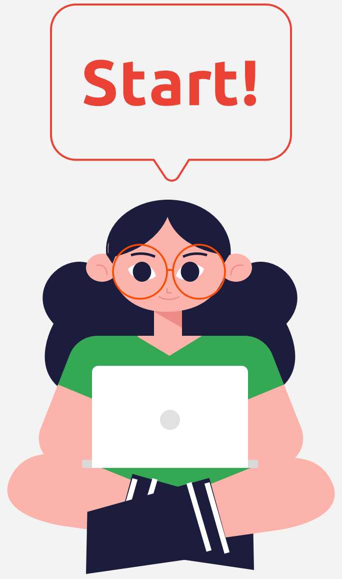
Object Oriented Programming - OOP

GDSC - ASUFE



Who Am I?

Waseem Faheem

- Sophomore Electrical Student
- ICPC trainee
- Embedded Systems and Machine Learning Student

Contact

+20 120 084 8141

Tusk#4236

Session Contents

What is OOP?

We have previously looked at two paradigms of programming -
imperative (using statements, loops, and functions as subroutines),
and **functional** (using pure functions, higher-order functions, and recursion).

Another very popular paradigm is **object-oriented programming** (OOP).
Objects are created using **classes**, which are actually the focal point of OOP.
The **class** describes what the object will be, but is separate from the object itself. In other words, a class can be described as an object's blueprint, description, or definition.

You can use the same class as a blueprint for creating multiple different objects.

What is OOP? - cont.

Objects are defined by three things: Identity, Attributes, and Behavior

A very famous example on OOP is a cookie!

You can make a cookie anywhere, anytime using its recipe.

A recipe is like a blueprint for making cookies (i.e. a class), and a cookie is one of many cookies that are made using that blueprint (i.e. an object).

The **recipe (class)** describes the type (Identity), the shape (Attributes), and the actions (Behaviors) that relate to the cookie (object).

What is OOP? - cont.

So, let's create a cookie!

First, we find the **recipe** of a cookie, not the recipe of a cake or a brownie, as we define the cookie's Identity as being **A** cookie not anything else.

Then, we choose suitable ingredients for the cookie so it turns out in the Shape we want it to be, as we define the cookie's Attributes by its Shape.

Finally, we start acting on the cookie, we mix the dough, sprinkle it, then bake, as we define the Behavior of the cookie by the Actions done on it.

What is OOP? - cont.

In programming, **objects** are independent units, and each has its own **identity**, just as objects in the real world do.

Objects also have **characteristics** that are used to describe them. For example, a car can be red or blue, a mug can be full or empty, and so on. These characteristics are also called **attributes**. An attribute describes the current **state** of an object.

In the real world, each object **behaves** in its own way. The car **moves**, the phone **rings**, and so on. The same applies to objects - behavior is specific to the object's type. Methods are functions used for Objects, and the describe the Object's behavior.

In programming, an object is **self-contained**, with its own **identity**. It is separate from other objects. Each object has its own **attributes**, which describe its current state. Each exhibits its own **behavior**, which demonstrates what they can do.

Constructors and Methods

So how exactly can we write what we just discussed in code?

Classes are created using the keyword **class** and an indented block, which contains class **methods** (which are functions).

```
class Class:
(ind) code
```

Constructors and Methods – cont.

