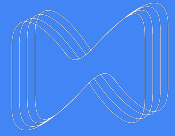
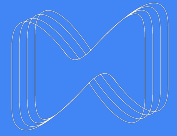
Course: Python development

Jan-Feb, 2025, Arobs

Agenda

- Inheritance
- Composition
- Abstract Classes
- Decorator
- ChatWrap





Hiding details - the public interface

The **key purpose** of an OOP design is **the interface**. The interface is the collection of attributes and methods of an object

Other objects should be able to access and interact with an object only via its interface. They do not need to know or are not allowed to access the internals of the object.

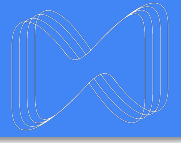
This process of hiding the implementation of an object - **information hiding**.

It is also sometimes referred to as **encapsulation**, but encapsulated data is not necessarily hidden.

We can use this definition for encapsulation as we don't actually have or need true information hiding in Python.

The distinction between encapsulation and information hiding is largely irrelevant at the design level.

Inheritance and Composition

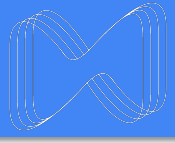


Both concepts help with Code Reusability

- Inheritance
  - a class wants to derive the parent class and then modify or extend its functionality
  - extending the functionality with extra features; allows overriding of methods
- Composition
  - We can only use that class, we can't modify or extend its functionality. It will not provide extra features. We combine other objects that add functionality.

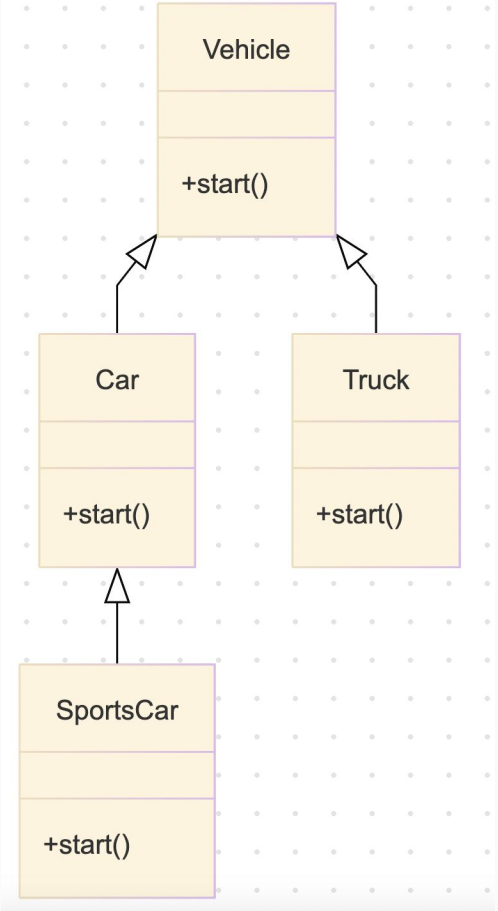
Composition is often better because it makes your code more **flexible and decoupled**. Instead of hardwiring behaviors via an inheritance chain, you build objects by combining simpler parts. This means you can **swap** or **update** parts without messing up the whole system, making your code easier to test and maintain. Plus, it sidesteps some pitfalls of deep inheritance hierarchies.

