Introducción: Herencia de Clases

Las clases pueden "heredar" métodos y variables de otras clases. Veremos la mecánica de cómo funciona esto en secciones posteriores. Primero, sin embargo, vamos a motivar por qué esto puede ser valioso. Resulta que la herencia no te permite hacer nada que no pudieras hacer sin ella, pero hace algunas cosas mucho más elegantes. También encontrarás que es útil cuando alguien más ha definido una clase en un módulo o biblioteca, y tú sólo quieres anular algunas cosas sin tener que reimplementar todo lo que han hecho.

Consideremos nuestro juego Tamagotchi. Supongamos que queremos hacer algunas clases diferentes de mascotas que tengan la misma estructura que otras mascotas, pero que tengan algunos atributos diferentes o se comporten de forma un poco distinta. Por ejemplo, supongamos que las mascotas perro deben mostrar su estado emocional de forma un poco diferente a los gatos o actuar de forma diferente cuando tienen hambre o cuando se les pide que busquen algo.

Podrías implementar esto haciendo una variable de instancia para el tipo de mascota y despachando sobre esa variable de instancia en varios métodos.

Aquí, sólo hemos implementado perros, gatos y otros - pero puedes imaginar las posibilidades.

Si hubiera muchos tipos diferentes de mascotas, esos métodos empezarían a tener largas y complejas cláusulas de código if..elif..elif, que pueden ser confusas. Y necesitarías eso en cada método donde el comportamiento fuera diferente para diferentes tipos de mascotas. La herencia de clases nos dará una forma más elegante de hacerlo.

Herencia de variables y métodos

Mecánica de la definición de una subclase

Dijimos que la herencia nos proporciona una forma más elegante de, por ejemplo, crear tipos Dog y Cat, en lugar de hacer una clase Pet muy compleja. En abstracto, esto es bastante intuitivo: todas las mascotas tienen ciertas cosas, pero los perros son diferentes de los gatos, que son diferentes de los pájaros. Yendo un paso más allá, un perro Collie es diferente de un perro Labrador, por ejemplo. La herencia nos proporciona una forma fácil y elegante de representar estas diferencias.

Básicamente, funciona definiendo una nueva clase, y utilizando una sintaxis especial para mostrar lo que la nueva subclase hereda de una superclase. Así, si se quiere definir una clase Dog como una clase especial de Pet, se diría que el tipo Dog hereda del tipo Pet. En la definición de la clase heredada, sólo hay que especificar los métodos y las variables de instancia que son diferentes de la clase padre (la clase padre, o la superclase, es como podemos llamar a la clase de la que se hereda. En el ejemplo que estamos discutiendo, Pet sería la superclase de Dog o Cat).

He aquí un ejemplo. Digamos que queremos definir una clase Cat que herede de Pet. Supongamos que tenemos la clase Pet que hemos definido antes.

Queremos que la clase Cat sea exactamente igual que Pet, excepto que queremos que los gatos de sonido comiencen sabiendo "meow" en lugar de "mrrp", y queremos que la clase Cat tenga su propio método especial llamado chasing_rats, que sólo los Gatos tienen.

Como referencia, aquí está el código original de Tamagotchi

```
from random import randrange

class Pet():
    boredom_decrement = 4
    hunger_decrement = 6
    boredom_threshold = 5
    hunger_threshold = 10
    sounds = ['Mrrp']
    def __init__(self, name = "Kitty", pet_type="dog"):
        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 make changes to it,
        #.............................we won't affect the other Pets in the class
        self.pet_type = pet_type

    def clock_tick(self):
        self.boredom += 1
        self.hunger += 1

    def mood(self):
        if self.hunger <= self.hunger_threshold and self.boredom <= self.boredom_threshold:
            if self.pet_type == "dog": # if the pet is a dog, it will express its mood in different
        #.............................ways from a cat or any other type of animal
                return "happy"
            elif self.pet_type == "cat":
                return "happy, probably"
            else:
                return "HAPPY"
        elif self.hunger > self.hunger_threshold:
            if self.pet_type == "dog": # same for hunger -- dogs and cats will express their hunger  \
            #.........................a little bit differently in this version of the class definition
                return "hungry, arf"
            elif self.pet_type == "cat":
                return "hungry, meeeeow"
            else:
                return "hungry"
        else:
            return "bored"

    def __str__(self):
        state = "     I'm " + self.name + ". "
        state += " I feel " + self.mood() + ". "
        return state

    def hi(self):
        print(self.sounds[randrange(len(self.sounds))])
        self.reduce_boredom()

    def teach(self, word):
        self.sounds.append(word)
        self.reduce_boredom()

    def feed(self):
        self.reduce_hunger()

    def reduce_hunger(self):
        self.hunger = max(0, self.hunger - self.hunger_decrement)

    def reduce_boredom(self):
        self.boredom = max(0, self.boredom - self.boredom_decrement)
```

La elegancia de la herencia nos permite especificar sólo las diferencias en la nueva clase heredada. En ese código extra, nos aseguramos de que la clase Cat herede de la clase Pet. Lo hacemos poniendo la palabra Pet entre paréntesis, clase Cat(Pet):. En la definición de la clase Cat, sólo necesitamos definir las cosas que son diferentes de las de la clase Pet.