__init__

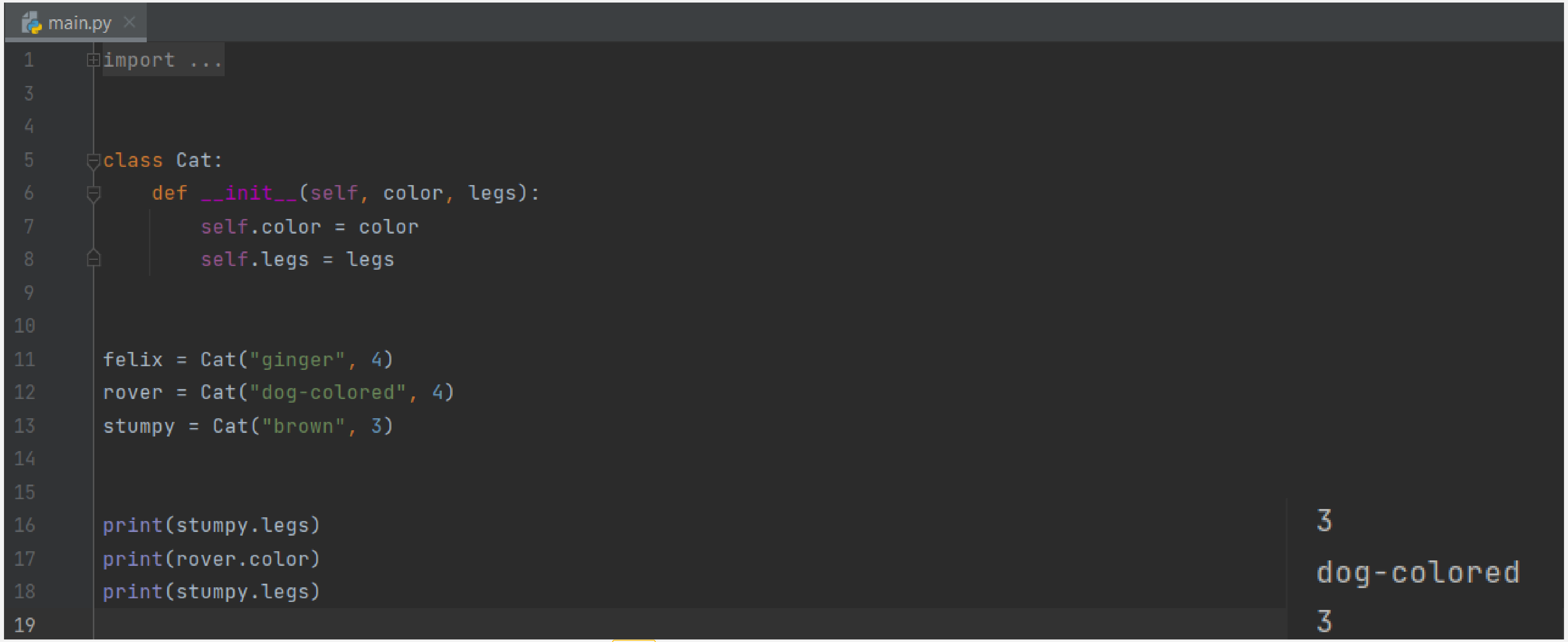
The **__init__** method is the most important method in a class.
This is called when an instance (object) of the class is created, using the class name as a function.

All methods must have **self** as their first parameter, although it isn't explicitly passed, Python adds the **self** argument to the list for you; you do not need to include it when you call the methods. Within a method definition, **self** refers to the instance calling the method.

Instances of a class have **attributes**, which are pieces of data associated with them.
In this example, **Cat** instances have attributes **color** and **legs**. These can be accessed by putting a **dot**, and the attribute name after an instance.
In an **__init__** method, **self.attribute** can therefore be used to set the initial value of an instance's attributes.

Classes can also have **class attributes**, created by assigning variables within the body of the class. These can be accessed either from instances of the class, or the class itself.

Constructors and Methods – cont.



main.py

```
import ...


class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs


felix = Cat("ginger", 4)
rover = Cat("dog-colored", 4)
stumpy = Cat("brown", 3)


print(stumpy.legs)
print(rover.color)
print(stumpy.legs)
```

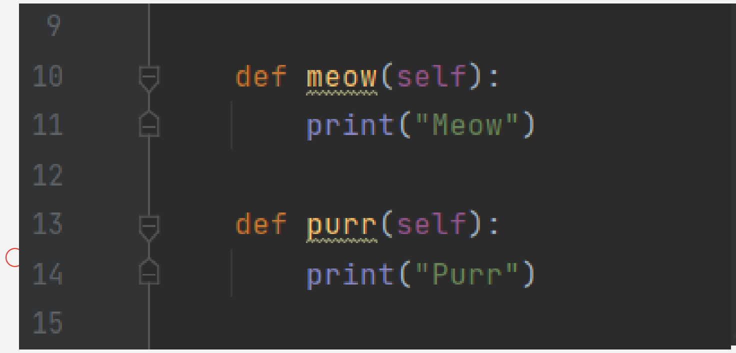
```
3
dog-colored
3
```

Constructors and Methods – cont.

Methods

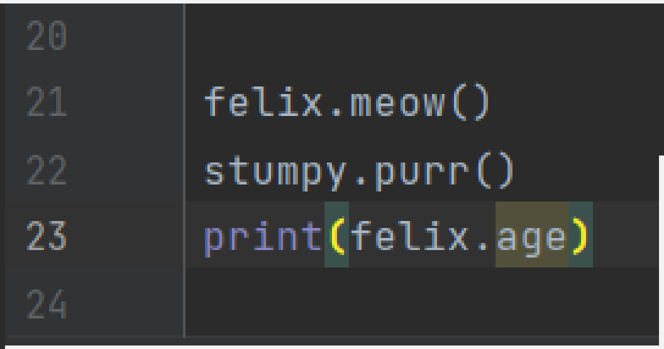
Classes can have other **methods** defined to add functionality to them. Remember, that all methods must have **self** as their first parameter.

These methods are accessed using the same **dot** syntax as attributes. Trying to access an attribute of an instance that isn't defined causes an **AttributeError**. This also applies when you call an undefined method.

