# `AgentRearrange` Class

The `AgentRearrange` class represents a swarm of agents for rearranging tasks. It allows you to create a swarm of agents, add or remove agents from the swarm, and run the swarm to process tasks based on a specified flow pattern.

## Attributes

---

| Attribute | Type | Description |
| --- | --- | --- |
| `id` | `str` | Unique identifier for the swarm |
| `name` | `str` | Name of the swarm |
| `description` | `str` | Description of the swarm's purpose |
| `agents` | `dict` | Dictionary mapping agent names to Agent objects |
| `flow` | `str` | Flow pattern defining task execution order |
| `max_loops` | `int` | Maximum number of execution loops |
| `verbose` | `bool` | Whether to enable verbose logging |
| `memory_system` | `BaseVectorDatabase` | Memory system for storing agent interactions |
| `human_in_the_loop` | `bool` | Whether human intervention is enabled |
| `custom_human_in_the_loop` | `Callable` | Custom function for human intervention |
| `return_json` | `bool` | Whether to return output in JSON format |
| `output_type` | `OutputType` | Format of output ("all", "final", "list", or "dict") |
| `docs` | `List[str]` | List of document paths to add to agent prompts |
| `doc_folder` | `str` | Folder path containing documents to add to agent prompts |
| `swarm_history` | `dict` | History of agent interactions |

## Methods

---

### `__init__(self, agents: List[Agent] = None, flow: str = None, max_loops: int = 1, verbose: bool = True)`

Initializes the `AgentRearrange` object.

| Parameter | Type | Description |
| --- | --- | --- |
| `agents` | `List[Agent]` (optional) | A list of `Agent` objects. Defaults to `None`. |
| `flow` | `str` (optional) | The flow pattern of the tasks. Defaults to `None`. |
| `max_loops` | `int` (optional) | The maximum number of loops for the agents to run. Defaults to `1`. |
| `verbose` | `bool` (optional) | Whether to enable verbose logging or not. Defaults to `True`. |

### `add_agent(self, agent: Agent)`

Adds an agent to the swarm.

| Parameter | Type | Description |
| --- | --- | --- |
| `agent` | `Agent` | The agent to be added. |

### `remove_agent(self, agent_name: str)`

Removes an agent from the swarm.

| Parameter | Type | Description |
| --- | --- | --- |
| `agent_name` | `str` | The name of the agent to be removed. |

### `add_agents(self, agents: List[Agent])`

Adds multiple agents to the swarm.

| Parameter | Type | Description |
| --- | --- | --- |
| `agents` | `List[Agent]` | A list of `Agent` objects. |

### `validate_flow(self)`

Validates the flow pattern.

**Raises:**

- `ValueError`: If the flow pattern is incorrectly formatted or contains duplicate agent names.

**Returns:**

- `bool`: `True` if the flow pattern is valid.

### `run(self, task: str = None, img: str = None, device: str = "cpu", device_id: int = 1, all_cores: bool = True, all_gpus: bool = False, *args, **kwargs)`

Executes the agent rearrangement task with specified compute resources.

| Parameter | Type | Description |
| --- | --- | --- |
| `task` | `str` | The task to execute |
| `img` | `str` | Path to input image if required |
| `device` | `str` | Computing device to use ('cpu' or 'gpu') |
| `device_id` | `int` | ID of specific device to use |
| `all_cores` | `bool` | Whether to use all CPU cores |
| `all_gpus` | `bool` | Whether to use all available GPUs |

**Returns:**

- `str`: The final processed task.

