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, and behavior, like the action each assembly line component performs.

In this tutorial, you'll learn how to:

- Create 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 instance, 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, students and teachers, and so on. OOP models real-world entities as software objects that 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, that flow sequentially in order to complete a task.

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

Define a Class in Python

Primitive data structures—like numbers, strings, and lists—are designed to represent simple 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, let's say you 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:

In [ ]:
```
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 the kirk list is declared, will you remember that the element with index 0 is the employee's name?

Second, it can introduce errors if not every employee has the same number of elements in the list. 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. 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 something should be defined. 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 is 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 has been filled out with information. Just like many people can fill out the same form with their own unique information, many instances can be created from a single 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. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a Dog class:

In [1]:
```
class Dog:
    pass
```

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

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, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

The properties that all Dog objects must have are defined in a method called `.__init__()`. Every time a new Dog object is created, `.__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 a new class instance is created, the instance is automatically passed to the self parameter in `.__init__()` so that new attributes can be defined on the object.

Let's update the Dog class with an `.__init__()` method that creates `.name` and `.age` attributes:

In [2]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Notice that the `.__init__()` method's signature is indented four spaces. The body of the method is indented 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 to it the value of the name parameter.
2. `self.age = age` creates an attribute called age and assigns to it the value of the age parameter.

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":

In [3]:
```
class Dog:
    # Class attribute
    species = "Canis familiaris"

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

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned 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 we have a Dog class, let's create some dogs!

Instantiate an Object in Python

Try the following:

In [4]:
```
class Dog:
    pass
```

This creates a new Dog class with no attributes or methods.

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

In [5]:
```
Dog()
```

Out[5]:
```
<__main__.Dog at 0x105bac890>
```

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 instantiated.

To see this another way, try the following:

In [6]:
```
a = Dog()
```

In [7]:
```
b = Dog()
```

In [8]:
```
a == b
```

Out[8]:
```
False
```

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.

In [9]:
```
a
```

Out[9]:
```
<__main__.Dog at 0x105bc6d90>
```

In [10]:
```
b
```

Out[10]:
```
<__main__.Dog at 0x105bc6910>
```

Class and Instance Attributes

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

In [11]:
```
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

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

In [12]:
```
Dog()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[12], line 1
----> 1 Dog()

TypeError: Dog.__init__() missing 2 required positional arguments: 'name' an
d 'age'
```

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

In [13]:
```
buddy = Dog("Buddy", 9)
miles = Dog("Miles", 4)
```

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

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

When you instantiate a Dog object, Python creates a new instance 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.

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

In [14]:
```
buddy.name
```

Out[14]:
```
'Buddy'
```

In [15]:
```
buddy.age
```

Out[15]:
```
9
```

```
Out[16]: 'Miles'
```

```
In [17]: miles.age
```

```
Out[17]: 4
```

You can access class attributes the same way:

```
In [18]: buddy.species
```

```
Out[18]: 'Canis familiaris'
```

```
In [19]: miles.species
```

```
Out[19]: 'Canis familiaris'
```

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 will always return a value.

Although the attributes are guaranteed to exist, their values can be changed dynamically:

```
In [20]: buddy.age = 10
```

```
In [21]: buddy.age
```

```
Out[21]: 10
```

```
In [22]: miles.species = "Felis silvestris"
```

```
In [23]: miles.species
```

```
Out[23]: 'Felis silvestris'
```

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 is valid Python!

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

Instance Methods

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like `.__init__()`, an instance method's first

parameter is always self.

Consider the following Dog class:

In [24]:
```
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 the dog makes.

