML headings.
	`<p>`	Defines a paragraph.

Tag Type	Tag	Description
	`<a href="url">`	Defines a hyperlink.
	`<img src="image.jpg" alt="My Image">`	Embeds an image.
	`<style>`	Defines internal CSS style of an HTML document.
	` `	Inserts a single line break.
	`<hr>`	Defines a thematic break in an HTML page (e.g., a shift of topic).
CSS	`color`	Sets the color of text.
	`font-size`	Sets the size of the font.
	`background-color`	Sets the background color of an element.
	`border`	Sets the border around an element.
	`padding`	Sets the padding (space inside the border) of an element.
	`margin`	Sets the margin (space outside the border) of an element.
	`display`	Specifies how an element should be displayed.
	`width and height`	Set the width and height of an element.

Resources

Bash

Bash is a linux shell that allows a user to interface with the operating systems kernal through a command-line. Bash commands allow for a high action-to-keystroke ratio. Also, through bash scripting, one can automate repetitive tasks.

Quick Sheet

Basic File and Directory Operations

Command	Command Description	Example	Optional Arguments
`ls`	List directory contents	`ls -l`	`-l` : use a long listing format `*.txt` : output only .txt files `-a` : do not ignore entries starting with . `-h` : with `-l` , print sizes in human readable format `-t` : sort by reverse chronological order
`cd`	Change the shell working directory	`cd /home/user/`	`cd ..` : up one directory `cd ~` : back to root directory `cd -` : recalls last directory `-L` : follow symbolic links `-P` : don't follow symbolic links
`pwd`	Print name of current/working directory	`pwd`	`-L` : print the value of `$PWD` if it names the current working directory `-P` : print the physical directory, without any symbolic links
`pushd`	allows you to push a dir onto the dir stack	`pushd ~\seng265`	
`popd`	Allows you to remove a dir from the dir stack, makes it your curr dir	`popd`	
`tree`	Outputs a tree diagram of a dir and its sub-dir	`tree`	lots of optional args, refer to `man tree`

File Manipulation

Command	Command Description	Example	Optional Arguments
`cat`	Concatenate and print (display) the content of files	`cat file.txt`	`-n` : number all output lines
`touch`	Change file timestamps	`touch newfile.txt`	`-a` : change only the access time `-m` : change only the modification time
`rm`	Remove files or directories	`rm file.txt`	`-r` : remove directories and their contents recursively `-f` : ignore nonexistent files and arguments, never prompt
`mkdir`	Make directories	`mkdir newdir`	`-p` : make parent directories as needed `-v` : print a message for each created directory
`rmdir`	Remove empty directories	`rmdir dir`	`-p` : remove DIRECTORY and its ancestors `-v` : output a diagnostic for every directory processed
`cp`	Copy files and directories, creates files if they don't exist	`cp file1.txt file2.txt`	`-i` : prompts user before overwriting `-b` : Create backup of file to be overwritten `-a` : preserve the specified attributes `-r` : copy directories recursively

Command	Command Description	Example	Optional Arguments
`mv`	Move (rename) files	`mv file1.txt file2.txt`	`-f` : do not prompt before overwriting `-i` : prompt before overwrite

Searching and Sorting

Command	Command Description	Example	Optional Arguments
`find`	Locate files on the system	`find /home -name "*.txt"`	`-name` : search for files by name `-type` : search by type
`grep`	Search text using patterns	`grep "hello" file.txt` `ls -lta | grep "^d"`	`-i` : ignore case distinctions `-c` : counts the # of occurences `-n` : prefixes each line with line # `^d` : retrieves lines starting with letter "d" `ing$` : retrieves lines ending with "ing" `-r` : read all files under each directory, recursively
`sort`	Sort lines of text files	`sort file.txt`	`-r` : reverse the result of comparisons `-n` : compare according to string numerical value

Changing Permissions

Command	Command Description	Example	Optional Arguments
`chmod`	Change file permissions	`chmod u+x file.txt`	`+x/-x` : add/remove execute perm. `+w/-w` : add/remove write perm. `+r/-r` : add/remove read perm. `u+` : add perm. to user `g+` : add perm. to group `o+` : add perm. to other
`chown`	Change file owner and group	`chown user:group file.txt`	`user:group` : change file owner and group

Archiving and Compression

Command	Command Description	Example	Optional Arguments
`tar`	Tape archive utility	`tar -cvf archive.tar file.txt`	`-c` : create a new archive `-v` : verbosely list the files processed `-f` : use archive file
`gzip`	Compress or expand files	`gzip file.txt`	`-d` : decompress
`gunzip`	Decompress gzip compressed files	`gunzip file.txt.gz`	None

