object-oriented-programming

Defineing a Class

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

- Classes define functions called methods.
- 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.
- a class is like a form or questionnaire. An instance is like a form that you've filled out with information.

Class Definition

- start all class definitions with the class keyword, then add the name of the class and a colon.

```
# dog.py

class Dog:
    pass
```

- 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.

Object Attributes

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

```
# dog.py

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

```
        self.name = name
        self.age = age
```

- This indentation is vitally important. It tells Python that the .**init**() method belongs to the Dog class
- the body of .**init**(), there are two statements using the self variable:
    - self.name = name creates an attribute called name and assigns the value of the name parameter to it.
    - 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(object) attributes. An instance(object) attribute's value is specific to a particular instance(object) of the class.

Class Attributes

- class attributes are attributes that have the same value for all class instances.
- we can define a class attribute by assigning a value to a variable name outside of .**init**().

example

- the following Dog class has a class attribute called species with the value "Canis familiaris"

```
# dog.py

class Dog:
    species = "Canis familiaris"

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

- class attributes directly beneath the first line of the class name and indent them by four spaces(same as def .**init**()).
- you create an instance of the class, then Python automatically creates and assigns class attributes to their initial values.

IMP POINTS

- 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.

instantiating a class

- Creating a new object from a class is called instantiating a class.

Without Class and Instance Attributes

- You can create a new object by typing the name of the class.
- followed by opening and closing parentheses.

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

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

With Class and Instance Attributes

```
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.

```
Dog("Miles", 4)
Dog("Buddy", 9)
```

- 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.

access their instance attributes using dot notation

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

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

access class attributes

sameway

```
buddy.species
```

advantages of using classes to organize data

- 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.

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

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

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

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.

```
# 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}"
```

```
>>> 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'
```

To use print:

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

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

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.

```
# inheritance.py

class Parent:
    hair_color = "brown"

class Child(Parent):
    pass
```

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

```
# overridden
class Child(Parent):
    hair_color = "purple"
```

```
# extended
# inheritance.py

class Parent:
    speaks = ["English"]
```

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