10 - Object Oriented Programming

What is OOP?

A programming paradigm that organises a software design using an object-oriented approach.

What is OOP?

A programming paradigm that organises a software design using an object-oriented approach.

It allows us to design software around objects instead of functions and variables.

What is OOP?

A programming paradigm that organises a software design using an object-oriented approach.

It allows us to design software around objects instead of functions and variables.

OOP is suitable for designing large complex systems, representing the **entities** and how they communicate with each other.

Object-Oriented Programming

OOP consists of four principles:

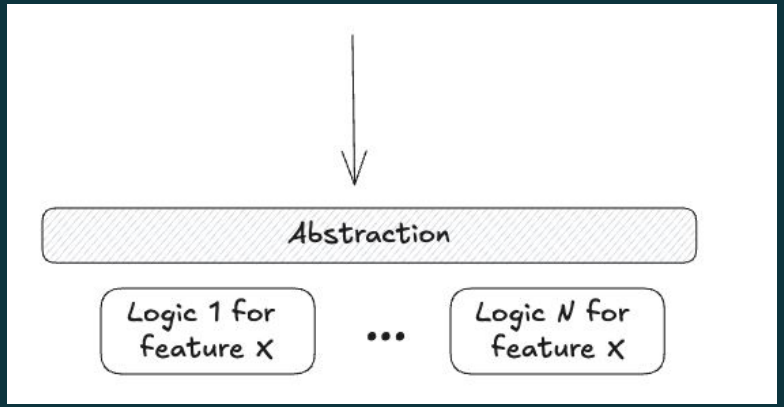
Object-Oriented Programming

OOP consists of four principles:

1. **Abstraction**
2. **Encapsulation**
3. **Inheritance**
4. **Polymorphism**

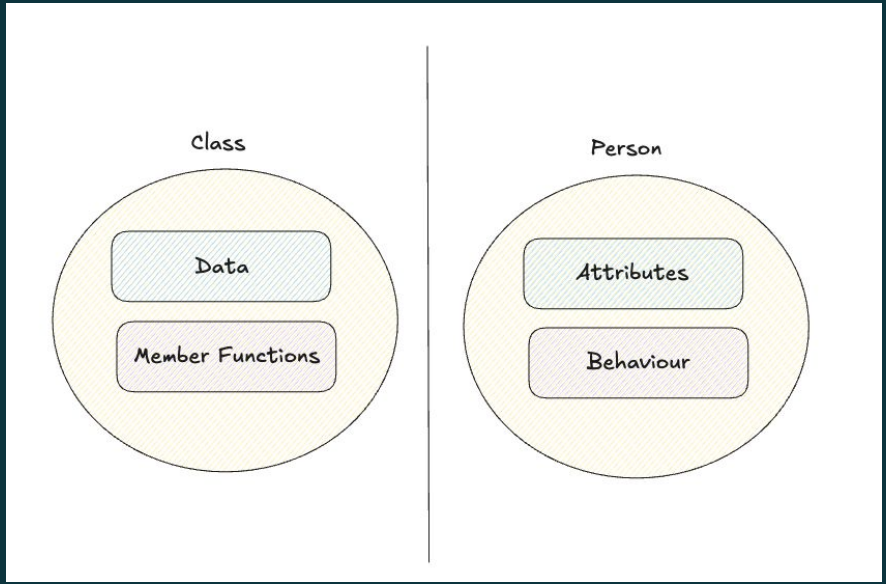
1. Abstraction

Abstracts away details from the user and exposes only essential features to them.