Viewing File and System Details

Command	Command Description	Example	Optional Arguments
`head`	Output the first part of files	`head -n 5 file.txt`	`-n` : output the first NUM lines
`tail`	Output the last part of files	`tail -n 5 file.txt`	`-n` : output the last NUM lines
`echo`	Display a line of text	`echo "Hello, World!"`	None
`date`	Show current date/time	`date`	`+%Y-%m-%d-%H-%M-%S` : yyyy-mm-dd-hh-MM-SS
`df`	Report file system disk space usage	`df -h`	`-h` : print sizes in human readable format
`du`	Estimate file and directory space usage	`du -sh /home/user`	`-s` : display only a total for each argument `-h` : print sizes in human readable format
`ps`	Report a snapshot of the current processes	`ps -e`	`-e` : select all processes
`man`	opens manual for bash cmd	`man -f chmod`	`-f` : Display a one-line description of the cmd `-k` : Search for the cmds related to a given arg `-a` : Display all matching man. pgs for the specified cmd

Control Operations

Command	Command Description	Example	Optional Arguments
`kill`	Send a signal to a process	`kill -9 12345`	`-9` : SIGKILL
`history`	Command Line History	`history`	`-c` : clear history list `!!` : Repeat last command `!25` : Repeat the last, 25 th*command* `!arg` : Repeat the last command starting with 'arg'
`exit`	Exit the shell	`exit`	
`pipe`	Pipe operator, connects commands	`command1 | command2`	
`>`	Redirection operator, redirects command output	`command > file`	
`>>`	Redirection/append operator, redirects command output & appends it	`command >> file`	
`<`	Input redirection operator	`stdin < file`	
`2> (2>>)'	Write to stderr (append)	`command 2> file (command 2>> file)`	
`&>`	Redirects both stdout and stderr	`command &> file`	

Networking

Command	Command Description	Example	Optional Arguments
`curl`	allows for data transfer between servers. Can use one of the following supported protocols. (HTTPS,HTTP, SCP, SFTP, and FTP)	`curl -O [URL]`	`-O` : download the file from remote server and save with URL name `-L` : allows for automatic redirection if the page has been moved and link provided `-u` : specify the user and password for server authentication

Scripting, Conditionals, & Loops

Scripting

`$` References the value of a variable

`((...))` Instructs arithmetic expansion, indicates that the operation inside the perenthesis is arithmetic.

`$0, $1, $2, ...` variables passed as arguments. `$0` is the name of the file. `$1` being the first arg

`$#` stores the number of arguments passed in the bash script.

`#` Used to make comments

`#!/bin/bash` Shebang that goes on the first line of every bash script

Conditionals

if - used to test a single condition

```
if [ condition ]
then
    # code to be executed if condition is true
fi
```

if-Else - used to test a 2 conditions

```
if [ condition ]
then
    # code to be executed if condition is true
else
    # code to be executed if condition is false
fi
```

if-Elif-Else - multi-conditional execution

```
if [ condition1 ]
then
    # code to be executed if condition1 is true
elif [ condition2 ]
then
    # code to be executed if condition1 is false and condition2 is tru
e
else
    # code to be executed if both condition1 and condition2 are false
fi
```

Case - route control to various cases.

```
case expression in
    pattern1 )
        # code to be executed if expression matches pattern1
        ;;
    pattern2 )
        # code to be executed if expression matches pattern2
        ;;
    * )
        # code to be executed if expression doesn't match any pattern
        ;;
esac
```

Loops

C-Style For

```
for ((initialize; condition; increment)); do
    [COMMANDS]
done
```

List/Range For - useful for incrementing arrays, lists or files sequentially

```
for item in [LIST]; do
    [COMMANDS]
done
```

While - execute loop until a particular condition is true

Vim

Vim is an open-source Unix text editor which allows for easy command-line editing of files.

Quick Sheet

Mode	Command	Command Description
Insert Mode Commands	`i`	Enter insert mode after the current character
	`a`	Enter insert mode at the end of the line
	`o`	Open a new line below and enter insert mode
	`O`	Open a new line above and enter insert mode
Visual Mode Commands	`v`	Start visual mode for selecting text
	`V`	Start visual line mode for selecting lines
	`ctrl + v`	Start visual block mode for selecting blocks
Yank, Delete, and Paste Commands	`y`	Yank (copy) the selected text
	`d`	Delete the selected text
	`p`	Paste the yanked or deleted text after cursor
	`P`	Paste the yanked or deleted text before cursor
Undo and Redo Commands	`u`	Undo the last operation
	`ctrl + r`	Redo the last undone operation
File and Buffer Commands	`:w`	Write (save) the file
	`:q`	Quit Vim
	`:wq`	Write and quit Vim
	`:q!`	Quit Vim without saving changes
	`:e filename`	Open a file in Vim
Search and Replace Commands	`/pattern`	Search for a pattern in the file
	`n`	Go to the next match of the last search
	`N`	Go to previous match of last search
	`:%s/old/new/gi`	Replace all occurrences of 'old' with 'new' in file
Movement Commands	`gg`	Go to first line of file
	`G`	Go to last line of file
	`#G`	Go to line number #

