Object Oriented Programming

Alessia Mondolo

July 11, 2019

Academy AI

What Is Object-Oriented Programming (OOP)?

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

Another common programming paradigm is **procedural programming** which structures a program like a recipe in that it provides a set of steps, in the form of functions and code blocks, which flow sequentially in order to complete a task.

NOTE: Since Python is a multi-paradigm programming language, you can choose the paradigm that best suits the problem at hand, mix different paradigms in one program, and/or switch from one paradigm to another as your program evolves.

Classes in Python I

The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things (the cost of something, the name of a poem, and your favorite colors, etc.). How can we represent something much more complicated?

Example: let's say you wanted to track a number of different animals. If you used a list, the first element could be the animal's name while the second element could represent its age.

Problems: How would you know which element is supposed to be which? What if you had 100 different animals? Are you certain each animal has both a name and an age, and so forth? What if you wanted to add other properties to these animals?

Classes in Python II

Classes are used to create new **user-defined data structures** that contain arbitrary information about something. In the case of an animal, we could create an `Animal()` class to track properties about the Animal like the name and age.

Important note: a class just provides structure — it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. The `Animal()` class may specify that the name and age are necessary for defining an animal, but it will not actually state what a specific animal's name or age is.

Hint: think of a class as an *idea* for how something should be defined.

Python Objects (Instances)

While the class is the blueprint, an instance is a copy of the class with actual values, literally an object belonging to a specific class.

How to visualize it:

- a class is like a form or questionnaire: it defines the needed information.
- an instance, or object, is the filled form: after you fill out the form, your specific copy is an instance of the class, which contains actual information relevant to you.

You can fill out multiple copies to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, we must first specify what is needed by defining a class.

How To Define a Class in Python

This is how you define a class in Python:

```
class Dog:
    pass
```

You start with the `class` keyword to indicate that you are creating a class, then you add the name of the class (using CamelCase notation).

Also, we used the Python keyword pass here. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.

Instance attributes I

All classes create objects, and all objects contain characteristics called attributes. Use the `__init__()` method to initialize an object's initial attributes by giving them their default value. This method must have at least one argument as well as the `self` variable, which refers to the object itself (e.g., Dog).

```
class Dog:

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age
```

Instance attributes II

NOTE: You will never have to call the `__init__()` method; it gets called automatically when you create a new 'Dog' instance.

Remember: the class is just for defining the Dog, not actually creating instances of individual dogs with specific names and ages; we'll get to that shortly.

The `self` variable is also an instance of the class: since not all dogs share the same name, we need to be able to assign different values to different instances. Hence the need for the special self variable, which will help to keep track of individual instances of each class.

Class attributes

While instance attributes are specific to each object, class attributes are the same for all instances — which in this case is all dogs.

```
class Dog:

    # Class Attribute
    species ='mammal'

    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age
```

So while each dog has a unique name and age, every dog will be a mammal.

Instantiating Objects

Instantiating Objects I

Instantiating is a fancy term for creating a new, unique instance of a class.

For example:

```
>>>class Dog:
... pass
...
>>>Dog()
<__main__.Dog object at 0x1004ccc50>
>>>Dog()
<__main__.Dog object at 0x1004ccc90>
>>>a =Dog()
>>>b =Dog()
>>>a ==b
False
```

Instantiating Objects II

What do you think the type of a class instance is?

```
>>>class Dog:
... pass
...
>>>a =Dog()
```

Instantiating Objects II

What do you think the type of a class instance is?

```
>>>class Dog:
... pass
...
>>>a =Dog()
>>>type(a)
<class '__main__.Dog'>
```

Instantiating Objects III

Let's look at a slightly more complex example:

```
class Dog:
    # Class Attribute
    species ='mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age

# Instantiate the Dog object
philo =Dog("Philo", 5)
mikey =Dog("Mikey", 6)
# Access the instance attributes
print("{} is {} and {} is {}.".format(
    philo.name, philo.age, mikey.name, mikey.age))
# Is Philo a mammal?
if philo.species =="mammal":
    print("{0} is a {1}!".format(philo.name, philo.species))
```

Instantiating Objects IV

Save this as `dog_class.py`, then run the program. You should see:

```
Philo is 5 and Mikey is 6.
Philo is a mammal!
```

We created a new instance of the `Dog()` class and assigned it to the variable `philo`. We then passed it two arguments, "Philo" and 5, which represent that dog's name and age, respectively.

These attributes are passed to the `__init__` method, which gets called any time you create a new instance, attaching the name and age to the object. You might be wondering why we didn't have to pass in the self argument.

This is Python magic; when you create a new instance of the class, Python automatically determines what self is (a Dog in this case) and passes it to the `__init__` method.

