```
from concurrent.futures import ThreadPoolExecutor
from typing import Any, Callable, List, Type

from pydantic import BaseModel

from agentparse.function_to_basemodel import (
    function_to_pydantic_schema,
)
from agentparse.json_output_parser import JsonOutputParser
from agentparse.yaml_output_parser import YamlOutputParser
from agentparse.agent_metadata import display_agents_info


class AgentParse:
    """
    AgentParse is a utility class designed to facilitate the conversion of functions to Pydantic models and the parsing of JSON data concurrently.
    It leverages the `function_to_pydantic_schema` function to create Pydantic models from function signatures and the `JsonOutputParser` to parse JSON data.
    """

    def __init__(
        self,
        workers: int = 1,
    ):
        """
```

Initializes the AgentParse instance with the specified number of workers for concurrent operations.

Args:

workers (int, optional): The number of workers to use for concurrent operations. Defaults to 1.

"""

self.workers = workers

```
def func_to_base_model(
    self,
    func: Callable[..., Any],
    name: str = None,
    *args,
    **kwargs,
) -> Type[BaseModel]:
```

"""

Converts a given function to a Pydantic BaseModel.

Args:

func (Callable[..., Any]): The function to convert to a Pydantic model.

name (str, optional): The name for the created model. Defaults to None.

*args: Additional arguments to pass to the `function_to_pydantic_schema` function.

**kwargs: Additional keyword arguments to pass to the `function_to_pydantic_schema` function.

```
        Returns:
            Type[BaseModel]: The created Pydantic BaseModel.
        """
        return function_to_pydantic_schema(
            func, name, *args, **kwargs
        )

    def convert_functions_concurrently(
        self,
        functions: List[Callable[..., Any]],
        names: List[str] = None,
        *args,
        **kwargs,
    ) -> List[Type[BaseModel]]:
        """
        Converts a list of functions to Pydantic models concurrently using a ThreadPoolExecutor.

        Args:
            functions (List[Callable[..., Any]]): A list of functions to convert to Pydantic models.
            names (List[str], optional): A list of names for the created models. Defaults to None.
            *args: Additional arguments to pass to the `function_to_pydantic_schema` function.
            **kwargs: Additional keyword arguments to pass to the `function_to_pydantic_schema` function.

        Returns:
            List[Type[BaseModel]]: A list of created Pydantic models.
```

```
    """
    if names is None:
        names = [None] * len(functions)

    with ThreadPoolExecutor(max_workers=self.workers) as executor:
        futures = [
            executor.submit(
                function_to_pydantic_schema,
                func,
                name,
                *args,
                **kwargs,
            )
            for func, name in zip(functions, names)
        ]
        models = [future.result() for future in futures]
    return models

def parse_json_with_base_model(
    self, base_model: BaseModel, json_data: Any
) -> BaseModel:
    """
    Parses JSON data using a given Pydantic BaseModel.

    Args:
        base_model (BaseModel): The Pydantic model to use for parsing.
```

```
            json_data (Any): The JSON data to parse.

        Returns:
            BaseModel: The parsed Pydantic model instance.
        """
        model = JsonOutputParser(base_model)
        return model.parse(json_data)

    def parse_json_concurrently(
        self,
        base_models: List[BaseModel],
        json_data: List[Any],
    ) -> List[BaseModel]:
        """
        Parses a list of JSON data concurrently using a ThreadPoolExecutor and a list of Pydantic models.

        Args:
            base_models (List[BaseModel]): A list of Pydantic models to use for parsing.
            json_data (List[Any]): A list of JSON data to parse.

        Returns:
            List[BaseModel]: A list of parsed Pydantic model instances.
        """
        with ThreadPoolExecutor(max_workers=self.workers) as executor:
            futures = [
```

```
            executor.submit(
                self.parse_json_with_base_model, model, data
            )
            for model, data in zip(base_models, json_data)
        ]
        parsed_models = [future.result() for future in futures]
    return parsed_models


def yaml_output_parse(
    self, base_model: BaseModel, yaml_data: Any
) -> BaseModel:
    model = YamlOutputParser(base_model)

    return model.parse(yaml_data)


def display_agents_in_table(self, agents: List[Callable]):
    return display_agents_info(agents)
```