Introduction to Inheritance

by Sophia

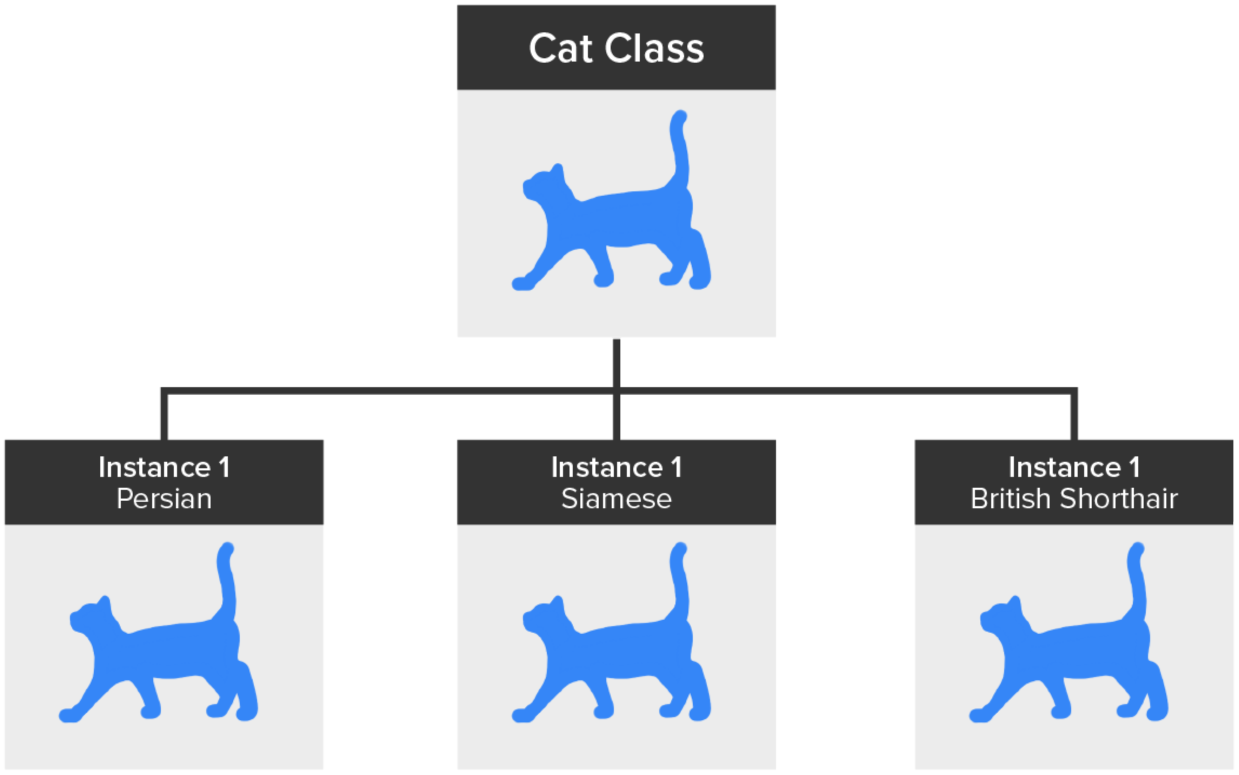
WHAT'S COVERED

In this lesson, we'll be looking at inheritance and how properties and methods are inherited from a parent class. Specifically, this lesson covers:

1. Real-World Example

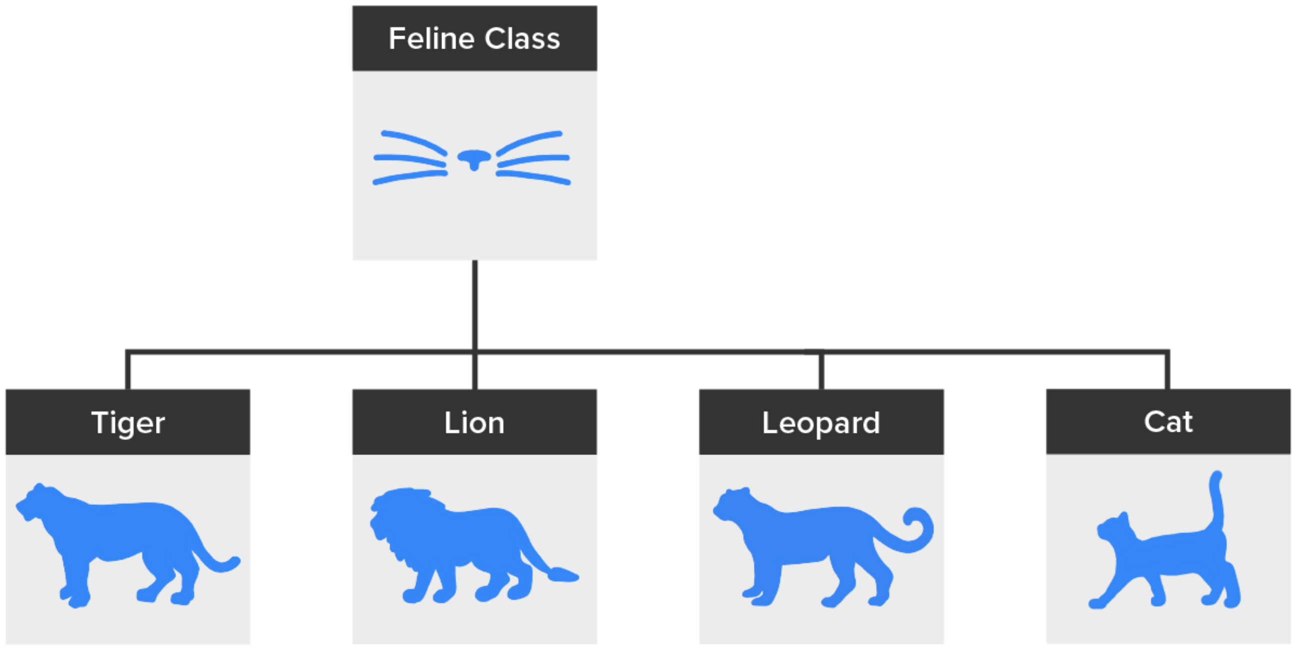
Frequently when we hear about object-oriented programming, there is a lot of discussion about class inheritance and subclasses, but those terms can sound complex. However, the concepts are things that you actually see in the real world.

If we consider a cat to be a Python class, we can put together all of the possible cats as a class of animals that we call cats. Even though each cat may be unique, every cat is still a type of cat as they are members of a class that we call "cat".



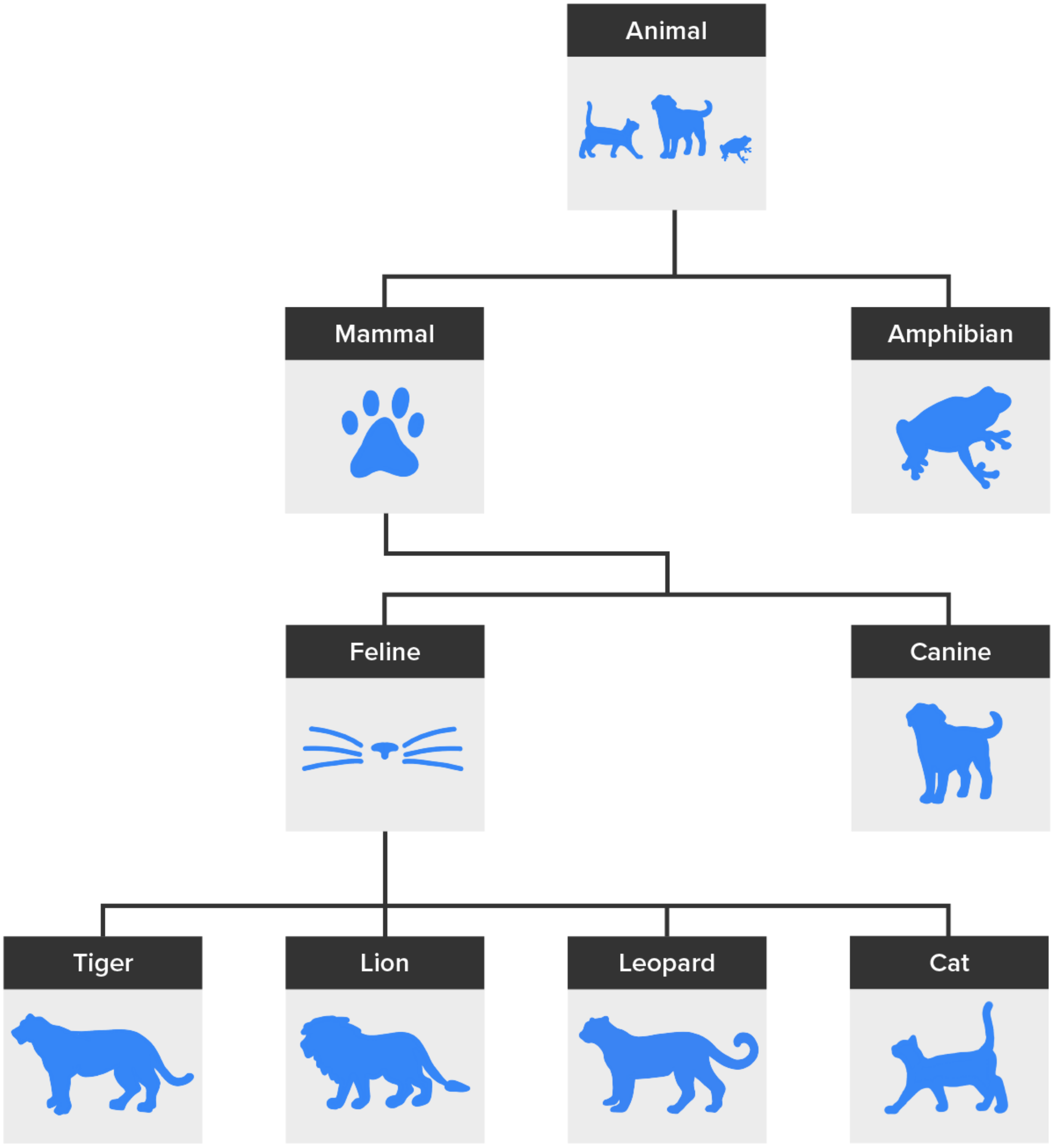
Looking at the graphic above, Persian, Siamese, and British Shorthair are unique breeds of cats but what makes cats similar to one another are the characteristics that they all inherit from the class Cat.

