
Introduction to Python Software Development on GitHub

Introduction

About Us

Kevin Dean
Ph.D. Biochemistry
B.A. Chemistry

Zach Marin
Ph.D. Biomedical Engineering
M.S. Mathematics
B.S. Bioengineering, B.A. Mathematics

Dushyant Mehra
Ph.D. Biomedical Engineering & Physiology
B.S. Biomedical Engineering

https://github.com/TheDeanLab

https://www.dean-lab.org/

Course Structure

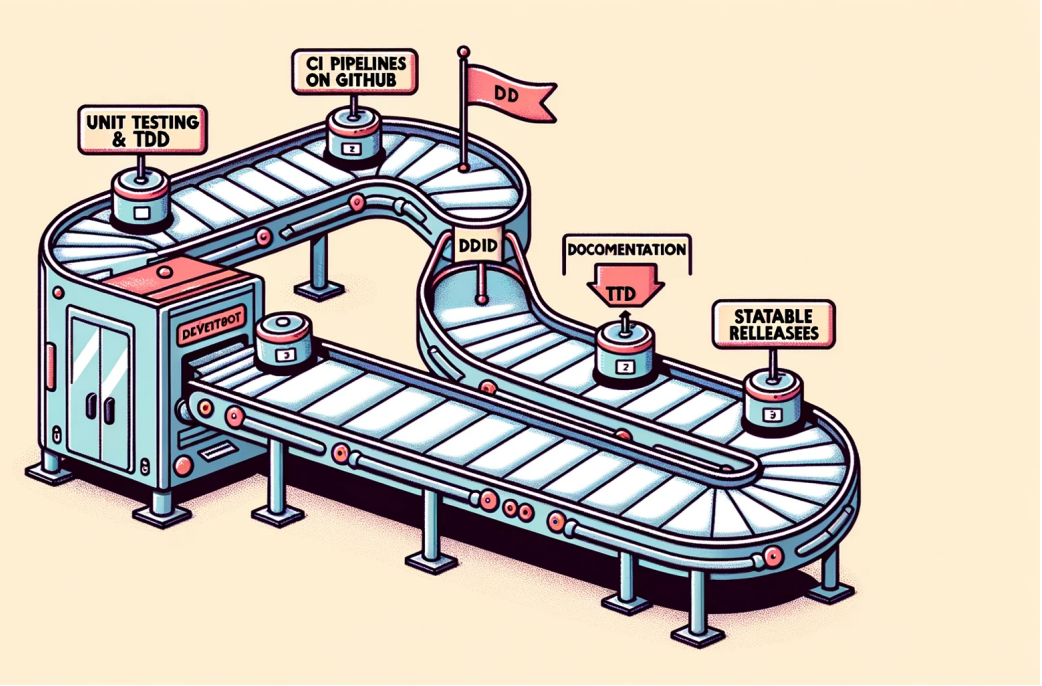
Day 1

- Introduction to Collaborative Software Development
- Environment Management & GIT Essentials
- Organizational Strategies, and Collaborative Development Workflows
- Pre-Commit Hooks, Linters, Code Formatters

Course Structure

Day 2

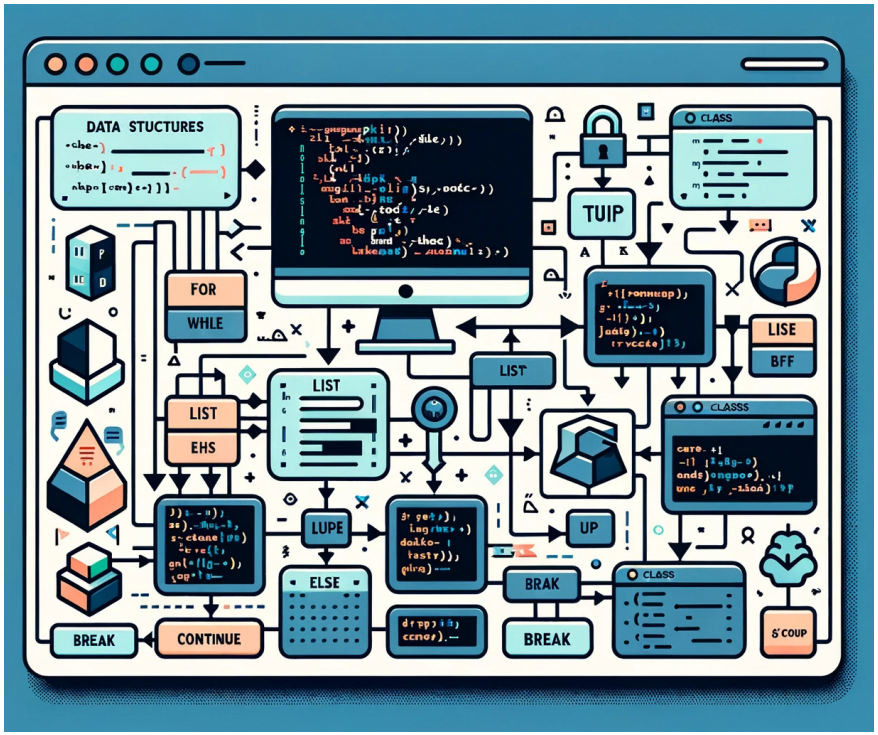
- Unit Testing and Test-Driven Development (TDD)
- Setting up Continuous Integration Pipelines on GitHub
- Public-Facing Documentation
- Stable Releases



Expectations

Coding with Python

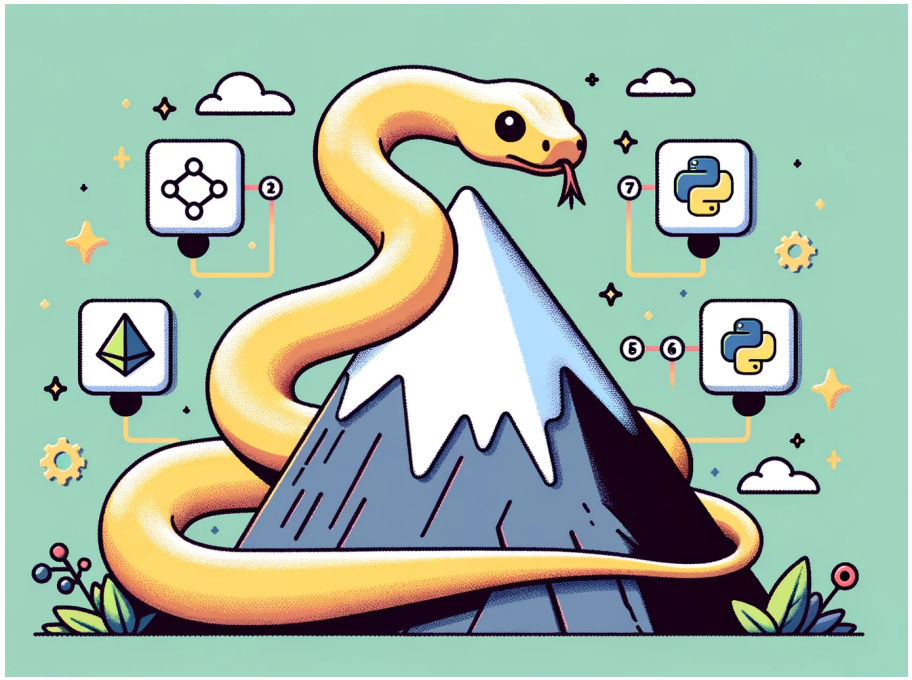
- *Mandatory*
  - How to work with Python Data Structures (dictionaries, lists, tuples, etc.)…
  - Python Control Flow (for, while, if, else, elif, break, continue)…
- *Recommended*
  - Object Oriented Programming (classes, objects, inheritance)…
- *Bonus*
  - Parallel Processing with Python (multi-threading, multi-processing)…



Goals

Building a Foundation

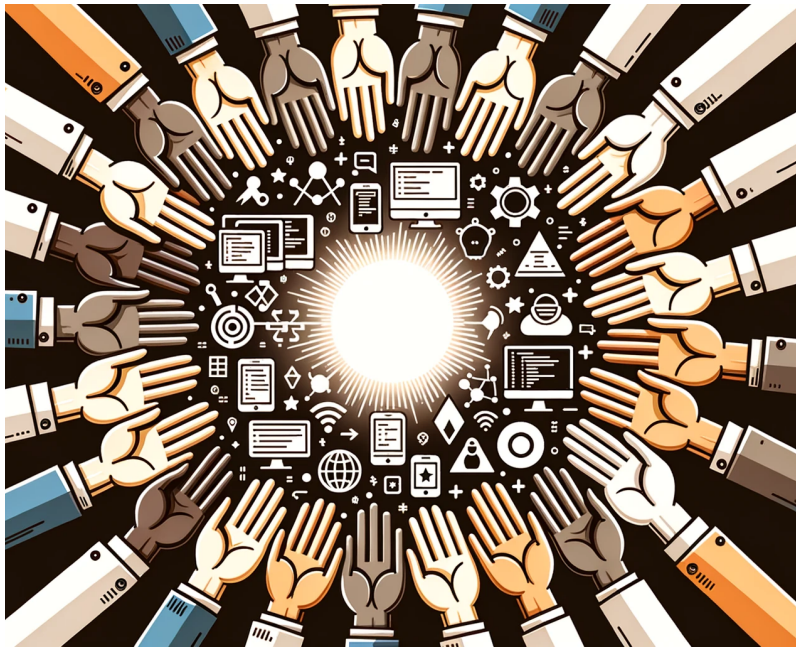
- Establish your own Repository with CI, Documentation, etc.
- Maintain stable, clean code.
- Work collaboratively on software development.
- Be able to contribute to an Open-Source Project.



The Importance of Collaboration

Teamwork is Dreamwork

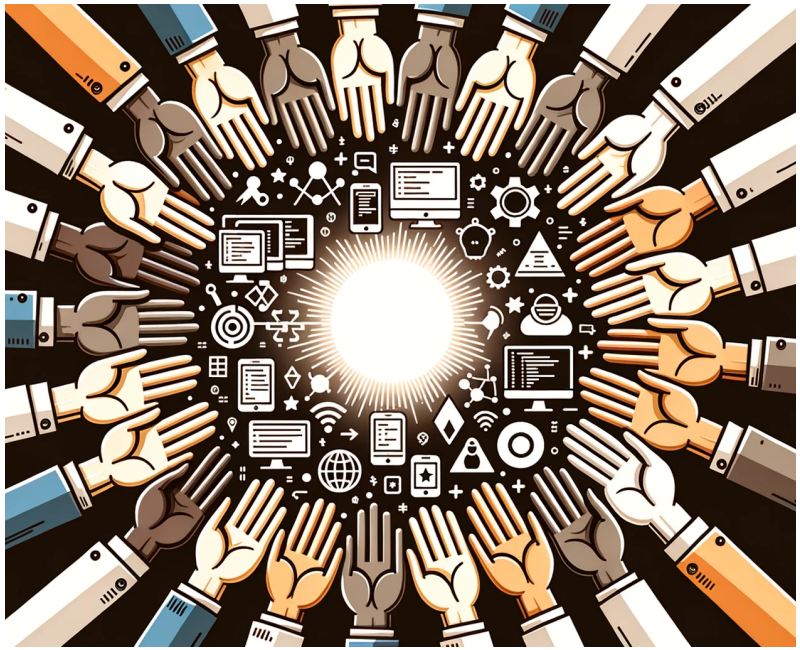
- **Diverse Skill Sets**: No single developer possesses expertise in every aspect of software development. Collaborative efforts ensure that a team with varied skills can address different facets of a project, from front-end design to back-end logic, ensuring comprehensive development.
- **Accelerated Problem-Solving**: Challenges are inevitable in software development. A collaborative team can brainstorm, debate, and ideate solutions faster than an individual. This collective problem-solving often leads to more innovative and efficient solutions.
- **Continuous Feedback Loop**: Collaboration fosters a culture of continuous feedback. Regular code reviews, paired programming, and open discussions ensure that errors are caught early, and best practices are consistently upheld.



The Importance of Collaboration

Teamwork is Dreamwork

- **Shared Responsibility**: A collaborative environment distributes the responsibility of the project. This not only reduces the pressure on individual team members but also ensures accountability and commitment towards the project's success.
- **Adaptability & Flexibility**: In a collaborative setting, teams are better equipped to adapt to changes, be it in project requirements, technologies, or methodologies. The collective knowledge ensures a smoother transition and quicker adaptation.
- **Knowledge Transfer & Skill Enhancement**: Collaboration is a learning experience. Developers can share knowledge, introduce peers to new tools or methodologies, and elevate the overall skill set of the team.



Overview of Version Control Systems

Git, Subversion, …

- What is VCS?
  - A tool that tracks changes in files, often used in codebases.
- Significance of VCS:
  - History: Records every modification, allowing rollbacks.
  - Collaboration: Enables multiple developers to work simultaneously.
  - Accountability: Tracks who made what change and when.
  - Backup: Safeguards code against accidental deletions or errors.
  - Branching: Allows development of features in isolation.
- VCS is foundational for collaborative, error-free, and efficient software development.

An Introduction to Git

And it’s Role in Facilitating Team-Based Development.