Exercise I: The oldest dog

Using the same Dog class, instantiate three new dogs, each with a different age. Then write a function called, `get_biggest_number()`, that takes any number of ages (*args) and returns the oldest one. Then output the age of the oldest dog like so:

```
The oldest dog is 7 years old.
```

***args** allows you to pass a variable number of arguments to a function:

```
def test_var_args(*args):
    for arg in args:
        print "another arg through *args :", arg
test_var_args('yasoob','python','eggs','test')
```

Output:

```
another arg through *args :python
another arg through *args :eggs
another arg through *args :test
```

Solution: The oldest dog

```
class Dog:
    # Class Attribute
    species ='mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age

# Instantiate the Dog object
jake =Dog("Jake", 7)
doug =Dog("Doug", 4)
william =Dog("William", 5)

# Determine the oldest dog
def get_biggest_number(*args):
    return max(args)

# Output
print("The oldest dog is {} years old.".format(
    get_biggest_number(jake.age, doug.age, william.age)))
```

Instance Methods

Instance Methods

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the `__init__` method, the first argument is always self.

Instance Methods

```
class Dog:
    # Class Attribute
    species ='mammal'
    # Initializer / Instance Attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age

    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)

    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
mikey =Dog("Mikey", 6)
# call our instance methods
print(mikey.description())
print(mikey.speak("Gruff Gruff"))
```

Modifying Attributes

You can change the value of attributes based on some behavior:

```
>>>class Email:
... def __init__(self):
... self.is_sent =False
... def send_email(self):
... self.is_sent =True
...
>>>my_email =Email()
>>>my_email.is_sent
False
>>>my_email.send_email()
>>>my_email.is_sent
True
```

Here, we added a method to send an email, which updates the `is_sent` variable to `True`.

Python Object Inheritance

Python Object Inheritance

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

It's important to note that child classes **override or extend** the functionality (e.g., attributes and behaviors) of parent classes. In other words, child classes inherit all of the parent's attributes and behaviors but can also specify different behavior to follow. The most basic type of class is an **object**, which generally all other classes inherit as their parent.

When you define a new class, Python 3 it implicitly uses object as the parent class. So the following two definitions are equivalent:

```
class Dog(object):
    pass

# In Python 3, this is the same as:
class Dog:
    pass
```

Dog park example

Let's pretend that we're at a dog park. How can we differentiate one dog from another? How about the dog's breed:

```
>>>class Dog:
... def __init__(self, breed):
... self.breed =breed
...
>>>spencer =Dog("German Shepard")
>>>spencer.breed
'German Shepard'
>>>sara =Dog("Boston Terrier")
>>>sara.breed
'Boston Terrier'
```

Each breed of dog has slightly different behaviors. To take these into account, let's create separate classes for each breed. These are child classes of the parent Dog class.

Extending the Functionality of a Parent Class

```
# Parent class
class Dog:
    # Class attribute
    species ='mammal'
    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age
    # instance method
    def description(self):
        return "{} is {} years old".format(self.name, self.age)
    # instance method
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)

# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

Extending the Functionality of a Parent Class

```
# Child classes inherit attributes and
# behaviors from the parent class
jim =Bulldog("Jim", 12)
print(jim.description())

# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))
```

Read the comments aloud as you work through this program to help you understand what's happening, then before you run the program, see if you can predict the expected output.

You should see:

```
Jim is 12 years old
Jim runs slowly
```

Parent vs. Child Classes I

The `isinstance()` function is used to determine if an instance is also an instance of a certain parent class:

```
class Dog():
    ...

# Child class (inherits from Dog() class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)

# Child class (inherits from Dog() class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

Parent vs. Child Classes II

```
# Child classes inherit attributes and
# behaviors from the parent class
jim =Bulldog("Jim", 12)
print(jim.description())
# Child classes have specific attributes
# and behaviors as well
print(jim.run("slowly"))

# Is jim an instance of Dog()?
print(isinstance(jim, Dog))

# Is julie an instance of Dog()?
julie =Dog("Julie", 100)
print(isinstance(julie, Dog))

# Is johnny walker an instance of Bulldog()
johnnywalker =RussellTerrier("Johnny Walker", 4)
print(isinstance(johnnywalker, Bulldog))

# Is julie and instance of jim?
print(isinstance(julie, jim))
```

Parent vs. Child Classes III

Output:

```
('Jim', 12)
Jim runs slowly
True
True
False
Traceback (most recent call last):
  File "dog_isinstance.py", line 50, in <module>
    print(isinstance(julie, jim))
TypeError: isinstance() arg 2 must be a class, type, or tuple
                                        of classes and types
```

Make sense? Both `jim` and `julie` are instances of the `Dog()` class, while `johnnywalker` is not an instance of the `Bulldog()` class. Then as a sanity check, we tested if `julie` is an instance of `jim`, which is impossible since `jim` is an instance of a class rather than a class itself — hence the reason for the `TypeError`.

