Fundamentals of Git and GitHub

1. Getting Started with Git

Git is a version control system that tracks code changes, enabling collaboration and version tracking. First, install Git on your system and configure it with your username and email:

```
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

2. Basic Git Commands

- **Initialize a Repository:** Create a new Git repository in a directory.

  ```
  git init
  ```

- **Check Repository Status:** See which files are staged, unstaged, and untracked.

  ```
  git status
  ```

- **Adding Changes:** Add changes to the staging area.

  ```
  git add <filename>
  ```

- **Commit Changes:** Save changes in the repository with a message.

  ```
  git commit -m "Your commit message"
  ```

3. GitHub Essentials

GitHub is a platform for hosting and collaborating on Git repositories. Once you've created an account:

- **Create a Repository:** You can create one on GitHub directly.
- **Push a Local Repository to GitHub:**

  ```
  git remote add origin https://github.com/your-username/your-repo.git
  git push -u origin main
  ```

4. Branching in Git

Branching allows you to create separate versions of your project to work on features independently.

- **Create a Branch:**

  ```
  git branch <branch-name>
  ```

- **Switch to a Branch:**

  ```
  git checkout <branch-name>
  ```

- **Merge Branches:** Merge changes from one branch into another.

  ```
  git checkout main
  git merge <branch-name>
  ```

5. Collaborative Features on GitHub

- **Forking:** Create a copy of someone else's repository on your GitHub to make changes independently.
- **Pull Requests:** After making changes, you can submit a pull request for review.

6. Advanced Git Topics

- **Rebasing:** Clean up commit history and integrate changes from one branch to another.

  ```
  git rebase <branch-name>
  ```

- **Stashing:** Temporarily save changes when switching branches.

  ```
  git stash
  git stash apply
  ```

- **Interactive Rebase:** Edit, reorder, or squash commits.

  ```
  git rebase -i HEAD~<number of commits>
  ```

7. Advanced GitHub Actions

- **GitHub Actions:** Automate workflows, like running tests or deploying code.
- **GitHub Pages:** Host static websites directly from a repository.
- **Protecting Branches:** Set rules to restrict direct pushes to critical branches.

Advanced Git Concepts

1. **Advanced Branching Strategies**
   - **Feature Branch Workflow:** Each feature has its own branch, allowing isolated development.
   - **Git Flow Workflow:** A well-known branching model that uses multiple branches for each stage of development (e.g., `main`, `develop`, `feature`, `release`, and `hotfix`).
   - **Trunk-Based Development:** Developers work on a single (typically main) branch with short-lived branches instead of creating multiple branches. This approach encourages frequent commits and merges.
2. **Rebase vs. Merge**
   - **Merge:** Combines branches by creating a "merge commit." It preserves the history of both branches, but the commit history can become cluttered.
   - **Rebase:** Transfers your changes from one branch to another, creating a linear history. This method is cleaner but can be risky if done incorrectly, as it rewrites history. Use `rebase` primarily for local branches, and avoid it on shared public branches.

   Example of Rebasing onto `main`:

   ```
   git checkout feature-branch
   git rebase main
   ```

   This replay commits from the `feature-branch` onto the latest `main` branch.

3. **Interactive Rebase**
   - Use interactive rebase to rewrite the commit history, allowing you to edit, reorder, or squash commits.

   ```
   git rebase -i HEAD~<number of commits>
   ```

   Squashing Commits: Combine multiple commits into one to keep the commit history clean. For example, if you have three commits that make minor changes to a feature, you might want to squash them into one.

4. **Cherry-Picking**
   - Cherry-pick allows you to apply a specific commit from one branch to another without merging the entire branch.

   ```
   git checkout target-branch
   git cherry-pick <commit-hash>
   ```

5. **Git Tags and Releases**
   - **Tags:** Mark specific points in your commit history as important, often used for releases.

     ```
     git tag -a v1.0 -m "Version 1.0 release"
     git push origin v1.0
     ```

- **Annotated vs Lightweight Tags:** Annotated tags contain metadata (author, date, message) and are generally preferred for releases.

6. **Amending Commits**
   - If you forgot to add changes to your last commit, you can amend it:

     ```
     git commit --amend
     ```

   Caution: Avoid amending commits if they've already been pushed to a shared branch.

7. **Managing Large Repositories and BFG Repo Cleaner**
   - Git can struggle with large files or repositories with long histories. Tools like BFG Repo-Cleaner help remove large or sensitive files from history.

Advanced GitHub Features

1. **GitHub Actions**
   - GitHub Actions automate tasks like running tests or deploying code. You can create workflows to trigger on events, such as push, pull request, or issue creation.
   - **Example Workflow:**

   ```
   name: CI