Resources

- Vim Comprehensive Command-List (https://phoenixnap.com/kb/wp-content/uploads/2021/11/vim-commands-cheat-sheet-by-pnap.pdf)
- https://swcarpentry.github.io/shell-novice/reference.html (https://swcarpentry.github.io/shell-novice/reference.html)
- https://en.wikipedia.org/wiki/Bash_(Unix_shell) (https://en.wikipedia.org/wiki/Bash_(Unix_shell))

GIT

Version control software that allows for incremental developement and management of code or documents. Allows for large teams to effective work together and introduce new features into original source after thorough testing. Git alongside a remote client is an integral part of software developement and project life cycle.

QuickSheet

Command	Description
`git init`	Initialize an existing directory as a Git repository
`git clone [url]`	Retrieve an entire repository from a hosted location via URL
`git status`	Show modified files in working directory, staged for your next commit
`git add [file]`	Add a file as it looks now to your next commit (stage)
`git reset [file]`	Unstage a file while retaining the changes in working directory
`git diff`	Diff of what is changed but not staged
`git diff --staged`	Diff of what is staged but not yet committed
`git commit -m “[descriptive message]”`	Commit your staged content as a new commit snapshot
`git branch`	List your branches
`git branch [branch-name]`	Create a new branch at the current commit
`git checkout`	Switch to another branch and check it out into your working directory
`git merge [branch]`	Merge the specified branch’s history into the current one
`git log`	Show all commits in the current branch’s history
`git remote add [alias] [url]`	Add a git URL as an alias
`git fetch [alias]`	Fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	Merge the specified branch’s history into the current one
`git remote add [alias] [url]`	Add a git URL as an alias
`git remote -v`	List all currently configured remote repositories
`git fetch [alias]`	Fetch down all the branches from that Git remote
`git pull [alias] [branch]`	Fetch and merge any commits from the tracking remote branch
`git push [alias] [branch]`	Push your commits to the remote repository

Resources

- https://git-scm.com/book/en/v2/Git-Basics-Summary (https://git-scm.com/book/en/v2/Git-

C

C is a general-purpose programming language that is at the heart of major software infrastructure that is in use today. It was created at AT&T Bell Labs in the early '70s to run on the Unix operating system, which was created shortly before, by Dennis Ritchie, Ken Thompson, and others. Its ability to allow for near-machine programming gives it a lot of flexibility and efficiency. As of Feb. 2024, the TIOBE Programming Community index places C as the 2^{nd} most popular language (10.97%) [2].

Resources

- https://cppreference.com (https://cppreference.com)
- https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html (https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html)

GCC

Command	Example	Description
-c	gcc myfile.c	Compiles the source file myfile.c, but does not link it - creates an object file myfile.o
-o	gcc -o myfile myfile.c	Compiles and creates an executable named myfile
-Wall	gcc -Wall myfile.c	Enables all warnings in GCC
-g	gcc -g myfile.c	Includes debugging information in the output file
-O	gcc -O myfile.c	Optimizes your code for better performance or smaller size
-l	gcc myfile.c -lm	Links against the library libm.a - Doesn't require the prefix 'lib'
-E	gcc -E myfile.c	Produces only the preprocessor output
-L	gcc -S myfile.c	Used to specify location of libraries that the linker should use.
-I	gcc -C myfile.c	Used to specify location of include files that the preprocessor uses.
-pedantic	gcc -save-temps myfile.c	Used to enforce strict ANSI compliance in code, allowing for better portability across compilers

C Focused Topics

C Structs for Object Oriented Design

A struct in C is a user-defined data type that allows you to group different types of variables together. It's like a blueprint for creating more complex data types.