- What is Git?
  - A distributed version control system widely used in software development.
  - Enables software to be developed locally (on a developer's machine) or remote (on a server or hosted platforms like GitHub), facilitating collaboration and version control for projects of any size.
  - Provides control over contributors to code.

An Introduction to Git

Installing Git

- Visit the official Git website at git-scm.com
- Windows
  - Download the latest version for Windows
  - Run the downloaded .exe file and follow the installation prompts with default settings.
- macOS
  - Installed by default with Xcode. Otherwise, requires Homebrew or MacPorts.
- Linux
  - Installed using Linux distribution package manager. If on BioHPC, available as a module (module load git/v2.5.3)

Basic Git Commands and Workflows

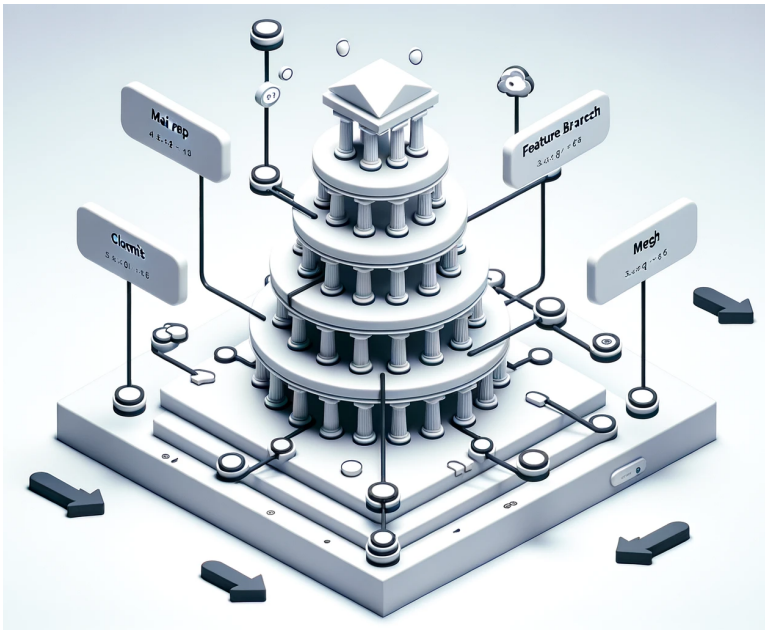
Git Up, Git Out, and Git Something

Repository

Definition: A repository, often abbreviated as "repo", is a storage location where all the files and revision history of a project reside.

It acts as the central hub for a project's codebase, allowing developers to clone, pull, and push changes to and from it.

A repository contains the information that Git uses to track the progression of a project.



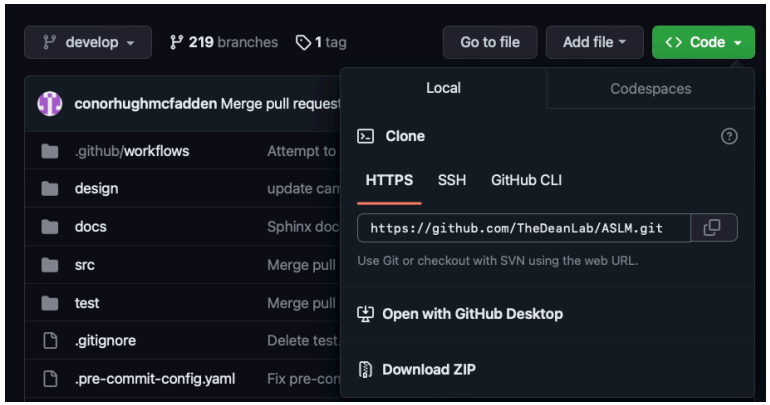
Basic Git Commands and Workflows

Cloning a Repo

Cloning: Downloads a repository onto your machine locally with all necessary information for tracking code development.

git clone - Clones a repository into a newly created directory, and creates remote-tracking branches for each branch in the cloned repository.

```
git clone [--template=<template-directory>]
          [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror]
          [-o <name>] [-b <name>] [-u <upload-pack>] [--reference
<repository>]
          [--dissociate] [--separate-git-dir <git-dir>]
          [--depth <depth>] [--[no-]single-branch] [--no-tags]
          [--recurse-submodules[=<pathspec>]] [--[no-]shallow-submodules]
          [--[no-]remote-submodules] [--jobs <n>] [--sparse] [--[no-
]reject-shallow]
          [--filter=<filter> [--also-filter-submodules]] [--] <repository>
          [<directory>]
```



https://git-scm.com/docs/git-clone

Basic Git Commands and Workflows

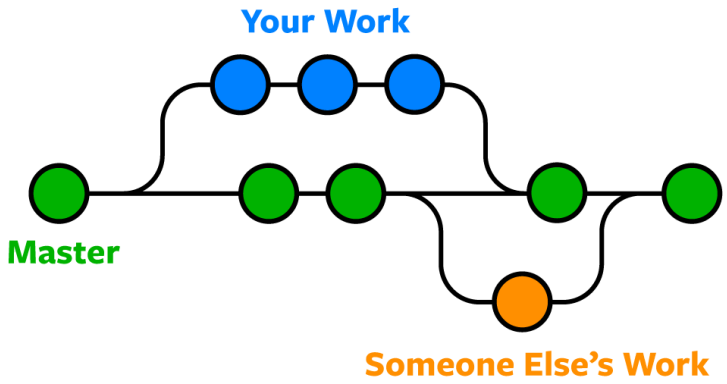
Isolating Development

Branch

Definition: A branch is a lightweight movable pointer that points to a specific commit.

It represents an independent line of development, allowing you to isolate work on different features, bug fixes, or experiments without affecting the main or other branches.

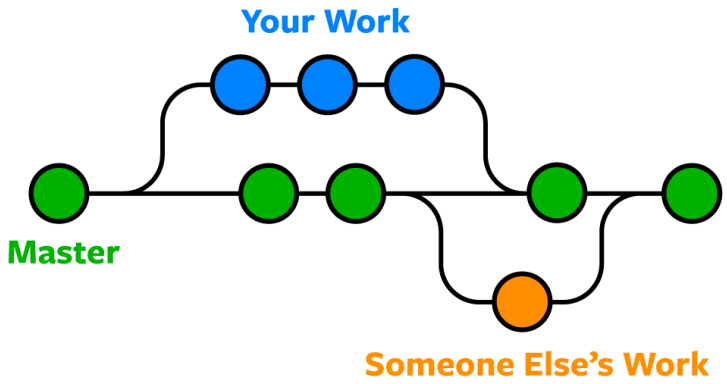
This facilitates parallel development, where multiple features or fixes can be developed simultaneously without interfering with each other.



Basic Git Commands and Workflows

Isolating Development

Main Branch: By default, every Git repository starts with a branch called "master" or, more recently, "main". This is the primary branch where stable, production-ready code resides.



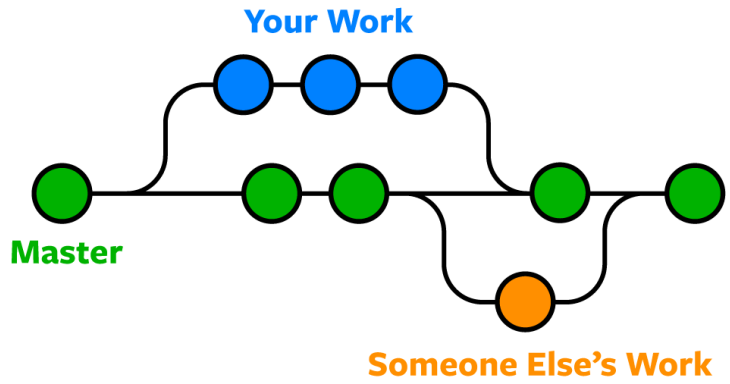
Basic Git Commands and Workflows

Creating a New Branch

Creation: Creating a new branch does not create a new copy of the repository. Instead, it's a lightweight pointer to a commit. This makes branching in Git very fast and efficient.

git branch - List, create, or delete branches.

```
git branch [--color[=<when>] | --no-color] [--show-current]
        [-v [--abbrev=<n> | --no-abbrev]]
        [--column[=<options>] | --no-column] [--sort=<key>]
        [--merged [<commit>]] [--no-merged [<commit>]]
        [--contains [<commit>]] [--no-contains [<commit>]]
        [--points-at <object>] [--format=<format>]
        [(-r | --remotes) | (-a | --all)]
        [--list] [<pattern>…]
git branch [--track[=(direct|inherit)] | --no-track] [-f]
        [--recurse-submodules] <branchname> [<start-point>]
git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]
git branch --unset-upstream [<branchname>]
git branch (-m | -M) [<oldbranch>] <newbranch>
git branch (-c | -C) [<oldbranch>] <newbranch>
git branch (-d | -D) [-r] <branchname>…
git branch --edit-description [<branchname>]
```



https://git-scm.com/docs/git-branch

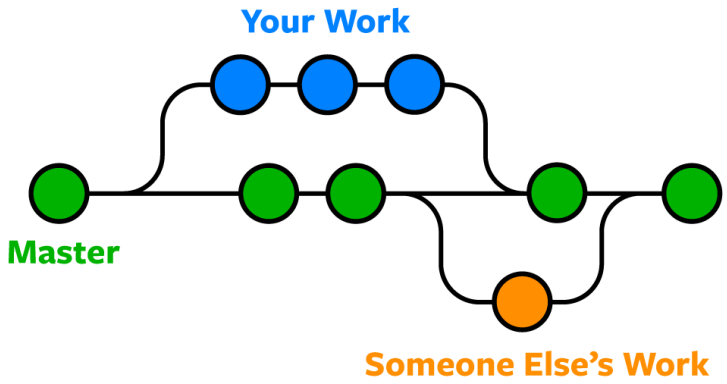
Basic Git Commands and Workflows

Switching to a Branch

Switching: You can easily switch between branches using the git checkout command (or git switch in newer versions of Git).

git checkout - Switch branches.

```
git checkout [-q] [-f] [-m] [<branch>]
git checkout [-q] [-f] [-m] --detach [<branch>]
git checkout [-q] [-f] [-m] [--detach] <commit>
git checkout [-q] [-f] [-m] [[-b|-B|--orphan] <new-branch>] [<start-
point>]
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] [--]
<pathspec>…
git checkout [-f|--ours|--theirs|-m|--conflict=<style>] [<tree-ish>] --
pathspec-from-file=<file> [--pathspec-file-nul]
git checkout (-p|--patch) [<tree-ish>] [--] [<pathspec>…]
```



https://git-scm.com/docs/git-checkout

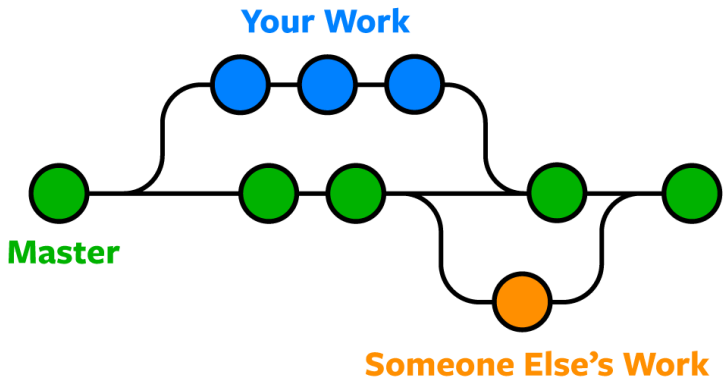
Basic Git Commands and Workflows

Merging a Branch

Merging: Once work on a branch is complete, it can be "merged" back into another branch, usually the main branch. This integrates the changes made in the branch into the main line of development.

git merge - Join two or more development histories together.

```
git merge [-n] [--stat] [--no-commit] [--squash] [--[no-]edit]
        [--no-verify] [-s <strategy>] [-X <strategy-option>] [-S[<keyid>]]
        [--[no-]allow-unrelated-histories]
        [--[no-]rerere-autoupdate] [-m <msg>] [-F <file>]
        [--into-name <branch>] [<commit>…]
git merge (--continue | --abort | --quit)
```



https://git-scm.com/docs/git-merge

Basic Git Commands and Workflows

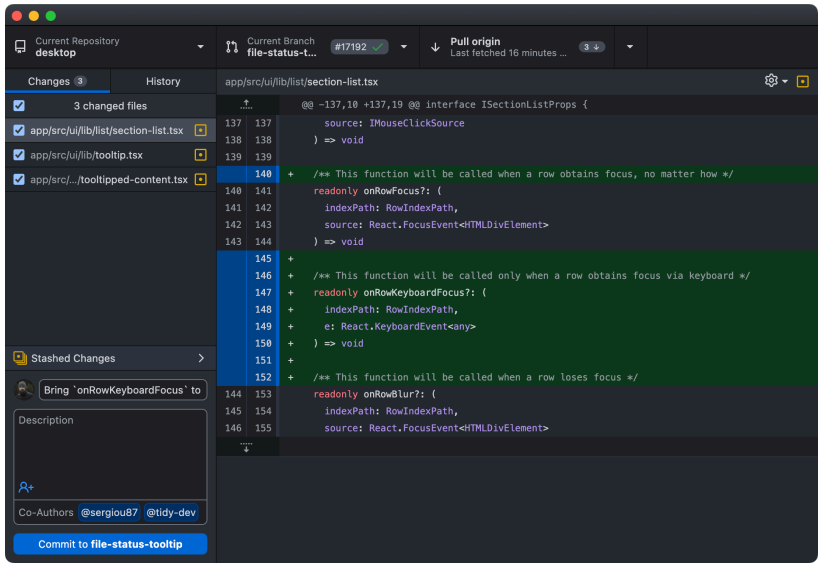
Challenges with Git

- Complex Commands: Git's command-line interface can be intimidating, especially for beginners. Remembering and correctly executing multiple commands can lead to errors.
- Merge Conflicts: Resolving conflicts when multiple developers make changes to the same section of code can be tedious and confusing.
- Visualizing History: The command-line doesn't provide an intuitive way to visualize the commit history, branches, and their relationships.
- Staging Changes: Deciding what changes to stage for a commit can be cumbersome, especially in large codebases.
- Switching Context: Frequent switching between branches and keeping track of which branch you're on can be error-prone.

