Introduction to Programming with Python

Dr. Anatol Wegner

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

anatol.wegner@uni-wuerzburg.de

Lecture 08
Version control with git

January 10, 2025

Recap

Introduction to NLP with NLTK:

- Overview of NLP tasks and common techniques.
- **Text Processing:**
  - Loading and cleaning text data.
  - Tokenization and basic text preprocessing.
- **Lemmatization:**
  - Using NLTK's lemmatizer to reduce words to their base forms.
- **Sentiment Analysis:**
  - Used CountVectorizer to convert text to features & trained a classifier for sentiment prediction.

Recap

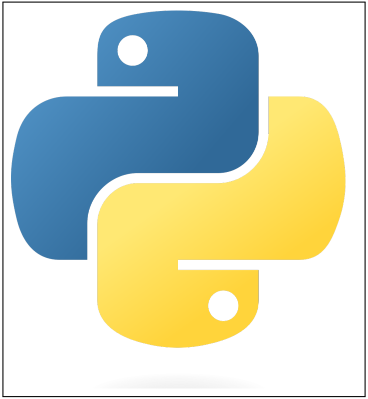
Introduction to NLP with NLTK:

- Overview of NLP tasks and common techniques.
- **Text Processing:**
  - Loading and cleaning text data.
  - Tokenization and basic text preprocessing.
- **Lemmatization:**
  - Using NLTK's lemmatizer to reduce words to their base forms.
- **Sentiment Analysis:**
  - Used CountVectorizer to convert text to features & trained a classifier for sentiment prediction.

Recap

Introduction to NLP with NLTK:

- Overview of NLP tasks and common techniques.
- **Text Processing:**
  - Loading and cleaning text data.
  - Tokenization and basic text preprocessing.
- **Lemmatization:**
  - Using NLTK's lemmatizer to reduce words to their base forms.
- **Sentiment Analysis:**
  - Used CountVectorizer to convert text to features & trained a classifier for sentiment prediction.



Today: Version control with Git

- Basic concepts and commands in git.
- Collaborative development via remote repositories.

What is Version Control?

- Like "track changes" in Word, but much more powerful!
- **Benefits:**
  - Track the history of your project.
  - Revert to previous versions.
  - Experiment without fear of losing work.
  - Collaborate seamlessly.

Why Git?

- Most popular version control system.
- Free and open-source.
- Widely used in various fields.

Installation

1. Download and install Git from the official website: `https://git-scm.com/`
2. Configure Git:
    - Set your name: ‘git config –global user.name "Your Name"‘
    - Set your email: ‘git config –global user.email "your.email@example.com"‘

Creating a git Repository

- **Repository:** A project folder tracked by Git.
  - Git can track multiple files of different types simultaneously.
  - Git can also track sub folders.
- **Initializing a git repo:**
  - Open your terminal/command prompt.
  - Navigate to your project folder.
  - Type ‘git init‘.

How Git Tracks Changes

Git uses a snapshot-based approach:

- Each commit is a snapshot of the entire project at a given point in time.
- Files that haven't changed are stored as links to the previous snapshot.

Core Components

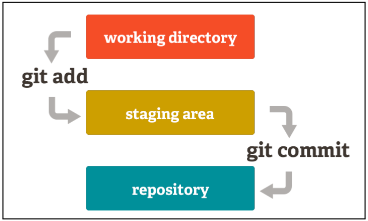
- **Objects:** Git stores everything as objects:
  - **Blob:** Stores file content.
  - **Tree:** Represents a directory, linking blobs and sub-trees.
  - **Commit:** Points to a tree and includes metadata (e.g., author, message).
- **Hashes:** Each object has a unique identifier (a SHA-1 hash).

Example:

```
Commit -> Tree -> Blob (file1)
                -> Blob (file2)
                -> Tree (subdir) -> Blob (file3)
```

Staging and Committing Changes in Git

- **Staging Area:** A temporary holding area for changes.
  - Allows you to select which modifications should be included in the next commit.
  - Provides granular control over version history.
- **Committing:**
  - Creates a snapshot of the staged changes, permanently recording them in the repository.
  - Each commit is accompanied by a message explaining the changes made.