Type defining: `struct node_t` is the type we are aliasing to `node_t` using keyword `typedef`.

```c
typedef struct node_t
{
    char track_name[200];
    char artist[200];
    unsigned int artist_count;
    struct tm date_;
    unsigned long streams;
    struct node_t *next;
} node_t;

int main() {
    node_t song_record; // This is a song record object of type node_t
    return 0;
```

Pre-declaration for Pointer to Structs
The pre-declaration typedef struct Node Node; informs the compiler that Node is a struct type, even if the specifics of this struct are not yet provided. This enables us to establish a NodePtr

```c
typedef struct Node Node; // Forward declaration of the struct Node
typedef Node* NodePtr; // Define NodePtr as a pointer to Node

struct Node { // Define the struct Node
    int data;
    NodePtr next; // Use NodePtr in the struct definition
};
```

Lists

Singly-Linked List

Each node has some data and a `next` pointer that points to the next node or `NULL`.
Defining

```c
typedef struct Node {
    int data;
    struct Node* next;
} Node;
```

Taversing

```
void traverse(Node* head) {
    Node* current = head;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
}
```

Insert at Beginning

```
Node* insert_at_beginning(Node* head, int data) {
    Node* new_node = malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = head;
    return new_node;
}
```

Insert After a Node

```
void insertAfter(Node* prev_node, int data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL");
        return;
    }

    // Create a new node
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;

    // Make next of new node as next of prev_node
    newNode->next = prev_node->next;

    // Move the next of prev_node as new_node
    prev_node->next = newNode;
}
```

Doubly-Linked List

Each node adds a `prev` pointer that points to previous node. This allows for forward and backwards traversing.

```
typedef struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
} Node;
```

Insert at Beginning

```
Node* insert_at_beginning(Node* head, int data) {
    Node* new_node = malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = head;
    new_node->prev = NULL;
    if (head != NULL) {
        head->prev = new_node;
    }
    return new_node;
}
```

Insert After a Node

```
void insertAfter(Node* prev_node, int data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL");
        return;
    }

    // Create a new node
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    newNode->prev = NULL;

    // Make next of new node as next of prev_node
    newNode->next = prev_node->next;

    // Make the next of prev_node as new_node
    prev_node->next = newNode;

    // Make prev_node as previous of new_node
    newNode->prev = prev_node;

    // Change previous of new_node's next node
    if (newNode->next != NULL)
        newNode->next->prev = newNode;
}
```

Circular List

Contains the same struct setup as a singly-linked list, but instead of the last node pointing to `NULL`, it points to the `head` node. **Traversing**

```c
void traverse(Node* head) {
    Node* current = head;
    do {
        printf("%d ", current->data);
        current = current->next;
    } while (current != head);
}
```

Inserting at Beginning

```c
Node* insert_at_beginning(Node* head, int data) {
    Node* new_node = malloc(sizeof(Node));
    new_node->data = data;
    new_node->next = head;

    Node* current = head;
    while (current->next != head) {
        current = current->next;
    }
    current->next = new_node;

    return new_node;
}
```

Function Pointers

A function pointer allows you to store the address of a callable function. This is useful for passing purpose build functions as parameters to other functions or storing within data structures.

- **Flexibility and Reusability**: Function pointers provide a way to write code that is both flexible and reusable. They allow you to create generic functions that accept function pointers as parameters. These functions can then perform different operations based on the passed-in function. This leads to a reduction in code duplication as the same function can be reused for various operations.
- **Dynamic Behavior**: Function pointers facilitate dynamic behavior by enabling you to call a function determined at runtime. This is particularly useful when the function to be called is based on user input or other conditions determined during runtime.
- **Code Organization and Readability**: Function pointers can enhance code organization and readability. They can be used to implement callbacks, which can make your code easier to comprehend. Additionally, they allow you to group related functions together, leading to more organized code.

Structure: `return_type (*pointer_name)(parameter_types);`

Function Pointer Implementation

Function pointer declaration
`int (*compare)(node_t *, node_t *, int);` - This is a function pointer named `compare` that points to a function taking two `node_t*` and an `int` as parameters and returning an `int`.

```
typedef int (*compare)(node_t *, node_t *, int);
```

Apply function for applying a function to a list Using a function pointer:
The apply function takes a linked list, a function pointer fn, and a void pointer arg as parameters. It applies the function `fn` to each node in the list.

```
void apply(node_t *list, void (*fn)(node_t *list, void *), void *arg)
{
    for (; list != NULL; list = list->next) {
        (*fn)(list, arg);
    }
}
```

Purpose Built Compare Functions:
`compare_by_streams` and `compare_by_apple_playlists` are functions that can be pointed to by the compare function pointer. They compare two nodes based on their `streams` and `in_apple_playlists` fields, respectively.

```
int compare_by_streams(node_t *a, node_t *b, int order) {
    if (order > 0) {
        return (a->streams > b->streams) - (a->streams < b->streams);
    } else {
        return (b->streams > a->streams) - (b->streams < a->streams);
    }
}
```