   on: [push, pull_request]

   jobs:
     build:
       runs-on: ubuntu-latest
       steps:
         - uses: actions/checkout@v2
         - name: Set up Python
           uses: actions/setup-python@v2
           with:
             python-version: '3.8'
         - name: Install dependencies
           run: pip install -r requirements.txt
         - name: Run tests
           run: pytest
   ```

2. **GitHub Pages**
   - Host static sites directly from a GitHub repository, which is ideal for project documentation or portfolio sites. Just push the HTML files to a `gh-pages` branch.
3. **GitHub Projects and Issues**
   - Use **GitHub Issues** to track bugs and features and assign labels, milestones, and assignees.
   - **GitHub Projects** lets you create project boards to organize tasks, set priorities, and track progress.
4. **Code Review Best Practices**
   - **Pull Requests (PRs):** Always create PRs for code reviews. You can set branch protections on critical branches, so every change must go through a PR.
   - **Review Suggestions:** Use the suggestion feature to propose specific code changes.
5. **Protecting Branches and Enforcing Rules**
   - Set branch protection rules to require PR reviews, disallow direct commits and enforce status checks.
   - Enable **required checks** on PRs, so builds or tests must pass before merging.
6. **Managing Access and Permissions**
   - Assign **collaborators** with different levels of access to repositories. Teams can use GitHub's **organization** structure to manage access across multiple repositories more efficiently.
7. **Insights and Analytics**
   - GitHub provides insights on contributors, commits, and code frequency, allowing you to monitor team activity and identify project development trends.

GitHub Actions Workflows

GitHub Actions enable you to automate tasks directly within GitHub. You can set up workflows to handle tasks such as continuous integration (CI), deployment, code linting, and testing. Workflows are defined in your repository in YAML files located in .github/workflows/.

Basic Workflow Structure

Each workflow follows a basic structure, including a name, event triggers, jobs, and steps:

```
name: CI Workflow

on:

  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.8'

      - name: Install dependencies
        run: pip install -r requirements.txt

      - name: Run tests
        run: pytest
```

Understanding the Components

- **Triggers (`on`)**: Define when the workflow should run. Common triggers include `push`, `pull_request`, `schedule` (for timed runs), and `workflow_dispatch` (manual run).
- **Jobs**: Each job runs in its own virtual environment and can have multiple steps. Jobs are parallel by default but can be sequential by defining dependencies.
- **Steps**: Actions or commands executed within a job. They include prebuilt actions (like `actions/checkout@v2`) or custom shell commands.

Advanced Workflow Examples

- **Conditional Workflows**: Only run jobs under specific conditions. For example, you can restrict actions to only run on specific paths or branches:
```
on:


  push:
    branches:
      - main
    paths:
      - "src/**"
```
- **Matrix Builds**: Test across multiple environments. Useful for testing compatibility across different versions of Python, Node, etc.
```
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.6, 3.7, 3.8, 3.9]
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python-version }}
      - run: pip install -r requirements.txt
      - run: pytest
```
- **Using Secrets**: Store sensitive data, like API keys, securely. Access them using `${{ secrets.SECRET_NAME }}`.
```
steps:
  - name: Deploy to server
    run: ssh user@server "deploy_script.sh"
    env:
      DEPLOY_KEY: ${{ secrets.DEPLOY_KEY }}
```

Advanced Merging Techniques

Efficient merging is key to maintaining a clean and understandable project history. Here's an overview of more advanced merging approaches:

1. Merge vs. Rebase

- **Merge**: Combines branches by creating a merge commit. This approach preserves the entire branch history, making it useful for understanding how branches diverged.

