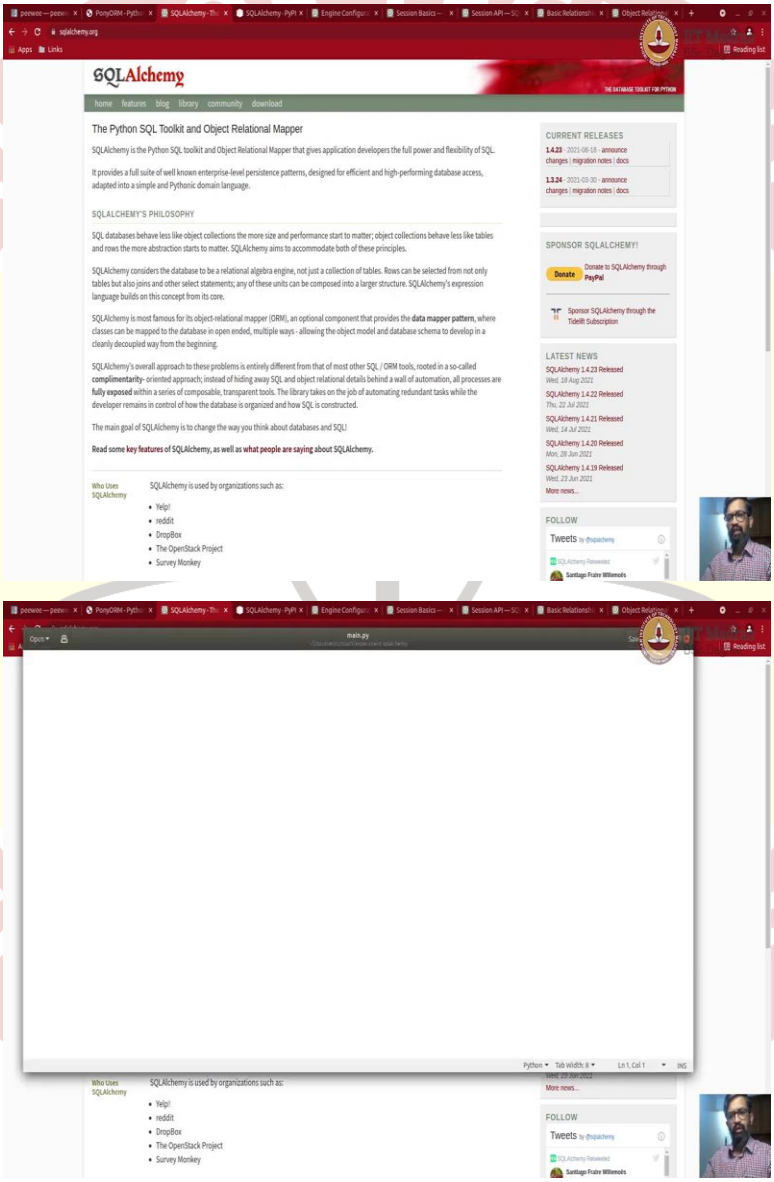
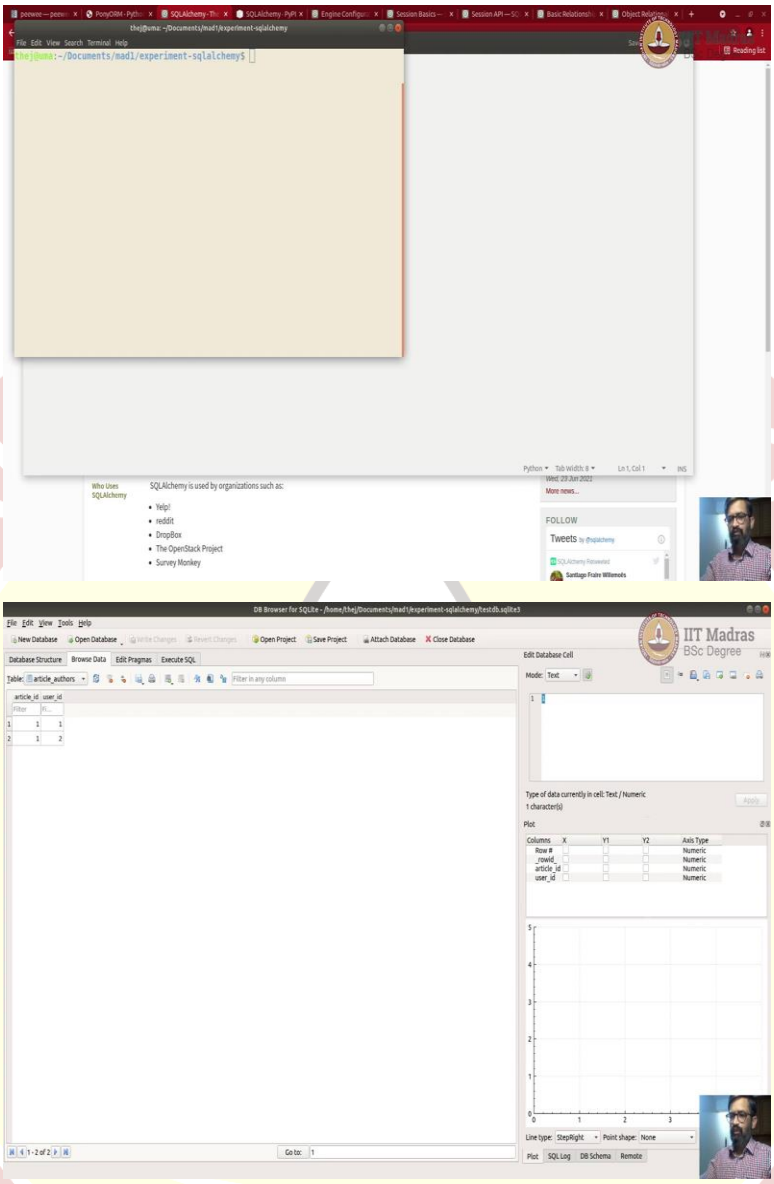
IIT Madras

ONLINE DEGREE

Modern application Development – I
Professor. Thejesh G N
Software Consultant
Indian Institute of Technology, Madras
How to Query Database using SQL Alchemy?

(Refer Slide Time: 0:15)



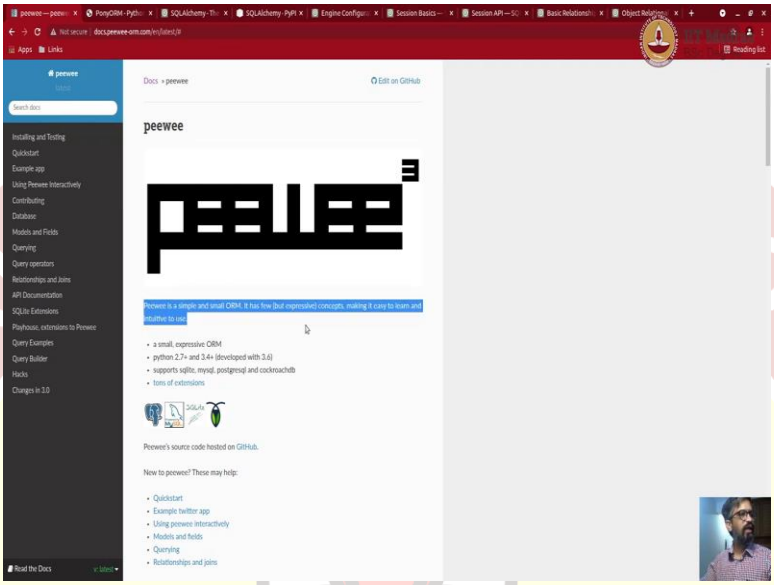


Welcome to the modern application development screencasts. In this short screencast, we will learn how to connect and query a database using SQL alchemy. Before we start, open the following applications on your Ubuntu desktop. So, that you can work along with me. Browser, preferably Firefox, Chromium or Brave is also fine, G edit text editor. Any other text editor would be fine. But G edit comes by default in your Ubuntu. Terminal or default terminal is fine. Please also make sure that you have Python 3 installed on your machine.

We would also need a DB browser, like we saw in the last SQL lite screencast to manipulate the database and work with it. In simple words, object relational mapping is a technique that allows the developer to query and manipulate their databases using object oriented paradigm. So,

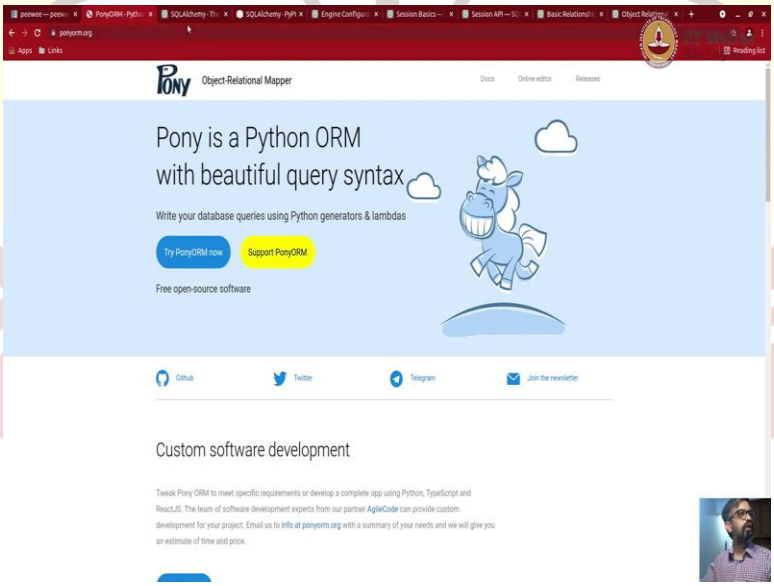
accessing the database becomes as easy as accessing classes and objects in Python. There are many toolkits available in Python world.

(Refer Slide Time: 1:36)



Some of the known ones are, for example PEEWEE, it is quite small and expressive.

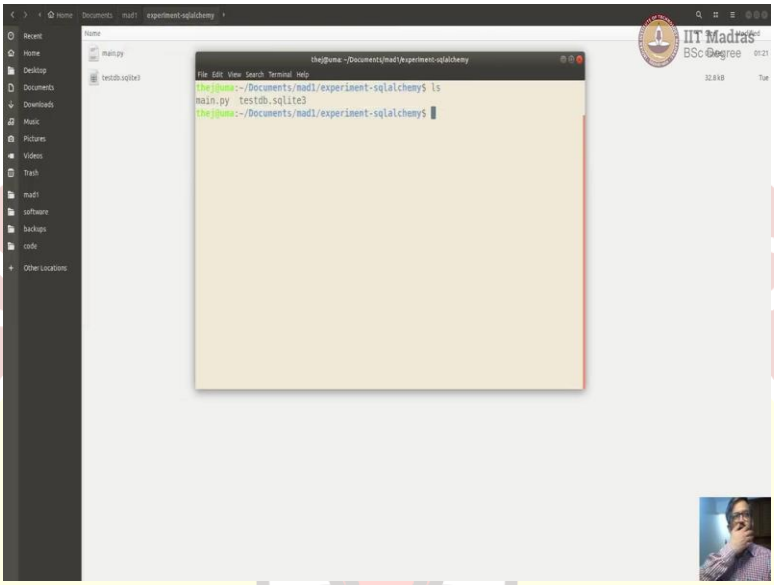
(Refer Slide Time: 1:37)



There is another one called Pony. And of course, there is an SQL alchemy. The most common one is SQL alchemy. SQL alchemy exposes a standard programming interface that is database agnostic. This helps us to write database agnostic code, which means we write code for one

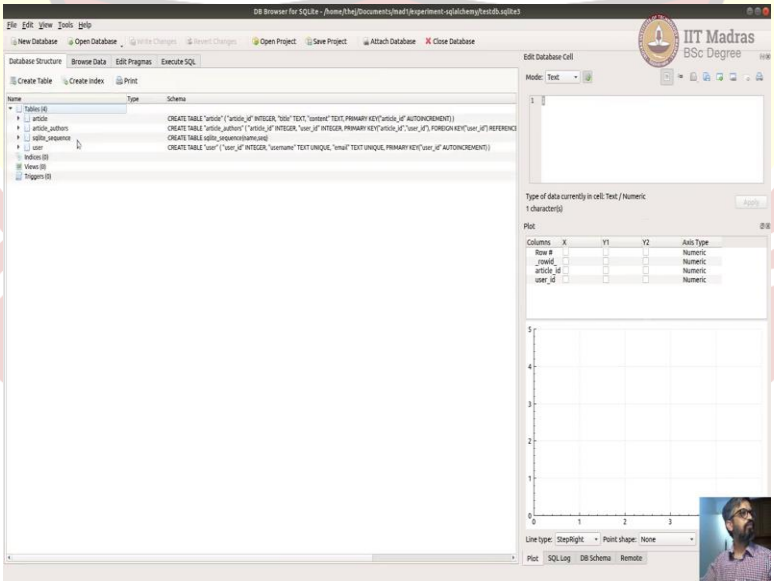
database, you should be able to change the database, and it should still work in most cases. Let us start with our experiment.

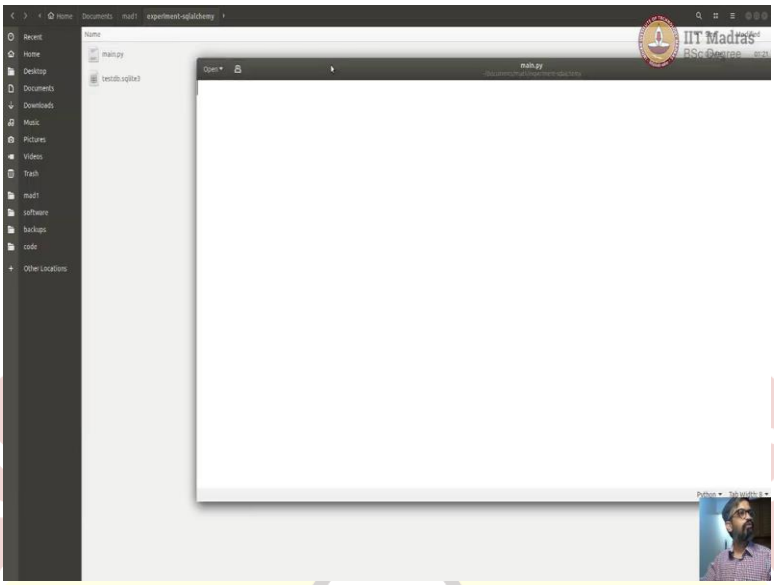
(Refer Slide Time: 2:11)

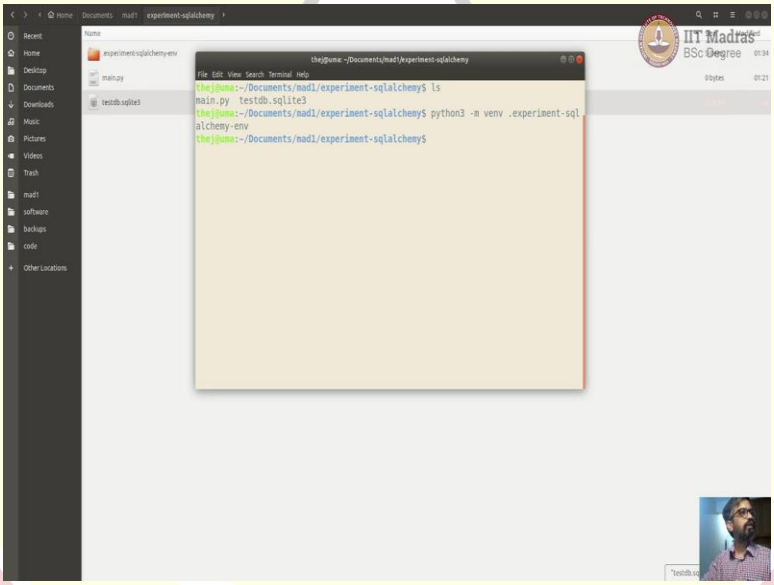


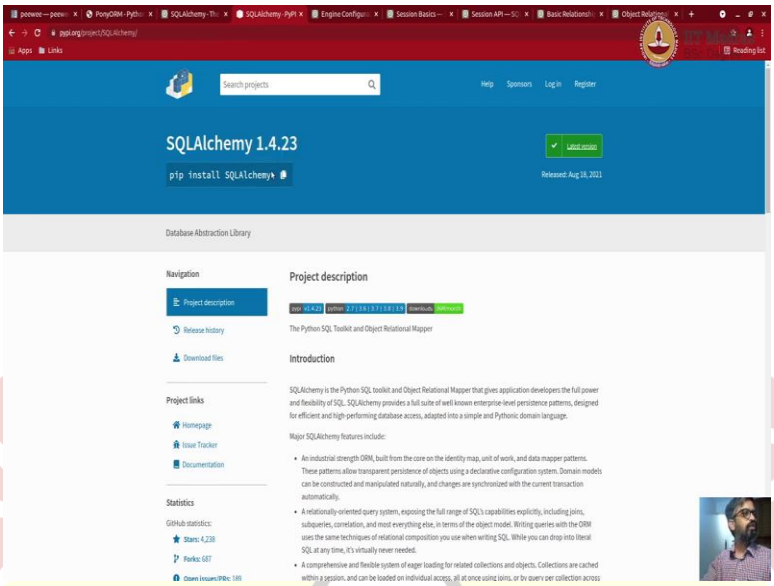
We have a folder called experiment-sqlalchemy. I have opened the command prompt, I have already seed it into that folder.

