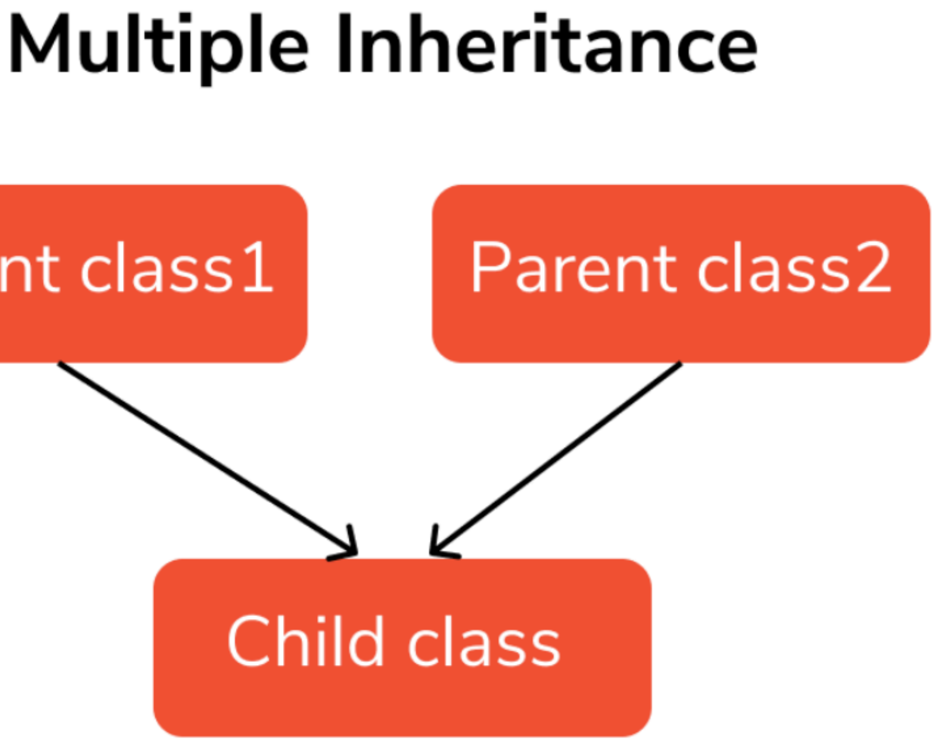
Object-Oriented Programming (OOP) in Python 2

Agenda

1. Advanced OOP Concepts
    - Multiple Inheritance
    - Class Methods and Static Methods
    - Magic Methods (Dunder Methods)
    - Abstract Classes, Methods and Interfaces
    - Abstract Base Classes (ABCs)
    - Composition vs. Inheritance
    - Mixins
    - Metaclasses
    - Decorators in OOP

Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class. This feature enables a derived class to inherit attributes and methods from multiple parent classes, providing more flexibility in designing classes.



1. The Diamond Problem
    - A potential issue with multiple inheritance is the "diamond problem," where a derived class inherits from two classes that both inherit from a common base class. This can lead to ambiguity in method resolution.

```
# In the below example, class D inherits from both B and C, and the method say_hello() is ca
class A:
    def say_hello(self):
        return "Hello from A"

class B(A):
    def say_hello(self):
        return "Hello from B"

class C(A):
    def say_hello(self):
        return "Hello from C"

class D(B, C):
    pass

d = D()
print(d.say_hello())
```

```
Hello from B
```

2. Method Resolution Order (MRO)
    - Python uses the C3 linearization algorithm to determine the MRO, which is the order in which base classes are searched when executing a method. You can view the MRO of a class using the **mro** attribute or mro() method.

```
print(D.mro())
```

```
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>,
```

Class Methods and Static Methods

Class Methods

- A class method is a method that is bound to the class and not the instance of the class. It can modify class state that applies across all instances of the class.
- How to Define: Class methods are defined using the @classmethod decorator.
- Parameter: The first parameter of a class method is cls, which refers to the class itself.
- Use Cases:
    - Accessing or modifying class variables (shared among all instances).
    - Creating alternative constructors.

```
class Car:
    # Class variable
    wheels = 4

    def __init__(self, make, model):
        self.make = make
        self.model = model

    # Class method
    @classmethod
    def change_wheels(cls, number):
        cls.wheels = number

    @classmethod
    def from_string(cls, car_str):
        make, model = car_str.split('-')
        return cls(make, model)

# Changing class variable using class method
Car.change_wheels(6)
```

```
print(Car.wheels)  # Output: 6

# Using class method as an alternative constructor
car = Car.from_string("Toyota-Corolla")
print(car.make, car.model)
```

```
6
Toyota Corolla
```

Static Method

- A static method is a method that belongs to a class but does not access or modify the class state or instance state. It behaves like a regular function but is called on the class.
- How to Define: Static methods are defined using the @staticmethod decorator.
- Parameter: Static methods do not take self or cls as their first parameter since they do not operate on an instance or the class.
- Use Cases:
  - Utility functions that perform a task in isolation, without needing access to class or instance data.
  - Grouping functions logically within a class that are related to the class but don't require instance or class variables.