  ```
  git checkout main
  git merge feature-branch
  ```

- **Rebase**: Moves your branch onto another branch by "replaying" commits. Rebase creates a linear history, which can simplify the commit log.

  ```
  git checkout feature-branch
  git rebase main
  ```

2. Interactive Rebase

Interactive rebasing (`git rebase -i`) allows you to edit, reorder, squash, or even delete commits. It's useful for cleaning up commit history before merging a feature branch.

```
git rebase -i HEAD~<number of commits>
```

In the interactive rebase editor:

- **pick** keeps the commit as-is.
- **squash** combines the commit with the previous one.
- **edit** lets you change the commit message or modify the commit content.

3. Cherry-Picking

Cherry-picking lets you apply specific commits from one branch to another, without merging the entire branch. This can be useful if you only need certain features or fixes.

```
git checkout main
git cherry-pick <commit-hash>
```

4. Squash Merging

On GitHub, when merging a pull request, you can choose **Squash and Merge** to combine all commits into a single commit on the main branch. This method is especially helpful for keeping a clean history in long-running projects where each PR represents a feature or bug fix.

Combining Actions and Advanced Git Techniques

A common use case: deploying code after merging to `main`. You could set up a workflow triggered by a merge into `main` that runs tests and, if they pass, deploys your code.

Example workflow that combines `push` and `deploy` actions:

```
name: Deploy on Merge to Main

on:
  push:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: '14'
      - run: npm install
      - run: npm test

  deploy:
    needs: test
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Deploy to Server
        run: ssh -i ~/.ssh/deploy_key user@server "deploy_script.sh"
        env:
          DEPLOY_KEY: ${{ secrets.DEPLOY_KEY }}
```

In this setup:

- The `test` job runs tests after every push to `main`.
- The `deploy` job only runs if `test` passes, ensuring deployment happens only when all tests succeed.

Setting Up a GitHub Actions Workflow with Specific Requirements

Imagine you have a Python project, and you want a workflow to:

1. Run tests on multiple Python versions.
2. Lint the code for style consistency.
3. Only deploy if tests pass and if the branch is `main`.

Here's a GitHub Actions workflow to achieve this:

1. Create the Workflow File

In your repository, create a new file `.github/workflows/ci.yml`.

2. Define the Workflow

Here’s a sample workflow file:

```
name: CI and Deploy Workflow

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main

jobs:
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: [3.8, 3.9, 3.10]
    steps:
      - name: Check out code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python-version }}

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run tests
        run: pytest

  lint:
    runs-on: ubuntu-latest
    steps:
```

```yaml
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: 3.8

      - name: Install linter
        run: pip install flake8

      - name: Run linter
        run: flake8 .

  deploy:
    needs: [test, lint]
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v2
      - name: Deploy to server
        run: ssh -i ~/.ssh/deploy_key user@server "deploy_script.sh"
        env:
          DEPLOY_KEY: ${{ secrets.DEPLOY_KEY }}
```

Workflow Explanation

- **Triggers** (`on`): The workflow runs on pushes and pull requests to `main`.
- **Jobs**:
  - **`test` job**: Runs tests on three Python versions using a matrix.
  - **`lint` job**: Uses `flake8` to check for style issues in the code.
  - **`deploy` job**: Runs only if the `test` and `lint` jobs succeed and only on the `main` branch.

With this setup, you'll catch errors early, enforce a consistent coding style, and ensure deployment only when tests and linter checks pass.

Resolving Conflicts during Rebase and Merge

Conflicts arise when changes in different branches affect the same line or area of a file. Let's look at how to handle conflicts during rebase and merge.

1. Resolving Conflicts during Merge

When you merge a branch with conflicting changes, Git will notify you of the conflicts and pause the merge.