(Refer Slide Time: 2:29)




Home
Documents
mad1
experiment-sqlalchemy
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
mad1
software
backups
code
Other Locations
Name
main.py
testdb.sqlite3
Open
main.py
IIT Madras
BSc Degree


Home
Documents
mad1
experiment-sqlalchemy
Recent
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
Trash
mad1
software
backups
code
Other Locations
Name
experiment-sqlalchemy-env
main.py
testdb.sqlite3
theJguna: ~/Documents/mad1/experiment-sqlalchemy
File Edit View Search Terminal Help
theJguna:~/Documents/mad1/experiment-sqlalchemy$ ls
main.py testdb.sqlite3
theJguna:~/Documents/mad1/experiment-sqlalchemy$ python3 -m venv .experiment-sql
alchemy-env
theJguna:~/Documents/mad1/experiment-sqlalchemy$
IIT Madras
BSc Degree

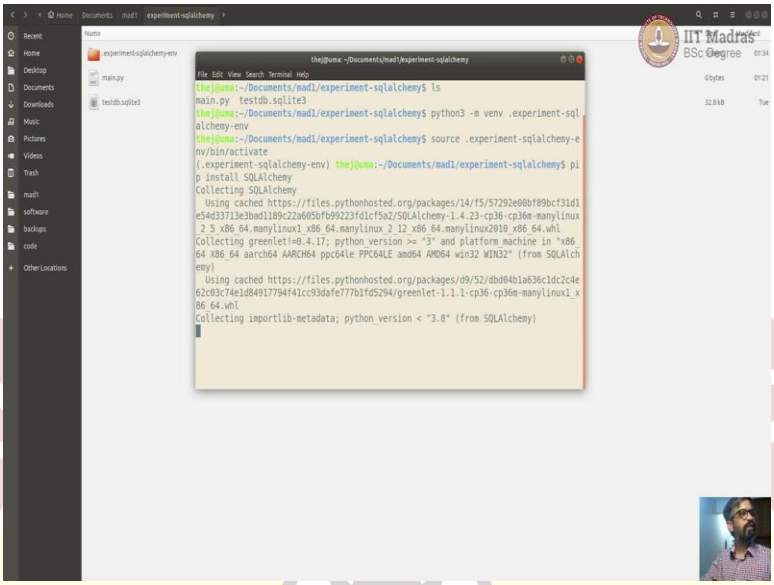


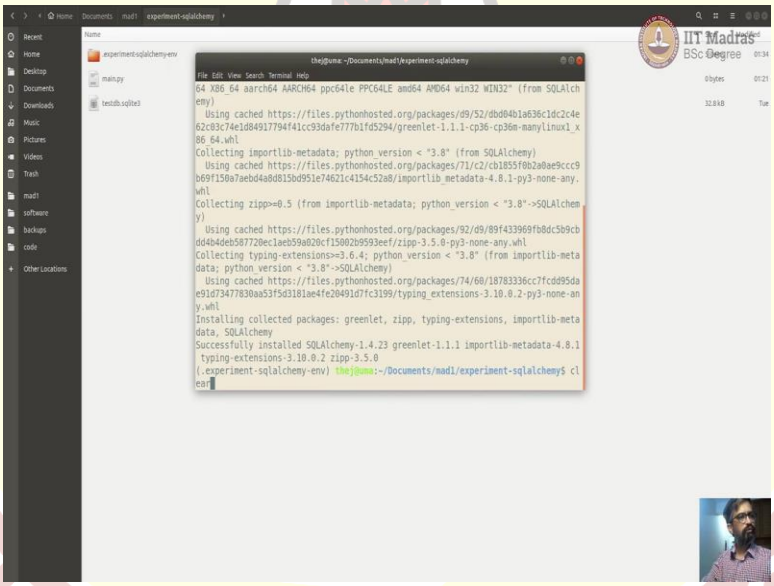
I now got the database from our last experiment opened. And in, in DB browser, you can see the same database and it is the same database here. And I have a main.py, which I have opened in, in a G edit which is empty. We are going to use this database that we created last time to experiment. We are going to use the same tables, that we created last time. If you are not seeing that, please watch it before you start this.

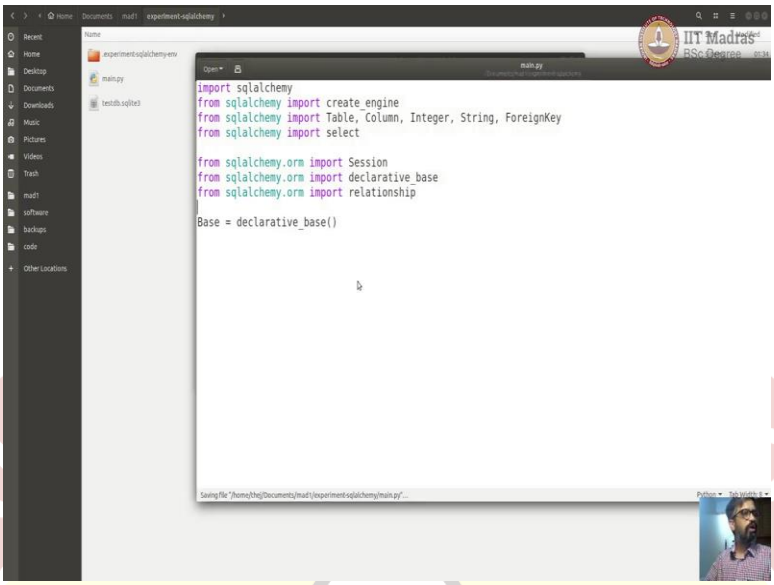
Let us start with the, creating a virtual environment. I am just going to paste the code to create the virtual environment. I am guessing you already know how to do this by now. It is already created. Now we need to install SQL alchemy. You can go to sqlalchemy.org see the latest version.

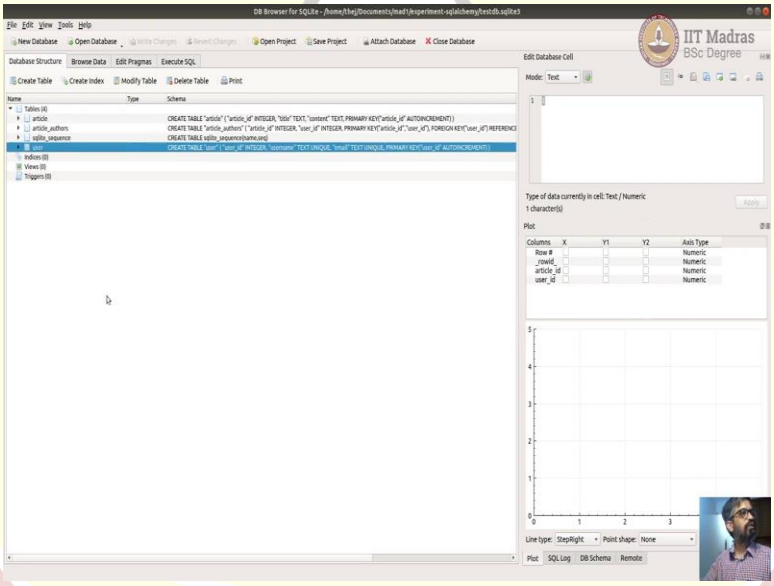
It is the latest version 1.4.23. You can also go to pypi site and see the latest version of the package. And here is the installation command. I think the latest version is 1.4.23, we will use the same thing. I am just going to copy this and install.

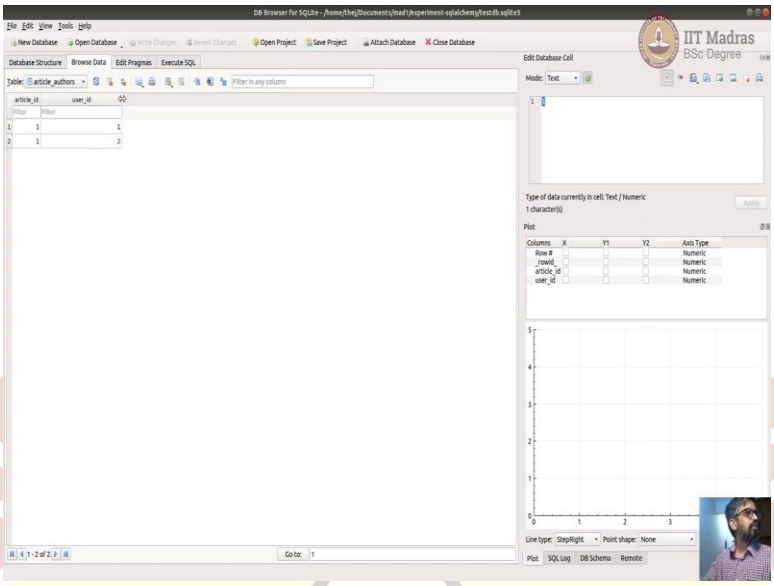
(Refer Slide Time: 3:42)






IIT Madras
BSc Degree
main.py
import sqlalchemy
from sqlalchemy import create_engine
from sqlalchemy import Table, Column, Integer, String, ForeignKey
from sqlalchemy import select
from sqlalchemy.orm import Session
from sqlalchemy.orm import declarative_base
from sqlalchemy.orm import relationship
Base = declarative_base()


IIT Madras
BSc Degree
DB Browser for SQLite - /home/thej/Documents/mad1/experiment-sqlalchemy/testdb.sqlite3

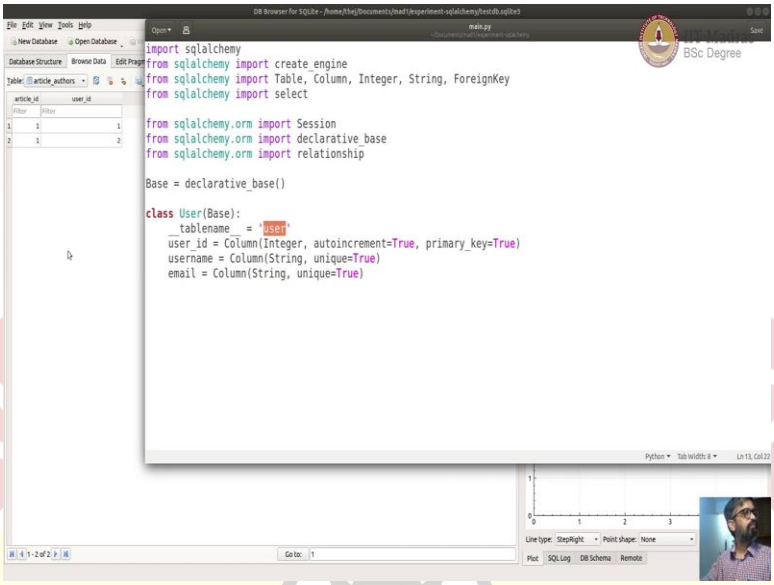


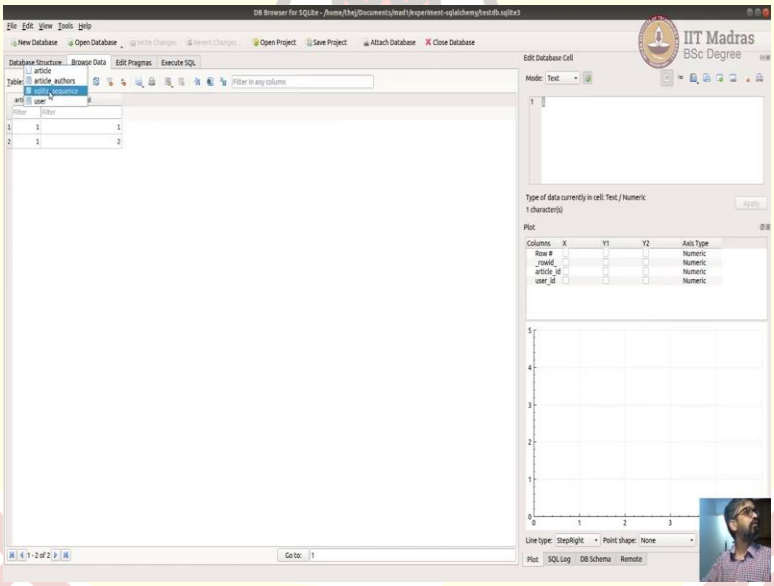
Before we install, we have to enable the virtual environment. So, I am just going to source it, bin activate. So, it is source now. Now I am going to install SQL alchemy, might take couple of minutes. Based on your internet connection installed. I am just going to clear, to install in our virtual environment. Now we are just going to edit the main point pi main.py and do our experimentation.

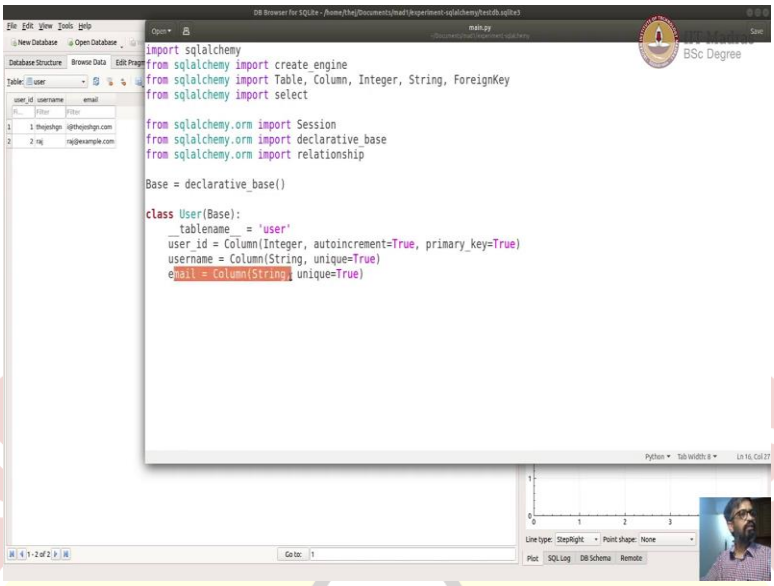
Before we start, I am just going to paste some of the imports. Instead of pasting one by one, I am just going to paste, some of them at once. Here I am just importing some of the classes that are required to do the experiment. Before we begin, actually we need to start writing the models for the tables that we have, that will be the first step. So, I am just going to get the base class for the model.

And write, models for all over tables. Like if you go to a database we have three tables, article, article_authors and user. Article is one which contains article, user contains all the user, article authors contains a link table, which is between the article table and the user table, which has the article_authors. It is a many to many relationship between article and user, that you can see here. article_id and user_id. Now let us create the models. I am just going to paste it and take you through it. Let us write for the user first.