Make

- **Make**: A build automation tool that uses Makefiles to construct programs and libraries.
- **Makefile**: A script for Make, detailing how to create the target program from its sources.
- **Rules**: Instructions in Makefiles for building your program. Can be explicit (user-defined) or implicit (predefined).
- **Variables**: Used in Makefiles to streamline and centralize the handling of compiler flags, directories, etc.
- **Automatic Variables**: Variables set by `make` for use within a rule, such as `$@` for the target and `$^` for all prerequisites.
- **Pattern Rules**: Rules that define how to build files based on their names, allowing a single rule to apply to multiple files.
- **Phony Targets**: Targets not tied to files, always considered out of date and thus always executed, useful for always-needed tasks like `clean`.

- **Functions**: Supported by `make` for various operations like string manipulation and file name manipulation.
- **Conditional Parts of Makefiles**: Parts of a Makefile included based on variable values, useful for environment or configuration-specific rules or settings.

Resources

- https://www.gnu.org/software/make/ (https://www.gnu.org/software/make/)
- https://makefiletutorial.com/ (https://makefiletutorial.com/)

Makefile Implementation

Set C compiler (CC) to gcc

```
CC=gcc
```

C-flags to be used:

```
CFLAGS= -c -Wall -g --DDEBUG -std=c99 -O0
```

- `-c` : generate object files from the source files. Useful for large projects where you don't want to recompile each single file everytime you make a change to one. Only need to recompile one, and then link all.
- `-Wall` : Enable all compiler warning messages.
- `-g` : Tells the compiler to include all debugging information in executable.
- `-DDEBUG` : Defines a preprocessor macro called DEBUG. Can be used to conditionally compile sections of code for debugging.
- `-std=c99` : C standard to be used for compiling.
- `-O0` : Tells the compiler not to optimize code. Helpful for debugging because errors and output are more in line with the code you have written.

`all` : indicates the default target to be built if no arugments is given to make. Ex. `make` or `make all`

```
all: default_file1 \
    default_file2
```

- continue list of default files to proceeding line using the backslash \

Create executable of `target_name` .

```
target_name: target_file1.o target_file2.o target_file3.o
    $(CC) target_name.o target_file2.o target_file3.o -o target_na
me
```

- first line describes target: and files associated

- second line uses the C compiler to create an executable, and output name `-o target_name`

Create an `target_name.o` object file.

```
target_name.o: target_name.c helper1.h helper2.h
    $(CC) $(CFLAGS) target_name.c
```

- list source and header files used
- secondline uses C compiler to compile, with c-flags, `target_name` source file.

Similarly as above, used to create object file of `helper1` use the needed src and header files

```
helper1.o: helper1.c helper1.h helper2.h
    $(CC) $(CFLAGS) helper1.c
```

- Compiling using C compiler and c-flags included.

Same as above but for `helper2`

```
helper2.o: helper2.c helper2.h
    $(CC) $(CFLAGS) helper2.c
```

Special target that doesn't depend on anything. Can be called in this instance to remove all object files in the current directory.

```
clean:
```

Python

Python is a high-level programming language that it known for its ease of use and is prevalent in the data science community and developer community as whole for its extensive library selection. Python is an interpreted language, utilizing line by line interpretation at runtime by a interpretor, versus a compiled language such as C which compiles the whole program into an executable before execution. Created in the 1980s by Guido van Rossum, Python has become the most popular programming language in the world [3]. As of Feb. 2024, the TIOBE Programming Community index places Python as the 1^{st} most popular language (15.16%) [2].

Style Guide [4]

PEP-8 – Style Guide for Python Code https://peps.python.org/pep-0008/ (https://peps.python.org/pep-0008/)

Code Layout:

- Use 4 spaces per indentation level.
- All lines should be a maximum of 79 characters.
- Blank lines should be used to separate functions and classes. Two lines between classes, one line between functions and methods.

Naming Conventions:

- Function names should be lowercase, with words separated by underscores as necessary to improve readability.
- Variable names and function names should follow the same convention.
- Class names are in CapWords.

Whitespace in Expressions and Statements:

- Avoid extraneous whitespace in the following situations:
  - Right inside parentheses, brackets or braces.
  - Right before a comma, semicolon, or colon.
  - Right before the open parenthesis that starts the argument list of a function call.

Comments:

- Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes.
- Comments should be complete sentences. The first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).
- If a comment is short, the period at the end can be omitted.
- Block comments consist of one or more complete sentences.

Programming Recommendations:

- Code should be written in a way that does not disadvantage other implementations of Python.
- You should always use `is` or `is not` when making comparisons with singletons such as `None`, instead of using equality operators.
- Also, beware of writing `if x` when you really mean `if x is not None`.

