# Python
# Object-Oriented Programming (OOP)

**OOP**, or **O**bject-**O**riented **P**rogramming, is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

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

An object contains data, like the raw or pre-processed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

**In this module, you will learn how to:**

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

Let's get started!

# 1 Define a Class

Primitive data structures—like numbers, strings, and lists—are designed to represent simple things, such as the cost of something, the name of a poem, and your favorite colors, respectively. What if you want to represent something much more complicated?

For example, let's say you wanted 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:

```
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, when you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the 0th element of the list is the employee's name? What if not every employee has the same number of elements in the list?

Second, 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 are used to create user-defined data structures that contain data pertaining to some object. Often, classes also contain special function, called **methods**, that define behaviors and actions that an object can perform with its data.

In this chapter you'll create a `Dog` class that stores some basic information about a dog.

It's important to note that a class just provides structure. A class is a blueprint for how something should be defined. It doesn't actually provide any real content itself. The `Dog` class may specify that the name and age are necessary for defining a dog, but it will not actually state what a specific dog's name or age is.

While the class is the blueprint, an **instance** is an object built from a class that 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. 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 of a form 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, you must first specify what is needed by defining a class.

## How to Define a Class

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon. This is similar to the signature of a function, except that you don't need to add any parameters in parentheses. Any code that is indented below the class definition is considered part of the class's body.

Here is an example of a simple `Dog` class:

```
class Dog:
    pass
```

The body of the `Dog` class consists of a single statement: the `pass` keyword. `pass` is often used as a place holder where code will eventually go. It allows you to run this code without throwing an error.

**Note**

Unlike functions and variables, the convention for naming classes in Python is to use CamelCase notation, starting with a capital letter. For example, a class for a specific breed of a dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all `Dog` objects should have. There are a number of properties that we can choose from, such as name, age, coat color, and breed. To keep things simple, we'll stick with just two for now: name and age.

To define the properties, or **instance attributes**, that all `Dog` objects must have, you need to define a special method called `.__init__()`. This method is run every time a new `Dog` object is created and tells Python what the initial **state**—that is, the initial values of the object's properties—of the object should be.

The first positional argument of `.__init__()` is always a variable that references the class instance. This variable is almost universally named `self`. After the `self` argument, you can specify any other arguments required to create an instance of the class.

The following updated definition of the `Dog` class shows how to write an `.__init__()` method that creates two instance attributes: `.name` and `.age`:

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

Notice that the function signature—the part that starts with the `def` keyword—is indented four spaces. The body of the function is indented by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the `Dog` class.

Without the indentation, Python would treat `__init__()` as just another function.

> **Note**
>
> Functions that belong to a class are called **instance methods** because they belong to the instance of a class. For example, `list.append()` and `string.find()` are instance methods.

In the body of the `.__init__()` method, there are two statements using the `self` variable. The first line, `self.name = name`, creates a class attribute called `name` and assigns to it the value of the `name` variable that was passed to the `.__init__()` method. The second line creates an `age` attribute and assigns to it the value of the `age` argument.

This might look kind of strange. The `self` variable is referring to an instance of the `Dog` class, but we haven't actually created an instance yet. It is a place holder that is used to build the blueprint. Remember, the class is used to define the `Dog` data structure. It does not actually create any instances of individual dogs with specific names and ages.

While instance attributes are specific to each object, **class attributes** are the same for all instances—which in this case is all dogs. In the next example, a class attribute called `species` is created and assigned the value `"Canis familiaris"`:

```
class Dog:
    # Class Attribute
    species = "Canis familiaris"

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

Class attributes are defined directly underneath the first line of the class and outside of any method definition. They must be assigned a value because they are created on a class instance without arguments to determine what their initial value should be.

You should use class attributes whenever a property should have the same initial value for all instances of a class. Use instance attributes for properties that must be specified before an instance is created.

Now that we have a `Dog` class, let's create some dogs!

## 2 Instantiate an Object

Once a class has been defined, you have a blueprint for creating—also known as **instantiating**—new objects. To instantiate an object, you simple type the name of the class, in the original CamelCase, followed by parentheses containing any values that must be passed to the class's `.__init__()` method.