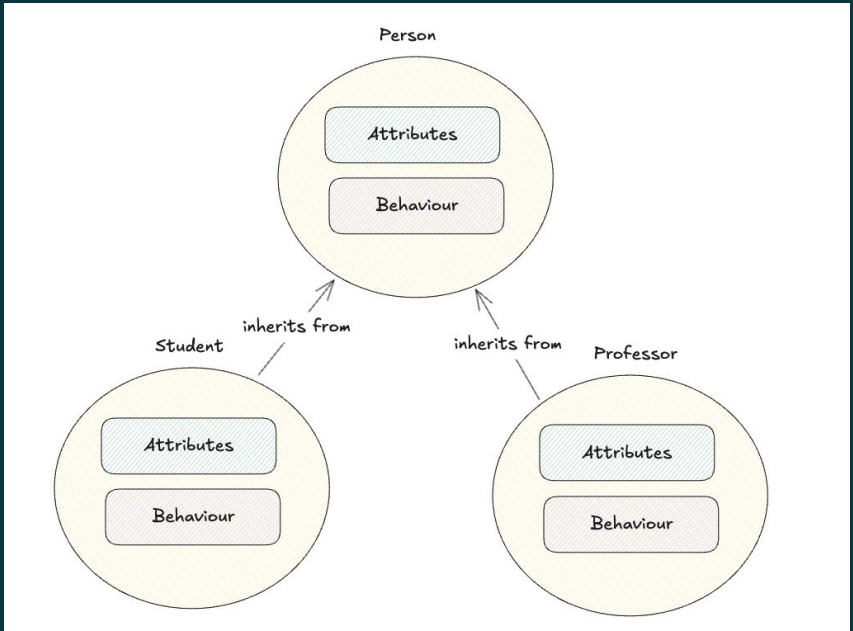
2. Encapsulation

Bundles data and methods that operate on the data into a single unit (entity), protecting data from outside inference.



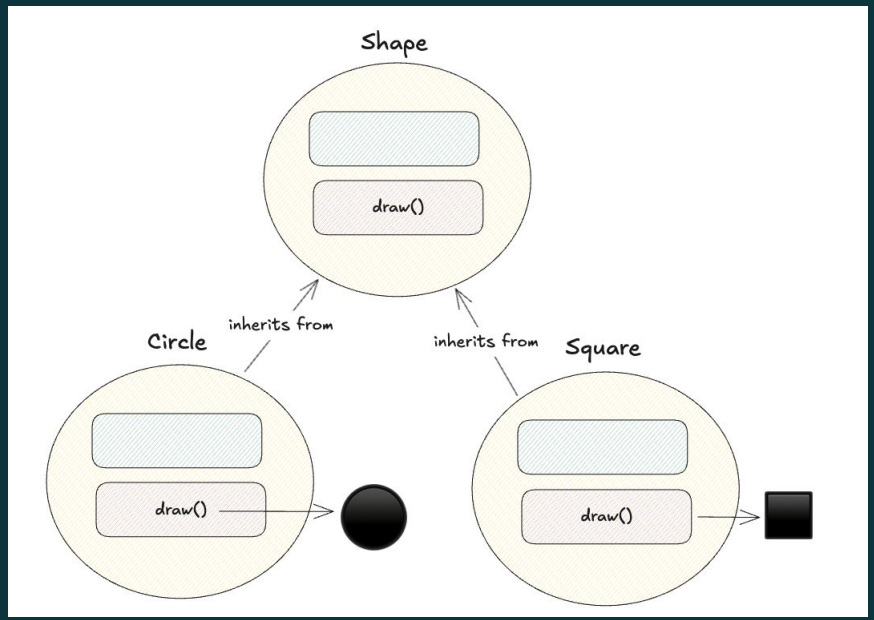
3. Inheritance

Allows a class (subclass) to inherit properties and behaviours from another class (superclass), facilitating code reuse and creating a hierarchy of classes.