Basic Git Commands

Command	**Description**
‘git status‘	Check the status of your files
‘git add filename ‘	Stage changes for commit
‘git reset filename ‘	Unstage changes
‘git commit -m "message"‘	Record changes with a message
‘git log‘	View the commit history

Example: Tracking a Text File

1. **Create a file:** Create a new file named `paper.txt` and write the first paragraph of your paper in `paper.txt`.
2. **Initialize a repository:** Open your terminal, navigate to the folder containing `paper.txt`, and type `git init`.
3. **Check status:** Type `git status`. You'll see `paper.txt` listed as untracked.
4. **Stage changes:** Type `git add paper.txt`.
5. **Commit changes:** Type `git commit -m "First draft of introduction"`.
6. **Make more changes:** Add a second paragraph to `paper.txt`.
7. **Stage and commit again:** Repeat steps 4-6 with a new commit message, e.g., "Added second paragraph".
8. **View history:** Type `git log` to see your commit history.

Understanding ‘git log‘

- ‘git log‘ displays the commit history of your repository. Each commit entry includes:
  - **Commit hash:** A unique identifier for the commit (e.g., `a1b2c3d`).
  - **Author & Date:** The name and email of the person who made the commit and the date and time of the commit.
  - **Commit message:** The message provided when the commit was created.

Understanding ‘git log‘

- Example output:

```
commit a1b2c3d (HEAD -> main)
Author: Your Name <your.email@example.com>
Date:   Thu Jan 9 14:00:00 2025 +0100

    Added second paragraph

commit 7f6e5d4c
Author: Your Name <your.email@example.com>
Date:   Thu Jan 9 13:30:00 2025 +0100

    First draft of introduction
```

Comparing Changes with ‘git diff‘

- **See what’s changed:** ‘git diff‘ shows you the differences between various states of your project.
- **Common use cases:**
  - **‘git diff‘**: See the changes you’ve made but haven’t staged yet.
  - **‘git diff –staged‘**: See the changes you’ve staged but haven’t committed yet.
  - **‘git diff commit_hash commit_hash ‘**: See the differences between two commits.
- **Understanding the output:** ‘git diff‘ uses a standard format to highlight additions and deletions.
- **Benefits:**
  - Review your changes before committing.
  - Understand the impact of different commits.
  - Identify the source of bugs or errors.

Peeking into the Past with ‘git checkout‘

- **Time travel without consequences:** ‘git checkout‘ allows you to temporarily switch to a different point in your project’s history without altering the current state.
- **Identify the commit:** Use ‘git log‘ to find the commit hash you want to explore.
- **Checkout a commit:** ‘git checkout commit_hash ‘ will temporarily switch your project to the state it was in at that commit. You can examine the files as they were at that specific point in time.
- **Return to the present:** ‘git checkout branch name ‘ (usually ‘main‘ or ‘master‘) will bring you back to your current working state.
- **Important notes:**
    - ‘git checkout‘ is like a "read-only" view of the past. Any changes you make in this state won’t be saved unless you create a new branch.
    - It’s a powerful tool for inspecting previous versions, understanding how your project evolved, and even recovering lost code.

Reverting to a Previous Version

- **Mistakes happen! Git allows you to go back in time.**
- **Identify the commit:** Use ‘git log‘ to find the commit hash you want to revert to.
- **Revert with ‘git revert commit_hash ‘:**
  - This command **undoes the changes** introduced in the specified commit.
  - It analyzes the changes made in that commit and applies the **opposite changes** to your current files.
  - For example, if the commit added a paragraph, ‘git revert‘ will delete that paragraph. If the commit deleted a sentence, ‘git revert‘ will add it back.
  - Git then creates a **new commit** with these "undo" changes, keeping a clear record of your actions.
- **Important:**
  - Reverting doesn’t erase the original commit from history.
  - It adds a new commit that counteracts the changes, maintaining a complete and traceable history.