Check the following to see your instance methods in action:

In [25]:
```
miles = Dog("Miles", 4)
miles.description()
```

Out[25]:
```
'Miles is 4 years old'
```

In [26]:
```
miles.speak("Woof Woof")
```

Out[26]:
```
'Miles says Woof Woof'
```

In [27]:
```
miles.speak("Bow Wow")
```

Out[27]:
```
'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:

In [28]:
```
names = ["Fletcher", "David", "Dan"]
print(names)
```

```
['Fletcher', 'David', 'Dan']
```

Let's see what happens when you `print()` the miles object:

In [29]:
```
print(miles)
```

```
<__main__.Dog object at 0x107a977d0>
```

When you `print(miles)`, you get a cryptic looking message telling you that miles is a Dog object at the memory address like **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__()`:

In [30]:
```
class Dog:
    species = "Canis familiaris"

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

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

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

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

In [31]:
```
miles = Dog("Miles", 4)
print(miles)
```

```
Miles is 4 years old
```

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. Although too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering object-oriented programming in Python.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes.

Your turn!

It is time to check your understanign, so try to complete the following exercise.

Inherit From Other Classes 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 child classes are derived from are called parent classes.

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 mother. It's an attribute you were born with. Let's say you decide to color your hair purple. Assuming your mother doesn't have purple hair, you've just **overridden** the hair color attribute that you inherited from your mom.

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.

Dog Park Example

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:

In [1]:
```
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}"
```

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

Now you can model the dog park by instantiating a bunch of different dogs.

In [2]:
```
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 .speak() every time you call it on a Dog instance:

In [3]:
```
buddy.speak("Yap")
```

Out[3]:
```
'Buddy says Yap'
```

In [4]:
```
jim.speak("Woof")
```

Out[4]:
```
'Jim says Woof'
```

In [5]:
```
jack.speak("Woof")
```

Out[5]:
```
'Jack says Woof'
```

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

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

Let's 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:

In [6]:
```
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. Add the following to the dog.py file to create three new child classes of the Dog class:

In [7]:
```
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:

In [8]:
```
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:

In [9]:
```
miles.species
```

Out[9]:
```
'Canis familiaris'
```

In [10]:
```
buddy.name
```

Out[10]:
```
'Buddy'
```

In [11]:
```
print(jack)
```

```
Jack is 3 years old
```

In [12]:
```
jim.speak("Woof")
```

Out[12]:
```
'Jim says Woof'
```

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

In [13]:
```
type(miles)
```

Out[13]:
```
__main__.JackRussellTerrier
```

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()`:

In [14]:
```
isinstance(miles, Dog)
```

Out[14]:
```
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``.

The `miles`, `buddy`, `jack`, and `jim` objects are all `Dog` instances, but `miles` is not a `Bulldog` instance, and `jack` is not a `Dachshund` instance:

In [15]:
```
isinstance(miles, Bulldog)
```

Out[15]:
```
False
```

In [16]:
```
isinstance(jack, Dachshund)
```

Out[16]:
```
False
```

In [17]:
```
isinstance(jack, Bulldog)
```

Out[17]:
```
True
```

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, let's give each breed its own sound.

Extend the Functionality of a Parent Class

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:

```
Russellterrier(Dog):
    def speak(self, sound="Arf"):
```

```
        return f"{self.name} saysssss {sound}"
```

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

You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to sound:

In [19]:
```
miles = JackRussellTerrier("Miles", 4)
miles.speak()
```

Out[19]:
```
'Miles saysssss Arf'
```

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

In [20]:
```
miles.speak("Grrr")
```

Out[20]:
```
'Miles saysssss Grrr'
```

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:

In [21]:
```
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"

    # Change the string returned by .speak()
    def speak(self, sound):
        return f"{self.name} barks: {sound}"
```

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

In [22]:
```
class Bulldog(Dog):
    pass
```

In [23]:
```
jim = Bulldog("Jim", 5)
'Woof")
```

```
Out[23]: 'Jim barks: Woof'
```

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

```
In [24]: class JackRussellTerrier(Dog):

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

```
In [25]: miles = JackRussellTerrier("Miles", 4)
         miles.speak()
```

```
Out[25]: 'Miles saysssss Arf'
```

Sometimes it makes 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 child `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the Dog class's `.speak()` inside of 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()`:

```
In [26]: 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.

So you can test it:

```
In [27]: miles = JackRussellTerrier("Miles", 4)
         miles.speak()
```

```
Out[27]: 'Miles barks: Arf'
```

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.

> `super()` 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.

Your turn!

Check your understanding, go to the exercise!