En este caso, la única diferencia es que la variable de clase sonidos comienza con la cadena "Meow" en lugar de la cadena "mrrp", y hay un nuevo método chasing_rats.

Podemos seguir utilizando todos los métodos Pet de la clase Cat, de esta manera. Puedes llamar al método __str__ en una instancia de Cat para imprimir una instancia de Cat, de la misma manera que podrías llamarlo en una instancia de Pet, y lo mismo es cierto para el método hi - es el mismo para instancias de Cat y Pet. Pero el método chasing_rats es especial: sólo es utilizable en instancias de Cat, porque Cat es una subclase de Pet que tiene ese método adicional.

```
# Here's the new definition of class Cat, a subclass of Pet.
class Cat(Pet): # the class name that the new class inherits from goes
    #.............in the parentheses, like so.
    sounds = ['Meow']

    def chasing_rats(self):
        return "What are you doing, Pinky? Taking over the world?!"
```

```
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())

#print(p1.chasing_rats()) # This line will give us an error.
#...........The Pet class doesn't have this method!
```

Y puedes continuar con el árbol de la herencia. Heredamos Cat de Pet Ahora digamos que queremos una subclase de Cat llamada Cheshire. Un gato Cheshire debería heredar todo de Cat, lo que significa que hereda todo lo que Cat hereda de Pet, también. Pero la clase Cheshire tiene su propio método especial, smile.

```
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!
new_cat.chasing_rats() # OK, because Cheshire inherits from Cat
new_cat.smile() # Only for Cheshire instances (and any classes that you make inherit
# ..............from Cheshire)

# cat1.smile() # This line would give you an error, because the Cat class does not have
#...............this method!

# None of the subclass methods can be used on the parent class, though.
p1 = Pet("Teddy")
p1.hi() # just the regular Pet hello

#p1.chasing_rats() # This will give you an error -- this method doesn't exist on instances
#of the Pet class.
#p1.smile() # This will give you an error, too. This method does not exist on instances
#of the Pet class.
```

Sobreescritura de métodos¶

Si se define un método para una clase, y también se define para su clase padre, se llama al método de la subclase y no al del padre. Esto se desprende de las reglas de búsqueda de atributos que viste en la sección anterior. Podemos utilizar la misma idea para entender la anulación de métodos.

Volvamos a nuestra idea de hacer que los Gatos, Perros y otras mascotas generen una cadena para su "estado de ánimo" de forma diferente.

Aquí está de nuevo la clase Pet original.

```
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 make changes
        #to it, we won't affect the other Pets in the class

    def clock_tick(self):
        self.boredom += 1
        self.hunger += 1

    def mood(self):
        if self.hunger <= self.hunger_threshold and self.boredom <= self.boredom_threshold:
            return "happy"
        elif self.hunger > self.hunger_threshold:
            return "hungry"
        else:
            return "bored"

    def __str__(self):
        state = "     I'm " + self.name + ". "
        state += " I feel " + self.mood() + ". "
        # state += "Hunger %d Boredom %d Words %s" % (self.hunger, self.boredom, self.sounds)
        return state

    def hi(self):
        print(self.sounds[randrange(len(self.sounds))])
        self.reduce_boredom()

    def teach(self, word):
        self.sounds.append(word)
        self.reduce_boredom()

    def feed(self):
        self.reduce_hunger()

    def reduce_hunger(self):
        self.hunger = max(0, self.hunger - self.hunger_decrement)

    def reduce_boredom(self):
        self.boredom = max(0, self.boredom - self.boredom_decrement)
```