Overriding the Functionality of a Parent Class

Remember that child classes can also **override** attributes and behaviors from the parent class. For examples:

```
>>>class Dog:
... species ='mammal'
>>>class SomeBreed(Dog):
... pass
>>>class SomeOtherBreed(Dog):
... species ='reptile'
>>>frank =SomeBreed()
>>>frank.species
'mammal'
>>>beans =SomeOtherBreed()
>>>beans.species
'reptile'
```

The `SomeBreed()` class inherits the species from the parent class, while the `SomeOtherBreed()` class overrides the species, setting it to reptile.

Exercise II: Dog Inheritance

Create a Pets class that holds instances of dogs; this class is completely separate from the Dog class. In other words, the Dog class does not inherit from the Pets class. Then assign three dog instances to an instance of the Pets class. Start with the following code below. Save the file as `pets_class.py`. Your output should look like this:

```
I have 3 dogs.
Tom is 6.
Fletcher is 7.
Larry is 9.
And they're all mammals, of course.
```

Exercise II: Starter Code

```
# Parent class
class Dog:
    # Class attribute
    species ='mammal'
    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age
    # instance methods
    def description(self):
        return "{} is {} years old".format(self.name, self.age)
    def speak(self, sound):
        return "{} says {}".format(self.name, sound)
# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "{} runs {}".format(self.name, speed)
```

Exercise II: Solution

```
# Parent class
class Pets:
    dogs =[]
    def __init__(self, dogs):
        self.dogs =dogs
# Parent class

class Dog:
    # Class attribute
    species ='mammal'
    # Initializer / Instance attributes
    def __init__(self, name, age):
        self.name =name
        self.age =age
    # Instance methods
    def description(self):
        return self.name, self.age
    def speak(self, sound):
        return "\%s says \%s" \% (self.name, sound)
```

Exercise II: Solution

```
# Child class (inherits from Dog class)
class RussellTerrier(Dog):
    def run(self, speed):
        return "\%s runs \%s" \% (self.name, speed)
# Child class (inherits from Dog class)
class Bulldog(Dog):
    def run(self, speed):
        return "\%s runs \%s" \% (self.name, speed)
# Create instances of dogs
my_dogs =[
    Bulldog("Tom", 6),
    RussellTerrier("Fletcher", 7),
    Dog("Larry", 9)
]
# Instantiate the Pets class
my_pets =Pets(my_dogs)
# Output
print("I have {} dogs.".format(len(my_pets.dogs)))
for dog in my_pets.dogs:
    print("{} is {}.".format(dog.name, dog.age))
print("And they're all {}s, of course.".format(dog.species))
```

Exercise III: Hungry Dogs

Using the same file, add an instance attribute of `is_hungry = True` to the Dog class. Then add a method called eat() which changes the value of `is_hungry` to False when called. Figure out the best way to feed each dog and then output "My dogs are hungry." if all are hungry or "My dogs are not hungry." if all are not hungry. The final output should look like this:

```
I have 3 dogs.
Tom is 6.
Fletcher is 7.
Larry is 9.
And they're all mammals, of course.
My dogs are not hungry.
```

Exercise IV: Dog Walking

Next, add a walk() method to both the Pets and Dog classes so that when you call the method on the Pets class, each dog instance assigned to the Pets class will walk(). Save this as `dog_walking.py`. This is slightly more difficult.

Start by implementing the method in the same manner as the speak() method. As for the method in the Pets class, you will need to iterate through the list of dogs, then call the method itself.

The output should look like this:

```
Tom is walking!
Fletcher is walking!
Larry is walking!
```

Exercise IV: Dog Walking

Answer the following questions about OOP to check your learning progress:

- What's a class?
- What's an instance?
- What's the relationship between a class and an instance?
- What's the Python syntax used for defining a new class?
- What's the spelling convention for a class name?
- How do you instantiate, or create an instance of, a class?
- How do you access the attributes and behaviors of a class instance?
- What's a method?
- What's the purpose of self?
- What's the purpose of the `__init__` method?
- Describe how inheritance helps prevent code duplication.
- Can child classes override properties of their parents?

Other resources

The Official Python Tutorial: Classes:

`http://docs.python.org/3/tutorial/classes.html`

Python-Patterns Repository on GitHub:

`https://github.com/faif/python-patterns`

Python Design Patterns:

`https://www.toptal.com/python/python-design-patterns`

Python 3 Object-Oriented Programming (400 pages book)