Subclasses

by Sophia

WHAT'S COVERED

In this lesson, we'll explore how subclasses are used and how they can extend functionality. Specifically, this lesson covers:

1. Customizing Subclasses

So, what we've learned so far is that the subclass accepts all the different parameters that the base class accepts and assigns them to attributes similar to the base class.

In a previous example, we created and defined a base class called `Member` and then created two subclasses called `Admin` and `User`. The subclasses inherited all the attributes and methods from the base class. However, the `Admin` and `User` were just subclasses without any unique characteristics. If you recall, we placed a `pass` statement in each of those subclasses to make sure that when executing the program, we did not get any errors since the subclasses were empty. Let's look at our prior example again.

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

#Subclass for us to use for administrators
class Admin(Member):
  pass

#Subclass for us to use for normal users
class User(Member):
  pass

TestMember = Member('Sophia','Python')
print(TestMember.first_name)
print(TestMember.last_name)
print(TestMember.expiry_date)

TestAdmin = Admin('root','admin')
print(TestAdmin.first_name)
print(TestAdmin.last_name)
print(TestAdmin.expiry_date)

TestUser = User('Artic','Smith')
print(TestUser.first_name)
print(TestUser.last_name)
print(TestUser.expiry_date)
```

And here was the output again.

```
Sophia
Python
2023-02-16
root
admin
2023-02-16
Arctic
Smith
2023-02-16
```

TRY IT

Directions: Go ahead and enter this code in the IDE and make sure you get the same output before moving on.

There is nothing specifically defined in either subclass, just the `pass` statement. To truly make the subclasses useful, we want them to have some differences.

One of the most common things we can do is to make an attribute that has a default value different from what's in the base class. For example, in the `Member` class, we set the `expiry_days` to 365. However, if we want a rule in place that we don't want the Admin accounts to expire until 100 years from now, we could change the `expiry_days` value to be 365 * 100.

EXAMPLE

```
#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100
```

This would result in the expiry days being 365 * 100 days longer. Whatever value that we pass into the `expiry_days` variable will override the value that is in the `Member` base class, which is why we see the date change in the output below.

```
root
admin
2122-01-23
```

TRY IT

Directions: Try adding the updated `expiry_days` variable in the subclass.

Sometimes a subclass has attributes that the base class does not. In that case, we may want to pass an argument to the subclass that doesn't exist in the base class. Doing so is a little more complicated; however, it is a common technique so we should be aware of the steps to do that.

As a starting point, our subclass will need its own `__init__` method that contains all the parameters that are in the base class's `__init__` method. On top of that, it will also need all of the extra parameters that we want to have passed and set. If our `Admin` subclass has a secret code that we want to set, we will have to pass in the first and last name too, so our `__init__` method line from the `Admin` subclass would look like the following.

EXAMPLE

```
  def __init__(self, first, last, secret):
```

See, it looks identical to the base class's `__init__` method except for the last parameter of `secret`.

Next, we want any parameters that belong to the base class `Member` to be passed. It uses a slightly different format than what we've seen before.

EXAMPLE

```
  def __init__(self, first, last, secret):
    super().__init__(first,last)
```

We are passing in the parameters that exist for the base class that we want to keep (namely `first` and `last`). The information that we're providing in the parameters should be everything that's already in the base class parameters. The unique part is the `super()` function call. The **super() function** allows a subclass to access the base class attributes and methods. In our example, the `super()` function calls the base class; in our case, we're calling the `__init__` of the `Member` base class from the `Admin` subclass. We are passing in the first name and last name. We still have the secret parameter in the subclass `__init__` method that hasn't been set yet, so we'll need to do that in the `Admin` subclass.

EXAMPLE

```
#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100

  def __init__(self, first, last, secret):
    super().__init__(first,last)
    self.secret_code = secret
```

Let’s test this altogether now with an updated `Admin` subclass and updated `Admin` instance using a new third argument for the secret parameter. We left the `User` subclass empty. Note: we placed a `print()` function with a string of lines to separate each subclass for easier viewing.

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100

  def __init__(self, first, last, secret):
    super().__init__(first,last)
    self.secret_code = secret

#Subclass for us to use for normal users
class User(Member):
  pass

TestAdmin = Admin('root','admin','ABRACADABRA')
print(TestAdmin.first_name)
print(TestAdmin.last_name)
print(TestAdmin.secret_code)
print(TestAdmin.expiry_date)
print("--------")

TestUser = User('Artic','Smith')
print(TestUser.first_name)
print(TestUser.last_name)
print(TestUser.expiry_date)
```

TRY IT

Directions: Enter the updated `User` subclass instance and see if you get the same output as below. Note that we are not accessing the base class attributes specifically this time as there is not an instance of `Member`. We are only creating instances of the subclasses.

```
root
admin
ABRACADABRA
2122-01-23
--------
Arctic
Smith
2023-02-16
```

In this example, we've created the `Admin` subclass with arguments of `root` for `first_name`, `admin` as `last_name`, and `ABRACADABRA` as the `secret_code` value. In the output, we can see that all the attributes are present, including the updated attribute `expiry_date` from the `Admin` subclass. We can also see that the `User` subclass hasn't been affected by any changes that were made to the `Admin` subclass.

Note: we can also test this by trying to output the `secret_code` parameter from the `User` subclass.