Libraries

The following list contains my top five favorite Python libraries, ranked from top to bottom. Pandas ranks number one for the sole reason that I'm most familiar with it. I have used it extensively this semester in assignments, as well as previously for manipulating Excel workbooks and connecting to online databases through an API. The libraries following Pandas are ones that intrigue me due to my future aspirations. I've lightly delved into them but will be further investigating as I advance in related courses.

Names	Summary	Resources
Pandas	Pandas is a free machine learning library for Python. It supports both supervised and unsupervised machine learning.	Pandas (https://pandas.pydata.org/docs/)

Names	Summary	Resources
TensorFlow	TensorFlow is an end-to-end open source platform for machine learning.	TensorFlow (https://www.tensorflow.org/api_docs)
Scikit-learn	Scikit-Learn is a free machine learning library for Python.	Scikit-learn (https://scikit-learn.org/stable/user_guide.html)
PyTorch	PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.	PyTorch (https://pytorch.org/docs/stable/index.html)
Keras	Keras is a high-level, user-friendly API used for building and training deep learning models.	Keras (https://keras.io/guides/)
NumPy	NumPy (Numerical Python) is an open-source Python library that's used in almost every field of science and engineering.	NumPy (https://numpy.org/doc/)
SciPy	SciPy is a scientific computation library that uses NumPy underneath.	SciPy (https://docs.scipy.org/doc//scipy/index.html)
Matplotlib	Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.	Matplotlib (https://matplotlib.org/stable/index.html)
Seaborn	Seaborn is a library for making statistical graphics in Python. It builds on top of matplotlib and integrates closely with pandas data structures.	Seaborn (https://seaborn.pydata.org/)
Theano	Theano is a Python library for fast numerical computation that can be run on the CPU or GPU.	Theano (https://media.readthedocs.org/pdf/theano/latest/theano.pdf)

Basics

- Python Beginners Guide (https://wiki.python.org/moin/BeginnersGuide)
- https://wiki.python.org/ (https://wiki.python.org/)

Data Structures:

- **Numeric Types (`int` , `float` , `complex`)**: These types represent numbers. Integers, floating point numbers and complex numbers fall under Python's numeric types.
- **Sequence Types (`str` , `list` , `tuple` , `range`)**: These types represent sequences of values. Strings, lists, tuples and ranges are examples of sequence types.
- **Mapping Type (`dict`)**: This type represents a collection of key-value pairs. Dictionaries are Python's sole mapping type.
- **Set Types (`set` , `frozenset`)**: These types represent a collection of unique elements. Sets and frozen sets are examples of set types.
- **Boolean Type (`bool`)**: This type represents the truth values `True` and `False` .

- **Binary Types (`bytes` , `bytearray` , `memoryview`)**: These types are used to handle binary data.
- **None Type (`NoneType`)**: This type represents the absence of a value or a null value.

Immutable: `int` , `float` , `complex` , `str` , `tuple` , `frozenset`

Mutable: `list` , `set` , `dict` , `bytearray` , `deque`

Namespace and Scope

Namespaces in Python are systems that ensure each object has a unique name within that namespace. This helps to prevent conflicts that can occur when different objects have the same name.

LEGB Rule: Python follows the LEGB rule for name resolution. The letters stand for Local, Enclosing, Global, and Built-in. Python first looks for a name in the Local scope (inside the current function), then in the Enclosing scope (in outer functions), then in the Global scope (at

In [1]:
```python
x = 'global x'  # Global scope

def outer():
    x = 'outer x'  # Enclosing scope

    def inner():
        x = 'inner x'  # Local scope
        print(f"inner: {x}")

    inner()
    print(f"outer: {x}")

outer()
print(f"global: {x}")
```

```
inner: inner x
outer: outer x
global: global x
```

Python Focused Topics:

Pythonic Programming

Pythonic programming refers to the conventions that yield clear, concise, and efficient code utilizing modern Python features.

Key aspects:

- **Readability**: Pythonic code is easy to read and understand.
- **Simplicity**: Pythonic code avoids complex or clever solutions when simpler ones are available.

- **Explicitness**: Pythonic code doesn't hide its behavior and makes its operations clear.
- **Consistency**: Pythonic code follows established conventions and styles, such as **PEP-8**.

Built-in Functions and Libraries

Utilizing Pythons rich set of built-in functions and libraries that have been designed for efficiency and readibility. [5]

Built-in Functions

A
abs()
aiter()
all()
anext()
any()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Comprehensions

Comprehensions replace multiline loops with a clear, concise syntax for initializing containers or generating data.