(Refer Slide Time: 5:40)



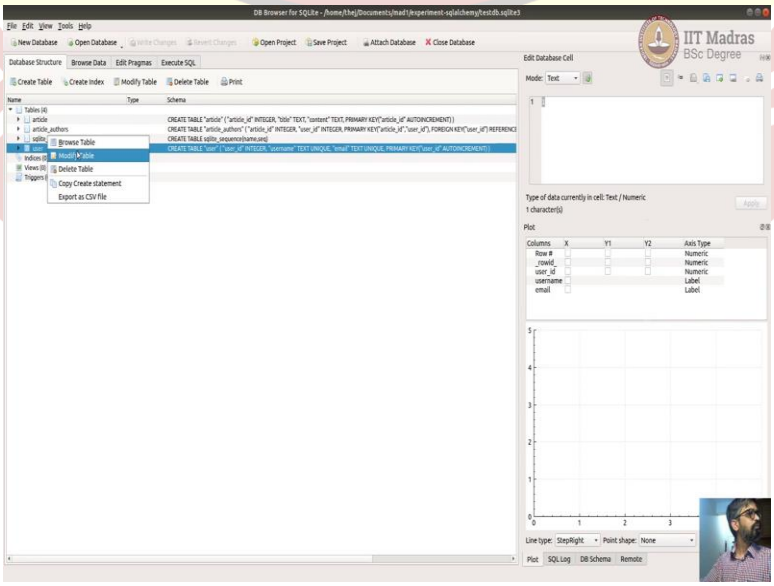


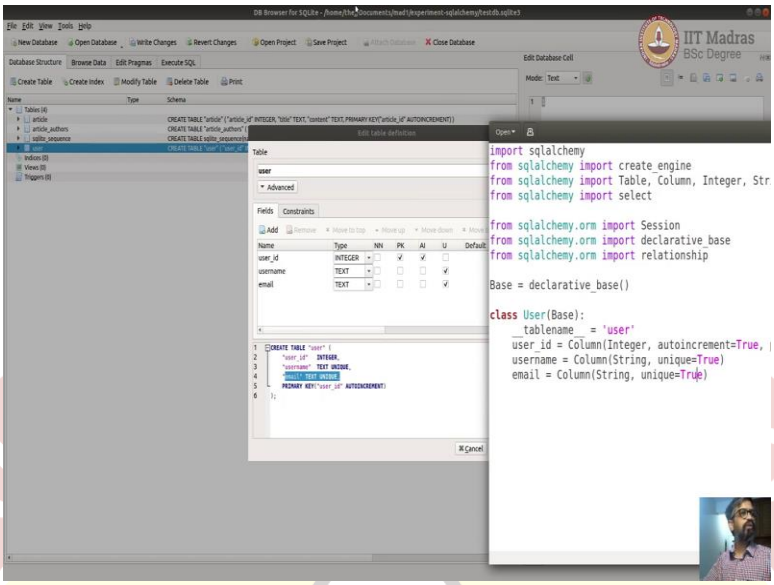


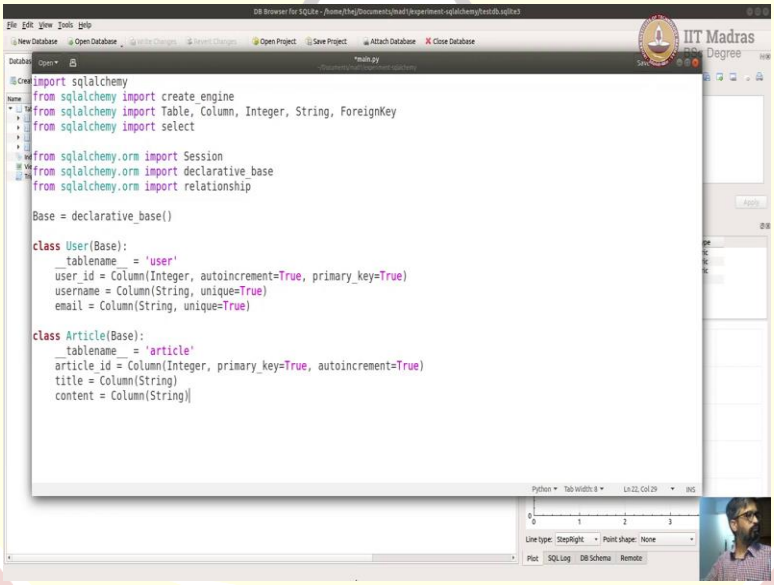
Here this is the user model, it extends the base and then the table name is given here. This should be the same table name as in your database. Here this user in lower caps, the same thing here. This could be anything, the class name could be anything. I am just going to use capital or camel case user. And it has three columns.

As you can see user_id, username and email. Just going to use that same name as in the database. And then we are defining the column as an integer, auto increment is true and it is a primary key. Similarly, user name is column, it is a string or a text and the uniqueness is going to true. Similarly email, it is a string and uniqueness is going to true.

(Refer Slide Time: 6:36)

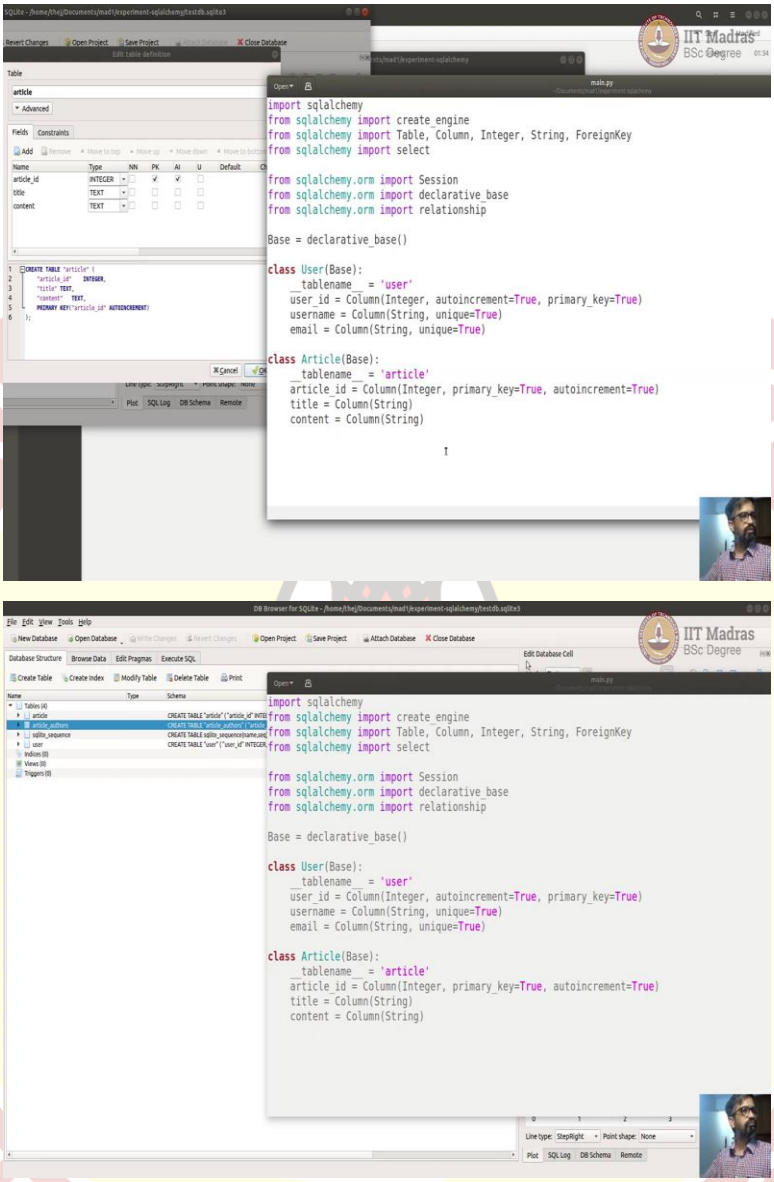






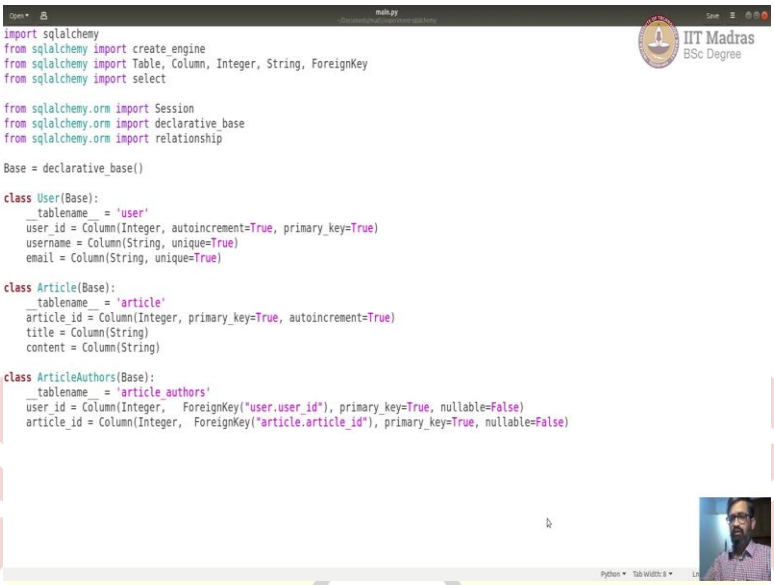
You can double check it here, go to modify it, yeah, yup. This should match, user and user, so table name matches. User_id, auto increment true and primary key correct username is unique and email is unique, username is unique, (())(06:56) is unique, to create it for user. Now let us go ahead and create an article.

(Refer Slide Time: 07:15)



So, here is article, let us open this table details over here. So, here it is, you can see it has three columns article, title and content article title and content. This is integer and auto increment primary key, you can see it here. It is defined here and you can also see it here. Same thing we should define here. Now, the next we are going to define, this article authors.

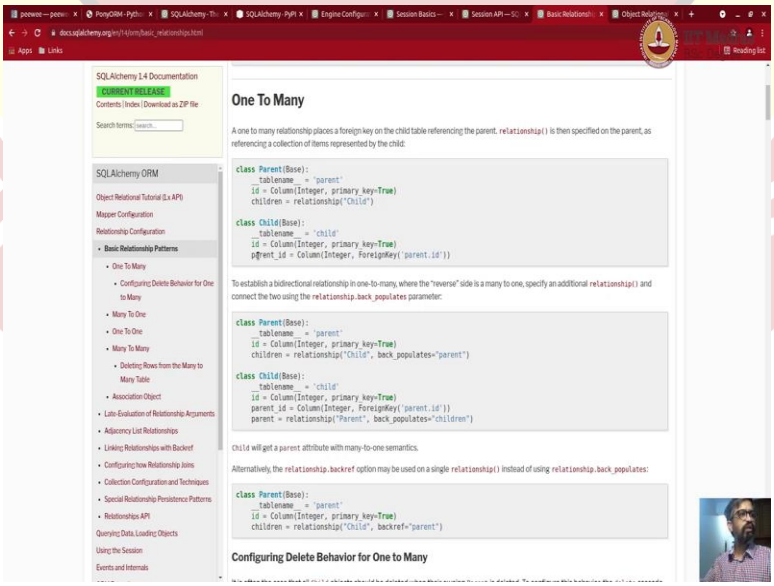
(Refer Slide Time: 7:52)

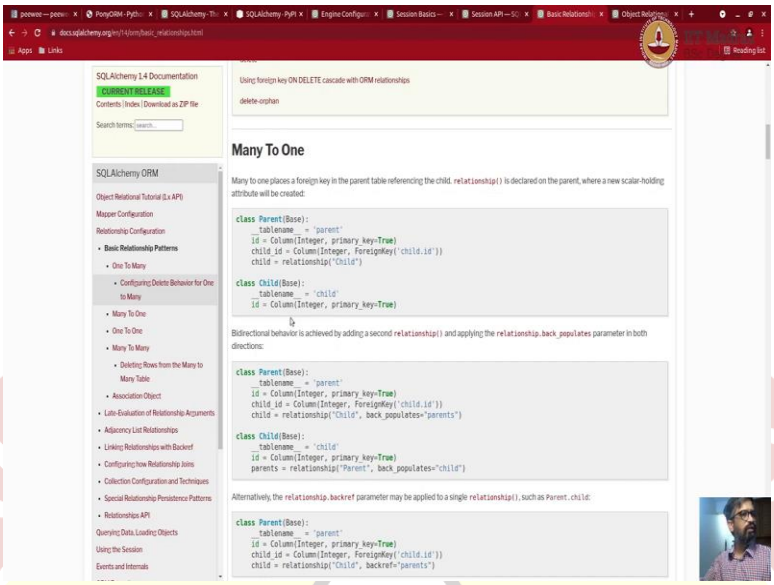


article_author extends base table in this case, user_id is a foreign key to user.user_id this. And it is a primary key and nullable is going to false. Similarly, article id is a primary key to article table, here. And it is a primary key true, nullable is false. This is a composite key, because both are primary keys. Now this is done.

Now, we have to define many to many relationship between users and articles. Basically, whenever I get article, I want to get users too, so we can define many to many relationship.

(Refer Slide Time: 8:32)





Using foreign key ON DELETE cascade with ORM relationships

delete-orphan

Many To One

Many to one places a foreign key in the parent table referencing the child. `relationship()` is declared on the parent, where a new scalar-holding attribute will be created:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
```

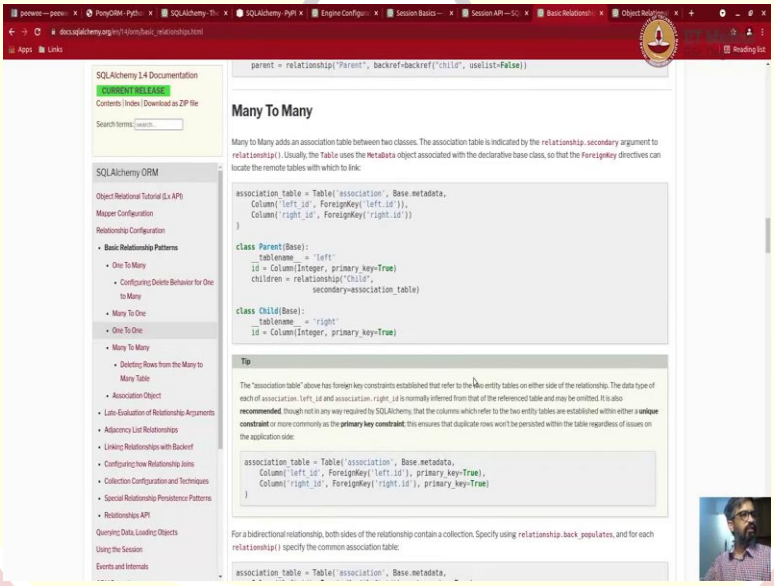
Bidirectional behavior is achieved by adding a second `relationship()` and applying the `relationship.back_populates` parameter in both directions:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", back_populates="parents")

class Child(Base):
    __tablename__ = 'child'
    id = Column(Integer, primary_key=True)
    parents = relationship("Parent", back_populates="child")
```

Alternatively, the `relationship.backref` parameter may be applied to a single `relationship()`, such as `Parent.child`:

```
class Parent(Base):
    __tablename__ = 'parent'
    id = Column(Integer, primary_key=True)
    child_id = Column(Integer, ForeignKey('child.id'))
    child = relationship("Child", backref="parents")
```



```
    parent = relationship("Parent", backref=backref("child", uselist=False))
```

Many To Many

Many to Many adds an association table between two classes. The association table is indicated by the `relationship.secondary` argument to `relationship()`. Usually, the `Table` uses the `MetaData` object associated with the declarative base class, so that the `ForeignKey` directives can locate the remote tables with which to link:

```
association_table = Table('association', Base.metadata,
    Column('left_id', ForeignKey('left.id')),
    Column('right_id', ForeignKey('right.id'))
)

class Parent(Base):
    __tablename__ = 'left'
    id = Column(Integer, primary_key=True)
    children = relationship("Child",
                    secondary=association_table)

class Child(Base):
    __tablename__ = 'right'
    id = Column(Integer, primary_key=True)
```

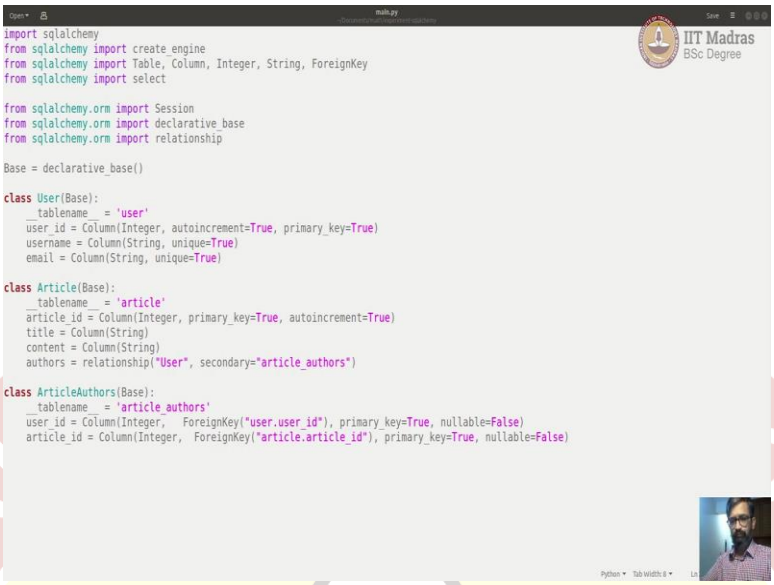
Tip

The "association table" above has foreign key constraints established that refer to the two entity tables on either side of the relationship. The data type of each of `association.left_id` and `association.right_id` is normally inferred from that of the referenced table and may be omitted. It is also **recommended**, though not in any way required by SQLAlchemy, that the columns which refer to the two entity tables are established within either a **unique constraint** or more commonly as the **primary key constraint**; this ensures that duplicate rows won't be persisted within the table regardless of issues on the application side:

```
association_table = Table('association', Base.metadata,
    Column('left_id', ForeignKey('left.id'), primary_key=True),
    Column('right_id', ForeignKey('right.id'), primary_key=True)
)
```

For a bidirectional relationship, both sides of the relationship contain a collection. Specify using `relationship.back_populates`, and for each `relationship()` specify the common association table:

```
association_table = Table('association', Base.metadata,
```



There are many ways to define relationships in SQL alchemy. You can go to the page here and in 1.4.23 version and go to basic relationship you can see how to define one to many relationship. Here you can see, there is a parent table and the child table. Parent table has a children, which is which can be many children, children by default is many. And then parent ID is only one which means a child can have only one parent, but the parent can have many children. Similarly, we can check other things. Many to one would be same thing, you can define. Other interesting is many to many.