```
git checkout main
git merge feature-branch
```

If there are conflicts, Git will output a list of conflicting files. Open each conflicting file in an editor, where you'll see conflict markers:

```
<<<<<<< HEAD
// Code from the main branch
=======
 // Code from feature-branch
>>>>>>> feature-branch
```

Choose the correct code (or combine both parts), then delete the conflict markers (<<<<<<<, =======, and >>>>>>>). After fixing the conflicts:

```
git add <filename>
git commit
```

This completes the merge with conflicts resolved.

2. Resolving Conflicts during Rebase

Conflicts are handled similarly when you rebase, but the commands differ slightly. Suppose you want to rebase `feature-branch` onto the `main`:

```
git checkout feature-branch
git rebase main
```

During the rebase, if conflicts arise, Git will pause and let you know. Open the conflicting files, resolve the conflicts, and then:

```
git add <filename>
git rebase --continue
```

If you need to cancel the rebase, use:

```
git rebase --abort
```

Note: Avoid rebasing shared branches as it rewrites commit history, which can cause issues for other contributors.

Handling Complex Merge Conflicts

– Managing Large Projects Or Branches With Long Histories, Where Conflicts Are More Challenging –

1. Understand the History with Git Log

When dealing with complex conflicts, understanding how a branch diverged can help clarify why the conflict arose. Use the `git log` command to examine recent commits:

```
git log --oneline --graph --all --decorate
```

This view shows the commit history, branch relationships, and changes' progress. Use `git log -p` to see the actual changes in each commit, which can provide context for resolving conflicts.

2. Use Git Diff to Identify Differences

You can compare branches before merging to see what conflicts might arise and prepare for them. For example, to see the differences between the `main` and `feature-branch`:

```
git diff main feature-branch
```

Or, if you're rebasing and want to see what's changed on `main`:

```
git diff feature-branch main
```

This helps identify areas where code might need merging or adjustments.

3. Use Git's `merge` Conflict Markers Effectively

When you're in a merge conflict, Git adds conflict markers (<<<<<<<, =======, and >>>>>>>). Here's a strategy to interpret and resolve them:

- **Left (HEAD)**: Code in your current branch.
- **Right (MERGE_HEAD)**: Code in the branch being merged

Choose one version, combine both, or rephrase to maintain each branch's contributions.

4. Using the `git mergetool` Command

Git provides external mergetools to assist with resolving conflicts visually, like `kdiff3`, `meld`, and `vimdiff`. To use them, configure Git:

```
git config --global merge.tool <your-preferred-tool>
```

Then run:

```
git mergetool
```

This opens the tool, showing changes side-by-side to resolve conflicts more easily.

5. Break Down Large Commits

For complex conflicts in large commits, it may help to split the work into smaller commits. This is easier to manage and makes it clearer what changes caused conflicts. To do this, consider using an **interactive rebase** to split large commits.

```
git rebase -i HEAD~<number of commits>
```

In the interactive rebase editor, change `pick` to `edit` for the commit you want to split. Then:

1. Amend the commit to include only part of the changes (`git add -p`).
2. Commit the changes (`git commit --amend`).
3. Repeat for the remaining changes.

This can simplify conflict resolution as you work with smaller, more manageable changes.

6. Use `git rerere` to Reuse Resolutions

If you encounter similar conflicts often, Git can remember how you resolved a conflict and automatically apply that resolution in the future. Enable this with:

```
git config --global rerere.enabled true
```

When `rerere` is enabled, Git records how you resolve each conflict. The next time a similar conflict occurs, Git automatically applies your previous resolution.

7. Understand 3-Way Merge vs. Recursive Merge

When you merge branches with more complex histories, Git typically performs a **3-way merge**, looking at the common ancestor between two branches. In recursive merges, Git can merge multiple branches in sequence, which helps in cases of multiple branches with conflicting changes. To force recursive merging:

```
git merge --strategy recursive
```

Or you can set it in Git config for regular use:

```
git config --global merge.conflictstyle diff3
```

This shows the common ancestor in conflicts, helping you understand the base for each change.

8. Use `git stash` to Set Aside Changes Temporarily

If you're in the middle of a conflict but need to work on something else, use `git stash` to save your current changes temporarily:

```
git stash
```

To retrieve the changes after completing other work:

```
git stash pop
```

This is helpful if you want to reattempt the merge or review some other parts of the code.