

- Inheritance allows a new class (child/subclass) to inherit attributes and methods from an existing class (parent/superclass).
- Supports code reuse and method overriding (custom behavior).
- Syntax:

```
python
class Puppy(Dog): # Puppy inherits from Dog
    def weep(self):
        return "Weep..."
    • Example overriding method:
python
class Puppy(Dog):
    def bark(self): # override
        return f"{self.name} says Yip!"
```

7. Polymorphism

- Polymorphism means “many forms.”
- It allows using a unified interface to objects of different classes.
- For example, different subclasses can have their own implementation of the same method name, and the correct method is called depending on the object type.

8. Abstraction

- Abstraction hides complex implementation details and exposes only essential features.
- Often implemented using abstract base classes and methods in Python (abc module).

Example Illustrating All Concepts

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

    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"

# Instantiation and polymorphism
animals = [Dog("Buddy"), Cat("Whiskers")]

for animal in animals:
    print(animal.speak()) # Calls the appropriate speak() based on object type
```

Advantages of OOP

- Modularity: Code organized into logical units.
- Reusability: Use inheritance and composition to avoid rewriting code.
- Maintainability: Easier to debug and extend.
- Encapsulation: Protects object's integrity by hiding internals.
- Polymorphism: Flexible code interfaces.

Summary

Python makes OOP accessible with simple syntax supporting essential concepts like classes/objects, encapsulation, inheritance, polymorphism, and abstraction. These principles help create scalable and robust software by modeling real-world entities and behaviors in a clear and modular way.

OOP is foundational for many Python frameworks and large projects, empowering developers to create maintainable, extensible applications.

Exception handling in Python

Exception handling in Python is a mechanism to gracefully manage errors that occur during program execution, allowing the program to continue running or fail gracefully rather than crashing abruptly.

Key Concepts of Exception Handling in Python

- Exceptions: Errors detected during execution, such as division by zero, invalid input, or file not found.
- Try block: Wraps code that might raise an exception.
- Except block: Defines how to handle specific exceptions if they occur.
- Else block: Code that runs if no exception occurs in the try block.
- Finally block: Code that always runs, regardless of exceptions, often used for cleanup.

Basic Syntax

```
python
try:
    # Code that may cause an exception
except SomeException:
    # Code to handle the exception
else:
    # Code to run if no exceptions occur
finally:
    # Code that runs no matter what
```

Example: Handling Division by Zero

```
python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful:", result)
finally:
    print("Execution complete.")
```

Catching Multiple Exceptions

```
python
try:
    val = int(input("Enter a number: "))
    result = 10 / val
except ValueError:
    print("Invalid input! Please enter a valid integer.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```