Unit 4: Object-Oriented Development and Python
Unit 5: Understanding UML
Unit 6: Hands-On with UML

On completion of this unit you will be able to:

- Design object-oriented models of a system.
- Develop object-oriented software using the Python programming languages.

Unified Modeling Language (UML)

- UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems.
- UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.
- OMG is continuously making efforts to create a truly industry standard.
- UML stands for **Unified Modeling Language**.
- UML is a pictorial language used to make software blueprints.
- UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design.

TYPES OF DIAGRAMS IN UML

CLASS, SEQUENCE, ACTIVITY AND STATE DIAGRAMS

CLASS DIAGRAM	ACTIVITY DIAGRAM	SEQUENCE DIAGRAM	STATE DIAGRAM
A class diagram models the static structure of a system. It shows relationships between classes, objects, attributes, and operations	depicts the control flowing from one activity to another, especially the logic of conditional structures, loops, concurrency.	depicting the sequence of messages flowing from one object to another, how their messages/events are exchanged in what time-order	describes the different possible states and state transitions triggered by various events (Seidl et al 2012),
	represents the flow of use cases	represents the interaction between classes or objects according to time	
	shows a workflow - a starting point, actions, decisions, splits and joins to show concurrent activities, and ending points. It's essentially a flowchart for a process or workflow, usually written using domain specific terms. It doesn't show classes, objects, or calls to methods/functions.	shows interactions between actors and objects and between two objects. Usually, this is at a method/function call level, perhaps even including parameters and return types. The lifelines also show the creation and deletion of instances of objects, if they occur during the sequence shown. capturing implementation details useful for helping people to write or understand code or to help in testing the software	

TIPS

- Create 'class' only if attributes will change
- .get = retrieve information / for printing or viewing. Same as Select from of SQL
- .set=modify/update/add/delete

CLASS DIAGRAMS

USE VISIBILITY MARKERS TO SIGNIFY WHO CAN ACCESS THE INFORMATION CONTAINED WITHIN A CLASS. PRIVATE VISIBILITY, DENOTED WITH A - SIGN, HIDES INFORMATION FROM ANYTHING OUTSIDE THE CLASS PARTITION. PUBLIC VISIBILITY, DENOTED WITH A + SIGN, ALLOWS ALL OTHER CLASSES TO VIEW THE MARKED INFORMATION. PROTECTED VISIBILITY, DENOTED WITH A # SIGN, ALLOWS CHILD CLASSES TO ACCESS INFORMATION THEY INHERITED FROM A PARENT CLASS.

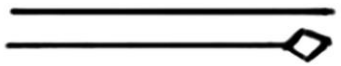
HTTPS://WWW.YOUTUBE.COM/WATCH?V=E3LEGOXEQUM

CLASS DIAGRAM: RELATIONSHIPS

Describe interections between objects

Dependencies : X uses Y

Associations/ Aggregations : X has a Y

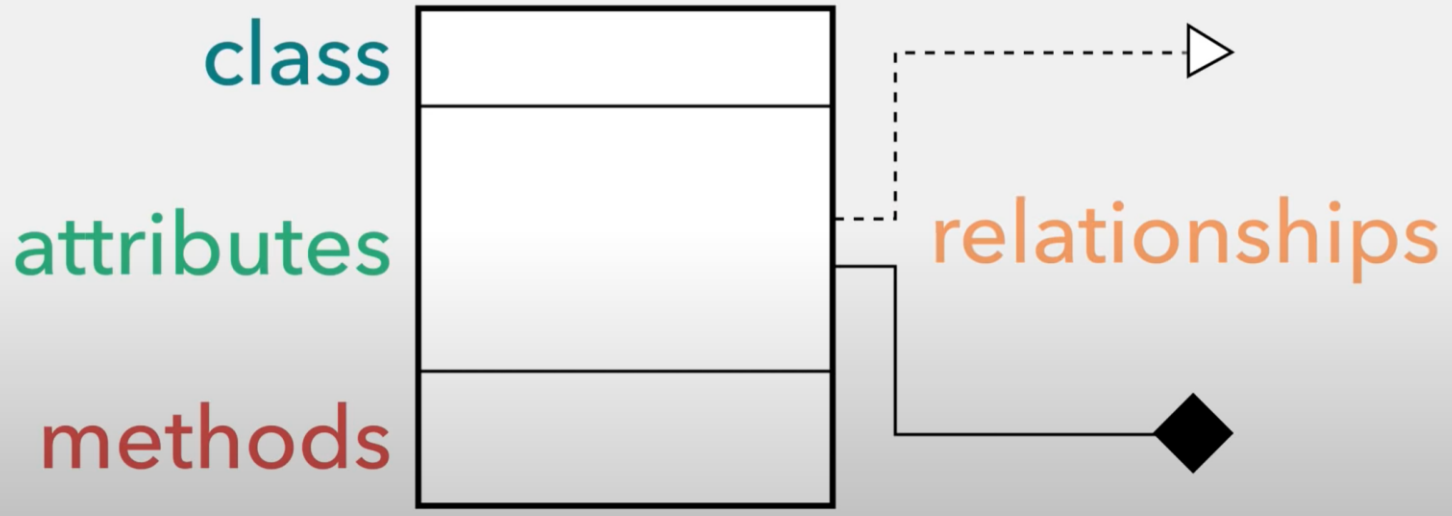


Generelization : X is a Y

And this can be expressed as x is a

UML Class Diagram Tutorial

UML CLASS DIAGRAMS

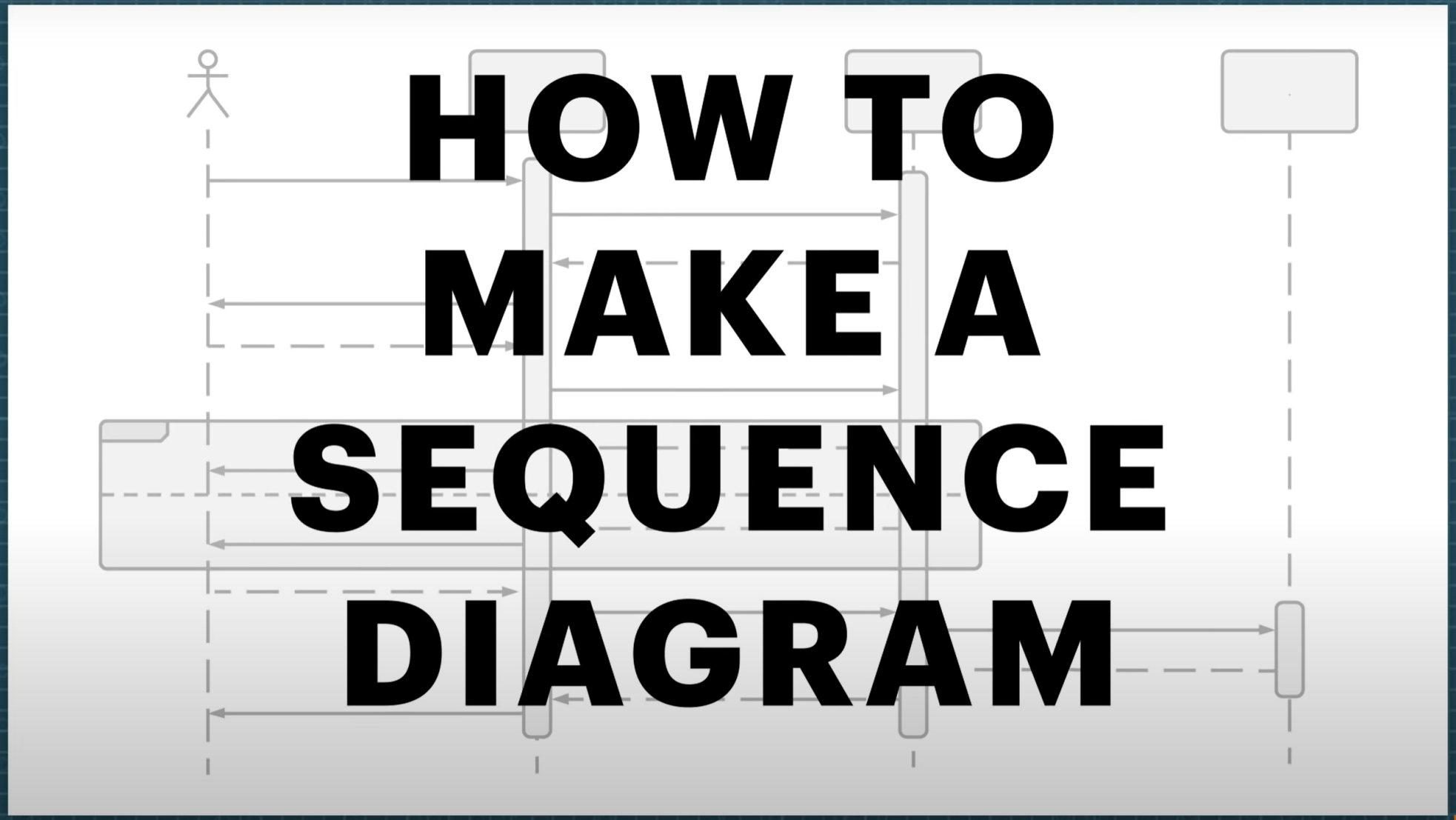


Then we'll talk about relationships.

