

PYTHON CLASSES AND OBJECTS

PYTHON – INHERITANCE



Introduction

Object-oriented programming creates reusable patterns of code to curtail redundancy in development projects. One way that object-oriented programming achieves recyclable code is through inheritance, when one subclass can leverage code from another base class.

In this session we will go through some of the major aspects of inheritance in Python, including how parent classes and child classes work, how to override methods and attributes, how to use the `super()` function, and how to make use of multiple inheritance.

What is Inheritance?

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents.

Classes called **child classes** or **subclasses** inherit methods and variables from **parent classes** or **base classes**.

We can think of a **parent class** called `Parent` that has class variables for `last_name`, `height`, and `eye_color` that the child class `Child` will inherit from the Parent.

Because the Child subclass is inheriting from the Parent base class, the Child class can reuse the code of Parent, allowing the programmer to use fewer lines of code and decrease redundancy.

Parent Classes

Parent or base classes create a pattern out of which child or subclasses can be based on. Parent classes allow us to create child classes through inheritance without having to write the same code over again each time. Any class can be made into a parent class, so they are each fully functional classes in their own right, rather than just templates.

Let's say we have a general `Bank_account` parent class that has `Personal_account` and `Business_account` child classes. Many of the methods between personal and business accounts will be similar, such as methods to withdraw and deposit money, so those can belong to the parent class of `Bank_account`. The `Business_account` subclass would have methods specific to it, including perhaps a way to collect business records and forms, as well as an `employee_identification_number` variable.

Similarly, an `Animal` class may have `eating()` and `sleeping()` methods, and a `Snake` subclass may include its own specific `hissing()` and `slithering()` methods.

Let's create a `Fish` parent class that we will later use to construct types of fish as its subclasses. Each of these fish will have first names and last names in addition to characteristics.

We'll create a new file called `fish.py` and start with the `__init__()` constructor method, which we'll populate with `first_name` and `last_name` class variables for each `Fish` object or subclass.

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name
```

We have initialized our `last_name` variable with the string `"Fish"` because we know that most fish will have this as their last name.

Let's also add some other methods:

```
class Fish:
    def __init__(self, first_name, last_name="Fish"):
        self.first_name = first_name
        self.last_name = last_name

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

We have added the methods `swim()` and `swim_backwards()` to the `Fish` class, so that every subclass will also be able to make use of these methods.

Since most of the fish we'll be creating are considered to be **bony fish** (as in they have a skeleton made out of bone) rather than **cartilaginous fish** (as in they have a skeleton made out of cartilage), we can add a few more attributes to the `__init__()` method:

```
class Fish:
    def __init__(self, first_name, last_name="Fish",
                  skeleton="bone", eyelids=False):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim(self):
        print("The fish is swimming.")

    def swim_backwards(self):
        print("The fish can swim backwards.")
```

Building a parent class follows the same methodology as building any other class, except we are thinking about what methods the child classes will be able to make use of once we create those.

Child Classes

Child or subclasses are classes that will inherit from the parent class. That means that each child class will be able to make use of the methods and variables of the parent class.

For example, a `Goldfish` child class that subclasses the `Fish` class will be able to make use of the `swim()` method declared in `Fish` without needing to declare it.

We can think of each child class as being a class of the parent class. That is, if we have a child class called `Rhombus` and a parent class called `Parallelogram`, we can say that a `Rhombus` is a `Parallelogram`, just as a `Goldfish` is a `Fish`.

The first line of a child class looks a little different than non-child classes as you must pass the parent class into the child class as a parameter:

```
class Trout(Fish):
```

The `Trout` class is a child of the `Fish` class. We know this because of the inclusion of the word `Fish` in parentheses.

With child classes, we can choose to add more methods, override existing parent methods, or accept the default parent methods with the `pass` keyword, which we'll do in this case:

```
...  
class Trout(Fish):  
    pass
```

We can now create a `Trout` object without having to define any additional methods.

```
...  
class Trout(Fish):  
    pass  
  
terry = Trout("Terry")  
print(terry.first_name + " " + terry.last_name)  
print(terry.skeleton)  
print(terry.eyelids)  
terry.swim()  
terry.swim_backwards()
```

We have created a `Trout` object `terry` that makes use of each of the methods of the `Fish` class even though we did not define those methods in the `Trout` child class. We only needed to pass the value of `"Terry"` to the `first_name` variable because all of the other variables were initialized.

When we run the program, we'll receive the following output:

Output

```
Terry Fish
```

```
bone
```

```
False
```

```
The fish is swimming.
```

```
The fish can swim backwards.
```

Next, let's create another child class that includes its own method. We'll call this class `Clownfish`, and its special method will permit it to live with sea anemone:

```
...  
class Clownfish(Fish):  
  
    def live_with_anemone(self):  
        print("The clownfish is coexisting with sea anemone.")
```

Next, let's create a `Clownfish` object to see how this works:

```
...  
casey = Clownfish("Casey")  
print(casey.first_name + " " + casey.last_name)  
casey.swim()  
casey.live_with_anemone()
```

When we run the program, we'll receive the following output:

```
Output  
Casey Fish  
The fish is swimming.  
The clownfish is coexisting with sea anemone.
```

The output shows that the `Clownfish` object `casey` is able to use the `Fish` methods `__init__()` and `swim()` as well as its child class method of `live_with_anemone()`.

