Today, we explore another powerful tool that comes with object-oriented programming — inheritance.

Suppose we want to write `Dog` and `Cat` classes. Here’s our first attempt:

```python
class Dog(object):
    def __init__(self, name, owner, color):
        self.name = name
        self.owner = owner
        self.color = color
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says woof!"

class Cat(object):
    def __init__(self, name, owner, lives=9):
        self.name = name
        self.owner = owner
        self.lives = lives
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        return self.name + " says meow!"
```

1 Inheritance
Notice that both the `Dog` and `Cat` classes have a `name`, `owner`, `eat` method, and `talk` method. That’s a lot of effort for so much repeated code!

This is where inheritance comes in. In Python, a class can inherit the instance variables and methods of a another class without having to type them all out again. For example:

```python
class Foo(object):
    # This is the superclass

class Bar(Foo):
    # This is the subclass
```

`Bar` inherits from `Foo`. We call `Foo` the superclass (the class that is being inherited) and `Bar` the subclass (the class that does the inheriting).

Notice that `Foo` also inherits from class, the `object` class. In Python, `object` is the top-level superclass — everything inherits from it, whether directly or through other superclasses. `object` provides basic functionality that is needed for other classes to work with Python.

### 1.1 When should we use inheritance?

One common use of inheritance is to represent a hierarchical relationship between two or more classes — one class is a more specific version of the other class. For example, dogs are a specific type of pet, and a pet is a specific type of animal.

Using inheritance, here is a second attempt at representing `Dog`

```python
class Pet(object):
    def __init__(self, name, owner):
        self.is_alive = True    # It’s alive!!!
        self.name = name
        self.owner = owner
    def eat(self, thing):
        print(self.name + " ate a " + str(thing) + "!")
    def talk(self):
        print('...')

class Dog(Pet):
    def __init__(self, name, owner, color):
        Pet.__init__(self, name, owner)
        self.color = color
    def talk(self):
        print('woof!')
```
Notice that, by using inheritance, we did not have to redefine `self.name`, `self.owner`, or the `eat` method. We did, however, redefine the `talk` method in the `Dog` class. In this case, we want `Dogs` to `talk` differently, so we `override` the method.

The line `Pet.__init__(self, name, owner)` in the `Dog` class is necessary for inheriting the instance attributes `self.is_alive`, `self.name`, and `self.owner`. Without this line, `Dog` will never inherit those instance attributes. Notice that when we call `Pet.__init__`, we need to pass in `self`, since `Pet` is a class, not an instance.

1.2 Questions

1. Implement the `Cat` class by inheriting from the `Pet` class. Make sure to use superclass methods wherever possible. In addition, add a `lose_life` method to the `Cat` class.

```python
class Cat(Pet):
    def __init__(self, name, owner, lives=9):
        ...

def talk(self):
    """A cat says meow! when asked to talk.""

def lose_life(self):
    """A cat can only lose a life if they have at least one life. When lives reach zero, the 'is_alive' variable becomes False."
    """
```
2. Assume these commands are entered in order. What would Python output?

```python
>>> class Foo(object):
...     def __init__(self, a):
...         self.a = a
...     def garply(self):
...         return self.baz(self.a)
>>> class Bar(Foo):
...     a = 1
...     def baz(self, val):
...         return val

>>> f = Foo(4)
>>> b = Bar(3)
>>> f.a

>>> b.a

>>> f.garply()

>>> b.garply()

>>> b.a = 9
>>> b.garply()

>>> f.baz = lambda val: val * val
>>> f.garply()
```
1.3 Extra Questions

1. More Cats!

    class NoisyCat(Cat):
        """A class that behaves just like a Cat, but always
        repeats things twice.
        """
        def __init__(self, name, owner, lives=9):
            # Is this method necessary? Why or why not?

        def talk(self):
            """A NoisyCat will always repeat what he/she said
twice.
            """

In computer science, an interface is a shared set of attributes, along with a specification of the attributes’ behavior. For example, an interface for vehicles might consist of the following methods:

- def drive(self): Drives the vehicle if it has stopped.
- def stop(self): Stops the vehicle if it is driving.

Data types can implement the same interface in different ways. For example, a Car class and a Train can both implement the interface described above, but the Car probably has a different mechanism for drive than the Train.