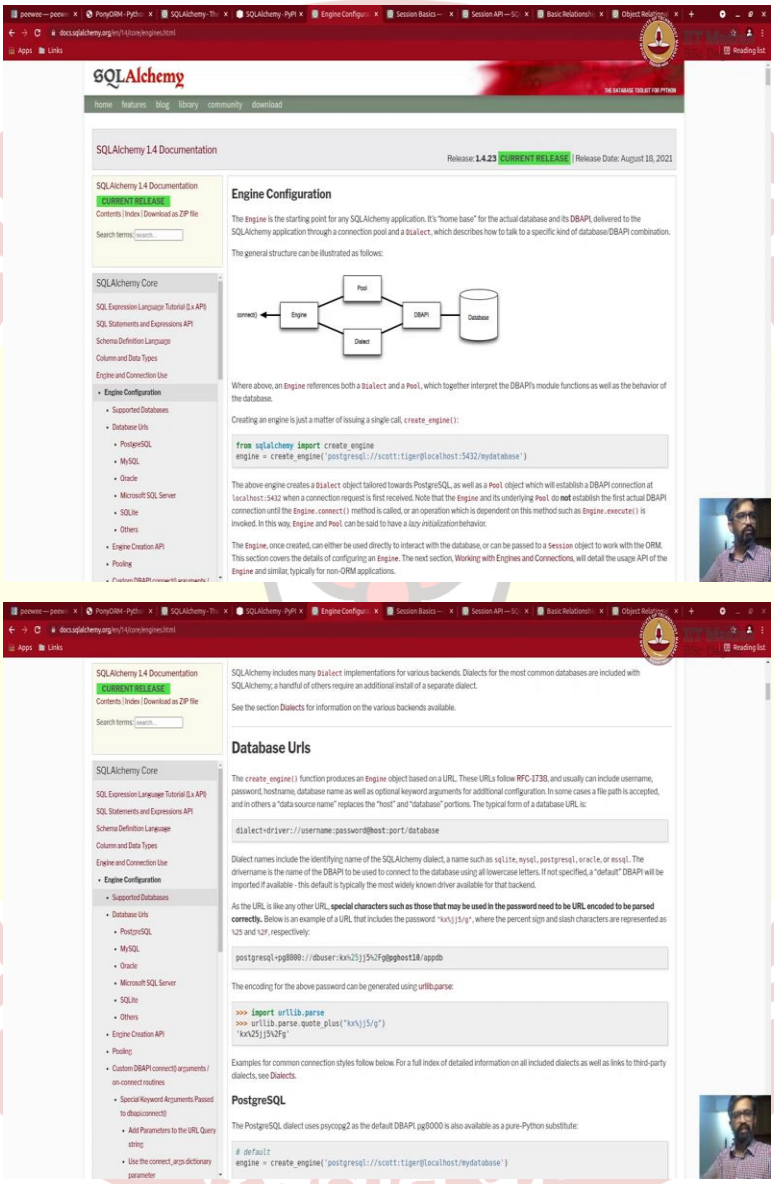
Here, if here is the definition of association table, which were created separately actually in our case, we do not have to worry. And then their relationship is managed, you can see that. You know, the relationship is managed here. Children is through a secondary table called association table. Because in this case, every child can now many parents, at least 2.

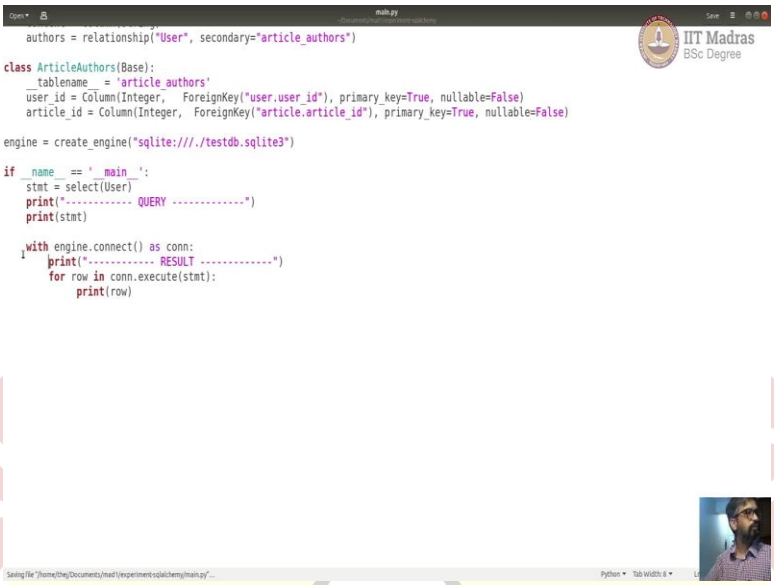
So, now in our case, an article can have many users as authors. That is what we are trying to do through this table. So, we can define a relationship here. Saying author's link into use a table through a secondary table called article_authors. So, this will be many to many relationship, which will have now many authors, and it can have a relationship user.

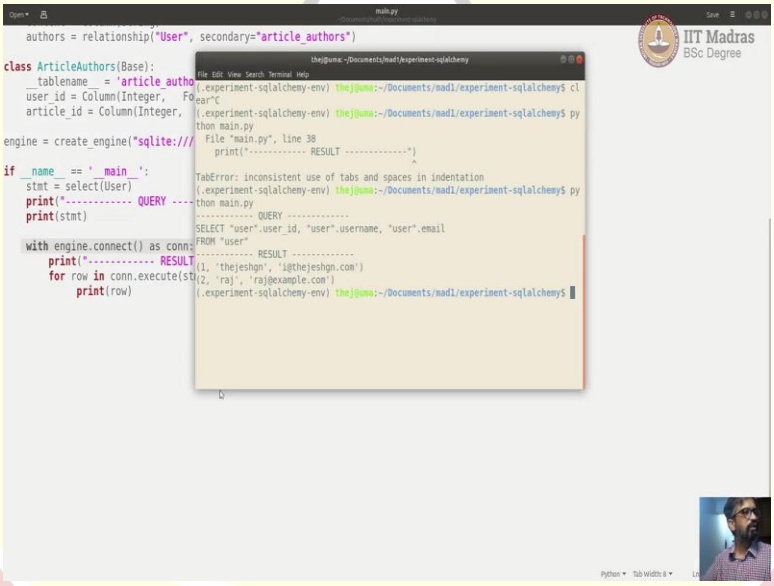
And through a secondary table, you can also similarly define, articles here. And this can link to this table. Basically, it will, it will link to all the users articles, link to article_authors. If you really want it, whenever you want to get user, you can also get articles. Similarly, now whenever you get article, you will get all the author, acceptor.

So, I am just going to remove this, because I do not want it to get all the articles related user all the time. Only this relationship I wanted to maintain. Because whenever I get an article, I also want to get authors. Now let us start our real experiment by connecting to a DB.

(Refer Slide Time: 11:19)







In SQL alchemy, you connect to a DB through an engine. Let us see here. An engine connects a pool and dialect together to provide a source of database connectivity and behavior. As you can see here, an engine has both pool and the dialect and it gives both of them for you to query against a database. You actually get an engine by calling create engine function and passing the database URL.

Now the database URL will be of the format, this. It will be dialect+driver://username: password@host:port/database. In our case, we are going to use SQL alchemy, SQLite. So, there will not be any host, or port. It will be just part of the database. So, let us try to do that, create an engine, here.

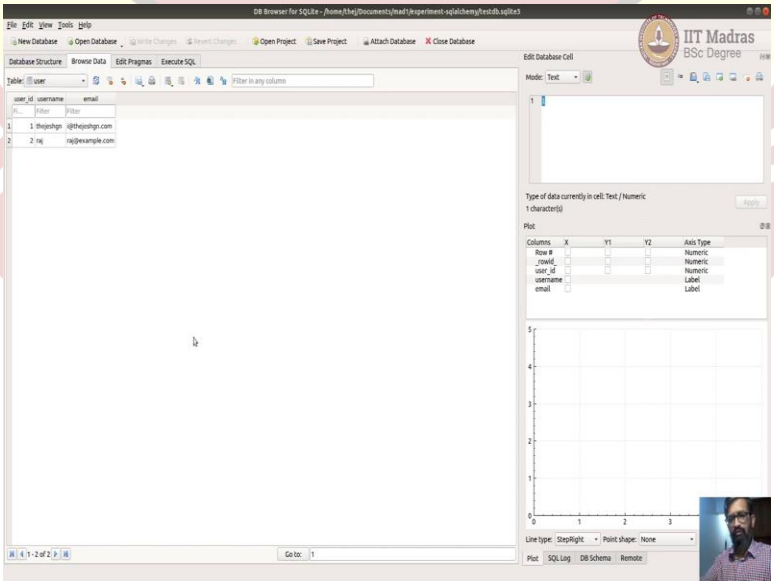
So here, I have not given any driver. So, you do not see any plus pi my sqlite or any of the drivers. It uses, it is going to use the default driver, which is installed. So, I am just going to remove this. It will be sqlite://, this is relative part of the database. Since it is in the same folder, it will be ./ test sqlite3.

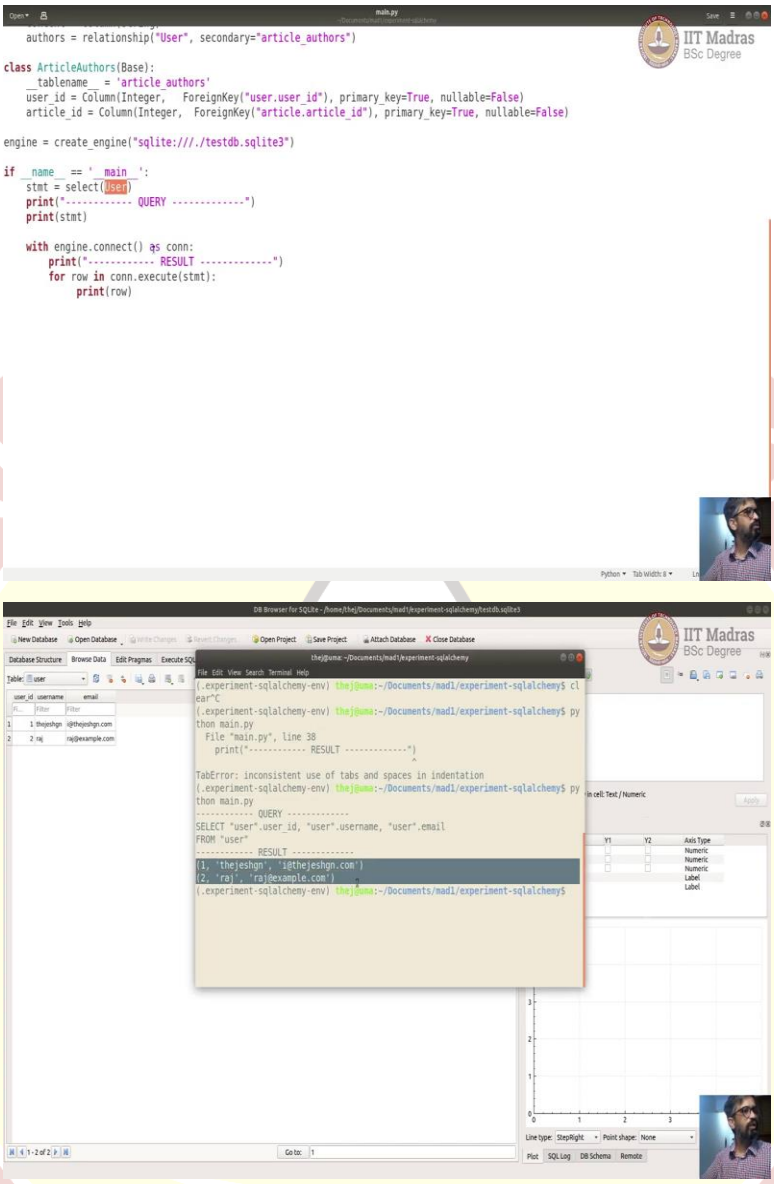
Now this engine uses, will give us a connection pool and dialect to talk to this database. Now we can run a connection, we can get a connection, and then a query. Let me show an example. Here, so here, you can see that I am creating a statement to run against a database, which is a select statement on user table. This is how you generally write the select, I am just printing it for our own reference, just want to make it clear. And then I am just going to print the…

So, here, I am just going to create a statement, I am just going to create a select statement from user table. And then we are going to print a statement. Then I am going to get a connection using engine or connect, it will give me a connection using this connection, I am going to execute this statement here. And I am just going to print the rows of the result.

We could, split it into multiple, statements, or you can use with combined statement here. And I am just going to run this, just going to run it again. You can see here, it is printing the statement SQL statement, which is a select * or select each column, from user here, this this statement. And then it is going to get a connection. And then run the query, get all.