Ahora hagamos dos subclases, Dog y Cat. Los perros son siempre felices a menos que estén aburridos y hambrientos. Los gatos, en cambio, sólo son felices si se les alimenta y si su nivel de aburrimiento está en un rango estrecho e, incluso entonces, sólo con probabilidad 1/2.

```
class Cat(Pet):
    sounds = ['Meow']

    def mood(self):
        if self.hunger > self.hunger_threshold:
            return "hungry"
        if self.boredom <2:
            return "grumpy; leave me alone"
        elif self.boredom > self.boredom_threshold:
            return "bored"
        elif randrange(2) == 0:
            return "randomly annoyed"
        else:
            return "happy"

class Dog(Pet):
    sounds = ['Woof', 'Ruff']

    def mood(self):
        if (self.hunger > self.hunger_threshold) and (self.boredom > self.boredom_threshold):
            return "bored and hungry"
        else:
            return "happy"

c1 = Cat("Fluffy")
d1 = Dog("Astro")

c1.boredom = 1
print(c1.mood())
c1.boredom = 3
for i in range(10):
    print(c1.mood())
print(d1.mood())
```

Invocación del método de la clase padre

A veces la clase padre tiene un método útil, pero usted sólo necesita ejecutar un poco de código extra al ejecutar el método de la subclase. Puede anular el método de la clase padre en el método de la subclase con el mismo nombre, pero también invocar el método de la clase padre. Así es como.

Digamos que quieres que la subclase Dog de Pet diga "¡Arf! Thanks!!!" cuando se llame al método alimentación, además de ejecutar el código del método original.

Y aquí hay una subclase que anula feed() invocando el método feed() de la clase padre; entonces también ejecuta una línea extra de código. Nótese la forma poco elegante de invocar el método de la clase padre. Hacemos referencia explícita a Pet.feed para obtener el objeto método/función. Lo invocamos con paréntesis. Sin embargo, como no estamos invocando el método de la forma normal, con <obj>.nombredelmétodo, tenemos que pasar explícitamente una instancia como primer parámetro. En este caso, la variable self en Dog.feed() estará ligada a una instancia de Dog, por lo que podemos pasar simplemente self: Pet.feed(self).

```
from random import randrange

class Dog(Pet):
    sounds = ['Woof', 'Ruff']

    def feed(self):
        Pet.feed(self)
        print("Arf! Thanks!")

d1 = Dog("Astro")

d1.feed()
```

Nota

Hay una forma mejor de invocar un método de la superclase. Desgraciadamente, la implementación de python en nuestras ventanas de ActiveCode no la soporta, por lo que no la vamos a utilizar aquí. En ese método alternativo, llamaríamos a super().feed(). Esto es bueno porque es más fácil de leer, y también porque pone la especificación de la clase de la que hereda Dog en un solo lugar, class Dog(Pet). En otros lugares, sólo se hace referencia a super() y python se encarga de buscar que la clase padre (super) de Dog es Pet.

Esta técnica se utiliza muy a menudo con el método __init__ para una subclase. Supongamos que se definen algunas variables de instancia adicionales para la subclase. Cuando se invoca el constructor, se pasan todos los parámetros normales de la clase padre, más los extra de la subclase. El método __init__ de la subclase almacena entonces los parámetros extra en variables de instancia y llama al método __init__ de la clase padre para almacenar los parámetros comunes en variables de instancia y hacer cualquier otra inicialización que haga normalmente.

Digamos que queremos crear una subclase de Pet, llamada Bird, y queremos que tome un parámetro extra, chirp_number, con un valor por defecto de 2, y que tenga una variable de instancia extra, self.chirp_number. Entonces, usaremos esto en el método hi para hacer más de un sonido.

```
class Bird(Pet):
    sounds = ["chirp"]
    def __init__(self, name="Kitty", chirp_number=2):
        Pet.__init__(self, name) # call the parent class's constructor
        # basically, call the SUPER -- the parent version -- of the cons
        self.chirp_number = chirp_number # now, also assign the new inst

    def hi(self):
        for i in range(self.chirp_number):
            print(self.sounds[randrange(len(self.sounds))])
        self.reduce_boredom()

b1 = Bird('tweety', 5)
b1.teach("Polly wanna cracker")
b1.hi()
```

Tamagotchi revisitado

Usando lo que sabemos sobre la herencia de clases, podemos hacer una nueva versión del juego Tamagotchi, en la que podemos adoptar diferentes tipos de mascotas que son ligeramente diferentes entre sí.