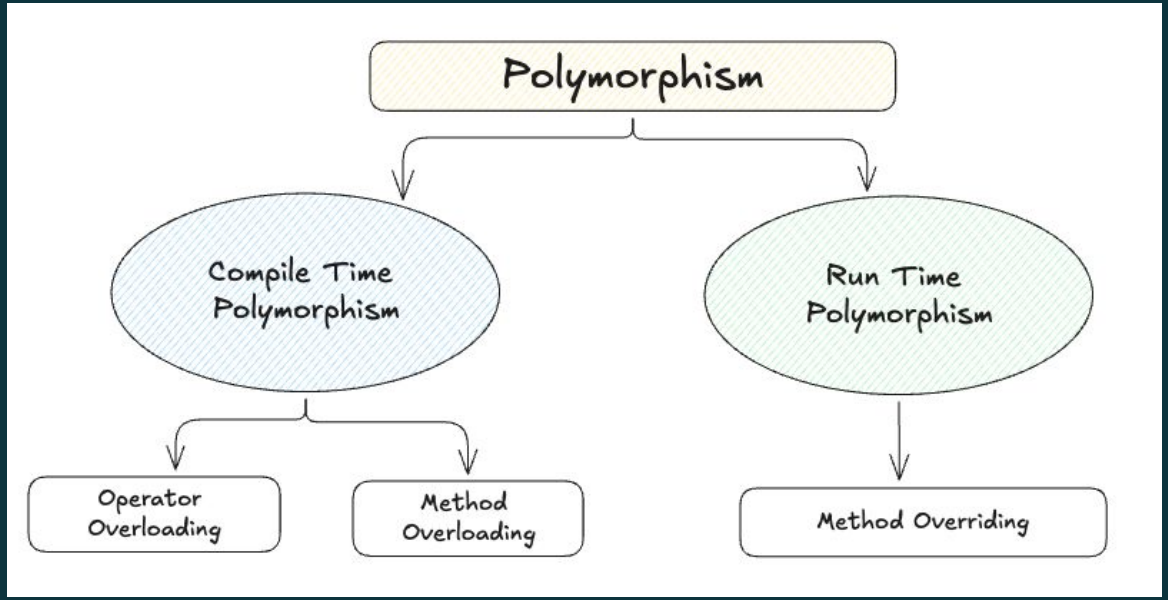


4. Polymorphism

Allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in method invocation.

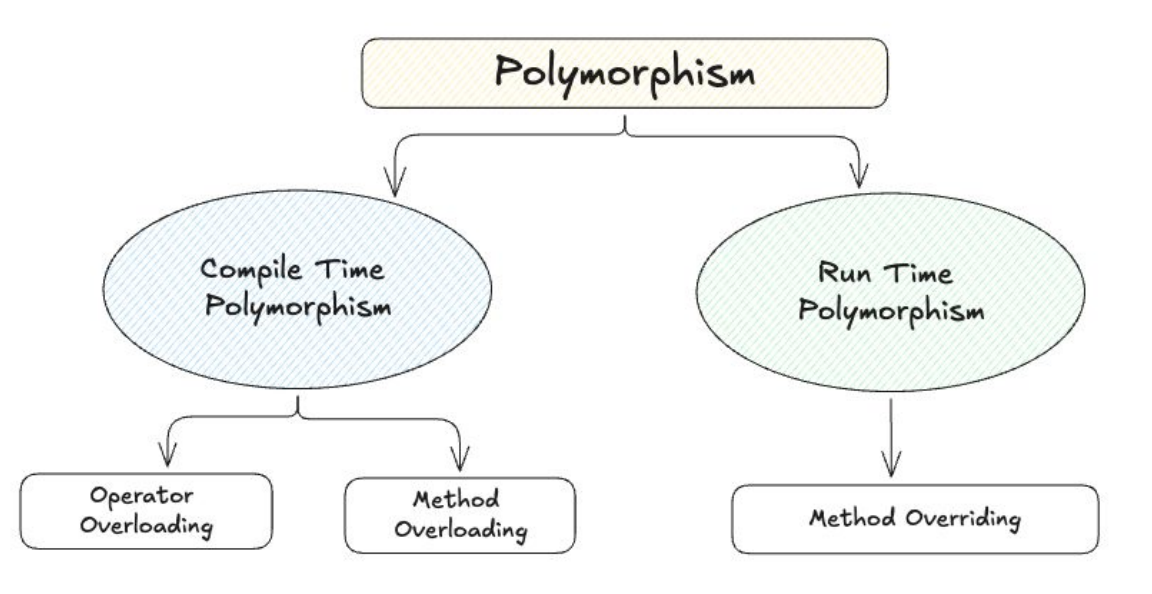


Polymorphism



Polymorphism

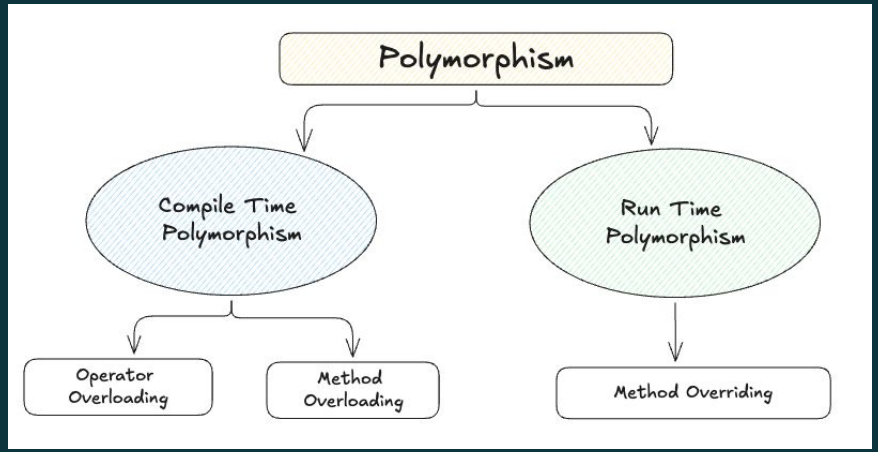
- Overloading happens when you keep the same method name but change the number or type of parameters.
- Overriding occurs when you keep the same method name and signature but change the implementation.