(Refer Slide Time: 14:42)

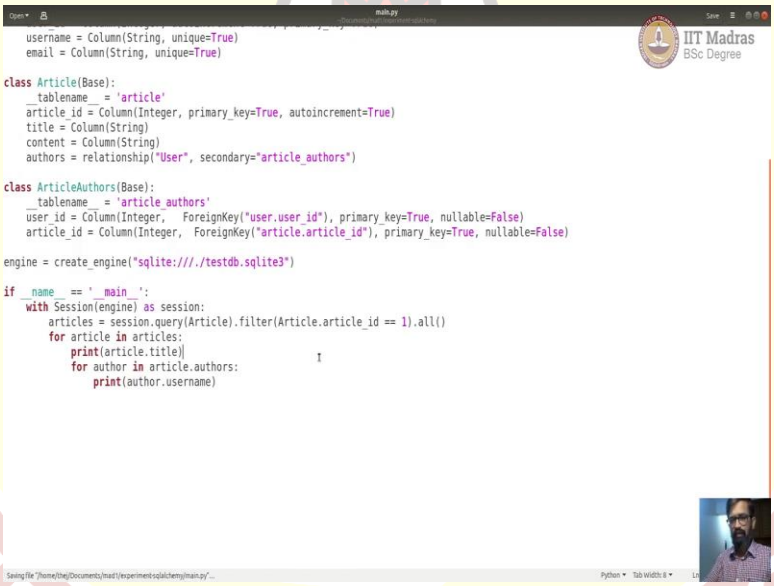


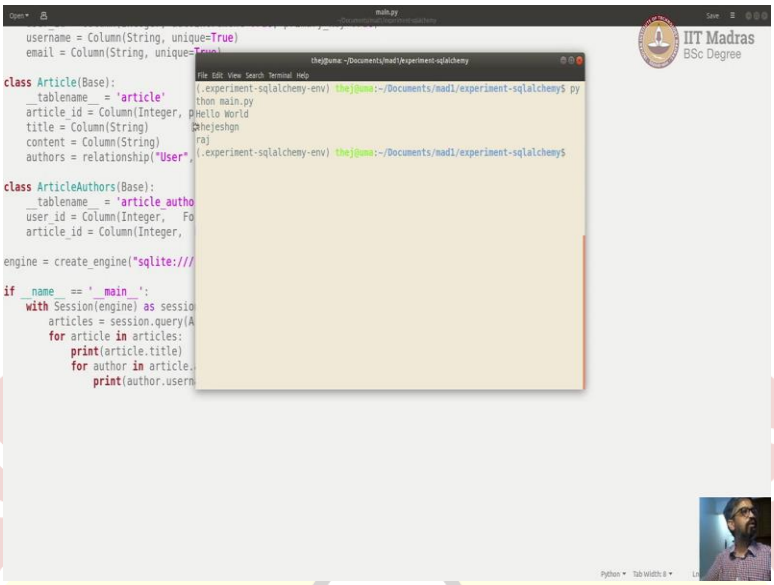


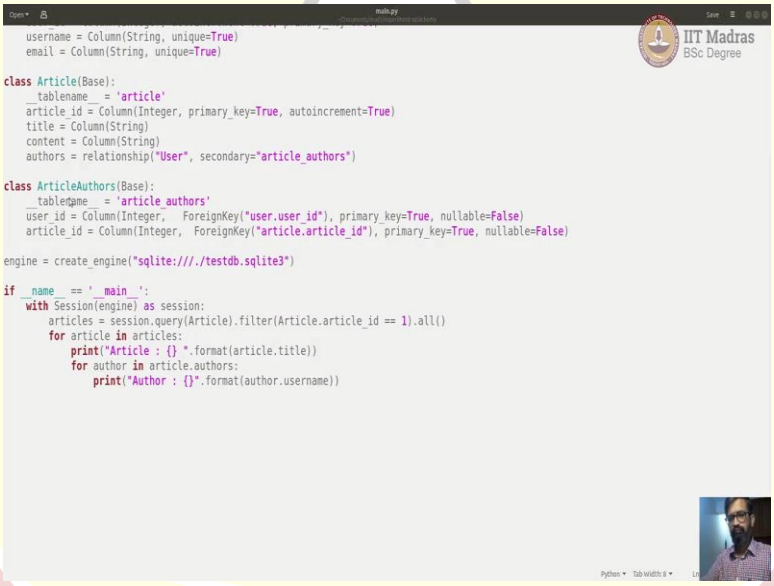
Since there are two rows in our database, in the article, which table are we trying to get? User table, in the user table there are two rows, we are actually printing two rows with all the columns. That, so you actually get a connection run a query. Now we could have made it simple, I mean, if you really wanted to separate it out, you can just connect, get a connection separately get all the rows separately and then iterate through it.

So, I just combined all of them, this is like an ideal way of doing it. Now, usually we do not try to get connection every time. We actually create a session and then use a session. The session establishes all the conversations with the database and represents a holding zone for all your objects, that you are loaded or associated with the (())(15:37) during the lifetime.

(Refer Slide Time: 15:40)







Let us go to the session, page and see the details here. When you take, take a little bit of time, and then go through it. There are many uses of sessions, we will go through a couple of them now. Let us run a query using a session. First you have to get a session. And you get a session using a session object or session class and passing the engine to it. Let me just paste a piece of code and show it to you. Here, I am getting a session by passing the engine which has connection pool and other things.

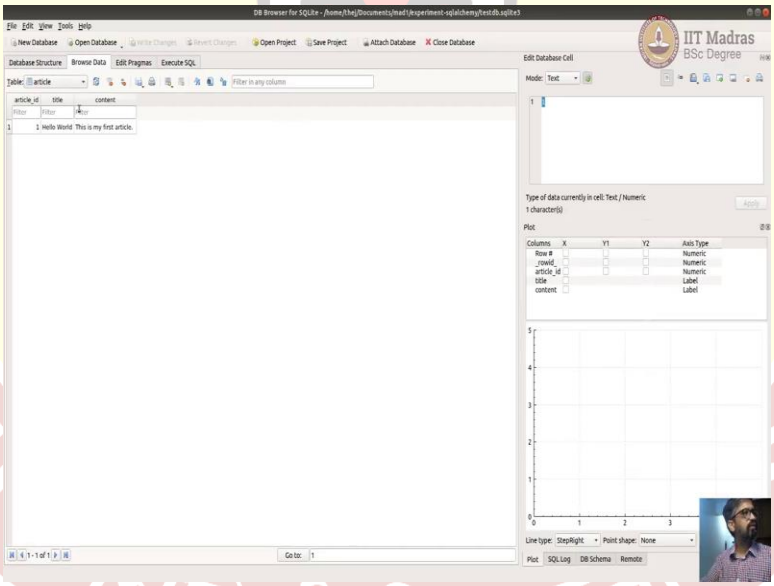
And then I am trying to get articles here, session gives a function called query and that you can use to query things here. Let me explain this, what is happening here. I am just querying the

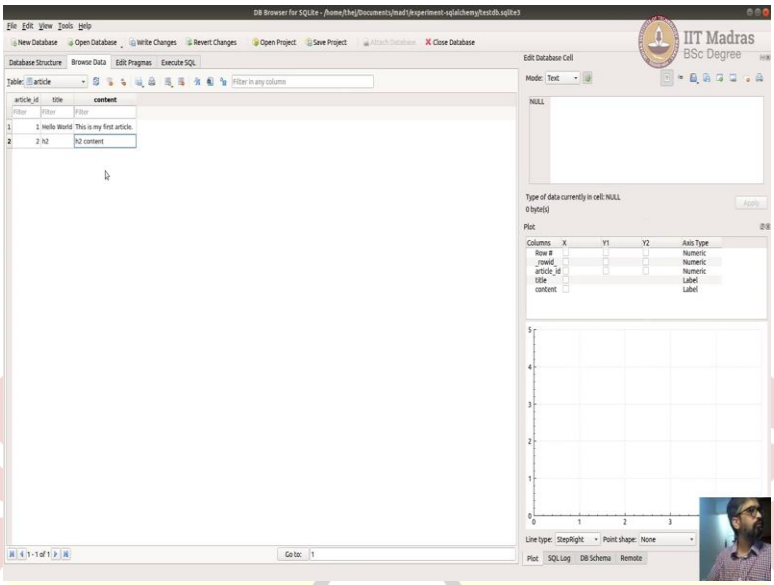
article table, then filtering the rows by article_id=1. And then getting all those filtered articles. And then I am iterating through those articles and printing the article title.

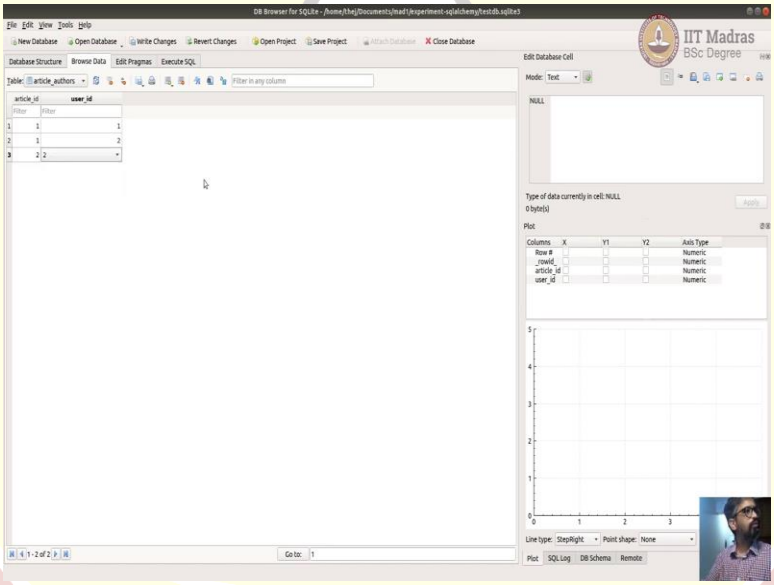
And then, for aurthor, author in article_authors I am printing author name, like I explained authors is a relationship as soon as we get article in this case it will also get all the related articles using this relational table and load it into the array. And then I can directly access like an attribute in this article and print them. The user name is part of author so I am just printing the user name. Let me run this, just going to clear.

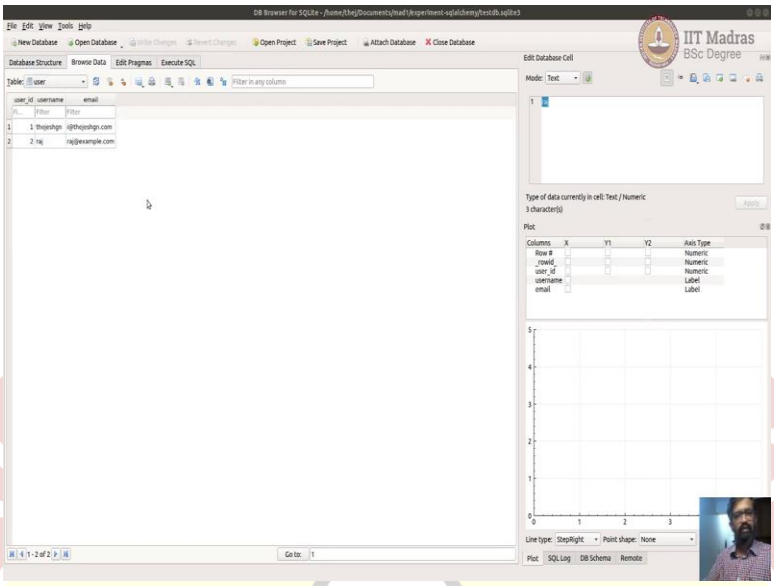
It is printing hello world, thejeshgn and raj. Hello world is a article title and you know there two authors. I am just going to make it clear by printing, in a better way. I am just going to make this also. For my 12, so that it is easy to read. I am just going to save and you can see that article is hello world author one is thejeshgn and author two is raj. Now you can also make it to article two.

(Refer Slide Time: 18:21)



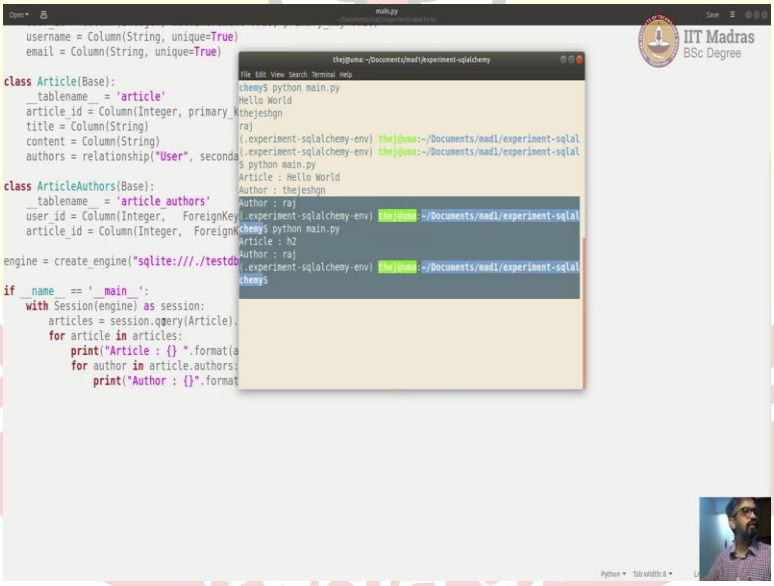

DB Browser for SQLite - /home/thej/Documents/mad1/experiment-sqlalchemy/testdb.sqlite3
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: article
Filter in any column
article_id
title
content
1 Hello World This is my first article.
2 h2 h2 content
Edit Database Cell
Mode: Text
NULL
Type of data currently in cell: NULL
0 byte(s)
Apply
Plot
Columns X Y1 Y2 Axis Type
Row # Numeric
rowid Numeric
article_id Numeric
title Label
content Label
Line type: StepRight
Point shape: None
Plot
SQL Log
DB Schema
Remote
Go to: 1
IIT Madras
BSc Degree

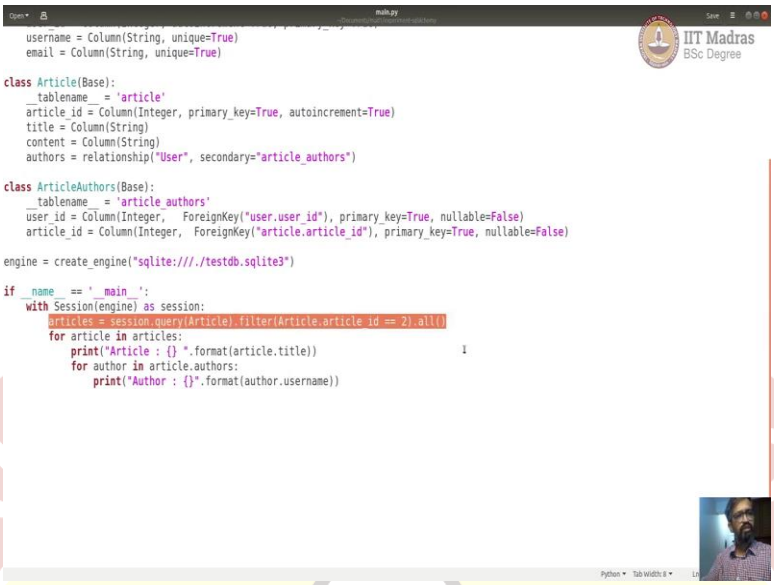

DB Browser for SQLite - /home/thej/Documents/mad1/experiment-sqlalchemy/testdb.sqlite3
File Edit View Tools Help
New Database
Open Database
Write Changes
Revert Changes
Open Project
Save Project
Attach Database
Close Database
Database Structure
Browse Data
Edit Pragmas
Execute SQL
Table: article_authors
Filter in any column
article_id
user_id
1 1 1
2 1 2
3 2 2
Edit Database Cell
Mode: Text
NULL
Type of data currently in cell: NULL
0 byte(s)
Apply
Plot
Columns X Y1 Y2 Axis Type
Row # Numeric
rowid Numeric
article_id Numeric
user_id Numeric
Line type: StepRight
Point shape: None
Plot
SQL Log
DB Schema
Remote
Go to: 1
IIT Madras
BSc Degree

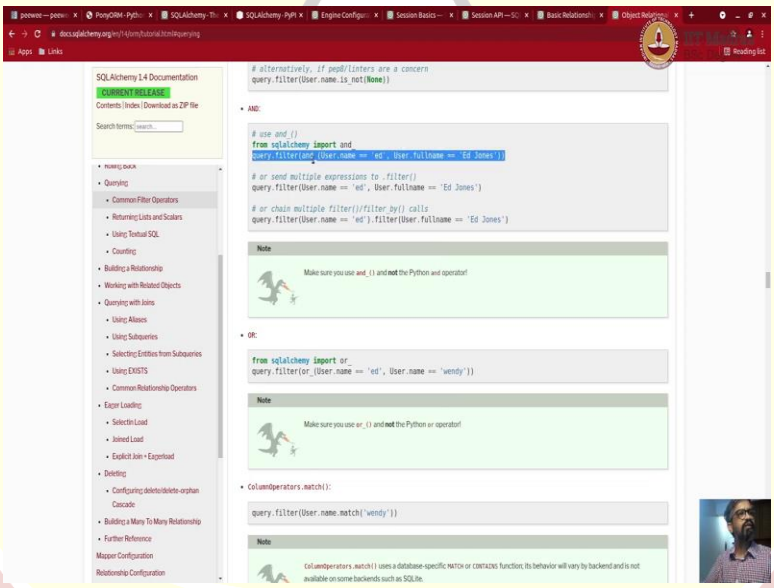


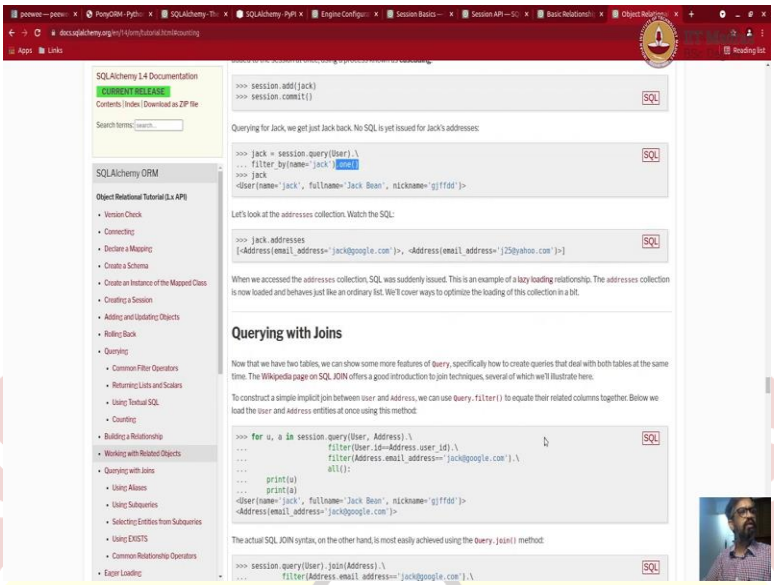
Let us check if there is another article, there is no another article. If we insert an article let us add an article to h2, h2 content. I am going to save it and go to article author and I did not throw 2 the author is 2, on that will be Raj.

