Behavourial

Observer / PubSub

It's common for different components of an app to respond to events or state changes, but how can we communicate these events?
The Observer pattern is a popular solution. We have a Subject (aka Publisher) which will be the source of events. And we could have multiple Observers (aka Subscribers) which will recieve events from the Subject in realtime.

```
class YoutubeChannel:
    def __init__(self, name):
        self.name = name
        self.subscribers = []

    def subscribe(self, sub):
        self.subscribers.append(sub)

    def notify(self, event):
        for sub in self.subscribers:
            sub.sendNotification(self.name, event)

from abc import ABC, abstractmethod

class YoutubeSubscriber(ABC):
    @abstractmethod
    def sendNotification(self, event):
        pass

class YoutubeUser(YoutubeSubscriber):
    def __init__(self, name):
        self.name = name

    def sendNotification(self, channel, event):
        print(f"User {self.name} received notification from {channel}: {event}")
```

```
channel = YoutubeChannel("neetcode")

channel.subscribe(YoutubeUser("sub1"))
channel.subscribe(YoutubeUser("sub2"))
channel.subscribe(YoutubeUser("sub3"))

channel.notify("A new video released")
```

Copy

In this case we have multiple Subscribers listening to a single published. But users could also be subscribed to multiple channels.
Since the Publishers & Subscribers don't have to worry about each others' implementations, they are loosely coupled.

> `User sub1 received notification from neetcode: A new video released`
>
> `User sub2 received notification from neetcode: A new video released`
>
> `User sub3 received notification from neetcode: A new video released`

Iterator

Many objects in python have built-in iterators. That's why we can conveniently iterate through an array using the key word `in`.

```
myList = [1, 2, 3]
for n in myList:
    print(n)
```

Copy

Output

> `1`
>
> `2`

```
3
```

For more complex objects, like Linked Lists or Binary Search Trees, we can define our own iterators.

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.next = None

class LinkedList:
    def __init__(self, head):
        self.head = head
        self.cur = None

    # Define Iterator
    def __iter__(self):
        self.cur = self.head
        return self

    # Iterate
    def __next__(self):
        if self.cur:
            val = self.cur.val
            self.cur = self.cur.next
            return val
        else:
            raise StopIteration

# Initialize LinkedList
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
myList = LinkedList(head)

# Iterate through LinkedList
for n in myList:
    print(n)
```

Copy

Output

```
1


2

3
```

Strategy

A Class may have different behaviour, or invoke a different method based on something we define (i.e. a Strategy). For example, we can filter an array by removing positive values; or we could filter it by removing all odd values. These are two filtering strategies we could implement, but we could add many more.

```
from abc import ABC, abstractmethod

class FilterStrategy(ABC):

    @abstractmethod
    def removeValue(self, val):
        pass

class RemoveNegativeStrategy(FilterStrategy):

    def removeValue(self, val):
        return val < 0

class RemoveOddStrategy(FilterStrategy):

    def removeValue(self, val):
        return abs(val) % 2


class Values:
    def __init__(self, vals):
        self.vals = vals
```

```
    def filter(self, strategy):
        res = []
        for n in self.vals:
            if not strategy.removeValue(n):
                res.append(n)
        return res

values = Values([-7, -4, -1, 0, 2, 6, 9])

print(values.filter(RemoveNegativeStrategy()))
print(values.filter(RemoveOddStrategy()))
```

Copy

Output

> [0, 2, 6, 9]
>
> [-4, 0, 2, 6]

A common alternative to this pattern is to simply pass in an inline / lambda function, which allows us to extend the behaviour of a method or class.

State Machines

When to use it?

State management is often part of a framework already, so building from scratch is rare. Different classes such as below can denote different states, but more pythonic way is to use different modules and functions, where each module can denote a different state. Can be used in such example like a "simple" and "advanced" mode of a GUI.

```
from dataclasses import dataclass
from typing import Protocol



class DocumentState(Protocol):
    def edit(self):
```

```
        ...

    def review(self):
        ...

    def finalize(self):
        ...


class DocumentContext(Protocol):
    content: list[str]

    def set_state(self, state: DocumentState) -> None:
        ...

    def edit(self):
        ...

    def review(self):
        ...

    def finalize(self):
        ...

    def show_content(self):
        ...


@dataclass
class Draft:
    document: DocumentContext

    def edit(self):
        print("Editing the document...")
        self.document.content.append("Edited content.")

    def review(self):
        print("The document is now under review.")
        self.document.set_state(Reviewed(self.document))
```

```
    def finalize(self):
        print("You need to review the document before finalizing.")


@dataclass
class Reviewed:
    document: DocumentContext

    def edit(self):
        print("The document is under review, cannot edit now.")

    def review(self):
        print("The document is already reviewed.")

    def finalize(self):
        print("Finalizing the document...")
        self.document.set_state(Finalized(self.document))


@dataclass
class Finalized:
    document: DocumentContext

    def edit(self):
        print("The document is finalized. Editing is not allowed.")

    def review(self):
        print("The document is finalized. Review is not possible.")

    def finalize(self):
        print("The document is already finalized.")


class Document:
    def __init__(self):
        self.state: DocumentState = Draft(self)
        self.content: list[str] = []

    def set_state(self, state: DocumentState):
        self.state = state
```

```
    def edit(self):
        self.state.edit()

    def review(self):
        self.state.review()

    def finalize(self):
        self.state.finalize()

    def show_content(self):
        print("Document content:", " ".join(self.content))


def main() -> None:
    document = Document()

    document.edit()  # Expected: "Editing the document..."
    document.show_content()  # Expected: "Document content: Edited content."
    document.finalize()  # Expected: "You need to review the document before finalizing."
    document.review()  # Expected: "The document is now under review."
    document.edit()  # Expected: "The document is under review, cannot edit now."
    document.finalize()  # Expected: "Finalizing the document..."
    document.edit()  # Expected: "The document is finalized. Editing is not allowed."


if __name__ == "__main__":
    main()
```

Revision #6
Created 14 February 2024 22:52:20 by victor
Updated 5 March 2024 23:44:15 by victor