Note

In the Notes, any code will be in blue, input is green and any output in red.

Object-oriented programming (OOP) is a method of structuring a program by bundling related properties and behaviours into individual **objects**.

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 pre-processed materials at each step on an assembly line, and behaviour, like the action each assembly line component performs.

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 behaviours are bundled into individual **objects**.

For instance, an object could represent a person with **properties** like a name, age, and address and **behaviours** such as walking, talking, breathing, and running. Or it could represent an email with properties like a recipient list, subject, and body and behaviours like adding attachments and sending.

Put another way, object-oriented programming is an approach for modelling 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 to complete a task.

The key takeaway is that objects are at the centre 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 favourite colours, 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:

```
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 behaviours and actions that an object created from the class can perform with its data.

Here, you'll create a Dog class that stores some information about the characteristics and behaviours that an individual dog can have.

A class is a blueprint for how something should be defined. It does not actually contain any data. The Dog class specifies that a name and an age are necessary for defining a dog, but it does not 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.

First Create a new file called "Dog.py"

Here’s an example of a 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 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 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 colour, 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:

```
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":

```
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

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:

When we create an instance of a Dog we must ensure that it has all the attributes it needs. In this case we must include a name and an age:

```
Dog("Bob",7)
```

```
print(Dog("Bob",7))
```

```
<__main__.Dog object at 0x000001E765240100>
```

You now have a new Dog object at 0x000001E765240100. This funny-looking string of letters and numbers is a **memory address** that indicates where the Dog object is stored in your computer's memory. Note that the address you see on your screen will be different.

Class and Instance Attributes

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

```
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**:

```
print(buddy.name)

'Buddy'

print(buddy.age)

9

print(miles.name)

'Miles'

print(miles.age)

4
```

You can access class attributes the same way:

```
Print( 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. 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:

```
buddy.age = 10

print(buddy.age)

10

miles.species = "Felis silvestris"

print(miles.species)

'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.

```
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.

```
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:

```
names = ["Fletcher", "David", "Dan"]

print(names)

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

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

```
print(miles)

<__main__.Dog object at 0x00aeff70>
```

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__().

Change the name of the Dog class's .description() method to .__str__():

```
class Dog:

    # Leave other parts of Dog class as-is

    # Replace .description() with __str__()

    def __str__(self):

        return f"{self.name} is {self.age} years old"
```

```
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.