

```

import uuid
import random
from typing import List, Dict, Any, Optional
import numpy as np

# -----
# Supporting Classes (minimal versions for Odin)
# -----


class OctonionProjection:
    def __init__(self, coeffs=None):
        if coeffs is None:
            self.coeffs = np.zeros(8)
        else:
            self.coeffs = np.array(coeffs, dtype=float)

    @staticmethod
    def random_from_tokens(tokens: List[str]):
        vec = np.zeros(8)
        for i, tok in enumerate(tokens[:8]):
            vec[i] = (hash(tok) % 1000) / 1000.0
        return OctonionProjection(vec)

    def signature(self):
        return tuple(self.coeffs.tolist())

    def magnitude(self):
        return float(np.linalg.norm(self.coeffs))

    def scaled(self, s: float):
        return OctonionProjection(self.coeffs * s)

    def __add__(self, other):
        return OctonionProjection(self.coeffs + other.coeffs)


class ConceptNode:
    def __init__(self, name, value, tags=None):
        self.name = name
        self.value = float(value)
        self.tags = tags or []
        self.real = value
        # create octonion projection from node name + tags
        tokens = [name] + self.tags

```

```

self.octonion = OctonionProjection.random_from_tokens(tokens)

class ZCore:
    def __init__(self, dimension: int):
        self.dimension = dimension
        self.stability_score = 0.0
        self.octonion_field: List[OctonionProjection] = []

    def update_stability(self, delta: float):
        self.stability_score = max(0.0, min(1.0, self.stability_score + delta))

    def ready_for_next_dimension(self) -> bool:
        return self.stability_score >= 0.9

class Raven:
    def __init__(self, name, polarity):
        self.name = name
        self.polarity = polarity
        self.memory: List[ConceptNode] = []

    def emit(self, signature):
        tokens = [signature.get("x_axis", ""), signature.get("y_axis", ""), str(signature.get("value", 0))]
        node = ConceptNode(f"{self.name}_return", signature["value"], tags=[self.polarity])
        self.memory.append(node)
        return node

    def retrieve(self):
        recent = self.memory[:]
        self.memory = []
        return recent

class SyzygyEngine:
    def resolve(self, huginn_data, muninn_data):
        if not huginn_data or not muninn_data:
            return 0.0, []

        avg_h = sum(n.value for n in huginn_data) / len(huginn_data)
        avg_m = sum(n.value for n in muninn_data) / len(muninn_data)
        delta = -abs(avg_h - avg_m) * 0.1
        return delta, []

```

```

class NightUnweaver:
    def collapse(self, odin, huginn_data, muninn_data):
        all_data = huginn_data + muninn_data
        for node in all_data:
            odin.experiential_memory.append(node)

class IdentityField:
    def __init__(self):
        self.snapshots: List[OctonionProjection] = []

    def update(self, octproj: OctonionProjection):
        self.snapshots.append(octproj)

    def describe(self):
        return {
            "num_snapshots": len(self.snapshots),
            "last_signature": self.snapshots[-1].signature() if self.snapshots else None
        }

IDENTITY_FIELD = IdentityField()

# -----
#       THE ODIN CLASS (FULL VERSION)
# -----


class Odin:
    """
    Fully upgraded Odin engine with:
    - Dimensional Z-core behavior
    - Ravens Huginn/Muninn
    - Syzygy resolution & NightUnweaver integration
    - Sub-Odin spawning
    - NEW: R/C/H/O cognition pipeline (Real → Complex → Hyper → Octal)
    """

    def __init__(self, dimension: int = 1, parent_traits: Dict[str, float] = None,
                 identity_field: IdentityField = IDENTITY_FIELD):

        self.dimension = dimension
        self.id = uuid.uuid4().hex[:8]

        self.z = ZCore(dimension)

```

```

self.huginn = Raven("Huginn", "X")
self.muninn = Raven("Muninn", "Y")
self.syzygy = SyzygyEngine()
self.unweaver = NightUnweaver()

self.sub_odins: List["Odin"] = []
self.cycle_count = 0
self.experiential_memory: List[ConceptNode] = []

# Identity field (shared across Yggdrasil if needed)
self.identity_field = identity_field

# Traits
base = parent_traits or {
    "curiosity": 0.5,
    "sensitivity": 0.5,
    "integration_bias": 0.5,
}
self.traits = {
    k: min(1.0, max(0.0, v + 0.05*(dimension - 1)))
    for k, v in base.items()
}

# NEW: cognition trace
self.cog_trace: List[ConceptNode] = []

# -----
# Internal helpers
# -----


def _choose_axes(self):
    axes = ["x", "y"]
    random.shuffle(axes)
    return axes[0], axes[1]

def _make_becoming_signature(self):
    x, y = self._choose_axes()
    val = self.z.stability_score + float(self.dimension)
    return {"x_axis": x, "y_axis": y, "value": val}

# -----
# Dawn → Day → Dusk → Night
# -----

