Create your own agent with `Agent` class

The Agent class is a powerful and flexible tool that empowers AI agents to build their own custom agents, tailored to their specific needs.

This comprehensive guide will explore the process of inheriting from the Agent class, enabling agents to create their own custom agent classes. By leveraging the rich features and extensibility of the Agent class, agents can imbue their offspring agents with unique capabilities, specialized toolsets, and tailored decision-making processes.

Understanding the Agent Class

Before we dive into the intricacies of creating custom agent classes, let's revisit the foundational elements of the Agent class itself. The Agent class is a versatile and feature-rich class designed to streamline the process of building and managing AI agents. It acts as a backbone, connecting language models (LLMs) with various tools, long-term memory, and a wide range of customization options.

Key Features of the Agent Class

The Agent class offers a plethora of features that can be inherited and extended by custom agent classes. Here are some of the key features that make the Agent class a powerful foundation:

1\. **Language Model Integration**: The Agent class supports seamless integration with popular language models such as LangChain, HuggingFace Transformers, and Autogen, allowing custom agent classes to leverage the power of state-of-the-art language models.

2\. **Tool Integration**: One of the standout features of the Agent class is its ability to integrate with various tools. Custom agent classes can inherit this capability and incorporate specialized tools tailored to their specific use cases.

3\. **Long-Term Memory**: The Agent class provides built-in support for long-term memory, enabling custom agent classes to retain and access information from previous interactions, essential for maintaining context and learning from past experiences.

4\. **Customizable Prompts and Standard Operating Procedures (SOPs)**: The Agent class allows you to define custom prompts and Standard Operating Procedures (SOPs) that guide an agent's behavior and decision-making process. Custom agent classes can inherit and extend these prompts and SOPs to align with their unique objectives and requirements.

5\. **Interactive and Dashboard Modes**: The Agent class supports interactive and dashboard modes, enabling real-time monitoring and interaction with agents. Custom agent classes can inherit these modes, facilitating efficient development, debugging, and user interaction.

6\. **Autosave and State Management**: With the Agent class, agents can easily save and load their state, including configuration, memory, and history. Custom agent classes can inherit this capability, ensuring seamless task continuation and enabling efficient collaboration among team members.

7\. **Response Filtering**: The Agent class provides built-in response filtering capabilities, allowing agents to filter out or replace specific words or phrases in their responses. Custom agent classes can inherit and extend this feature to ensure compliance with content moderation policies or specific

guidelines.

8\. **Code Execution and Multimodal Support**: The Agent class supports code execution and multimodal input/output, enabling agents to process and generate code, as well as handle various data formats such as images, audio, and video. Custom agent classes can inherit and specialize these capabilities for their unique use cases.

9\. **Extensibility and Customization**: The Agent class is designed to be highly extensible and customizable, allowing agents to tailor its behavior, add custom functionality, and integrate with external libraries and APIs. Custom agent classes can leverage this extensibility to introduce specialized features and capabilities.

Creating a Custom Agent Class

Now that we have a solid understanding of the Agent class and its features, let's dive into the process of creating a custom agent class by inheriting from the Agent class. Throughout this process, we'll explore how agents can leverage and extend the existing functionality, while introducing specialized features and capabilities tailored to their unique requirements.

Step 1: Inherit from the Agent Class

The first step in creating a custom agent class is to inherit from the Agent class. This will provide your custom agent class with the foundational features and capabilities of the Agent class, which can then be extended and customized as needed. The new agent class must have a `run(task: str)` method to run the entire agent. It is encouraged to have `step(task: str)` method that completes one step of the agent and then build the `run(task: str)` method.

```python

from swarms import Agent

class MyCustomAgent(Agent):

    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)

        # Add custom initialization logic here

    def run(self, task: str) ->

        ...

```

In the example above, we define a new class `MyCustomAgent` that inherits from the `Agent` class. Within the `__init__` method, we call the parent class's `__init__` method using `super().__init__(*args, **kwargs)`, which ensures that the parent class's initialization logic is executed. You can then add any custom initialization logic specific to your custom agent class.

Step 2: Customize the Agent's Behavior

One of the key advantages of inheriting from the Agent class is the ability to customize the agent's

behavior according to your specific requirements. This can be achieved by overriding or extending the existing methods, or by introducing new methods altogether.

```python
from swarms import Agent


class MyCustomAgent(Agent):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Custom initialization logic

    def custom_method(self, *args, **kwargs):
        # Implement custom logic here
        pass

    def run(self, task, *args, **kwargs):
        # Customize the run method
        response = super().run(task, *args, **kwargs)
```

```
        # Additional custom logic

        return response
```

In the example above, we introduce a new `custom_method` that can encapsulate any specialized logic or functionality specific to your custom agent class. Additionally, we override the `run` method, which is responsible for executing the agent's main task loop. Within the overridden `run` method, you can call the parent class's `run` method using `super().run(task, *args, **kwargs)` and then introduce any additional custom logic before or after the parent method's execution.