Polymorphism

A note on Polymorphism:

- *Python **does not** support Method Overloading (compile-time polymorphism).*
- *Python **supports** Method Overriding (run-time polymorphism).*
- *Python **supports** Operator Overloading (via dunder methods like `__eq__()`, etc.)*



Object-Oriented Programming

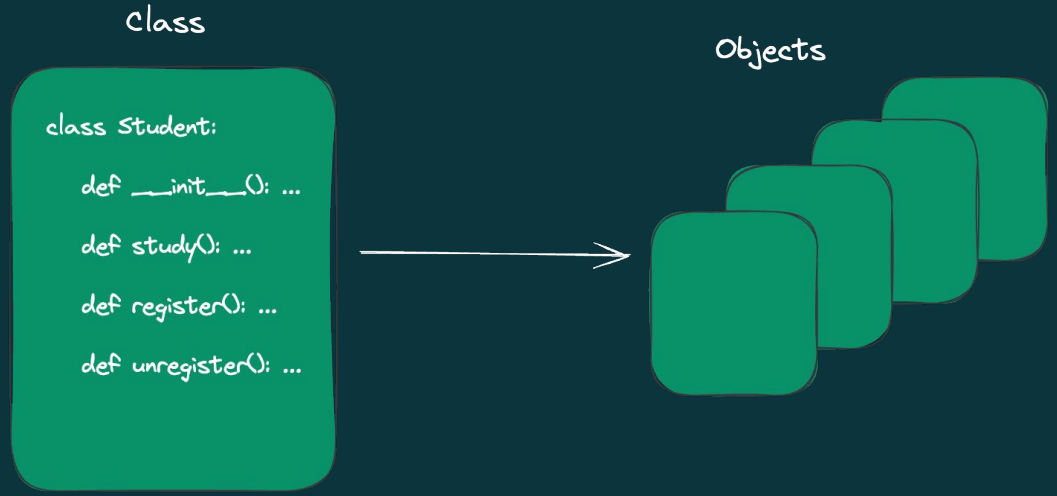
OOP consists of four principles:

1. **Abstraction:** allows information hiding and exposing only essential features to users.
2. **Encapsulation:** bundling data and methods that operate on the data into a single unit, protecting data from outside inference.
3. **Inheritance:** allows a class (subclass) to inherit properties and behaviours from another class (superclass), facilitating code reuse and creating a hierarchy of classes.
4. **Polymorphism:** allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in method invocation.

Classes and Objects

Classes and Objects

- **Class** - a blueprint for creating objects.
  - An object is a collection of **attributes** and **behaviours**.
- **Object** - an instantiation of the blueprint.



Defining a Class in Python

Class

- Blueprint for creating objects.
- Defines **fields** (variables) and **methods** (functions).

Object

- An instance of a class.
- Represents a specific entity in the real world.