### `batch_run(self, tasks: List[str], img: Optional[List[str]] = None, batch_size: int = 10, device: str = "cpu", device_id: int = None, all_cores: bool = True, all_gpus: bool = False, *args, **kwargs)`

Process multiple tasks in batches.

| Parameter | Type | Description |

| --- | --- | --- |

| `tasks` | `List[str]` | List of tasks to process |

| `img` | `List[str]` | Optional list of images corresponding to tasks |

| `batch_size` | `int` | Number of tasks to process simultaneously |

| `device` | `str` | Computing device to use |

| `device_id` | `int` | Specific device ID if applicable |

| `all_cores` | `bool` | Whether to use all CPU cores |

| `all_gpus` | `bool` | Whether to use all available GPUs |

### `concurrent_run(self, tasks: List[str], img: Optional[List[str]] = None, max_workers: Optional[int] = None, device: str = "cpu", device_id: int = None, all_cores: bool = True, all_gpus: bool = False, *args, **kwargs)`

Process multiple tasks concurrently using ThreadPoolExecutor.

| Parameter | Type | Description |

| --- | --- | --- |

| `tasks` | `List[str]` | List of tasks to process |

| `img` | `List[str]` | Optional list of images corresponding to tasks |

| `max_workers` | `int` | Maximum number of worker threads |

| `device` | `str` | Computing device to use |

| `device_id` | `int` | Specific device ID if applicable |

| `all_cores` | `bool` | Whether to use all CPU cores |

| `all_gpus` | `bool` | Whether to use all available GPUs |

## Documentation for `rearrange` Function

==========

The `rearrange` function is a helper function that rearranges the given list of agents based on the specified flow.

## Parameters

---

| Parameter | Type | Description |
| --- | --- | --- |
| `agents` | `List[Agent]` | The list of agents to be rearranged. |
| `flow` | `str` | The flow used for rearranging the agents. |
| `task` | `str` (optional) | The task to be performed during rearrangement. Defaults to `None`. |
| `*args` | - | Additional positional arguments. |
| `**kwargs` | - | Additional keyword arguments. |

## Returns

---

The result of running the agent system with the specified task.

### Example

---

```python
agents = [agent1, agent2, agent3]
flow = "agent1 -> agent2, agent3"
task = "Perform a task"
rearrange(agents, flow, task)
```

### Example Usage

---

Here's an example of how to use the `AgentRearrange` class and the `rearrange` function:

```python
from swarms import Agent, AgentRearrange
from typing import List

# Initialize the director agent
director = Agent(
    agent_name="Accounting Director",
    system_prompt="Directs the accounting tasks for the workers",
    llm=Anthropic(),
    max_loops=1,
    dashboard=False,
    streaming_on=True,
```

```
    verbose=True,
    stopping_token="<DONE>",
    state_save_file_type="json",
    saved_state_path="accounting_director.json",
)

# Initialize worker 1
worker1 = Agent(
    agent_name="Accountant 1",
    system_prompt="Processes financial transactions and prepares financial statements",
    llm=Anthropic(),
    max_loops=1,
    dashboard=False,
    streaming_on=True,
    verbose=True,
    stopping_token="<DONE>",
    state_save_file_type="json",
    saved_state_path="accountant1.json",
)

# Initialize worker 2
worker2 = Agent(
    agent_name="Accountant 2",
    system_prompt="Performs audits and ensures compliance with financial regulations",
    llm=Anthropic(),
    max_loops=1,
```

```
    dashboard=False,
    streaming_on=True,
    verbose=True,
    stopping_token="<DONE>",
    state_save_file_type="json",
    saved_state_path="accountant2.json",
)

# Create a list of agents
agents = [director, worker1, worker2]

# Define the flow pattern
flow = "Accounting Director -> Accountant 1 -> Accountant 2"

# Using AgentRearrange class
agent_system = AgentRearrange(agents=agents, flow=flow)
output = agent_system.run("Process monthly financial statements")
print(output)
```

In this example, we first initialize three agents: `director`, `worker1`, and `worker2`. Then, we create a list of these agents and define the flow pattern `"Director -> Worker1 -> Worker2"`.

We can use the `AgentRearrange` class by creating an instance of it with the list of agents and the flow pattern. We then call the `run` method with the initial task, and it will execute the agents in the

specified order, passing the output of one agent as the input to the next agent.

Alternatively, we can use the `rearrange` function by passing the list of agents, the flow pattern, and the initial task as arguments.

Both the `AgentRearrange` class and the `rearrange` function will return the final output after processing the task through the agents according to the specified flow pattern.