0:07 / 10:16

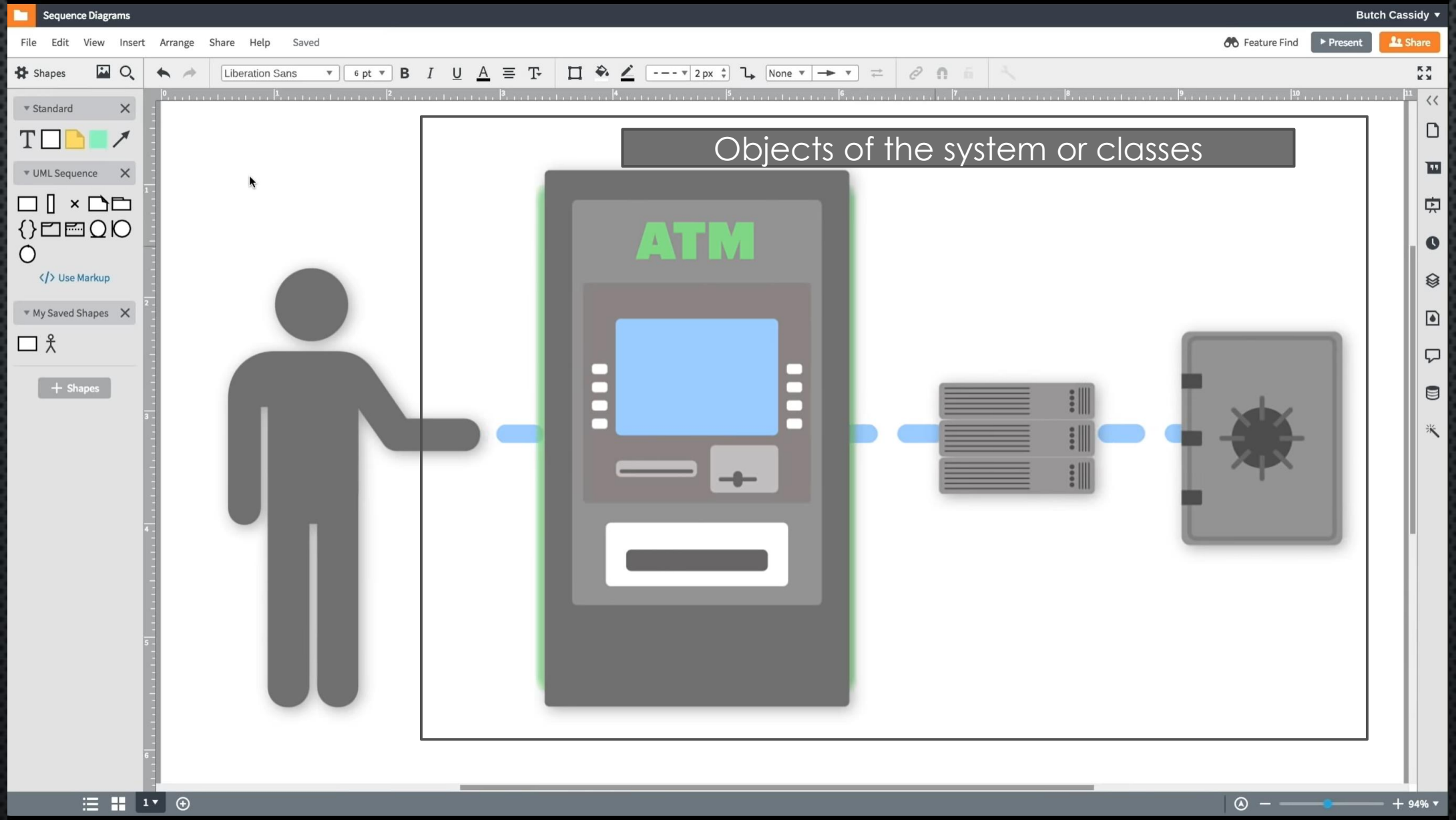
Scroll for details

How to Make a UML Sequence Diagram

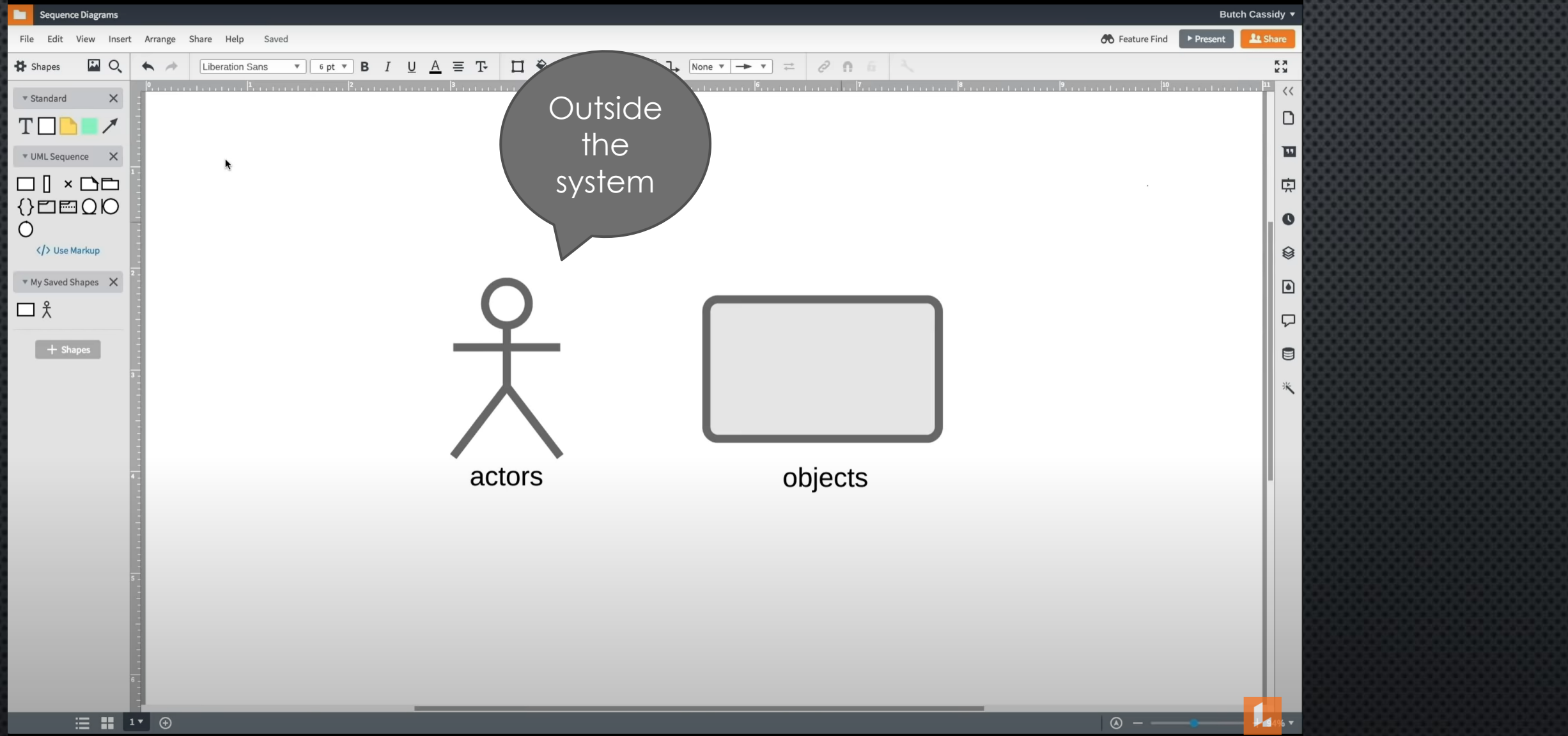


0:43 / 8:37

Scroll for details


Sequence Diagrams
Butch Cassidy
File Edit View Insert Arrange Share Help Saved
Feature Find
Present
Share
Shapes
Liberation Sans
6 pt
2 px
None
Standard
UML Sequence
Use Markup
My Saved Shapes
Shapes
Objects of the system or classes
ATM
1
94%

