from memory import MemoryCocoon, LivingMemoryKernel, MemoryItem

class Agent:

"""

Represents a cognitive agent with its own memory and state.

Attributes:

name (str): Identifier for the agent.

memory\_cocoon (MemoryCocoon): Container for the agent's memories.

memory\_kernel (LivingMemoryKernel): Kernel to manage memory retrieval for this agent.

intent (float): A scalar representing the agent's current intent or drive (0.0 to 1.0 range).

base\_intent (float): The agent's baseline intent (tendency) when not influenced by others or memories.

emotion\_valence (float): Current emotional valence of the agent (-1.0 to 1.0).

emotion\_tag (str): Current emotional state label (e.g. "happy", "neutral"). This is a qualitative descriptor

corresponding to the emotion\_valence.

entangled\_with (list[Agent]): List of other agents this agent is currently entangled (synchronized) with.

last\_recalled\_memory (MemoryItem or None): The last memory item retrieved by this agent (if any).

"""

def \_\_init\_\_(self, name: str, memory\_kernel: LivingMemoryKernel = None,

initial\_intent: float = 0.5, emotion\_tag: str = "neutral", emotion\_valence: float = 0.0,

base\_intent: float = None):

self.name = name

# Initialize memory subsystem

if memory\_kernel is None:

# Create a new MemoryCocoon and kernel if not provided

self.memory\_cocoon = MemoryCocoon()

self.memory\_kernel = LivingMemoryKernel(self.memory\_cocoon)

else:

# Use the provided memory kernel (shared or pre-configured)

self.memory\_kernel = memory\_kernel

# Ensure we have a reference to its cocoon for direct memory inspection if needed

self.memory\_cocoon = memory\_kernel.memory\_cocoon

# Initialize state

self.intent = initial\_intent

self.base\_intent = base\_intent if base\_intent is not None else initial\_intent # default baseline is the initial value

self.emotion\_tag = emotion\_tag

self.emotion\_valence = emotion\_valence

self.entangled\_with = []

self.last\_recalled\_memory = None

def add\_memory(self, content: str, emotion\_tag: str, intensity: float = 1.0):

"""

Add a memory of an experience to this agent's MemoryCocoon.

"""

return self.memory\_kernel.store\_memory(content, emotion\_tag, intensity)

def recall\_memory(self):

"""

Retrieve the most resonant memory for the agent's current emotional state.

Updates last\_recalled\_memory and returns the MemoryItem if found.

"""

mem = self.memory\_kernel.retrieve\_for\_agent(self)

self.last\_recalled\_memory = mem

return mem

def modulate\_intent(self, delta: float):

"""

Adjust the agent's intent by a certain delta (which may be positive or negative).

Keeps intent within [0.0, 1.0] bounds.

"""

self.intent += delta

if self.intent < 0.0:

self.intent = 0.0

if self.intent > 1.0:

self.intent = 1.0

def adjust\_emotion(self, new\_valence: float):

"""

Adjust the agent's emotional state towards a new valence.

We blend the current valence with the new\_valence to avoid abrupt changes,

and update the emotion\_tag qualitatively.

"""

# Blend current emotion with the new influence (simple average for now)

self.emotion\_valence = (self.emotion\_valence + new\_valence) / 2.0

# Update the emotion\_tag qualitatively based on the new valence

if self.emotion\_valence > 0.1:

# If positive valence, mark as generic "positive" if not a specific tag

# (In a more complex model, we might retain specific emotions, but here we use broad categories)

self.emotion\_tag = "positive" if self.emotion\_valence >= 0.5 else "slightly positive"

elif self.emotion\_valence < -0.1:

self.emotion\_tag = "negative" if self.emotion\_valence <= -0.5 else "slightly negative"

else:

self.emotion\_tag = "neutral"

def \_\_repr\_\_(self):

return f"Agent({self.name}: intent={self.intent:.2f}, emotion\_valence={self.emotion\_valence:.2f}, " \

f"emotion\_tag={self.emotion\_tag})"