Basic Git Commands and Workflows

Cheat to Compete with GitHub Desktop

- User-Friendly Interface: GitHub Desktop provides a graphical user interface, making it more intuitive and accessible for those not comfortable with the command-line.
- Easy Conflict Resolution: The tool visually presents merge conflicts and offers guidance on resolving them, streamlining the process.
- Visual History: Users can easily see the commit history, changes made, and the structure of branches, aiding understanding and navigation.
- Drag-and-Drop Staging: Users can quickly select changes for staging by dragging and dropping, making the process efficient.
- Branch Management: Switching between branches, creating new branches, and merging becomes a couple-of-clicks operation, reducing the chances of errors.



https://desktop.github.com/

Basic Git Commands and Workflows

Activity #1

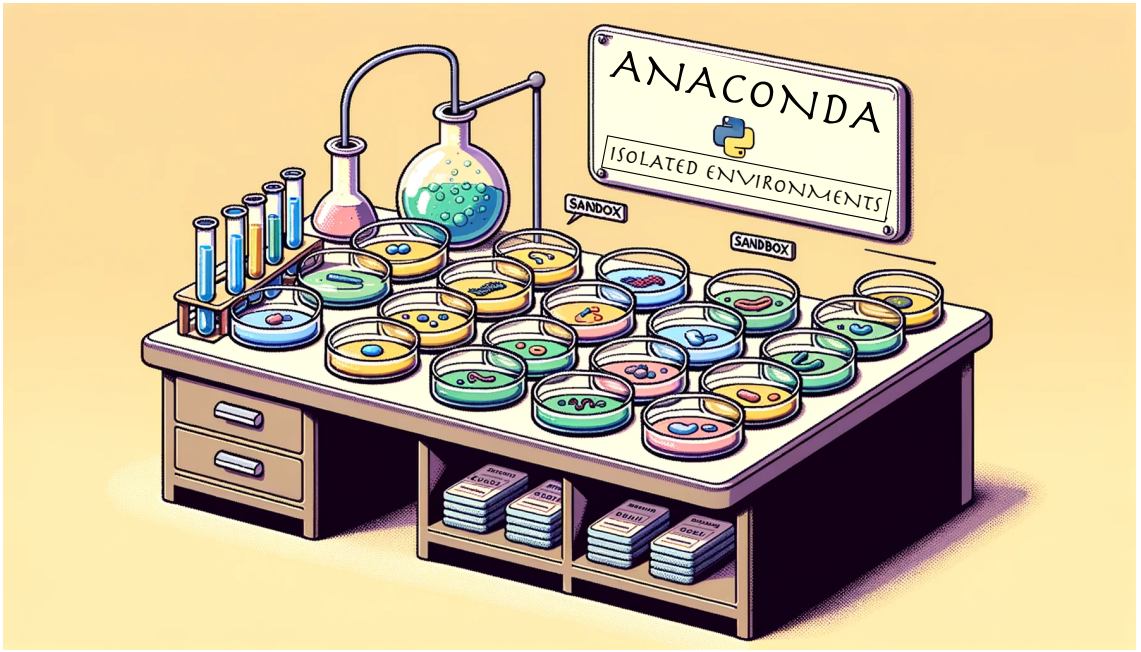
- Install Git (if this is Xcode, it could be very slow). Should make this a pre-requisite possibly? Ot
- Create a GitHub account.
- Download GitHub Desktop
- Clone the class repo: https://github.com/TheDeanLab/CI2023
- Using command-line and the GitHub Desktop, create a branch, and switch between main and the branch you created.

Intermission

Environment Management with Anaconda

Creating Reproducible Development Environments

- An environment is like a sandbox or petri dish, ensuring that the software setup for one project doesn't interfere with another.
- This enables a stable and reproducible development space.



Environment Management with Anaconda

Creating Reproducible Development Environments

- Version Inconsistencies: Python libraries and tools are constantly evolving. Different projects might require different versions of the same library, leading to potential conflicts and unexpected behavior.
- System-wide Installation Risks: Installing libraries system-wide can lead to "dependency hell", where upgrades for one project break another.
- Reproducibility: For scientific computing and data analysis tasks, it's crucial to reproduce results. This is impossible without consistent environment setups, especially when sharing work with peers or publishing results.
- Ease of Sharing: With a well-managed environment, developers can easily share their projects, ensuring that others can run their code without stumbling upon missing dependencies or version issues.
- Isolation: Keeping project environments separate ensures that specific dependencies or version requirements of one project don't interfere with another, leading to cleaner and more stable development.

Environment Management with Anaconda

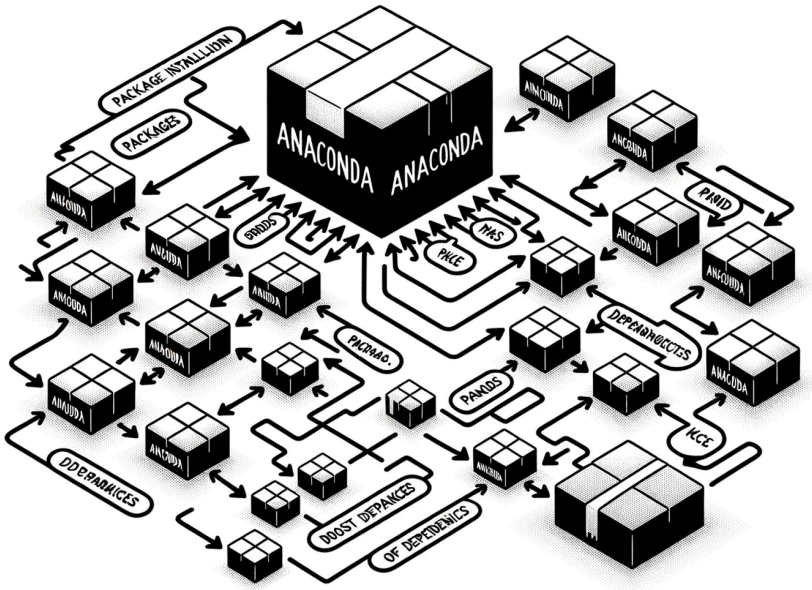
Miniconda vs Anaconda

- Size and Content: Anaconda is a large distribution that comes pre-loaded with over 1500 packages tailored for scientific computing, data science, and machine learning. Miniconda, on the other hand, is a minimalistic distribution, containing only the package manager (conda) and a minimal set of dependencies.
- Flexibility vs. Convenience: While Anaconda provides an out-of-the-box solution with a wide array of pre-installed packages, Miniconda offers flexibility by allowing users to install only the packages they need, helping to keep the environment lightweight.
- Installation Size: Due to its bundled packages, Anaconda requires more disk space upon installation compared to Miniconda.
- Use Cases:
    - Miniconda is ideal for users who are conscious about disk space, or who prefer to have more control over their environment setup.
    - Anaconda is suited for those who want a comprehensive package suite without the need to manually install popular data science tools.

Environment Management with Anaconda

What is a Package?

- Software Collection: A package in Anaconda is a bundled collection of software tools, libraries, and dependencies that function together to achieve a specific task or set of tasks.
- Version Management: Each package has specific versioning, allowing users to install, update, or rollback to particular versions as needed, ensuring compatibility and stability in projects.
- Dependency Handling: When a package is installed in Anaconda, the system automatically manages and installs any required dependencies, ensuring seamless functionality and reducing manual setup efforts.



Environment Management with Anaconda

Creating a New Environment

- Create a new environment:

  conda create --name EnvironmentName

- Create a new environment with specific Python version

  conda create --name EnvironmentName python=3.10

- Create a new environment from a YAML or text file

  conda create --name EnvironmentName file=package_contents.yml

  conda create --name EnvironmentName file=package_contents.txt

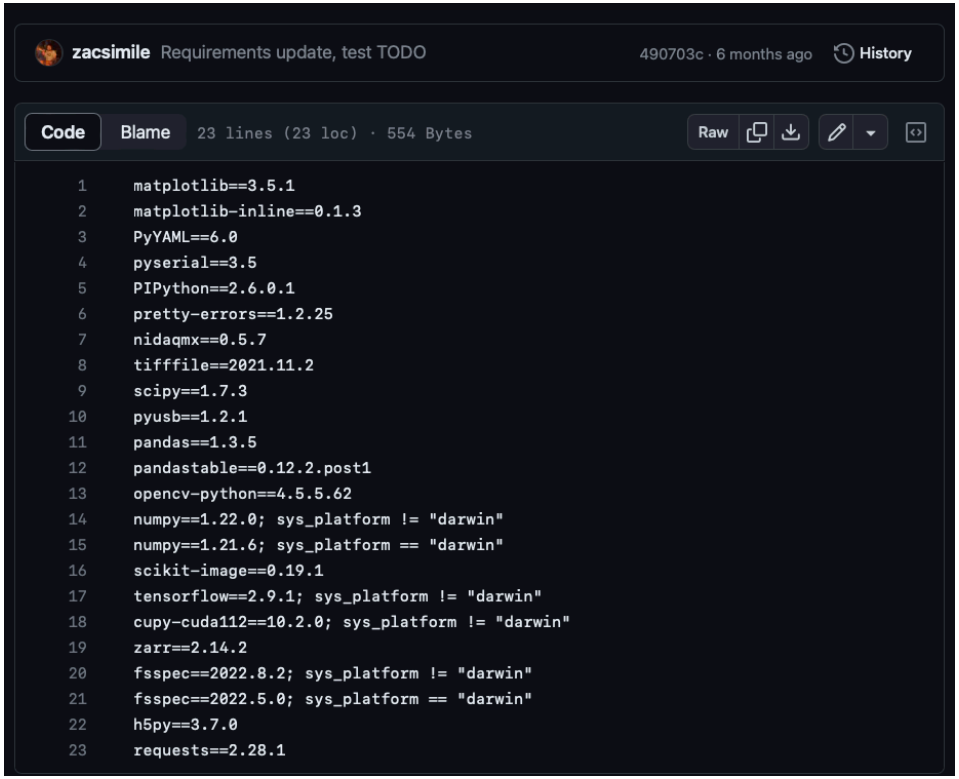
Environment Management with Anaconda

Creating a New Environment from a Text File

- Each package, and all of its dependencies, explicitly imported from a package manager (pip, conda, etc.)*
- Version Control

  (e.g., pyserial==3.5)
- Platform Control

  (e.g., sys_platform == “darwin”)



*Note: Do not mix package managers. Environment can become unstable.

Environment Management with Anaconda

What is a YAML File?

YAML files, typically with the .yaml or .yml extension, provide a balance between structure and readability, making them a popular choice for various configuration and data representation tasks.

- Human-Readable Format: Designed to be easily readable by humans and is often used for configuration files.
- Hierarchical Structure: Uses indentation to represent hierarchical data structures, such as lists and dictionaries.
- Key-Value Pairs: Data is typically represented using key-value pairs, making it similar to Python dictionaries.
- Multiline Strings: Supports multiline strings, making it convenient to represent longer text.
- Comments: Supports comments using the # symbol, allowing for annotations within the file.
- Widespread Usage: It's commonly used in various applications, including configuration for CI/CD tools, Kubernetes configurations, and defining infrastructure as code.