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

my_car = Car('Toyota', 'Corolla')
print(my_car.make)  # Output: Toyota
print(my_car.model) # Output: Corolla
```

Instance & Class variables

There are two types of variables when defining in a class.

Instance & Class variables

There are two types of variables when defining in a class.

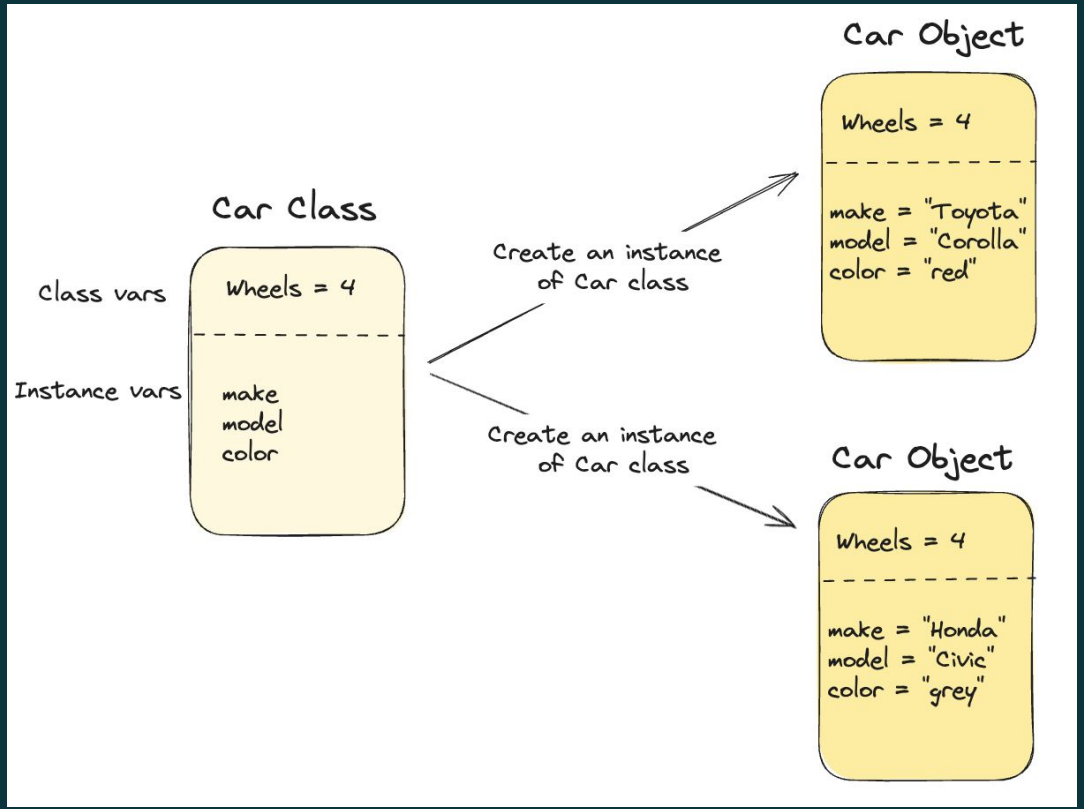
- Instance variables
  - A variables that belongs to an **instance** of a class.
  - Every object or instance created from a class can have its own separate values for instance variables.

Instance & Class variables

There are two types of variables when defining in a class.

- Instance variables
  - A variables that belongs to an **instance** of a class.
  - Every object or instance created from a class can have its own separate values for instance variables.
- Class variables
  - A class variables is **shared across all instances** of a class.
  - It belongs to the class itself, and not to any specific object.

Instance & Class variables



Instance & Class variables

- Class variables
  - Useful for sharing data across all instances of the class.
- Instance variables
  - Useful for initialising data specific to the instance of the class.

```
class Car:

    wheels = 4  # Class variable

    def __init__(self, make, model):

        self.make = make  # Instance variable

        self.model = model  # Instance variable

car1 = Car('Toyota', 'Corolla')
car2 = Car('Honda', 'Civic')