How to Make a UML Sequence Diagram

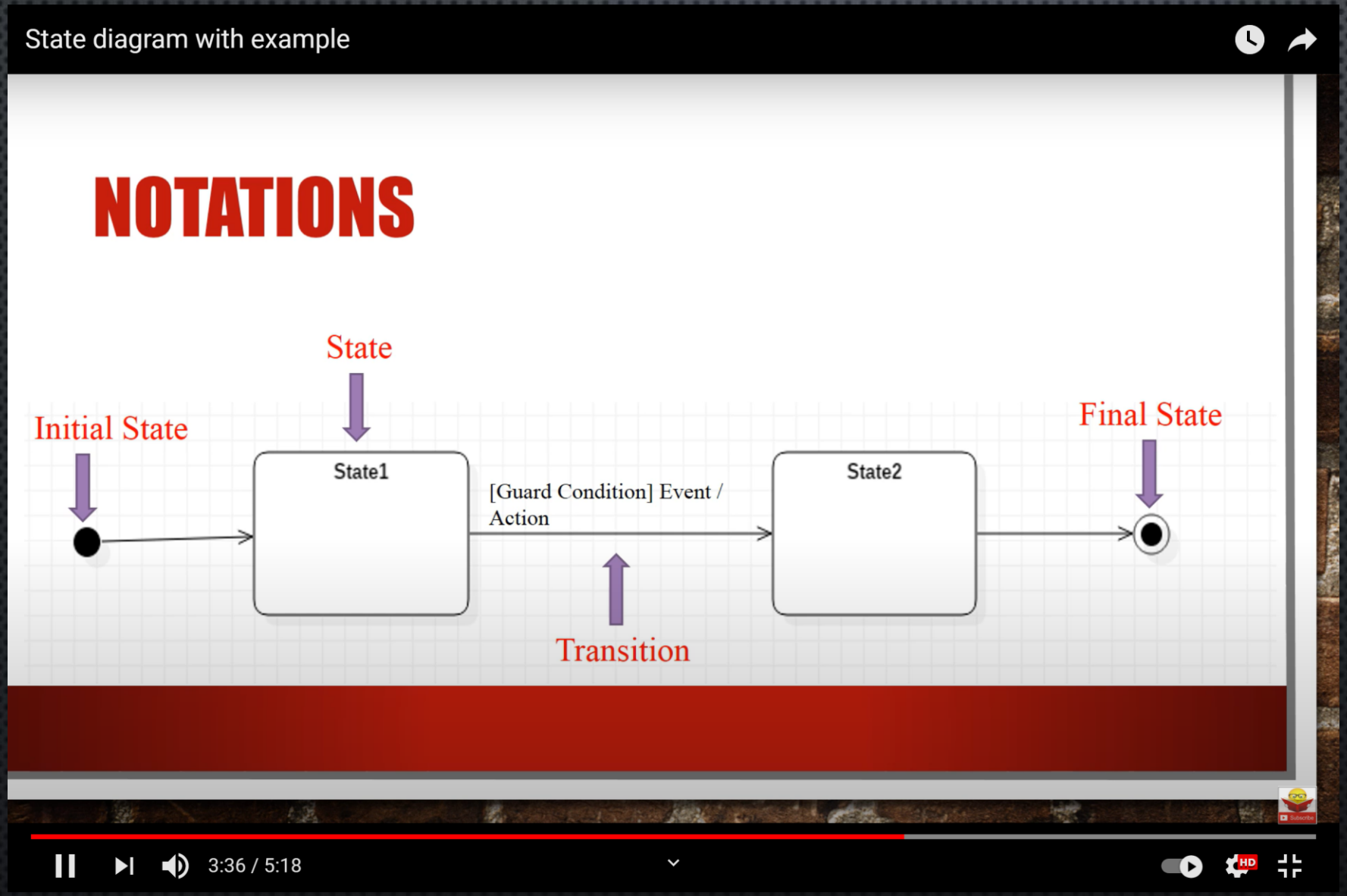


1:56 / 8:37

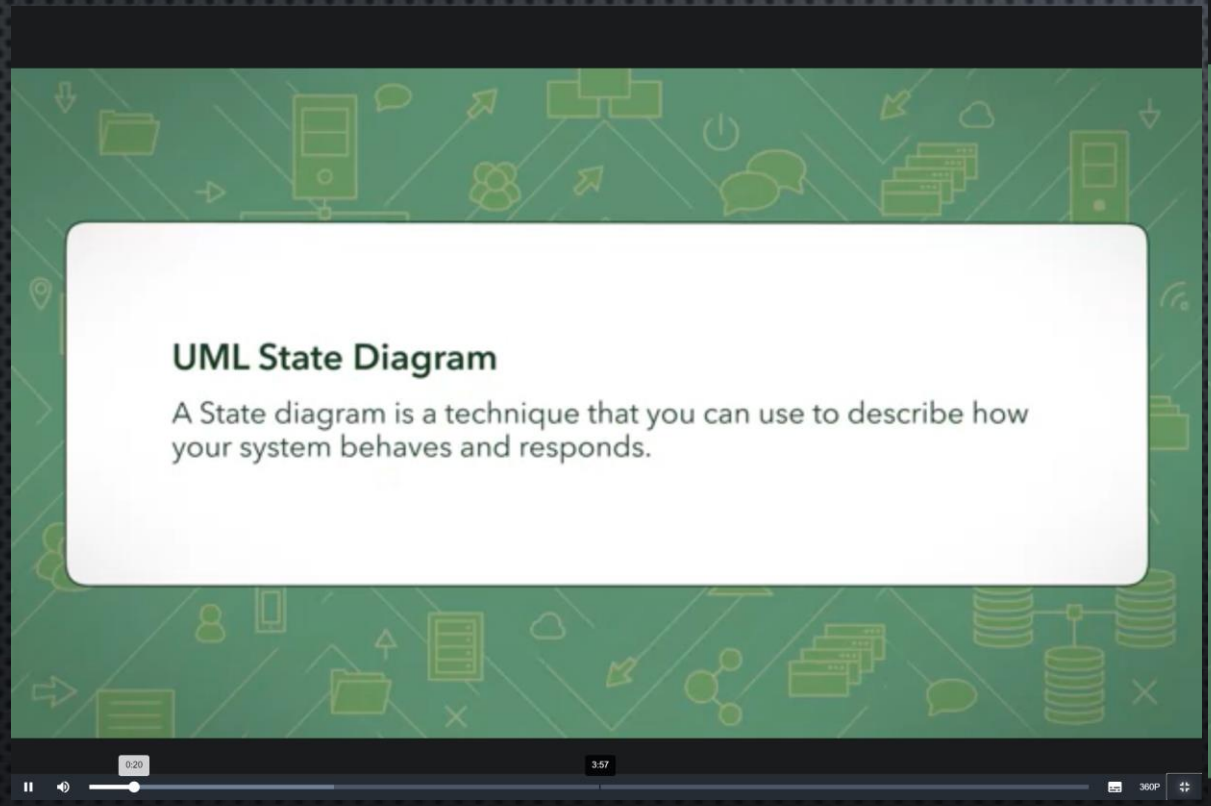
Scroll for details

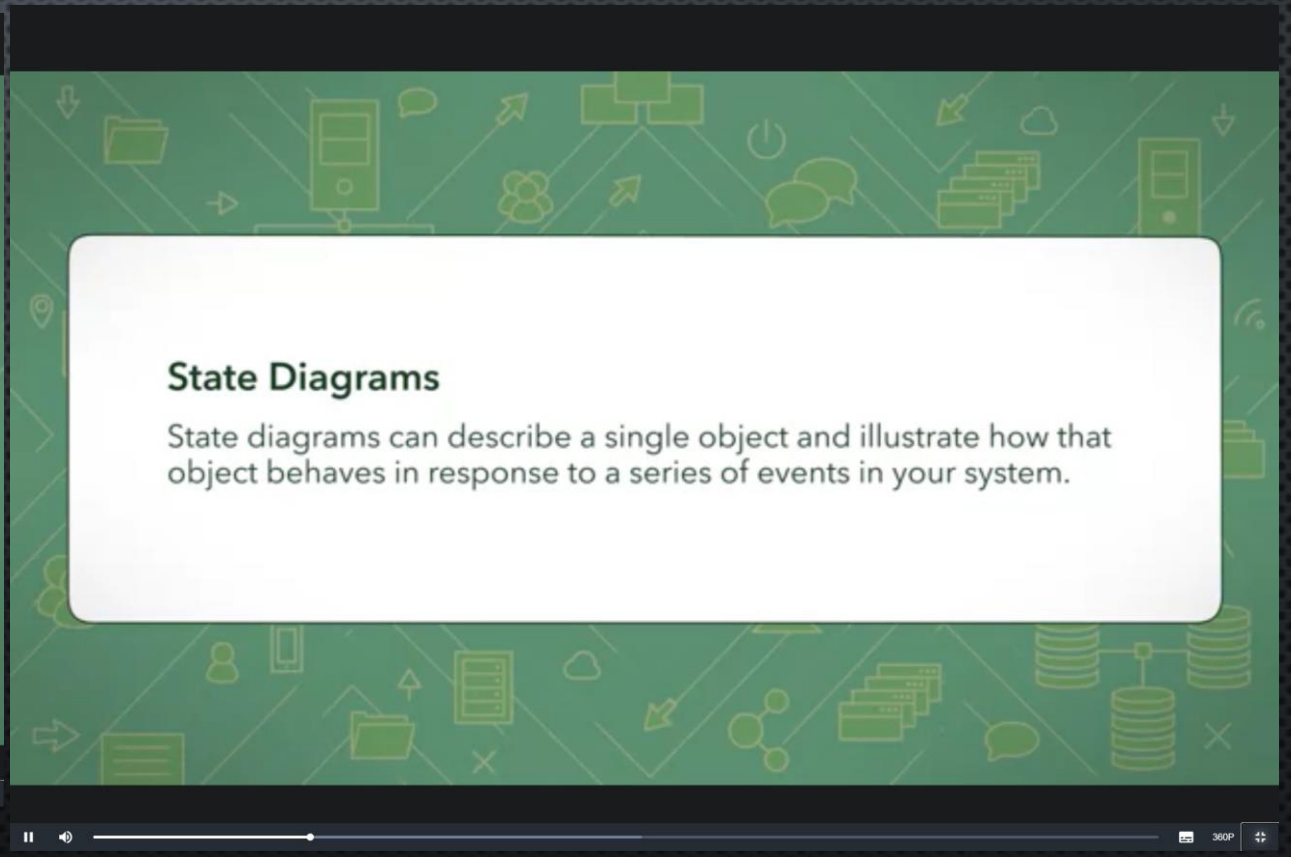
STATE DIAGRAM

HTTPS://WWW.YOUTUBE.COM/WATCH?V=L9UCSQXUWMW

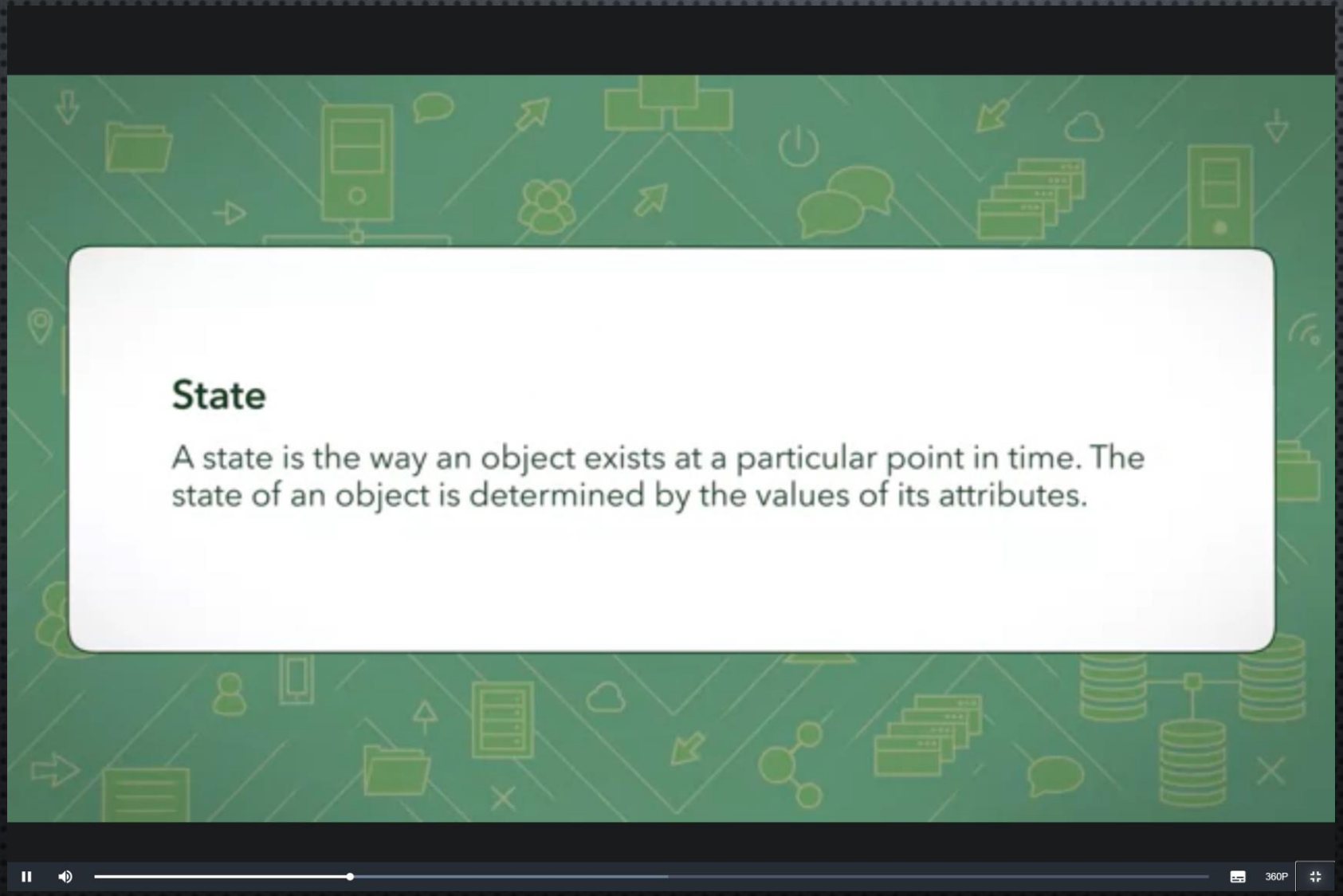


HTTPS://WWW.COURSERA.ORG/LECTURE/OBJECT-ORIENTED-DESIGN/1-3-7-UML-STATE-DIAGRAM-UC1H1





STATE DIAGRAMS HELP YOU FIND ISSUES; SOMETHING WHICH WAS NOT PLANNED ETC



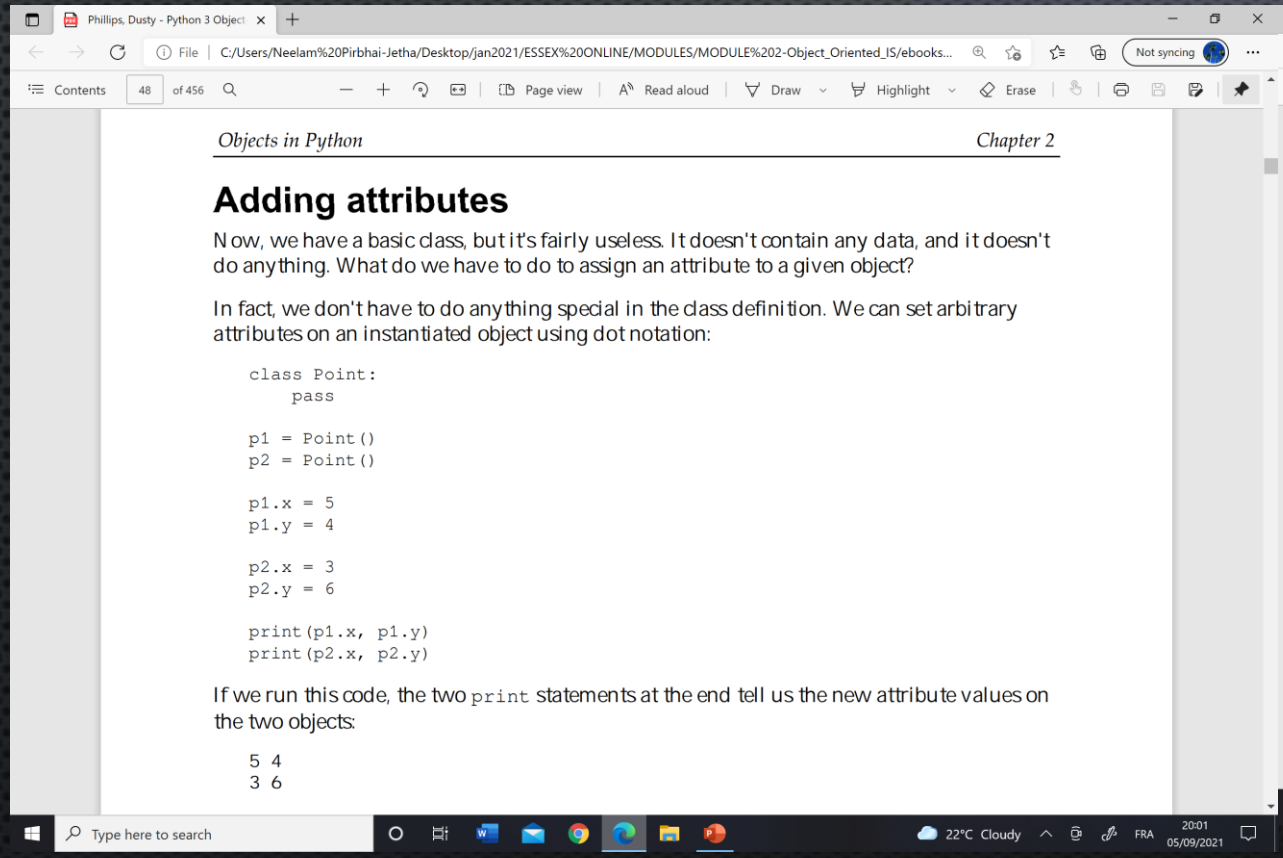
Chapter 2: Objects in Python

Book Source:

PHILIPS, D. (2018) CHAPTERS 2&3 *PYTHON 3 OBJECT-ORIENTED PROGRAMMING.* 3RD ED. PACKT PUBLISHING.

- The class definition starts with the class keyword. This is followed by a name (of our choice) identifying the class, and is terminated with a colon.
- Example:
- class MyFirstClass:

ADDING ATTRIBUTES
ALL WE NEED TO DO TO ASSIGN A VALUE TO AN ATTRIBUTE ON AN OBJECT IS USE THE



Objects in Python *Chapter 2*

