Object Oriented Programming

Object oriented programming is a method of structuring a program by bundling related properties and behaviors into individual objects.

For example an object could represent an email with properties like recipient list, subject, body, etc and behaviors like saving a draft, adding attachments, sending, etc.

OOP is an approach to programming for modeling real world things as well as relation between things.

For example, one object could represent a company and another object could represent an employee. The company object would be made up of employee objects and the employee objects would have an attribute that points to the company object.

Classes

Objects, or instances are created from classes in a similar way a building is created from a blueprint.

Classes are made up of instructions and variables that define the attributes and behaviors of an object. Let's see how to define a class and instantiate objects from it.

In [1]:

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

```
### The .__init__() method is called whenever an object is instantiated. It define the properties, or instance attributes, that all Dog objects must have.
### The self variable refers to the object being created, it is essentially used as a placeholder to refer to a future object.
```

The values of instance attributes change from object to object. You can define class attributes that remain the same for all instances.

Class attributes are defined outside any method definition. They must be assigned a value, because they start the same for all objects and receive no arguments to determine what their initial value should be

In [2]:
```
class Dog:
    species = "Canis lupus familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Let's instantiate two Dog objects

In [13]:
```
paddy = Dog("Paddy", 11)
michael = Dog("Michael", 6)
```

We can see that they have the same type, yet they are not equal. This is because they are 2 distinct objects stored separately in the memory

In [17]:
```
type(paddy) == type(michael)
```

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

In [18]:
```
paddy == michael
```

Out[18]:
```
False
```

You can access the attributes of the Dog object

In [19]:
```
print(paddy.name)
print(paddy.age)
print(paddy.species)
```

```
Paddy
11
Canis lupus familiaris
```

Attributes of an object can be modified

In [20]:
```
paddy.age += 1
print(paddy.age)
```

```
12
```

Instance Methods

Instance methods are functions defined inside a class. They cannot be called without reference the object itself.

In [23]:
```
class Dog:
    species = "Canis lupus familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

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

In [24]:
```
paddy = Dog("Paddy", 11)
michael = Dog("Michael", 6)
```

You can call the objects methods

In [25]:
```
print(paddy.description())
print(paddy.speak("Woof, woof"))
```

```
Paddy is 11 years old
Paddy says Woof, woof
```

Inheritance

Inheritance is the process by which one class (the child class) inherits the attributes and methods of another (the parent class)

Child classes can override and extend the attributes and methods of parent classes.

Let's write some child classes that inherit, override and extend attributes and methods from the Dog class

In [29]:
```
class GoldenRetriever(Dog):
    pass # The pass keyword is often used as a placeholder
```

In [30]:
```
milo = GoldenRetriever("milo", 4)
print(milo.description())
```

```
milo is 4 years old
```

In order to override the .speak() method in the Dog class, we rewrite the method in the child class. here we will be specifiying a default value for the parameter sound

In [35]:
```
class GoldenRetriever(Dog):
    def speak(self, sound = "Woof"):
        return f"{self.name} says {sound}"

milo = GoldenRetriever("milo", 4)
print(milo.speak())
```

```
milo says Woof
```

while the sound parameter has a default value, if we pass an argument for the sound parameter, it will take that on.

In [36]:
```
milo.speak("Grrr")
```

Out[36]:
```
'milo says Grrr'
```

The super() keyword

If we decide to modify the .speak() method in the Dog class, because of the way we overrode the method in the GoldenRetriever class it will not reflect the change. The super() keyword allows child classes to inherit behavior better.

In [39]:
```
class Dog:
    species = "Canis lupus familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

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

class GoldenRetriever(Dog):
    def speak(self, sound = "Woof"):
        return super().speak(sound)

milo = GoldenRetriever("milo", 4)
milo.speak()
```

Out[39]:
```
'milo growls Woof'
```