print(car1.wheels)  # Output: 4
print(car2.wheels)  # Output: 4
Car.wheels = 5
print(car1.wheels)  # Output: 5
print(car2.wheels)  # Output: 5
```

Instance, Static, and Class Methods

Instance, Static, and Class Methods

There are three types of methods in a class:

Instance, Static, and Class Methods

There are three types of methods in a class:

1. Instance methods

- Operate on an instance of the class.

Instance, Static, and Class Methods

There are three types of methods in a class:

1. Instance methods

- Operate on an instance of the class.

2. Static methods

- Do not operate on an instance or class. Defined using **@staticmethod**.

Instance, Static, and Class Methods

There are three types of methods in a class:

1. Instance methods

- Operate on an instance of the class.

2. Static methods

- Do not operate on an instance or class. Defined using **@staticmethod**.

3. Class methods

- Operate on the class itself. Defined using **@classmethod**.

Instance, Static, and Class Methods

When defining methods:

- Instance methods **take `self`** as their first argument.
- Class methods **take `cls`** as their first argument.
- Static methods **don't take `self` or `cls`** as their first argument.

Instance, Static, and Class Methods

When to Use Each Method?

Instance, Static, and Class Methods

When to Use Each Method?

- Instance Methods: When you need to work with the object's attributes.
- Class Methods: When you need to work with the class itself (e.g., class-level variables).
- Static Methods: When you need a utility function that relates to the class but doesn't require access to the class or its instances.

Dunder methods

Dunder methods

- Dunder methods are predefined methods in Python that have double underscores (hence called dunder) at the beginning and end of their names.

Dunder methods

- Dunder methods are predefined methods in Python that have double underscores (hence called dunder) at the beginning and end of their names.
- These methods provide a way to define specific behaviors for built-in operations or functionalities in Python classes.
    - Examples are `__add__`, `__eq__`, `__str__`, etc.

Dunder methods

- Dunder methods are predefined methods in Python that have double underscores (hence called dunder) at the beginning and end of their names.
- These methods provide a way to define specific behaviors for built-in operations or functionalities in Python classes.
  - Examples are `__add__`, `__eq__`, `__str__`, etc.
- You'll find a summarised **dunder-methods-cheat-sheet** attached in this lecture.

Inheritance

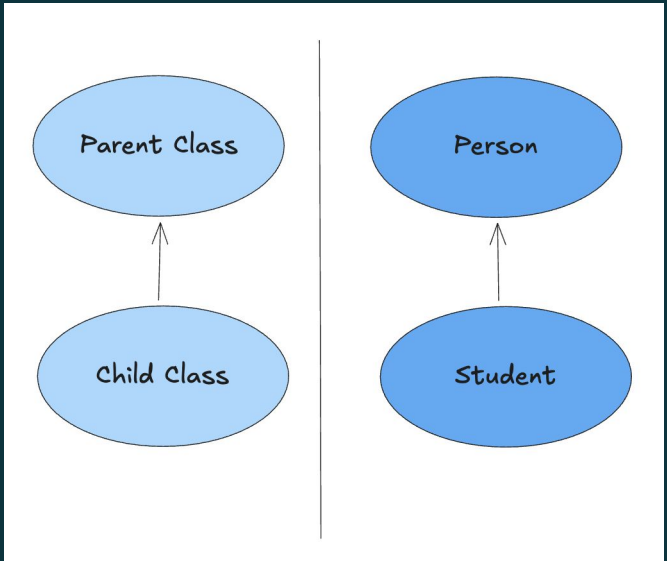
Technique to create a new class by reusing attributes and methods of an existing class.

Helps reduces code redundancy by reusing the same logic defined in another class.

Inheritance

Technique to create a new class by reusing attributes and methods of an existing class.

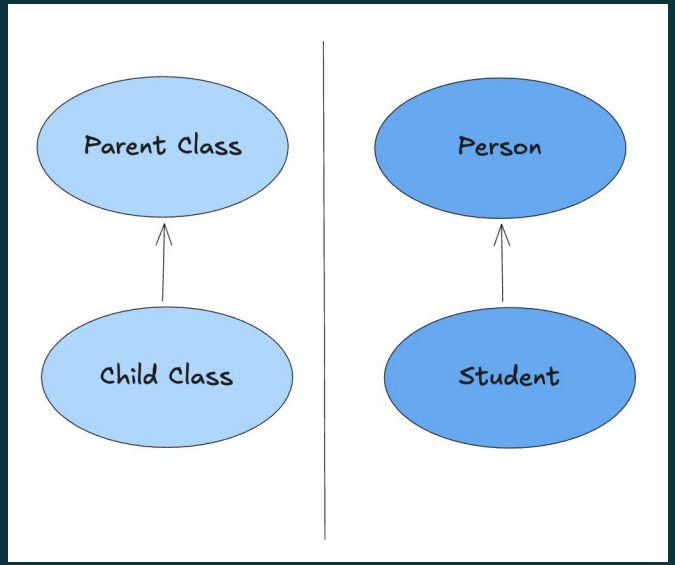
Helps reduces code redundancy by reusing the same logic defined in another class.

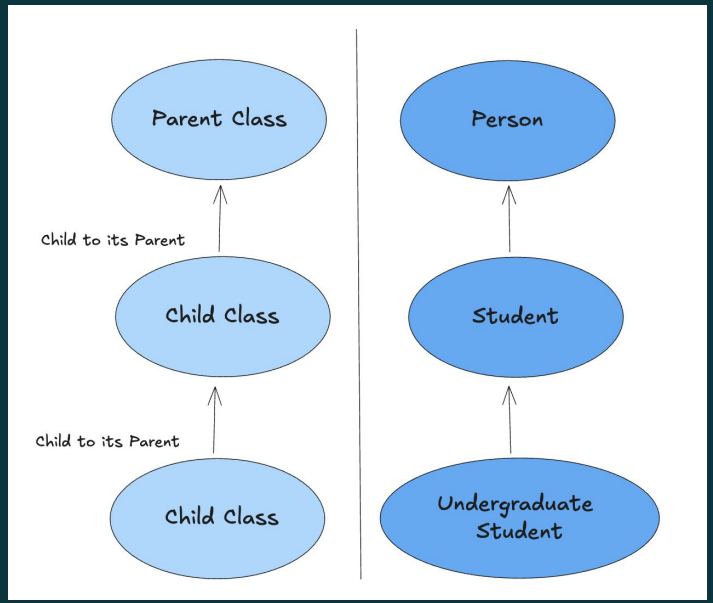


Inheritance

Technique to create a new class by reusing attributes and methods of an existing class.

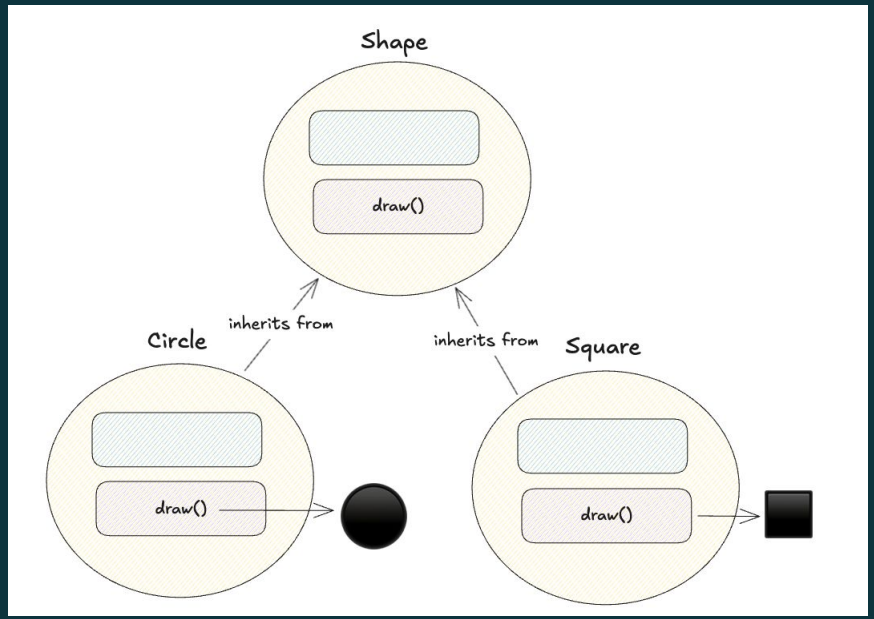
Helps reduces code redundancy by reusing the same logic defined in another class.





Polymorphism

Ability of different objects to respond, each in its own way, to identical messages (methods).



Abstract Classes

Abstract Classes

- A class that cannot be instantiated.
- Contains abstract methods that must be implemented by derived classes.

Use Case:

- Enforces a contract for subclasses.

Abstract Classes

Example of creating a class called **Animal** with an **abstract method** called **speak()** that cannot be instantiated at runtime.

However, class **Dog** inherits from **Animal** and implements the **speak()** method.

Dog Class is what we call a **Concrete Class**. In other words, a Class that can be instantiated at runtime.