Adding attributes

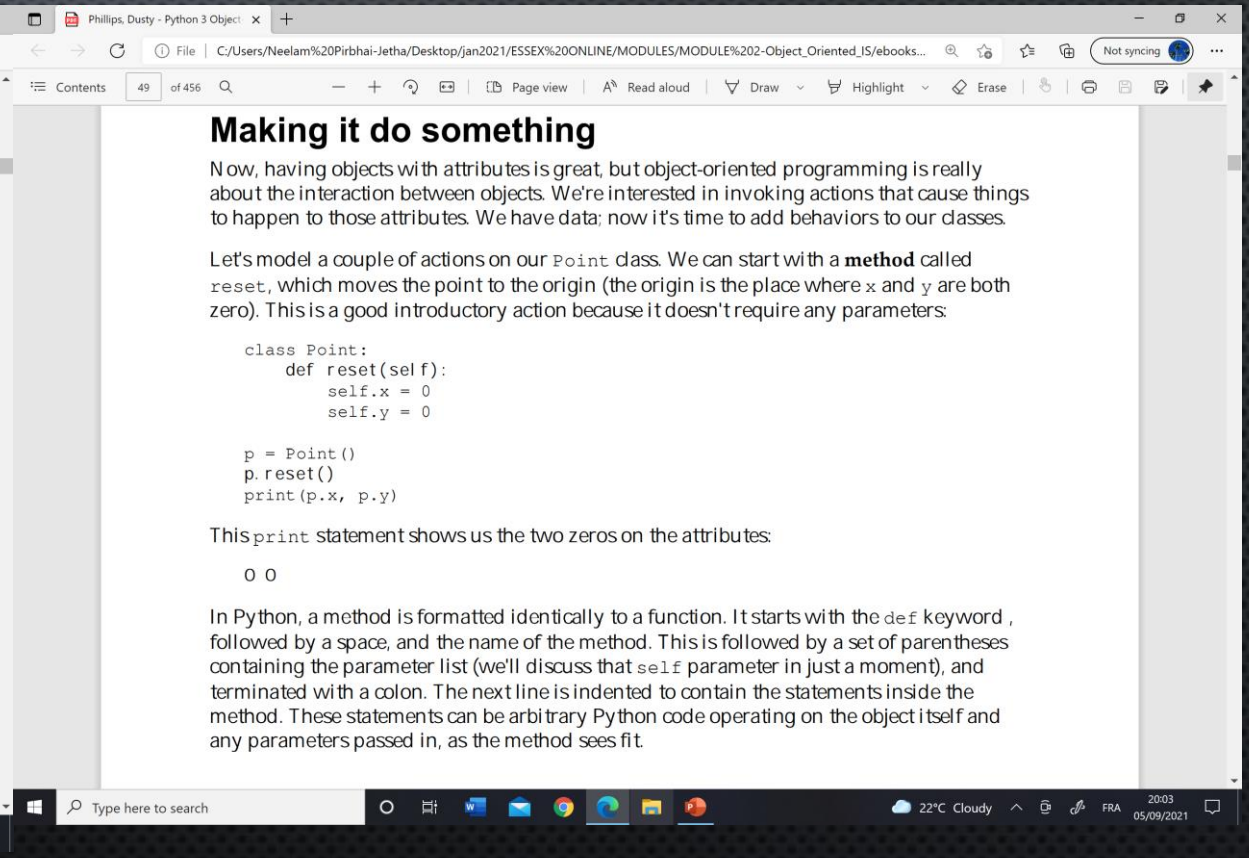
Now, we have a basic class, but it's fairly useless. It doesn't contain any data, and it doesn't do anything. What do we have to do to assign an attribute to a given object?

In fact, we don't have to do anything special in the class definition. We can set arbitrary attributes on an instantiated object using dot notation:

```
class Point:
    pass

p1 = Point()
p2 = Point()

p1.x = 5
p1.y = 4

p2.x = 3
p2.y = 6

print(p1.x, p1.y)
print(p2.x, p2.y)
```

If we run this code, the two `print` statements at the end tell us the new attribute values on the two objects:

```
5 4
3 6
```



Making it do something

Now, having objects with attributes is great, but object-oriented programming is really about the interaction between objects. We're interested in invoking actions that cause things to happen to those attributes. We have data; now it's time to add behaviors to our classes.

