```
import threading
import time
import uuid
from typing import Any, Callable, Dict, List, Optional

from swarms.utils.any_to_str import any_to_str
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="swarm_arange")


def swarm_id():
    return uuid.uuid4().hex


class SwarmArrangeInput:
    id: str = uuid.uuid4().hex
    time_stamp: str = time.strftime("%Y-%m-%d %H:%M:%S")
    name: str
    description: str
    swarms: List[Callable] = []
    output_type: str
    flow: str = ""


class SwarmArrangeOutput:
    input_config: SwarmArrangeInput = None



class SwarmRearrange:
    """
    A class representing a swarm of swarms for rearranging tasks.

    Attributes:
        id (str): Unique identifier for the swarm arrangement
        name (str): Name of the swarm arrangement
        description (str): Description of what this swarm arrangement does
        swarms (dict): A dictionary of swarms, where the key is the swarm's name and the value is the
swarm object
        flow (str): The flow pattern of the tasks
        max_loops (int): The maximum number of loops to run the swarm
        verbose (bool): A flag indicating whether to log verbose messages
        human_in_the_loop (bool): A flag indicating whether human intervention is required
        custom_human_in_the_loop (Callable[[str], str], optional): A custom function for
human-in-the-loop intervention
        return_json (bool): A flag indicating whether to return the result in JSON format
        swarm_history (dict): A dictionary to keep track of the history of each swarm
        lock (threading.Lock): A lock for thread-safe operations

    Methods:
        __init__(id: str, name: str, description: str, swarms: List[swarm], flow: str, max_loops: int,
verbose: bool,
                        human_in_the_loop: bool, custom_human_in_the_loop: Callable, return_json: bool):
Initializes the SwarmRearrange object
        add_swarm(swarm: swarm): Adds an swarm to the swarm
        remove_swarm(swarm_name: str): Removes an swarm from the swarm
        add_swarms(swarms: List[swarm]): Adds multiple swarms to the swarm
        validate_flow(): Validates the flow pattern
        run(task): Runs the swarm to rearrange the tasks
    """

    def __init__(
        self,
        id: str = swarm_id(),
        name: str = "SwarmRearrange",
        description: str = "A swarm of swarms for rearranging tasks.",
        swarms: List[Any] = [],
        flow: str = None,
        max_loops: int = 1,
        verbose: bool = True,
        human_in_the_loop: bool = False,
        custom_human_in_the_loop: Optional[
            Callable[[str], str]
        ] = None,
        return_json: bool = False,
        *args,
        **kwargs,
    ):
        """
        Initializes the SwarmRearrange object.

        Args:
            id (str): Unique identifier for the swarm arrangement. Defaults to generated UUID.
            name (str): Name of the swarm arrangement. Defaults to "SwarmRearrange".
            description (str): Description of what this swarm arrangement does.
            swarms (List[swarm]): A list of swarm objects. Defaults to empty list.
            flow (str): The flow pattern of the tasks. Defaults to None.
            max_loops (int): Maximum number of loops to run. Defaults to 1.
            verbose (bool): Whether to log verbose messages. Defaults to True.
            human_in_the_loop (bool): Whether human intervention is required. Defaults to False.
             custom_human_in_the_loop (Callable): Custom function for human intervention. Defaults to None.
            return_json (bool): Whether to return results as JSON. Defaults to False.
        """
        self.id = id
        self.name = name
        self.description = description
        self.swarms = {swarm.name: swarm for swarm in swarms}
        self.flow = flow if flow is not None else ""
        self.max_loops = max_loops if max_loops > 0 else 1
        self.verbose = verbose
        self.human_in_the_loop = human_in_the_loop
        self.custom_human_in_the_loop = custom_human_in_the_loop
        self.return_json = return_json
        self.swarm_history = {swarm.name: [] for swarm in swarms}
        self.lock = threading.Lock()
        self.id = uuid.uuid4().hex if id is None else id

        # Run the reliability checks
        self.reliability_checks()

        # Logging configuration
        if self.verbose:
            logger.add("swarm_rearrange.log", rotation="10 MB")

    def reliability_checks(self):
        logger.info("Running reliability checks.")
        if not self.swarms:
            raise ValueError("No swarms found in the swarm.")

