



CoGrammar

Lecture 16: Modules



**SKILLS
FOR LIFE**

SKILLS BOOTCAMPS



Department
for Education

Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(FBV: Mutual Respect.)
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes.
You can submit these questions here: [Open Class Questions](#)

Lecture Housekeeping cont.

- For all **non-academic questions**, please submit a query:
www.hyperiondev.com/support
- Report a **safeguarding** incident:
www.hyperiondev.com/safeguardreporting
- We would love your **feedback** on lectures: [Feedback on Lectures](#)


Lecture Objectives


- **Learn how to setup a professional project**
- **Learn about Git strategies**
- **Learn how to organize files in your project**



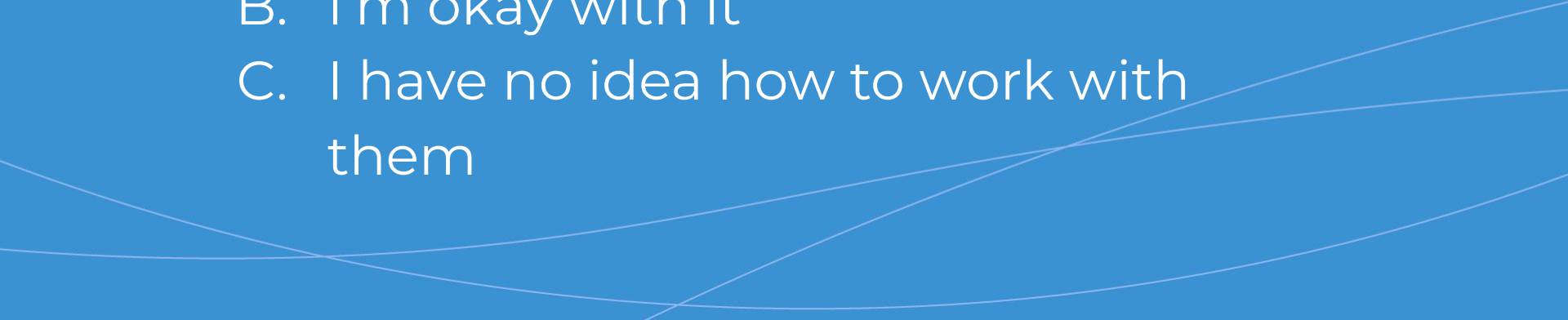
How often are you making use of Git in your projects?



- A. I use it in every project
 - B. I use it for big projects
 - C. I never use it.
- 




How comfortable are you when it comes to working with external modules?

- A. Very comfortable
 - B. I'm okay with it
 - C. I have no idea how to work with them
- 



How familiar are you with PEP8 standards?

- A. Very familiar
 - B. Somewhat familiar
 - C. What are those?
- 

Recap

Last week we went over designing our systems and making use of class diagrams and MVC, in this session, we are going to build on that knowledge and see how we can implement all of this in our actual applications.

Writing Professional Code

Having an understanding of data structure and algorithms, OOP and testing is important, but all of this knowledge can be obscured when the code looks unprofessional, there are a few other things that we need to think about as software engineers

1. Development Environment
2. Dependency management
3. Version Control
4. File Structure
5. Python Linters

Development Environment

As developers, we work with a lot of different packages and modules in our projects, it's important to think about a few things:

- How are we keeping track of the dependencies in our application
- Will other developers in our team be able to emulate our development environment
- Are the versions of the tools we want to work with the same as the ones installed on our machine.

Note: Although you might not have come across these issues yet, you will at some point in your career.

VENV

Python comes with a built-in environment tool that allows us to manage our dependencies on a project by project basis, the module allows us to:

- Install modules and packages in an isolated environment
- Makes it easier to reproduce the same environment on different machines without causing conflicts with local modules and packages
- Safer experimentation since each package is isolated allowing us to test new and potentially unstable modules without overwriting local stable modules.

Setting Up VENV

Windows

Create

```
python -m venv .venv
```

Activate

```
.venv/Scripts/activate
```

Deactivate

```
deactivate
```

Mac/Linux

Create

```
python3 -m venv .venv
```

Activate

```
.venv/bin/activate
```

Deactivate

```
deactivate
```

Dependency Management

Virtual Environments should be specific to a single users machine, but the same dependencies still need to be accessible on different machines.

To make sure that we can recreate environments, we need to use dependency management

- Allows us to store the name and version of the packages that we are using
- Provides a file that can be uploaded to our version control repository
- Allows other developers to run a command to install all of the dependencies in their own virtual environments

Dependency Management

Windows

```
pip freeze > requirements.txt
```

Mac/Linux

```
pip3 freeze > requirements.txt
```

Version Control

Version control is a very important part of any project, it allows you to keep track of the different stages of your application.