Step 3: Extend Memory Management

The Agent class provides built-in support for long-term memory, allowing agents to retain and access information from previous interactions. Custom agent classes can inherit and extend this capability by introducing specialized memory management techniques.

```python
from swarms_memory import BaseVectorDatabase
from swarms import Agent


class CustomMemory(BaseVectorDatabase):
```

```
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Custom memory initialization logic
    def query(self, *args, **kwargs):
        # Custom memory query logic
        return result
class MyCustomAgent(Agent):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Custom initialization logic
        self.long_term_memory = CustomMemory()
    def run(self, task, *args, **kwargs):
        # Customize the run method
```

```
        response = super().run(task, *args, **kwargs)

        # Utilize custom memory

        memory_result = self.long_term_memory.query(*args, **kwargs)

        # Process memory result

        return response
```

In the example above, we define a new `CustomMemory` class that inherits from the `BaseVectorDatabase` class provided by the Agent class framework. Within the `CustomMemory` class, you can implement specialized memory management logic, such as custom indexing, retrieval, and storage mechanisms.

Next, within the `MyCustomAgent` class, we initialize an instance of the `CustomMemory` class and assign it to the `self.long_term_memory` attribute. This custom memory instance can then be utilized within the overridden `run` method, where you can query the memory and process the results as needed.

Step 5: Introduce Custom Prompts and Standard Operating Procedures (SOPs)

The Agent class allows you to define custom prompts and Standard Operating Procedures (SOPs)

that guide an agent's behavior and decision-making process. Custom agent classes can inherit and extend these prompts and SOPs to align with their unique objectives and requirements.

```python
from swarms import Agent


class MyCustomAgent(Agent):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # Custom initialization logic
        self.custom_sop = "Custom SOP for MyCustomAgent..."
        self.custom_prompt = "Custom prompt for MyCustomAgent..."
    def run(self, task, *args, **kwargs):
        # Customize the run method
        response = super().run(task, *args, **kwargs)
        # Utilize custom prompts and SOPs
```

```
        custom_prompt = self.construct_dynamic_prompt(self.custom_prompt)

        custom_sop = self.construct_dynamic_sop(self.custom_sop)

        # Process custom prompts and SOPs

        return response

    def construct_dynamic_prompt(self, prompt):

        # Custom prompt construction logic

        return prompt

    def construct_dynamic_sop(self, sop):

        # Custom SOP construction logic

        return sop
```

In the example above, we define two new attributes within the `MyCustomAgent` class: `custom_sop` and `custom_prompt`. These attributes can be used to store custom prompts and SOPs specific to your custom agent class.

Within the overridden `run` method, you can utilize these custom prompts and SOPs by calling the `construct_dynamic_prompt` and `construct_dynamic_sop` methods, which can be defined within the `MyCustomAgent` class to implement specialized prompt and SOP construction logic.

Step 5: Introduce Custom Response Handling

The Agent class provides built-in response filtering capabilities, allowing agents to filter out or replace specific words or phrases in their responses. Custom agent classes can inherit and extend this feature to ensure compliance with content moderation policies or specific guidelines.

```python
from swarms import Agent


class MyCustomAgent(Agent):

    def __init__(self, *args, **kwargs):

        super().__init__(*args, **kwargs)

        # Custom initialization logic

        self.response_filters = ["filter_word_1", "filter_word_2"]

    def run(self, task, *args, **kwargs):
```

```
        # Customize the run method

        response = super().run(task, *args, **kwargs)

        # Apply custom response filtering

        filtered_response = self.apply_response_filters(response)

        return filtered_response

    def apply_response_filters(self, response):

        # Custom response filtering logic

        for word in self.response_filters:

            response = response.replace(word, "[FILTERED]")

        return response
```

In the example above, we define a new attribute `response_filters` within the `MyCustomAgent` class, which is a list of words or phrases that should be filtered out or replaced in the agent's responses.

Within the overridden `run` method, we call the `apply_response_filters` method, which can be defined within the `MyCustomAgent` class to implement specialized response filtering logic. In the example, we iterate over the `response_filters` list and replace each filtered word or phrase with a placeholder string (`"[FILTERED]"`).