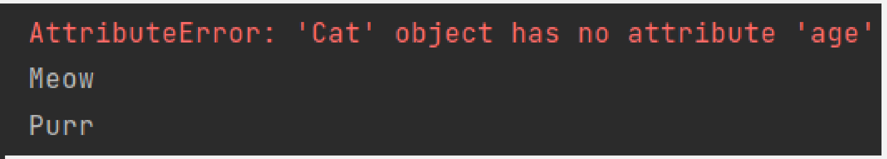


```
    def meow(self):
        print("Meow")

    def purr(self):
        print("Purr")
```



```
felix.meow()
stumpy.purr()
print(felix.age)
```



```
AttributeError: 'Cat' object has no attribute 'age'
Meow
Purr
```

Inheritance, Encapsulation, and Polymorphism

We have already seen the modeling power of OOP using the class and object functions by combining data and methods.

There are three more important concept, **inheritance**, which makes the OOP code more modular, easier to reuse and build a relationship between classes.

Encapsulation can hide some of the private details of a class from other objects.

While **polymorphism** can allow us to use a common operation in different ways. In this section, we will briefly discuss them.

Inheritance, Encapsulation, and Polymorphism – cont.

Inheritance

Let's expand our cookie example from before and talk about the dough. A dough can be baked into a pizza, a cake, or a cookie depending on the ingredients, but there is one main thing that connects them all together, they are all doughs!

So, we can say that a dough(Parent Class) is a blueprint for a cookie dough(Child Class) which is a blueprint for a cookie(Class)

Inheritance, Encapsulation, and Polymorphism – cont.

Another way to look at it is by thinking about animals.
Dogs are the descendent of Wolves, so Dogs **Inherit** some traits from Wolves.

They both have claws, fangs, and wiggly tails, however Wolves cry "AWOO" and Dogs cry "WOOF".

We can say that dogs(Child Class) **inherits** some attributes from wolves(Parent Classes) while **changing (Overriding)** some of their actions (Methods) like crying.

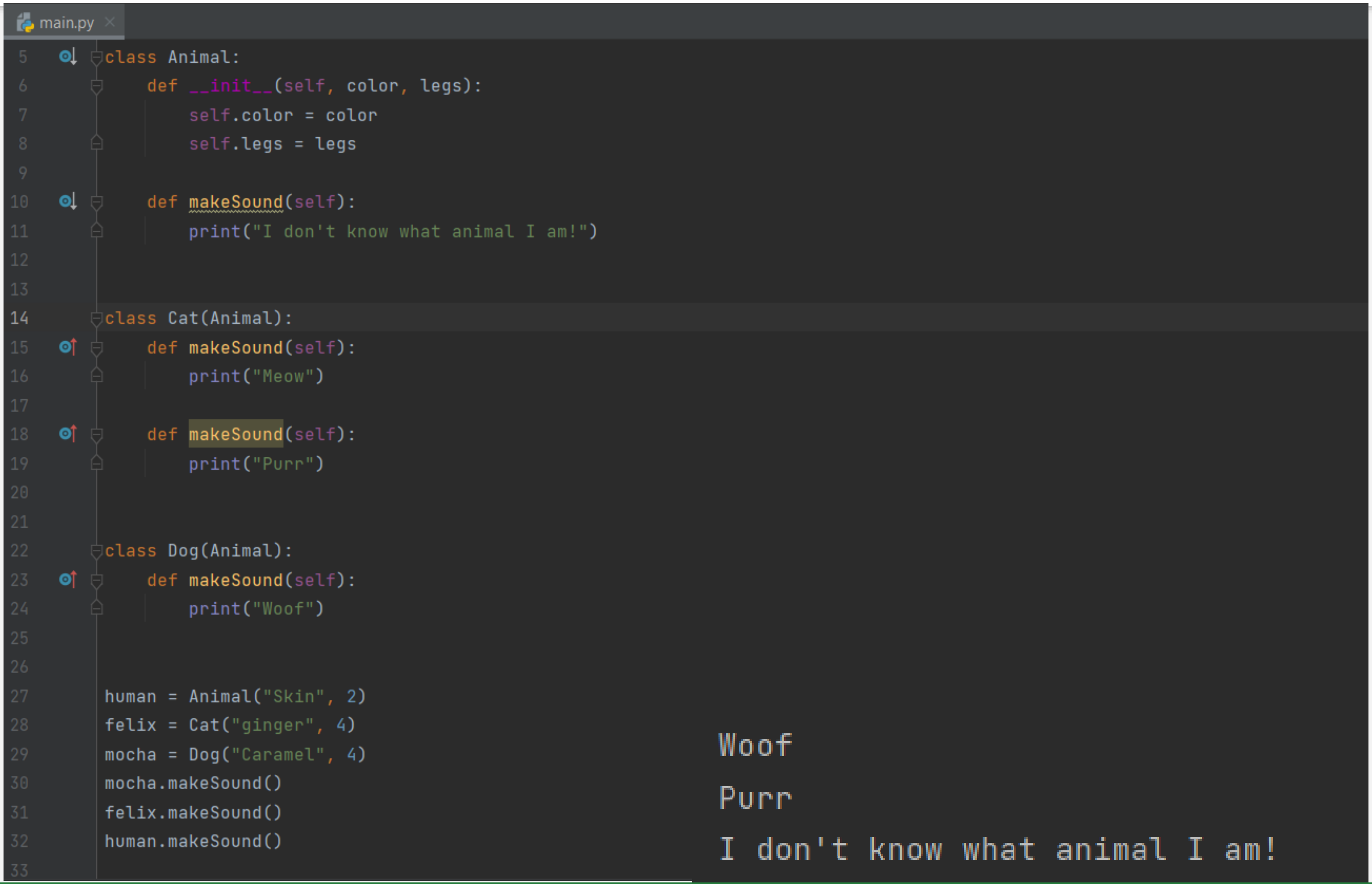
Inheritance, Encapsulation, and Polymorphism – cont.