Let's take a look at an actual example. Open IDLE's interactive window and type the following:

```
>>> class Dog:
...     pass
...
```

This creates a new `Dog` class with no attributes and methods.

Next, instantiate a new `Dog` object:

```
>>> Dog()
<__main__.Dog object at 0x106702d30>
```

The output indicates that you now have a new `Dog` object at memory address `0x106702d30`. Note that the address you see on your screen will very likely be different from the address shown here.

Now let's instantiate another `Dog` object:

```
>>> Dog()
<__main__.Dog object at 0x0004ccc90>
```

The new `Dog` instance is located at a different memory address. This is because it is an entirely new instance, completely unique from the first `Dog` object you instantiated.

To see this another way, type the following:

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

Two new `Dog` objects are created and assigned to the variables `a` and `b`. When `a` and `b` are compared using the `==` operator, the result is `False`. For user defined classes, the default behavior of the `==` operator is to compare the memory addresses of two objects and return `True` if the address is the same and `False` otherwise.

What this means is that even though the `a` and `b` object are both instances of the `Dog` class and have the exact same attributes and methods—namely, no attributes or methods, in this case—`a` and `b` represent two distinct objects in memory.

You can use the `type()` function to determine an object's class:

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

Of course, even though both `a` and `b` are distinct `Dog` instances, they have the same type:

```
>>> type(a) == type(b)
True
```

## Class and Instance Attributes

Let's look at a slightly more complex example using the `Dog` class we defined with `.name` and `.age` instance attributes:

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

After declaring the new `Dog` class, two new instances are created—one `Dog` whose name is Buddy and is nine years old, and another named Miles who is four years old.

Does anything look a little strange about how the `Dog` objects are instantiated? The `.__init__()` method takes three arguments, so why are only two arguments specified in the example instead of three?

When you instantiate a `Dog` object, Python creates a new instance and passes it to the first argument of `.__init__()`. This happens for you behind the scenes, so you don't have to worry about it.

You can access instance attributes by using **dot notation**:

```
>>> buddy.name
'Buddy'
>>> buddy.age
9

>>> miles.name
'Miles'
>>> miles.age
4
```

Class attributes are accessed the same way:

```
>>> buddy.species
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect:

```
>>> buddy.species == miles.species
True
```

Both `buddy` and `miles` have the `.species` attribute. Contrast this to the method of using lists to represent similar data structures that you saw at the beginning of the previous section. With a class you no longer have to worry that an attribute may be missing.

Both instance and class attributes can be modified dynamically:

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

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

The important takeaway here is that custom objects are mutable by default. Recall that an object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are not—they are immutable.

Now that you know the difference between a class and an instance, how to create instances and set class and instance attributes, the next step is to look at instance methods in more detail.

## Instance Methods

Instance methods are functions defined inside of a class. This means that they only exist within the context of the object itself and cannot be called without referencing the object. Just like `.__init__()`, the first argument of an instance method is always `self`:

```
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}"
```

In this example, two new instance methods are defined: `.description()` and `.speak()`. The `.description()` method returns a string displaying the name and age of the dog, and `.speak()` takes one argument called `sound` and returns a string containing the dog's name and the sound the dog makes.

Let's see how instance methods work in practice. To avoid typing out the whole class in the interactive window, you can save the modified `Dog` class in a script in IDLE and run it. Then open the interactive window and type the following to see instance methods in action:

```
>>> 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'
```

The `.description()` method defined in the above `Dog` class returns a string containing information about the `Dog` instance `miles`. When writing your own classes, it is 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 the `print()` function to display a string that looks like the list:

```
>>> names = ["Fletcher", "David", "Dan"]
>>> print(names)
['Fletcher', 'David', 'Dan']
```

Let's see what happens if we try and `print()` the `miles` object:

```
>>> print(miles)
<__main__.Dog object at 0x00aeff70>
```

When you `print(miles)`, you get a somewhat cryptic looking message telling you that `miles` is a `Dog` object located at some memory address.

You can specify what should be printed by defining a special instance method called `.__str__()` on the `Dog` class. Let's change `.description()` to `.__str__()` in the `Dog` class:

```python
class Dog:
    # Class attributes and other methods ommitted...
    # Replace description with __str__
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Now when you `print(miles)` you get much friendlier output:

```python
>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

**Note**

Methods like `.__str__()` are commonly called **dunder methods** because they begin and end with double underscores. There are a number of dunder methods available that allow your classes to work well with other Python language features.

You should now have a pretty good idea of how to create a class that stores some data and provides some methods to interact with that data and define behaviors for an object.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes. But first, check your understanding with the following review exercises.

## Review Exercises

1. Modify the `Dog` class to include a third instance attribute called `coat_color` that stores the color of the dog's coat as a string. Store your new class in a script and test it out by adding the following code at the bottom of the script:

```
philo = Dog("Philo", 5, "brown")
print(f"{philo.name}'s coat is {philo.coat_color}.")
```

The output of your script should be:

```
Philo's coat is brown.
```

2. Create a `Car` class with two instance attributes: `.color`, which stores the name of the car's color as a string, and `.mileage`, which stores the number of miles on the car as an integer. Then instantiate two `Car` object—a blue car with 20,000 miles, and a red car with 30,000 miles, and print out their colors and mileage. Your output should look like the following:

```
The blue car has 20,000 miles.
The red car has 30,000 miles.
```

3. Modify the `Car` class with an instance method called `.drive()` that takes a number as an argument and adds that number to the `.mileage` attribute. Test that your solution works by instantiating a car with 0 miles, then call `.drive(100)` and print the `.mileage` attribute to check that it is set to `100`.

# 3 Inherit From Other Classes

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**.

Child classes can override and 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 different attributes and methods that are unique to themselves, or even redefine methods from their parent class.

The concept of object inheritance can be thought of sort of like genetic inheritance, even though the analogy isn't perfect.

For example, you may have inherited your hair color from your mother. It's an attribute you were born with. You may decide that you want to color you hair purple. Assuming your mother doesn't have purple hair, you have just **overridden** the hair color attribute you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you will also speak English. One day, you may decide to learn a second language, like German. In this case you are **extending** attributes, because you have added an attribute that your parents do not have.

## The `object` Class

The most basic type of class is an `object`, which generally all other classes inherit from as their parent. When you define a new class, Python 3 implicitly uses `object` as the parent class, so the following two definitions are equivalent:

```python
class Dog(object):
    pass

# In Python 3, this is the same as:

class Dog:
    pass
```

The inheritance from `object` is stated explicitly in the first definition by putting `object` in between parentheses after the `Dog` class name. This is the same pattern used to create child classes from your own custom classes.

> **Note**
>
> In Python 2 there's a distinction between new-style and old-style classes. We won't cover this distinction, because it doesn't apply to Python 3. Just know that in Python 3, there is an `object` class that all classes inherit from, even though you don't have to explicitly state that in your code.

Let's see how and why you might create child classes from a parent class.

## Dog Park Example

Pretend for a moment that you are 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 you wrote in the previous section can distinguish dogs by name and age, but not by breed.

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

```
class Dog:
    species = "Canis familiaris"

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

The instance methods defined earlier are omitted here because they aren't important for this discussion.

Now, to model the dog park, you could instantiate a bunch of different dogs:

```
>>> miles = Dog("Miles", 4, "Jack Russell Terrier")
>>> buddy = Dog("Buddy", 9, "Dachshund")
```

```
>>> jack = Dog("Jack", 3, "Bulldog")
>>> jim = Dog("Jim", 5, "Bulldog")
```

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 the `.speak()` method every time you call it on a `Dog` instance:

```
>>> buddy.speak("Yap")
'Buddy says Yap'

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

>>> jack.speak("Woof")
'Jack says Woof'
```

Passing a string to every call to `.speak()` is repetitive and inconvenient. What’s worse, the string representing the sound each `Dog` instance makes depends on the `.breed` attribute, but there is nothing stopping you, or someone using the `Dog` class you have created, from passing any string they wish.