Let's model a couple of actions on our `Point` class. We can start with a **method** called `reset`, which moves the point to the origin (the origin is the place where `x` and `y` are both zero). This is a good introductory action because it doesn't require any parameters:

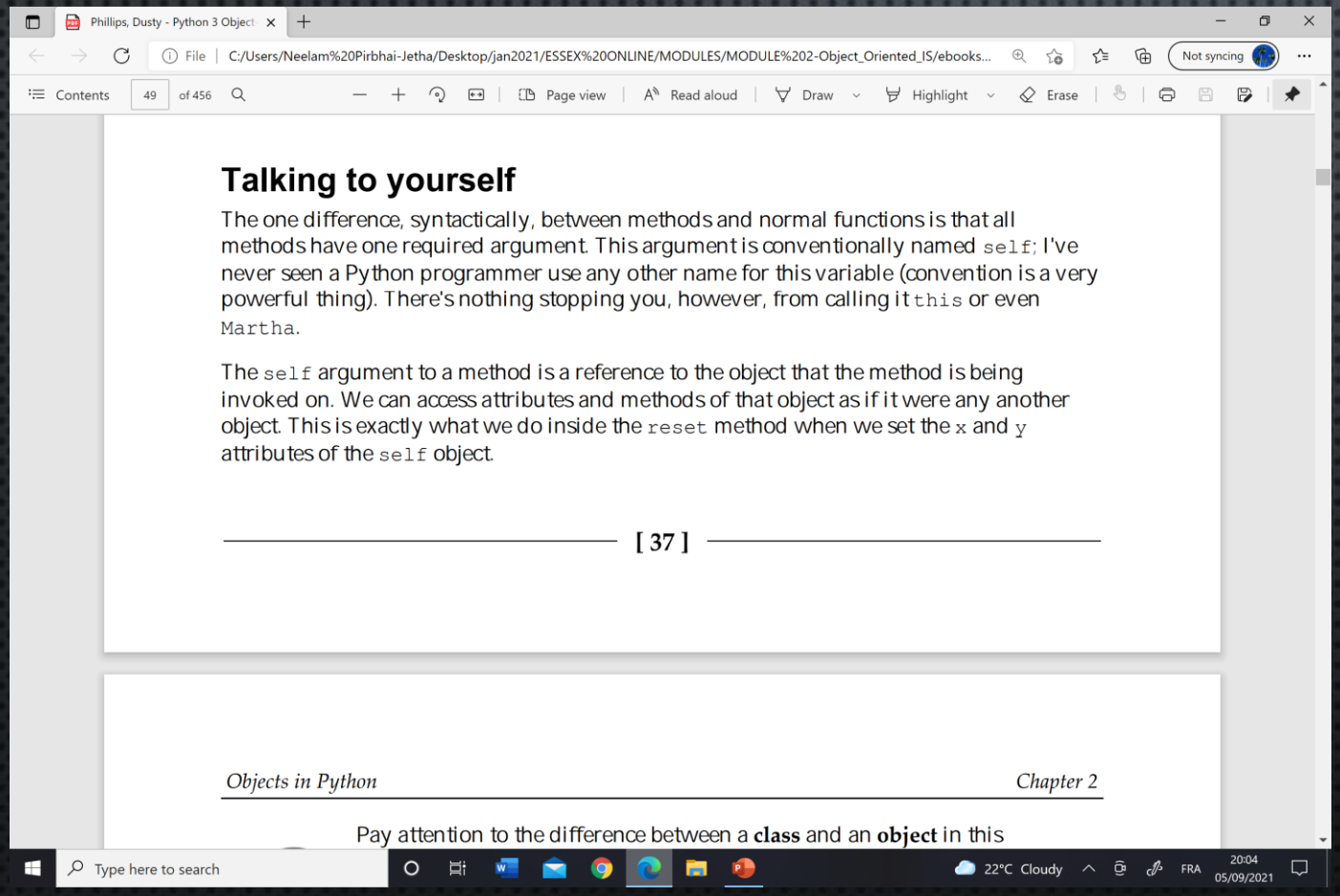
```
class Point:
    def reset(self):
        self.x = 0
        self.y = 0

p = Point()
p.reset()
print(p.x, p.y)
```

This `print` statement shows us the two zeros on the attributes:

```
0 0
```

In Python, a method is formatted identically to a function. It starts with the `def` keyword, followed by a space, and the name of the method. This is followed by a set of parentheses containing the parameter list (we'll discuss that `self` parameter in just a moment), and terminated with a colon. The next line is indented to contain the statements inside the method. These statements can be arbitrary Python code operating on the object itself and any parameters passed in, as the method sees fit.

‘SELF’ ARGUMENT



Some Explanations

- Python supports this through the use of docstrings. Each class, function, or method header can have a standard Python string as the first line following the definition (the line that ends in a colon).
- This line should be indented the same as the code that follows it. Docstrings are simply Python strings enclosed with apostrophes (') or quotation marks (") characters. Often, docstrings are quite long and span multiple lines (the style guide suggests that the line length should not exceed 80 characters), which can be formatted as multi-line strings, enclosed in matching triple apostrophe (''') or triple quote (""") characters. A docstring should clearly and concisely summarize the purpose of the class or method it is describing. It should explain any parameters whose usage is not immediately obvious, and is also a good place to include short examples of how to use the API. Any caveats or problems an unsuspecting user of the API should be aware of should also be noted.

. Terms to explain

- create classes and instantiate objects
- modules. Modules are simply Python files, nothing more. The single file in our small program is a module. Two Python files are two modules. If we have two files in the same folder, we can load a class from one module for use in the other module.
- Some sources say that we can import all classes and functions from the database module using this syntax: from database import * Don't do this. Most experienced Python programmers will tell you that you should never use this syntax: Code maintenance becomes a nightmare.

- As a project grows into a collection of more and more modules, we may find that we want to add another level of abstraction, some kind of nested hierarchy on our modules' levels. However, we can't put modules inside modules; one file can hold only one file after all, and modules are just files. Files, however, can go in folders, and so can modules. A package is a collection of modules in a folder. The name of the package is the name of the folder. We need to tell Python that a folder is a package to distinguish it from other folders in the directory. To do this, place a (normally empty) file in the folder named __init__.py. If we forget this file, we won't be able to import modules from that folder.

able to import modules from that folder.

Let's put our modules inside an `ecommerce` package in our working folder, which will also contain a `main.py` file to start the program. Let's additionally add another package inside the `ecommerce` package for various payment options. The folder hierarchy will look like this:

```
parent_directory/
    main.py
    ecommerce/
        __init__.py
        database.py
        products.py
        payments/
            __init__.py
            square.py
            stripe.py
```

Chapter 3: When Objects Are Alike

- Technically, every class we create uses inheritance. All Python classes are subclasses of the special built-in class named object.
- Code for inheritance: Simply include the name of the parent class inside parentheses between the class name and the colon that follows.

```
class MySubClass(object):
```

Some terms explaines…

- Inheritance is great for adding new behavior to existing classes, but what about changing behavior?
- Multiple inheritance
- Polymorphism is one of the most important reasons to use inheritance in many objectoriented contexts. Because any objects that supply the correct interface can be used interchangeably in Python, it reduces the need for polymorphic common superclasses. Inheritance can still be useful for sharing code, but if all that is being shared is the public interface, duck typing is all that is required. This reduced need for inheritance also reduces the need for multiple inheritance; often, when multiple inheritance appears to be a valid solution, we can just use duck typing to mimic one of the multiple superclasses.

Some terms explained

- If you aren't familiar with the **kwargs syntax, it basically collects any keyword arguments passed into the method that were not explicitly listed in the parameter list. These arguments are stored in a dictionary named kwargs (we can call the variable whatever we like, but convention suggests kw, or kwargs). When we call a different method (for example, super().__init__) with a **kwargs syntax, it unpacks the dictionary and passes the results to the method as normal keyword arguments.

STEP-BY-STEP GUIDE TO PYTHON OBJECT-ORIENTED PROGRAMMING

#create class

#class: allow us to logically group data and functions in order to re-use, to build upon

#attribute

#Method=function associated with a class

Source Schafer, C. (June 20, 2016) *Python OOP*

Available from: https://www.youtube.com/watch?v=ZDa-Z5JzLYM [Accessed 06 September 2021]

Tutorial 1:Classes an Instances.

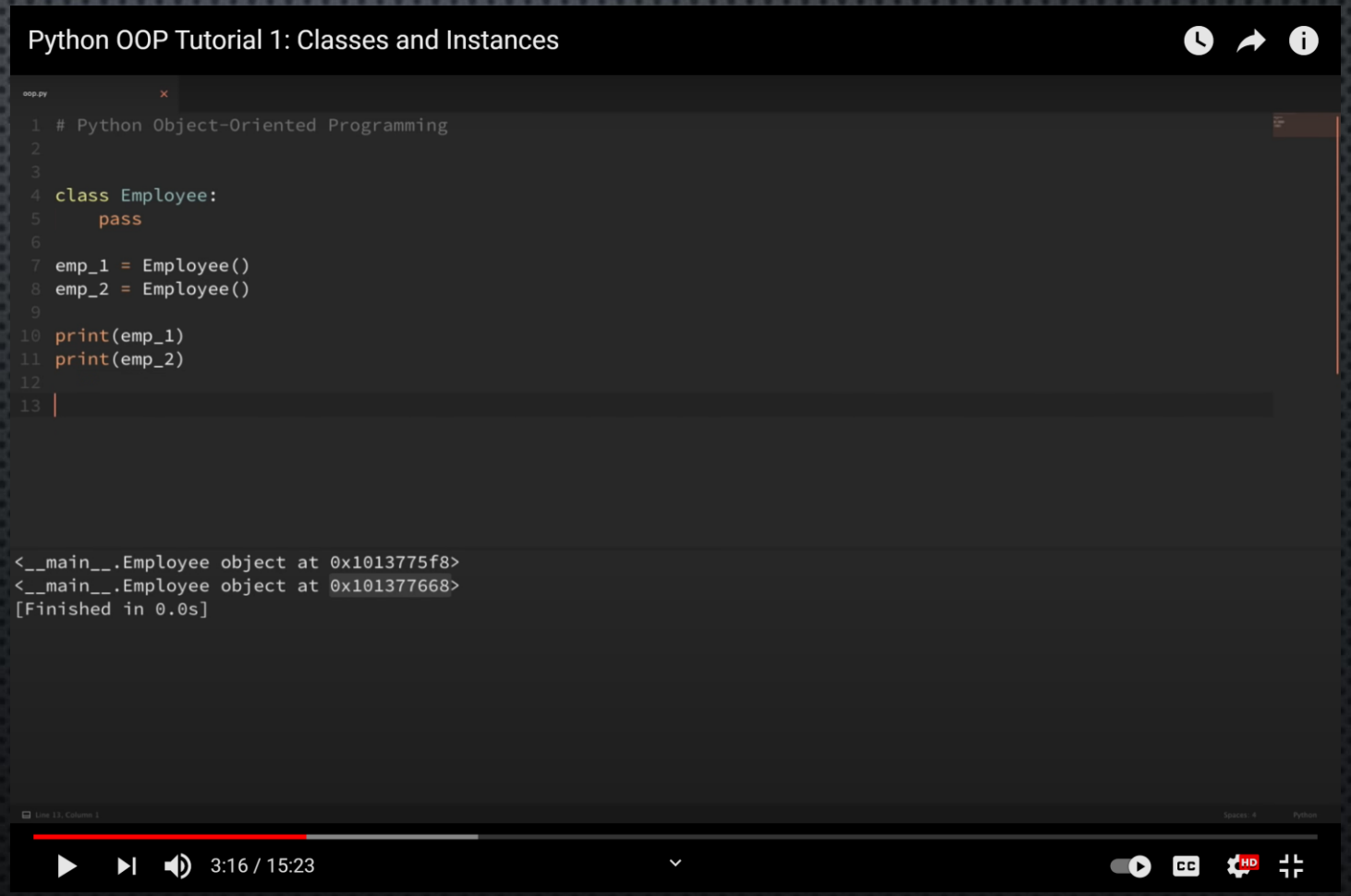
Check also: https://www.youtube.com/watch?v=-pEs-Bss8Wc

