Inherit From Other Classes in Python

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called child classes, and the classes that child classes are derived from are called parent classes.

Child classes can override or extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent's attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn't perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your mother. It's an attribute you were born with. Let's say you decide to color your hair purple. Assuming your mother doesn't have purple hair, you've just **overridden** the hair color attribute that you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you'll also speak English. Now imagine you decide to learn a second language, like German. In this case you've **extended** your attributes because you've added an attribute that your parents don't have.

Dog Park Example

Pretend for a moment that you're at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The Dog class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the Dog class in the editor window by adding a `.breed` attribute:

In [1]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

```
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

The instance methods defined earlier are omitted here because they aren't important for this discussion.

Now you can model the dog park by instantiating a bunch of different dogs.

In [2]:
```
miles = Dog("Miles", 4, "Jack Russell Terrier")
buddy = Dog("Buddy", 9, "Dachshund")
jack = Dog("Jack", 3, "Bulldog")
jim = Dog("Jim", 5, "Bulldog")
```

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like woof, but dachshunds have a higher-pitched bark that sounds more like yap.

Using just the Dog class, you must supply a string for the sound argument of .speak() every time you call it on a Dog instance:

In [3]:
```
buddy.speak("Yap")
```

Out[3]:
```
'Buddy says Yap'
```

In [4]:
```
jim.speak("Woof")
```

Out[4]:
```
'Jim says Woof'
```

In [5]:
```
jack.speak("Woof")
```

Out[5]:
```
'Jack says Woof'
```

Passing a string to every call to `.speak()` is repetitive and inconvenient. Moreover, the string representing the sound that each Dog instance makes should be determined by its `.breed` attribute, but here you have to manually pass the correct string to `.speak()` every time it's called.

You can simplify the experience of working with the Dog class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for `.speak()`.

Parent Classes vs Child Classes

Let's create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here's the full definition of the Dog class:

In [6]:
```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. Add the following to the dog.py file to create three new child classes of the Dog class:

In [7]:
```
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

With the child classes defined, you can now instantiate some dogs of specific breeds:

In [8]:
```
miles = JackRussellTerrier("Miles", 4)
buddy = Dachshund("Buddy", 9)
jack = Bulldog("Jack", 3)
jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

In [9]:
```
miles.species
```

Out[9]:
```
'Canis familiaris'
```

In [10]:
```
buddy.name
```

Out[10]:
```
'Buddy'
```

In [11]:
```
print(jack)
```

```
Jack is 3 years old
```

In [12]:
```
jim.speak("Woof")
```

Out[12]:
```
'Jim says Woof'
```

 which class a given object belongs to, you can use the built-in `type()`:

In [13]:
```
type(miles)
```

Out[13]:
```
__main__.JackRussellTerrier
```

What if you want to determine if miles is also an instance of the Dog class? You can do this with the built-in `isinstance()`:

In [14]:
```
isinstance(miles, Dog)
```

Out[14]:
```
True
```

Notice that `isinstance()` takes two arguments, an object and a class. In the example above, `isinstance()` checks if miles is an instance of the Dog class and returns `True``.

The `miles`, `buddy`, `jack`, and `jim` objects are all `Dog` instances, but `miles` is not a `Bulldog` instance, and `jack` is not a `Dachshund` instance:

In [15]:
```
isinstance(miles, Bulldog)
```

Out[15]:
```
False
```

In [16]:
```
isinstance(jack, Dachshund)
```

Out[16]:
```
False
```

In [17]:
```
isinstance(jack, Bulldog)
```

Out[17]:
```
True
```

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, let's give each breed its own sound.

Extend the Functionality of a Parent Class

Since different breeds of dogs have slightly different barks, you want to provide a default value for the sound argument of their respective `.speak()` methods. To do this, you need to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, you define a method with the same name on the child class. Here's what that looks like for the `JackRussellTerrier` class:

```
Russellterrier(Dog):
    def speak(self, sound="Arf"):
```

```
        return f"{self.name} saysssss {sound}"
```

Now `.speak()` is defined on the `JackRussellTerrier` class with the default argument for sound set to "Arf".

You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to sound:

In [19]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[19]:
```
'Miles saysssss Arf'
```

Sometimes dogs make different barks, so if Miles gets angry and growls, you can still call `.speak()` with a different sound:

In [20]:
```
miles.speak("Grrr")
```

Out[20]:
```
'Miles saysssss Grrr'
```

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, in the editor window, change the string returned by `.speak()` in the `Dog` class:

In [21]:
```
class Dog:

    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    # Change the string returned by .speak()
    def speak(self, sound):
        return f"{self.name} barks: {sound}"
```

Now, when you create a new `Bulldog` instance named `jim`, `jim.speak()` returns the new string:

In [22]:
```
class Bulldog(Dog):
    pass
```

In [23]:
```
jim = Bulldog("Jim", 5)
'Woof")
```

Out[23]:
```
'Jim barks: Woof'
```

However, calling `.speak()` on a `JackRussellTerrier` instance won't show the new style of output:

In [24]:
```
class JackRussellTerrier(Dog):

    def speak(self, sound="Arf"):
        return f"{self.name} saysssss {sound}"
```

In [25]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[25]:
```
'Miles saysssss Arf'
```

Sometimes it makes sense to completely override a method from a parent class. But in this instance, we don't want the `JackRussellTerrier` class to lose any changes that might be made to the formatting of the output string of `Dog.speak()`.

To do this, you still need to define a `.speak()` method on the child `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the Dog class's `.speak()` inside of the child class's `.speak()` using the same arguments that you passed to `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using `super()`:

In [26]:
```
class JackRussellTerrier(Dog):

    def speak(self, sound="Arf"):
        return super().speak(sound)
```

When you call `super().speak(sound)` inside `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable sound.

So you can test it:

In [27]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[27]:
```
'Miles barks: Arf'
```

Now when you call `miles.speak()`, you'll see output reflecting the new formatting in the `Dog` class.

> **Note:** In the above examples, the class hierarchy is very straightforward. The `JackRussellTerrier` class has a single parent class, `Dog`. In real-world examples, the class hierarchy can get quite complicated.

> `super()` does much more than just search the parent class for a method or an attribute. It traverses the entire class hierarchy for a matching method or attribute. If you aren't careful, `super()` can have surprising results.

Your turn!

Check your understanding, go to the exercise!