Resetting to a Previous State with ‘git reset‘

- **A more forceful way to go back in time:** ‘git reset‘ moves the HEAD pointer (which indicates your current position in the project history) to a specific commit.
- **How it works:**
  - It’s like rewinding your project’s history to a specific point.
  - By default, it doesn’t delete commits, but it makes them inaccessible through the normal ‘git log‘.
  - Think of it like removing pages from a table of contents; the pages still exist in the book, but you can’t easily find them anymore.
- **Use with caution!**: ‘git reset‘ can alter your project history, so it’s important to understand its implications.

Resetting to a Previous State with ‘git reset‘

- **Reset options:**
  - **‘git reset –soft commit_hash ‘**: Keeps your changes staged.
  - **‘git reset –mixed commit_hash ‘**: Unstages your changes (default).
  - **‘git reset –hard commit_hash ‘**: Discards all uncommitted changes and any commits after the specified commit.
- **Example:** ‘git reset –hard a1b2c3d‘ will reset your project to the state it was in at commit ‘a1b2c3d‘, discarding any changes made after that commit.
- **Warning:** Be extremely careful with ‘git reset –hard‘ as it permanently deletes any uncommitted changes and makes it difficult to recover the "lost" commits!

Example: Using ‘git revert‘ and ‘git reset‘

Scenario:

We have a file named ‘myfile.txt‘ with three lines added in separate commits.

1. **Revert the last commit:**

   ```
   git revert HEAD
   ```

   This undoes the changes introduced in the last commit (adding the third line) by creating a new commit.

2. **Reset to the first commit:**

   ```
   git reset --hard HEAD~2
   ```

   This resets the repository to the state of the first commit, discarding the second and third commits and any unstaged changes. **Use with caution!**

3. **Verify the changes:**

   ```
   cat myfile.txt
   ```

   This will show that the file now only contains the first line.

Ignoring Files with ‘.gitignore‘

- **Keep your repository clean:** Not all files belong in version control (e.g., temporary files, generated files, large datasets).
- **The ‘.gitignore‘ file:** A special file that tells Git which files and folders to ignore.
- **How it works:**
  - Create a file named ‘.gitignore‘ in your repository’s root directory.
  - List the files and folders you want to ignore, one per line.
  - Use wildcards (*) to match multiple files (e.g., ‘*.tmp‘ to ignore all files with the ‘.tmp‘ extension).
- **Example:**

```
*.tmp
temp/
data/large_dataset.csv
```

- **Benefits:**
  - Keeps your repository organized and focused.
  - Prevents accidental commits of unnecessary files.
  - Reduces the size of your repository.

What are Branches?

- **Independent lines of development:** Branches allow you to create separate versions of your project.
- **Like parallel universes:** Each branch can evolve independently without affecting other branches.
- **Experiment freely:** Branches provide a safe space to try new ideas, explore different approaches, or fix bugs without disrupting the main project.

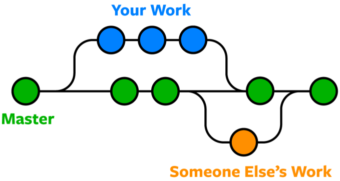


Figure: Branching in Git

Working with Branches

- **Create a new branch:** ‘git branch branch_name ‘ (e.g., ‘git branch feature-x‘)
- **Switch to a branch:** ‘git checkout branch_name ‘
- **List all branches:** ‘git branch‘ (the current branch will be highlighted)
- **Merge branches:** ‘git merge branch_name ‘ (merges the specified branch into the current branch)
- **Delete a branch:** ‘git branch -d branch_name ‘ (only after merging it)

Why Use Branches?

- **Feature development:** Isolate new features or experimental changes from the main codebase.
- **Bug fixing:** Create a dedicated branch to fix bugs without interrupting other work.
- **Collaboration:** Allow multiple people to work on different parts of the project simultaneously.
- **Versioning:** Maintain different versions of your project (e.g., for different publications or audiences).