        if not self.flow:
            raise ValueError("No flow found in the swarm.")

        if self.max_loops <= 0:
            raise ValueError("Max loops must be a positive integer.")

        logger.info(
            "SwarmRearrange initialized with swarms: {}".format(
                list(self.swarms.keys())
            )
        )

    def set_custom_flow(self, flow: str):
        self.flow = flow
        logger.info(f"Custom flow set: {flow}")

    def add_swarm(self, swarm: Any):
        """
        Adds an swarm to the swarm.

        Args:
            swarm (swarm): The swarm to be added.
        """
        logger.info(f"Adding swarm {swarm.name} to the swarm.")
        self.swarms[swarm.name] = swarm

    def track_history(
        self,
        swarm_name: str,
        result: str,
    ):
        self.swarm_history[swarm_name].append(result)

    def remove_swarm(self, swarm_name: str):
        """
        Removes an swarm from the swarm.
```

```
        Args:
            swarm_name (str): The name of the swarm to be removed.
        """
        del self.swarms[swarm_name]

    def add_swarms(self, swarms: List[Any]):
        """
        Adds multiple swarms to the swarm.

        Args:
            swarms (List[swarm]): A list of swarm objects.
        """
        for swarm in swarms:
            self.swarms[swarm.name] = swarm

    def validate_flow(self):
        """
        Validates the flow pattern.

        Raises:
            ValueError: If the flow pattern is incorrectly formatted or contains duplicate swarm names.

        Returns:
            bool: True if the flow pattern is valid.
        """
```

```
if "->" not in self.flow:
    raise ValueError(
        "Flow must include '->' to denote the direction of the task."
    )

swarms_in_flow = []

# Arrow
tasks = self.flow.split("->")

# For the task in tasks
for task in tasks:
    swarm_names = [name.strip() for name in task.split(",")]

    # Loop over the swarm names
    for swarm_name in swarm_names:
        if (
            swarm_name not in self.swarms
            and swarm_name != "H"
        ):
            raise ValueError(
                f"swarm '{swarm_name}' is not registered."
            )
        swarms_in_flow.append(swarm_name)

# If the length of the swarms does not equal the length of the swarms in flow
```

```
    if len(set(swarms_in_flow)) != len(swarms_in_flow):
        raise ValueError(
            "Duplicate swarm names in the flow are not allowed."
        )

    logger.info("Flow is valid.")
    return True

def run(
    self,
    task: str = None,
    img: str = None,
    custom_tasks: Optional[Dict[str, str]] = None,
    *args,
    **kwargs,
):
    """
    Runs the swarm to rearrange the tasks.

    Args:
        task: The initial task to be processed.
        img: An optional image input.
        custom_tasks: A dictionary of custom tasks for specific swarms.

    Returns:
        str: The final processed task.
```

```
"""
try:
    if not self.validate_flow():
        return "Invalid flow configuration."


