Welcome to this CoGrammar Lecture:

Iteration and GitHub

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators answering questions.


SKILLS FOR LIFE
SKILLS BOOTCAMPS
Department for Education
CoGrammar
Iteration and GitHub
March 2024

Agenda

- While Loops
- For Loops
- Ranges
- Trace Tables
- Git
- GitHub

Iterations

While & For-Loops

CoGrammar

Iterations

Iterations refer to the process of repeatedly executing a set of instructions or a block of code. In programming, iterations are commonly associated with loops, where the same code block is executed multiple times, either for a specified number of times or until a certain condition is met.

Iterations in Programming

- Each execution of the code block within a loop is called an **iteration**. Iterations are essential for automating repetitive tasks and processing collections of data efficiently.

  " Loops are like magic tricks in programming that help us avoid doing the same thing over and over again. Instead of writing the same code again, we use loops to make the computer do it for us. This saves time and makes our programs neat and tidy. "

- In coding, there are two kinds of loops: **for loops**, which we use when we know exactly how many times we want to repeat something, and **while loops**, which we use when we want to keep doing something until a certain condition is **true.**

While Loops

CoGrammar

While Loops

- While loops are control flow structures that repeatedly execute a block of code as long as a specified condition is true.

- These are used when you want to execute a block of code repeatedly as long as a specified condition is true. They continue iterating until the condition becomes false.

```
while condition:
    # code block to be executed
```

While Loops Example

```
count = 0
while count < 5:
    print("Count is:", count)
    count += 1
# This will print numbers from 0 to 4.
```

- while count < 5: This is the beginning of a while loop. It checks if the value of count is less than 5. If this condition is true, the code block inside the loop will execute. If the condition is false, the loop will terminate.
- print("Count is:", count): This line prints the current value of count along with the text "Count is:". Since count is initially 0, it will print "Count is: 0".
- count += 1: This line increments the value of count by 1 in each iteration of the loop. So, after the first iteration, count becomes 1, then 2, and so on.

For Loops

CoGrammar

For Loops

- For loops are control flow structures used to iterate over a sequence (such as a list, tuple, string, etc.) and execute a block of code for each element in the sequence.

- For loops are used when you know the number of times you want to execute a block of code.

```
for item in sequence:
    # code block to be executed
```

For Loops Example

```
pokemon_list = ["Pikachu", "Charizard", "Squirtle", "Slowpoke"]
for pokemon in pokemon_list:
    print(pokemon)
```

```
# Results
Pikachu
Charizard
Squirtle
Slowpoke

# This will print each pokemon in the list.
```

For Loops P2

Range() function

CoGrammar

Range()

- Range is a built-in Python function used to generate a sequence of numbers. It is commonly used with for loops.

- Ranges in for loops are a way to specify a sequence of numbers that you want to iterate over. The range() function generates this sequence of numbers based on the arguments you provide.

```
range(start, stop, step)
```

Range()

- **Range()** takes three arguments: start, stop, and step.
  - **start:** The starting value of the sequence (inclusive). If not provided, it defaults to 0.
  - **stop:** The ending value of the sequence (exclusive). This is a required argument.
  - **step:** The increment between each value in the sequence. If not provided, it defaults to 1.

Range() Example

```
for i in range(start, stop, step):
    # code block to be executed
```

```
for i in range(1, 6):  # This will iterate from 1 to 5
    print(i)
```

- This loop will print numbers 1 through 5. Remember, the stop value is exclusive, so the loop stops before reaching 6.

Let's take a break

CoGrammar

Trace Tables

CoGrammar

Trace Tables

- A trace table is like a step-by-step record of your program's journey through each line of code. It helps you keep track of how the values of variables change as the program runs.

“ Imagine you have a friend who's trying to bake a cake by following a recipe. A trace table is like a detailed checklist your friend uses to make sure they don't forget any steps and to see how the cake changes at each step. “

Trace Tables

```
x = 5
y = 2
z = x + y
print(x, y, z)
```

- Consider the following code:
  - A trace table would track the values of `x`, `y`, and `z` at each step of execution.