Y ahora podemos jugar al juego Tamagotchi con algunos pequeños cambios, de forma que podamos adoptar diferentes tipos de mascotas.

```
import sys
sys.setExecutionLimit(60000)
from random import randrange

class Pet(object):
    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
        make changes to it, we won't affect the other Pets in the class

    def clock_tick(self):
        self.boredom += 1
        self.hunger += 1

    def mood(self):
        if self.hunger <= self.hunger_threshold and self.boredom <=
        self.boredom_threshold:
            return "happy"
        elif self.hunger > self.hunger_threshold:
            return "hungry"
        else:
            return "bored"

    def __str__(self):
        state = "     I'm " + self.name + ". "
        state += " I feel " + self.mood() + ". "
        # state += "Hunger %d Boredom %d Words %s" % (self.hunger, self.boredom,
        self.sounds)
        return state

    def hi(self):
        print(self.sounds[randrange(len(self.sounds))])
        self.update_boredom()

    def teach(self, word):
        self.sounds.append(word)
        self.update_boredom()

    def feed(self):
        self.update_hunger()

    def update_hunger(self):
        self.hunger = max(0, self.hunger - self.hunger_decrement)

    def update_boredom(self):
        self.boredom = max(0, self.boredom - self.boredom_decrement)

class Cat(Pet):
    sounds = ['Meow']

    def mood(self):
        if self.hunger > self.hunger_threshold:
            return "hungry"
        if self.boredom <2:
            return "grumpy; leave me alone"
        elif self.boredom > self.boredom_threshold:
            return "bored"
        elif randrange(2) == 0:
            return "randomly annoyed"
        else:
            return "happy"

class Dog(Pet):
    sounds = ['Woof', 'Ruff']
```

```
69
70      def mood(self):
71          if (self.hunger > self.hunger_threshold) and (self.boredom >
            self.boredom_threshold):
72              return "bored and hungry"
73          else:
74              return "happy"
75
76      def feed(self):
77          Pet.feed(self)
78          print("Arf! Thanks!")
79
80  class Bird(Pet):
81      sounds = ["chirp"]
82      def __init__(self, name="Kitty", chirp_number=2):
83          Pet.__init__(self, name) # call the parent class's constructor
84          # basically, call the SUPER -- the parent version -- of the constructor,
            with all the parameters that it needs.
85          self.chirp_number = chirp_number # now, also assign the new instance variable
86
87      def hi(self):
88          for i in range(self.chirp_number):
89              print(self.sounds[randrange(len(self.sounds))])
90          self.update_boredom()
91
92  class Lab(Dog):
93      def fetch(self):
94          return "I found the tennis ball!"
95
96      def hi(self):
97          print(self.fetch())
98          print(self.sounds[randrange(len(self.sounds))])
99
100 class Poodle(Dog):
101     def dance(self):
102         return "Dancin' in circles like poodles do."
103
104     def hi(self):
105         print(self.dance())
106         Dog.hi(self)
107
108 def whichone(petlist, name):
109     for pet in petlist:
110         if pet.name == name:
111             return pet
112     return None # no pet matched
113
114 pet_types = {'dog': Dog, 'lab': Lab, 'poodle': Poodle, 'cat': Cat, 'bird': Bird}
115 def whichtype(adopt_type="general pet"):
116     return pet_types.get(adopt_type.lower(), Pet)
117
118 def play():
119     animals = []
120
121     option = ""
122     base_prompt = """
123         Quit
124         Adopt <petname_with_no_spaces> <pet_type - choose dog, cat, lab, poodle,
            bird, or another unknown pet type>
125         Greet <petname>
126         Teach <petname> <word>
127         Feed <petname>
128
129         Choice: """
130     feedback = ""
131     while True:
132         action = input(feedback + "\n" + base_prompt)
133         feedback = ""
134         words = action.split()
135         if len(words) > 0:
136             command = words[0]
```

```
137             else:
138                 command = None
139             if command == "Quit":
140                 print("Exiting...")
141                 return
142             elif command == "Adopt" and len(words) > 1:
143                 if whichone(animals, words[1]):
144                     feedback += "You already have a pet with that name\n"
145                 else:
146                     # figure out which class it should be
147                     if len(words) > 2:
148                         Cl = whichtype(words[2])
149                     else:
150                         Cl = Pet
151                     # Make an instance of that class and append it
152                     animals.append(Cl(words[1]))
153             elif command == "Greet" and len(words) > 1:
154                 pet = whichone(animals, words[1])
155                 if not pet:
156                     feedback += "I didn't recognize that pet name. Please try again.\n"
157                     print()
158                 else:
159                     pet.hi()
160             elif command == "Teach" and len(words) > 2:
161                 pet = whichone(animals, words[1])
162                 if not pet:
163                     feedback += "I didn't recognize that pet name. Please try again."
164                 else:
165                     pet.teach(words[2])
166             elif command == "Feed" and len(words) > 1:
167                 pet = whichone(animals, words[1])
168                 if not pet:
169                     feedback += "I didn't recognize that pet name. Please try again."
170                 else:
171                     pet.feed()
172             else:
173                 feedback+= "I didn't understand that. Please try again."
174 
175             for pet in animals:
176                 pet.clock_tick()
177                 feedback += "\n" + pet.__str__()
178 
179     play()
180 
```