Inheritance allows us to define a class that inherits all the methods and attributes from another class. Convention denotes the new class as **child class**, and the one that it **inherits** from is called **parent class** or **superclass**.

We can see the structure for basic **inheritance** is **class ClassName(superclass)**.

When we inherit from a parent class, we can change the implementation of a method provided by the parent class, this is called method **overriding**.

Inheritance, Encapsulation, and Polymorphism – cont.



main.py

```python
class Animal:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

    def makeSound(self):
        print("I don't know what animal I am!")


class Cat(Animal):
    def makeSound(self):
        print("Meow")

    def makeSound(self):
        print("Purr")


class Dog(Animal):
    def makeSound(self):
        print("Woof")


human = Animal("Skin", 2)
felix = Cat("ginger", 4)
mocha = Dog("Caramel", 4)
mocha.makeSound()
felix.makeSound()
human.makeSound()
```

```
Woof
Purr
I don't know what animal I am!
```

Inheritance, Encapsulation, and Polymorphism – cont.

Now, you've got your dough, made it into a cookie dough, and baked a lot of cookies, but you don't want your sibling to eat one of these cookies!

So, you decide to keep your cookies in a private jar, so they cannot eat a cookie except if you give them access.

This process is called **encapsulation**.

Inheritance, Encapsulation, and Polymorphism – cont.

Encapsulation is one of the fundamental concepts in OOP.

It describes the idea of restricting access to methods and attributes in a class.

This will hide the complex details from the users, and prevent data being modified by accident.

In Python, this is achieved by using private methods or attributes using underscore as prefix, i.e., single “_” or double “__”. Let us see the following example.

Inheritance, Encapsulation, and Polymorphism – cont.

Polymorphism is another fundamental concept in OOP, which means multiple forms.

Polymorphism allows us to use a single interface with different underlying forms such as data types or classes.

For example, we can have commonly named methods for parent class and child class, they may both have the same method, but they have different implementation.

This ability of using single named method with many forms acting differently in different situations greatly reduces our complexities.

Inheritance, Encapsulation, and Polymorphism – cont.

Another important concept to know is the concept of **abstraction.**

In OOPs, **Abstraction** is the method of getting information where the information needed will be taken in such a simplest way that solely the required components are extracted, and the ones that are considered less significant are unnoticed.

The concept of **abstraction** only shows necessary information to the users.

It reduces the complexity of the program by hiding the implementation complexities of programs.

Inheritance, Encapsulation, and Polymorphism – cont.

The **main difference** between **abstraction** and **inheritance** is that **abstraction** allows hiding the internal details and displaying only the functionality to the users, while **inheritance** allows using properties and methods of an already existing class.

While the **main difference** between **abstraction** and **encapsulation** is that **abstraction** is the process or method of gaining the information while hiding the unwanted information, while **encapsulation** is the process or method to contain the information hiding the data in a single entity or unit along with a method to protect information from outside.

Summary

- OOP and POP are different. OOP has many benefits and is often more appropriate for use in large-scale projects.
- Class is the blueprint of the structure that allows us to group data and methods, while object is an instance from the class.
- The concept of “**inheritance**” is key to OOP, which allows us to refer attributes or methods from the superclass.
- The concept of “**encapsulation**” allows us to hide some of the private details of a class from other objects.
- The concept of “**polymorphism**” allows us to use common operation in different ways for different data input.

My codes, alongside some examples can be found on my GitHub account:

https://github.com/GyroZeppeliLovesBalls/Python_OOP_Session_GDSCASUFE/tree/main

THANK YOU!