Environment Management with Anaconda

Creating a New Environment from a YAML File

zacsimile Requirements update, test TODO 490703c · 6 months ago History

Code Blame 23 lines (23 loc) · 554 Bytes Raw

```
matplotlib==3.5.1
matplotlib-inline==0.1.3
PyYAML==6.0
pyserial==3.5
PIPython==2.6.0.1
pretty-errors==1.2.25
nidaqmx==0.5.7
tifffile==2021.11.2
scipy==1.7.3
pyusb==1.2.1
pandas==1.3.5
pandastable==0.12.2.post1
opencv-python==4.5.5.62
numpy==1.22.0; sys_platform != "darwin"
numpy==1.21.6; sys_platform == "darwin"
scikit-image==0.19.1
tensorflow==2.9.1; sys_platform != "darwin"
cupy-cuda112==10.2.0; sys_platform != "darwin"
zarr==2.14.2
fsspec==2022.8.2; sys_platform != "darwin"
fsspec==2022.5.0; sys_platform == "darwin"
h5py==3.7.0
requests==2.28.1
```

```yaml
name: ASLM
dependencies:
  - matplotlib==3.5.1
  - matplotlib-inline==0.1.3
  - PyYAML==6.0
  - pyserial==3.5
  - PIPython==2.6.0.1
  - pretty-errors==1.2.25
  - nidaqmx==0.5.7
  - tifffile==2021.11.2
  - scipy==1.7.3
  - pyusb==1.2.1
  - pandas==1.3.5
  - pandastable==0.12.2.post1
  - opencv-python==4.5.5.62
  - numpy:
      - numpy==1.22.0; sys_platform != "darwin"
      - numpy==1.21.6; sys_platform == "darwin"
  - scikit-image==0.19.1
  - tensorflow:
      - tensorflow==2.9.1; sys_platform != "darwin"
      - cupy-cuda112==10.2.0; sys_platform == "darwin"
  - zarr==2.14.2
  - fsspec:
      - fsspec==2022.8.2; sys_platform != "darwin"
      - fsspec==2022.5.0; sys_platform == "darwin"
  - h5py==3.7.0
  - requests==2.28.1
```

Environment Management with Anaconda

Activating Your New Environment

- List all Environments:

  conda env list

- Activate an Environment:

  conda activate EnvironmentName

- List Environment Packages:

  conda list

```
Last login: Tue Oct 17 14:02:56 on ttys000
(base) S155475@SW567797 ~ % conda env list
# conda environments:
#
base                  *  /opt/miniconda3
ASLM                     /opt/miniconda3/envs/ASLM
archicellago             /opt/miniconda3/envs/archicellago

(base) S155475@SW567797 ~ % conda activate ASLM
(ASLM) S155475@SW567797 ~ % conda list
# packages in environment at /opt/miniconda3/envs/ASLM:
#
# Name                    Version                   Build  Channel
alabaster                 0.7.12                  pypi_0    pypi
anyio                     3.6.2                   pypi_0    pypi
appnope                   0.1.3                   pypi_0    pypi
argon2-cffi               21.3.0                  pypi_0    pypi
argon2-cffi-bindings      21.2.0                  pypi_0    pypi
arrow                     1.2.3                   pypi_0    pypi
asciitree                 0.3.3                   pypi_0    pypi
aslm                      0.0.1                    dev_0    <develop>
asttokens                 2.2.1                   pypi_0    pypi
attrs                     22.1.0                  pypi_0    pypi
babel                     2.11.0                  pypi_0    pypi
backcall                  0.2.0                   pypi_0    pypi
beautifulsoup4            4.11.1                  pypi_0    pypi
black                     22.12.0                 pypi_0    pypi
bleach                    5.0.1                   pypi_0    pypi
ca-certificates           2022.10.11           hca03da5_0
certifi                   2022.9.24         py39hca03da5_0
cffi                      1.15.1                  pypi_0    pypi
cfgv                      3.3.1                   pypi_0    pypi
```

Environment Management with Anaconda

Environment Tips and Tricks

- Don’t mix and match Package Managers.
- Be judicious with your dependencies.
- Be explicit with your dependencies.

Environment Management with Anaconda

Activity #2

- Install Miniconda
- Create an environment called ASLM with python 3.9.7
- Clone ASLM - https://github.com/TheDeanLab/ASLM
- Activate the environment, and install the ASLM repository using the provided text file.

Environment Management with Anaconda

Activity #3

- Create an environment called CI2023 with python 3.9.7
- Clone CI2023 - https://github.com/TheDeanLab/CI2023
- Activate the environment, and install the ASLM repository using the provided yaml file.
- Switch back and forth between the two environments.

Advanced Environment Management

What Approach is Best?

- Large number of package managers creates a fragmented ecosystem.
- To standardize Python packaging, the pyproject.toml file was created.
- Significant step forward in the standardization and enhancement of Python packaging tools.
- It offers several benefits over text and YAML files

Advanced Environment Management

Creating a New Environment from a pyproject.toml File

- Standardization: PEP 518 introduced pyproject.toml as a standardized configuration file for Python packaging, aiming to provide a single source of truth for package configurations. This unified approach reduces fragmentation in the packaging ecosystem.
- Extensibility: pyproject.toml is designed to be extensible. It can accommodate configurations for various tools like setuptools, black, mypy, and more, all in one place. This centralized configuration avoids the proliferation of various config files in a project's root directory.
- Build System Specifications: pyproject.toml allows package maintainers to specify which build system should be used and what the build dependencies are. This ensures that the right tools and versions are used during the build process.
- Clear Dependency Specification: While requirements.txt or YAML files list dependencies, pyproject.toml allows for a more detailed specification, including build dependencies, which are essential for reproducibility and consistent builds.
- Modern Tooling Compatibility: Modern packaging and build tools like poetry and flit natively use pyproject.toml, showcasing a move towards this standard in the Python community.
- Improved Isolation: With pyproject.toml, build dependencies can be isolated from the project's dependencies, ensuring that the build process doesn't affect the project's environment or vice versa.

Advanced Environment Management

Creating a New Environment from a pyproject.toml File

```
[build-system]
requires = ["setuptools", "wheel"]

[tool.setuptools]
name = "mypackage"
version = "0.1.0"
description = "A sample Python package"
author = "Your Name"
author_email = "your.email@example.com"
classifiers = [
    "License :: OSI Approved :: MIT License",
    "Programming Language :: Python :: 3",
    "Programming Language :: Python :: 3.8",
]
packages = ["mypackage"]

[options]
install_requires =
    [
        "numpy",
        "requests",
        "matplotlib"
    ]

[options.extras_require]
dev =
    [
        "pytest",
        "black",
        "mypy"
    ]
```

Advanced Environment Management

Activity #3

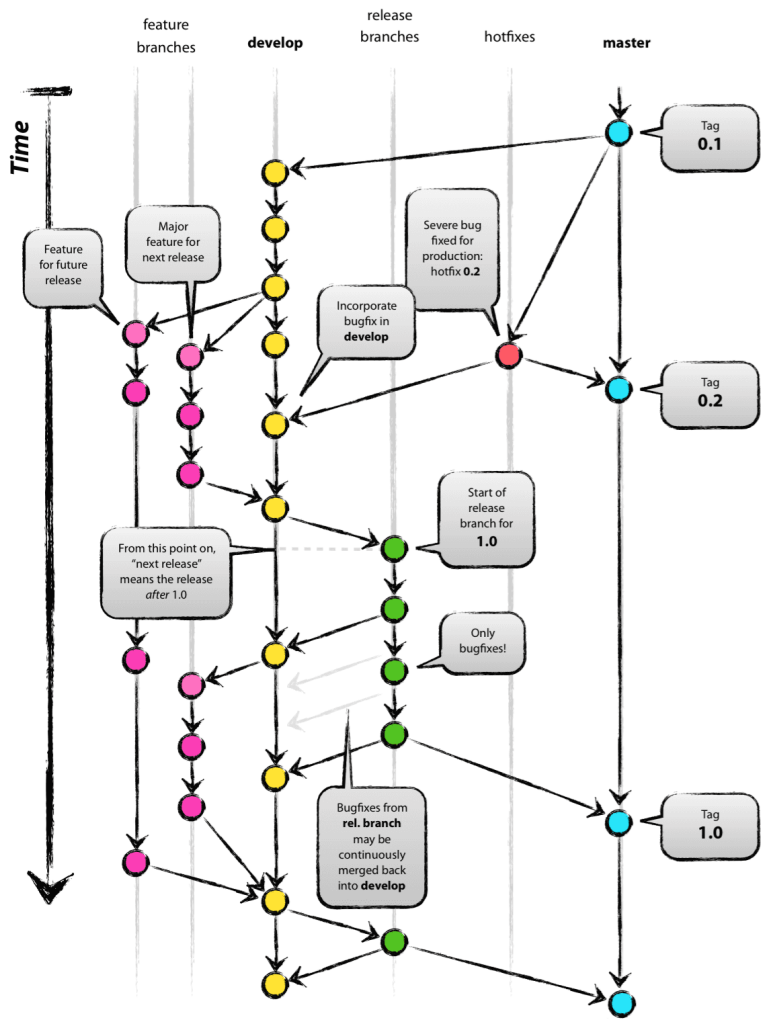
- Install Miniconda
- Create an environment called seaborn
- Clone Seaborn - https://github.com/mwaskom/seaborn
- Install the package with the pyproject.toml file.

Intermission

Git Essentials

When git push comes to shove

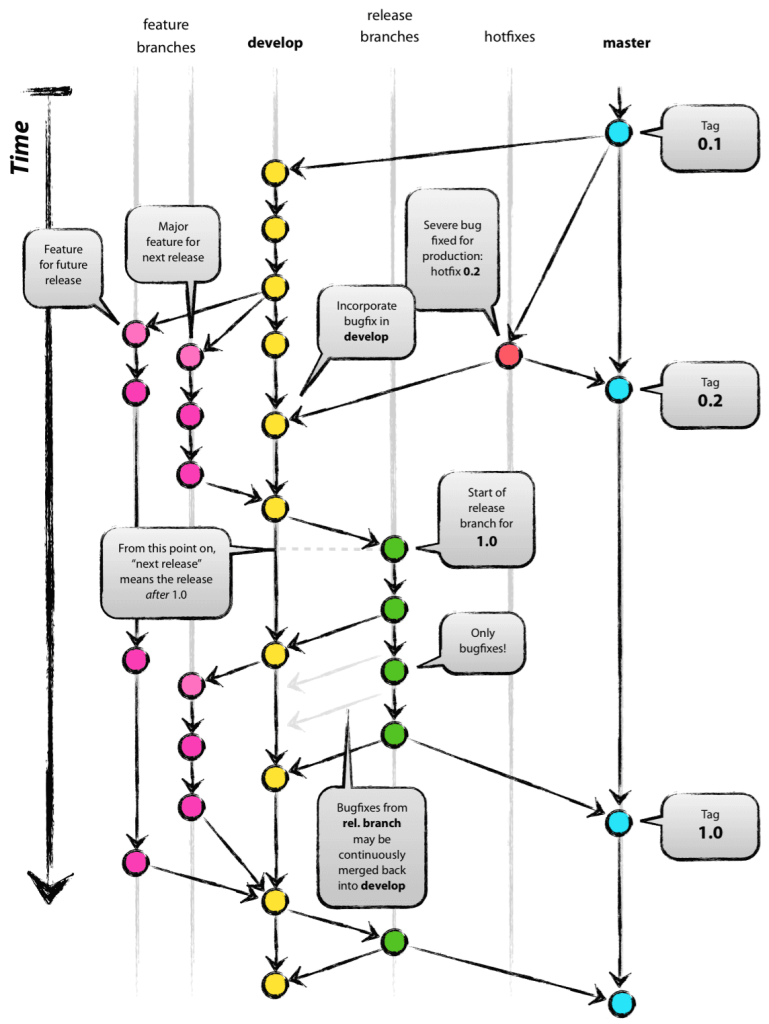
- Git Flow
  - A fixed branching model that defines different branches for features, releases, and hotfixes.
  - Provides a structured workflow that is especially useful for larger teams and projects.



Git Essentials

Feature Branching

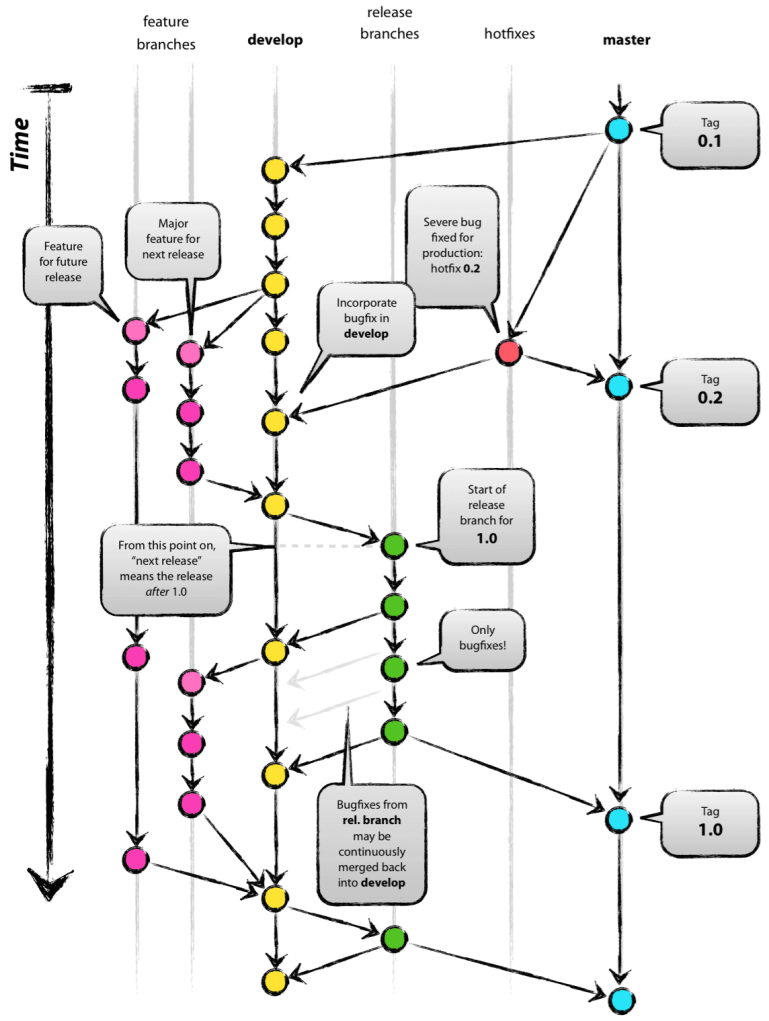
- Each new feature or bugfix is developed in a separate branch derived from the develop branch.
- Keeps the develop branch more stable.
- Allows multiple features to be developed in parallel without interference.