E.G CREATE CLASS ‘EMPLOYEES’ FOR A COMPANY

```
#create a class
Class Employee:
#leave it blank for later and for Python not to run it
    pass

#class vs instance of a class
#class = blueprint for creating instances
#employee1=instance of employee class
```

CREATE INSTANCE VARIABLES

```
emp_1 = Employee()
emp_2 = Employee()

print(emp_1)
print(emp_2)
```



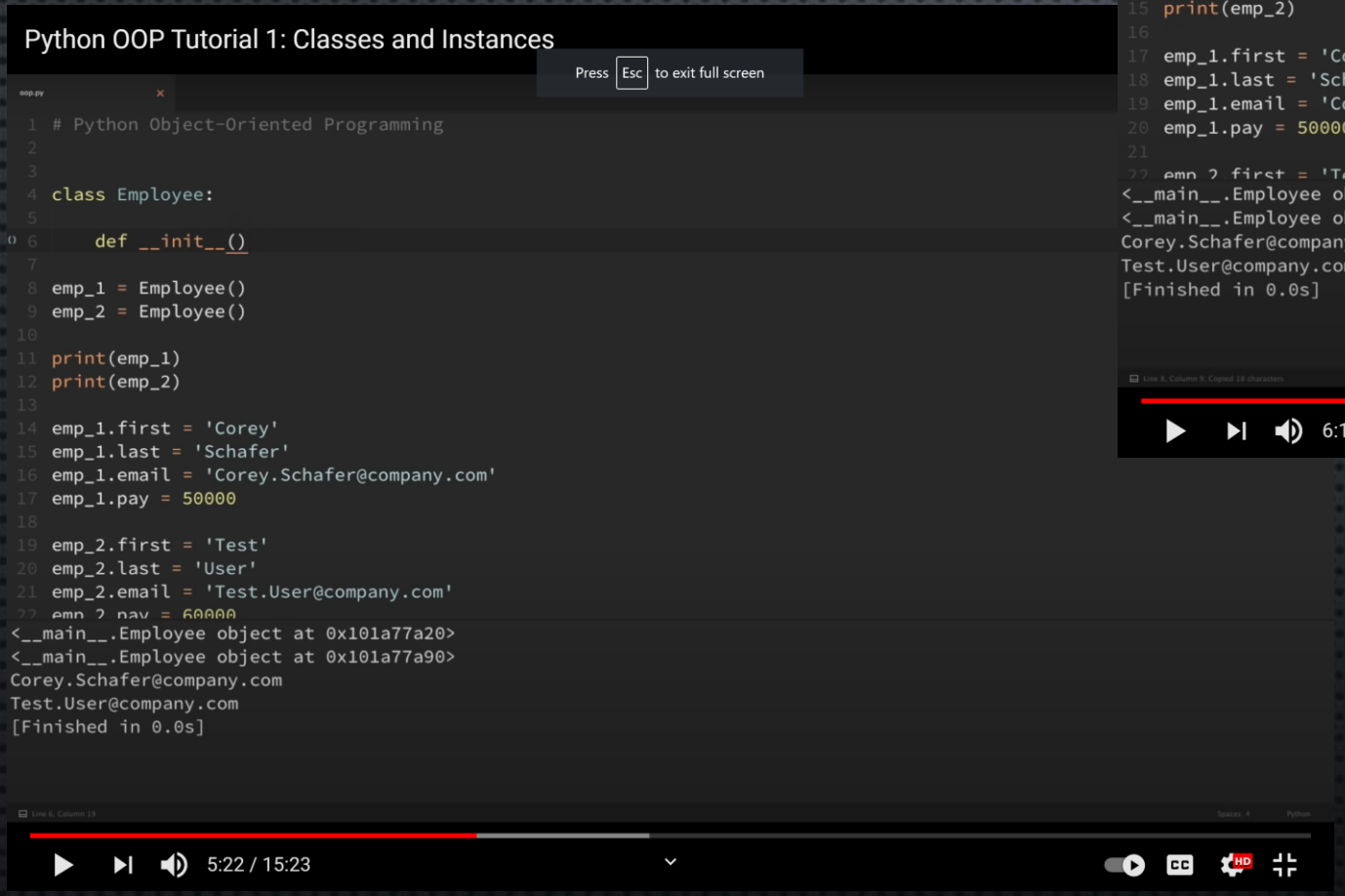
DATA UNIQUE TO EACH INSTANCE (E.G EACH EMPLOYEE HAS AN ATTRIBUTE UNIQUE TO THEM SUCH AS FIRST_NAME, LAST_NAME, EMA

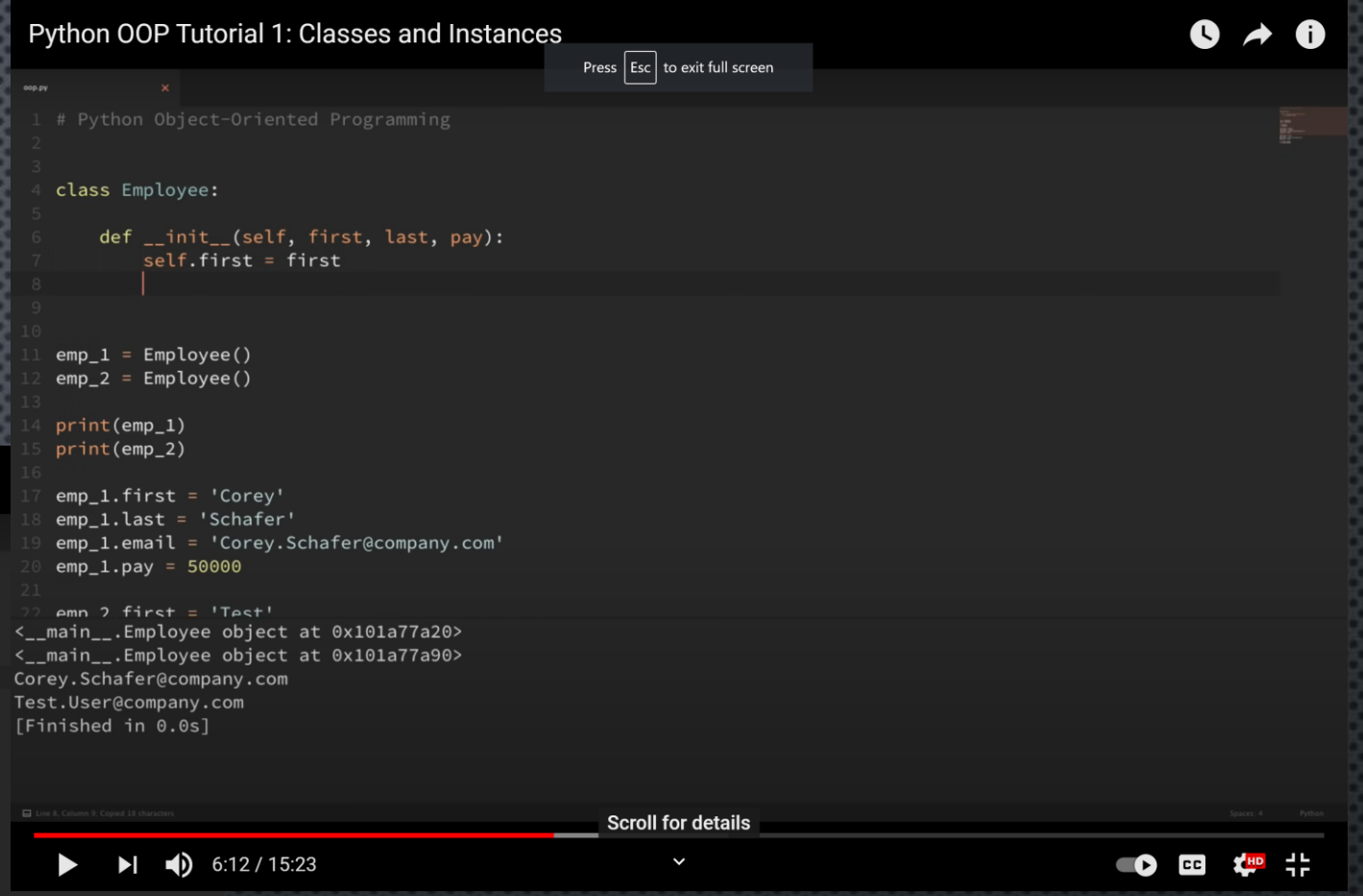
MANUAL INPUT OF DATA

```
EMP_1.FIRST="NEELAM"
EMP_1.LAST= "PIRBHAI"
EMP_1.EMAIL= "NEELAMPIRBHAI@HOTMAIL.COM"
EMP_1.PAY=50000

EMP_2.FIRST="TEST"
EMP_2.LAST= "USER"
EMP_2.EMAIL= "TEST.USER@HOTMAIL.COM"
EMP_2.PAY=60000
PRINT(EMP_1.EMAIL)
PRINT(EMP_2.EMAIL)
```

But this is the manual way of inputting data one by one

To do it automatically, under the Employee class, create init mthod.
def__init__ (self)
self=instance (meaning that it's same as emp_1.first etc. but input of data won't be done manually)
Add the arguments (such as first_name etc.)
See next pages