Object-oriented principles - Inheritance

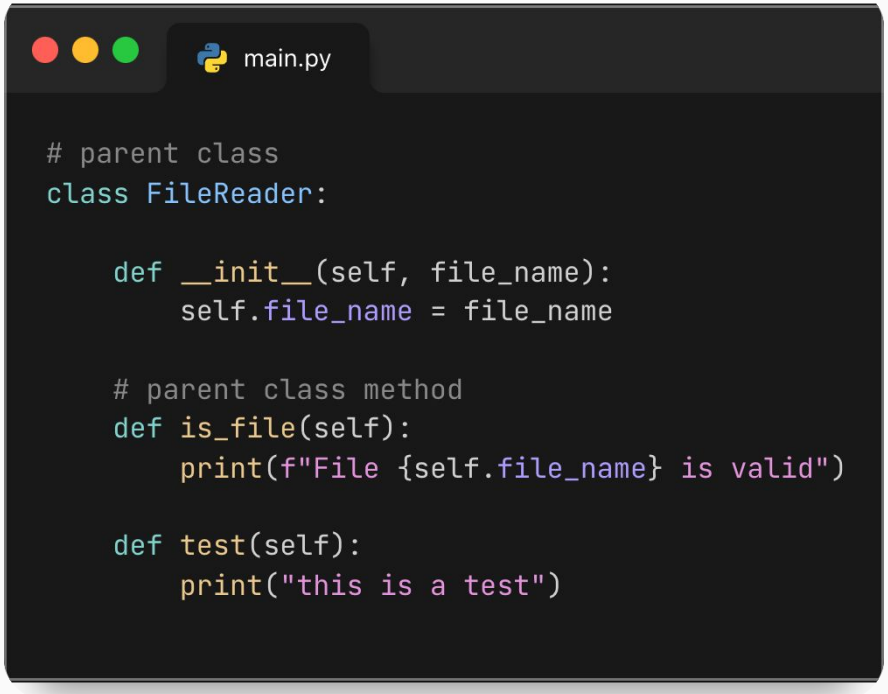


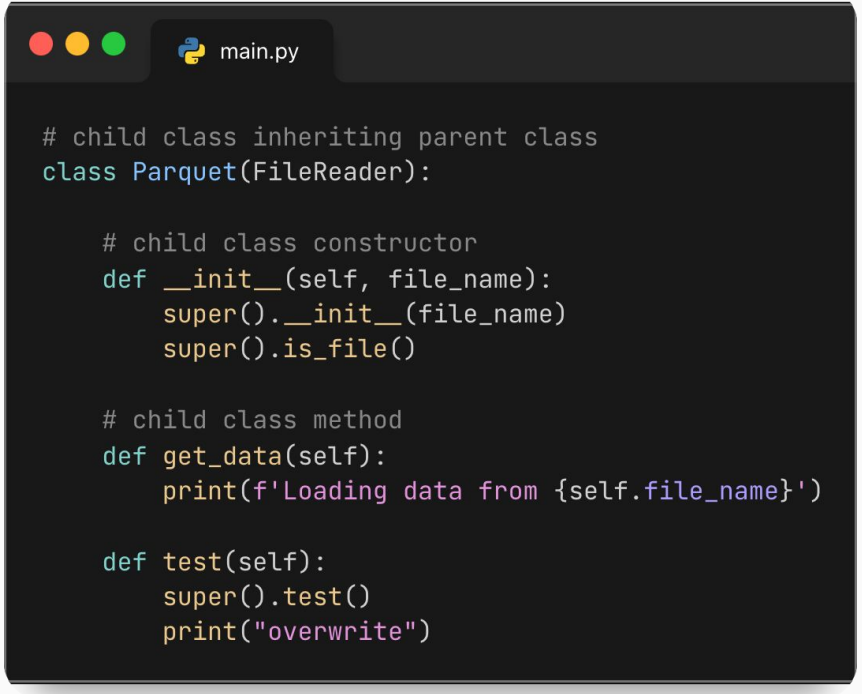
Inheritance

- one class can inherit attributes and methods from another class
- the most famous, well-known, and over-used relationship in object-oriented programming

