Super Method

This lesson covers multiple use-cases of the super method in Python and how it makes inheritance easier.

WE'LL COVER THE FOLLOWING

Introduction

As we discussed before, you can access the constructor and methods of the parent class in the child class using

```
ParentClassName.__init__(self, methodParameter)
```

and

```
ParentClassName.methodName(self, methodParameter)
```

respectively.

This method tends to get confusing when the hierarchy has multiple levels because you have to remember the exact names of the parent classes and where to use them. An easier way to do this is by using the `super()` method.

The built-in `super` function allows the child class (or subclass) to access inherited methods of the parent class (or superclass) that have been overwritten in the child class.

According to the official Python documentation:

> Super is used to return a proxy object that delegates method calls to a parent or sibling class of type. This is useful for accessing inherited methods that have been overridden in a class.

Using the `super` Function

The `super` method can be used in multiple scenarios.

Single Inheritance

We can use `super` in single inheritance to refer to the parent (or super) class or multiple classes without explicitly naming them. This is just a shortcut and helps make your code maintainable and easy to understand.

Multiple Inheritance

As discussed earlier, it is possible in Python for a class to inherit from multiple parent classes. You can use the `super` method for multiple or collaborative inheritance. This can be done only in Python because some languages do not support inheritance from multiple classes.

> **Note:** `super()` is not limited to use inside methods. You can specify the appropriate references by specifying the arguments of the `super()` method.

In Python 3 and above, the syntax for `super` is:

```
super().methodName(args)
```

Now consider the following code snippet where the `Student` class inherits from (or "is a") `Person`.

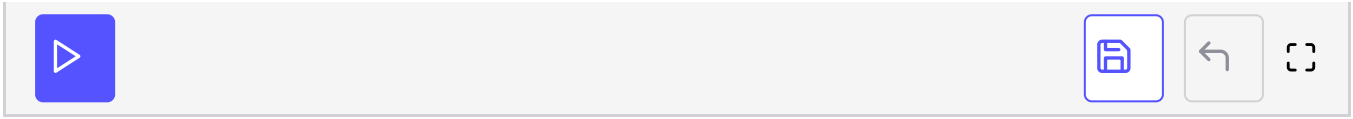
```
class Person(object): # Super class
  def __init__(self, name):
    self.name = name
  def greet(self):
    print ("Hi, I'm " + self.name + ".") # Super class does something

class Student(Person): # Subclass inheriting from the super class
  def __init__(self, name, degree):
    self.name = name
    self.degree = degree
```

```
    Person.__init__(self, name) # calls constructor of super class
  def greet(self):
    Person.greet(self) # calls method of super class

    print ("I am a " + self.degree + " student.")

student = Student("Ali", "PhD") # Create an object of the subclass
student.greet()
```

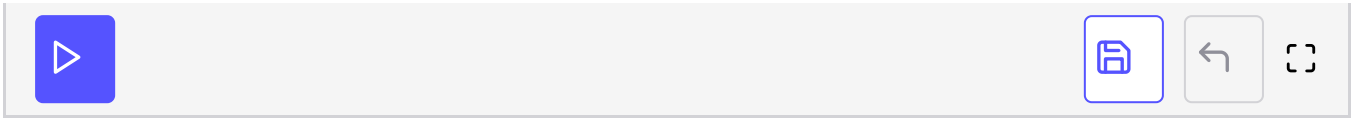


Notice that when you create a `Student` object, it can access the `Parent` class's `greet()` method and write its own too. Now we will modify the above code to include the `super` method.

```
class Person(object): # Super class
  def __init__(self, name):
    self.name = name
  def greet(self):
    print ("Hi, I'm " + self.name + ".") # Super class does something

class Student(Person): # Subclass inheriting from the super class
  def __init__(self, name, degree):
    self.name = name
    self.degree = degree
    super().__init__(name) # calls constructor of super class
  def greet(self):
    super().greet() # calls method of super class
    print ("I am a " + self.degree + " student.")

student = Student("Ali", "PhD") # Create an object of the subclass
student.greet()
```



Notice in the above code playground that when we called the parent class's methods `super().__init__(name)` and `super().greet()`, we didn't explicitly add `self` as a parameter to these methods.

Let's solve some exercises on this concept.