EXAMPLE

```
TestUser = User('Artic','Smith')
print(TestUser.first_name)
print(TestUser.last_name)
print(TestUser.secret_code)
print(TestUser.expiry_date)
```

However, if we did that, we would get an error.

```
root
admin
ABRACADABRA
2124-09-23
--------
Artic
```

```
Smith
Traceback (most recent call last):
  File "/home/main.py", line 34, in <module>
    print(TestUser.secret_code)
AttributeError: 'User' object has no attribute 'secret_code'
```

This is because the `secret_code` attribute only belongs to the `Admin` subclass and not the `User` subclass. What we define in a subclass does not reflect to other subclasses of the same base class.

Methods in the base class work the same for subclasses. Let's add a new method called `showexpiry()` in the `Member` base class.

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

  def show_expiry(self):
    return f'{self.first_name} {self.last_name} expires on {self.expiry_date}'
```

When called, the `show_expiry()` method should return a formatted string that contains the member's first name, last name, a string, and expiration date. Leaving the subclasses untouched, we'll make the same call to that base class method.

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

  def show_expiry(self):
    return f'{self.first_name} {self.last_name} expires on {self.expiry_date}'
```

```
#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100

  def __init__(self, first, last, secret):
    super().__init__(first,last)
    self.secret_code = secret

#Subclass for us to use for normal users
class User(Member):
  pass

TestAdmin = Admin('root','admin','ABRACADABRA')
print(TestAdmin.first_name)
print(TestAdmin.last_name)
print(TestAdmin.secret_code)
print(TestAdmin.show_expiry())

print("--------")

TestUser = User('Artic','Smith')
print(TestUser.first_name)
print(TestUser.last_name)
print(TestUser.show_expiry())
```

TRY IT

Directions: Update your code to reflect the new method in the base class and run the program.

```
root
admin
ABRACADABRA
root admin expires on 2124-09-23
--------
Arctic
Smith
Arctic Smith expires on 2025-10-17
```

There are also instances where we may have the same method name across the base class and the subclass. When that happens, Python will use the most specific one that's tied to the subclass. It will use the more generic method if nothing in that subclass has that method name.

For example, we'll add a method called `show_status()` to the base class and each subclass. The `show_status()` method returns a formatted string of first name, last name, and what class it is from.

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

  def show_expiry(self):
    return f'{self.first_name} {self.last_name} expires on {self.expiry_date}'

  def show_status(self):
    return f'{self.first_name} {self.last_name} is a Member'

#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100

  def __init__(self, first, last, secret):
    super().__init__(first,last)
    self.secret_code = secret
  def show_status(self):
    return f'{self.first_name} {self.last_name} is an Admin'

#Subclass for us to use for normal users
class User(Member):
  def show_status(self):
    return f'{self.first_name} {self.last_name} is a User'

TestMember = Member('Sophia','Python')
print(TestMember.show_status())

print("--------")

TestAdmin = Admin('root','admin','ABRACADABRA')
print(TestAdmin.show_status())

print("--------")
```

```
TestUser = User('Artic','Smith')
print(TestUser.show_status())
```

TRY IT

Directions: Add the new method to all classes as well as the `print()` functions below them.

When we output the same `show_status()` method from each object, this is the result that we should see.

```
Sophia Python is a Member
--------
root admin is an Admin
--------
Arctic Smith is a User
```

If we removed `show_status()` from the `User` class, let's see what happens:

EXAMPLE

```
import datetime

class Member:
  expiry_days = 365
  def __init__(self, first, last):
    self.first_name = first
    self.last_name = last

    self.expiry_date = datetime.date.today() + datetime.timedelta(days = self.expiry_days)

  def show_expiry(self):
    return f'{self.first_name} {self.last_name} expires on {self.expiry_date}'

  def show_status(self):
    return f'{self.first_name} {self.last_name} is a Member'

#Subclass for us to use for administrators
class Admin(Member):
  expiry_days = 365 * 100

  def __init__(self, first, last, secret):
    super().__init__(first,last)
    self.secret_code = secret
  def show_status(self):
    return f'{self.first_name} {self.last_name} is an Admin'
```

```
#Subclass for us to use for normal users
class User(Member):
  pass

TestMember = Member('Sophia','Python')
print(TestMember.show_status())

print("--------")

TestAdmin = Admin('root','admin','ABRACADABRA')
print(TestAdmin.show_status())

print("--------")

TestUser = User('Artic','Smith')
print(TestUser.show_status())
```

Notice that Artic Smith is showing the word “Member” instead.

```
Sophia Python is a Member
--------
root admin is an Admin
--------
Arctic Smith is a Member
```

That’s because no `show_status()` method was defined in the `User` class. As such, it had to look in the `Member` class (base class) and use the one that was found there. If it was not found there, and if there was another base class, it would look there.

TRY IT

Directions: Try removing the method from the `User` class and see if you get the same output.

TERM TO KNOW

super()

The `super()` function allows a subclass to access the base class attributes and methods.

SUMMARY

In this lesson, we learned that subclasses do not only have to use the attributes and methods that they inherit from the base class. **Subclasses can be customized** or extended from the base class. To do this, subclasses can have their own `__init__` method. With the use of the `super()` function, a subclass can call the base class methods directly. We were able to build out subclasses with extra attributes and methods. We also learned that if the name of a method is the same, Python will use the most specific one that's tied to the subclass. It will use the base class method if nothing in that subclass has that method name.

Best of luck in your learning!

TERMS TO KNOW

super()
The `super()` function allows a subclass to access the base class attributes and methods.