22.2. Inheriting Variables and Methods

22.2.1. Mechanics of Defining a Subclass

We said that inheritance provides us a more elegant way of, for example, creating `Dog` and `Cat` types, rather than making a very complex `Pet` class. In the abstract, this is pretty intuitive: all pets have certain things, but dogs are different from cats, which are different from birds. Going a step further, a Collie dog is different from a Labrador dog, for example. Inheritance provides us with an easy and elegant way to represent these differences.

Basically, it works by defining a new class, and using a special syntax to show what the new sub-class *inherits from* a super-class. So if you wanted to define a `Dog` class as a special kind of `Pet`, you would say that the `Dog` type inherits from the `Pet` type. In the definition of the inherited class, you only need to specify the methods and instance variables that are different from the parent class (the **parent class**, or the **superclass**, is what we may call the class that is *inherited from*. In the example we're discussing, `Pet` would be the superclass of `Dog` or `Cat`).

Here is an example. Say we want to define a class `Cat` that inherits from `Pet`. Assume we have the `Pet` class that we defined earlier.

We want the `Cat` type to be exactly the same as `Pet`, *except* we want the sound cats to start out knowing "meow" instead of "mrrp", and we want the `Cat` class to have its own special method called `chasing_rats`, which only `Cat` s have.

For reference, here's the original Tamagotchi code

Save & Run Original - 1 of 1 Show in CodeLens

```
from random import randrange

# Here's the original Pet class
class Pet():
    boredom_decrement = 4
    hunger_decrement = 6
    boredom_threshold = 5
    hunger_threshold = 10
    sounds = ['Mrrp']
    def __init__(self, name = "Kitty"):
        self.name = name
        self.hunger = randrange(self.hunger_threshold)
        self.boredom = randrange(self.boredom_threshold)
        self.sounds = self.sounds[:]  # copy the class attribute, so that when we m
```

Activity: 1 -- ActiveCode (inheritance_cat_example)

All we need is the few extra lines at the bottom of the ActiveCode window! The elegance of inheritance allows us to specify just the differences in the new, inherited class. In that extra code, we make sure the `Cat` class inherits from the `Pet` class. We do that by putting the word Pet in parentheses, `class Cat(Pet):`. In the definition of the class `Cat`, we only need to define the things that are different from the ones in the `Pet` class.

In this case, the only difference is that the class variable `sounds` starts out with the string `"Meow"` instead of the string `"mrrp"`, and there is a new method `chasing_rats`.

We can still use all the `Pet` methods in the `Cat` class, this way. You can call the `__str__` method on an instance of `Cat` to `print` an instance of `Cat`, the same way you could call it on an instance of `Pet`, and the same is true for the `hi` method – it's the same for instances of `Cat` and `Pet`. But the `chasing_rats` method is special: it's only usable on `Cat` instances, because `Cat` is a subclass of `Pet` which has that additional method.

In the original Tamagotchi game in the last chapter, you saw code that created instances of the `Pet` class. Now let's write a little bit of code that uses instances of the `Pet` class AND instances of the `Cat` class.

Save & Run Original - 1 of 1 Show in CodeLens

```
p1 = Pet("Fido")
print(p1) # we've seen this stuff before!

p1.feed()
p1.hi()
print(p1)

cat1 = Cat("Fluffy")
print(cat1) # this uses the same __str__ method as the Pets do

cat1.feed() # Totally fine, because the cat class inherits from the Pet class!
cat1.hi()
print(cat1)

print(cat1.chasing_rats())
```

```
    I'm Fido.  I feel happy.
Mrrp
    I'm Fido.  I feel happy.
    I'm Fluffy.  I feel happy.
Meow
    I'm Fluffy.  I feel happy.
What are you doing, Pinky? Taking over the world?!
```

Activity: 2 -- ActiveCode (tamagotchi_2)

And you can continue the inheritance tree. We inherited `Cat` from `Pet`. Now say we want a subclass of `Cat` called `Cheshire`. A Cheshire cat should inherit everything from `Cat`, which means it inherits everything that `Cat` inherits from `Pet`, too. But the `Cheshire` class has its own special method, `smile`.