Trace Tables cont.

	A	B	C	D	E
1	Step	x	y	z	
2	1	5	2	-	
3	2	5	2	7	
4	3	5	2	7	
5					
6					

- In the first step, we assign the value 5 to the variable x and the value 2 to the variable y. At this point, we haven't calculated z yet, so its value is None.

- In the second step, we calculate the sum of x and y, which is 5 + 2 = 7, and assign it to the variable z. Then, we print the values of x, y, and z, which are 5, 2, and 7 respectively.

Git & Github

CoGrammar

Version Control

- **Version control** is a system that records changes to files over time, allowing you to recall specific versions later.
- This system not only documents modifications but also facilitates the retrieval of precise versions at later stages, ensuring transparency, reproducibility, and efficient collaboration among team members.
- **Git** is a powerful version control system that provides developers with the tools they need to track changes, manage versions, collaborate effectively, and maintain the integrity of their projects over time.

What is Github ?

- ★ **GitHub** is a online hub for your code where you can work on projects with others, keep track of changes, and showcase your work to the world!
- ★ GitHub improves **Git**'s version control capabilities by offering an intuitive platform equipped with collaborative tools such as pull requests, issue tracking, and project management features.
- ★ Additionally, GitHub serves as a showcase for your work, enabling you to share your projects with a global audience, receive feedback, and contribute to the wider community of developers.

Key Terminology

- **Repository (Repo):**
  - A repository serves as a designated location, typically a folder or directory, where your project files reside alongside additional metadata.

- **Commit:**
  - A commit is a snapshot of your project at a specific point in time. It represents a set of changes made to the files in your repository.

- **Branch:**
  - A branch is a separate line of development within a repository. It allows you to work on new features or fixes without affecting the main project.

Key Terminology cont.

- **Merge:**
  - Merging combines changes from one branch into another, typically used to incorporate changes made in a feature branch back into the main branch.

- **Pull Request:**
  - A pull request (PR) is a request to merge changes from one branch into another. It allows for code review and collaboration before changes are merged.

Key Commands

git init	Initialise a new Git repository in your project directory.
git add	Add files to the staging area, preparing them to be committed.
git commit	Record changes to the repository with a descriptive commit message.
git status	View the status of files in your repository, including untracked, modified, or staged files.
git push	Push commits from your local repository to a remote repository, such as GitHub.
git pull	Fetch changes from a remote repository and merge them into your local repository.
git clone	Clone a remote repository to your local machine.

Github P2

CoGrammar

Git & Github Auth.

- Git Authentication
- Types of Authentication
- Remote and Local Repository
- Branches
- DAG (Directed Acyclic Graph)

What is Git Authentication ?

- **Git authentication** is the process of verifying the identity of users who interact with Git repositories. It ensures that only authorised individuals or systems can access and perform actions such as pushing changes, pulling updates, and performing administrative tasks within a Git repository.

  “ Think of Git authentication like a door lock for your code. It checks who you are (like a key) before allowing you to enter your code storage (the repository). This keeps unauthorised people from messing with your code! ”

Benefits of Git Authentication ?

- **Access Control:** Git authentication restricts access to a codebase, ensuring only authorised users can modify it, vital for security and collaboration.
- **Data Integrity:** Git authentication verifies users, guaranteeing only authorised individuals can modify the codebase, fostering accountability for contributions and code quality.
- **Protection Against Unauthorised Access:** Git authentication safeguards intellectual property and confidential data within repositories by blocking unauthorised users from tampering or stealing information.
- **Secure Collaboration:** Git authentication streamlines secure collaboration by verifying users before pushing changes, creating branches, or merging, preventing unauthorised modifications and safeguarding code quality.
- **Compliance and Auditing:** For compliance in regulated industries, Git authentication tracks user actions, creating an audit trail that proves accountability and adherence to regulations.

Git Authentication cont.

