

CS: Objects in Python

Inheritance

Parent & Child Classes

Learning Objectives - Parents & Children

- Define the terms inheritance, parent class, and child class
- Explain the relationship between the parent class and child class
- Explain the role of the `super()` keyword
- Create a child class from a given parent class
- Use `isinstance` to determine an object's parent class
- Use `issubclass` to determine a class's parent class

What is Inheritance?

Defining Inheritance

Imagine you want to create two Python classes, `Person` and `Superhero`. These respective classes might look something like this:

Person Class:

```
name
age
occupation

say_hello()
say_age()
```

Superhero Class:

```
name
age
occupation
secret_identity
nemesis

say_hello()
say_age()
reveal_secret_identity()
```

No_Inheritance

There are some similarities between the `Person` class and the `Superhero` class. If the `Person` class already exists, it would be helpful to “borrow” from the `Person` class so you only have to create the new attributes and methods for the `Superhero` class. This situation describes inheritance — one class copies the attributes and methods from another class.

Inheritance Syntax

In the IDE on the left, the `Person` class is already defined. To create the `Superhero` class that inherits from the `Person` class, add the following code at the end of the program. Notice how `Superhero` takes `Person` as a parameter. This is how you indicate to Python that the `Superhero` class inherits from the `Person` class. You can also say that the `Person` class is the parent and the `Superhero` class is the child.

```
class Superhero(Person):  
    pass  
  
hero = Superhero("Jessica Jones", 29, "private investigator")  
print(hero.name)  
print(hero.occupation)
```

The Superhero class can access the attributes of the Person class because Superheor inherits from Person.

challenge

Try this variation:

- Print the age attribute for the hero object.
- Call the say_hello method with the hero object.

▼ **Solution**

```
print(hero.age)  
hero.say_hello()
```

Visualizing Inheritance

Python has a function called `help` which can aid you in understanding what is being inherited. Add the following line of code after instantiating hero as an instance of the Superhero class.

```
hero = Superhero("Jessica Jones", 29, "private investigator")  
print(help(Superhero))  
print(hero.name)  
print(hero.occupation)
```

You should see a visual representation of the Superhero class. The first section is Method resolution order. This shows the steps Python takes with inheritance. Python first starts with the Superhero class. So when you say

`print(hero.name)`, Python will look to the `Superhero` class for the attribute `name`. This does not exist. So Python moves to the next step, which is the `Person` class. Since the `Person` class has a `name` attribute, Python can stop here. Notice too that the `say_name` and `say_age` methods are also inherited.

challenge

Try this variation:

- Print the result of using the `help` function on the `Person` class -
`print(help(Person))`
 - ▼ **Why does the `Person` class inherit from `builtins.object` class?**
In Python, all classes inherit from the `objects` class. It is not necessary to write `class Person(object):` when declaring a class. Python will perform the inheritance automatically.

Super

The `super()` Keyword

Another way to call a method from the parent class (sometimes called the super class), is to use the `super()` keyword. Alter the end of the program to look like the following code.

```
class Superhero(Person):  
    def say_hello(self):  
        super().say_hello()  
  
hero = Superhero("Wonder Woman", 27, "intelligence officer")  
hero.say_hello()
```

The `Superhero` class has a method called `say_hello`. The `super()` keyword tells Python to go to the parent class; the `.say_hello()` tells Python to call this method. So `super().say_hello()` is calling `say_hello` from the `Person` class.

challenge

Try this variation:

- Declare the function `say_age` for the `Superhero` class using inheritance.

▼ Solution

```
class Superhero(Person):
    def say_hello(self):
        super().say_hello()

    def say_age(self):
        super().say_age()

hero = Superhero("Wonder Woman", 27, "intelligence officer")
hero.say_hello()
hero.say_age()
```

`super()` and `__init__`

If the `super()` keyword is used to call methods from the parent class, how come `super()` was not used with the `__init__` method? In fact, the `__init__` was never called in the `Superhero` class. A child class will automatically inherit the `__init__` method if it is not defined. `__init__` is called when an object is instantiated, and `super()` does not need to be used.

However, it is possible to create an `__init__` method for the `Superhero` class using `super()`. Later on, you will see the benefits of doing this. Change the code for `Superhero` to look like the code below. You should still be able to print the attributes just as before.

```
class Superhero(Person):
    def __init__(self, name, age, occupation):
        super().__init__(name, age, occupation)

hero = Superhero("Batman", 32, "CEO")
print(hero.name)
```

▼ The lack of self in super().__init__

When calling `super().__init__` the keyword `self` was not used. That is because you were not defining the `init` method of the parent class, you were calling it. Just like any other method in Python, `self` is the first parameter when you declare a method, but it is not used when call the method. This is the first time that we are explicitly called the `init` method. Usually this happens automatically when an object is instantiated.

challenge

Try this variation:

- Using the `super()` keyword, create the method `say_two_things` for the `Superhero` class. This method should print the name and age of the hero object.

▼ Solution

```
class Superhero(Person):
    def say_two_things(self):
        super().say_hello()
        super().say_age()

hero = Superhero("Rorschach", 34, "conspiracy theorist")
hero.say_two_things()
```

Inheritance Hierarchy

Inheritance Hierarchy

You have seen how the `Superhero` class becomes the child of the `Person` class through inheritance. The relationship between these two classes is called inheritance (or class) hierarchy. Python has some built-in functions to help you determine the hierarchy of classes.

The first function is `isinstance`. This function takes an object and a class name. It returns `True` if the object is an instance of the class. Look at the code on the left. `ClassA` is the parent of `ClassB`, and `ClassC` is the parent of `ClassD`. The `isinstance` function can be used to test these relationships. Add the following code to the program.

```
print(isinstance(object_b, ClassA))
print(isinstance(object_d, ClassC))
```

challenge

Try this variation:

- `print(isinstance(object_b, ClassC))`
- `print(isinstance(object_d, ClassA))`
- `print(isinstance(object_a, ClassA))`

Is a Subclass?

Another function that can be used to determine inheritance hierarchy is `issubclass`. This function takes a class and a class name. It returns `true` if the class is a subclass (or child) of the class name. This is very similar to `isinstance`, but the difference is important. `issubclass` checks to see if a class (not an object) is the child of another class. Add the following code to the program.


```
print(issubclass(ClassB, ClassA))
print(issubclass(ClassD, ClassC))
```

challenge

Try this variation:

- `print(issubclass(ClassD, ClassA))`
- `print(issubclass(ClassA, ClassB))`
- `print(issubclass(object_b, ClassA))`

▼ What went wrong?

The last `print` statement produced an error because `issubclass` requires two classes as parameters. `object_b` is an instance of a class, not a class itself.