- Allows us to track and manage our applications
- Makes it easy to build features in parallel to our deployed application
- GitHub allows us to have a central point of truth
- GitHub connects with major cloud providers making deployments really easy

Setting Up Git

1. Initialize Git
2. Before doing a commit, make sure that you have created a `.gitignore` file
 - a. You can copy the content [here](#)
3. Create an initial commit
4. If applicable, push to remote repository

Setting Up Git

```
git init
```

```
touch .gitignore / ni .gitignore
```

```
git add .
```

```
git commit -m "inital commit"
```

```
git push
```

Git Strategy

Using Git is easy, but using Git correctly requires a little bit more thought to prevent the many issues that you might run into. Creating strategies for different operations will make it easier to manage your applications and git implementation.

- Branching
- Commit
- Push/Pull/Fetch

Branching Strategy

Branches are a really powerful tool in our Git toolbox, the way we interact with them can make a lot of different operations more efficient, especially when working on large systems.

Some types of branches that you can include are: :

- **Feature/Issue branches**, these allow us to improve our application without having to interrupt the main application until we are ready to release the new feature or fixes.
- **Release branches**, these allow us to work on different versions of the same application at the same time

NOTE: These are just representations of what the branches can be used for and not a list of actual different types of branches

Commit Strategy

Commits can be complex to handle once you get into a large team, but there are a few different approaches that you can take depending on your circumstances.

- **Solo Developer**

- It's a good idea to commit frequently so make sure that you have a granular commit history
- Since no one else will be working on the repository as well, it's easy to keep track with the latest commit

- **Working on a team**

- Working on individual branches is the most important first step
- Frequent commits on an individual branch can be achieved, but to avoid merge conflicts, you will need to squash the merge so that it acts like a single commit

Note: ``git merge -squash feature-branch`` is used for merging a single branch as a single commit

Push/Pull Strategy

Staying up to date with the code is really important, and depending on the size of the team that you are working on, you will need to be able to keep your application up to date at the correct intervals to prevent issues.

- **Solo Developer**

- It's a good idea to push as you commit to make sure that your code is always updated on the cloud
- If you are working on different machines, after following the first point, you should pull as soon as you open the code to make sure that both machines are in sync

- **Working on a team**

- Teams will usually have a schedule for merging code to the main branch, so perform your pulls according to the schedule.
- Perform **Fetch** requests regularly to make sure that all other branches are up to date without overwriting your local changes.
- Make sure that you are pushing your feature branch with each commit.

File Structure

Once we have set up our environments, we need to make sure that we are also structuring our code correctly

- A good structure groups related files together, making it easier to navigate our code
- Promotes a good separation of concern
- Makes working in teams a lot easier as it's less likely that each developer will be working on the same file.

Example File Structure

```
.
├── my_project/
│   ├── .git/
│   ├── .venv/
│   ├── documentation/
│   ├── src/
│   │   ├── models/
│   │   ├── views/
│   │   ├── controllers/
│   │   ├── services/
│   │   └── repositories/
│   ├── tests/
│   ├── .gitignore
│   ├── .env
│   ├── main.py
│   ├── utilities.py
│   ├── constants.py
│   └── README.md
```

Root File

.git/

Manages git for our project

.venv/

Manages our virtual environment

Documentation/

Stores our application documentation

src/

Stores our code and functionality

test/

Stores our unit tests

.gitignore

Files to exclude in our Git operations

.env

Stores our application secrets

main.py

The main entry point for our application

utilities.py

Operations that we want to share throughout the application

constants.py

Constant variables that need to be used throughout the application.

README.md

Information about our repository

src/

models/

- Stores our model classes, used to represent database tables
- Connects to all of the other files.

views/

- Stores our code files for UI views
- Connects to controllers

controllers/

- Stores our files for controllers
- Connects to our services

services/

- Stores the code for our business logic
- Connects to our repositories

repositories/

- Stores the code for connecting to our data sources

Ground Up Approach

File structures can be overwhelming when you first get into it, but it's important to start off simple and build on that.

Excluding our Git and environment files, we can start of with the following:

```
.  
├── my project/  
│   ├── main.py  
│   ├── file_access.py  
│   ├── user_model.py  
│   ├── user view.py  
│   └── user service.py
```

Ground Up Approach

Down the line, we might need a *task_model.py* file as well, so at this point, knowing that having two models in a the same directory would be messy, we can clean it up.

This single change can be extended to the other potential groups and we'd end up with the following:

```
.
├── my_project/
│   ├── models/
│   │   ├── user_model.py
│   │   └── task_model.py
│   ├── views/
│   │   └── user_view.py
│   ├── services/
│   │   └── user_service.py
│   ├── repositories/
│   │   └── file_access.py
│   └── main.py
```

Linting

Linters are an important tool for standardizing our code and doing some errors checks for us.

- Enforce coding standards
- Catches errors early
- Reduces technical debt by catching potential errors early
- Code reviewers can focus on just the logic, design and functionality

Let's Breathe

Let's take a small break before moving on to the next topic.

CoGrammar

Q & A SECTION

**Please use this time to ask
any questions relating to the
topic, should you have any.**



CoGrammar

Thank you for joining!