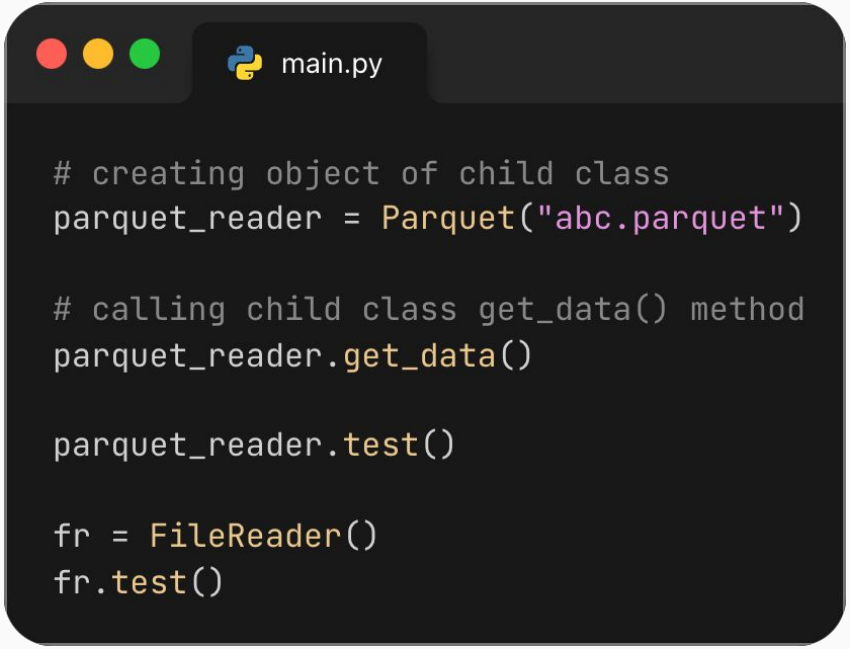


Object-oriented principles - Inheritance





Object-oriented principles - Inheritance



- A child class will inherit all the methods of the parent class
- We can overwrite methods from the parent class
- We can still use the parent class methods when overwriting them

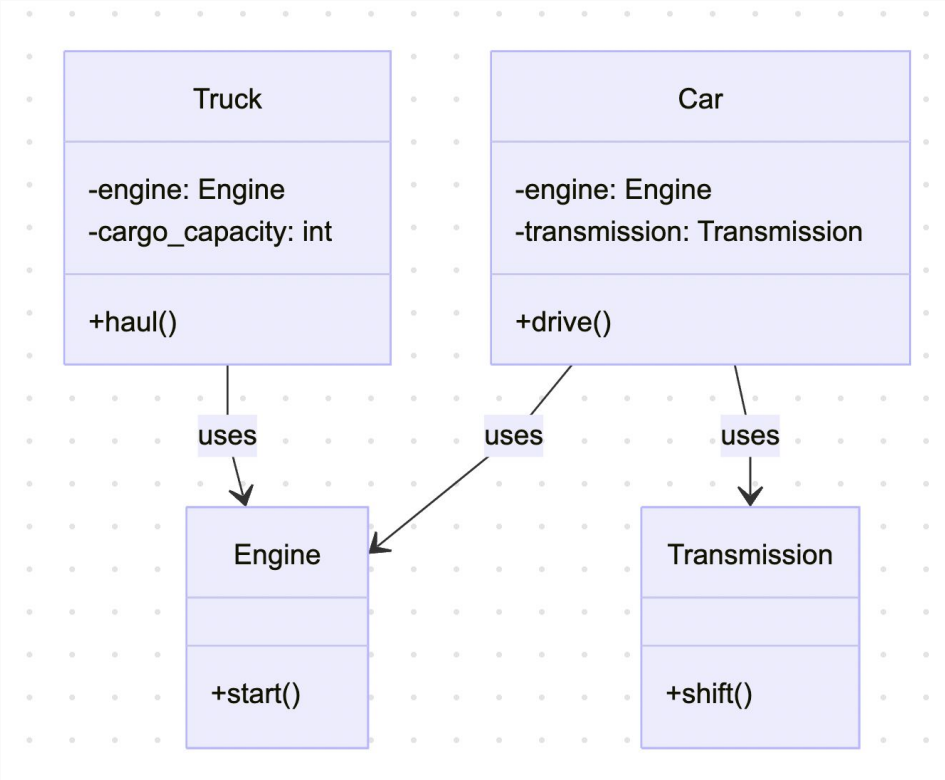
Observation

- We add a constructor to the parent class which assigns a file name to its attributes
- We make the child class re-use its parent constructor super().__init__(file_name)
- The child extends the constructor by validating the file at creation time

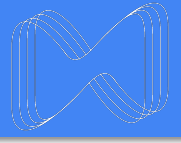
Object-oriented principles - Composition

Composition

- the act of collecting several objects together to create a new one.One object is part of another object
- composition is a special type of aggregation. Aggregated objects can exist independently