(Refer Slide Time: 18:56)




main.py
IIT Madras
BSc Degree
username = Column(String, unique=True)
email = Column(String, unique=True)
class Article(Base):
__tablename__ = 'article'
article_id = Column(Integer, primary_key=True, autoincrement=True)
title = Column(String)
content = Column(String)
authors = relationship("User", secondary="article_authors")
class ArticleAuthors(Base):
__tablename__ = 'article_authors'
user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)
engine = create_engine("sqlite:///./testdb.sqlite3")
if __name__ == '__main__':
with Session(engine) as session:
articles = session.query(Article).filter(Article.article_id == 2).all()
for article in articles:
print("Article : {} ".format(article.title))
for author in article.authors:
print("Author : {}".format(author.username))

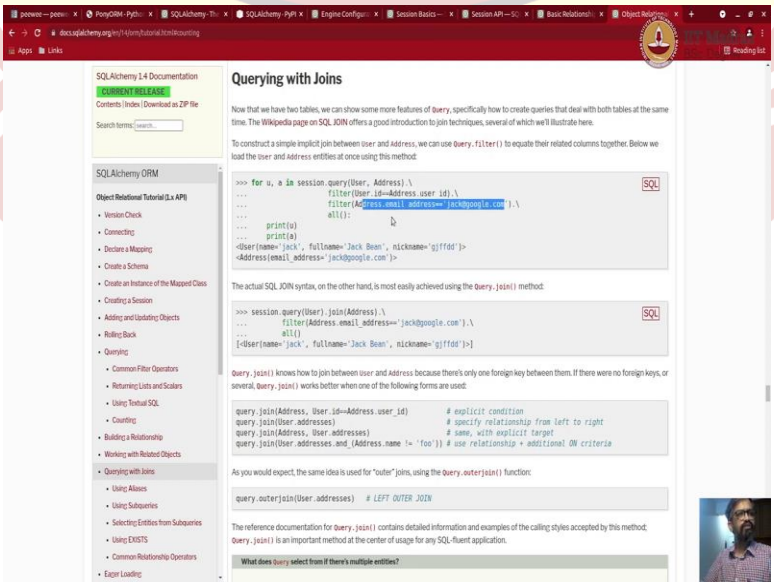

SQLAlchemy 1.4 Documentation
CURRENT RELEASE
Contents | Index | Download as ZIP file
Search terms:
Querying
Common Filter Operators
Returning Lists and Scalars
Using Textual SQL
Counting
Building a Relationship
Working with Related Objects
Querying with Joins
Using Aliases
Using Subqueries
Selecting Entities from Subqueries
Using EXISTS
Common Relationship Operators
Eager Loading
Selectin Load
Joined Load
Explicit Join + Eagerload
Deleting
Configuring delete/delete-orphan Cascade
Building a Many To Many Relationship
Further Reference
Mapper Configuration
Relationship Configuration
# alternatively, if pep8/linters are a concern
query.filter(User.name.is_not(None))
AND:
# use and_()
from sqlalchemy import and_
query.filter(and_(User.name == 'ed', User.fullname == 'Ed Jones'))
# or send multiple expressions to .filter()
query.filter(User.name == 'ed', User.fullname == 'Ed Jones')
# or chain multiple filter()/filter_by() calls
query.filter(User.name == 'ed').filter(User.fullname == 'Ed Jones')
Note
Make sure you use and_() and not the Python and operator!
OR:
from sqlalchemy import or_
query.filter(or_(User.name == 'ed', User.name == 'wendy'))
Note
Make sure you use or_() and not the Python or operator!
ColumnOperators.match():
query.filter(User.name.match('wendy'))
Note
ColumnOperators.match() uses a database-specific MATCH or CONTAINS function; its behavior will vary by backend and is not available on some backends such as SQLite.



So, now let us go back here, change it to 2 and then run it again. Now it changed article h2 author is Raj. And you know there are many functions and many way to query. We can go to querying part and see many of the operators available and you know how do you do join in and left join, right join, inner join and outer join etcetera on this thing and it is straightforward.

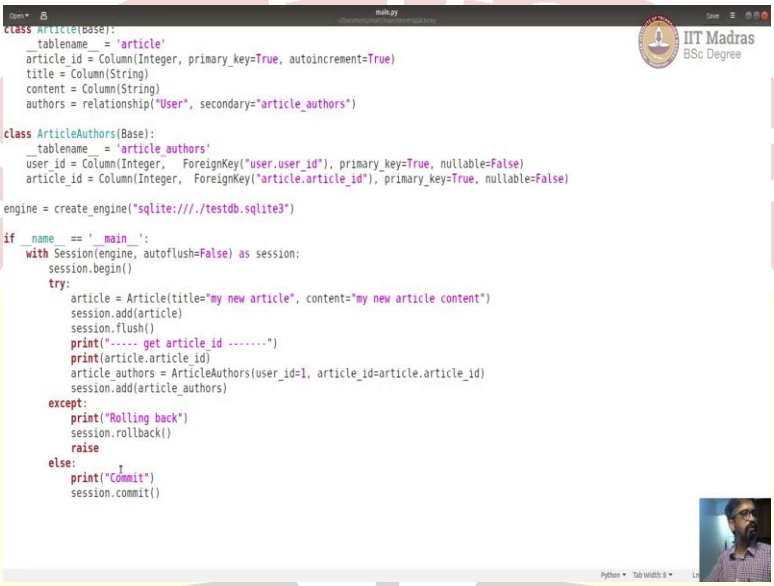
For example, let us see if there is an example here. So, here is where actually we are filtering by multiple conditions using an and. You are importing an and, then making sure both user name and user full name is this. This of (())(19:53) and. And if you want to get only one record you can see dot one, etcetera.

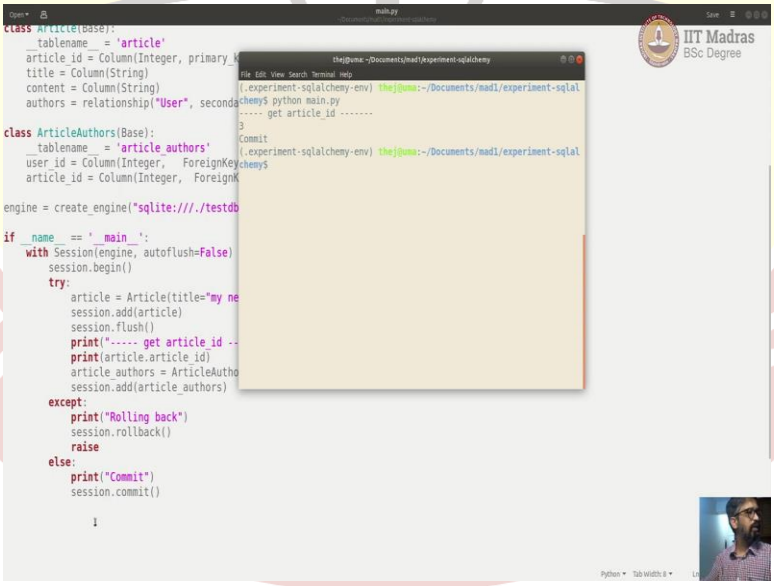
(Refer Slide Time: 20:04)

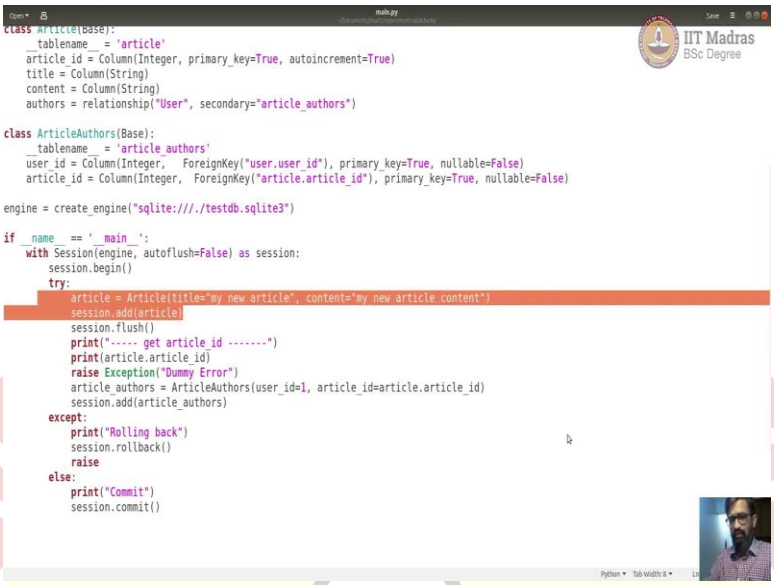


And then there are joints. And you can use the joints to query as well. For example, here, they are joining user table and address table, using ID is across them and then filtering by email address of the user. This is how we write joins, there are many other options available, I think you should just explore this page on your own, and learn about various ways of querying.

(Refer Slide Time: 20:28)







Now, there is another use of session that is you can using session you can create transactions. Transactions is a unit of work, you want to either completely be done, or you want to completely roll back, you do not want a piece of work to commit into database. Let us say you have 3 or 4 steps to do. In our case, we have two steps, first we say the article, and then say we say the authors of the article.

If, after saving the article, something happens on article authors does not get saved, then we will have this random article with no authors. We do not want that to happen. So, ideally, we want both queries to and both the inserts to happen, or nothing to happen. Let us see a general case of and then by error case handling of it. I am just going to write a piece of code here. So, here I am creating a session with autoflush=false.

Autoflush is flushing in SQL alchemy is a way to send the commands to a database and make it execute. And then here, I am making it false by default, so we can manually call it by default it is auto. I am creating a session, then I am actually creating article and adding that article to the session, which is, we just want to insert an article, and then session, session.flash, I am sending it to the database now.

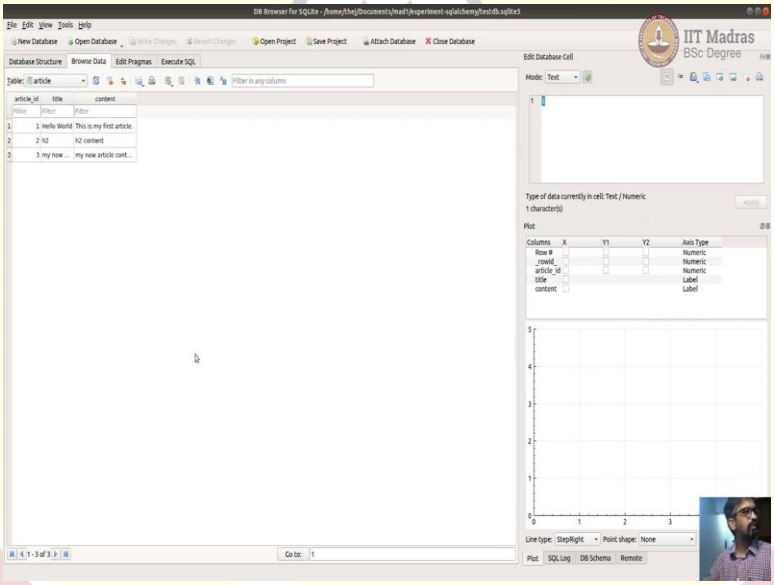
And then it sends me back an article_id, which is an auto incremented id, and then I take this id, and then I am going to insert into article_authors. And then if everything goes well, then it goes to else part and commit, if something happens in between then it goes to roll back section in this

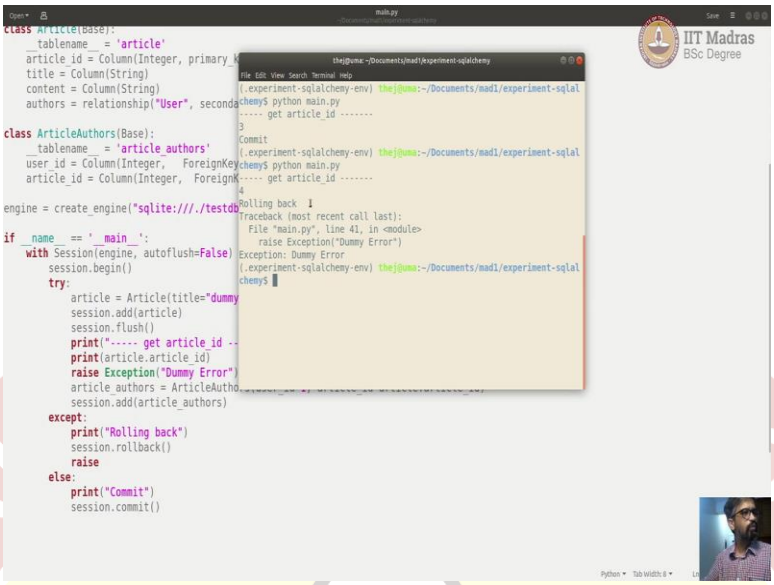
except section, and then it rolls back embraces the thing raises an exception again, and then ends the program.

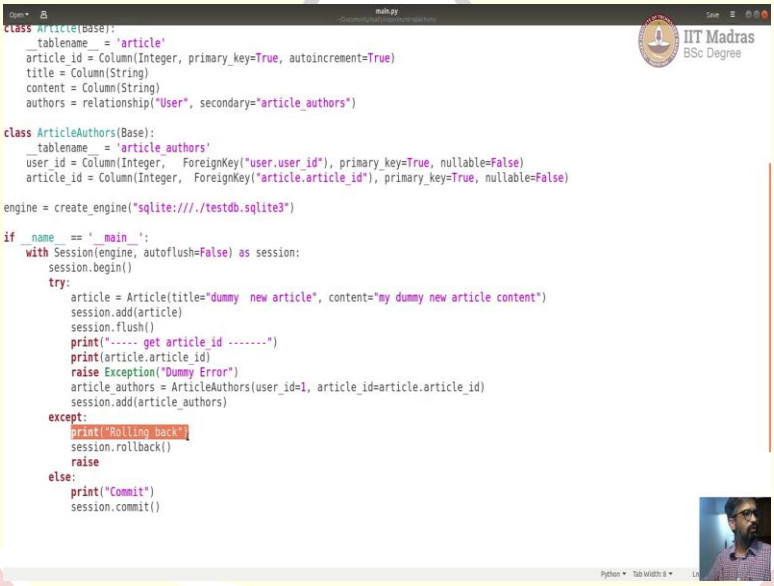
Here is when we just make it clear. So, let us run this, this goes clear. So yes, so it got article_id, then it went into commit area, and it committed. Let us assume, if something happens here, after getting the article_id, and before we create and save, of the article authors. And because of that, we have to roll back an article should not get saved.