If we try to use the `live_with_anemone()` method in a `Trout` object, we'll receive an error:

Output

```
terry.live_with_anemone()  
AttributeError: 'Trout' object has no attribute 'live_with_anemone'
```

This is because the method `live_with_anemone()` belongs only to the `Clownfish` child class, and not the `Fish` parent class.

Child classes inherit the methods of the parent class it belongs to, so each child class can make use of those methods within programs.

Overriding parent methods

So far, we have looked at the child class `Trout` that made use of the `pass` keyword to inherit all of the parent class `Fish` behaviors, and another child class `Clownfish` that inherited all of the parent class behaviors and also created its own unique method that is specific to the child class. Sometimes, however, we will want to make use of some of the parent class behaviors but not all of them. When we change parent **class methods** we override them.

When constructing parent and child classes, it is important to keep program design in mind so that overriding does not produce unnecessary or redundant code.

We'll create a `Shark` child class of the `Fish` parent class. Because we created the `Fish` class with the idea that we would be creating primarily bony fish, we'll have to make adjustments for the `Shark` class that is instead a cartilaginous fish. In terms of program design, if we had more than one non-bony fish, we would most likely want to make separate classes for each of these two types of fish.

Sharks, unlike bony fish, have skeletons made of cartilage instead of bone. They also have eyelids and are unable to swim backwards. Sharks can, however, maneuver themselves backwards by sinking.

In light of this, we'll be overriding the `__init__()` constructor method and the `swim_backwards()` method. We don't need to modify the `swim()` method since sharks are fish that can swim.

Let's review this child class:



```
...
class Shark(Fish):
    def __init__(self, first_name, last_name="Shark",
                  skeleton="cartilage", eyelids=True):
        self.first_name = first_name
        self.last_name = last_name
        self.skeleton = skeleton
        self.eyelids = eyelids

    def swim_backwards(self):
        print("The shark cannot swim backwards, but can sink backwards.")
```

We have overridden the initialized parameters in the `__init__()` method, so that the `last_name` variable is now set equal to the string `"Shark"`, the `skeleton` variable is now set equal to the string `"cartilage"`, and the `eyelids` variable is now set to the Boolean value `True`. Each instance of the class can also override these parameters.

The method `swim_backwards()` now prints a different string than the one in the `Fish` parent class because sharks are not able to swim backwards in the way that bony fish can.

We can now create an instance of the `Shark` child class, which will still make use of the `swim()` method of the `Fish` parent class:

```
...
sammy = Shark("Sammy")
print(sammy.first_name + " " + sammy.last_name)
sammy.swim()
sammy.swim_backwards()
print(sammy.eyelids)
print(sammy.skeleton)
```

When we run this code, we'll receive the following output:

Output

```
Sammy Shark  
The fish is swimming.  
The shark cannot swim backwards, but can sink backwards.  
True  
cartilage
```

The `Shark` child class successfully overrode the `__init__()` and `swim_backwards()` methods of the `Fish` parent class, while also inheriting the `swim()` method of the parent class.

When there will be a limited number of child classes that are more unique than others, overriding parent class methods can prove to be useful.

The `super()` Function

With the `super()` function, you can gain access to inherited methods that have been overwritten in a class object.

When we use the `super()` function, we are calling a parent method into a child method to make use of it. For example, we may want to override one aspect of the parent method with certain functionality, but then call the rest of the original parent method to finish the method.

The `super()` function is most commonly used within the `__init__()` method because that is where you will most likely need to add some uniqueness to the child class and then complete initialization from the parent.

To see how this works, let's modify our `Trout` child class. Since trout are typically freshwater fish, let's add a `water` variable to the `__init__()` method and set it equal to the string `"freshwater"`, but then maintain the rest of the parent class's variables and parameters:

```
...  
class Trout(Fish):  
    def __init__(self, water = "freshwater"):  
        self.water = water  
        super().__init__(self)  
...
```

We have overridden the `__init__()` method in the `Trout` child class, providing a different implementation of the `__init__()` that is already defined by its parent class `Fish`. Within the `__init__()` method of our `Trout` class we have explicitly invoked the `__init__()` method of the `Fish` class.

Because we have overridden the method, we no longer need to pass `first_name` in as a parameter to `Trout`, and if we did pass in a parameter, we would reset `freshwater` instead. We will therefore initialize the `first_name` by calling the variable in our object instance.

Now we can invoke the initialized variables of the parent class and also make use of the unique child variable. Let's use this in an instance of `Trout`:

```
...
terry = Trout()

# Initialize first name
terry.first_name = "Terry"

# Use parent __init__() through super()
print(terry.first_name + " " + terry.last_name)
print(terry.eyelids)

# Use child __init__() override
print(terry.water)

# Use parent swim() method
terry.swim()
```

Output

```
Terry Fish
False
freshwater
The fish is swimming.
```


The output shows that the object `terry` of the `Trout` child class is able to make use of both the child-specific `__init__()` variable `water` while also being able to call the `Fish` parent `__init__()` variables of `first_name`, `last_name`, and `eyelids`.

The built-in Python function `super()` allows us to utilize parent class methods even when overriding certain aspects of those methods in our child classes.

**See you at
the next
session!**

