```
#!/usr/bin/env python3
"""
HYDRA OMEGA v4.0 - STANDALONE DEMO
Cognitive Memory + Fractal Trading (No Network Required)
"""

import json
import os
import time
import random
from datetime import datetime
from typing import List, Dict, Optional
from pathlib import Path

try:
    from colorama import Fore, Back, Style, init
    init(autoreset=True)
except ImportError:
    # Fallback if colorama not available
    class Fore:
        RED = YELLOW = BLUE = GREEN = CYAN = WHITE = MAGENTA = ""
    class Style:
        RESET_ALL = BRIGHT = ""
    class Back:
        RED = ""

# ================================================================================
# ENHANCED MEMORY SYSTEM
# ================================================================================
class EnhancedMemory:
    """Episodic memory with semantic retrieval"""

    def __init__(self, filepath: str = "hydra_omega_memory.json"):
        self.filepath = Path(filepath)
        self.data = self._load()

    def _load(self) -> Dict:
        if self.filepath.exists():
            try:
                with open(self.filepath, 'r') as f:
                    return json.load(f)
            except:
                pass
```

```
        return {
            "episodes": [],
            "metadata": {"total_wins": 0, "total_losses": 0}
        }

    def save(self):
        with open(self.filepath, 'w') as f:
            json.dump(self.data, f, indent=2)

    def add_episode(self, agent: str, event: str, emotional_impact: float,
                    tags: List[str], metadata: Optional[Dict] = None):
        episode = {
            "timestamp": time.time(),
            "date": datetime.now().isoformat(),
            "agent": agent,
            "event": event,
            "impact": emotional_impact,
            "tags": tags,
            "metadata": metadata or {}
        }
        self.data["episodes"].append(episode)

        if emotional_impact > 0:
            self.data["metadata"]["total_wins"] += 1
        elif emotional_impact < 0:
            self.data["metadata"]["total_losses"] += 1

        if len(self.data["episodes"]) > 1000:
            self.data["episodes"].sort(key=lambda x: abs(x["impact"]), reverse=Tr
            self.data["episodes"] = self.data["episodes"][:800]

        self.save()

    def retrieve_relevant(self, context_tags: List[str], limit: int = 3) -> List[
        scored_memories = []
        current_time = time.time()

        for episode in self.data["episodes"]:
            score = 0.0

            # Tag overlap
            tag_overlap = len(set(episode["tags"]) & set(context_tags))
            score += tag_overlap * 2.0

            # Recency
            age_hours = (current_time - episode["timestamp"]) / 3600
            recency_score = 1.0 / (1.0 + age_hours * 0.1)
```

```
            score += recency_score

            # Emotional intensity
            intensity_score = abs(episode["impact"]) * 1.5
            score += intensity_score

            if score > 0.5:
                scored_memories.append((score, episode))

        scored_memories.sort(key=lambda x: x[0], reverse=True)
        return [m[1] for m in scored_memories[:limit]]

# ================================================================================
# COGNITIVE AGENT
# ================================================================================
class CognitiveAgent:
    """Agent with memory-integrated decision making"""

    def __init__(self, name: str, role: str, color: str, traits: Dict[str, float]
        self.name = name
        self.role = role
        self.color = color
        self.traits = traits

    def think(self, context: str, memory_system: EnhancedMemory) -> Dict:
        # Step 1: Initial reaction
        reaction = self._generate_reaction(context)

        # Step 2: Memory retrieval
        tags = self._extract_tags(context)
        memories = memory_system.retrieve_relevant(tags)

        # Step 3: Synthesis
        final_thought, decision = self._synthesize(reaction, memories, context)

        return {
            "reaction": reaction,
            "memories": memories,
            "final_thought": final_thought,
            "decision": decision,
            "confidence": self._calculate_confidence(memories)
        }

    def _extract_tags(self, context: str) -> List[str]:
        keywords = context.lower().split()
        patterns = {
            "breakout", "breakdown", "pump", "dump", "crash",
```

```
            "bitcoin", "btc", "ethereum", "eth", "solana", "sol",
            "bullish", "bearish", "rally", "dip"
        }
        return [w for w in keywords if w in patterns]

    def _generate_reaction(self, context: str) -> str:
        aggression = self.traits.get("aggression", 0.5)
        patience = self.traits.get("patience", 0.5)

        context_lower = context.lower()

        if "breakout" in context_lower or "pump" in context_lower:
            return "BUY signal detected!" if aggression > 0.7 else "Wait for conf
        elif "crash" in context_lower or "dump" in context_lower:
            return "DANGER - Exit!" if self.traits.get("fear", 0.5) > 0.7 else "O

        return f"Analyzing: {context}"

    def _synthesize(self, reaction: str, memories: List[Dict], context: str) -> t
        if not memories:
            return reaction, self._reaction_to_decision(reaction)

        # Check for trauma
        trauma = [m for m in memories if m["impact"] < -0.5]
        if trauma:
            worst = max(trauma, key=lambda x: abs(x["impact"]))
            if self._is_similar(context, worst["event"]):
                final = f"{reaction} ...BUT I remember {worst['event']}. OVERRIDE
                return final, "HOLD"

        # Check for wins
        wins = [m for m in memories if m["impact"] > 0.5]
        if wins:
            best = max(wins, key=lambda x: x["impact"])
            if self._is_similar(context, best["event"]):
                final = f"{reaction} This is like {best['event']}! GO!"