The whole concept around classes and class inheritance in Python didn’t simply get created by a bunch of developers trying to make things more difficult. Much of the information in the real world can be stored, categorized, and understood by using classes and subclasses. For example, you may have noticed other creatures similar to cats that are in the real world. They would be similar to the cats in that they inherit features from a higher-level class that we could call felines.



We could have the class Feline and under that, we would have tiger, leopard, lion, and cat as subclasses.

We don’t have to stop there either. On the way up above Feline, we could have mammals as a layer on top of that, as all felines are mammals. Then on top of that, we could have animals above that as all mammals are animals.



Obviously, this concept doesn't just apply to cats, but it can apply to other animals like dogs or amphibians. If you search online to see the hierarchy of animals, you'll see that there are many ways to classify animals and see how inheritance works all the way down the line.

From a coding perspective, the easiest way to do inheritance is to create "child" classes (or subclasses) from "parent" classes or (base classes).

KEY CONCEPT

Ever heard the term, “You look just like your mother/father”?

You have your mother's eyes, you have the same patience as your father... The reason is behind the term “inheritance”. You inherited those objects or traits without having to do anything.

From the Python and programming world, inheritance means the same thing—inheriting characteristics (data) and behaviors (methods) from a “parent” class to a “child” class without modifying anything.

BIG IDEA

You will hear a few terms in this Challenge that may be defined differently in other resources you may be using, so it makes sense to call this out.

Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

The **base class** is also known as the parent class. It is the class that is being inherited from.

The **subclass** is also known as the child class. It is the class that inherits from another class. A subclass is known to “extend” the base class, meaning that it can define additional data that will not affect the base class.

From here on we will be using the terms base class and subclass as opposed to “parent” and “child”.

A base class defines all of the things that apply to all instances of that class. Therefore, a subclass automatically inherits what was defined in the base class. Once the subclass is created, though, any data/methods defined in that subclass are relevant only to that subclass. Anything that is defined in the subclass will not change or replace anything that is coming from the base class.

So what is the benefit of inheritance in Python?

Inheritance allows us, as programmers, to create subclasses that have the properties and methods of an existing class. This supports code reusability, maintenance, and optimization; and it speeds up development time. We don’t have to “reinvent the wheel” with each class. We can create it once and use it multiple times. Back to our feline example. If we have a base class with all the common attributes and standard behaviors (methods) that every feline should have, we should never need to alter that base class when creating subclasses. We inherit all those properties and methods automatically, so we do not need to define those again. Also, if something changes in the base class, once that change is made, it automatically cascades those changes to each subclass.

TERMS TO KNOW

Inheritance
Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

Base Class
The base class is also known as the parent class. It is the class that is being inherited from.

Subclass
The subclass is also known as the child class. It is the class that inherits from another class. A subclass is known to “extend” the base class, meaning that it can define additional data that will not affect the base class.

---

2. Base Class

Subclasses inherit all of the attributes (properties) and methods (behaviors/actions) of a higher level class or the base class. The base class is no different than any other class but it is simply the class that a subclass would inherit from.

To help us define these inheritance concepts and get looking at code, let’s start with a class called `Member`.

