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*.

For instance, an object could represent a person with a name property, age, address, etc., with behaviors like walking, talking, breathing, and running. Or an email with properties like recipient list, subject, body, etc., and behaviors like adding attachments and sending.

Put another way, object-oriented programming is an approach for modeling concrete, real-world things like cars as well as relations between things like companies and employees, students and teachers, etc. OOP models real-world entities as software objects, which have some data associated with them and can perform certain functions.

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.

The key takeaway is that objects are at the center of the object-oriented programming paradigm, not only representing the data, as in procedural programming, but in the overall structure of the program as well.

NOTE: REMEMBER AMAR SIR'S CLASS EXAMPLE THE TRIANGLE

Classes in Python

Focusing first on the data, each thing or object is an instance of some *class*.

The primitive data structures available in Python, like numbers, strings, and lists are designed to represent simple things like the cost of something, the name of a poem, and your favorite colors, respectively.

What if you wanted to represent something much more complicated?

For 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.

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? This lacks organization, and it's the exact need for *classes*.

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.

It's important to note that 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.

It may help to 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. It's not an idea anymore; it's an actual animal, like a dog named Roger who's eight years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class; it 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

Defining a class is simple 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, starting with a capital letter.)

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

All classes create objects, and all objects contain characteristics called attributes (referred to as properties in the opening paragraph). Use the `__init__()` method to initialize (e.g., specify) an object's initial attributes by giving them their default value (or state). 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
```

In the case of our `Dog()` class, each dog has a specific name and age, which is obviously important to know for when you start actually creating different dogs. 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.

Similarly, the `self` variable is also an instance of the class. Since instances of a class have varying values we could state `Dog.name = name` rather than `self.name = name`. But 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.

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

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 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
```

We started by defining a new `Dog()` class, then created two new dogs, each assigned to different objects. So, to create an instance of a class, you use the the class name, followed by parentheses. Then to demonstrate that each instance is actually different, we instantiated two more dogs, assigning each to a variable, then tested if those variables are equal.

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

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

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))
```

NOTE: Notice how we use dot notation to access attributes from each object.

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

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


What's Going On?

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.

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`:

```
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"))
```

Save this as *dog_instance_methods.py*, then run it:

```
Mikey is 6 years old
Mikey says Gruff Gruff
```

In the latter method, `speak()`, we are defining behavior. What other behaviors could you assign to a dog? Look back to the beginning paragraph to see some example behaviors for other objects.

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

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:
    pass
```

Dog Park Example

Let's pretend that we're at a dog park. There are multiple Dog objects engaging in Dog behaviors, each with different attributes. In regular-speak that means some dogs are running, while some are stretching and some are just watching other dogs. Furthermore, each dog has been named by its owner and, since each dog is living and breathing, each ages.

What's another way to 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

Create a new file called *dog_inheritance.py*:

```
# 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 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)


# 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
```

We haven't added any special attributes or methods to differentiate a `RussellTerrier` from a `Bulldog`, but since they're now two different classes, we could for instance give them different class attributes defining their respective speeds.

Parent vs. Child Classes

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

Save this as *dog_isinstance.py*:

```
# 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 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)


# 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))
```


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`.