```
import os
import subprocess
from loguru import logger
from swarm_models.tiktoken_wrapper import TikTokenizer


class CodeExecutor:
    """
    A class to execute Python code and return the output as a string.

    The class also logs the input and output using loguru and stores the outputs
    in a folder called 'artifacts'.

    Methods:
        execute(code: str) -> str:
            Executes the given Python code and returns the output.
    """

    def __init__(
        self,
        max_output_length: int = 1000,
        artifacts_directory: str = "artifacts",
        language: str = "python3",
    ) -> None:
        """
        Initializes the CodeExecutor class and sets up the logging.
        """
        self.max_output_length = max_output_length
        self.artifacts_dir = artifacts_directory
        self.language = language

        os.makedirs(self.artifacts_dir, exist_ok=True)
        self.setup_logging()
        self.tokenizer = TikTokenizer()

    def setup_logging(self) -> None:
        """
        Sets up the loguru logger with colorful output.
        """
        logger.add(
            os.path.join(self.artifacts_dir, "code_execution.log"),
            format="{time} {level} {message}",
            level="DEBUG",
        )
        logger.info(
            "Logger initialized and artifacts directory set up."
        )

    def format_code(self, code: str) -> str:
        """
        Formats the given Python code using black.
    Args:
        code (str): The Python code to format.

    Returns:
        str: The formatted Python code.

    Raises:
        ValueError: If the code cannot be formatted.
    """
    try:
        import black

        formatted_code = black.format_str(
            code, mode=black.FileMode()
        )
        return formatted_code
    except Exception as e:
        logger.error(f"Error formatting code: {e}")
        raise ValueError(f"Error formatting code: {e}") from e

def execute(self, code: str) -> str:
    """
    Executes the given Python code and returns the output.

    Args:
        code (str): The Python code to execute.
```

```
Returns:
    str: The output of the executed code.


Raises:
    RuntimeError: If there is an error during the execution of the code.
"""
try:
    formatted_code = self.format_code(code)
    logger.info(f"Executing code:\n{formatted_code}")
    completed_process = subprocess.run(
        [self.language, "-c", formatted_code],
        capture_output=True,
        text=True,
        check=True,
    )
    output = completed_process.stdout
    logger.info(f"Code output:\n{output}")
    token_count = self.tokenizer.count_tokens(output)
    print(token_count)


    if (
        self.max_output_length
        and token_count > self.max_output_length
    ):
        logger.warning(
                f"Output length exceeds {self.max_output_length} characters. Truncating output."
            )
            output = output[: self.max_output_length] + "..."

        return output
    except subprocess.CalledProcessError as e:
        logger.error(f"Error executing code: {e.stderr}")
        raise RuntimeError(
            f"Error executing code: {e.stderr}"
        ) from e



# # Example usage:
# if __name__ == "__main__":
#     executor = CodeExecutor(max_output_length=300)
#     code = """
# import requests
# from typing import Any

# def fetch_financial_news(api_key: str, query: str, num_articles: int) -> Any:
#     try:
#         url = f"https://newsapi.org/v2/everything?q={query}&apiKey={api_key}"
#         response = requests.get(url)
#         response.raise_for_status()
#         return response.json()
#     except requests.RequestException as e:
```

```
#         print(f"Request Error: {e}")
#         raise
#     except ValueError as e:
#         print(f"Value Error: {e}")
#         raise

# api_key = ""
# result = fetch_financial_news(api_key, query="Nvidia news", num_articles=5)
# print(result)
#     """
#     result = executor.execute(code)
#     print(result)
```