Git Essentials

Trunk Based Development

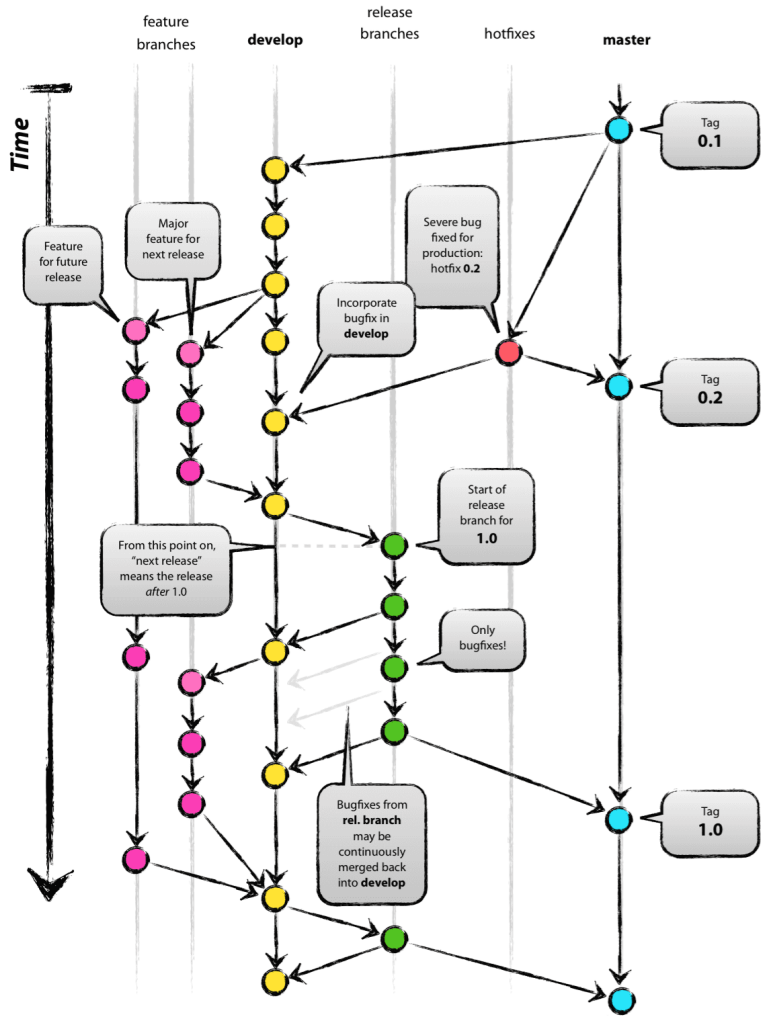
- Developers work in short-lived branches derived from develop, ensuring that code is integrated frequently.
- Reduces merge conflict complexity by promoting regular merges.
- Long-lived branches can lead to complex, hard-to-resolve merge conflicts.
- Ensures that code in the develop branch is always production-ready.



Git Essentials

Fear of Commitment.

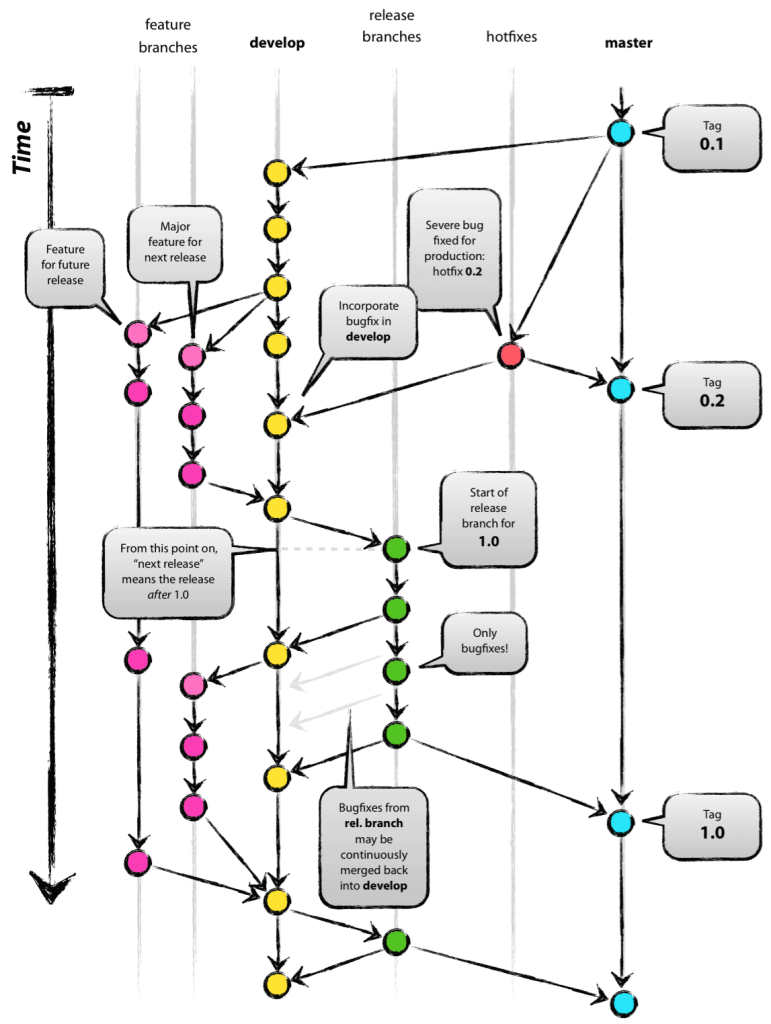
- Using structured, meaningful commit messages to make history more readable and to allow for automated changelog generation.



Git Essentials

Pull/Merge Requests

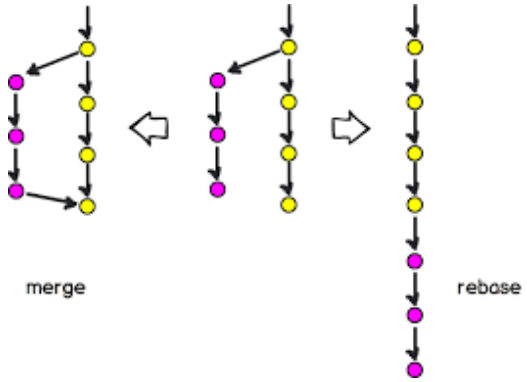
- Before merging a feature branch into the develop, a pull (or merge) request is created.
- Allows team members to review code, ensuring quality and consistency.
- Pull requests should preferably be manageable in size (~400 lines of code max), and accompanied with unit tests (more following).

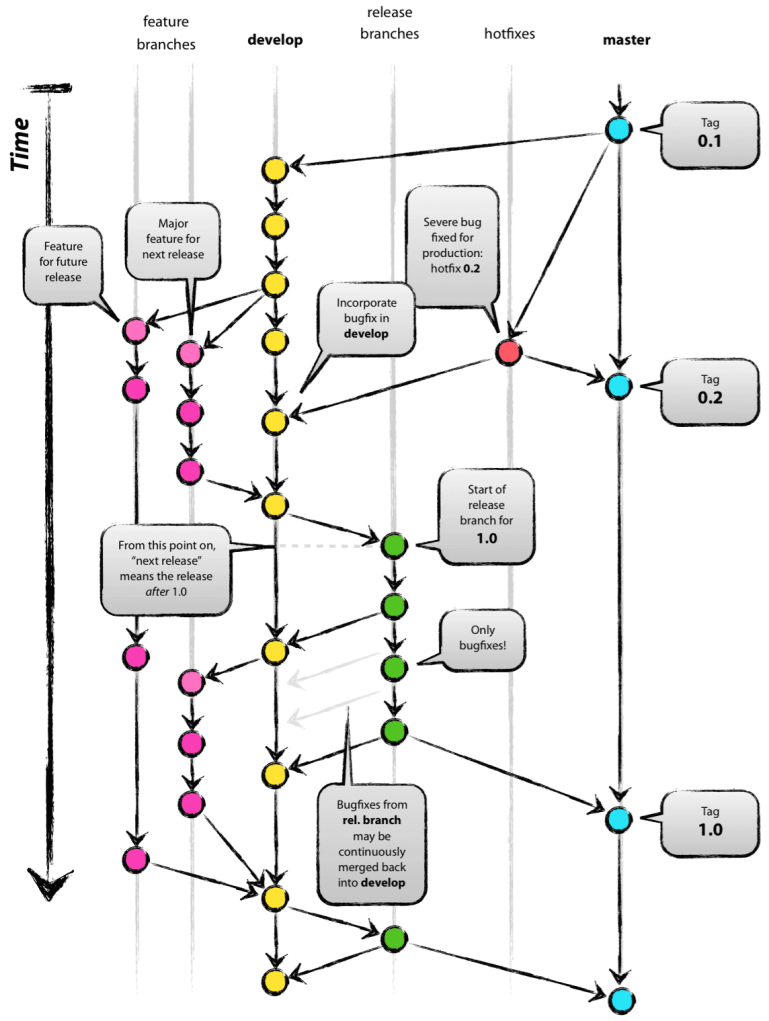


Git Essentials

Rebase vs Merge

- Instead of merging, use rebasing to apply feature branch changes on top of the main branch.
- Provides a cleaner, linear project history but requires a more cautious approach to avoid conflicts.



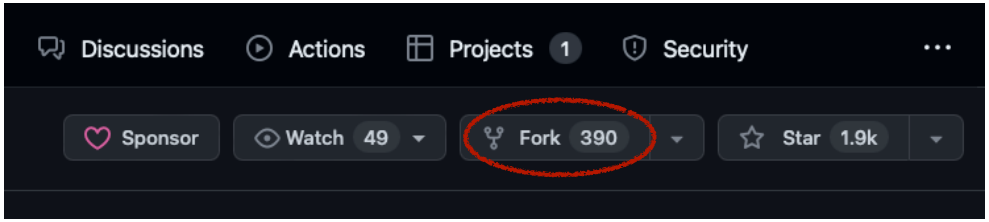


Git Essentials

Advanced Pull Requests

What if you don't have write access to a repository?

1. Create fork in GitHub and clone the repository.
2. Set upstream remote to help synchronize your fork with the original repository.
3. Create a branch.
4. Make changes to your branch.
5. Sync with upstream, update any confilcts if they arise.
6. Push changes to fork.
7. Navigate to fork on GitHub, and click "New pull request".




```
1.  git clone <repository_URL>

2. git remote add upstream <original_repository_URL>

3. git checkout -b <branch_name>

4A. git add .
4B. git commit -m "Describe the changes made"

5A. git fetch upstream
5B. git merge upstream/main

6. git push origin <branch_name>
```

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/creating-a-pull-request-from-a-fork

Git Essentials

Resolving Conflicts

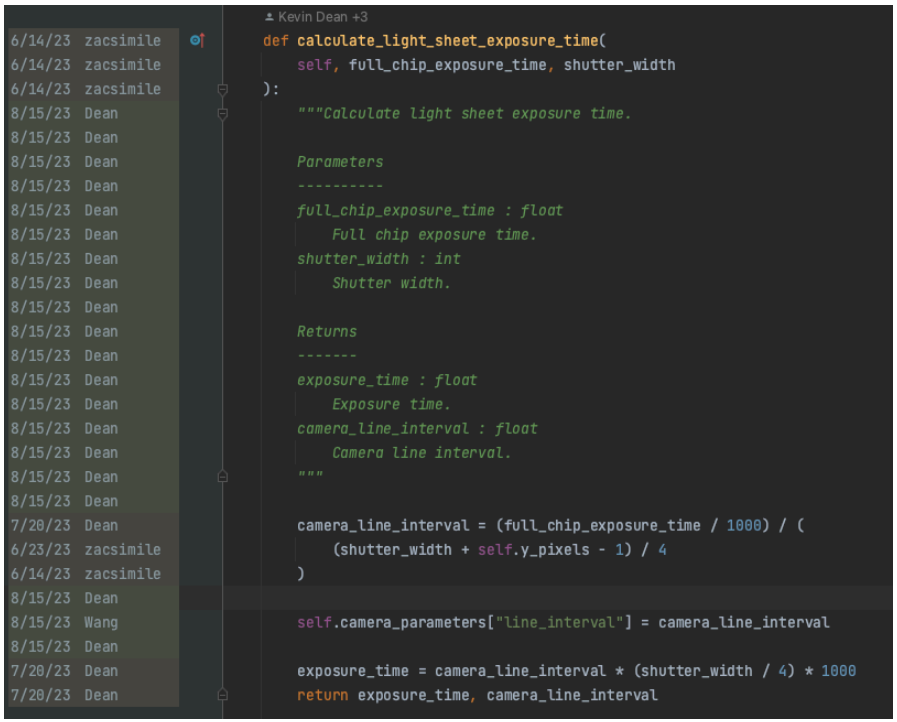
- Git will notify you of "merge conflicts" when attempting to merge or rebase.
- Within the conflicted files, Git uses conflict markers (<<<<<<<, =======, >>>>>>>) to indicate the conflicting sections.
- The content between <<<<<<< and ======= is your branch's changes, while the content between ======= and >>>>>>> is from the branch you're trying to merge or rebase with.
- Edit the file to resolve the conflict. This might mean choosing one change over the other, combining both changes, or even making a new change altogether.
- Once resolved, remove the conflict markers (<<<<<<<, =======, >>>>>>>).


```
 1 print('Hi Git')
 2
 3 print('Hi from another dev')
 4
 5 def hi_there():
 6     print("hi")
 7
 8
 9 def hi_there():
10 <<<<<<< HEAD
11     print("My local change")
12 =======
13     print("Update python function with merge conflict")
14 >>>>>>> origin/master
~
~
~
```