Branching Best Practices

- **Keep your branches focused:** Each branch should have a specific purpose (e.g., a new feature, a bug fix).
- **Use descriptive names:** Choose names that clearly indicate the purpose of the branch (e.g., ‘feature-new-chapter‘, ‘fix-bibliography‘).
- **Commit frequently:** Make small, frequent commits with clear messages to track your progress.
- **Merge regularly:** Merge your branches back into the main branch regularly to avoid large, complex merges.
- **Don’t abandon branches:** Delete branches after they have been merged to keep your repository organized.
- **Communicate with collaborators:** If you’re working on a shared project, communicate your branching strategy and any merge conflicts with your collaborators.

Example: Branching for a Research Paper

- **Main branch (main):** Contains the stable version of your paper.
- **Feature branch (analysis-chapter):** Used to work on a new chapter with in-depth analysis.
- **Bug fix branch (fix-typos):** Used to correct typos and grammar errors.
- **Workflow:**
  1. Create the feature and bug fix branches from the main branch.
  2. Work on each branch independently.
  3. Merge the branches back into the main branch when they are ready.

Example: Working with Branches

Scenario

We want to add a new section to our research paper without affecting the main draft.

1. **Create a new branch:**

   ```
   git branch new-section
   ```

   This creates a new branch called "new-section".

2. **Switch to the new branch:**

   ```
   git checkout new-section
   ```

   This switches to the "new-section" branch.

3. **Make changes and commit:**

   ```
   # Edit the paper.txt file to add the new section
   git add paper.txt
   git commit -m "Added new section"
   ```

   This adds and commits the changes to the "new-section" branch.

Example: Working with Branches

Scenario

We want to add a new section to our research paper without affecting the main draft.

4. **Switch back to the main branch:**

   ```
   git checkout main
   ```

   This switches back to the "main" branch.

5. **Merge the new branch:**

   ```
   git merge new-section
   ```

   This merges the changes from the "new-section" branch into the "main" branch.

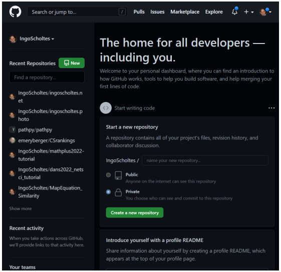
6. **(Optional) Delete the branch:**

   ```
   git branch -d new-section
   ```

   This deletes the "new-section" branch after it has been merged.

What are Remote Repositories?

- **Repositories on a server:** Remote repositories are versions of your project stored on a server, like GitHub, GitLab, or Bitbucket.
- **Centralized hub:** They act as a central hub for collaboration, allowing multiple users to access and contribute to the same project.
- **Backup and sharing:** Remote repositories provide a backup of your work and enable easy sharing with others.



Recent Repositories

IngoScholtes/ingoscholtes.net
IngoScholtes/ingoscholtes.photo
pathpy/pathpy
emeryberger/CSrankings
IngoScholtes/mathplus2022-tutorial
IngoScholtes/dans2022_netsci_tutorial
IngoScholtes/MapEquation_Similarity

Recent activity

When you take actions across GitHub, we'll provide links to that activity here.

Your teams

The home for all developers — including you.

Welcome to your personal dashboard, where you can find an introduction to how GitHub works, tools to help you build software, and help merging your first lines of code.

Start writing code

Start a new repository

A repository contains all of your project's files, revision history, and collaborator discussion.

IngoScholtes / name your new repository...

○ Public
Anyone on the internet can see this repository

● Private
You choose who can see and commit to this repository

Introduce yourself with a profile README

Share information about yourself by creating a profile README, which appears at the top of your profile page.

Figure: A remote repository

Setting up Git for Remote Repositories

- **Choose a hosting service:**
  - Popular options include GitHub, GitLab, and Bitbucket.
  - GitHub is generally recommended for beginners due to its user-friendly interface and large community.
- **Create an account:** Sign up for an account on your chosen platform.
- **Generate SSH keys (optional but recommended):**
  - SSH keys provide a secure way to authenticate with the remote repository without needing to enter your password every time.
  - Follow the instructions on your chosen platform to generate SSH keys and add them to your account.
- **Create a new repository (or connect to an existing one):**
  - On the platform, create a new repository or obtain the URL of an existing repository you want to contribute to.
- **Connect your local repository to the remote:**
  - If you created a new repository, use ‘git remote add origin repository_url ‘ to connect your local repository to the remote.
  - If you’re connecting to an existing repository, you’ll typically do this during the ‘git clone‘ step.

