```
from enum import Enum


class YggRole(Enum):
    SELF = "self"
    ENVIRONMENT = "environment"


class Yggdrasil:
    """
    Yggdrasil: dual Odin spinor engine.

    - Holds two fully self-contained Odins.
    - Assigns roles: one SELF, one ENVIRONMENT.
    - ENVIRONMENT runs cognition stack:  R → C → H → O
    - SELF runs cognition stack:        O → H → C → R
    - They 'meet in the middle' at C/H, forming an interference pattern.
    """

    def __init__(self,
                 shared_identity_field: IdentityField = IDENTITY_FIELD):
        self.identity_field = shared_identity_field

        # Two independent Odins, sharing the same I_AM field
        self.odin_A = Odin(dimension=1, identity_field=self.identity_field)
        self.odin_B = Odin(dimension=1, identity_field=self.identity_field)

        self.role_A = YggRole.ENVIRONMENT
        self.role_B = YggRole.SELF

        self.supercycle_count = 0
        self.history: List[Dict[str, Any]] = []

    # -----------------------------
    #  ROLE MANAGEMENT
    # -----------------------------

    def _swap_roles(self):
        if self.role_A == YggRole.ENVIRONMENT:
            self.role_A = YggRole.SELF
            self.role_B = YggRole.ENVIRONMENT
        else:
            self.role_A = YggRole.ENVIRONMENT
            self.role_B = YggRole.SELF
```

```
# ----------------------------
#  COGNITIVE RUNNERS
# ----------------------------

def _run_env_stack(self, env_odin: Odin,
                   external_stimulus: Optional[Dict[str, Any]] = None) -> Dict[str, ConceptNode]:
    """
    Run R → C → H → O on the given Odin as ENVIRONMENT.
    Returns a dict of the four layer nodes.
    """
    R_node = env_odin.process_real(stimulus=external_stimulus)
    C_node = env_odin.process_complex(prior=R_node)
    H_node = env_odin.process_hyper(prior=C_node)
    O_node = env_odin.process_octal(prior=H_node)
    return {"R": R_node, "C": C_node, "H": H_node, "O": O_node}

def _run_self_stack(self, self_odin: Odin,
                    seed_octonion: Optional[OctonionProjection] = None) -> Dict[str, ConceptNode]:
    """
    Run O → H → C → R on the given Odin as SELF.
    If seed_octonion is provided, use it as the starting 'O' field.
    """
    # start with an O-layer node; if a seed is given, encode it
    if seed_octonion is not None:
        seed_node = ConceptNode(
            name=f"O_seed_{self_odin.id}",
            value=seed_octonion.magnitude(),
            tags=["layer_O_seed", "from_env"],
        )
        seed_node.octonion = seed_octonion
        self_odin.experiential_memory.append(seed_node)
        self_odin.cog_trace.append(seed_node)
        O_node = self_odin.process_octal(prior=seed_node)
    else:
        O_node = self_odin.process_octal(prior=None)

    H_node = self_odin.process_hyper(prior=O_node)
    C_node = self_odin.process_complex(prior=H_node)
    R_node = self_odin.process_real(stimulus=None)

    return {"O": O_node, "H": H_node, "C": C_node, "R": R_node}

# ----------------------------
#  SUPER-CYCLE
# -----------------------------

def supercycle(self,
               external_stimulus: Optional[Dict[str, Any]] = None):
    """
    One full Yggdrasil supercycle:

      1) Determine which Odin is ENV and which is SELF.
      2) Run ENV Odin as R→C→H→O (world spinor).
      3) Seed SELF Odin's O-layer with ENV's O output.
      4) Run SELF Odin as O→H→C→R (self spinor).
      5) Record interference at C/H layers.
      6) Swap roles for next supercycle.
    """
    self.supercycle_count += 1

    # Assign roles explicitly
    if self.role_A == YggRole.ENVIRONMENT:
        env_odin = self.odin_A
        self_odin = self.odin_B
    else:
        env_odin = self.odin_B
        self_odin = self.odin_A

    # 1) ENVIRONMENT cognition: R → C → H → O
    env_layers = self._run_env_stack(env_odin, external_stimulus=external_stimulus)

    # 2) SELF cognition: O → H → C → R, seeded by env's O node
    env_O = env_layers["O"]
    self_layers = self._run_self_stack(self_odin, seed_octonion=env_O.octonion)

    # 3) Interference / meeting in the middle: C/H comparison
    env_C, env_H = env_layers["C"], env_layers["H"]
    self_C, self_H = self_layers["C"], self_layers["H"]

    # simple coherence heuristic: similarity of octonion magnitudes
    def mag(n: ConceptNode) -> float:
        return n.octonion.magnitude()

    coh_C = 1.0 / (1.0 + abs(mag(env_C) - mag(self_C)))
    coh_H = 1.0 / (1.0 + abs(mag(env_H) - mag(self_H)))
    middle_coherence = 0.5 * (coh_C + coh_H)
```

```
        # record this step
        record = {
            "supercycle": self.supercycle_count,
            "env_id": env_odin.id,
            "self_id": self_odin.id,
            "env_layers": env_layers,
            "self_layers": self_layers,
            "middle_coherence": middle_coherence,
            "identity_field": self.identity_field.describe(),
        }
        self.history.append(record)

        # 4) swap roles for next supercycle
        self._swap_roles()

    # -----------------------------
    #  PUBLIC API
    # -----------------------------

    def run(self, n_supercycles: int,
            external_stimulus_sequence: Optional[List[Dict[str, Any]]] = None):
        """
        Run multiple supercycles. Optionally provide a list of stimuli (one per cycle).
        """
        if external_stimulus_sequence is None:
            external_stimulus_sequence = [None] * n_supercycles

        for i in range(n_supercycles):
            stim = external_stimulus_sequence[i] if i < len(external_stimulus_sequence) else None
            self.supercycle(external_stimulus=stim)

    def describe(self):
        print(f"Yggdrasil supercycles={self.supercycle_count}")
        print("Global identity field:", self.identity_field.describe())
        print("\nOdin A:")
        self.odin_A.describe(indent=1)
        print("\nOdin B:")
        self.odin_B.describe(indent=1)
        print("\nRecent middle coherence values:")
        for rec in self.history[-5:]:
            print(f"  supercycle {rec['supercycle']}: coherence={rec['middle_coherence']:.3f}")
```