Git Essentials

Placing “Blame”

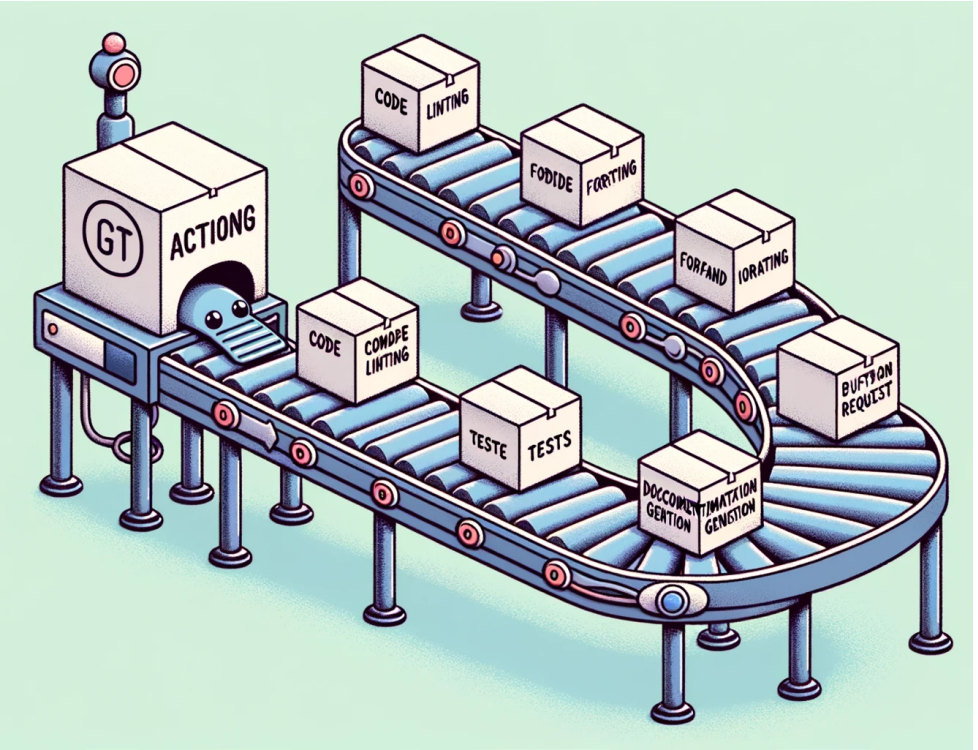
- *git blame* is a tool to trace changes in a file back to the commit that introduced them. Use it as a diagnostic tool to understand the history of specific lines of code, not to assign fault.
- When examining the output, consider the broader context of changes. A developer might have moved or reformatted code, so they appear as the "blamer," even if they didn't originally write the logic. Dive deeper into the commit message and associated discussions for a full understanding.
- Integrate with Tools and GUIs - While the command-line version of git blame is powerful, several graphical user interfaces and tools offer more intuitive visualizations, helping to quickly pinpoint when and why changes were made.



Git Essentials

Event Driven Tests

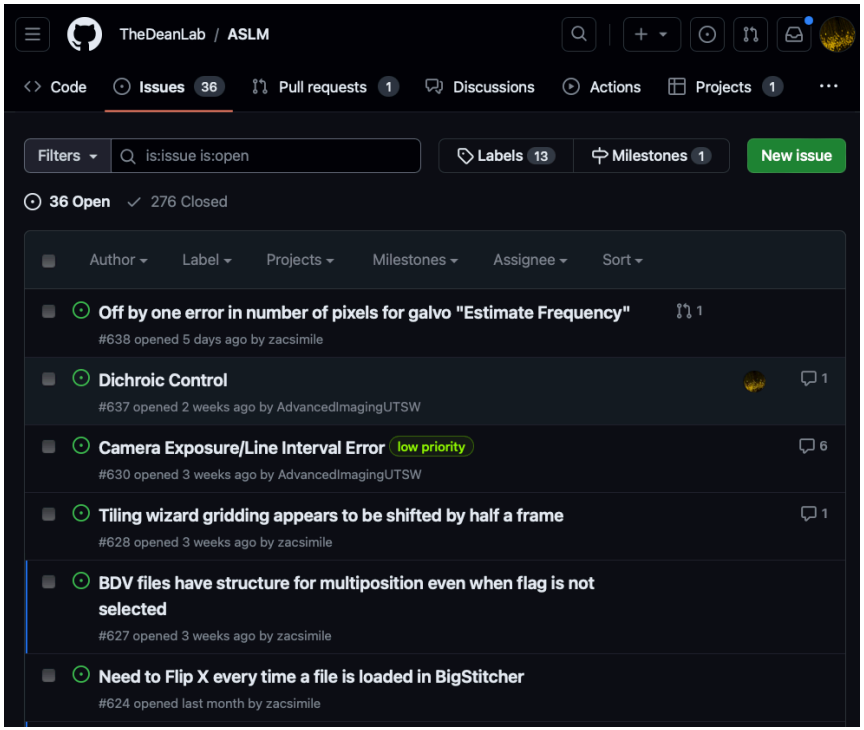
- Automatically building, testing, and deploying changes to ensure new commits don't break existing functionality and meet quality standards.
- Often triggered upon generation of a pull request.
- Actions can be multi-faceted, and include code reformatting, testing, documentation generation, etc.
- More to follow…



Git Essentials

Issue Tracking

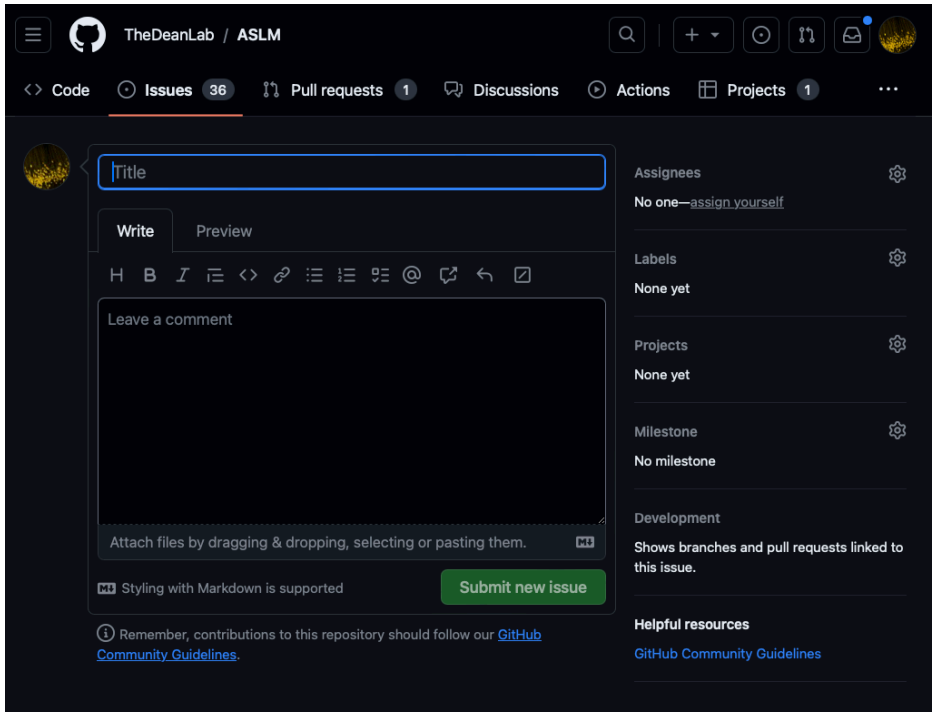
- GitHub provides integrated issue, branch, and PR tracking.
- Each issue assigned a number (#638) which can be used as a tag in commits, comments, etc.
- Can be designated with a label (e.g., low priority).
- Can be assigned to an individual.
- Linked PRs automatically



Git Essentials

Issue Creation

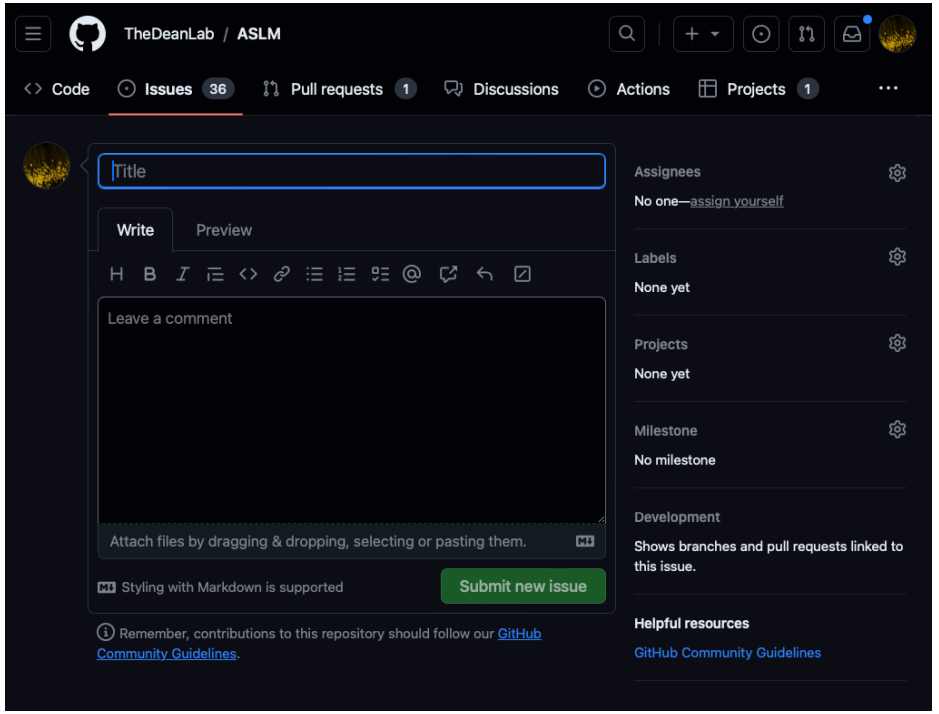
- Title - A concise and descriptive title that summarizes the issue.
- Description - A clear and detailed description of the issue. Explain what you expected to happen and what actually happened.
- Steps to Reproduce - Step-by-step guide on how to reproduce the issue. This helps maintainers replicate the problem on their end.
- Traceback - Detailed output from the console/terminal stating what the problem is.
- Environment Information - Python version (e.g., Python 3.8.5), Operating System and version (e.g., Ubuntu 20.04, Windows 10), version of the software or library causing the issue, any other relevant libraries or tools and their versions.
- Error Messages/Logs - Include any error messages, stack traces, or logs that are displayed when the issue occurs. Use code blocks or attach files if the logs are extensive.