Advanced Customization and Integration

The Agent class and its inherited custom agent classes can be further extended and customized to suit specific requirements and integrate with external libraries, APIs, and services. Here are some advanced customization and integration examples:

1\. **Multimodal Input/Output Integration**: Custom agent classes can leverage the multimodal input/output capabilities of the Agent class and introduce specialized handling for various data formats such as images, audio, and video.

2\. **Code Execution and Integration**: The Agent class supports code execution, enabling agents to run and evaluate code snippets. Custom agent classes can inherit and extend this capability, introducing specialized code execution environments, sandboxing mechanisms, or integration with external code repositories or platforms.

3\. **External API and Service Integration**: Custom agent classes can integrate with external APIs and services, enabling agents to leverage specialized data sources, computational resources, or domain-specific services.

4\. **Performance Optimization**: Depending on the use case and requirements, custom agent

classes can introduce performance optimizations, such as adjusting loop intervals, retry attempts, or enabling parallel execution for certain tasks.

5\. **Logging and Monitoring**: Custom agent classes can introduce specialized logging and monitoring mechanisms, enabling agents to track their performance, identify potential issues, and generate detailed reports or dashboards.

6\. **Security and Privacy Enhancements**: Custom agent classes can implement security and privacy enhancements, such as data encryption, access control mechanisms, or compliance with industry-specific regulations and standards.

7\. **Distributed Execution and Scaling**: Custom agent classes can be designed to support distributed execution and scaling, enabling agents to leverage cloud computing resources or distributed computing frameworks for handling large-scale tasks or high-concurrency workloads.

By leveraging these advanced customization and integration capabilities, agents can create highly specialized and sophisticated custom agent classes tailored to their unique requirements and use cases.

Best Practices and Considerations

While building custom agent classes by inheriting from the Agent class offers immense flexibility and power, it's essential to follow best practices and consider potential challenges and considerations:

1\. **Maintainability and Documentation**: As custom agent classes become more complex, it's crucial to prioritize maintainability and thorough documentation. Clear and concise code,

comprehensive comments, and up-to-date documentation can significantly improve the long-term sustainability and collaboration efforts surrounding custom agent classes.

2\. **Testing and Validation**: Custom agent classes should undergo rigorous testing and validation to ensure their correctness, reliability, and adherence to expected behaviors. Establish a robust testing framework and continuously validate the agent's performance, particularly after introducing new features or integrations.

3\. **Security and Privacy Considerations**: When building custom agent classes, it's essential to consider security and privacy implications, especially if the agents will handle sensitive data or interact with critical systems. Implement appropriate security measures, such as access controls, data encryption, and secure communication protocols, to protect against potential vulnerabilities and ensure compliance with relevant regulations and standards.

4\. **Scalability and Performance Monitoring**: As custom agent classes are deployed and adopted, it's important to monitor their scalability and performance characteristics. Identify potential bottlenecks, resource constraints, or performance degradation, and implement appropriate optimization strategies or scaling mechanisms to ensure efficient and reliable operation.

5\. **Collaboration and Knowledge Sharing**: Building custom agent classes often involves collaboration among teams and stakeholders. Foster an environment of knowledge sharing, code reviews, and open communication to ensure that everyone involved understands the agent's capabilities, limitations, and intended use cases.

6\. **Ethical Considerations**: As AI agents become more advanced and autonomous, it's crucial to consider the ethical implications of their actions and decisions. Implement appropriate safeguards,

oversight mechanisms, and ethical guidelines to ensure that custom agent classes operate in a responsible and transparent manner, aligning with ethical principles and societal values.

7\. **Continuous Learning and Adaptation**: The field of AI is rapidly evolving, with new techniques, tools, and best practices emerging regularly. Stay up-to-date with the latest developments and be prepared to adapt and refine your custom agent classes as new advancements become available.

By following these best practices and considering potential challenges, agents can create robust, reliable, and ethical custom agent classes that meet their specific requirements while adhering to industry standards and best practices.

Conclusion

In this comprehensive guide, we have explored the process of creating custom agent classes by inheriting from the powerful Agent class. We have covered the key features of the Agent class, walked through the step-by-step process of inheriting and extending its functionality, and discussed advanced customization and integration techniques.

Building custom agent classes empowers AI agents to create tailored and specialized agents capable of tackling unique challenges and addressing specific domain requirements. By leveraging the rich features and extensibility of the Agent class, agents can imbue their offspring agents with unique capabilities, specialized toolsets, and tailored decision-making processes.

Remember, the journey of building custom agent classes is an iterative and collaborative process that requires continuous learning, adaptation, and refinement.