Design Patterns in Modern Python and JavaScript

Design patterns are general, reusable solutions to common software design problems. They provide a standard terminology and best practices for writing clean, maintainable, and scalable code. This write-up explores some core design patterns and how they are implemented in **Python** and **JavaScript**.

1. Singleton Pattern

Ensures a class has only one instance and provides a global point of access.

Python:

```
class Singleton:
    _instance = None

    def __new__(cls):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(cls)
        return cls._instance
```

JavaScript:

```
const Singleton = (function () {
  let instance;
  function createInstance() {
    return { id: Date.now() };
  }
  return {
    getInstance: function () {
      if (!instance) instance = createInstance();
      return instance;
    }
  };
})();
```

3. Observer Pattern

Defines a one-to-many dependency so that when one object changes state, all its dependents are notified.

Python:

```python
class Subject:
    def __init__(self):
        self._observers = []

    def notify(self, message):
        for obs in self._observers:
            obs.update(message)

    def subscribe(self, observer):
        self._observers.append(observer)

class Observer:
    def update(self, message):
        print(f"Received: {message}")
```

JavaScript:

```javascript
class Subject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  notify(msg) {
    this.observers.forEach(observer => observer(msg));
  }
}
```

5. Strategy Pattern

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

Python:

```python
class Strategy:
    def execute(self, data):
        pass

class ConcreteStrategyA(Strategy):
    def execute(self, data):
        return sorted(data)

class Context:
    def __init__(self, strategy):
        self.strategy = strategy

    def do_something(self, data):
        return self.strategy.execute(data)
```

JavaScript:

```javascript
class StrategyA {
  execute(data) {
    return data.sort();
  }
}
```

```
class Context {
  constructor(strategy) {
    this.strategy = strategy;
  }

  executeStrategy(data) {
    return this.strategy.execute(data);
  }
}
```

Conclusion

Understanding and applying design patterns in Python and JavaScript empowers developers to write robust, scalable, and cleaner applications. Whether you're developing backend APIs, frontend interfaces, or microservices, these patterns help align your codebase with proven software engineering practices.