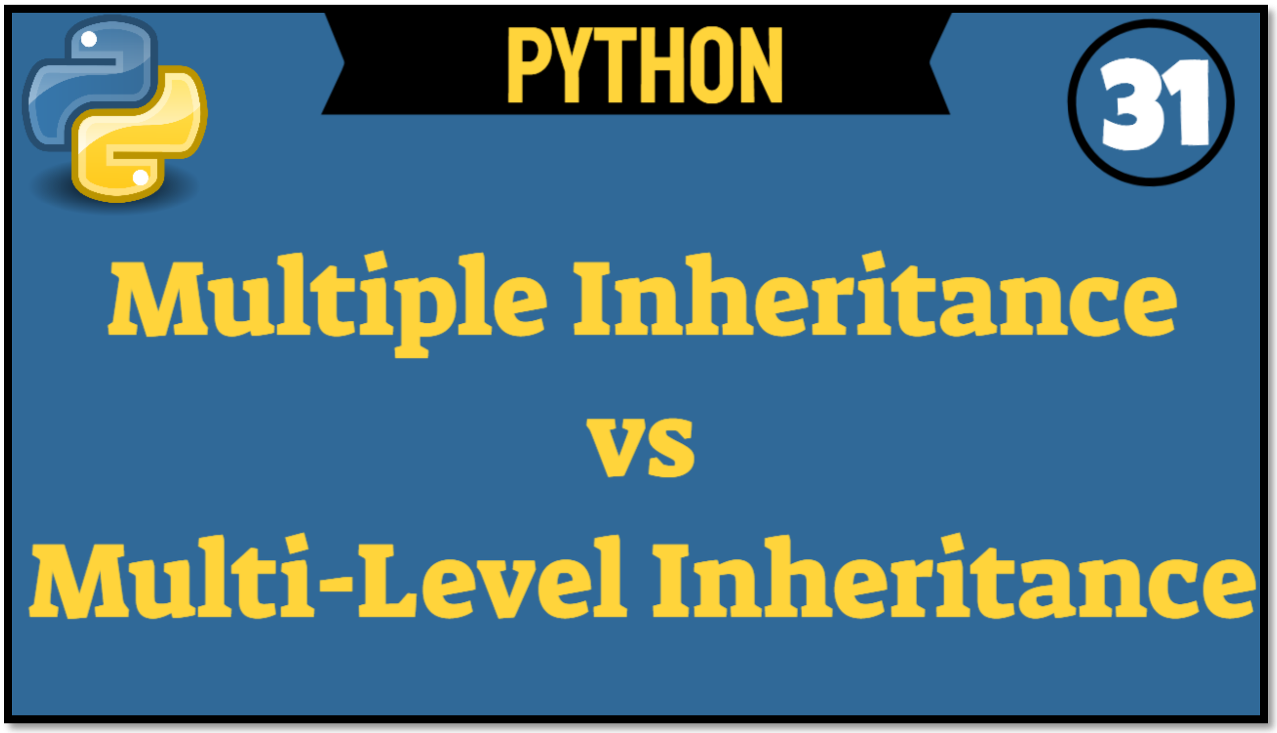
Rex
Jones II

Multiple vs Multi Level Inheritance



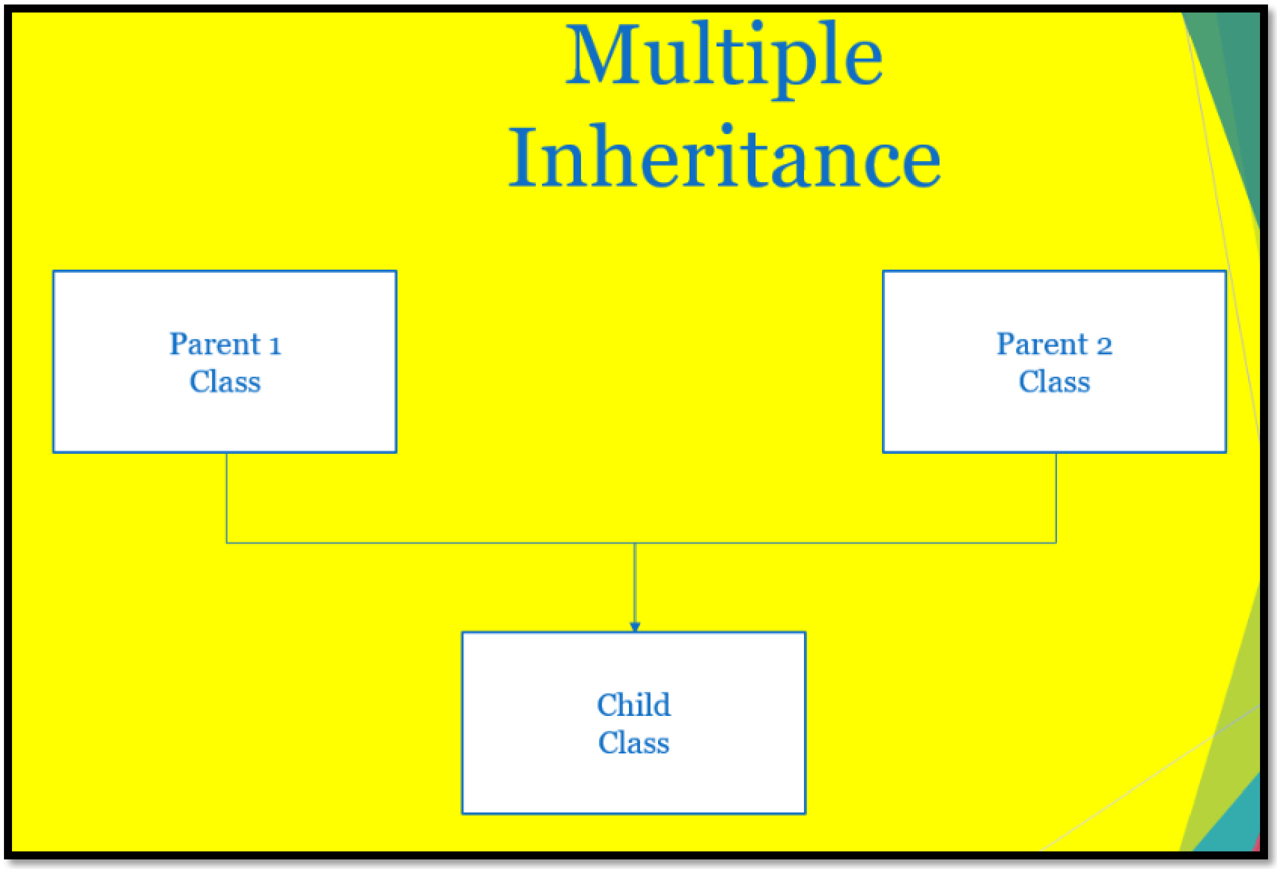
Python Video = https://youtu.be/lETGeteNt1o

In this session, I will continue from the previous Python session that talked about Inheritance. There is a difference between Multiple Inheritance and Multi-Level Inheritance.

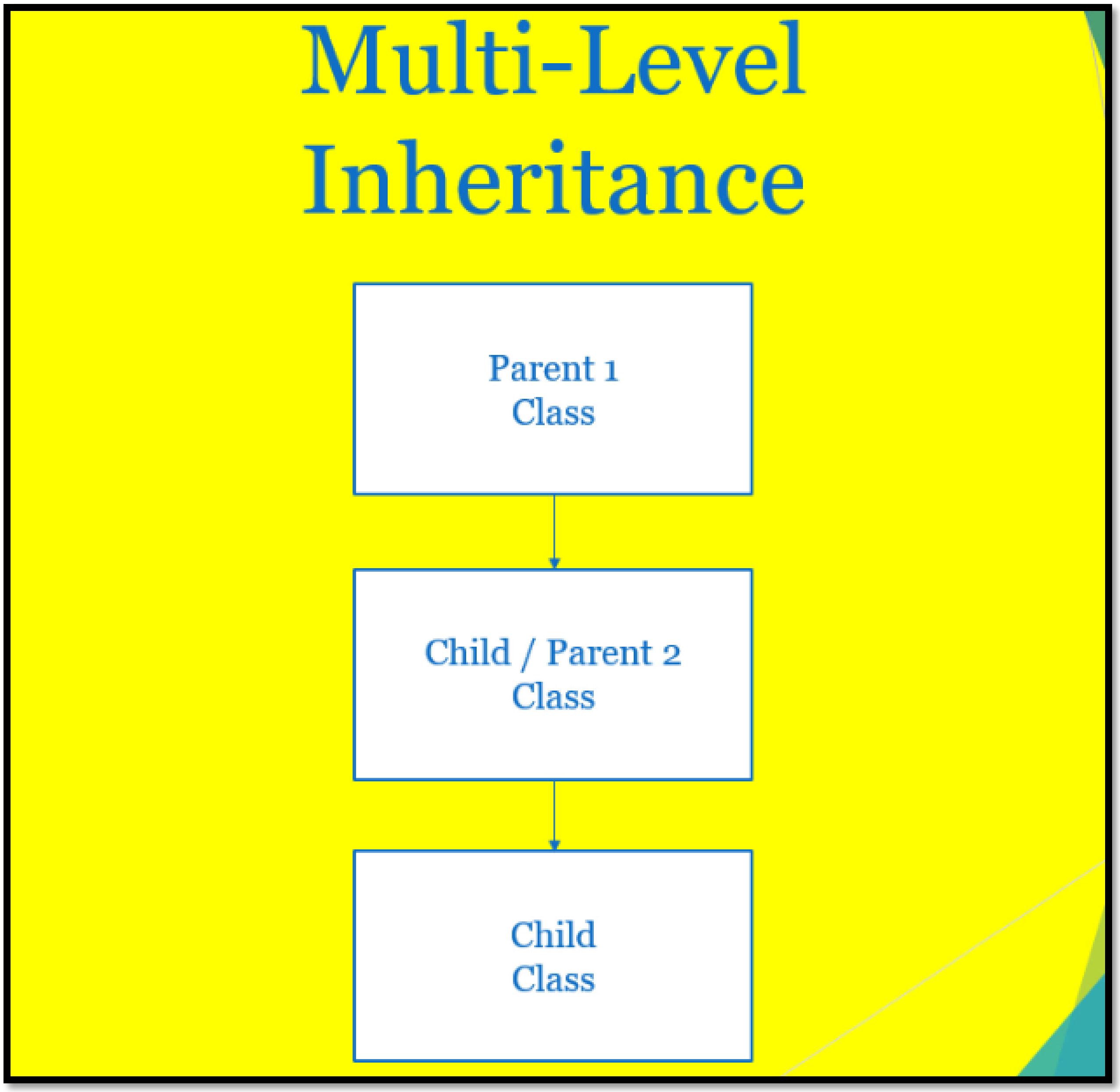
When it comes to Multiple Inheritance, the Child class can receive class variables, attributes, and methods from more than 1 Parent class. Therefore, Python allows us to inherit multiple classes. Multi-Level Inheritance refers to a derived class inheriting another derived class. In other words, it's when a child class inherits a parent class and that parent class also has a parent.

I'm going to show you the difference between Multiple Inheritance and Multi-Level Inheritance plus demo the Method Resolution Order.

Here's a diagram of Multiple Inheritance.



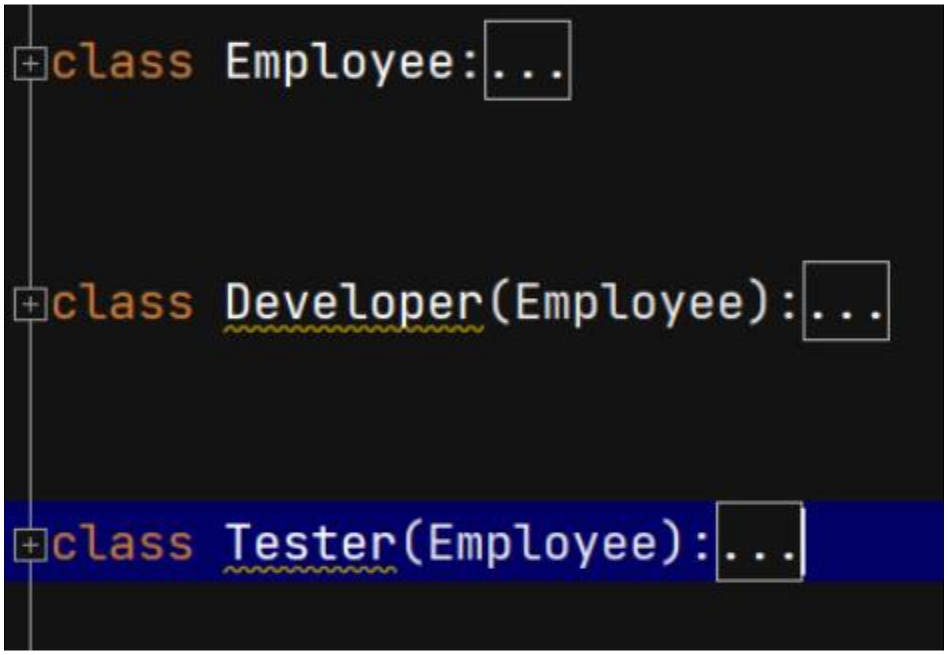
We see the Child Class at the bottom inherits from Parent 1 and Parent 2. For Multi-Level Inheritance, we see there are 2 Child Classes and 2 Parent Classes.



Multi-Level Inheritance

The class at the top is only a parent while the class in the middle is a child and a parent. However, the bottom class is only a child. In the demo, I'm going to combine Multiple Inheritance and Multi-Level Inheritance.

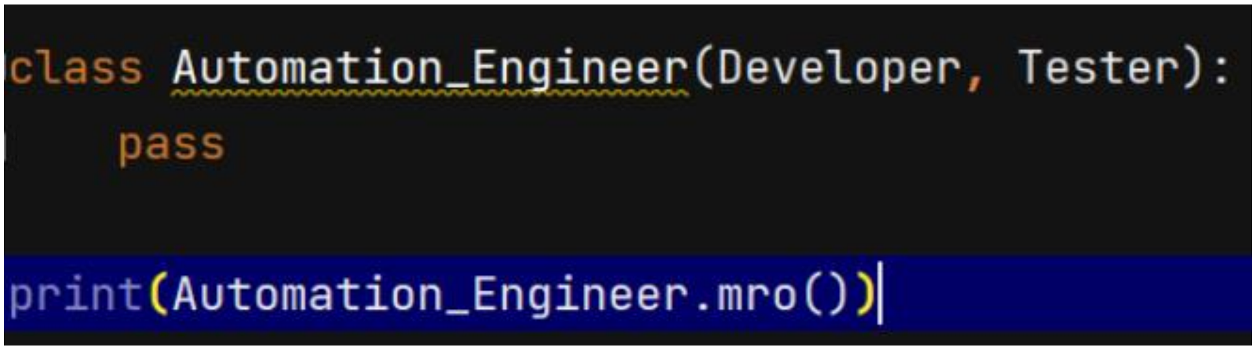
The IDE has an Employee class which is a parent. Developer and Tester are child classes that inherit the Employee class.



```
class Employee:...


class Developer(Employee):...


class Tester(Employee):...
```

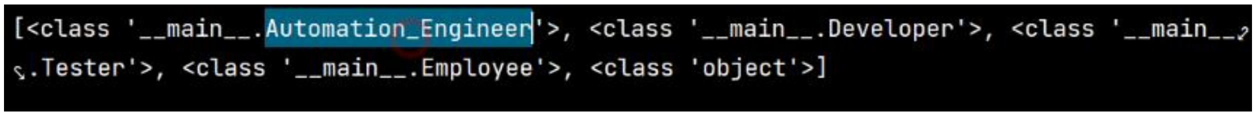
Now, I will create a class called Automation_Engineer(). The Automation_Engineer class will have 2 parent classes. It has Developer, and Tester as the parents. Write pass so there is no body.

At this point, Automation_Engineer represents Multi-Level Inheritance that inherits multiple parents. Let me show you the MRO which means Method Resolution Order. It is the order when Python looks for a method in a class hierarchy. There are 2 ways to see the MRO. The 1st way start by writing the subclass name Automation_Engineer.mro() then print() the types.



```
class Automation_Engineer(Developer, Tester):
    pass

print(Automation_Engineer.mro())
```

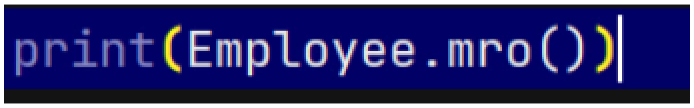
Notice, the order shows Developer then Tester. When I run, we see Automation_Engineer.



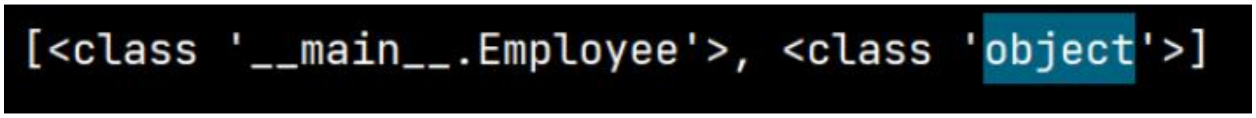
```
[<class '__main__.Automation_Engineer'>, <class '__main__.Developer'>, <class '__main__.Tester'>, <class '__main__.Employee'>, <class 'object'>]
```

Next is Developer then Tester. After Tester we see the Employee class which is the parent class. Last is object. Object is the parent class of all classes. Therefore, it will always be last.

We know the Employee class is the parent class but if I check the .mro.



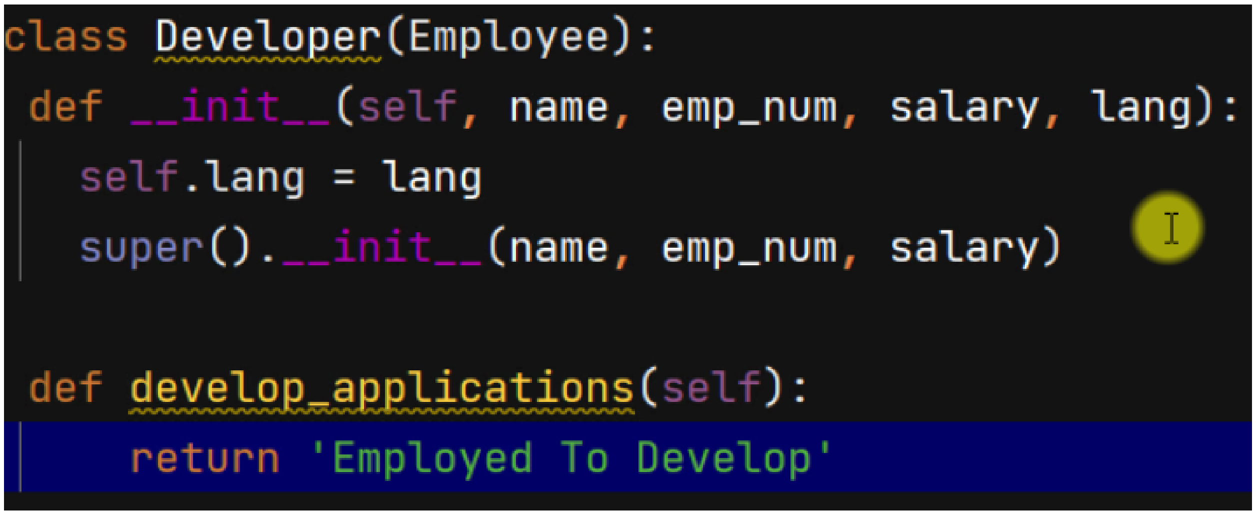
```
print(Employee.mro())
```



```
[<class '__main__.Employee'>, <class 'object'>]
```

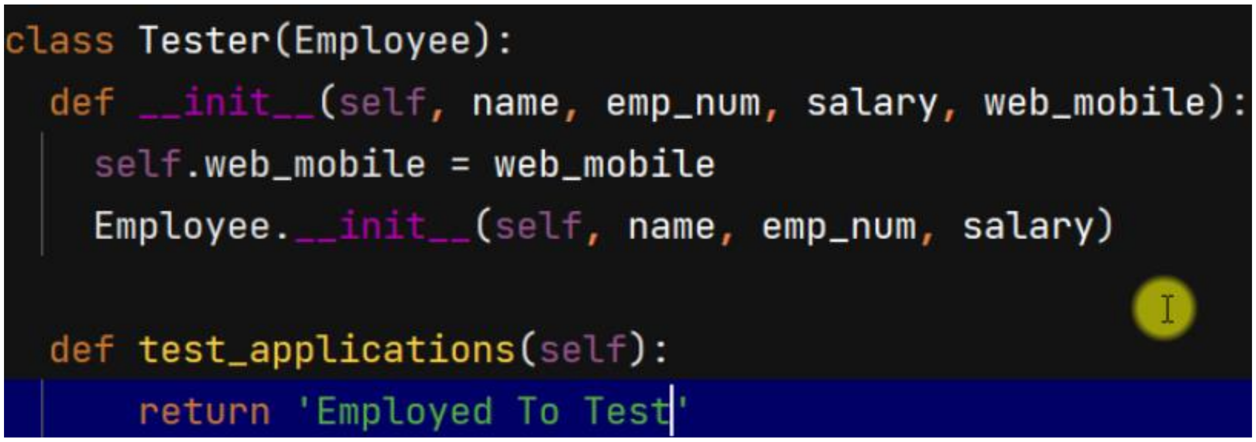
The type will show in the console object. It still shows object although we did not specify a parent because object is the superclass of all classes.

Let's add some methods to show how Automation_Engineer inherit from both parents. For Developer, we write def to define the develop_applications(self): / return 'Employed To Develop'.



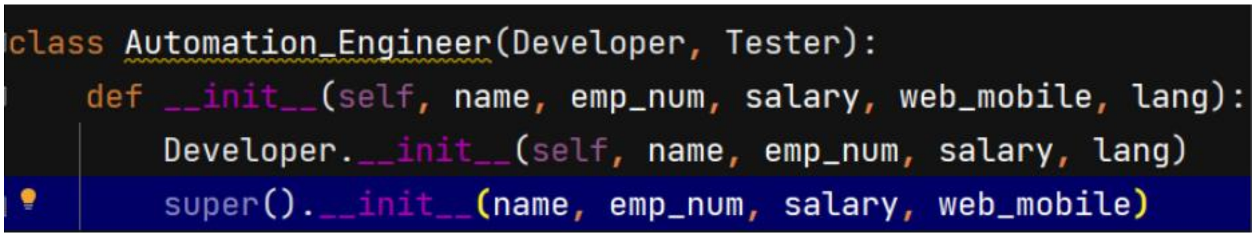
```
class Developer(Employee):
 def __init__(self, name, emp_num, salary, lang):
   self.lang = lang
   super().__init__(name, emp_num, salary)

 def develop_applications(self):
     return 'Employed To Develop'
```

For Tester, write def test_applications(self): / return 'Employed To Test'.



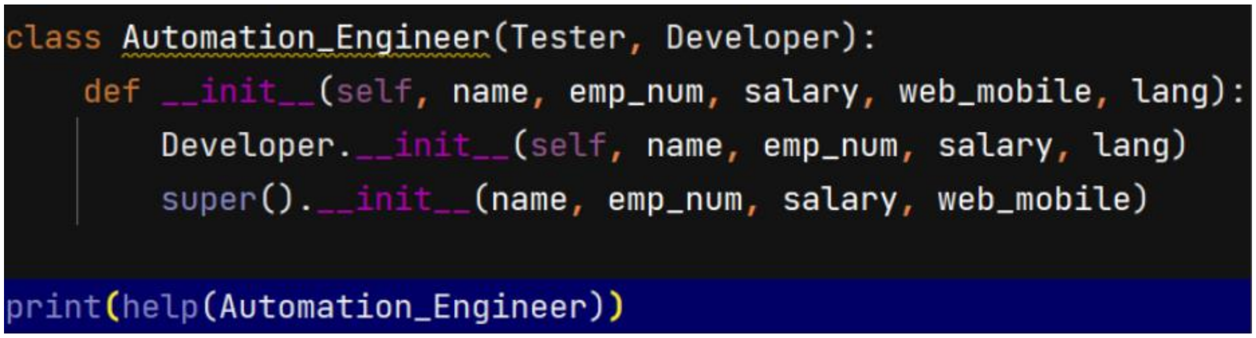
```
class Tester(Employee):
  def __init__(self, name, emp_num, salary, web_mobile):
    self.web_mobile = web_mobile
    Employee.__init__(self, name, emp_num, salary)

  def test_applications(self):
      return 'Employed To Test'
```

Now, customize the Automation Engineer class by creating our initializer and to create the initializer. We write def __init__(self, name, emp_emp_num, salary, web_mobile, lang): We have the option of writing super or the parent class name like I mentioned in the previous session to call the super class. Start with parent class which is Developer.__init__(self, name, emp_num, salary, lang) / super().__init__(name, emp_num, salary, web_mobile). When we use super, we do not need to write the keyword self.



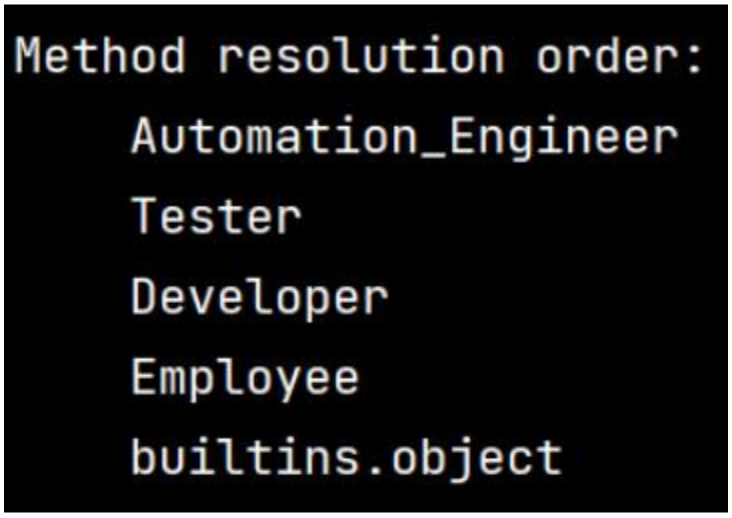
```
class Automation_Engineer(Developer, Tester):
    def __init__(self, name, emp_num, salary, web_mobile, lang):
        Developer.__init__(self, name, emp_num, salary, lang)
        super().__init__(name, emp_num, salary, web_mobile)
```

Before, I call each method from the super classes, let me show you how the 2nd way will look up the chain of classes using the Method Resolution Order. You will see how the order in which we declare each parent class makes a difference. If I change the parent class to be Tester then Developer. You're going to see how the mro will also update. We use the 2nd way by writing the help() function and pass in Automation_Engineer. print() the types then Run.



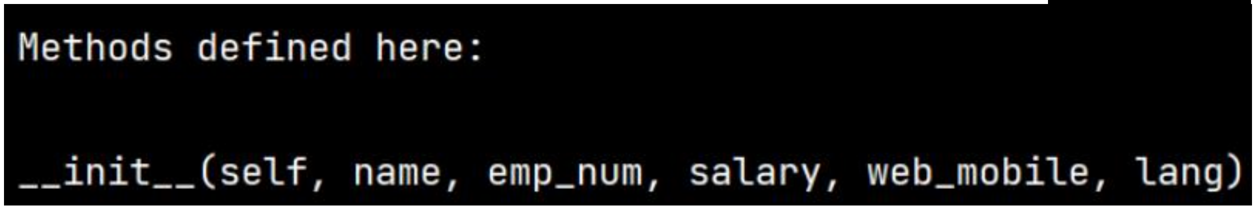
```
class Automation_Engineer(Tester, Developer):
    def __init__(self, name, emp_num, salary, web_mobile, lang):
        Developer.__init__(self, name, emp_num, salary, lang)
        super().__init__(name, emp_num, salary, web_mobile)


print(help(Automation_Engineer))
```

As expected, the Method Resolution Order shows Automation_Engineer, Tester, Developer, Employee, then object.



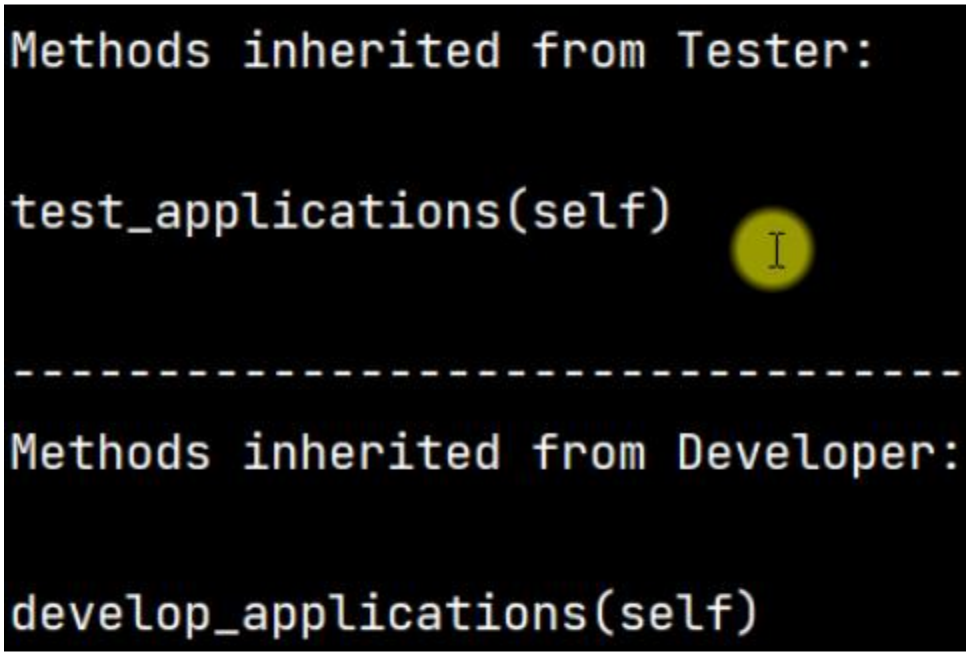
```
Method resolution order:
    Automation_Engineer
    Tester
    Developer
    Employee
    builtins.object
```

We also see the initializer method defined in Automation_Engineer.

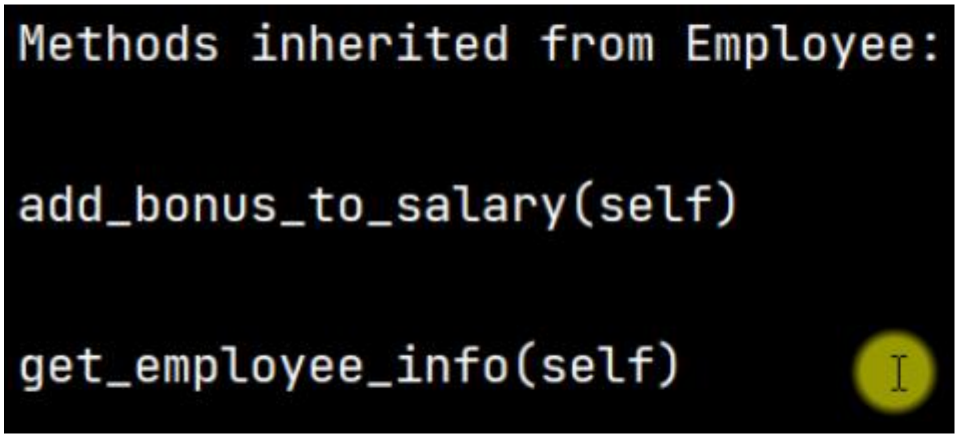


```
Methods defined here:

__init__(self, name, emp_num, salary, web_mobile, lang)
```

Plus the methods inherited from the Tester “test_applications”, Developer “develop_applications”, and both methods from the Employee class “add_bonus_to_salary & get_employee_info”.

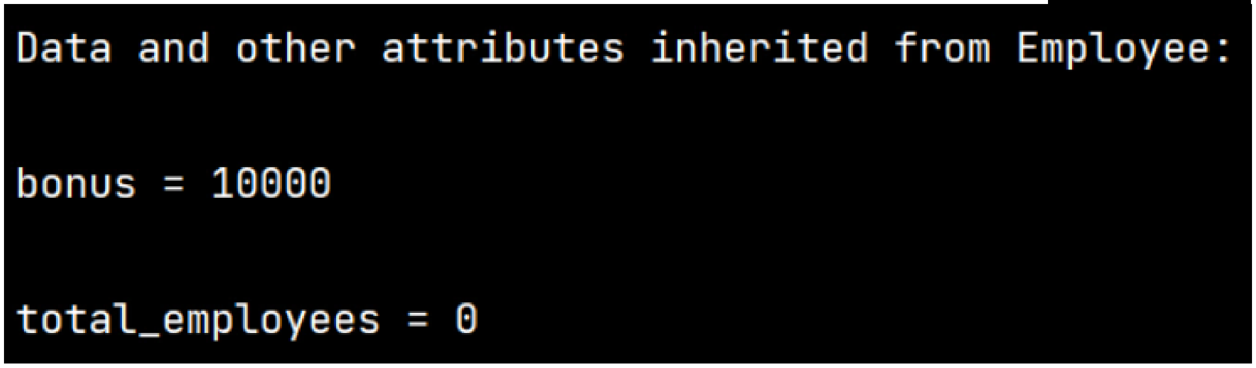


```
Methods inherited from Tester:

test_applications(self)

--------------------------------
Methods inherited from Developer:

develop_applications(self)
```



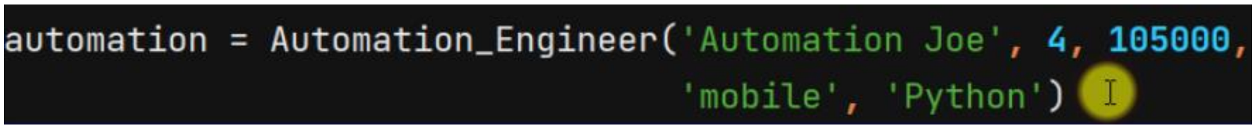
```
Methods inherited from Employee:

add_bonus_to_salary(self)

get_employee_info(self)
```

At the bottom, we data and other attributes inherited from Employee and those are bonus & total_employee class variables.



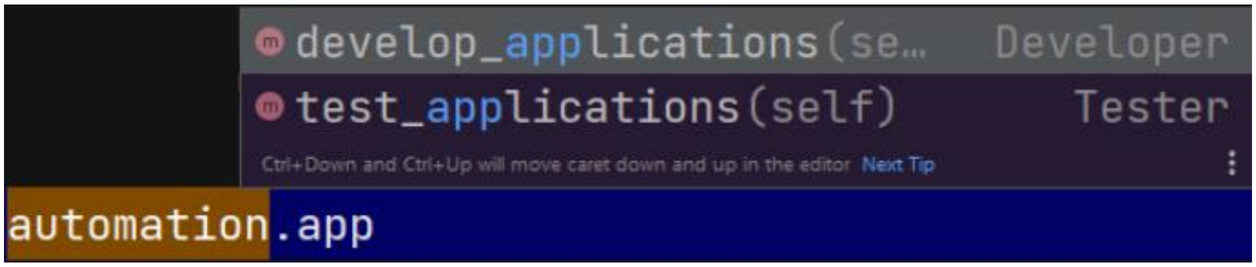
```
Data and other attributes inherited from Employee:

bonus = 10000

total_employees = 0
```

Last, let me show you how the Automation_Engineer class can call the methods from both parent classes. I'm going to start by writing automation = Automation_Engineer('Automation Joe', 4, 105000, 'mobile', 'Python')



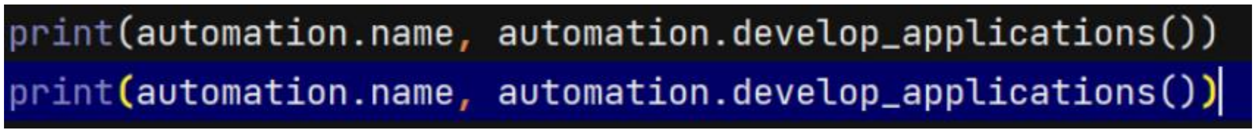
```
automation = Automation_Engineer('Automation Joe', 4, 105000,
                                 'mobile', 'Python')
```

When I use the reference automation and type automation.app We see both methods.



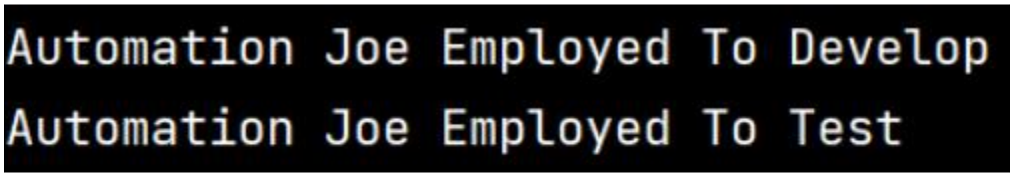
```
develop_applications(se…   Developer
test_applications(self)    Tester
Ctrl+Down and Ctrl+Up will move caret down and up in the editor  Next Tip
automation.app
```

The 1st method shows develop_applications from the Developer class and 2nd method shows test_applicaitons from the Tester class. automation.develop_applications. Let's also print Automation Joe by writing print(automation.name, automation.develop_applications) and print(automation.name, automation.test_applications).



```
print(automation.name, automation.develop_applications())
print(automation.name, automation.develop_applications())
```

The console shows Automation Joe Employed To Develop – Automation Joe Employed To Test.



```
Automation Joe Employed To Develop
Automation Joe Employed To Test
```

Rex Jones II

Contact Info

- ✓ Email Rex.Jones@Test4Success.org
- ✓ YouTube https://www.youtube.com/c/RexJonesII/videos
- ✓ Facebook https://facebook.com/JonesRexII
- ✓ Twitter https://twitter.com/RexJonesII
- ✓ GitHub https://github.com/RexJonesII/Free-Videos
- ✓ LinkedIn https://www.linkedin.com/in/rexjones34/