```
import json
from typing import Any, Callable, Dict, List, Optional, Union

from pydantic import BaseModel, Field

from swarms.tools.func_calling_executor import openai_tool_executor
from swarms.tools.func_to_str import function_to_str, functions_to_str
from swarms.tools.function_util import process_tool_docs
from swarms.tools.py_func_to_openai_func_str import (
    get_openai_function_schema_from_func,
    load_basemodels_if_needed,
)
from swarms.tools.pydantic_to_json import (
    base_model_to_openai_function,
    multi_base_model_to_openai_function,
)
from swarms.utils.loguru_logger import initialize_logger

logger = initialize_logger(log_folder="base_tool")

ToolType = Union[BaseModel, Dict[str, Any], Callable[..., Any]]


class BaseTool(BaseModel):
    verbose: Optional[bool] = None
    base_models: Optional[List[type[BaseModel]]] = None
```

```
autocheck: Optional[bool] = None

auto_execute_tool: Optional[bool] = None

tools: Optional[List[Callable[..., Any]]] = None

tool_system_prompt: Optional[str] = Field(

    None,

    description="The system prompt for the tool system.",

)

function_map: Optional[Dict[str, Callable]] = None

list_of_dicts: Optional[List[Dict[str, Any]]] = None


def func_to_dict(

    self,

    function: Callable[..., Any] = None,

    name: Optional[str] = None,

    description: str = None,

    *args,

    **kwargs,

) -> Dict[str, Any]:

    try:

        return get_openai_function_schema_from_func(

            function=function,

            name=name,

            description=description,

            *args,

            **kwargs,

        )
    except Exception as e:
        logger.error(f"An error occurred in func_to_dict: {e}")
        logger.error(
            "Please check the function and ensure it is valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise


def load_params_from_func_for_pybasemodel(
    self,
    func: Callable[..., Any],
    *args: Any,
    **kwargs: Any,
) -> Callable[..., Any]:
    try:
        return load_basemodels_if_needed(func, *args, **kwargs)
    except Exception as e:
        logger.error(
            f"An error occurred in load_params_from_func_for_pybasemodel: {e}"
        )
        logger.error(
            "Please check the function and ensure it is valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise


def base_model_to_dict(
    self,
    pydantic_type: type[BaseModel],
    output_str: bool = False,
    *args: Any,
    **kwargs: Any,
) -> dict[str, Any]:
    try:
        return base_model_to_openai_function(
            pydantic_type, output_str, *args, **kwargs
        )
    except Exception as e:
        logger.error(
            f"An error occurred in base_model_to_dict: {e}"
        )
        logger.error(
            "Please check the Pydantic type and ensure it is valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise
```

```
def multi_base_models_to_dict(
    self, return_str: bool = False, *args, **kwargs
) -> dict[str, Any]:
    try:
        if return_str:
            return multi_base_model_to_openai_function(
                self.base_models, *args, **kwargs
            )
        else:
            return multi_base_model_to_openai_function(
                self.base_models, *args, **kwargs
            )
    except Exception as e:
        logger.error(
            f"An error occurred in multi_base_models_to_dict: {e}"
        )
        logger.error(
            "Please check the Pydantic types and ensure they are valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise


def dict_to_openai_schema_str(
    self,
    dict: dict[str, Any],
) -> str:
    try:
        return function_to_str(dict)
    except Exception as e:
        logger.error(
            f"An error occurred in dict_to_openai_schema_str: {e}"
        )
        logger.error(
            "Please check the dictionary and ensure it is valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise


def multi_dict_to_openai_schema_str(
    self,
    dicts: list[dict[str, Any]],
) -> str:
    try:
        return functions_to_str(dicts)
    except Exception as e:
        logger.error(
            f"An error occurred in multi_dict_to_openai_schema_str: {e}"
        )
        logger.error(
            "Please check the dictionaries and ensure they are valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise

def get_docs_from_callable(self, item):
    try:
        return process_tool_docs(item)
    except Exception as e:
        logger.error(f"An error occurred in get_docs: {e}")
        logger.error(
            "Please check the item and ensure it is valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise

def execute_tool(
    self,
    *args: Any,
    **kwargs: Any,
) -> Callable:
    try:
        return openai_tool_executor(
            self.list_of_dicts,
            self.function_map,
            self.verbose,
            *args,
            **kwargs,
        )
    except Exception as e:
        logger.error(f"An error occurred in execute_tool: {e}")
        logger.error(
            "Please check the tools and function map and ensure they are valid."