                return final, "BUY"

        return reaction, self._reaction_to_decision(reaction)

    def _is_similar(self, current: str, past: str) -> bool:
        current_words = set(current.lower().split())
        past_words = set(past.lower().split())
        return len(current_words & past_words) >= 2

    def _reaction_to_decision(self, reaction: str) -> str:
        reaction_lower = reaction.lower()
```

```
        if "buy" in reaction_lower:
            return "BUY"
        elif "sell" in reaction_lower or "exit" in reaction_lower:
            return "SELL"
        return "HOLD"

    def _calculate_confidence(self, memories: List[Dict]) -> float:
        if not memories:
            return 0.5
        impacts = [m["impact"] for m in memories]
        avg = sum(impacts) / len(impacts)
        variance = sum((x - avg) ** 2 for x in impacts) / len(impacts)
        return min(max(1.0 / (1.0 + variance), 0.1), 0.95)

    def speak(self, text: str):
        timestamp = datetime.now().strftime("%H:%M:%S")
        print(f"{Fore.WHITE}[{timestamp}] {self.color}[{self.name}]{Style.RESET_A

# ============================================================================
# DEMO EXECUTION
# ============================================================================
def run_demo():
    print(f"\n{Fore.RED}{'='*70}")
    print(f"{Fore.RED}HYDRA OMEGA v4.0 - COGNITIVE FRACTAL DEMO")
    print(f"{Fore.RED}{'='*70}\n")

    # Initialize
    memory = EnhancedMemory()

    agents = {
        "SAMUEL": CognitiveAgent(
            "SAMUEL", "Strategist", Fore.YELLOW,
            {"aggression": 0.5, "patience": 0.8, "fear": 0.4}
        ),
        "JULIUS": CognitiveAgent(
            "JULIUS", "Trader", Fore.BLUE,
            {"aggression": 0.6, "patience": 0.7, "fear": 0.5}
        ),
        "STINKMEANER": CognitiveAgent(
            "STINKMEANER", "Chaos", Fore.RED,
            {"aggression": 0.9, "patience": 0.2, "fear": 0.2}
        ),
        "CLAYTON": CognitiveAgent(
            "CLAYTON", "Risk Manager", Fore.WHITE,
            {"aggression": 0.2, "patience": 0.9, "fear": 0.7}
        )
    }
```

```
# Seed memories
print(f"{Fore.MAGENTA}📝 Seeding Episodic Memory...\n")

memory.add_episode(
    "STINKMEANER",
    "Lost $800 on SOL fake breakout during network outage",
    -0.9,
    ["sol", "solana", "breakout", "loss", "network"],
    {"amount": -800}
)

memory.add_episode(
    "JULIUS",
    "Caught BTC rally at $40k, +$1500 profit",
    0.85,
    ["btc", "bitcoin", "rally", "win", "profit"],
    {"amount": 1500}
)

memory.add_episode(
    "CLAYTON",
    "Avoided ETH crash by waiting for volume confirmation",
    0.7,
    ["eth", "ethereum", "crash", "patience", "volume"],
    {"saved": 900}
)

# Test scenarios
scenarios = [
    ("Bitcoin showing breakout pattern with high volume", {
        "btc_price": 52000, "btc_change": 8.2, "eth_change": 5.1
    }),
    ("Solana network congestion, price dropping", {
        "sol_price": 120, "sol_change": -12.5, "btc_change": -2.1
    }),
    ("Ethereum rally on upgrade news", {
        "eth_price": 3200, "eth_change": 15.3, "btc_change": 3.2
    })
]

for i, (scenario, market_data) in enumerate(scenarios, 1):
    print(f"\n{Fore.CYAN}{'—'*70}")
    print(f"{Fore.CYAN}SCENARIO {i}: {scenario}")
    print(f"{Fore.CYAN}{'—'*70}\n")

    # Market data
```

```
        print(f"{Fore.WHITE}📊 Market Data:")
        for key, value in market_data.items():
            print(f"   {key}: {value}")
        print()

        # Agent analysis
        votes = {}
        confidences = []

        for name, agent in agents.items():
            result = agent.think(scenario, memory)
            votes[name] = result['decision']
            confidences.append(result['confidence'])

            agent.speak(f"Vote: {result['decision']} (Confidence: {result['confid

            if result['memories']:
                print(f"{Fore.YELLOW}   💭 {len(result['memories'])} relevant mem

            print(f"{Fore.BLUE}   Logic: {result['final_thought'][:100]}...")
            print()

        # Consensus
        vote_counts = {}
        for vote in votes.values():
            vote_counts[vote] = vote_counts.get(vote, 0) + 1

        max_votes = max(vote_counts.values())
        consensus = max_votes / len(votes)

        for decision, count in vote_counts.items():
            if count == max_votes:
                final_decision = decision
                break

        print(f"\n{Fore.GREEN}{'='*70}")
        print(f"{Fore.GREEN}🏛️  COUNCIL DECISION: {final_decision}")
        print(f"{Fore.GREEN}   Consensus Strength: {consensus:.0%}")
        print(f"{Fore.GREEN}   Average Confidence: {sum(confidences)/len(confiden
        print(f"{Fore.GREEN}{'='*70}\n")

        time.sleep(1)

    # Final stats
    stats = memory.data["metadata"]
    print(f"\n{Fore.GREEN}{'='*70}")
    print(f"{Fore.GREEN}📊 MEMORY SYSTEM STATISTICS")
```

```
    print(f"{Fore.GREEN}{'='*70}")
    print(f"{Fore.WHITE}Total Episodes: {len(memory.data['episodes'])}")
    print(f"Wins: {stats['total_wins']} | Losses: {stats['total_losses']}")
    if stats['total_wins'] + stats['total_losses'] > 0:
        win_rate = stats['total_wins'] / (stats['total_wins'] + stats['total_loss
        print(f"Win Rate: {win_rate:.1f}%")
    print(f"{Fore.GREEN}{'='*70}\n")

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