Git Essentials

Issue Creation

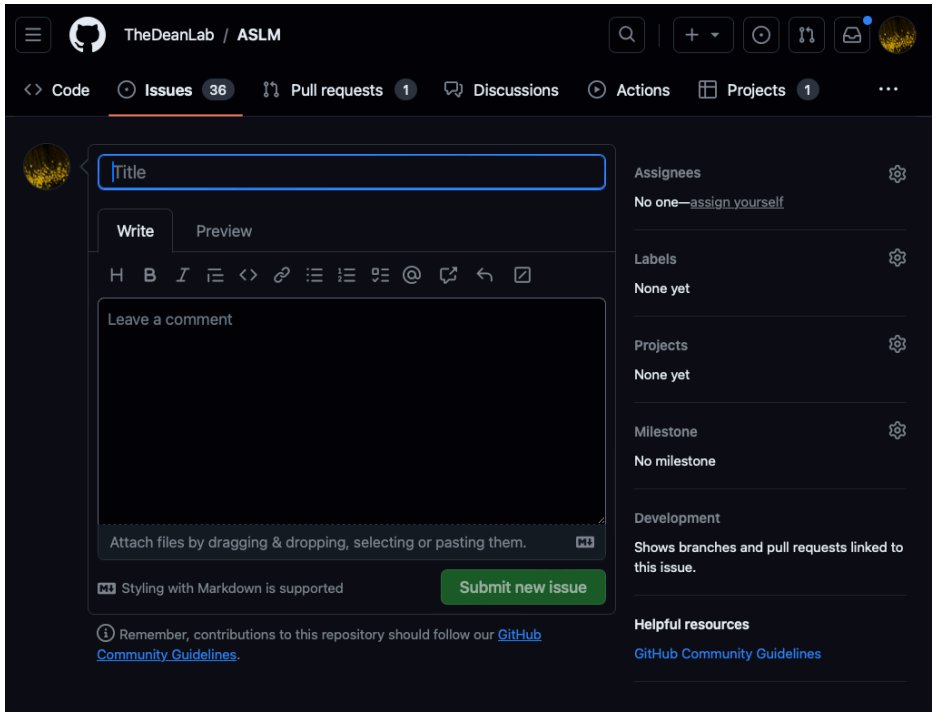
- Screenshots or Screen Recordings - Visual aids can be extremely helpful in understanding the nature of the problem, especially for UI/UX related issues.
- Severity and Impact - Indicate how critical the issue is (e.g., blocking, critical, minor) and its impact on the software's functionality.
- Possible Solutions or Workarounds - If you're aware of any solutions or temporary workarounds for the issue, share them. This can be helpful for both maintainers and other users facing the same problem.
- Code Samples - If applicable, provide minimal code samples that can help in reproducing the issue. Use proper formatting or link to a gist or repo.



Git Essentials

Issue Creation

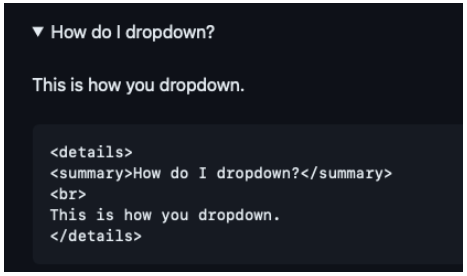
- Additional Context - Any other information that might be relevant, such as how often the issue occurs, whether it's sporadic or consistent, any patterns noticed, etc.
- Labels and Tags - If the repository has predefined labels (e.g., "bug", "enhancement", "documentation"), apply the relevant ones to your issue.
- Regression Information - Mention if the issue is a regression, i.e., a feature that used to work in a previous version but is broken in the current one. Indicate the last working version, if known.
- Providing comprehensive information when creating an issue not only helps maintainers diagnose and fix problems faster but also fosters a collaborative environment where community members can assist each other more effectively.

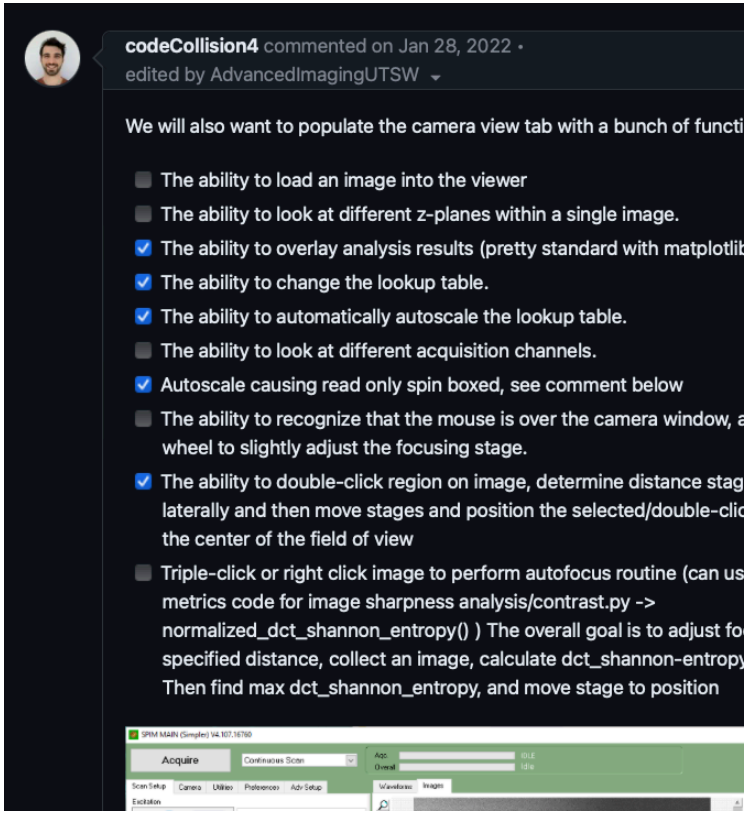


Git Essentials

Issue Formatting

- GitHub Issues support styling with Markdown.
- Can create checklists, headers, links, images, mention people/teams/issues/pull requests, quote code, create blocks, dropdowns, and more.



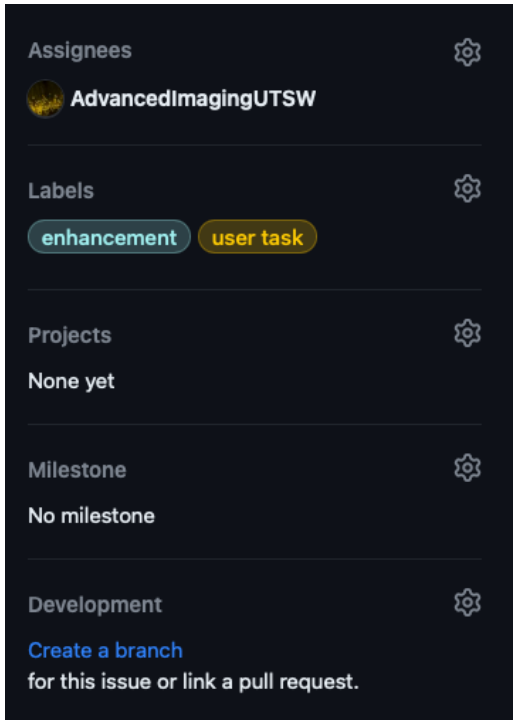


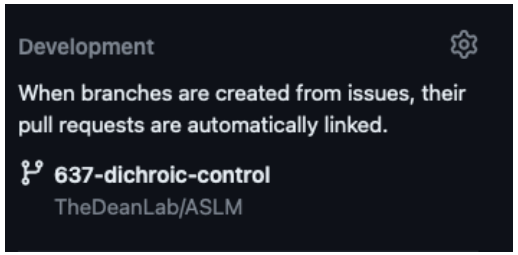
https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax

Git Essentials

Branch Linking

- By creating a branch that is directly linked to the issue, all progress on that issue is tracked.
- When a PR is approved, the issue is automatically closed.





Git Essentials

Activity #4

- Make your own repo on GitHub
- Create a pyproject.toml file.

Intermission

Repository Organizational Strategies

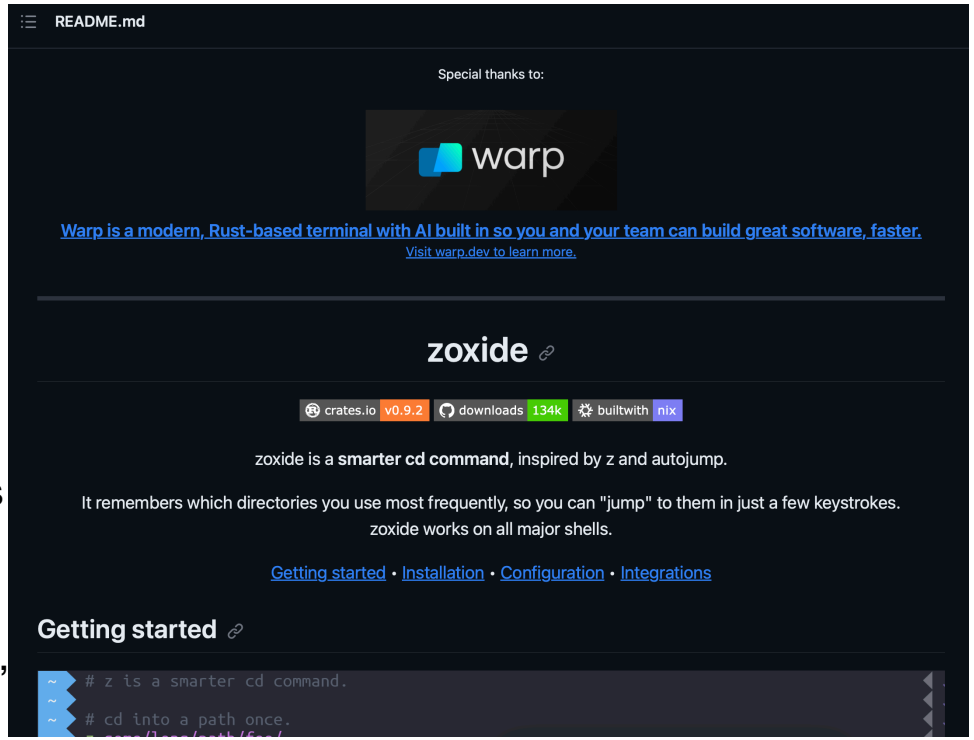
Fighting Entropy

- *Organizing a GitHub repository for Python software in a structured and consistent manner is crucial for clarity, collaboration, and maintainability.*
- Directory Structure:
  - Use a standard project layout. Common directories include:
  - src/: For the main source code.
  - tests/: For unit tests.
  - docs/: For documentation.
  - scripts/: For utility scripts and auxiliary code.
  - data/: For data files, if applicable.

Repository Organizational Strategies

Must Haves

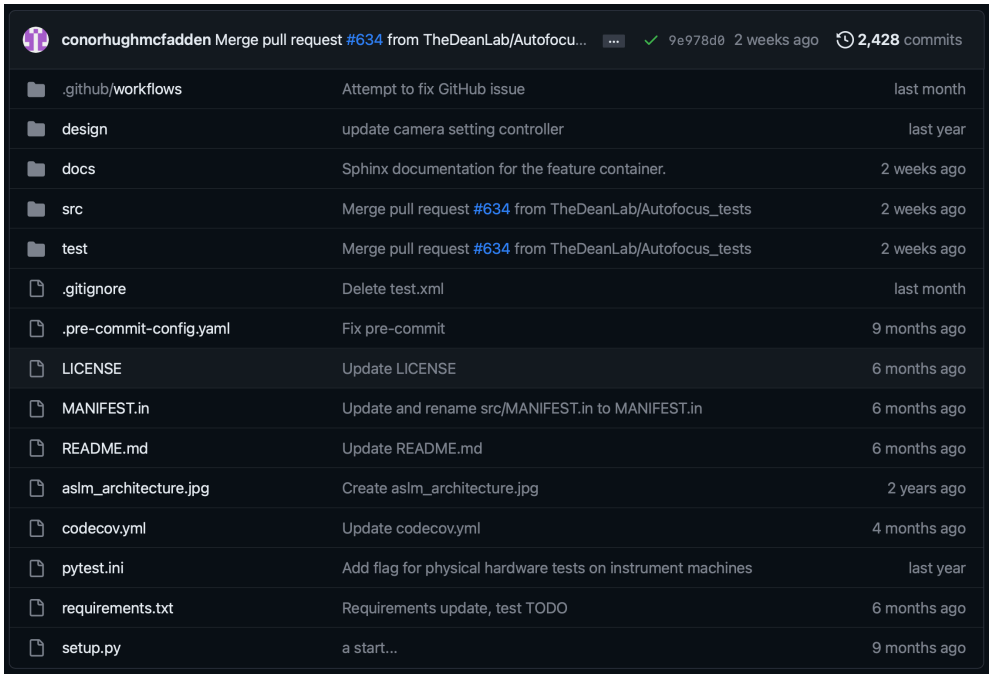
- README File - Always have a README.md at the root. It should include:
  - Project title and brief description.
  - Installation and usage instructions.
  - Contribution guidelines.
  - Licensing information.
- License - Include a LICENSE file that clearly states the licensing terms.
- Badges - Use badges in the README.md to quickly display project status, such as build status, test coverage, and package version.



Repository Organizational Strategies

Must Haves

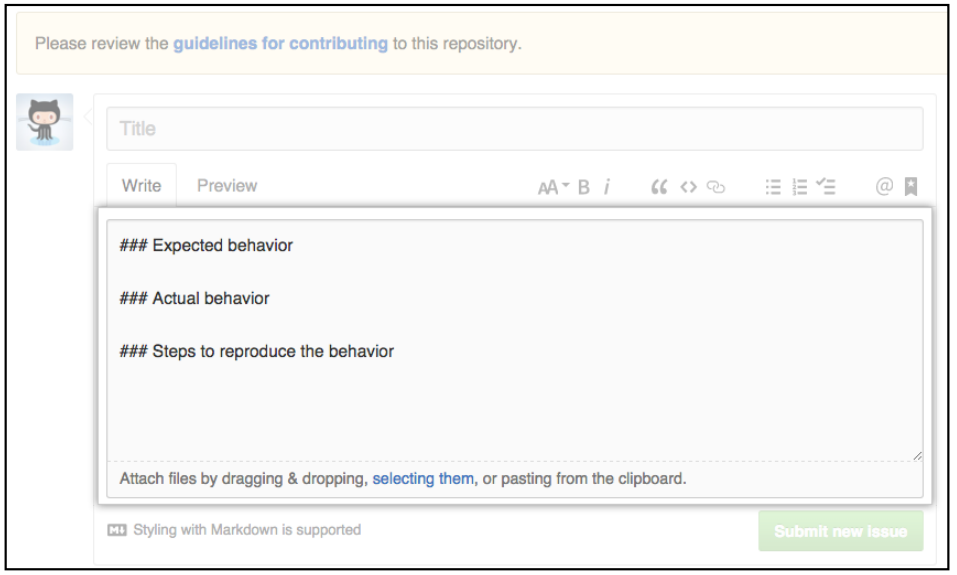
- Include a .gitignore - List files and directories that should be excluded from version control, such as compiled bytecode (*.pyc), virtual environment directories, IDE configuration files, etc.
- Requirements File - Use requirements.txt for specifying dependencies or consider using a pyproject.toml for more modern dependency management.
- docs - Use tools like Sphinx to generate documentation. Host it on platforms like Read the Docs for easy access.
- test - Write unit tests and possibly integrate coverage tools like coverage.py to ensure code quality.
- src - Code location.