The power of interfaces is that other programs don’t have to know how each data type implements the interface — only that they have implemented the interface. The following travel function can work with both Cars and Trains:

    def travel(vehicle):
        while not at_destination():
            vehicle.drive()
            vehicle.stop()
2.1 Interfaces in Python

Python defines many interfaces that can be implemented by user-defined classes. For example, the interface for arithmetic consists of the following methods:

- def \_\_add\_\_(self, other): Allows objects to do self + other.
- def \_\_sub\_\_(self, other): Allows objects to do self - other.
- def \_\_mul\_\_(self, other): Allows objects to do self * other.

In addition, there is also an interface for sequences:

- def \_\_len\_\_(self): Allows objects to do len(self).
- def \_\_getitem\_\_(self, index): Allows objects to do self[i].

2.2 Questions

Let’s implement a Vector class that support basic operations on vectors. These include adding and subtracting vectors of the same length, multiplying a vector with a scalar, and taking the dot product of two vectors. The results of these operations are shown in the table below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Vector([1, 2, 3])</td>
<td>Vector([-1, -2, -3])</td>
</tr>
<tr>
<td>Vector([1, 2, 3]) + Vector([4, 5, 6])</td>
<td>Vector([5, 7, 9])</td>
</tr>
<tr>
<td>Vector([4, 5, 6]) - Vector([1, 2, 3])</td>
<td>Vector([3, 3, 3])</td>
</tr>
<tr>
<td>Vector([1, 2, 3]) * Vector([1, 2, 3])</td>
<td>14</td>
</tr>
<tr>
<td>Vector([1, 2, 3]) * 4</td>
<td>Vector([4, 8, 12])</td>
</tr>
<tr>
<td>10 * Vector([1, 2, 3])</td>
<td>Vector([10, 20, 30])</td>
</tr>
<tr>
<td>len(Vector([1, 2, 3]))</td>
<td>3</td>
</tr>
<tr>
<td>Vector([1, 2, 3])[1]</td>
<td>2</td>
</tr>
</tbody>
</table>

We begin with an implementation of the Vector class:

```python
class Vector:
    def \_\_init\_\_(self, vector):
        self.vector = vector

    def \_\_neg\_\_(self):
        return Vector([-self.vector[i] for i in range(len(self.vector))])

    def \_\_add\_\_(self, other):
        return Vector([self.vector[i] + other.vector[i] for i in range(len(self.vector))])

    def \_\_sub\_\_(self, other):
        return self + (-other)

    def \_\_mul\_\_(self, other):
        return Vector([self.vector[i] * other.vector[i] for i in range(len(self.vector))])

    def \_\_rmul\_\_(self, other):
        return self * other

    def \_\_len\_\_(self):
        return len(self.vector)

    def \_\_getitem\_\_(self, n):
        return self.vector[n]
```
1. Implement `__neg__`, which returns a new `Vector` that is the negation of the current vector, `self`. Try using list comprehensions.

```python
def __neg__(self):
    return Vector(____________________________________________
```

2. Implement `__add__`, which takes in two vectors of the same length and returns a new vector which is their sum. Try using list comprehensions.

```python
def __add__(self, other):
    assert type(other) == Vector, "Invalid operation!"
    assert len(self) == len(other), "Invalid dimensions!"
```

3. Implement `__mul__`, which takes in a value, and performs a scalar product if the value is a number, or a vector product if the value is another vector.

```python
def __mul__(self, other):
    if type(other) == int or type(other) == float:
        "*** YOUR CODE HERE ***"
    elif type(other) == Vector:
        "*** YOUR CODE HERE ***"
```
2.3 Extra Questions

1. Now that we have a definition of a vector and its basic operations using type dispatching, we can write more complex expressions using Python’s operator syntax.

\[
\text{Length}(v) = ||v|| = \sqrt{v \cdot v} \\
\text{Norm}(v) = \frac{v}{||v||} \\
\text{Proj}(u, v) = \frac{u \cdot v}{v \cdot v}
\]

Now write these vector functions using the Python operators we’ve just defined. Notice how much cleaner this is compared to using function calls.

```python
from math import sqrt
def vector_length(v):
    return __________________________

def normalize(v):
    return __________________________

def proj(u, v):
    return __________________________
```