## Error Handling

---

The `AgentRearrange` class includes error handling mechanisms to validate the flow pattern. If the flow pattern is incorrectly formatted or contains duplicate agent names, a `ValueError` will be raised with an appropriate error message.

### Example:

```python
# Invalid flow pattern
invalid_flow = "Director->Worker1,Worker2->Worker3"
agent_system = AgentRearrange(agents=agents, flow=invalid_flow)
output = agent_system.run("Some task")`
```

This will raise a `ValueError` with the message `"Agent 'Worker3' is not registered."`.

## Parallel and Sequential Processing

---

The `AgentRearrange` class supports both parallel and sequential processing of tasks based on the specified flow pattern. If the flow pattern includes multiple agents separated by commas (e.g., `"agent1, agent2"`), the agents will be executed in parallel, and their outputs will be concatenated with a semicolon (`;`). If the flow pattern includes a single agent, it will be executed sequentially.

### Parallel processing

`parallel_flow = "Worker1, Worker2 -> Director"`

### Sequential processing

`sequential_flow = "Worker1 -> Worker2 -> Director"`

In the `parallel_flow` example, `Worker1` and `Worker2` will be executed in parallel, and their outputs will be concatenated and passed to `Director`. In the `sequential_flow` example, `Worker1` will be executed first, and its output will be passed to `Worker2`, and then the output of `Worker2` will be passed to `Director`.

## Logging

---

The `AgentRearrange` class includes logging capabilities using the `loguru` library. If `verbose` is set to `True` during initialization, a log file named `agent_rearrange.log` will be created, and log

messages will be written to it. You can use this log file to track the execution of the agents and any potential issues or errors that may occur.

```bash
2023-05-08 10:30:15.456 | INFO | agent_rearrange:__init__:34 - Adding agent Director to the swarm.
2023-05-08 10:30:15.457 | INFO | agent_rearrange:__init__:34 - Adding agent Worker1 to the swarm.
2023-05-08 10:30:15.457 | INFO | agent_rearrange:__init__:34 - Adding agent Worker2 to the swarm.
2023-05-08 10:30:15.458 | INFO | agent_rearrange:run:118 - Running agents in parallel: ['Worker1', 'Worker2']
2023-05-08 10:30:15.459 | INFO | agent_rearrange:run:121 - Running agents sequentially: ['Director']`
```

## Additional Parameters

---

The `AgentRearrange` class also accepts additional parameters that can be passed to the `run` method using `*args` and `**kwargs`. These parameters will be forwarded to the individual agents during execution.

`agent_system = AgentRearrange(agents=agents, flow=flow)`

`output = agent_system.run("Some task", max_tokens=200, temperature=0.7)`

In this example, the `max_tokens` and `temperature` parameters will be passed to each agent during execution.

## Customization

---

The `AgentRearrange` class and the `rearrange` function can be customized and extended to suit specific use cases. For example, you can create custom agents by inheriting from the `Agent` class and implementing custom logic for task processing. You can then add these custom agents to the swarm and define the flow pattern accordingly.

Additionally, you can modify the `run` method of the `AgentRearrange` class to implement custom logic for task processing and agent interaction.

## Limitations

---

It's important to note that the `AgentRearrange` class and the `rearrange` function rely on the individual agents to process tasks correctly. The quality of the output will depend on the capabilities and configurations of the agents used in the swarm. Additionally, the `AgentRearrange` class does not provide any mechanisms for task prioritization or load balancing among the agents.

## Conclusion

---

The `AgentRearrange` class and the `rearrange` function provide a flexible and extensible framework for orchestrating swarms of agents to process tasks based on a specified flow pattern. By combining the capabilities of individual agents, you can create complex workflows and leverage the strengths of different agents to tackle various tasks efficiently.

While the current implementation offers basic functionality for agent rearrangement, there is room for future improvements and customizations to enhance the system's capabilities and cater to more specific use cases.

Whether you're working on natural language processing tasks, data analysis, or any other domain where agent-based systems can be beneficial, the `AgentRearrange` class and the `rearrange` function provide a solid foundation for building and experimenting with swarm-based solutions.