File Breakdown: `src/agents/base_agent.py`

File Location

`src/agents/base_agent.py`

Overview

The `base_agent.py` file defines the abstract base class `BaseAgent` that all specialized agents (Nova, Emil, Ivan, Lola) inherit from. It provides the common interface and functionality that all agents must implement, establishing a consistent pattern for task handling, argument collection, and knowledge base interaction.

Key Responsibilities

- Define the base structure for all agent implementations
- Provide methods for handling tasks (both synchronous and asynchronous)
- Implement user interaction for collecting missing arguments
- Log function calls for debugging and monitoring
- Interact with the knowledge base

Core Functionality

Class Definition

```
class BaseAgent:
    def __init__(self, name: str, knowledge_base: KnowledgeBase, function_map:
Dict[str, Any]):
        self.name = name
        self.kb = knowledge_base
        self.function_map = function_map
        # Create an async_function_map that will be populated with async versions
        self.async_function_map = {}
```

The constructor initializes the agent with:

- A name identifier
- A reference to the shared knowledge base
- A mapping of function names to callable functions
- An empty async function map for asynchronous operations

Task Handling

Synchronous task handling method that specialized agents must implement:

```
@log_function_call
def handle_task(self, task: Task):
    """
    Main entry point for an agent to handle an incoming task.
    Agents can parse the task to see if it maps to a function or requires subtask
creation.
    Original synchronous method preserved.
```

```
    """
    raise NotImplementedError()
```

Asynchronous task handling with default implementation:

```
@log_function_call
async def handle_task_async(self, task: Task):
    """
    Asynchronous version of handle_task.
    Default implementation runs the synchronous version in a thread pool.
    Agents should override this with a true async implementation.
    """
    # Default implementation runs the sync version in a thread pool
    return await asyncio.to_thread(self.handle_task, task)
```

Parameter Collection

Synchronous method for collecting missing arguments from the user:

```
@log_function_call
def ask_user_for_missing_args(self, missing_args: List[str]) -> Dict[str, Any]:
    """
    If the agent detects missing arguments, it can prompt the user for them.
    Original synchronous method preserved.
    """
    print(f"{self.name} is asking user for arguments: {missing_args}")
    collected = {}
    for arg in missing_args:
        # For demonstration, just prompt in console:
        val = input(f"Please provide a value for {arg}: ")
        collected[arg] = val
    return collected
```

Asynchronous version of parameter collection:

```
@log_function_call
async def ask_user_for_missing_args_async(self, missing_args: List[str]) -> Dict[str,
Any]:
    """
    Asynchronous version of ask_user_for_missing_args.
    Uses asyncio to run the input operations in a thread pool.
    """
    print(f"{self.name} is asking user for arguments: {missing_args}")
    collected = {}

    for arg in missing_args:
        # Run input in a thread pool since it's blocking
        val = await asyncio.to_thread(input, f"Please provide a value for {arg}: ")
        collected[arg] = val

    return collected
```

Design Patterns

1. **Template Method Pattern**: The base class defines the skeleton of operations, deferring some steps to subclasses
2. **Decorator Pattern**: Uses the `@log_function_call` decorator for logging and monitoring function execution
3. **Strategy Pattern**: The function_map dictionary allows for different implementations to be injected

Integration

- Used as the parent class for all specialized agents (Nova, Emil, Ivan, Lola)
- Interacts with the knowledge base for data storage and retrieval
- Processes Task objects from the task manager
- Provides hooks for asynchronous processing with asyncio

Implementation Notes

- The base class itself doesn't implement the `handle_task` method, requiring subclasses to provide their own implementation
- Provides a default asynchronous implementation that delegates to the synchronous method
- Uses type hints for better code readability and IDE support
- Includes decorated methods for automatic logging of function calls