Key Commands for Remote Collaboration

- **Clone a repository:**
  - ‘git clone repository_url ‘
  - Creates a local copy of the remote repository on your computer.
  - Example: ‘git clone https://github.com/username/project.git‘
- **Push your changes:**
  - ‘git push origin branch_name ‘
  - Uploads your local commits to the remote repository.
  - ‘origin‘ usually refers to the default remote (the original repository you cloned from).
  - Example: ‘git push origin my-feature-branch‘
- **Pull changes from others:**
  - ‘git pull origin branch_name ‘
  - Downloads and merges changes from the remote repository into your local branch.
  - Example: ‘git pull origin main‘

Pull Requests: The Heart of Collaboration

- **Proposing changes:** A pull request is a formal way to propose changes to a project.
- **Discussion and review:** It initiates a discussion around your proposed changes, allowing for feedback, suggestions, and improvements before they are integrated.
- **Quality control:** Pull requests help maintain code quality by ensuring that changes are reviewed and approved before being merged.
- **Transparency:** They provide a transparent record of all proposed changes and the discussions surrounding them.

Pull Requests: The Heart of Collaboration

- **Example Workflow (on GitHub):**
  1. **Push your branch:** ‘git push origin my-feature-branch‘
  2. **Open a pull request:** Go to the GitHub repository in your web browser and click "New pull request".
  3. **Select branches:** Choose your branch (‘my-feature-branch‘) as the "compare" branch and the main branch (‘main‘) as the "base" branch.
  4. **Provide details:** Give your pull request a title and a clear description of the changes you’ve made.
  5. **Request review:** Request a review from your collaborators.
  6. **Discuss and revise:** Address any feedback or suggestions from your collaborators.
  7. **Merge the pull request:** Once approved, click "Merge pull request" to integrate your changes into the main branch.

Benefits of Collaboration with Git

- **Simultaneous work:** Multiple people can work on the same project without interfering with each other.
- **Organized contributions:** Branches and pull requests help manage contributions and ensure code quality.
- **Clear history:** Git tracks all changes and contributions, providing a transparent record of the project's evolution.
- **Efficient communication:** Pull requests facilitate communication and discussion around code changes.

Key Takeaways

- **Version control essentials:**
  - Track changes to your files.
  - Revert to previous versions.
  - Experiment without fear.
- **Git basics:**
  - Staging and committing changes.
  - Branching for parallel development.
  - Viewing history and differences.
- **Collaboration with Git:**
  - Remote repositories (GitHub).
  - Cloning, pushing, and pulling.
  - Branching and merging.
  - Pull requests for code review.

Self-Study Questions

1. What are the main benefits of using version control?
2. How do you initialize a Git repository?
3. Explain the difference between staging and committing changes.
4. What is the purpose of a commit message?
5. How can you view the history of changes in a Git repository?
6. What are branches, and why are they useful?
7. How do you create, switch, and merge branches?
8. What is a remote repository, and how does it facilitate collaboration?
9. Explain the purpose of a pull request.
10. Can you think of a specific example of how you could use Git in a project you're working on?

Additional Resources

- **Official Git Website:** `https://git-scm.com/`
  - Comprehensive documentation, downloads, and tutorials.
- **GitHub Guides:** `https://guides.github.com/`
  - Beginner-friendly guides and tutorials on using Git and GitHub.
- **Atlassian Git Tutorials:** `https://www.atlassian.com/git/tutorials`
  - In-depth tutorials on various Git concepts and workflows.
- **Pro Git Book (online):** `https://git-scm.com/book/en/v2`
  - A free online book that covers Git in detail.
- **Oh My Git! (Interactive Tutorial):** `https://ohmygit.org/`
  - A fun and interactive way to learn Git commands.