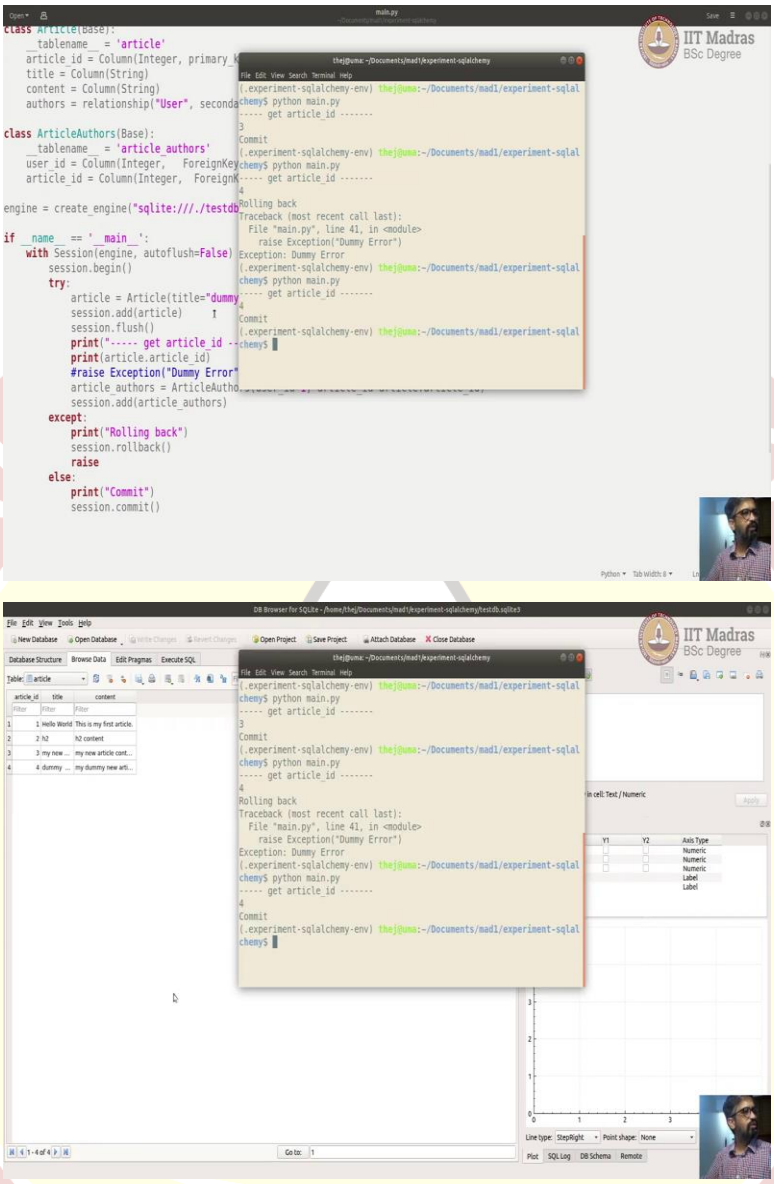
So, let us just raise a dummy exception. I am just going to raise a dummy exception here. And just throw, I am just going to call dummy error. As soon as the error happens, it goes to the except part and rolls back. Hence, even though we run it again, this should not get inserted.

(Refer Slide Time: 23:57)




main.py
class Article(Base):
__tablename__ = 'article'
article_id = Column(Integer, primary_k
title = Column(String)
content = Column(String)
authors = relationship("User", seconda
class ArticleAuthors(Base):
__tablename__ = 'article_authors'
user_id = Column(Integer, ForeignKey
article_id = Column(Integer, ForeignK
engine = create_engine("sqlite:///./testdb
if __name__ == '__main__':
with Session(engine, autoflush=False)
session.begin()
try:
article = Article(title="dummy
session.add(article)
session.flush()
print("----- get article_id --
print(article.article_id)
raise Exception("Dummy Error")
article_authors = ArticleAutho
session.add(article_authors)
except:
print("Rolling back")
session.rollback()
raise
else:
print("Commit")
session.commit()
thej@umx: ~/Documents/mad1/experiment-sqlalchemy
File Edit View Search Terminal Help
(.experiment-sqlalchemy-env) thej@umx:~/Documents/mad1/experiment-sqlal
chemy$ python main.py
----- get article_id -------
3
Commit
(.experiment-sqlalchemy-env) thej@umx:~/Documents/mad1/experiment-sqlal
chemy$ python main.py
----- get article_id -------
4
Rolling back
Traceback (most recent call last):
File "main.py", line 41, in <module>
raise Exception("Dummy Error")
Exception: Dummy Error
(.experiment-sqlalchemy-env) thej@umx:~/Documents/mad1/experiment-sqlal
chemy$
IIT Madras
BSc Degree


main.py
class Article(Base):
__tablename__ = 'article'
article_id = Column(Integer, primary_key=True, autoincrement=True)
title = Column(String)
content = Column(String)
authors = relationship("User", secondary="article_authors")
class ArticleAuthors(Base):
__tablename__ = 'article_authors'
user_id = Column(Integer, ForeignKey("user.user_id"), primary_key=True, nullable=False)
article_id = Column(Integer, ForeignKey("article.article_id"), primary_key=True, nullable=False)
engine = create_engine("sqlite:///./testdb.sqlite3")
if __name__ == '__main__':
with Session(engine, autoflush=False) as session:
session.begin()
try:
article = Article(title="dummy new article", content="my dummy new article content")
session.add(article)
session.flush()
print("----- get article_id -------")
print(article.article_id)
raise Exception("Dummy Error")
article_authors = ArticleAuthors(user_id=1, article_id=article.article_id)
session.add(article_authors)
except:
print("Rolling back")
session.rollback()
raise
else:
print("Commit")
session.commit()
IIT Madras
BSc Degree

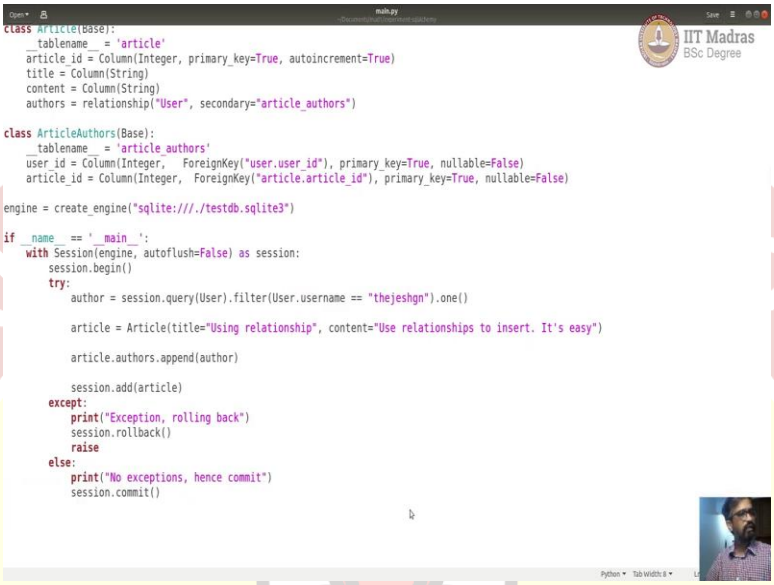


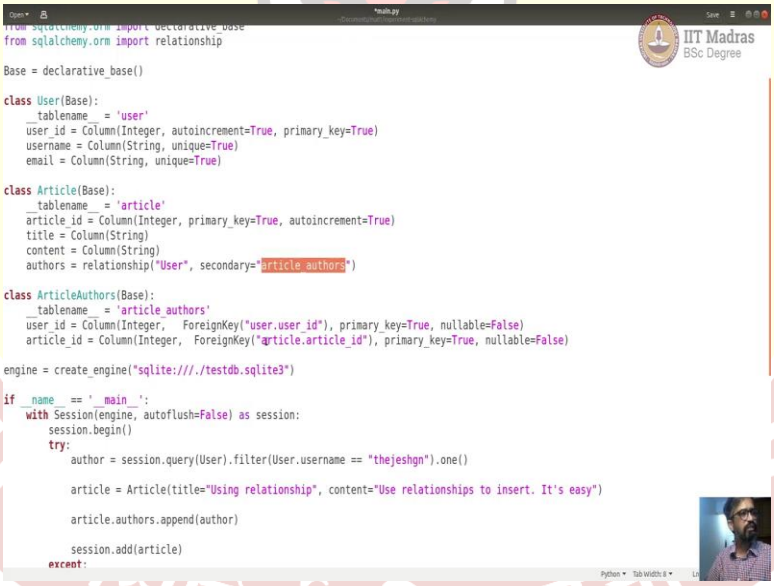
Let us just check whether the last time it got inserted. Actually, it has got inserted. So, I am just going to call this my dummy a new article. Let us run this now. It threw an error, you can see a dummy error. And it came here. It is rolling back, it is printed rolling back and rolled back actually, this would not get inserted. So, I can refresh it. It is not inserted. Now, we can just mask this, which means there is no error again, and run it again. We, got committed and inserted, I can see a committed and inserted.

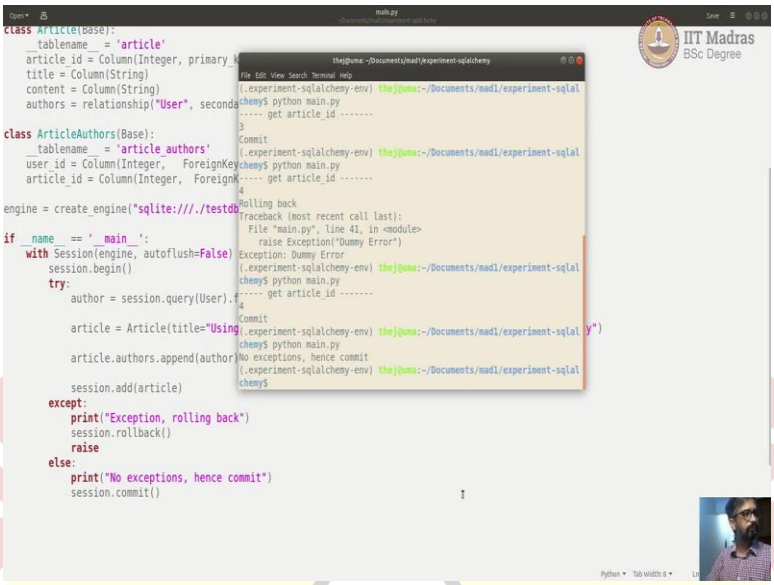
So, this is how we use transactions to handle errors, you if the database transaction has multiple steps, and you want to do many things before you commit everything at once. Ideally, we do not

do this this way we use relationships to add the data into the DB in simple cases, since this is a simple case, we could just use the relationship to insert.

(Refer Slide Time: 25:25)



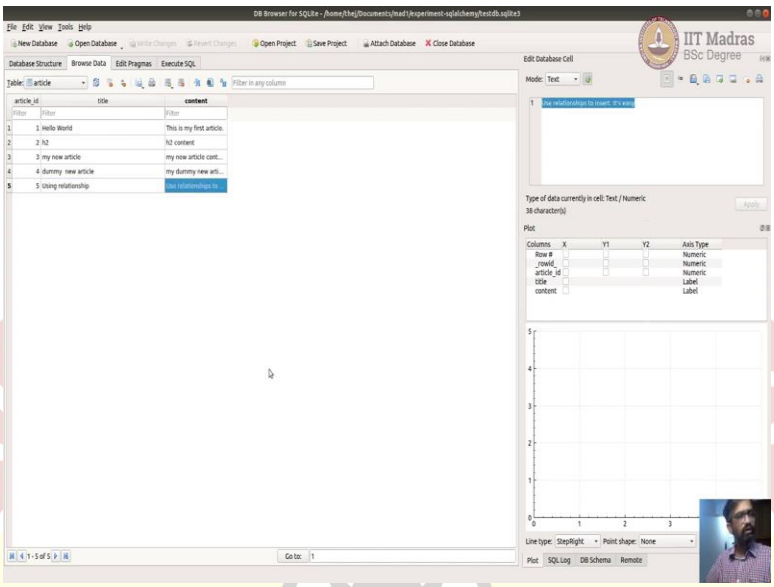


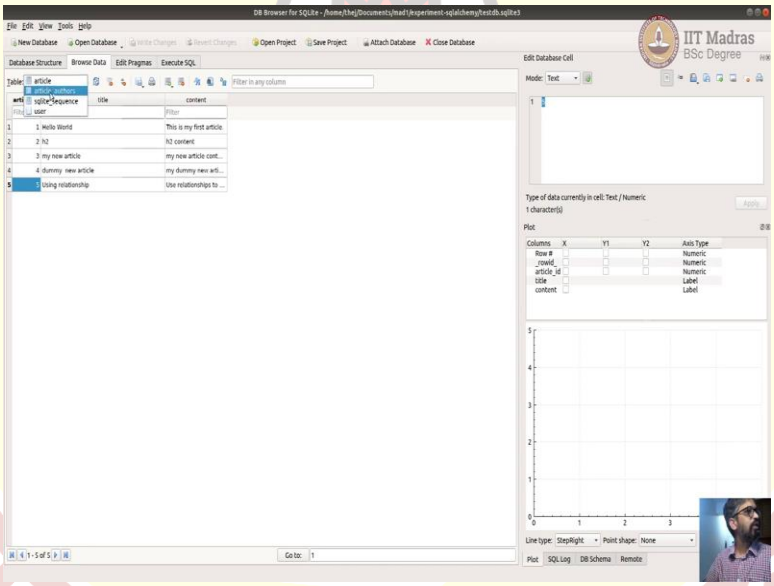


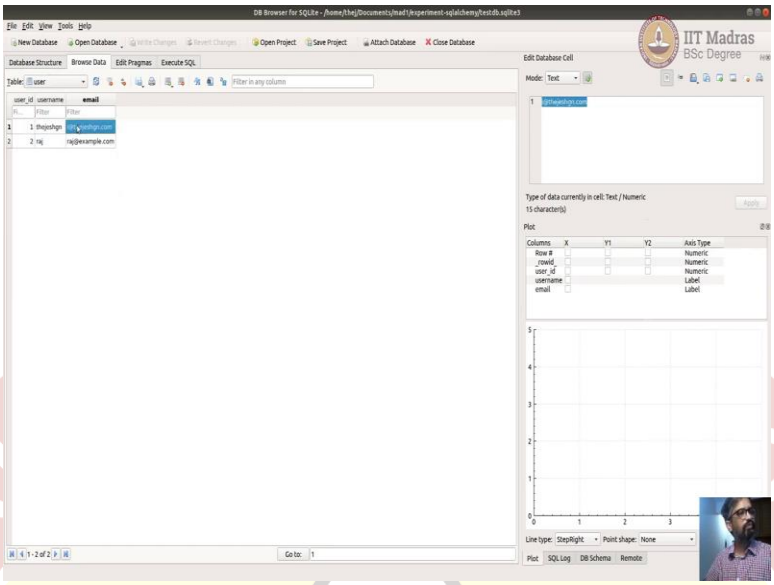
I will show one of the examples here, I am still making autoflush is false. And then I am getting an author object, by querying the user table by filtering Thejeshgn, he is going to be my author. And then I am creating an article by passing article title and article content here. Now, I need to insert article authors. Now, instead of like, last time, inserting into article authors directly, we are going to use relationships here, we are going to add an author into, articles author's attribute, and then save the article.

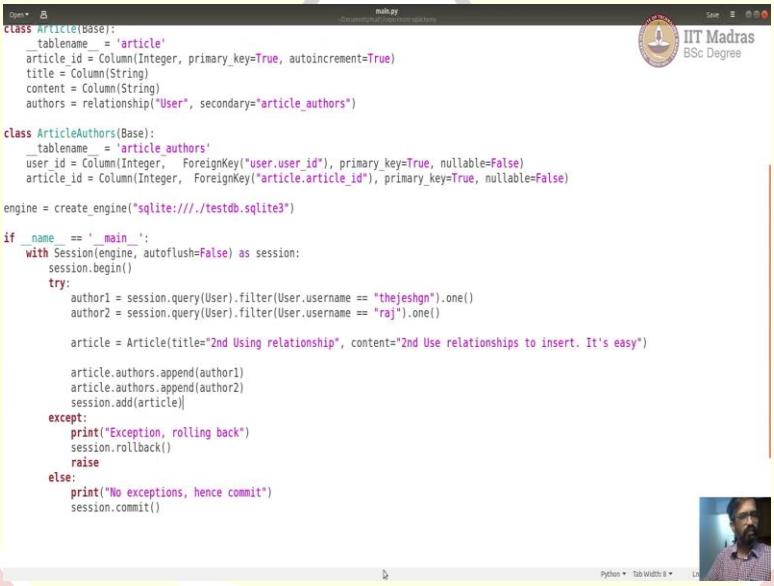
And you know, and then commit, indirectly what it will do is it will actually commit the article, then it will commit article authors based on the relationship, which is, authors here, and based on the relationship, it goes to article authors and inserts respective row and then saves it. This is cleaner, simpler and pythonic way, this is what we meant by it is easier, almost feels like accessing Python things, even though you are accessing database related stuff. Now, let us run this. No exception, and hence commit. So, come here, and then committed, let us see whether it got inserted, using relationship.

