



Welcome to this **CoGrammar** session:

OOP - Classes and ORMs

The session will start shortly...

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



Software Engineering Session Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.
(Fundamental British Values: Mutual Respect and Tolerance)
- 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 Academic Sessions. You can submit these questions here: [Questions](#)

Software Engineering Session 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](#)

Software Engineering Session Housekeeping cont.

- "Please check your spam folders for any important communication from us. If you have accidentally unsubscribed, please reach out to our support team."
- Career Services, Support, etc will send emails that contain NB information as we gear up towards the end of the programme. Students may miss job interview opportunities, etc.

Skills Bootcamp

8-Week Progression Overview

✓ Criterion 3: Course Progress

- **Completion:** All mandatory tasks, including Build Your Brand and resubmissions by study period end
- **Interview Invitation:** Within 4 weeks post-course
- **Guided Learning Hours:** Minimum of 112 hours by support end date (10.5 hours average, each week)

✓ Criterion 4: Demonstrating Employability

- **Final Job or Apprenticeship Outcome:** Document within 12 weeks post-graduation
- **Relevance:** Progression to employment or related opportunity

Learning Outcomes

- Explain what a class is in Python
- Create your own classes within your projects
- Explain the role of inheritance in classes
- Implement inheritance within your own projects
- Explain what an ORM is
- Explain database migration
- Implement the use of an ORM within your own projects

Classes

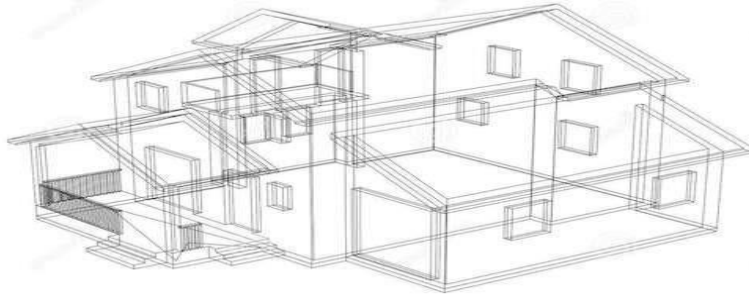


What is Object Oriented Programming?

- OOP is a way of organising and structuring code around objects, which are self-contained modules that contain both data and instructions that operate on that data.

Classes and Objects

- A class is a blueprint or template for creating objects. It defines the attributes and methods that all objects of that class will have.



- An object is an instance of a class. Objects are created based on the structure defined by the class.

Attributes

- Attributes are values that define the characteristics associated with an object.
- They define the state of an object and provide information about its current condition.
- For a class named 'House', some relevant attributes could be:
 - Number of bedrooms
 - Year built

Methods (Behaviours)

- Methods, also known as behaviours, define the actions or behaviours that objects can perform.
- They encapsulate the functionality of objects and allow them to interact with each other and the outside world.
- For a class named 'House', some relevant method could be:
 - `set_location()`: Allows updating the location of the house

Constructor

- A constructor is a **special method** that gets called when an object is instantiated. It is used to **initialise the object's attributes**.

```
def __init__(self, name, age, graduated):  
    self.name = name  
    self.age = age  
    self.graduated = graduated
```

Creating a Class

- `__init__()` method is called when the class is instantiated.

```
class Student:  
    def __init__(self, name, age, graduated):  
        self.name = name  
        self.age = age  
        self.graduated = graduated
```

Class Instantiation

- This Class takes in three values: a **name**, **age** and **graduation status**.
- When you instantiate a class, you **create an instance** or an object of that class.

```
luke = Student("Luke Skywalker", 23, True)
```

Creating and Calling Methods

- `change_location()` method is called below:

```
class House:

    def __init__(self, location):
        self.location = location

    def change_location(self, new_location):
        self.location = new_location

house = House("London")
house.change_location("Manchester")
```


Access Control - Attributes

- Access control mechanisms (`public`, `protected`, `private`) restrict or allow the access of certain attributes within a class.

```
class MyClass:
    def __init__(self):
        # Public attribute
        self.public_attribute = "I am public"

        # Protected attribute (by convention)
        self._protected_attribute = "I am protected"

        # Private attribute
        self.__private_attribute = "I am private"
```


Access Control - Methods

- Access control mechanisms (`public`, `protected`, `private`) can also restrict or allow the `access of certain methods` within a class.

```
def public_method(self):  
    return "This is a public method"  
  
def _protected_method(self):  
    return "This is a protected method"  
  
def __private_method(self):  
    return "This is a private method"
```

Applying the Access Control

```
# Create an instance of MyClass
obj = MyClass()

# Accessing public attributes and methods
print(obj.public_attribute)      # Output: I am public
print(obj.public_method())       # Output: This is a public method

# Accessing protected attributes and methods (not enforced, just a convention)
print(obj._protected_attribute)  # Output: I am protected
print(obj._protected_method())   # Output: This is a protected method

# Accessing private attributes and methods (name mangling applied)
# Note: It's still possible to access, but it's discouraged
print(obj._MyClass__private_attribute)  # Output: I am private
print(obj._MyClass__private_method())   # Output: This is a private method
```

Inheritance



What is Inheritance?

- Sometimes we require a class with the same attributes and properties as another class but we want to extend some of the behaviour or add more attributes.
- Using inheritance we can create a new class with all the properties and attributes of a base class instead of having to redefine them.

Implementing Inheritance

- Parent/Base class

- The parent or base class contains all the attributes and properties we want to inherit.

```
class BaseClass:  
    # Base class definition  
  
class SubClass(BaseClass):  
    # Derived class definition
```

- Child/Subclass

- The sub class will inherit all of its attributes and properties from the parent class.

ORM



CoGrammar

Traditional Database Interaction

- **Manual SQL:** Writing SQL queries directly to interact with databases.
 - Example:
 - **SELECT * FROM** users **WHERE** age > 30;
 - Drawbacks: Verbose, error-prone, and tightly coupled to database specifics.

ORM

- **ORM (Object-Relational Mapping):**
 - A technique to interact with databases using an object-oriented programming (OOP) language.
 - Simplifies database operations by mapping database tables to class structures in code.
 - We'll use **SQLAlchemy** ORM tool

ORM

```
# Define the models
class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    profile = relationship(back_populates="user")
    comment = relationship(back_populates="user")
```

User
Python
Class

ORM

Mapper

id	name
1	John Doe
2	Mark Henry

User Table

ORM as an Abstraction Layer

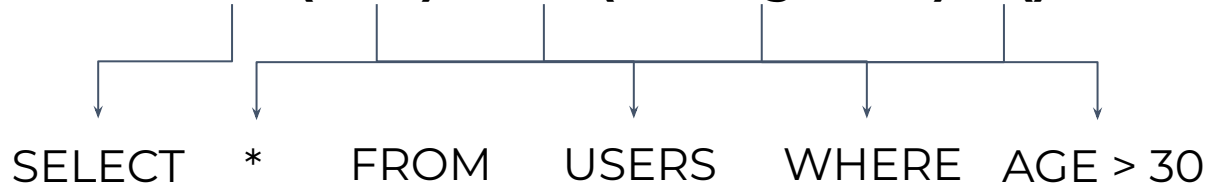
- **Abstraction:**
 - Provides a layer between application code and the database.
 - Enables developers to work with high-level objects instead of raw SQL.

ORM as an Abstraction Layer

```
session.execute(select(User).where(User.age == 'John Doe')).all()
```

○ Example:

■ **`select(User).where(User.age > 30).all()`**



A diagram mapping the ORM code `select(User).where(User.age > 30).all()` to the SQL query `SELECT * FROM USERS WHERE AGE > 30`. Vertical lines connect the code segments to the SQL keywords: `select` to `SELECT`, `*` to `*`, `where` to `WHERE`, `User` to `USERS`, and `age > 30` to `AGE > 30`. The `FROM` keyword is present in the SQL query but has no corresponding code segment. An arrow from the `session.execute()` call in the top code block points to the `all()` method in this example.

SELECT * FROM USERS WHERE AGE > 30

Benefits of Using ORM

- **Easier Data Handling:**
 - Write less code and avoid SQL syntax errors.
 - Focus on business logic rather than database details.
- **Security:**
 - Reduces risk of SQL injection attacks.
 - Automatically escapes query inputs.
- **Code Readability and Maintenance:**
 - Improved readability with clear class definitions and relationships.
 - Easier to maintain and update as database schema evolves.
- **Productivity:**
 - Faster development with less boilerplate code.
 - Auto-generates SQL queries from code.

Migrations


Migrations are a way of propagating changes you make to your models (adding a field, deleting a model, etc.) into your database schema. They're designed to be mostly automatic, but you'll need to know when to make migrations, when to run them, and the common problems you might run into. [Django's Migrations](#)

Migrations

Let's assume that you have this Python Class, we need to add **age**:

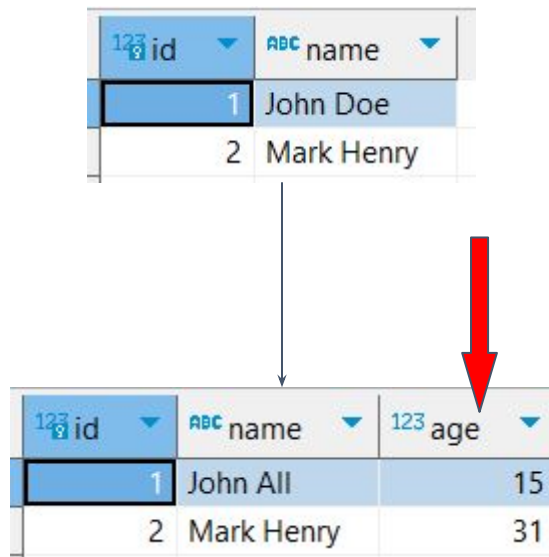
```
class User(Base):
    __tablename__ = 'users'
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    name: Mapped[str] = mapped_column(String, nullable=False)
    profile: Mapped['Profile'] = relationship(back_populates="user")
    comment: Mapped['Comment'] = relationship(back_populates="user")
```

```
class User(Base):
    __tablename__ = 'users'
    id: Mapped[int] = mapped_column(Integer, primary_key=True)
    name: Mapped[str] = mapped_column(String, nullable=False)
    age: Mapped[int] = mapped_column(Integer, nullable=False)
    profile: Mapped['Profile'] = relationship(back_populates="user")
    comment: Mapped['Comment'] = relationship(back_populates="user")
```



Migrations

Now we have the new column age **added**:



The diagram illustrates a database migration. It shows two tables. The top table has two columns: 'id' and 'name'. The bottom table has three columns: 'id', 'name', and 'age'. A red arrow points to the new 'age' column, indicating its addition. The data in the 'name' column has been updated to reflect the migration.

id	name
1	John Doe
2	Mark Henry

id	name	age
1	John All	15
2	Mark Henry	31

SQLAlchemy

SQLAlchemy

- **SQLAlchemy** is a library used to interact with a wide variety of databases. It enables you to create data models and queries in a manner that feels like normal Python classes statements.
- SQLAlchemy needs to be installed first. To do so, we have to use following code at Terminal or CMD.

pip install sqlalchemy

**Let's take a short
break**



Questions and Answers



Summary

- **Classes:** Allow us to encapsulate data and methods that operate on the data within a single unit (class), hiding details.
- **Inheritance:** We can inherit properties and attributes from other classes allowing us to reuse and extend the code.
- **ORM Basics:** Simplify database interaction by mapping objects to tables (and vice versa).
- **CRUD Operations with ORMs:** Easily Create, Read, Update, and Delete data using ORM functionalities.
- **Model Relationships:** Manage connections between tables using one-to-one, one-to-many, and many-to-many relationships.
- **ORM Models:** Define database structure with Python classes, representing tables and their columns.
- **Database Migrations:** Keep your database schema in sync with your Python models using migrations.

Thank you for attending



Department
for Education

CoGrammar