Types of Authentication

1) SSH Keys
2) HTTPS Authentication
3) Personal Access Tokens (PATs)
4) Username and Password Authentication

Creating a local Repository

- Initialise a New Git Repository

```
git init
```

- Add Files and Make Commits:

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

Setting Up a Remote Repository

- Create a Remote Repository on GitHub
  - Go to GitHub and log in to your account
  - Github Account
  - Click on the "+" icon in the top-right corner and select "New repository."
- Link Local Repository to Remote Repository
  - Copy the URL of your newly created remote repository.

```
git remote add origin <your remote_repository_url>
```

Pushing to local repository

- Push Local Changes to Remote Repository

```
git push -u origin master
```

- Syncing local repo with remote changes
  - Fetch Remote Changes

```
git fetch origin
```

  - Merge Remote Changes with Local Branch

```
git merge origin/master
```

Branches in Git

- Within Git, a branch functions as a distinct and adaptable reference point within the commit history. This enables developers to work on various features or project iterations simultaneously, maintaining a clear separation from the primary codebase. Branches contribute to efficient development by isolating changes, facilitating experimentation with new features, and fostering a collaborative environment.

Branches in Git

- Create Git Branch

```
git branch <issue/feature_branch_name>
```

- Switch Between Branches

```
git checkout <issue/feature_branch_name>
```

- Merging Branches

```
git merge issue/feature
```

GitHub's Directed Acyclic Graph (DAG)

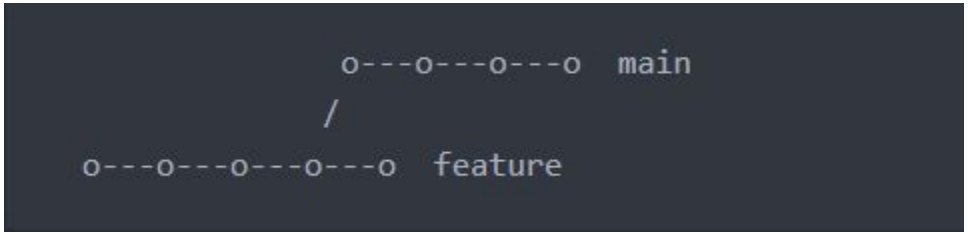
Imagine GitHub's DAG as a family tree for your code. It visually shows how changes (commits) are connected, branching out and merging back together over time. This helps you understand the history of your project.

Understanding branches and the DAG is crucial for working together on projects using Git. It keeps everyone organized and allows you to track the history of your code easily.

D.A.G cont.

- Imagine your code as a growing document.
  - **Commits:** These are like snapshots you take at different points in time, capturing the state of your code at that moment.
  - **Branches:** Think of these as forks in the road. You can create a branch to work on a new feature or fix a bug without affecting the main code.
  - **Merges:** When you're happy with your changes on a branch, you can merge them back into the main codebase, combining the work you've done.

D.A.G Example



- Each circle (`**o**`) represents a commit.
- The `**main**` branch has four commits, and the `**feature**` branch has five commits.
- The branch pointers (`**main**` and `**feature**`) point to the latest commits in each branch.
- There is a merge commit where changes from the `**feature**` branch were merged into the `**main**` branch.

Summary

- **Branches**: These are like separate workspaces where you and your teammates can make changes without affecting each other's work. You can experiment with new features or fix bugs in your branch without messing with the main project.

- **GitHub's DAG (like a project history map)**: This shows you a visual timeline of all the changes made (commits) and how they're connected. It's like a map of your project's history, showing how branches connect and merge back together.

Tips and Best Practices

1. **Branch Tracking:** It's helpful to set up tracking between your local branches and remote branches. This allows Git to automatically determine the remote branch when pushing or pulling changes.
2. **Review Changes Before Pushing:** Always review your changes before pushing them to the remote repository to ensure they are correct and meet project standards.
3. **Pull Before Push:** It's a good practice to pull changes from the remote repository before pushing your changes. This helps avoid conflicts and ensures that you're working with the latest codebase.
4. **Authentication:** Ensure that you have the necessary permissions and authentication configured to push changes to the remote repository.
5. On GitHub, you can visualize the DAG by navigating to the "Insights" tab of a repository and selecting "Network." This displays a graphical representation of the repository's history, including branches, commits, and merges.

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar