```
from functools import wraps

from time import time

from typing import Any, Callable


from swarms.utils.loguru_logger import initialize_logger


logger = initialize_logger("try_except_wrapper")



def retry(

    max_retries: int = 3,

) -> Callable[[Callable[..., Any]], Callable[..., Any]]:

    """

    A decorator that retries a function a specified number of times if an exception occurs.


    Args:

        max_retries (int): The maximum number of retries. Default is 3.


    Returns:

        Callable[[Callable[..., Any]], Callable[..., Any]]: The decorator function.

    """


    def decorator_retry(

        func: Callable[..., Any]

    ) -> Callable[..., Any]:

        @wraps(func)
        def wrapper_retry(*args, **kwargs) -> Any:
            """
            The wrapper function that retries the decorated function.

            Args:
                *args: Variable length argument list.
                **kwargs: Arbitrary keyword arguments.

            Returns:
                Any: The result of the decorated function.
            """
            for _ in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    logger.error(f"Error: {e}, retrying...")
            return func(*args, **kwargs)

        return wrapper_retry

    return decorator_retry


def log_execution_time(
    func: Callable[..., Any]
) -> Callable[..., Any]:
```

```
"""
A decorator that logs the execution time of a function.


Args:
    func (Callable[..., Any]): The function to be decorated.


Returns:
    Callable[..., Any]: The decorated function.
"""


@wraps(func)
def wrapper(*args, **kwargs) -> Any:
    """
    The wrapper function that logs the execution time and calls the decorated function.


    Args:
        *args: Variable length argument list.
        **kwargs: Arbitrary keyword arguments.


    Returns:
        Any: The result of the decorated function.
    """
    start = time()
    result = func(*args, **kwargs)
    end = time()
    logger.info(
            f"Execution time for {func.__name__}: {end - start} seconds"
        )
        return result

    return wrapper



def try_except_wrapper(verbose: bool = False):
    """
    A decorator that wraps a function with a try-except block.
    It catches any exception that occurs during the execution of the function,
    prints an error message, and returns None.
    It also prints a message indicating the exit of the function.

    Args:
        func (function): The function to be wrapped.

    Returns:
        function: The wrapped function.

    Examples:
    >>> @try_except_wrapper(verbose=True)
    ... def divide(a, b):
    ...     return a / b
    >>> divide(1, 0)
    An error occurred in function divide: division by zero
```

```
Exiting function: divide
"""


def decorator(func: Callable[..., Any]):
    @wraps(func)
    @retry()
    @log_execution_time
    def wrapper(*args, **kwargs):
        try:
            result = func(*args, **kwargs)
            return result
        except Exception as error:
            if verbose:
                logger.error(
                    f"An error occurred in function {func.__name__}:"
                    f" {error}"
                )
            else:
                logger.error(
                    f"An error occurred in function {func.__name__}:"
                    f" {error}"
                )
                return None
        finally:
            print(f"Exiting function: {func.__name__}")
        return wrapper

    return decorator



# @try_except_wrapper(verbose=True)
# def divide(a, b):
#     """Multiply two numbers."""
#     return a / b



# # This will work fine
# result = divide(2, 0)
# print(result)  # Output: 6
```