List, Set, Dict Comprehensions:

```
# Generating a list of squares from 0 to 9
squares = [x**2 for x in range(10)]

# Generating a set of squares from 0 to 9
squares = {x**2 for x in range(10)}

# Generating a Dictionary of squares from 0 to 9
squares = {x: x**2 for x in range(10)}
```

Generator Comprehension:

```
# Generating on-the-fly values instead of storing the whole sequence i
n memory.
squares = (x**2 for x in range(10))
```

Generators

A generator is a function that preserves the state of all variables and returns an iterator. It allows for on-the-fly generation of variables, which is more memory-efficient and provides a faster implementation of an iterable, especially when working with large data sets. The keyword `yield` is used instead of `ret` to preserve the function's state between calls.

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

# Create a generator
fib = fibonacci(10)

# Iterate over the generator
for number in fib:
    print(number)
```

Default Arguments

Utilizing default argument `None` when declaring variables, then initializing with actual defualt within function/method declarators. This ensures subtle bugs are not introduced into the program.

```
def add_key_value(key, value, dictionary=None):
    """Adds a key-value pair to a dictionary.
    If no dictionary is provided, creates a new one."""
    if dictionary is None:
        dictionary = {}
    dictionary[key] = value
    return dictionary

# Call the function with a custom dictionary
custom_dict = add_key_value("Alice", 25, {"Bob": 30})
print(custom_dict)  # Outputs: {'Bob': 30, 'Alice': 25}

# Call the function with the default dictionary
default_dict = add_key_value("Charlie", 35)
```

Random Numbers

Pseudo-random numbers can be generated using the `random` module. These very long repeating sequences based of a particular seed value can give random-like number generation. Default seed value is the current time from the system clock. Use should be restricted to modeling and simulation; it's stronly discourage for any sort of cryptographic needs. If security is needed, the `secrets` module is designed to produce cryptographically strong random numbers.

In [11]:

```
import random

divider = '--------------------'
# set seed to integer 1
random.seed(1)

# Prints a random integer between 1 and 10
print(random.randint(1, 10))

# Prints a random float between 1.0 and 10.0
print(random.uniform(1.0, 10.0))

# Random Sample
list1 = [1, 2, 3, 4, 5, 6]

# Prints a list with 3 random items from list1
print(random.sample(list1, 3))

# Random Choice from Weighted List
weights = [10, 20, 30, 40, 50, 60]

# Prints a list with 3 random items from list1,
# with a higher chance for items with higher weights
print(random.choices(list1, weights, k=3))

# set seed to system clock
print(divider)
random.seed()
print(random.randint(1, 10))
print(random.uniform(1.0, 10.0))
print(random.sample(list1, 3))
print(random.choices(list1, weights, k=3))

# Set seed back to integer 1 to attain same sequences
print(divider)
random.seed(1)
print(random.randint(1, 10))
print(random.uniform(1.0, 10.0))
print(random.sample(list1, 3))
print(random.choices(list1, weights, k=3))
```

```
3
6.12283487339991
[1, 3, 6]
[5, 4, 5]
--------------------
9
8.970427495928153
[2, 1, 6]
[6, 5, 6]
--------------------
3
6.12283487339991
[1, 3, 6]
[5, 4, 5]
```

Object-Oriented Design (OOD)

Software design methology that puts fucus on software implementation through objects and classes to model real-world entities and their behavior.

Key Principles

- **High Cohesion**: Designing modules or classes that contain elements which are functionally related. Each class should have a singlely defined task to perform that allows the class to be easily understood and broadly incorporated into other projects without redesign.
- **Low Coupling**: Designing classes and modules that have low dependences with other classes/modules. Allows for easy modification to the individual classes without having a chain reaction effect to other segments.
- **Encapsulation**: Mechanism of binding related code or data together and writing specific methods for accessing said data in a set way.
- **Abstraction**: Method of hiding the inner workings of a component and just shows the functionality of it to the user.
- **Inheritance**: A hierarchy system that allows related objects to inherit similar attributes/functions to reuse purpose built functionalities.
- **Polymorphism**: Design practice that creates a common interface which can operate of a variety of data types. There is two types:

    1. **Static Polymorphism**: A compile-time polymorphism where methods/functions can be overloaded and to accept varying data types based off compile-time data.
    2. **Dynamic Polymorphism**: A run-time polymorphism where depending on runtime variables specific members will override base class members to perform a task.
- **Modularity**: A design principle that enphasizes the separation of a program into independent modules that perform a well defined task and has low coupling. It improves reusability and maintainability, becoming a strong foundation where other additions can be done without compromising previous modules.

Techniques:

- Functions and Methods:
- Classes and Objects

- Modules and Packages
- Layered architecture
- Pipes & filters architecture

ODD Implementation

A Banking system design that demonstrate software engineering principles and hierarchy design. Most notably, Is-A relationship where a `SavingsAccount` is-a type of the base class `Account`, who it inherits common functionality from:

Account Class: This class demonstrates ***High Cohesion*** by having a single responsibility of managing account data. It also demonstrates ***Encapsulation*** by hiding its internal state (_balance) and exposing a limited interface (deposit, withdraw).

