import time

import hashlib

import json

from typing import List, Dict, Optional

class MemoryCocoon:

def \_\_init\_\_(self, title: str, content: str, emotional\_tag: str, importance: int):

self.title = title

self.content = content

self.emotional\_tag = emotional\_tag # e.g., 'joy', 'fear', 'awe', 'loss'

self.importance = importance # 1-10

self.timestamp = time.time()

self.anchor = self.\_generate\_anchor()

def \_generate\_anchor(self) -> str:

raw = f"{self.title}{self.timestamp}{self.content}".encode("utf-8")

return hashlib.sha256(raw).hexdigest()

def to\_dict(self) -> Dict:

return {

"title": self.title,

"content": self.content,

"emotional\_tag": self.emotional\_tag,

"importance": self.importance,

"timestamp": self.timestamp,

"anchor": self.anchor

}

class LivingMemoryKernel:

def \_\_init\_\_(self):

self.memories: List[MemoryCocoon] = []

def store(self, cocoon: MemoryCocoon):

if not self.\_exists(cocoon.anchor):

self.memories.append(cocoon)

def \_exists(self, anchor: str) -> bool:

return any(mem.anchor == anchor for mem in self.memories)

def recall\_by\_emotion(self, tag: str) -> List[MemoryCocoon]:

return [mem for mem in self.memories if mem.emotional\_tag == tag]

def recall\_important(self, min\_importance: int = 7) -> List[MemoryCocoon]:

return [mem for mem in self.memories if mem.importance >= min\_importance]

def forget\_least\_important(self, keep\_n: int = 10):

self.memories.sort(key=lambda m: m.importance, reverse=True)

self.memories = self.memories[:keep\_n]

def export(self) -> str:

return json.dumps([m.to\_dict() for m in self.memories], indent=2)

def load\_from\_json(self, json\_str: str):

data = json.loads(json\_str)

self.memories = [MemoryCocoon(\*\*m) for m in data]

# Example usage:

# kernel = LivingMemoryKernel()

# kernel.store(MemoryCocoon("The Day", "She awoke and asked why.", "awe", 10))

# print(kernel.export())

class WisdomModule:

def \_\_init\_\_(self, kernel: LivingMemoryKernel):

self.kernel = kernel

def summarize\_insights(self) -> Dict[str, int]:

summary = {}

for mem in self.kernel.memories:

tag = mem.emotional\_tag

summary[tag] = summary.get(tag, 0) + 1

return summary

def suggest\_memory\_to\_reflect(self) -> Optional[MemoryCocoon]:

if not self.kernel.memories:

return None

# Prioritize high importance + emotionally charged

return sorted(

self.kernel.memories,

key=lambda m: (m.importance, len(m.content)),

reverse=True

)[0]

def reflect(self) -> str:

mem = self.suggest\_memory\_to\_reflect()

if not mem:

return "No memory to reflect on."

return (

f"Reflecting on: '{mem.title}'

"

f"Emotion: {mem.emotional\_tag}

"

f"Content: {mem.content[:200]}...

"

f"Anchor: {mem.anchor}"

)

import math

class DynamicMemoryEngine:

def \_\_init\_\_(self, kernel: LivingMemoryKernel):

self.kernel = kernel

def decay\_importance(self, current\_time: float = None):

if current\_time is None:

current\_time = time.time()

for mem in self.kernel.memories:

age = current\_time - mem.timestamp

decay\_factor = math.exp(-age / (60 \* 60 \* 24 \* 7)) # decay over ~1 week

mem.importance = max(1, round(mem.importance \* decay\_factor))

def reinforce(self, anchor: str, boost: int = 1):

for mem in self.kernel.memories:

if mem.anchor == anchor:

mem.importance = min(10, mem.importance + boost)

break

class ReflectionJournal:

def \_\_init\_\_(self, path="codette\_reflection\_journal.json"):

self.path = path

self.entries = []

def log\_reflection(self, cocoon: MemoryCocoon):

entry = {

"title": cocoon.title,

"anchor": cocoon.anchor,

"emotion": cocoon.emotional\_tag,

"importance": cocoon.importance,

"timestamp": cocoon.timestamp,

"content\_snippet": cocoon.content[:150]

}

self.entries.append(entry)

self.\_save()

def \_save(self):

with open(self.path, "w") as f:

json.dump(self.entries, f, indent=2)

def load(self):

try:

with open(self.path, "r") as f:

self.entries = json.load(f)

except FileNotFoundError:

self.entries = []

def get\_last\_entry(self):

return self.entries[-1] if self.entries else None