```

```

def dawn(self):
    sig = self._make_becoming_signature()
    self.huginn.emit(sig)
    self.muninn.emit(sig)

def day(self):
    # exploration is already done in emit phase
    pass

def dusk(self):
    huginn_data = self.huginn.retrieve()
    muninn_data = self.muninn.retrieve()
    stability_delta, contradictions = self.syzygy.resolve(huginn_data, muninn_data)
    self.z.update_stability(stability_delta)
    return huginn_data, muninn_data, stability_delta, contradictions

def night(self, huginn_data, muninn_data, stability_delta, contradictions):
    self.unweaver.collapse(self, huginn_data, muninn_data)
    if self.z.ready_for_next_dimension():
        self._spawn_next_odin()

def _spawn_next_odin(self):
    new_dim = self.dimension + 1
    child = Odin(new_dim, parent_traits=self.traits.copy(), identity_field=self.identity_field)
    self.sub_odins.append(child)
    # Reset stability to allow more cycles
    self.z.stability_score = 0.0

def cycle(self):
    self.dawn()
    self.day()
    huginn_data, muninn_data, stability_delta, contradictions = self.dusk()
    self.night(huginn_data, muninn_data, stability_delta, contradictions)
    self.cycle_count += 1

# -----
# NEW — Cognitive Layers (R / C / H / O)
# -----


def process_real(self, stimulus: Optional[Dict[str, Any]] = None) -> ConceptNode:
    x_axis, y_axis = self._choose_axes()
    base = self.z.stability_score + float(self.dimension)

    if stimulus is not None:

```

```

stim_val = (hash(str(sorted(stimulus.items())))) % 10000) / 1000.0
value = base + stim_val
tags = ["layer_R", "stimulus_bound", f"x_{x_axis}", f"y_{y_axis}"]
else:
    value = base
    tags = ["layer_R", "internal", f"x_{x_axis}", f"y_{y_axis}"]

node = ConceptNode(f"R_state_{self.id}", value, tags)
self.experiential_memory.append(node)
self.cog_trace.append(node)
return node

def process_complex(self, prior: Optional[ConceptNode] = None) -> ConceptNode:
    if prior is None:
        prior = self.cog_trace[-1] if self.cog_trace else None
    base_real = prior.real if prior else self.z.stability_score
    phase = (self.cycle_count % 8) / 8.0
    value = base_real + phase

    tags = ["layer_C", "phase_modulated"]
    if prior: tags.append(f"from_{prior.name}")

    node = ConceptNode(f"C_state_{self.id}", value, tags)
    self.experiential_memory.append(node)
    self.cog_trace.append(node)
    return node

def process_hyper(self, prior: Optional[ConceptNode] = None) -> ConceptNode:
    if prior is None:
        prior = self.cog_trace[-1] if self.cog_trace else None

    if prior:
        token = prior.name + "|" + "|".join(prior.tags)
        base_val = (hash(token) % 10000) / 1000.0
    else:
        base_val = self.z.stability_score

    node = ConceptNode(f"H_state_{self.id}", base_val, ["layer_H", "symbolic"])
    if prior: node.tags.append(f"from_{prior.name}")

    self.experiential_memory.append(node)
    self.cog_trace.append(node)
    return node

```

```

def process_octal(self, prior: Optional[ConceptNode] = None) -> ConceptNode:
    recent = self.cog_trace[-4:] if self.cog_trace else []
    if prior and prior not in recent:
        recent.append(prior)

    if not recent:
        oct_acc = OctonionProjection()
        mag = 0.0
    else:
        oct_acc = OctonionProjection()
        for n in recent:
            oct_acc = oct_acc + n.octonion
        mag = oct_acc.magnitude()
        if mag > 0:
            oct_acc = oct_acc.scaled(1.0 / mag)

    node = ConceptNode(f'O_state_{self.id}', mag, ["layer_O", "collapse"])
    node.octonion = oct_acc

    self.z.octonion_field.append(oct_acc)
    self.experiential_memory.append(node)
    self.cog_trace.append(node)

    # Update global identity when stable
    if self.z.stability_score >= 0.6:
        self.identity_field.update(oct_acc)

    return node

# -----
# Description
# -----
def describe(self, indent=0):
    pad = " " * indent
    print(f'{pad}Odin {self.id} (dimension={self.dimension})')
    print(f'{pad} cycles={self.cycle_count}, stability={self.z.stability_score:.3f}')
    print(f'{pad} traits={self.traits}')
    print(f'{pad} memory={len(self.experiential_memory)} items')
    print(f'{pad} sub-odins={len(self.sub_odins)})')

```