(Refer Slide Time: 27:02)







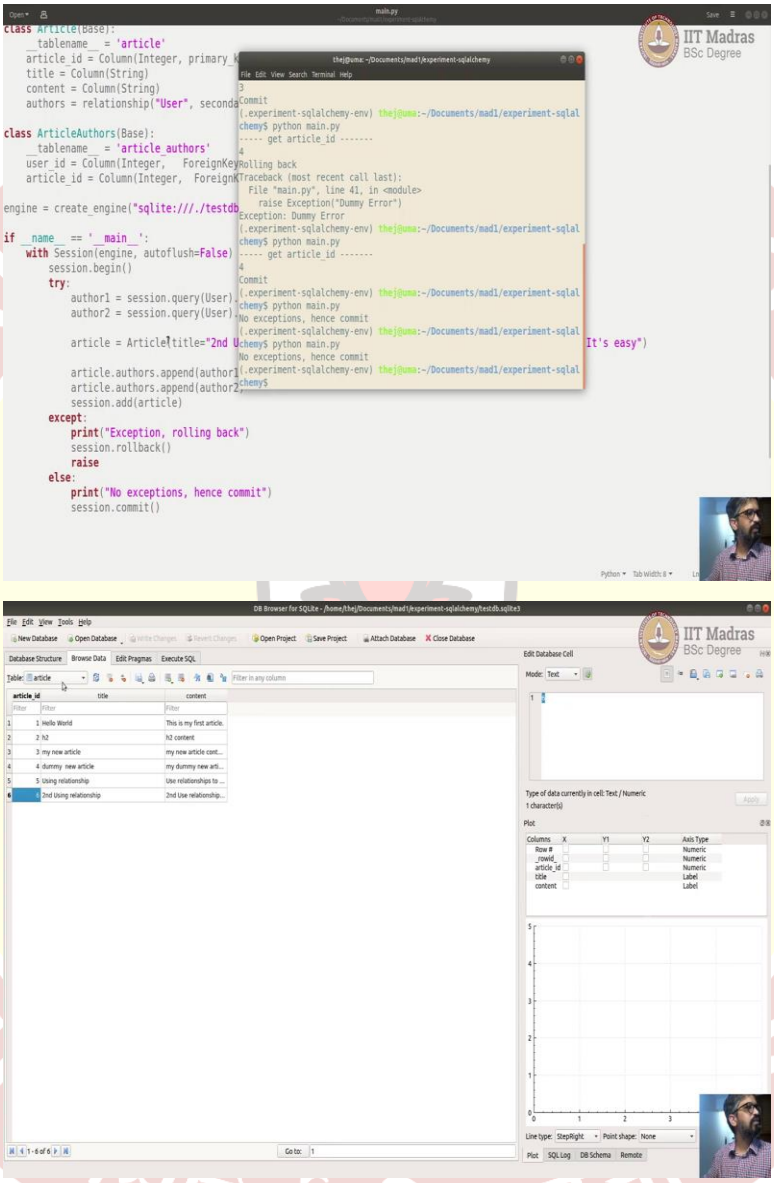


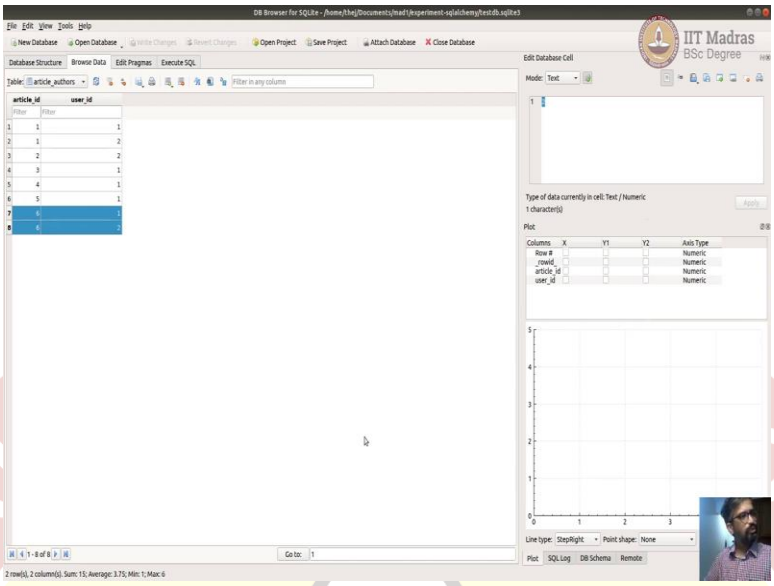
I am going to refresh here and go. It has inserted this. And now actually, the most important thing is whether it has inserted this part, or equivalent of this part, which is connecting this to this. And this title is user relationship, the article_id is 5. And if you go to article 5, it is linked to one, which is if you want to be user table, it takes one.

And that is what we wanted here. You could also done many, like, let us run one more. I would say author 1, author 2, and let us say this was Raj. And second, using relationship, just want to do random thing and do one author, you can add another author two authors. I am just going to print the article_id. We can just compare it using this title. So, here I am doing the same thing I

quiet for two authors. And then, I am creating an article object, I am adding two authors to that article, and just saving it. Let us run it.

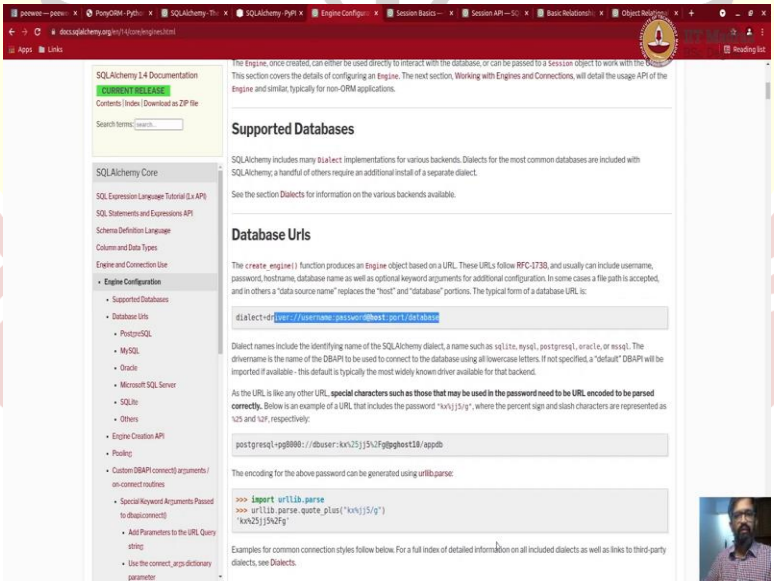
(Refer Slide Time: 28:50)

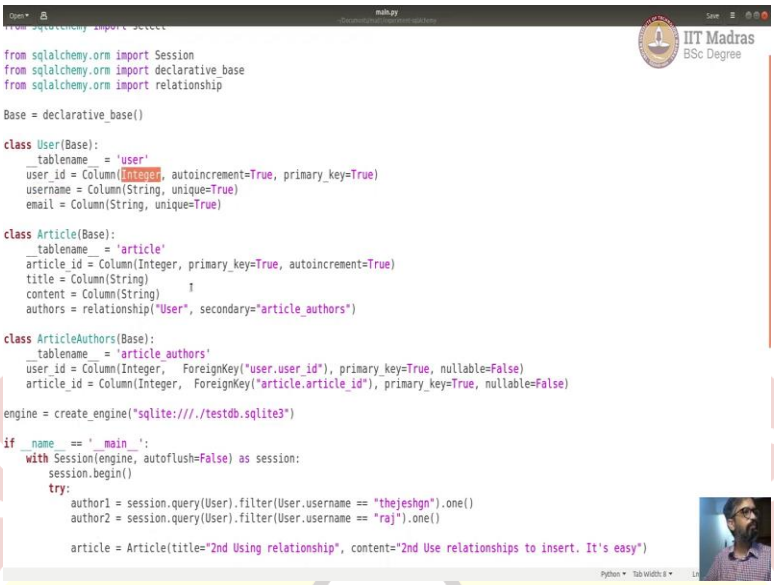


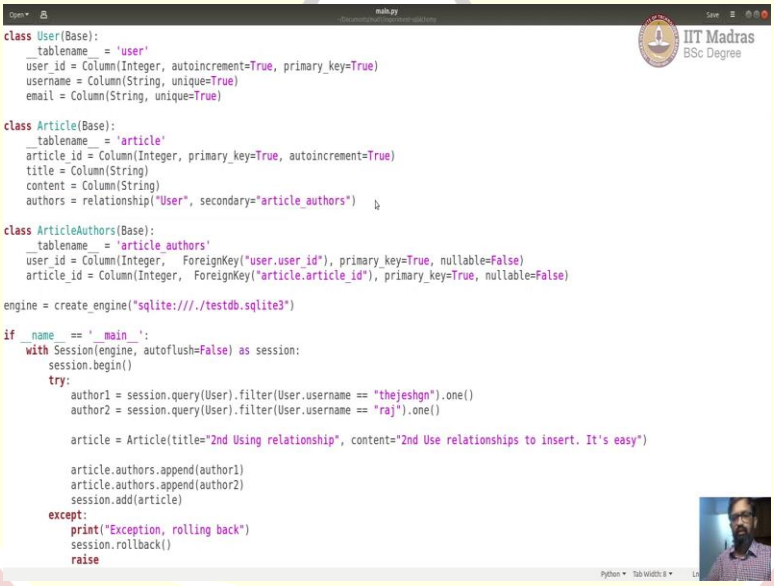


He got no exception hence commit. Let us see what whether this is this article with this title got inserted. Because it got inserted second using relationship and the article id is 6. Let us go, to the article author 6. You can see there are two relationships. It is quite easy to do using the relationship and both acquiring, inserting, even deleting for that matter. So, this is the end of this small introductory session to SQL alchemy, and generally the concept of ORM.

(Refer Slide Time: 29:35)



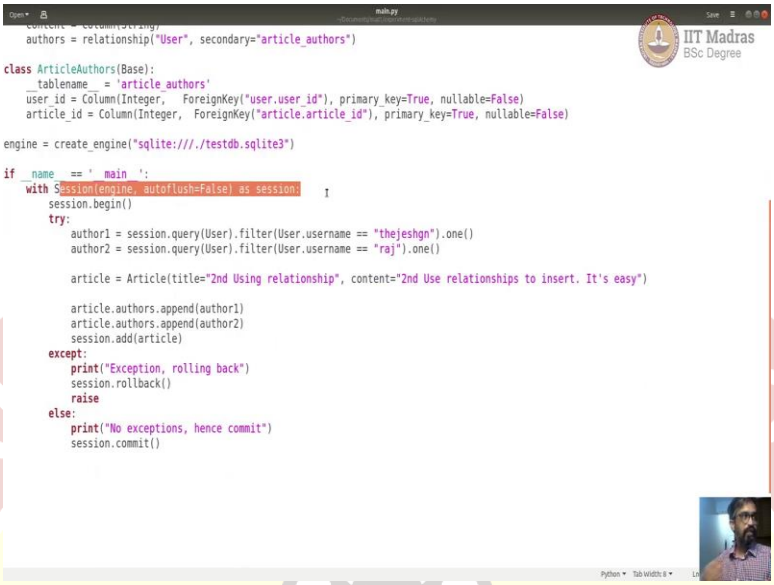


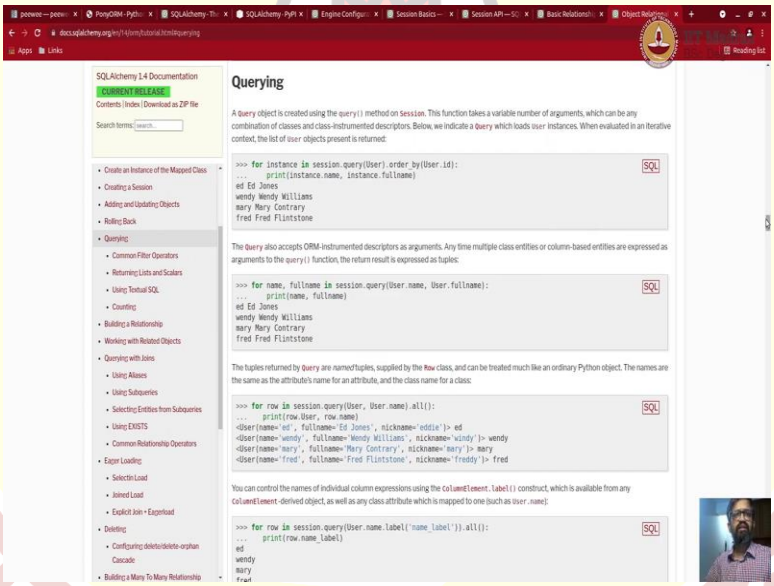


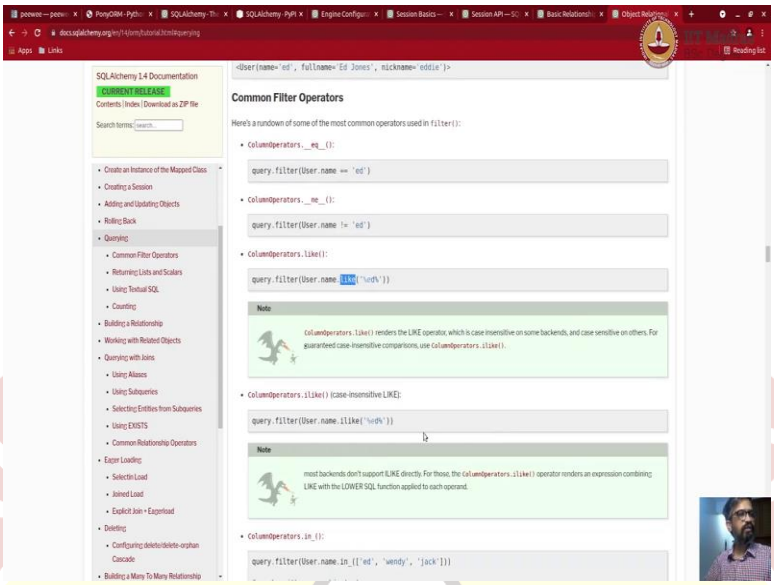
There are a couple of things that you should remember one how to get the database URL. In our case it is straightforward because we are going to use sqlite, it is just a part of the escalator. Since it is going to use relate to path you have to start with the slash(/) and use this (./)dot slash for the local directory. Usually it will begin with sqlite:///. We are not giving any driver mean, just using that default driver name.

Then if you already have a database, you have to write the models for the database, it is in the base class. And then, add also models. Now, you are going to use the same name as the column name or the attribute name, give this specific of, column types here, and all the conditions here. And also make sure that you establish the relationship the way you want it to be.

(Refer Slide Time: 30:35)







If you want more details, you can go to this page and learn a lot about relationship (())(30:38), to learn about at least four types of these relationships. So, one to many, many to one, one to one, and many to many, all of them are useful. You will, use them on day to day basis when you are accessing.

The third thing is how to create a session, you create a session by passing the engine. Engine has both dialect and the pool connection pool, you do not usually you will create one only one engine in the application context. And then you pass on that engine to the session and use the session to do all your stuff.

Use the, ORM way to do it use, object relations to create or delete objects. And then use the query on the session to query the database. You should actually go to this page, which has many examples on querying. There is a whole section on querying many ways, including all kind of joint, external and internal, and doing tops and operators like equal not equal, like operators, etcetera, etcetera. You should try most of them is straightforward to try with the given example.

And then you will learn about sessions and transactions, how to pickling, how to commit, how to roll back. This is how we handle errors when you are dealing with databases. That actually if you know, this many concept, I think you are kind of ready to jumpstart into using ORM in your application. This is in a standard way. We are also in the next coming sessions we will learn how to use it with the web application. Thank you for watching. Bye.