Python OOP Tutorial 1: Classes and Instances
Press Esc to exit full screen
oop.py
# Python Object-Oriented Programming
class Employee:
def __init__()
emp_1 = Employee()
emp_2 = Employee()
print(emp_1)
print(emp_2)
emp_1.first = 'Corey'
emp_1.last = 'Schafer'
emp_1.email = 'Corey.Schafer@company.com'
emp_1.pay = 50000
emp_2.first = 'Test'
emp_2.last = 'User'
emp_2.email = 'Test.User@company.com'
<__main__.Employee object at 0x101a77a20>
<__main__.Employee object at 0x101a77a90>
Corey.Schafer@company.com
Test.User@company.com
[Finished in 0.0s]
5:22 / 15:23


Python OOP Tutorial 1: Classes and Instances
Press Esc to exit full screen
oop.py
# Python Object-Oriented Programming
class Employee:
def __init__(self, first, last, pay):
self.first = first
emp_1 = Employee()
emp_2 = Employee()
print(emp_1)
print(emp_2)
emp_1.first = 'Corey'
emp_1.last = 'Schafer'
emp_1.email = 'Corey.Schafer@company.com'
emp_1.pay = 50000
<__main__.Employee object at 0x101a77a20>
<__main__.Employee object at 0x101a77a90>
Corey.Schafer@company.com
Test.User@company.com
[Finished in 0.0s]
Scroll for details
6:12 / 15:23

- For the email, create a code to automatically create an email for each employee

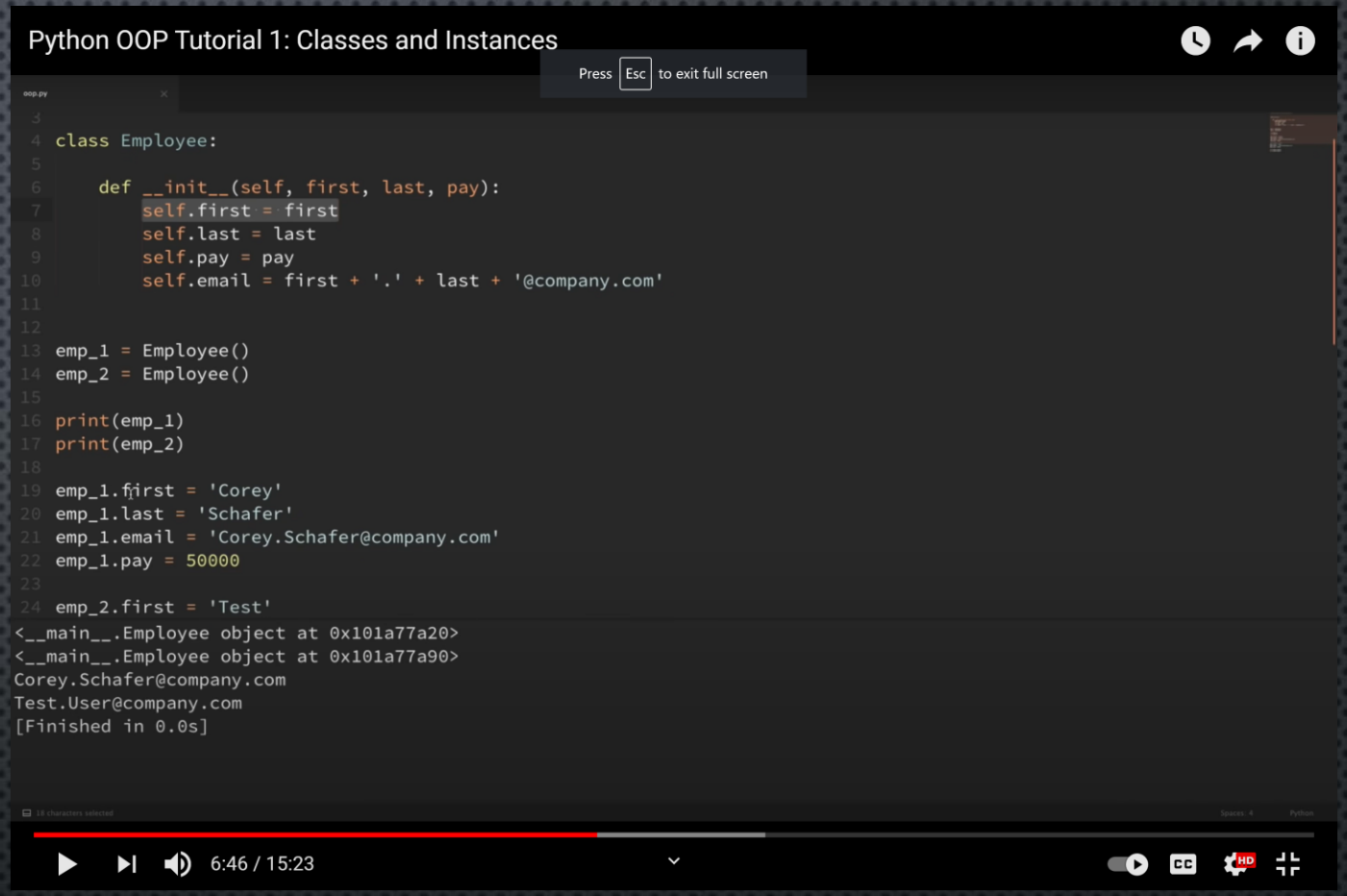
self.email=first + ' . ' + last+' @company.com

IMPORTANT

After this, you should pass on information in the same order you've defined the class

emp_1 = Employee('Neelam', 'Pirbhai', 50000) #Email will be created automatically

emp_2 = Employee('Test', 'User, 60000)



oop.py

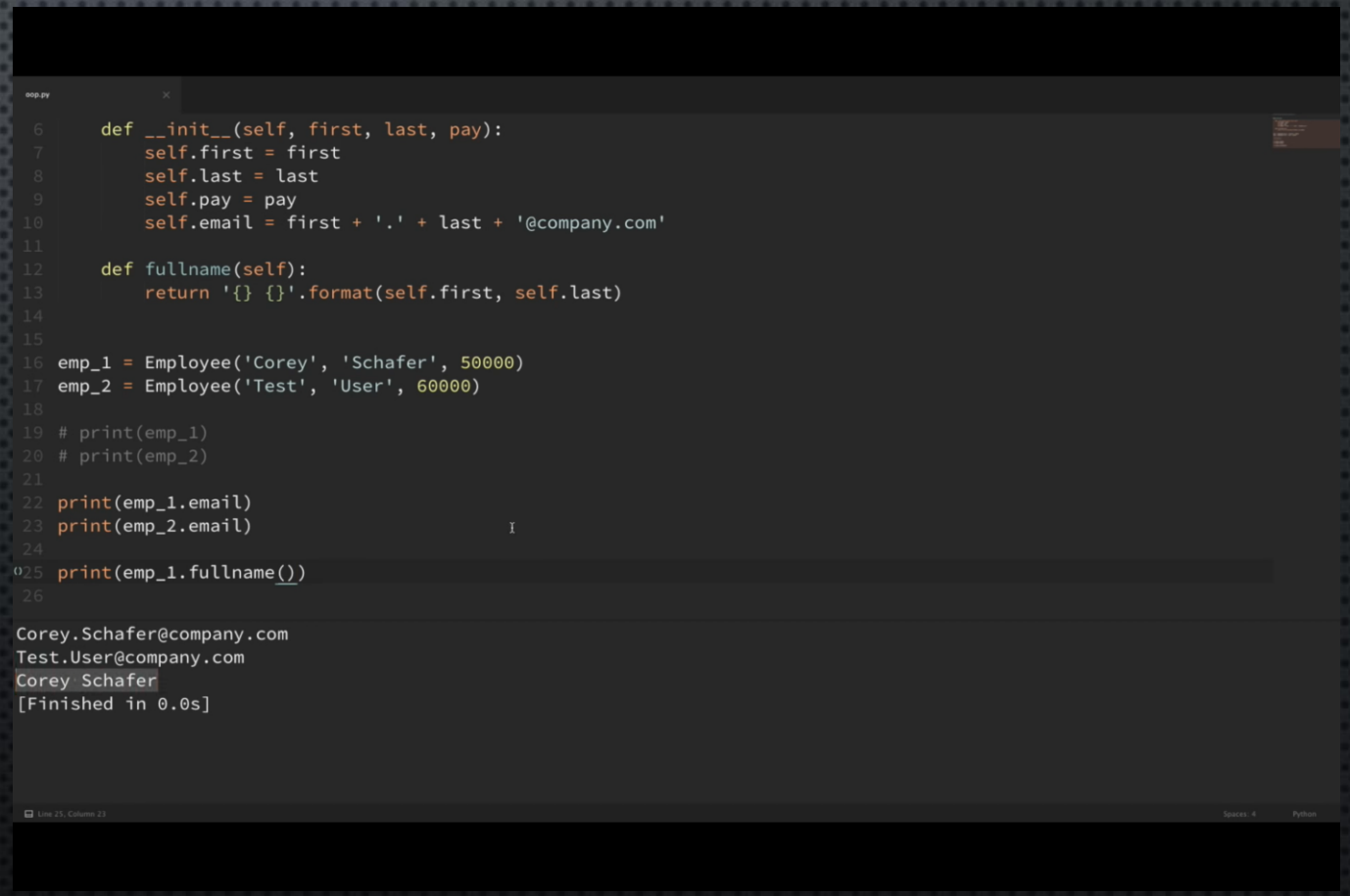
```python
# Python Object-Oriented Programming


class Employee:

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

# print(emp_1)
# print(emp_2)

print(emp_1.email)
print(emp_2.email)

```

```
Corey.Schafer@company.com
Test.User@company.com
[Finished in 0.0s]
```

Line 17, Column 15 Spaces: 4 Python

To perform some actions, add methods to the class

- To get the full name, for instance.
- In the class employee, create a method called
- def fullname(self):
  - Return '{} {}'.format(se.f.first, self.last)
  - Print(emp_1.fullname())



ANOTHER WAY OF DOING: PASS ON THE INSTANCE BY USING CLASS

EMPLOYEE.FULLNAME(EMP_1)