Object-oriented principles - Composition



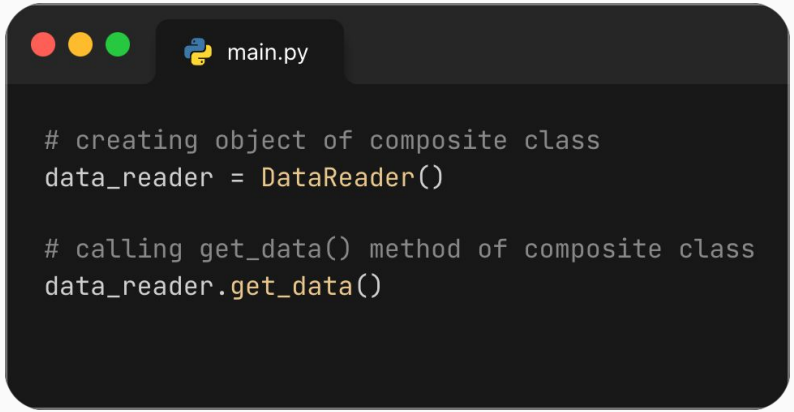
main.py

```
class DataReader:

    # constructor
    def __init__(self):
        # creating object of component class
        self.file_reader = Parquet()

        print('DataReader class object also created...')

    # composite class instance method
    def get_data(self):
        print('DataReader class get_data() method executed...')

        # calling load() method of component class
        self.file_reader.load()
```

main.py

```
class Parquet:

    def __init__(self):
        print('ParquetReader class object created...')

    # composite class instance method
    def load(self):
        print('Parquet class load() method executed...')

class Csv:

    def __init__(self):
        print('CsvReader class object created...')

    # composite class instance method
    def load(self):
        print('Csv class load() method executed...')
```

Object-oriented principles - Composition



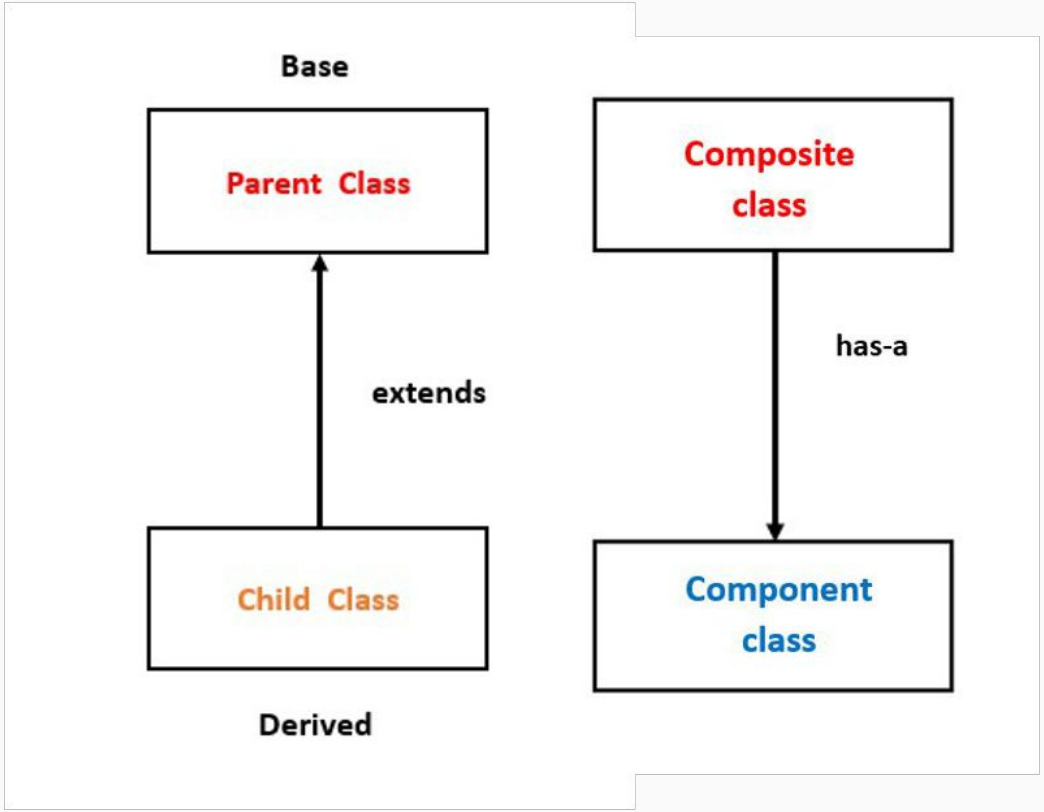
- We instantiate a file reader that we need in the DataReader, which now gets the capabilities to read parquet files
- We can give new capabilities to the DataReader by instantiating a different type of object, e.g. csv
- The code becomes easier to change and test.