```
class Account:
    """Account class"""
    def __init__(self, initial_balance=0):
        self._balance = initial_balance  # Encapsulated attribute

    def deposit(self, amount):  # Abstraction
        self._balance += amount

    def withdraw(self, amount):  # Abstraction
        if amount > self._balance:
            raise ValueError("Insufficient balance")
        self._balance -= amount

    def get_balance(self):  # Abstraction
        return self._balance
```

SavingsAccount Class: This class demonstrates ***Inheritance*** by inheriting from the Account class. It also demonstrates ***Polymorphism*** by overriding the withdraw method.

```
class SavingsAccount(Account):
    """SavingsAccount class"""
    def __init__(self, initial_balance=0, interest_rate=0):
        """super() used to initialize parent class inherited attribut
e"""
        super().__init__(initial_balance)
        self._interest_rate = interest_rate  # New attribute

    def withdraw(self, amount):  # Method overriding
        if amount > self._balance:
            raise ValueError("Insufficient balance")
        self._balance -= amount
        self._balance -= amount * self._interest_rate  # Deduct intere
st
```

Bank Class: This class demonstrates ***Low Coupling*** by not relying on the internal workings of the Account or SavingsAccount classes. It also demonstrates ***High Cohesion*** by having a single responsibility of managing bank data.

```
class Bank:
    """Bank class"""
    def __init__(self):
        self._accounts = {}  # Encapsulated attribute

    def add_account(self, account_number, account):  # Abstraction
        self._accounts[account_number] = account

    def get_account_balance(self, account_number):  # Abstraction
        if account_number not in self._accounts:
            raise ValueError("Account number not found")
        return self._accounts[account_number].get_balance()
```

perform_transactions Function: This function demonstrates ***Modularity*** by performing a well-defined task that can be used across different parts of the program.

```
def perform_transactions(bank):
    """perform_transactions function"""
    # Deposit money into account number 123
    bank._accounts[123].deposit(100)

    # Withdraw money from account number 456
    try:
        bank._accounts[456].withdraw(50)
    except ValueError as e:
```

Tacit and Implicit Knowledge

Tacit Knowledge: This is knowledge that's gained through personal experience and is often hard to articulate or document. It includes insights, intuitions, and hunches. In the context of software engineering, it could be a developer's knack for efficient algorithm design or their intuitive understanding of system performance. The following are some examples.

- A developer's knack for designing efficient algorithms.
- Intuitive understanding of system performance and optimization.
- The ability to predict and mitigate potential risks in a project.

Implicit Knowledge: This is knowledge that's not explicitly stated but is implied or understood. It's often shared through common practices, assumptions, or values. In software engineering, it could be the unwritten rules about code organization or the assumed best practices for code reviews. The following are some examples.

- Unwritten rules about how code should be organized in a project.
- Assumed best practices for conducting code reviews.

- Shared understanding of the project's quality standards and expectations.

References

[1] "Markdown Cheat Sheet | Markdown Guide," Markdown Guide. [Online]. Available: https://www.tiobe.com/tiobe-index/ (https://www.tiobe.com/tiobe-index/). [Accessed: Feb. 16, 2024].

[2] “TIOBE Index,” TIOBE, Jan. 2024. [Online]. Available: https://www.geeksforgeeks.org/what-is-python/ (https://www.geeksforgeeks.org/what-is-python/). [Accessed: Feb. 16, 2024].

[3] “What is Python?,” geeksforgeeks, 2024. [Online]. Available: https://www.markdownguide.org/cheat-sheet/ (https://www.markdownguide.org/cheat-sheet/). [Accessed: Feb. 16, 2024].

[4] G. van Rossum, B. Warsaw, and A. Coghlan, "PEP 8 – Style Guide for Python Code," 2001. [Online]. Available: https://peps.python.org/pep-0008// (https://peps.python.org/pep-0008//). [Accessed: Apr. 5, 2024].

[5] “Built-in Functions,” Python.org, 2024. [Online]. Available: https://docs.python.org/3/library/functions.html (https://docs.python.org/3/library/functions.html). [Accessed: April. 4, 2024].