        )
        logger.error(
            "If the issue persists, please seek further assistance."
        )
        raise

def detect_tool_input_type(self, input: ToolType) -> str:
    if isinstance(input, BaseModel):
        return "Pydantic"
    elif isinstance(input, dict):
        return "Dictionary"
    elif callable(input):
        return "Function"
    else:
        return "Unknown"


def dynamic_run(self, input: Any) -> str:
    """
    Executes the dynamic run based on the input type.


    Args:
        input: The input to be processed.


    Returns:
        str: The result of the dynamic run.


    Raises:
        None


    """
    tool_input_type = self.detect_tool_input_type(input)
    if tool_input_type == "Pydantic":
        function_str = base_model_to_openai_function(input)
    elif tool_input_type == "Dictionary":
        function_str = function_to_str(input)
    elif tool_input_type == "Function":
        function_str = get_openai_function_schema_from_func(input)
    else:
        return "Unknown tool input type"
    if self.auto_execute_tool:
        if tool_input_type == "Function":
            # Add the function to the functions list
            self.tools.append(input)

        # Create a function map from the functions list
        function_map = {
            func.__name__: func for func in self.tools
        }

        # Execute the tool
        return self.execute_tool(
            tools=[function_str], function_map=function_map
        )
    else:
        return function_str

def execute_tool_by_name(
    self,
    tool_name: str,
) -> Any:
    """
    Search for a tool by name and execute it.

    Args:
```

```
    tool_name (str): The name of the tool to execute.



Returns:
    The result of executing the tool.

Raises:
    ValueError: If the tool with the specified name is not found.
    TypeError: If the tool name is not mapped to a function in the function map.
"""
# Search for the tool by name
tool = next(
    (
        tool
        for tool in self.tools
        if tool.get("name") == tool_name
    ),
    None,
)

# If the tool is not found, raise an error
if tool is None:
    raise ValueError(f"Tool '{tool_name}' not found")

# Get the function associated with the tool
func = self.function_map.get(tool_name)
```

```
        # If the function is not found, raise an error
        if func is None:
            raise TypeError(
                f"Tool '{tool_name}' is not mapped to a function"
            )

        # Execute the tool
        return func(**tool.get("parameters", {}))

    def execute_tool_from_text(self, text: str) -> Any:
        """
        Convert a JSON-formatted string into a tool dictionary and execute the tool.

        Args:
            text (str): A JSON-formatted string that represents a tool. The string should be convertible into a dictionary that includes a 'name' key and a 'parameters' key.
            function_map (Dict[str, Callable]): A dictionary that maps tool names to functions.

        Returns:
            The result of executing the tool.

        Raises:
            ValueError: If the tool with the specified name is not found.
            TypeError: If the tool name is not mapped to a function in the function map.
        """
        # Convert the text into a dictionary
        tool = json.loads(text)

        # Get the tool name and parameters from the dictionary
        tool_name = tool.get("name")
        tool_params = tool.get("parameters", {})

        # Get the function associated with the tool
        func = self.function_map.get(tool_name)

        # If the function is not found, raise an error
        if func is None:
            raise TypeError(
                f"Tool '{tool_name}' is not mapped to a function"
            )

        # Execute the tool
        return func(**tool_params)

    def check_str_for_functions_valid(self, output: str):
        """
         Check if the output is a valid JSON string, and if the function name in the JSON matches any name in the function map.