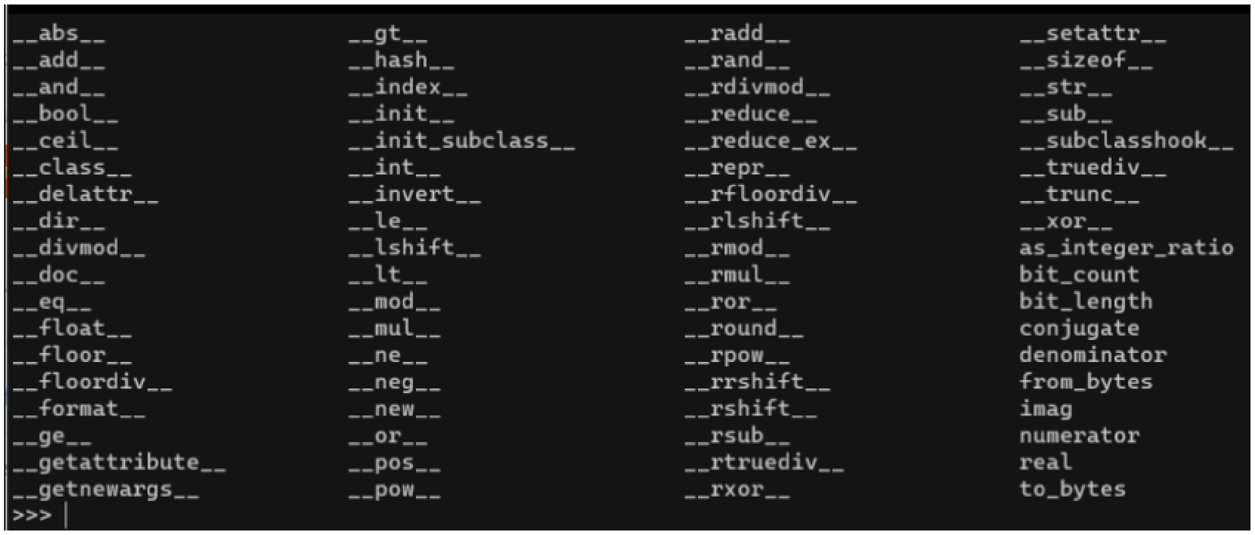
```
class MathOperations:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def multiply(x, y):
        return x * y

# Calling static methods
print(MathOperations.add(5, 3))
print(MathOperations.multiply(5, 3))
```

```
8
15
```

Magic Methods (Dunder Methods)

- Magic methods (also known as dunder methods) are special methods in Python that allow you to define how objects of your class interact with built-in Python operations such as arithmetic operations, comparisons, and string representations.

- Common Magic Methods:
  - **init**(): Constructor, called when an instance is created.
  - **str**() and **repr**(): Define string representations of objects.
  - **add**(), **sub**(), etc.: Overload arithmetic operators.
  - **eq**(), **lt**(), etc.: Overload comparison operators.
  - **getitem**(), **setitem**(), etc.: Allow objects to be indexed.



```
__abs__                 __gt__                  __radd__                __setattr__
__add__                 __hash__                __rand__                __sizeof__
__and__                 __index__               __rdivmod__             __str__
__bool__                __init__                __reduce__              __sub__
__ceil__                __init_subclass__       __reduce_ex__           __subclasshook__
__class__               __int__                 __repr__                __truediv__
__delattr__             __invert__              __rfloordiv__           __trunc__
__dir__                 __le__                  __rlshift__             __xor__
__divmod__              __lshift__              __rmod__                as_integer_ratio
__doc__                 __lt__                  __rmul__                bit_count
__eq__                  __mod__                 __ror__                 bit_length
__float__               __mul__                 __round__               conjugate
__floor__               __ne__                  __rpow__                denominator
__floordiv__            __neg__                 __rrshift__             from_bytes
__format__              __new__                 __rshift__              imag
__ge__                  __or__                  __rsub__                numerator
__getattribute__        __pos__                 __rtruediv__            real
__getnewargs__          __pow__                 __rxor__                to_bytes
>>> |
```

```
# Example
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
print(v1 + v2)
```

```
Vector(6, 8)
```

Abstract Classes, Methods and Interfaces

Abstract Classes

- An abstract class is a class that cannot be instantiated on its own and is meant to be subclassed. It can contain abstract methods (without implementation) and concrete methods (with implementation).
- Abstract classes are used to define a common interface for a group of subclasses, ensuring that they implement specific methods.

```
from abc import ABC, abstractmethod

class Shape(ABC):  # Abstract class
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Trying to instantiate an abstract class will raise an error
# shape = Shape()  # This will raise a TypeError

# Subclass implementing the abstract methods
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Usage
rectangle = Rectangle(5, 10)
print(f"Area: {rectangle.area()}")
print(f"Perimeter: {rectangle.perimeter()}")
```

```
Area: 50
Perimeter: 30
```

Abstract Methods

- An abstract method is a method that is declared in an abstract class but does not have an implementation. Subclasses of the abstract class are required to implement these methods. Abstract methods enforce that certain methods must be implemented by any subclass, ensuring a consistent interface.from abc import ABC, abstractmethod

```
class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Usage
dog = Dog()
cat = Cat()

print(dog.sound())
print(cat.sound())
```

```
Bark
Meow
```

Abstract Interfaces

- In Python, an abstract interface refers to an abstract class that defines a set of methods that must be implemented by any concrete (non-abstract) subclass. Abstract interfaces provide a way to enforce a contract for subclasses, ensuring that they implement specific methods.
- Why Use Abstract Interfaces?
    - Enforce a Contract: Abstract interfaces ensure that subclasses adhere to a specific method signature, promoting a consistent API.
    - Design Flexibility: Abstract classes allow you to define methods that must be implemented while providing a common interface.
    - Code Reusability: Common functionality can be defined in the abstract class, and specific implementations can be provided in subclasses.

Abstract Base Classes (ABCs)

- Abstract Base Classes (ABCs) in Python are a way to define abstract classes and methods, providing a framework for creating and enforcing contracts in object-oriented programming. They are part of the abc module, which stands for Abstract Base Classes.
- Using the 'abc' Module:

1. Importing abc Module

```
from abc import ABC, abstractmethod
```

2. Defining an Abstract Base Class

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        """Method to calculate the area of the shape"""
        pass

    @abstractmethod
    def perimeter(self):
        """Method to calculate the perimeter of the shape"""
        pass
```

3. Implementing Subclasses

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        import math
        return math.pi * (self.radius ** 2)
    def perimeter(self):
        import math
        return 2 * math.pi * self.radius
```

- Practical Example with ABCs:
    - In this example, the Vehicle class serves as an abstract base class with methods that must be implemented by any concrete subclass, such as Car and Motorcycle. This ensures that all vehicle types have a consistent interface for starting and stopping their engines.

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        return "Car engine started"

    def stop_engine(self):
        return "Car engine stopped"

class Motorcycle(Vehicle):
    def start_engine(self):
        return "Motorcycle engine started"

    def stop_engine(self):
        return "Motorcycle engine stopped"

car = Car()
print(car.start_engine())
print(car.stop_engine())

motorcycle = Motorcycle()
print(motorcycle.start_engine())
print(motorcycle.stop_engine())
```

```
Car engine started
Car engine stopped
Motorcycle engine started
Motorcycle engine stopped
```

Composition vs. Inheritance

Composition

- Composition is a design principle where a class is composed of one or more objects from other classes, effectively using these objects to build functionality. It represents a "has-a" relationship.
- Key Concepts
  - Has-a Relationship: A class contains instances of other classes, using their functionality to achieve its goals.
  - Delegation: The class delegates tasks to the objects it contains.
  - Encapsulation: Composition encapsulates functionality within objects that can be swapped or changed independently.
- Advantages:
  - Flexibility: Composition provides greater flexibility as you can easily change or replace components without affecting other parts of the system.
  - Loose Coupling: Classes are less tightly coupled compared to inheritance, making changes to one component less likely to impact others.
  - Reusability: Allows reuse of existing functionality in different contexts without modifying existing code.
- Disadvantages:
  - More Boilerplate: May require additional boilerplate code to delegate methods to composed objects.
  - Complexity: Can add complexity to the codebase if not managed properly, especially with multiple levels of composition.

```
# Composition example
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()  # Composition: Car "has-a" Engine

    def start(self):
        return self.engine.start()  # Delegating the start method to Engine
```

```
# Instantiate Car
car = Car()
print(car.start())
```

```
Engine started
```

Inheritance

- Inheritance is a mechanism where a new class (subclass or derived class) inherits attributes and methods from an existing class (base class or superclass). This allows the subclass to reuse code from the superclass and extend or override it.
- Key Concepts
  - Base Class: The class being inherited from.
  - Derived Class: The class inheriting from the base class.
  - Method Overriding: A subclass can provide a specific implementation of a method that is already defined in its superclass.
  - Super Keyword: Used to call methods from the base class.
- Advantages:
  - Code Reuse: Avoids duplication by allowing subclasses to reuse existing code.
  - Hierarchy Representation: Models hierarchical relationships naturally (e.g., a dog is an animal).
  - Polymorphism: Allows for objects of different classes to be treated as objects of a common superclass.
- Disadvantages:
  - Tight Coupling: Subclasses are tightly coupled to their base classes, which can make changes to the base class affect all derived classes.
  - Inheritance Depth: Deep inheritance hierarchies can become complex and hard to manage.
  - Fragile Base Class Problem: Changes in the base class can inadvertently break the subclasses

```
# Inheritance example
class Animal:
    def eat(self):
        return "Eating"

    def sleep(self):
        return "Sleeping"
```

```
class Dog(Animal):
    def bark(self):
        return "Woof!"

# Instantiate Dog
dog = Dog()
print(dog.eat())
print(dog.bark())
```

```
Eating
Woof!
```

Mixins

- What are Mixins?
    - A mixin is a class that provides methods that can be used by other classes. Mixins are not intended to be instantiated on their own; instead, they are meant to be inherited by other classes to augment their functionality.
    - To add reusable methods to classes without forcing a class to inherit from a base class or to build complex class hierarchies.

Characteristics of Mixins

- Single Responsibility: Each mixin class typically provides a single piece of functionality or a specific behavior.
- No Instantiation: Mixins are not intended to be instantiated directly. They are designed to be inherited by other classes.
- Composition Over Inheritance: Mixins promote the use of composition to combine different functionalities into a single class.

How Mixins Work

1. Defining Mixins
    - A mixin class defines methods that can be shared across multiple other classes. Here's an example:

```
class JsonMixin:
    import json

    def to_json(self):
        return self.json.dumps(self.__dict__)

class XmlMixin:
    def to_xml(self):
        import xml.etree.ElementTree as ET
        root = ET.Element(self.__class__.__name__)
        for key, value in self.__dict__.items():
            child = ET.SubElement(root, key)
            child.text = str(value)
        return ET.tostring(root, encoding='unicode')
```

2. Using Mixins in Other Classes
    - Mixins can be combined with other classes to add additional functionality:

```
class Person(JsonMixin, XmlMixin):
    def __init__(self, name, age):
        self.name = name
        self.age = age

# Instantiate Person
person = Person("Alice", 30)

# Use mixin methods
print(person.to_json())
print(person.to_xml())
```

```
{"name": "Alice", "age": 30}
<Person><name>Alice</name><age>30</age></Person>
```

```
# Here’s a practical example that uses mixins to provide logging and validation functionalit
class LoggingMixin:
    def log(self, message):
        print(f"[LOG] {message}")

class ValidationMixin:
    def validate(self):
        if not self.name or not self.age:
            raise ValueError("Name and age must be provided")

class User(LoggingMixin, ValidationMixin):
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.validate()  # Ensure validation occurs upon instantiation

# Instantiate User
user = User("Bob", 25)
user.log("User created successfully")
```

```
[LOG] User created successfully
```

Advanced Usage

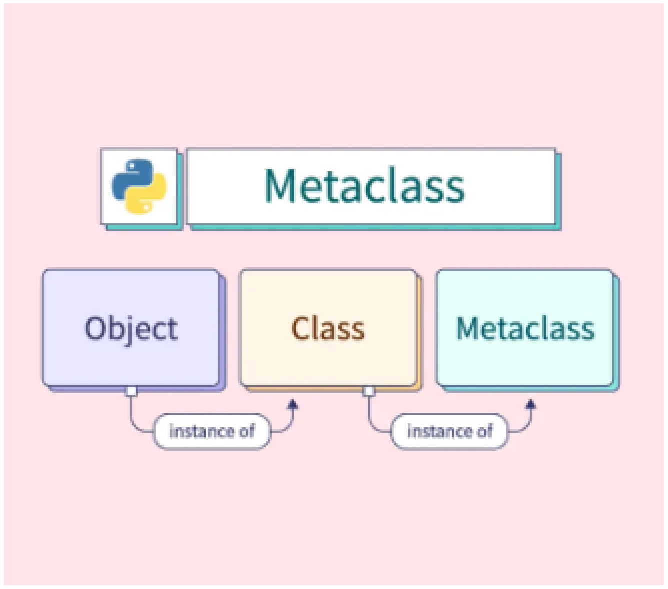
- Mixins can be combined with other patterns and techniques, such as decorators or context managers, to provide advanced functionality:

```
class SerializableMixin:
    def serialize(self, format='json'):
        if format == 'json':
            import json
            return json.dumps(self.__dict__)
        elif format == 'xml':
            import xml.etree.ElementTree as ET
            root = ET.Element(self.__class__.__name__)
            for key, value in self.__dict__.items():
                child = ET.SubElement(root, key)
                child.text = str(value)
            return ET.tostring(root, encoding='unicode')
        else:
            raise ValueError("Unsupported format")

class Product(SerializableMixin):
    def __init__(self, name, price):
        self.name = name
        self.price = price

# Instantiate Product
product = Product("Laptop", 999.99)
print(product.serialize('json'))
print(product.serialize('xml'))
```

```
{"name": "Laptop", "price": 999.99}
<Product><name>Laptop</name><price>999.99</price></Product>
```

Metaclasses

- What is Metaclass?
    - A metaclass is a class of a class. Just as a class defines how an object behaves, a metaclass defines how a class behaves.
    - Metaclasses allow you to control the creation and behavior of classes. They let you modify class definitions before the class is instantiated.



The type Metaclass

- The type function in Python serves a dual purpose:
    1. To check the type of an object:

```
print(type(5))
```

```
<class 'int'>
```

2. To create a new class dynamically:

```
# Create a new class dynamically using type
MyDynamicClass = type('MyDynamicClass', (object,), {'x': 5})
print(MyDynamicClass)
print(MyDynamicClass().x)
```

```
<class '__main__.MyDynamicClass'>
5
```

Custom Metaclasses

You can create custom metaclasses by subclassing type. Custom metaclasses allow you to intercept and modify class creation.

1. 1. Basic Custom Metaclass:

```
class MyMeta(type):
    def __new__(cls, name, bases, dct):
        print(f"Creating class {name}")
        return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=MyMeta):
    def hello(self):
        print("Hello, World!")

# Instantiate MyClass
obj = MyClass()
obj.hello()
```

```
Creating class MyClass
Hello, World!
```

2. Adding Custom Behavior:

    - You can use metaclasses to enforce coding standards or automatically modify class attributes:

```
class UpperCaseAttributeMeta(type):
    def __new__(cls, name, bases, dct):
        new_attrs = {}
        for attr_name, attr_value in dct.items():
            if not attr_name.startswith('__'):
                new_attrs[attr_name.upper()] = attr_value
            else:
                new_attrs[attr_name] = attr_value
        return super().__new__(cls, name, bases, new_attrs)

class MyClass(metaclass=UpperCaseAttributeMeta):
    x = 5
    y = 10

# Instantiate MyClass
obj = MyClass()
print(hasattr(obj, 'x'))
print(hasattr(obj, 'X'))
print(obj.X)
```

```
False
True
5
```

Understanding the Method Resolution Order (MRO)

- When dealing with multiple inheritance, understanding the Method Resolution Order (MRO) is crucial, especially when using metaclasses. MRO determines the order in which base classes are searched when executing a method. Python uses the C3 linearization algorithm to compute MRO.

```
# You can check the MRO of a class using
print(MyClass.mro())
```

```
[<class '__main__.MyClass'>, <class 'object'>]
```

Decorators in OOP

- Decorators are a powerful tool in Python that allows you to modify the behavior of a function or a method without changing its code. In the context of OOP, decorators can be used to add functionality to methods or classes.

Method Decorators

- Method decorators are functions that are applied to methods to modify their behavior. Common examples include @staticmethod, @classmethod, and custom decorators.

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Before calling the function")
        result = func(*args, **kwargs)
        print("After calling the function")
        return result
    return wrapper

class MyClass:
    @my_decorator
    def my_method(self):
        print("Inside the method")

obj = MyClass()
obj.my_method()
```

```
Before calling the function
Inside the method
After calling the function
```

Class Decorators

- Class decorators are applied to entire classes to modify their behavior. A class decorator is a function that takes a class as an argument and returns a modified class.

```
    def add_method(cls):
```