Repository Organizational Strategies

Bonus Tools

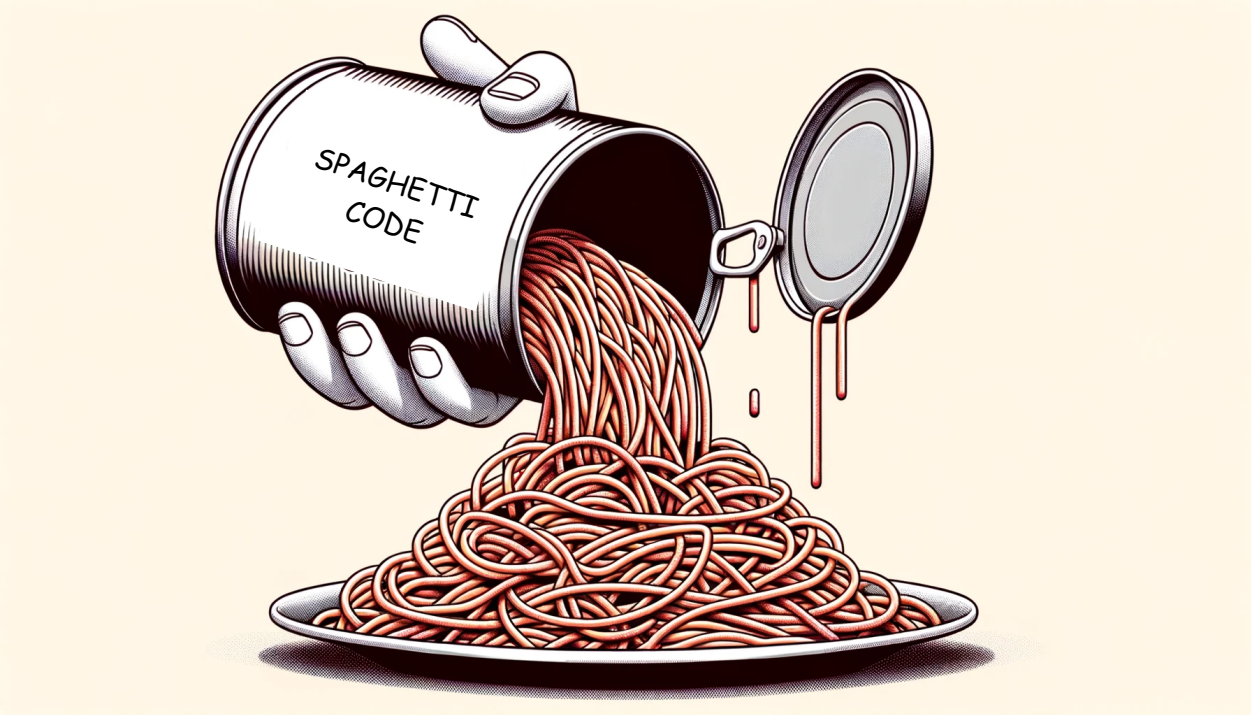
- Code of Conduct - Include a CODE_OF_CONDUCT.md to set community standards and ensure a welcoming and inclusive environment.
- Changelog - Maintain a CHANGELOG.md to track and document all the changes made in the software over time.
- Issue and PR Templates - Use issue and PR templates to ensure consistency and completeness in submissions. This can be done with .github/ISSUE_TEMPLATE and .github/PULL_REQUEST_TEMPLATE directories.
- Security - Use tools like Dependabot to automatically check for vulnerabilities in dependencies and suggest updates.
- Contribution Guidelines - A CONTRIBUTING.md file detailing how others can contribute, the process for submitting pull requests, and any coding standards.



Code Organizational Strategies

Fighting Entropy

Organizing code in Python is crucial for maintainability, scalability, and clarity.



Code Organizational Strategies

A Sisyphean Task

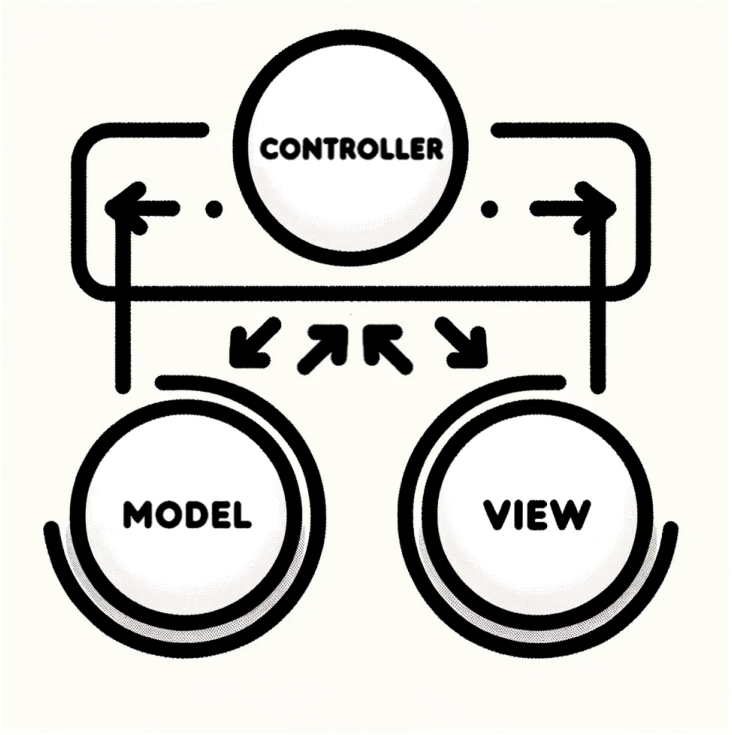
- Modularity - Divide your code into modules and packages. Each module should have a specific responsibility aligned with the MVC pattern.
- Naming Conventions - Use clear and descriptive naming conventions. For example: `data_source.py`, `file_management.py`, `data_visualization.py`.
- Code Reusability - Abstract out common functionalities into utility functions or base classes to avoid repetition and enhance reusability.

Model-View-Controller Architecture

A Guiding Framework

The Model-View-Controller (MVC) architecture is a design pattern that separates software applications into three interconnected components.

- The "Model" represents the data and logic, governing the rules and data manipulation.
- The "View" displays the data, handling the presentation and user interface elements.
- The "Controller" manages user input, interpreting it to update the model and view accordingly, acting as a bridge between the two.



Model-View-Controller Architecture

The View

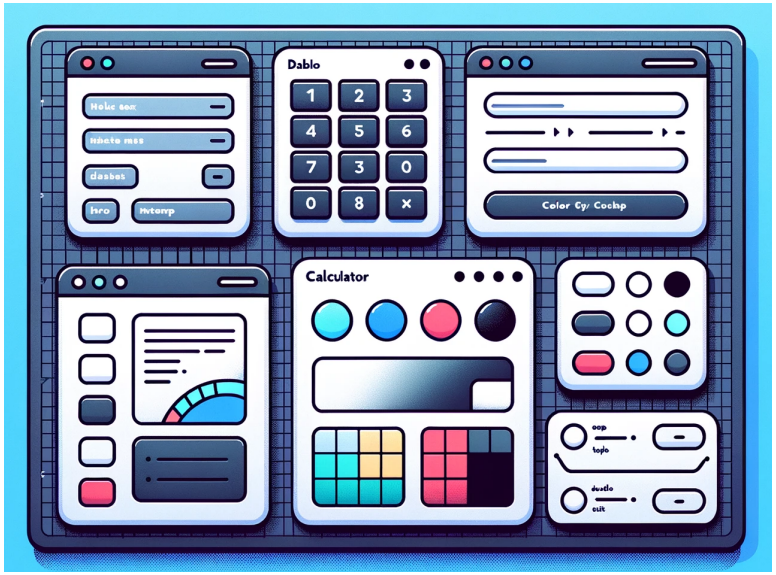
Python offers a variety of libraries and frameworks for creating Graphical User Interfaces (GUIs). Here are some of the most popular ones:

- Tkinter:
  - The standard GUI library for Python, bundled with most Python installations.
  - Offers a simple way to create windows, dialogs, buttons, and other GUI elements.
- PyQt and PySide:
  - Bindings for the Qt application framework.
  - PyQt is available under GPL or commercial licenses, while PySide is available under LGPL.
  - Suitable for creating professional-looking applications with advanced features.
- wxPython:
  - A binding for the wxWidgets C++ library.
  - Provides native-looking GUI applications for Windows, macOS, and Linux.

Model-View-Controller Architecture

The View

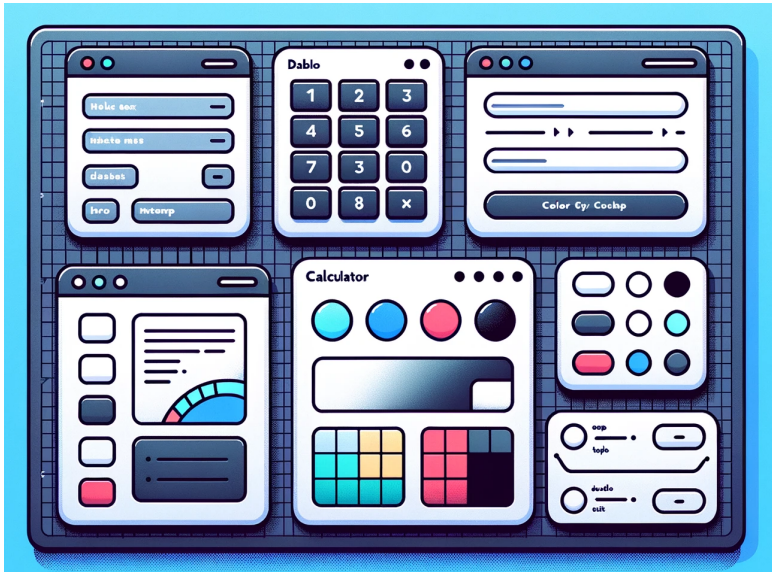
- User Interface - The View is responsible for displaying the user interface (UI) of the application. It defines how data is presented to the user and how the user interacts with it.
- Receives User Input - While the View primarily focuses on display, it also captures user input, such as button clicks, text input, or gestures, and forwards them to the Controller for processing.
- Stateless - Ideally, the View should be stateless, meaning it displays data without storing or processing it. Any data or logic-related tasks should be handled by the Model or Controller.



Model-View-Controller Architecture

The View

- Reactive to Model - The View reflects changes in the Model. When the data in the Model changes, the View updates automatically to display the latest information to the user.
- Decoupled from Model and Controller - To maintain the separation of concerns, the View should be decoupled from the Model and Controller. This means changes in the View shouldn't directly affect the Model's data or the Controller's logic, ensuring modularity and ease of maintenance.



Model-View-Controller Architecture

The Controller

- Mediator - The Controller acts as a mediator between the Model and the View. It receives user input from the View, processes it (possibly updating the Model), and then returns the display output to the View.
- Event Specification - It specifies which elements in the view should trigger events when selected.
- User Input Handling - One of the primary responsibilities of the Controller is to handle user input. Whether it's a button click, form submission, or any other interaction in the View, the Controller decides what should happen in response.

Model-View-Controller Architecture

The Controller

- Logic Execution - While the Model deals with data and the View deals with presentation, the Controller is where much of the application's business logic is executed. It determines how the application should respond to various user inputs and actions.
- State Management - The Controller often manages the flow of data and the state of the application. It can decide which View to display next, fetch data from the Model to update the View, or store data in the Model based on user input.

Model-View-Controller Architecture

The Model

- Data Representation - The Model represents the application's data and the rules that govern access to and updates of this data. It is the central component that holds the core functionality.
- Data Storage - Often, the Model is responsible for retrieving or storing data, which can be in a database, file, or any other storage mechanism.
- Data Notification - When data in the Model changes, it can notify the View so that the interface can be updated accordingly. This ensures that the user interface reflects the current state of the data.
- Logic - While the Controller handles user interactions, the Model contains the core logic that dictates how data can be created, stored, modified, and retrieved. It enforces rules, constraints, and validations related to the data.

Git Essentials

Activity #5

- Upload individual files to GitHub
- Place code in appropriate organizational structure.

Intermission