Object-Oriented Programming (OOP) in Python 3

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviors into individual **objects**. In this tutorial, you'll learn the basics of object-oriented programming in Python.

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line, a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or preprocessed materials at each step on an assembly line. In addition, the object contains behavior, like the action that each assembly line component performs.

In this tutorial, you'll learn how to:

- Define a **class**, which is like a blueprint for creating an object
- Use classes to **create new objects**
- Model systems with **class inheritance**

What Is Object-Oriented Programming in Python?

Object-oriented programming is a programming paradigm that provides a means of structuring programs so that properties and behaviors are bundled into individual **objects**.

For example, an object could represent a person with **properties** like a name, age, and address and **behaviors** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body 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 or students and teachers. OOP models real-world entities as software objects that have some data associated with them and can perform certain operations.

The key takeaway is that objects are at the center of object-oriented programming in Python. In other programming paradigms, objects only represent the data. In OOP, they additionally inform the overall structure of the program.

How Do You Define a Class in Python?

In Python, you define a class by using the `class` keyword followed by a name and a colon. Then you use `.__init__()` to declare which attributes each instance of the class should have:

Python

But what does all of that mean? And why do you even need classes in the first place? Take a step back and consider using built-in, primitive data structures as an alternative.

Primitive data structures—like numbers, strings, and lists—are designed to represent straightforward pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, you might want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

Python

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where you declared the `kirk` list, will you remember that the element with index `0` is the employee's name?

Second, it can introduce errors if employees don't have the same number of elements in their respective lists. In the `mccoy` list above, the age is missing, so `mccoy[1]` will return `"Chief Medical Officer"` instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

Classes vs Instances

Classes allow you to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this tutorial, you'll create a `Dog` class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how to define something. It doesn't actually contain any data. The `Dog` class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an **instance** is an object that's built from a class and contains real data. An instance of the `Dog` class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that you've filled out with information. Just like many people can fill out the same form with their own unique information, you can create many instances from a single class.

Class Definition

You start all class definitions with the `class` keyword, then add the name of the class and a colon. Python will consider any code that you indent below the class definition as part of the class's body.

Here's an example of a `Dog` class:

Python

The body of the `Dog` class consists of a single statement: the `pass` keyword. Python programmers often use `pass` as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note: Python class names are written in CapitalizedWords notation by convention. For example, a class for a specific breed of dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so you'll spruce it up a bit by defining some properties that all `Dog` objects should have. There are several properties that you can choose from, including name, age, coat color, and breed. To keep the example small in scope, you'll just use name and age.

You define the properties that all `Dog` objects must have in a method called `.__init__()`. Every time you create a new `Dog` object, `.__init__()` sets the initial **state** of the object by assigning the values of the object's properties. That is, `.__init__()` initializes each new instance of the class.

You can give `.__init__()` any number of parameters, but the first parameter will always be a variable called `self`. When you create a new class instance, then Python automatically passes the instance to the `self` parameter in `.__init__()` so that Python can define the new **attributes** on the object.

Update the `Dog` class with an `.__init__()` method that creates `.name` and `.age` attributes:

Python

Make sure that you indent the `.__init__()` method's signature by four spaces, and the body of the method by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the `Dog` class.

In the body of `.__init__()`, there are two statements using the `self` variable:

1. **`self.name = name`** creates an attribute called `name` and assigns the value of the `name` parameter to it.
2. **`self.age = age`** creates an attribute called `age` and assigns the value of the `age` parameter to it.

Attributes created in `.__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All `Dog` objects have a name and an age, but the values for the `name` and `age` attributes will vary depending on the `Dog` instance.

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following `Dog` class has a class attribute called `species` with the value `"Canis familiaris"`:

Python

```
# dog.py

class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

You define class attributes directly beneath the first line of the class name and indent them by four spaces. You always need to assign them an initial value. When you create an instance of the class, then Python automatically creates and assigns class attributes to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that you have a `Dog` class, it's time to create some dogs!

How Do You Instantiate a Class in Python?

Creating a new object from a class is called **instantiating** a class. You can create a new object by typing the name of the class, followed by opening and closing parentheses:

Python

You first create a new `Dog` class with no attributes or methods, and then you instantiate the `Dog` class to create a `Dog` object.

In the output above, you can see that you now have a new `Dog` object at `0x106702d30`. This funny-looking string of letters and numbers is a **memory address** that indicates where Python stores the `Dog` object in your computer's memory. Note that the address on your screen will be different.

Now instantiate the `Dog` class a second time to create another `Dog` object:

Python

The new `Dog` instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first `Dog` object that you created.

To see this another way, type the following:

Python

In this code, you create two new `Dog` objects and assign them to the variables `a` and `b`. When you compare `a` and `b` using the `==` operator, the result is `False`. Even though `a` and `b` are both instances of the `Dog` class, they represent two distinct objects in memory.

Class and Instance Attributes

Now create a new `Dog` class with a class attribute called `.species` and two instance attributes called `.name` and `.age`:

Python

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
```

To instantiate this `Dog` class, you need to provide values for `name` and `age`. If you don't, then Python raises a `TypeError`:

Python

```
>>> Dog()
Traceback (most recent call last):
  ...
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

To pass arguments to the `name` and `age` parameters, put values into the parentheses after the class name:

Python

This creates two new `Dog` instances—one for a four-year-old dog named Miles and one for a nine-year-old dog named Buddy.

The `Dog` class's `.__init__()` method has three parameters, so why are you only passing two arguments to it in the example?

When you instantiate the `Dog` class, Python creates a new instance of `Dog` and passes it to the first parameter of `.__init__()`. This essentially removes the `self` parameter, so you only need to worry about the `name` and `age` parameters.

Note: Behind the scenes, Python both creates and initializes a new object when you use this syntax. If you want to dive deeper, then you can read the dedicated tutorial about the Python class constructor.

After you create the `Dog` instances, you can access their instance attributes using **dot notation**:

Python

You can access class attributes the same way:

Python

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All `Dog` instances have `.species`, `.name`, and `.age` attributes, so you can use those attributes with confidence, knowing that they'll always return a value.

Although the attributes are guaranteed to exist, their values *can* change dynamically:

Python

In this example, you change the `.age` attribute of the `buddy` object to `10`. Then you change the `.species` attribute of the `miles` object to `"Felis silvestris"`, which is a species of cat. That makes Miles a pretty strange dog, but it's valid Python!

The key takeaway here is that custom objects are mutable by default. An object is mutable if you can alter it dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

Instance Methods

Instance methods are functions that you define inside a class and can only call on an instance of that class. Just like `.__init__()`, an instance method always takes `self` as its first parameter.

Open a new editor window in IDLE and type in the following `Dog` class:

Python

```
# dog.py

class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

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

This `Dog` class has two instance methods:

1. **`.description()`** returns a string displaying the name and age of the dog.
2. **`.speak()`** has one parameter called `sound` and returns a string containing the dog's name and the sound that the dog makes.

Save the modified `Dog` class to a file called `dog.py` and press F5 to run the program. Then open the interactive window and type the following to see your instance methods in action:

Python

```
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

In the above `Dog` class, `.description()` returns a string containing information about the `Dog` instance `miles`. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class. However, `.description()` isn't the most Pythonic way of doing this.

When you create a `list` object, you can use `print()` to display a string that looks like the list:

Python

Go ahead and print the `miles` object to see what output you get:

Python

When you print `miles`, you get a cryptic-looking message telling you that `miles` is a `Dog` object at the memory address `0x00aeff70`. This message isn't very helpful. You can change what gets printed by defining a special instance method called `.__str__()`.

In the editor window, change the name of the `Dog` class's `.description()` method to `.__str__()`:

Python

```
# dog.py

class Dog:
    # ...

    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Save the file and press F5. Now, when you print `miles`, you get a much friendlier output:

Python

Methods like `.__init__()` and `.__str__()` are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Understanding dunder methods is an important part of mastering object-oriented programming in Python, but for your first exploration of the topic, you'll stick with these two dunder methods.

Note: Check out When Should You Use `.__repr__()` vs `.__str__()` in Python? to learn more about `.__str__()` and its cousin `.__repr__()`.

If you want to reinforce your understanding with a practical exercise, then you can click on the block below and work on solving the challenge:

When you're done with your own implementation of the challenge, then you can expand the block below to see a possible solution:

When you're ready, you can move on to the next section. There, you'll see how to take your knowledge one step further and create classes from other classes.

How Do You Inherit From Another Class in Python?

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 you derive child classes from are called **parent classes**.

You inherit from a parent class by creating a new class and putting the name of the parent class into parentheses:

Python

In this minimal example, the child class `Child` inherits from the parent class `Parent`. Because child classes take on the attributes and methods of parent classes, `Child.hair_color` is also `"brown"` without your explicitly defining that.

Note: This tutorial is adapted from the chapter "Object-Oriented Programming (OOP)" in *Python Basics: A Practical Introduction to Python 3*. If you enjoy what you're reading, then be sure to check out the rest of the book and the learning path.

You can also check out the Python Basics: Building Systems With Classes video course to reinforce the skills that you'll develop in this section of the tutorial.

Child classes can override or extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent's attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn't perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your parents. It's an attribute that you were born with. But maybe you decide to color your hair purple. Assuming that your parents don't have purple hair, you've just **overridden** the hair color attribute that you inherited from your parents:

Python

If you change the code example like this, then `Child.hair_color` will be `"purple"`.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you'll also speak English. Now imagine you decide to learn a second language, like German. In this case, you've **extended** your attributes because you've added an attribute that your parents don't have:

Python

```
# inheritance.py

class Parent:
    speaks = ["English"]

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.speaks.append("German")
```

You'll learn more about how the code above works in the sections below. But before you dive deeper into inheritance in Python, you'll take a walk to a dog park to better understand why you might want to use inheritance in your own code.