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)
```

In the code, you may find it quite familiar as we used something very similar in a prior lesson. We are importing the module `datetime` to use the current date functionality. Again, we will be seeing module importation and usage in the next challenge. Then we defined our class called `Member`. There is an attribute variable created called `expiry_date`, and we set it to 365. It is used to calculate the expiration date from the current date. We then defined the `_init_` method with three parameters. The first is the `self` parameter which is used to point to the instance being referenced. Next are the first and last parameters which are then assigned to the variables `first_name` and `last_name`. The `expiry_date` variable is set to 365 days after today when we run the code. The `.timedelta()` method (included in the `datetime` module) calculates the difference of time between the two arguments.

Next, we will create an instance of this class with the variable `TestMember` and create some `print()` functions so we can see some output to the screen.

Let’s give this a test.

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)

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

Below are the results of this small program.

```
Sophia
Python
2023-02-16
```

See anything new? Not really. This is very similar to the class we built out in the last challenge. We created this class to use as a base class for the examples to follow. Now our intent is going to be to create two different kinds of subclasses of the `Member` class.

In the next topic, we will create an `Admin` and a `User` subclass. Both of these subclasses will have attributes that the `Member` class includes. These new classes will be subclasses of our `Member` class. They will automatically include the attributes and methods of our base class (`Member`).

This way, since we define those types of members as a subclass of `Member`, they will automatically get the same attributes and methods if there are any defined.

3. Subclasses

To create and define a subclass, we need to ensure that the subclass is below the base class with no indentation. This is because the subclass is not part of or contained within the base class. The subclass needs to be below it because the base class needs to be defined before the subclass can use it. We would use the following syntax:

EXAMPLE

```
class subclassname(baseclassname):
```

We would replace “subclassname” with what we want to name the subclass. Then we would replace “baseclassname” with the name of the base class. For example, to make a subclass of our base class `Member` and name it `Admin`, we would add the following code.

EXAMPLE

```
class Admin(Member):
```

If we leave the subclass empty (with no additional lines of code), we won't be able to test it as we'll get an error that the class is empty. However, we can use the `pass` statement as the first command. We used the `pass` statement back in Unit 1 when we were building conditional statements. Does this ring a bell?

EXAMPLE

```
if temperature < 0 :
    pass          # need to handle negative temperatures!!
```

HINT

If you remember, `pass` is a reserved keyword in Python; a `pass` statement is essentially a null statement (a statement that will do nothing). In this example, we placed a `pass` statement in an `if` statement (condition) followed by a comment explaining that we would need to come back to the `if` statement and complete the statement later. The difference between a `pass` statement and comments is that comments are completely ignored by the interpreter whereas the `pass` statement is not. The `pass` statement is seen but tells the compiler to do nothing.

We can use the `pass` statement for classes as well. By using the `pass` statement, we're telling Python that we know the class is empty but don't throw an error message and just let it pass.

The code to do that would look like the following.

EXAMPLE

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

We can do the same thing for the `User` class as well.

EXAMPLE

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

To just do a quick test to ensure that this works, we'll create an `Admin` and a `User` instance as subclasses that will simply inherit all of the attributes and methods from our base class (`Member`). We will also include some `print()` functions, so once we run this program, we can see some output.

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)
```

We added some different arguments to the subclass instances besides `Sophia` and `Python` so the output will be clearer. Again, using the same `print()` functions for each of the instances, the results of this would look like the following.

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

```
Smith
2023-02-16
```

Here we see the outputs for the base class (`Member`) and both instances of the subclasses (`Admin` and `User`). Remember, the subclasses `Admin` and `User` had different arguments but nothing defined in its body of code. We simply “told” the compiler to ignore any errors since the subclasses were empty using the `pass` statement. All of the attributes and methods from the base class were inherited. All the parameters that the base class `Member` accepts are also available to the subclass. However, this will generally not always be the case as subclasses could inherit from multiple base classes. In the next lesson, we’ll explore subclasses in more detail.

TRY IT

Directions: If you want to try your hand at building out a base class and creating a subclass, go ahead and practice on some of the code above or create your own.

SUMMARY

In this lesson, we learned about inheritance by looking at some **real-world examples**. Using the relationship model of inheritance when building and using classes increases the efficiency of our programs. It also follows the principle of "build once, use many times." We saw that any defined class can be the **base class** from whom attributes (properties) and methods (behaviors/actions) can be inherited. **Subclasses** were defined as the classes that inherit from the base class. A subclass is known to “extend” the base class, meaning that it can define additional data that will not affect the base class. Finally, we saw the output of two subclass instances inherit attributes from a base class that we built.

Best of luck in your learning!

TERMS TO KNOW

Base Class

The base class is also known as the parent class. It is the class that is being inherited from.

Inheritance

Inheritance is a relationship model that allows us to define a class that inherits all the attributes and methods from another class.

Subclass

The subclass is also known as the child class. It is the class that inherits from another class. A subclass is known to “extend” the base class, meaning that it can define additional data that will not affect the base class.