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 each child class inherits, including specifying a default argument for `.speak()`.

## Parent Classes vs Child Classes

Let’s create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here is the full definition of the `Dog` class:

```
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}"
```

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. The following creates three new child classes of the `Dog` class:

```
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

With the child classes defined, you can now instantiate some dogs of specific breeds:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> buddy = Dachshund("Buddy", 9)
>>> jack = Bulldog("Jack", 3)
>>> jim = Bulldog("Jim", 5)
```

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

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

>>> buddy.name
'Buddy'

>>> print(jack)
Jack is 4 years old

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

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

```
>>> type(miles)
<class '__main__.JackRussellTerrier'>
```

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

```
>>> isinstance(miles, Dog)
True
```

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`.

All of the `miles`, `buddy`, `jack` and `jim` objects are instances of the `Dog` class, but `miles` is not an instance of the `Bulldog` class, and `jack` is not an instance of the `Dachshund` class:

```
>>> isinstance(miles, Bulldog)
False

>>> isinstance(jack, Dachshund)
False
```

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 got some child classes created for some different breeds of dogs, let's give each breed its own sound.

## Extending the Functionality of a Parent Class

At this point, we have four classes floating around: a parent class—`Dog`—and three child classes—`JackRussellTerrier`, `Dachshund` and `Bulldog`. All three child classes inherit every attribute and method from the parent class, including the `.speak()` method.

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

Let's see what this looks like for the `JackRussellTerrier` class:

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

The `.speak()` method is now defined on the `JackRussellTerrier` class with the default argument for `sound` set to `"Arf"`. Now you can call `.speak()` and a `JackRussellTerrier` instance without passing an argument to `sound`:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

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

```
>>> miles.speak("Grrr")
'Miles says Grrr'
```

One advantage of class inheritance is that changes to the parent class will automatically propagate to their child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, let's say you decide to change the string returned by `.speak()` in the `Dog` class:

```
class Dog:
    # Other attributes and methods omitted...

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

Now, when you create a new `Bulldog` instance named `jim`, the result of `jim.speak("Woof")` will be `'Jim barks: Woof'` instead of `'Jim says Woof'`:

```
>>> jim = Bulldog("Jim", 5)
>>> jim.speak("Woof")
'Jim barks: Woof'
```

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

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

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

To do this, you still need to define a `.speak()` method on the `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the `Dog` class's `.speak()` method *inside* of the child class's `.speak()` method and make sure to pass to it the whatever is passed to `sound` argument of `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using the `super()` function. Here's how you could re-write the `JackRussellTerrier.speak()` method using `super()`:

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

When you call `super().speak(sound)` inside of `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable `sound`. Now, when you call `miles.speak()`, you will see output reflecting the new formatting in the `Dog` class:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles barks: Arf'
```

**Important**

In the above examples, the **class hierarchy** is very simple: the `JackRussellTerrier` class has a single parent class—`Dog`.

In many real world examples, the class hierarchy can get quite complicated with one class inheriting from a parent class, which inherits from another parent class, which inherits from another parent class, and so on.

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.

In this section, you learned how to make new classes from existing classes utilizing an OOP concept called **inheritance**. You saw how to check if an object is an instance of a class or parent class using the `isinstance()` function. Finally, you learned how to extend the functionality of a parent class by using `super()`.

## Review Exercises

1. Create a `GoldenRetriever` class that inherits from the `Dog` class. Give the `sound` argument of the `GoldenRetriever.speak()` method a default value of `"Bark"`. Use the following code for your parent `Dog` class:

```python
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}"
```

2. Write a `Rectangle` class that must be instantiated with two attributed: `length` and `width`. Add a `.area()` method to the class that returns the area (`length * width`) of the rectangle. Then write a `Square` class that inherits from the `Rectangle` class and that is instantiated with a single attribute called `side_length`. Test your `Square` class by instantiating a `Square` with a `side_length` of `4`. Calling the `.area()` method should return `16`.