```
from abc import ABC, abstractmethod


class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass
class Dog(Animal):
    def speak(self):
        return 'Bark'


dog = Dog()
print(dog.speak())  # Output: Bark
```

Class Generics

Class Generics

- Generics is a technique to define functions, classes, or methods that can operate on multiple types while maintaining type safety.

Class Generics

- Generics is a technique to define functions, classes, or methods that can operate on multiple types while maintaining type safety.
- Generics make it is possible to write reusable code that can be used with different data types.

Class Generics

- Generics is a technique to define functions, classes, or methods that can operate on multiple types while maintaining type safety.
- Generics make it is possible to write reusable code that can be used with different data types.
- It ensures promoting code flexibility and type correctness.

Class Generics

- Generics is a technique to define functions, classes, or methods that can operate on multiple types while maintaining type safety.
- Generics make it is possible to write reusable code that can be used with different data types.
- It ensures promoting code flexibility and type correctness.
- *Python 3.12 offers a newly improved syntax for typing Generics.*

Generics in Python

The syntax **Stack[T]** implies that you're defining a generic class parametrized by T. This implicitly defines **T** as a type variable that you can refer to within the class definition.

```
class Stack(T):
    def __init__(self):
        self.items = []

    def push(self, item: T):
        self.items.append(item)

    def pop(self) -> T:
        return self.items.pop()

stack = Stack[int]()
stack.push(1)
stack.push(2)
print(stack.pop())  # Output: 2
```

Generics in Python

The syntax **Stack[T]** implies that you're defining a generic class parametrized by T. This implicitly defines **T** as a type variable that you can refer to within the class definition.

Prior to Python 3.12, you would need to write more syntax for the same logic by importing **Generic**, **TypeVar** and explicitly define **T** makes your code cleaner and more readable.

This new syntax in Python 3.12 makes your code cleaner and more readable.