Save & Run | Original - 1 of 1 | Show in CodeLens

```
class Cheshire(Cat): # this inherits from Cat, which inherits from Pet

    def smile(self): # this method is specific to instances of Cheshire
        print(":D :D :D")

# Let's try it with instances.
cat1 = Cat("Fluffy")
cat1.feed() # Totally fine, because the cat class inherits from the Pet class!
cat1.hi() # Uses the special Cat hello.
print(cat1)

print(cat1.chasing_rats())

new_cat = Cheshire("Pumpkin") # create a Cheshire cat instance with name "Pumpkin"
new_cat.hi() # same as Cat!
```

```
Meow
     I'm Fluffy.  I feel happy.
What are you doing, Pinky? Taking over the world?!
Meow
:D :D :D
Mrrp
```

Activity: 3 -- ActiveCode (inheritance_cheshire_example)

22.2.2. How the interpreter looks up attributes

So what is happening in the Python interpreter when you write programs with classes, subclasses, and instances of both parent classes and subclasses?

This is how the interpreter looks up attributes:

1. First, it checks for an instance variable or an instance method by the name it's looking for.
2. If an instance variable or method by that name is not found, it checks for a class variable. (See the previous chapter for an explanation of the difference between **instance variables** and **class variables**.)
3. If no class variable is found, it looks for a class variable in the parent class.
4. If no class variable is found, the interpreter looks for a class variable in THAT class's parent (the "grandparent" class).
5. This process goes on until the last ancestor is reached, at which point Python will signal an error.

Let's look at this with respect to some code.

Say you write the lines:

```
new_cat = Cheshire("Pumpkin")
print(new_cat.name)
```

In the second line, after the instance is created, Python looks for the instance variable `name` in the `new_cat` instance. In this case, it exists. The name on this instance of `Cheshire` is `Pumpkin`. There you go!

When the following lines of code are written and executed:

```
cat1 = Cat("Sepia")
cat1.hi()
```

The Python interpreter looks for `hi` in the instance of `Cat`. It does not find it, because there's no statement of the form `cat1.hi = ...`. (Be careful here – if you *had* set an instance variable on Cat called `hi` it would be a bad idea, because you would not be able to use the **method** that it inherited anymore. We'll see more about this later.)

Then it looks for hi as a class variable (or method) in the class Cat, and still doesn't find it.

Next, it looks for a class variable `hi` on the parent class of `Cat`, `Pet`. It finds that – there's a **method** called `hi` on the class `Pet`. Because of the `()` after `hi`, the method is invoked. All is well.

However, for the following, it won't go so well

```
p1 = Pet("Teddy")
p1.chasing_rats()
```

The Python interpreter looks for an instance variable or method called `chasing_rats` on the `Pet` class. It doesn't exist. `Pet` has no parent classes, so Python signals an error.

Check your understanding

intro-9-1: After you run the code, `new_cat = Cheshire("Pumpkin")`, how many instance variables exist for the new_cat instance of Cheshire?

- A. 1
- B. 2
- C. 3
- D. 4

Check me | Compare me

✔ Neither Cheshire nor Cat defines an __init__ constructor method, so the grandaprent class, Pet, will have it's __init__ method called. That constructor method sets the instance variables name, hunger, boredom, and sounds.

Activity: 4 -- Multiple Choice (question_inheritance_1)

intro-9-2: What would print after running the following code:

```
new_cat = Cheshire("Pumpkin")
class Siamese(Cat):
  def song(self):
    print("We are Siamese if you please. We are Siamese if you don't please.")
another_cat = Siamese("Lady")
another_cat.song()
```

- (•) A. We are Siamese if you please. We are Siamese if you don't please.
- ( ) B. Error
- ( ) C. Pumpkin
- ( ) D. Nothing. There's no print statement.

Check me | Compare me

✔ another_cat is an instance of Siamese, so its song() method is invoked.

Activity: 5 -- Multiple Choice (question_inheritance_2)

intro-9-3: What would print after running the following code:

```
new_cat = Cheshire("Pumpkin")
class Siamese(Cat):
  def song(self):
    print("We are Siamese if you please. We are Siamese if you don't please.")
another_cat = Siamese("Lady")
new_cat.song()
```

- ( ) A. We are Siamese if you please. We are Siamese if you don't please.
- (•) B. Error
- ( ) C. Pumpkin
- ( ) D. Nothing. There's no print statement.

Check me | Compare me

✔ You cannot invoke methods defined in the Siamese class on an instance of the Cheshire class. Both are subclasses of Cat, but Cheshire is not a subclass of Siamese, so it doesn't inherit its methods.

Activity: 6 -- Multiple Choice (question_inheritance_3)

You have attempted 7 of 6 activities on this page

22.1. Introduction: Class Inheritance">
22.3. Overriding Methods">
troduction: Class Inheritance">
✔ Completed. Well Done!
22.3. Overriding Methods">Next Section - 22.3. Overriding Methods

| Back to top