Example: Dog Park

Pretend for a moment that you're at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The `Dog` class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the `Dog` class in the editor window by adding a `.breed` attribute:

Python

```
# dog.py

class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Press F5 to save the file. Now you can model the dog park by creating a bunch of different dogs in the interactive window:

Python

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like *woof*, but dachshunds have a higher-pitched bark that sounds more like *yap*.

Using just the `Dog` class, you must supply a string for the `sound` argument of `.speak()` every time you call it on a `Dog` instance:

Python

Passing a string to every call to `.speak()` is repetitive and inconvenient. Moreover, the `.breed` attribute should determine the string representing the sound that each `Dog` instance makes, but here you have to manually pass the correct string to `.speak()` every time you call it.

You can simplify the experience of working with the `Dog` class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for `.speak()`.

Parent Classes vs Child Classes

In this section, you'll create a child class for each of the three breeds mentioned above: Jack Russell terrier, dachshund, and bulldog.

For reference, here's the full definition of the `Dog` class that you're currently working with:

Python

```
# dog.py

class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

After doing the dog park example in the previous section, you've removed `.breed` again. You'll now write code to keep track of a dog's breed using child classes instead.

To create a child class, you create a new class with its own name and then put the name of the parent class in parentheses. Add the following to the `dog.py` file to create three new child classes of the `Dog` class:

Python

Press F5 to save and run the file. With the child classes defined, you can now create some dogs of specific breeds in the interactive window:

Python

Instances of child classes inherit all of the attributes and methods of the parent class:

Python

```
>>> miles.species
'Canis familiaris'

>>> buddy.name
'Buddy'

>>> print(jack)
Jack is 3 years old

>>> jim.speak("Woof")
'Jim says Woof'
```

To determine which class a given object belongs to, you can use the built-in `type()`:

Python

What if you want to determine if `miles` is also an instance of the `Dog` class? You can do this with the built-in `isinstance()`:

Python

Notice that `isinstance()` takes two arguments, an object and a class. In the example above, `isinstance()` checks if `miles` is an instance of the `Dog` class and returns `True`.

The `miles`, `buddy`, `jack`, and `jim` objects are all `Dog` instances, but `miles` isn't a `Bulldog` instance, and `jack` isn't a `Dachshund` instance:

Python

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, you can give each breed its own sound.

Parent Class Functionality Extension

Since different breeds of dogs have slightly different barks, you want to provide a default value for the `sound` argument of their respective `.speak()` methods. To do this, you need to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, you define a method with the same name on the child class. Here's what that looks like for the `JackRussellTerrier` class:

Python

```
# dog.py

# ...

class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"

# ...
```

Now `.speak()` is defined on the `JackRussellTerrier` class with the default argument for `sound` set to `"Arf"`.

Update `dog.py` with the new `JackRussellTerrier` class and press F5 to save and run the file. You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to `sound`:

Python

Sometimes dogs make different noises, so if Miles gets angry and growls, you can still call `.speak()` with a different sound:

Python

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, in the editor window, change the string returned by `.speak()` in the `Dog` class:

Python

Save the file and press F5. Now, when you create a new `Bulldog` instance named `jim`, `jim.speak()` returns the new string:

Python

However, calling `.speak()` on a `JackRussellTerrier` instance won't show the new style of output:

Python

Sometimes it makes sense to completely override a method from a parent class. But in this case, you don't want the `JackRussellTerrier` class to lose any changes that you might make to the formatting of the `Dog.speak()` output string.

To do this, you still need to define a `.speak()` method on the child `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the `Dog` class's `.speak()` from *inside* the child class's `.speak()` using the same arguments that you passed to `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using `super()`:

Python

```
# dog.py

# ...

class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)

# ...
```

When you call `super().speak(sound)` inside `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable `sound`.

Update `dog.py` with the new `JackRussellTerrier` class. Save the file and press F5 so you can test it in the interactive window:

Python

Now when you call `miles.speak()`, you'll see output reflecting the new formatting in the `Dog` class.

Note: In the above examples, the **class hierarchy** is very straightforward. The `JackRussellTerrier` class has a single parent class, `Dog`. In real-world examples, the class hierarchy can get quite complicated.

The `super()` function does much more than just search the parent class for a method or an attribute. It traverses the entire class hierarchy for a matching method or attribute. If you aren't careful, `super()` can have surprising results.

If you want to check your understanding of the concepts that you learned about in this section with a practical exercise, then you can click on the block below and work on solving the challenge:

When you're done with your own implementation of the challenge, then you can expand the block below to see a possible solution:

Nice work! In this section, you've learned how to override and extend methods from a parent class, and you worked on a small practical example to cement your new skills.

Conclusion

In this tutorial, you learned about object-oriented programming (OOP) in Python. Most modern programming languages, such as Java, C#, and C++, follow OOP principles, so the knowledge that you gained here will be applicable no matter where your programming career takes you.

In this tutorial, you learned how to:

- Define a **class**, which is a sort of blueprint for an object
- Instantiate a class to create an **object**
- Use **attributes** and **methods** to define the **properties** and **behaviors** of an object
- Use **inheritance** to create **child classes** from a **parent class**
- Reference a method on a parent class using `super()`
- Check if an object inherits from another class using `isinstance()`