```
class Stack[T]:
    def __init__(self):
        self.items = []

    def push(self, item: T):
        self.items.append(item)

    def pop(self) -> T:
        return self.items.pop()

stack = Stack[int]()
stack.push(1)
stack.push(2)
print(stack.pop())  # Output: 2
```

Enums

Enums

What is an Enum?

- Enum (Enumeration): A symbolic name for a set of unique values. Enums are used to define a collection of constant values that are related.

Enums

What is an Enum?

- Enum (Enumeration): A symbolic name for a set of unique values. Enums are used to define a collection of constant values that are related.

Why Use Enums?

Enums

What is an Enum?

- Enum (Enumeration): A symbolic name for a set of unique values. Enums are used to define a collection of constant values that are related.

Why Use Enums?

- Clarity: Improves code readability by using meaningful names instead of raw values.
- Type Safety: Ensures that only valid values are assigned to variables.
- Grouping: Groups related constants together in a single data structure.

Enums

Key Features:

- **Enums are iterable:** You can loop through an Enum class.
- **Unique values:** Enum members are unique; two members cannot have the same value.
- **Immutable:** Enum members are constants and cannot be modified.

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3


red = Color.RED
green = Color.GREEN

print(red == Color.RED) # True
print(red == green)     # False
print(Color.RED.name)   # RED
print(Color.RED.value)  # 1
```

Duck Typing

Duck Typing

“If it looks like a duck and quacks like a duck, then it probably is a duck."

- Duck Typing means that Python doesn’t check the type of an object. Instead, it checks whether an object has the necessary methods and attributes to perform a task.
- In other words, type checking is determined by methods and properties rather than the actual type.

Protocols

Protocols

- A Protocol defines a set of methods or properties that a class must implement to be considered as "compatible" with that protocol.
- It formalizes the idea of **Duck Typing** by saying, "*If an object implements this set of methods, it can be treated as if it conforms to the Protocol.*"

Protocols introduced in PEP 544: https://peps.python.org/pep-0544/

Duck Typing with Protocols

Duck Typing is a powerful feature of Python that allows us to write flexible code by focusing on the behavior of objects, not their type.

```
from typing import Protocol

class Flyer(Protocol):
    def fly(self) -> str:
        ...

class Bird:
    def fly(self) -> str:
        return 'Bird is flying'

class Airplane:
     def fly(self) -> str:
         return 'Airplane is flying'

def lift_off(flyer: Flyer):
    print(flyer.fly())

bird = Bird()
airplane = Airplane()
lift_off(bird)        # Output: Bird is flying
lift_off(airplane)    # Output: Airplane is flying
```

Duck Typing with Protocols

Duck Typing is a powerful feature of Python that allows us to write flexible code by focusing on the behavior of objects, not their type.

Protocols give us a way to formalize Duck Typing by defining clear contracts that objects must adhere to.

```
from typing import Protocol

class Flyer(Protocol):
    def fly(self) -> str:
        ...

class Bird:
    def fly(self) -> str:
        return 'Bird is flying'

class Airplane:
     def fly(self) -> str:
         return 'Airplane is flying'

def lift_off(flyer: Flyer):
    print(flyer.fly())

bird = Bird()
airplane = Airplane()
lift_off(bird)        # Output: Bird is flying
lift_off(airplane)    # Output: Airplane is flying
```

Duck Typing with Protocols

Duck Typing is a powerful feature of Python that allows us to write flexible code by focusing on the behavior of objects, not their type.

Protocols give us a way to formalize Duck Typing by defining clear contracts that objects must adhere to.

Protocols offer the best of both worlds: the flexibility of Python's dynamic typing and the type safety of static typing.

```
from typing import Protocol

class Flyer(Protocol):
    def fly(self) -> str:
        ...

class Bird:
    def fly(self) -> str:
        return 'Bird is flying'

class Airplane:
     def fly(self) -> str:
         return 'Airplane is flying'

def lift_off(flyer: Flyer):
    print(flyer.fly())

bird = Bird()
airplane = Airplane()
lift_off(bird)        # Output: Bird is flying
lift_off(airplane)    # Output: Airplane is flying
```

Conclusion

- OOP Principles
  - Abstraction, Encapsulation, Inheritance, and Polymorphism
- Classes and objects
- Instance and class variables
- Method types (instance, class, static)
- Dunder methods / special methods
- Inheritance and Polymorphism
- Abstract classes
- Class Generics
- Enums
- Protocols