Dependency injection is a principle that helps to decrease coupling and increase cohesion.

High coupling. If the coupling is high it's like using superglue or welding. No easy way to disassemble.

High cohesion. High cohesion is like using screws. Quite easy to disassemble and reassemble in a different way. It is an opposite to high coupling.

Challenge: Refactor the example in order to increase the cohesion and decrease the coupling.



Inheritance Composition

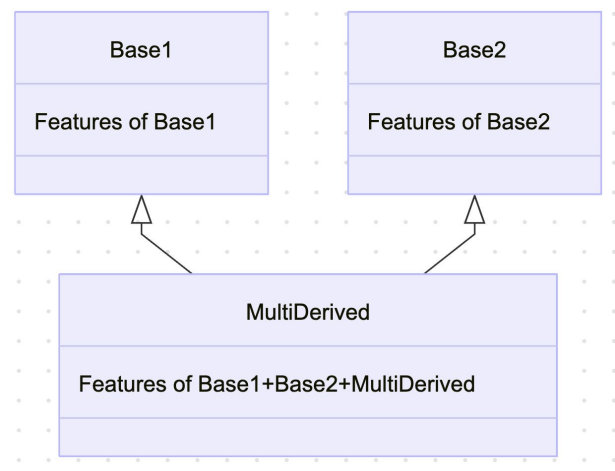
Multiple Inheritance

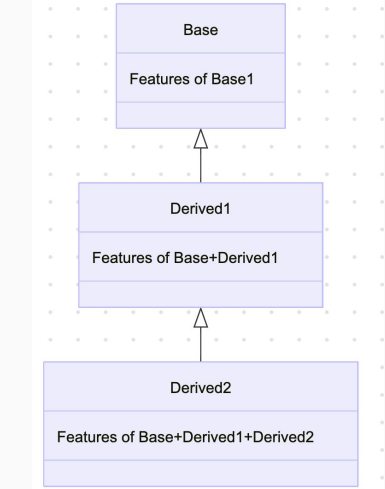
main.py

```
class Base1:
    pass
class Base2:
    pass
class MultiDerived(Base1, Base2):
    pass
```

main.py

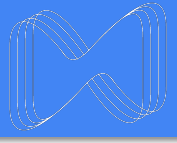
```
class Base:
    pass
class Derived1(Base):
    pass
class Derived2(Derived1):
    pass
```





- Any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching the same class twice.
- MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. The set of rules used to find this order is called Method Resolution Order (**MRO**).
- MRO of a class can be viewed as the __mro__ attribute or the mro() method. The former returns a tuple while the latter returns a list.

Abstract Classes



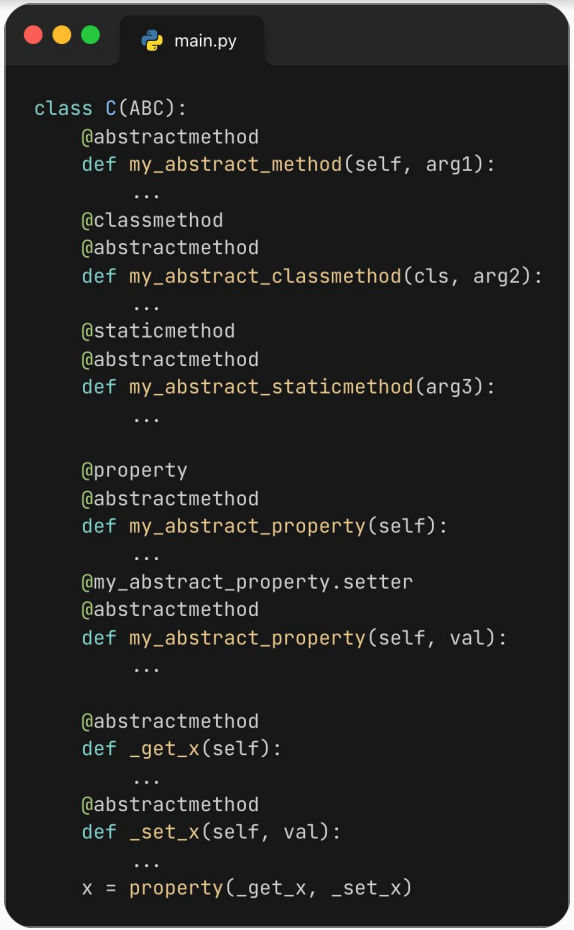
main.py

```python
from abc import ABC, abstractmethod

class FileReader(ABC):

    @abstractmethod
    def read(self):
        pass


class Parquet(FileReader):

    def read(self):
        print("reading a parquet file")


class Csv(FileReader):

    def __init__(self):
        print("We created a new csv class")

    def read(self):
        print("reading a csv file")

p = Parquet()
c = Csv()

p.read()
c.read()
```

An abstract class

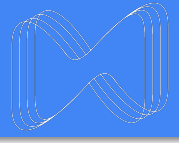
- a blueprint for other classes
- it allows you to create a set of methods that must be created within any child classes built from the abstract class
- a class which contains one or more abstract methods is called an abstract class
- an abstract method is a method that has a declaration but does not have an implementation
- By default, Python does not provide abstract classes
- Python comes with a module that provides the base for defining Abstract Base Classes(ABC) and that module name is ABC https://docs.python.org/3/library/abc.html

Abstract Classes



- The ABC class ensures that C cannot be instantiated directly.
- Any subclass of C **must** implement all abstract methods and properties.
- **@abstractmethod**: Requires the subclass to provide an implementation.
- @classmethod, @staticmethod, and @property can also be abstract.
- **@abstractmethod, x.setter -** Ensures subclasses **must** define both getter and setter.

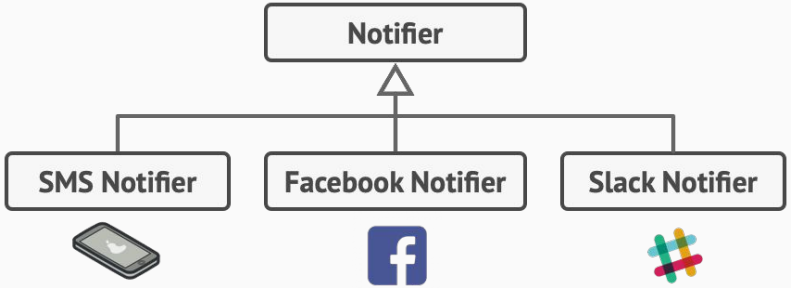
Interfaces

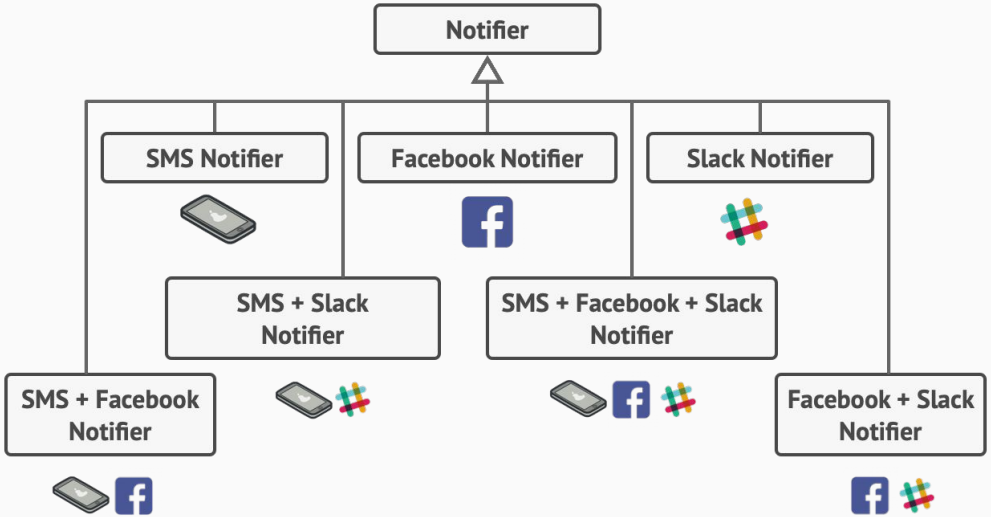


- An abstract class allows you to create functionality that subclasses can implement or override. An interface only allows you to define functionality, not implement it.
- Python doesn't have (and doesn't need) a formal Interface contract
- For example, Java uses interfaces because it doesn't have multiple inheritance
- Python has multiple inheritance
- For extra points examine how to design interfaces in python https://realpython.com/python-interface/

Decorator – General presentation

Problem: How can we improve the existing objects, enhance the response of a component, how can we support multiple optional behaviors, all these without changing too much their intended functionality.

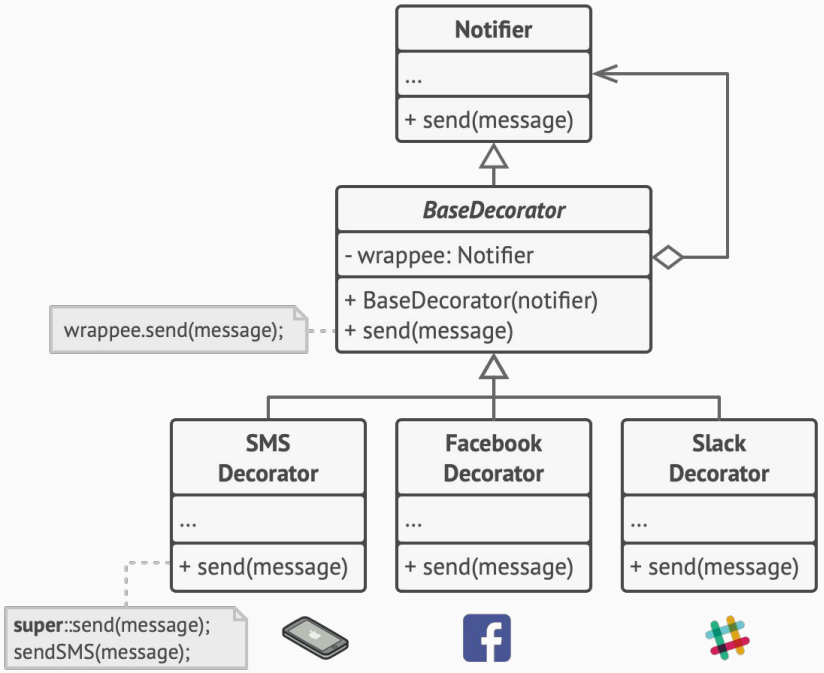




Decorator – General presentation

Solution: Decorator

- a suitable alternative to multiple inheritance. We prefer “**Composition over inheritance**”
- the alternative nickname of the decorator is “**Wrapper**”, that clearly expresses the main idea of the pattern
- A wrapper is an object that can be **linked with** some **target object**, it contains the same set of methods and delegates to it all requests it receives. However, the wrapper may alter the result by doing something either before or after it passes the request to the target.

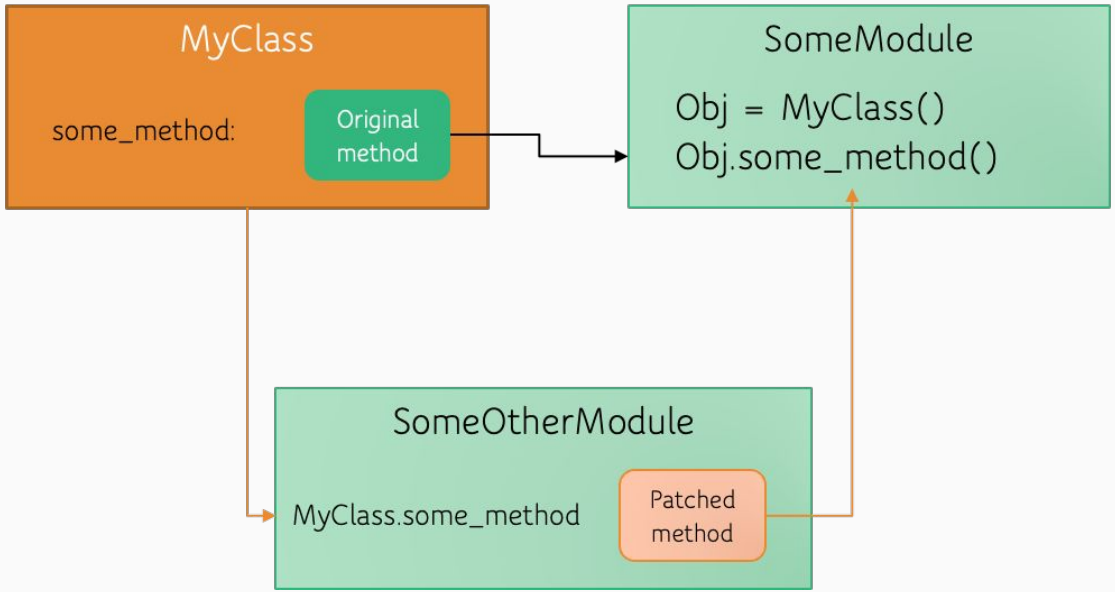


Monkey patching

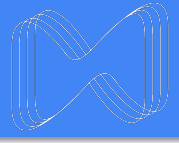
- dynamic (run-time) modifications of a class or module
- we can do this because Python classes are **mutable**, and methods are just attributes of the class
- we can even replace classes and functions in a module in exactly the same way

Cautions

- Once “monkey-patched”, the code will use the “patched” version everywhere, so beware. This can be good or bad!
- If there is already a variable pointing to the original method, this will not change and the original is still used.
- These effects might create confusion while reading the codebase



The “Decorator Pattern” ≠ Python “decorators”!



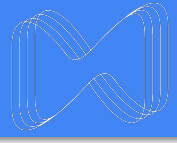
Decorator Pattern - the decorator pattern is a design pattern that allows behavior to be added to an existing object dynamically. The decorator pattern can be used to extend (decorate) the functionality of a certain object at run-time, independently of other instances of the same class

Decorators in Python - Python decorators are not an implementation of the decorator pattern. Python decorators add functionality to functions and methods at definition time, and thus are a higher-level construct than decorator-pattern classes.

Use-cases

- Log method calls that would normally work silently
- Perform extra setup or cleanup around a method
- Pre-process method arguments
- Post-process return values
- Forbid actions that the wrapped object would normally allow

Python Decorators

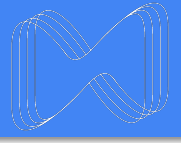


- A function in Python can return other function
- Functions in python can be nested
- There is a special character “@” that symbolizes the decorated functions
- A function supports multiple decorators
- functools.wraps() standard python function

main.py

```python
def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__} with {args} {kwargs}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@log_calls
def add(a, b):
    return a + b

# Example usage
add(3, 5)
```

ChatWrap



ChatWrap - A Lightweight Python SDK for AI Chat APIs

Implement a lightweight Python SDK that provides a unified interface for interacting with AI chat APIs such as OpenAI's ChatGPT, Ollama, and LMStudio. It simplifies API calls, supports streaming responses with callbacks, and offers essential customization options.

Here we practice Python packaging, project structuring, and OOP concepts while building a practical and reusable library.

Project Goals

- **Unified API Wrapper**: A single interface for multiple chat APIs.
- **Streaming Support**: Enable real-time responses with callbacks.
- **Customization**: Allow setting parameters like `temperature`, `max_tokens`, etc.
- **Utility Functions**:
  - Token counting.
- **Command-Line Interface (CLI)**:
  - Send prompts and receive responses from the terminal.
  - Select models and enable streaming.
- **Logging & Debugging**:
  - Debug mode for detailed logs.
  - Log API requests and responses for troubleshooting.

https://platform.openai.com/docs/api-reference/making-requests

Possible structure

main.py

```
chatwrap/
├── chatwrap/
│   ├── __init__.py
│   ├── client.py            # Main API wrapper for OpenAI, Ollama, LMStudio
│   ├── cli.py               # Command-line interface
│   ├── utils.py             # Utility functions (token counting, JSON mode, etc.)
│   └── logger.py            # Logging utilities
├── tests/
│   ├── test_client.py       # Tests for API client
│   └── test_cli.py          # Tests for CLI functionality
├── setup.py                 # Packaging setup
├── pyproject.toml           # Modern packaging configuration
├── README.md                # Documentation
├── LICENSE                  # License file
└── .gitignore               # Git ignores
```