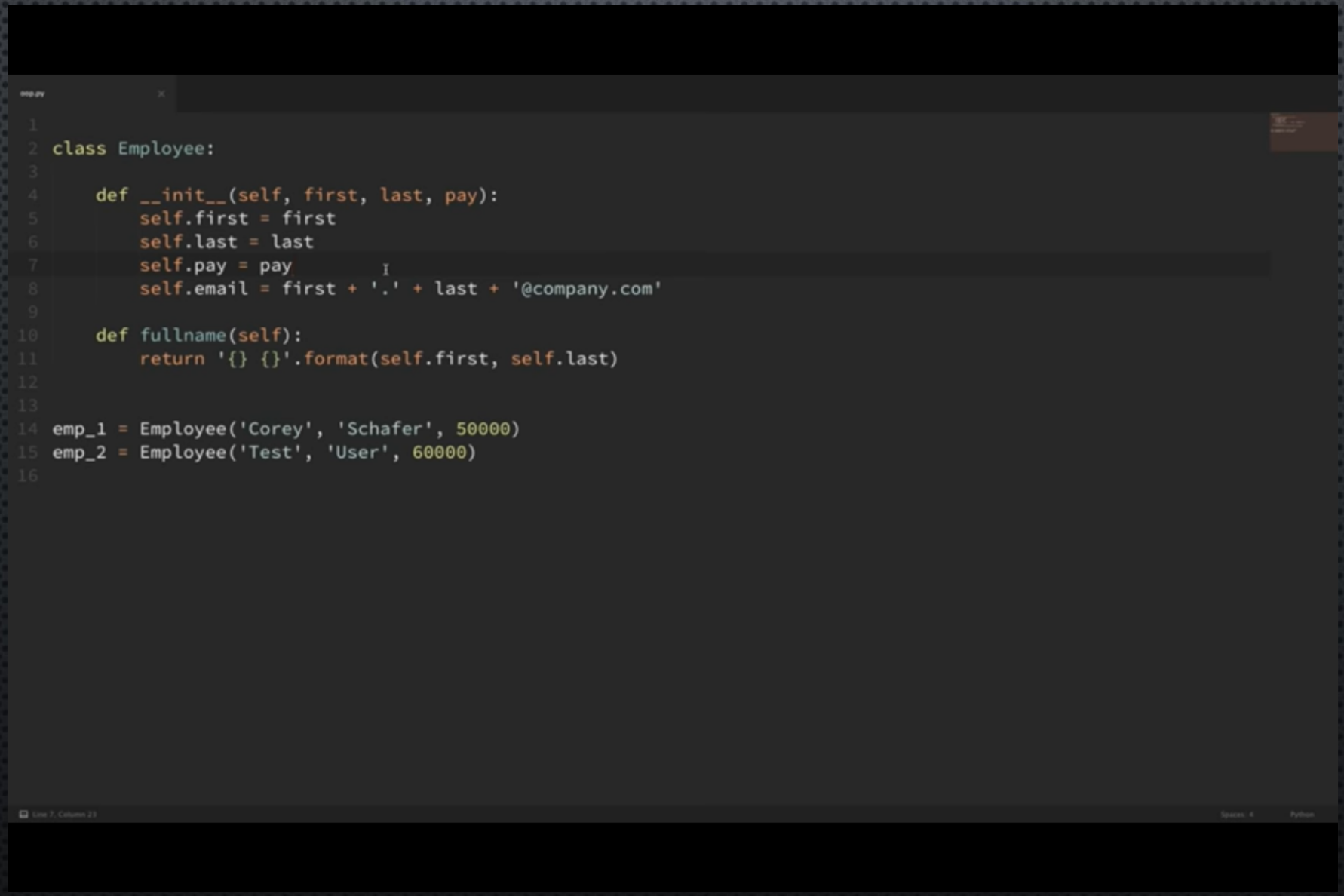
MEANS CALLING METHOD ON CLASS,

THEREFORE MUST PASS IN THE INSTANCE (EMP_1)

Same as:
emp_1.fullname()

Emp_1 = instance
.fullname = call method

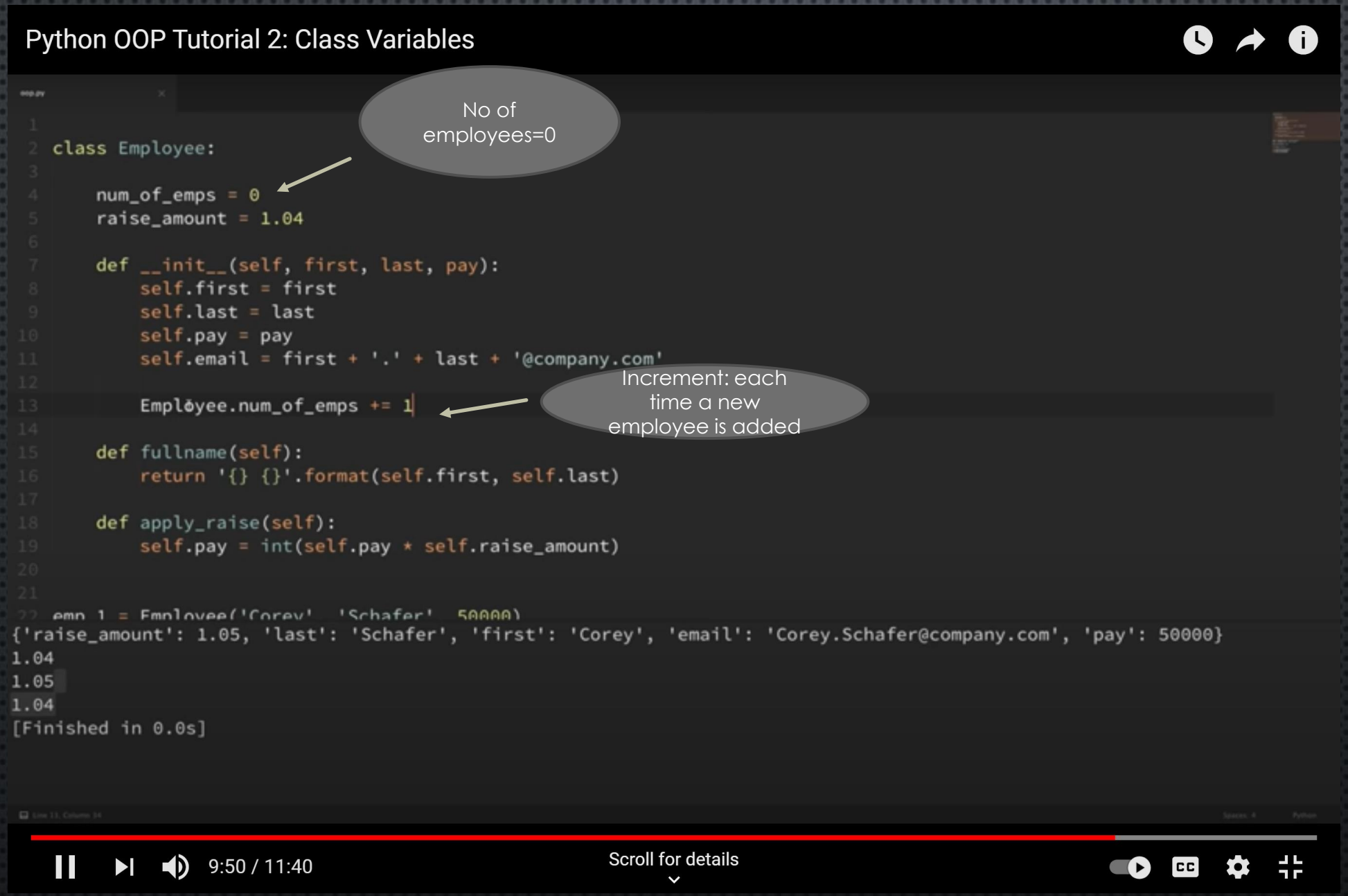
HTTPS://WWW.YOUTUBE.COM/WATCH?V=BJ-VVGYQXHO



```python
class Employee:

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay
        self.email = first + '.' + last + '@company.com'

    def fullname(self):
        return '{} {}'.format(self.first, self.last)


emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)
```

TO GET ALL DETAILS: _ _DICT_ _)

- PRINT(EMP_1._ _DICT_ _)


Python OOP Tutorial 2: Class Variables
No of employees=0
class Employee:
num_of_emps = 0
raise_amount = 1.04
def __init__(self, first, last, pay):
self.first = first
self.last = last
self.pay = pay
self.email = first + '.' + last + '@company.com'
Employee.num_of_emps += 1
Increment: each time a new employee is added
def fullname(self):
return '{} {}'.format(self.first, self.last)
def apply_raise(self):
self.pay = int(self.pay * self.raise_amount)
{'raise_amount': 1.05, 'last': 'Schafer', 'first': 'Corey', 'email': 'Corey.Schafer@company.com', 'pay': 50000}
1.04
1.05
1.04
[Finished in 0.0s]
9:50 / 11:40
Scroll for details

```
        Employee.num_of_emps += 1

    def fullname(self):
        return '{} {}'.format(self.first, self.last)

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)


print(Employee.num_of_emps)

emp_1 = Employee('Corey', 'Schafer', 50000)
emp_2 = Employee('Test', 'User', 60000)

print(Employee.num_of_emps)

```

To know no of employees

Employees created

Ans=0 as no employees were created

Same code written after input of data on employees Ans=2

```
0
2
[Finished in 0.0s]
```

REFERENCES

- Philips, D. (2018) Chapters 2&3 *Python 3 Object-Oriented programming.* 3rd ed. Packt Publishing.
- https://www.lucidchart.com/blog/types-of-UML-diagrams
- https://www.youtube.com/watch?v=pCK6prSq8aw
- https://stackoverflow.com/questions/38184182/whats-the-difference-between-activity-diagram-and-sequence-diagram
- https://www.quora.com/Whats-the-difference-between-an-activity-diagram-and-a-sequence-diagram-in-UML
- https://www.youtube.com/watch?v=JeznW_7DlB0
- https://www.youtube.com/watch?v=ZDa-Z5JzLYM
- 6 h class: Intro to python: https://www.youtube.com/watch?v=_uQrJ0TkZlc
- https://www.youtube.com/watch?v=-pEs-Bss8Wc

TO WATCH STEP BY STEP TO LEARN OOP (BY COREY SHAFER)

- Video1:
- https://www.youtube.com/watch?v=ZDa-Z5JzLYM
- Video2:
- https://www.youtube.com/watch?v=BJ-VvGyQxho
- Video'3: https://www.youtube.com/watch?v=rq8cL2XMM5M
- Video 4: https://www.youtube.com/watch?v=RSl87lqOXDE