        Args:
            output (str): The output to check.
            function_map (dict): A dictionary mapping function names to functions.


        Returns:
            bool: True if the output is valid and the function name matches, False otherwise.
        """
        try:
            # Parse the output as JSON
            data = json.loads(output)


            # Check if the output matches the schema
            if (
                data.get("type") == "function"
                and "function" in data
                and "name" in data["function"]
            ):


                # Check if the function name matches any name in the function map
                function_name = data["function"]["name"]
                if function_name in self.function_map:
                    return True


        except json.JSONDecodeError:
            logger.error("Error decoding JSON with output")
            pass


        return False
```

```
    def convert_funcs_into_tools(self):
        if self.tools is not None:
            logger.info(
                "Tools provided make sure the functions have documentation ++ type hints, otherwise tool
execution won't be reliable."
            )

            # Log the tools
            logger.info(
                f"Tools provided: Accessing {len(self.tools)} tools"
            )

            # Transform the tools into an openai schema
            self.convert_tool_into_openai_schema()

            # Now update the function calling map for every tools
            self.function_map = {
                tool.__name__: tool for tool in self.tools
            }

        return None

    def convert_tool_into_openai_schema(self):
        logger.info(
            "Converting tools into OpenAI function calling schema"
        )

        tool_schemas = []

        for tool in self.tools:
            # Transform the tool into a openai function calling schema
            if self.check_func_if_have_docs(
                tool
            ) and self.check_func_if_have_type_hints(tool):
                name = tool.__name__
                description = tool.__doc__

                logger.info(
                        f"Converting tool: {name} into a OpenAI certified function calling schema. Add
documentation and type hints."
                )
                tool_schema = get_openai_function_schema_from_func(
                    tool, name=name, description=description
                )

                logger.info(
                    f"Tool {name} converted successfully into OpenAI schema"
                )

                tool_schemas.append(tool_schema)
            else:
                logger.error(
                    f"Tool {tool.__name__} does not have documentation or type hints, please add them to
make the tool execution reliable."
                )

        # Combine all tool schemas into a single schema
        if tool_schemas:
            combined_schema = {
                "type": "function",
                "functions": [
                    schema["function"] for schema in tool_schemas
                ],
            }
            return json.dumps(combined_schema, indent=4)

        return None

    def check_func_if_have_docs(self, func: callable):
        if func.__doc__ is not None:
            return True
        else:
            logger.error(
                f"Function {func.__name__} does not have documentation"
            )
            raise ValueError(
                f"Function {func.__name__} does not have documentation"
            )

    def check_func_if_have_type_hints(self, func: callable):
        if func.__annotations__ is not None:
            return True
        else:
            logger.info(
                f"Function {func.__name__} does not have type hints"
            )
            raise ValueError(
                f"Function {func.__name__} does not have type hints"
            )


# # Example function definitions and mappings
# def get_current_weather(location, unit='celsius'):
#     return f"Weather in {location} is likely sunny and 75° {unit.title()}"

# def add(a, b):
#     return a + b

# # Example tool configurations
# tools = [
#     {
#         "type": "function",
#         "function": {
```

```
#             "name": "get_current_weather",
#             "parameters": {
#                 "properties": {
#                     "location": "San Francisco, CA",
#                     "unit": "fahrenheit",
#                 },
#             },
#         },
#     },
#     {
#         "type": "function",
#         "function": {
#             "name": "add",
#             "parameters": {
#                 "properties": {
#                     "a": 1,
#                     "b": 2,
#                 },
#             },
#         },
#     }
# ]

# function_map = {
#     "get_current_weather": get_current_weather,
#     "add": add,
# }


# # Creating and executing the advanced executor

# tool_executor = BaseTool(verbose=True).execute_tool(tools, function_map)


# try:

#     results = tool_executor()

#     print(results)  # Outputs results from both functions

# except Exception as e:

#     print(f"Error: {e}")
```