    tasks = self.flow.split("->")
    current_task = task


    # Check if custom_tasks is a dictionary and not empty
    if isinstance(custom_tasks, dict) and custom_tasks:
        c_swarm_name, c_task = next(
            iter(custom_tasks.items())
        )


        # Find the position of the custom swarm in the tasks list
        if c_swarm_name in tasks:
            position = tasks.index(c_swarm_name)


            # If there is a previous swarm, merge its task with the custom tasks
            if position > 0:
                tasks[position - 1] += "->" + c_task
            else:
                # If there is no previous swarm, just insert the custom tasks
                tasks.insert(position, c_task)


    # Set the loop counter
```

```
loop_count = 0

while loop_count < self.max_loops:

    for task in tasks:

        swarm_names = [

            name.strip() for name in task.split(",")

        ]

        if len(swarm_names) > 1:

            # Parallel processing

            logger.info(

                f"Running swarms in parallel: {swarm_names}"

            )

            results = []

            for swarm_name in swarm_names:

                if swarm_name == "H":

                    # Human in the loop intervention

                    if (

                        self.human_in_the_loop

                        and self.custom_human_in_the_loop

                    ):

                        current_task = (

                            self.custom_human_in_the_loop(

                                current_task

                            )

                        )

                    else:

                        current_task = input(
                            "Enter your response: "
                        )
                else:
                    swarm = self.swarms[swarm_name]
                    result = swarm.run(
                        current_task, img, *args, **kwargs
                    )
                    result = any_to_str(result)
                    logger.info(
                        f"Swarm {swarm_name} returned result of type: {type(result)}"
                    )
                    if isinstance(result, bool):
                        logger.warning(
                            f"Swarm {swarm_name} returned a boolean value: {result}"
                        )
                        result = str(
                            result
                        )  # Convert boolean to string
                    results.append(result)

            current_task = "; ".join(
                str(r) for r in results if r is not None
            )
        else:
            # Sequential processing
            logger.info(
            f"Running swarms sequentially: {swarm_names}"
        )
        swarm_name = swarm_names[0]
        if swarm_name == "H":
            # Human-in-the-loop intervention
            if (
                self.human_in_the_loop
                and self.custom_human_in_the_loop
            ):
                current_task = (
                    self.custom_human_in_the_loop(
                        current_task
                    )
                )
            else:
                current_task = input(
                    "Enter the next task: "
                )
        else:
            swarm = self.swarms[swarm_name]
            result = swarm.run(
                current_task, img, *args, **kwargs
            )
            result = any_to_str(result)
            logger.info(
                f"Swarm {swarm_name} returned result of type: {type(result)}"
```

```
                            )
                            if isinstance(result, bool):
                                logger.warning(
                                    f"Swarm {swarm_name} returned a boolean value: {result}"
                                )
                                result = str(
                                    result
                                )  # Convert boolean to string
                            current_task = (
                                result
                                if result is not None
                                else current_task
                            )
                loop_count += 1

            return current_task

        except Exception as e:
            logger.error(f"An error occurred: {e}")
            return str(e)


def swarm_arrange(
    name: str = "SwarmArrange-01",
    description: str = "Combine multiple swarms and execute them sequentially",
    swarms: List[Callable] = None,
    output_type: str = "json",
    flow: str = None,
    task: str = None,
    *args,
    **kwargs,
):
    """
    Orchestrates the execution of multiple swarms in a sequential manner.

    Args:
        name (str, optional): The name of the swarm arrangement. Defaults to "SwarmArrange-01".
            description (str, optional): A description of the swarm arrangement. Defaults to "Combine
multiple swarms and execute them sequentially".
        swarms (List[Callable], optional): A list of swarm objects to be executed. Defaults to None.
        output_type (str, optional): The format of the output. Defaults to "json".
        flow (str, optional): The flow pattern of the tasks. Defaults to None.
        task (str, optional): The task to be executed by the swarms. Defaults to None.
        *args: Additional positional arguments to be passed to the SwarmRearrange object.
        **kwargs: Additional keyword arguments to be passed to the SwarmRearrange object.

    Returns:
        Any: The result of the swarm arrangement execution.
    """
    try:
        swarm_arrangement = SwarmRearrange(
            name,
            description,
            swarms,
            output_type,
            flow,
        )
        result = swarm_arrangement.run(task, *args, **kwargs)
        result = any_to_str(result)
        logger.info(
            f"Swarm arrangement {name} executed successfully with output type {output_type}."
        )
        return result
    except Exception as e:
        logger.error(
            f"An error occurred during swarm arrangement execution: {e}"
        )
        return str(e)
```