**Title:**

Python Code Refactoring and Unit Testing – Task Report

Internship @ CyArt

Submitted by: Vedant Patil

**Introduction:**

This report documents the refactoring and testing task assigned during the second day of my internship at CyArt. The goal was to transform an unstructured Python script into a modular, testable system using advanced programming principles such as **dataclasses**, **dunder** **methods**, **context** **managers**, and **SOLID** design principles. The task also included unit testing using **pytest**.

# **Dataclasses**

Python’s **@dataclass** decorator was used to create structured, boilerplate-free classes. It automatically generates methods like **\_\_init\_\_, \_\_repr\_\_, and \_\_eq\_\_**, making code more readable and maintainable.

Ex.

from dataclasses import dataclass

@dataclass

class LogRecord:

time\_stamp: float

message: str

level: str

description: str

# **Dunder Methods and Context Managers**

Implemented **\_\_enter\_\_** and **\_\_exit\_\_** to create a custom context manager class Timer, which logs execution time of code blocks.

Ex.

class Timer:

def \_\_enter\_\_(self):

self.start\_time = time.time()

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

self.duration = time.time() - self.start\_time

# **SOLID Principles**

* **Single Responsibility**: Each module/class has one clear purpose (e.g., logging, timing)
* **Open/Closed Principle**: Easily extensible without modifying core classes.
* **Liskov Substitution, Interface Segregation, Dependency Inversion**: Understood conceptually.

# **Factory Pattern**

Studied conceptually: It helps dynamically instantiate classes based on input, useful for scalable logging systems.

# **Pytest**

Lightweight test framework used to validate functionality of dataclass and timing context.

Original Problem:

The provided script combined logic, timing, and output into a single function, making it hard to scale or test:

def process\_log():

start = time.time()

logs = ["error: failed", "info: started"]

for log in logs:

print(f"Log: {log}")

print(f"Time taken: {time.time() - start}")

|  |  |
| --- | --- |
| **File** | **Description** |
| <log_record.py> | Defines **LogRecord** dataclass for structured logging |
| <timer_context.py> | Implements **Timer** context manager using dunder methods |
| <main.py> | Uses both modules to log simulated events and measure time |
| <test_timer.py> | Tests that **Timer** correctly logs execution duration using **pytest** |

**Main Code Highlights**

log\_record.py

from dataclasses import dataclass

from typing import Literal

@dataclass

class LogRecord:

time\_stamp: float

message: str

level: Literal["INFO", "WARNING", "ERROR", "DEBUG"]

description: str

timer\_context.py

import time

from log\_record import LogRecord

class Timer:

def \_\_init\_\_(self, message: str = "Execution completed", level: str = "INFO"):

self.message = message

self.level = level

self.start\_time = 0.0

def \_\_enter\_\_(self):

self.start\_time = time.time()

return self

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

end\_time = time.time()

self.duration = end\_time - self.start\_time

log = LogRecord(

time\_stamp=end\_time,

message=self.message,

level=self.level,

description=f"Time taken: {self.duration:.4f} seconds"

)

print(log)

main.py

from timer\_context import Timer

from log\_record import LogRecord

import time

def process\_logs():

logs = [

("Failed to connect", "ERROR"),

("System booted", "INFO"),

("Retrying...", "WARNING"),

]

for message, level in logs:

log = LogRecord(

time\_stamp=time.time(),

message=message,

level=level,

description="Logged event"

)

print(log)

if \_\_name\_\_ == "\_\_main\_\_":

with Timer("Processing logs", level="INFO"):

process\_logs()

test\_timer.py

from timer\_context import Timer

import time

def test\_timer\_execution\_time():

with Timer("Testing", level="DEBUG") as t:

time.sleep(0.1)

assert 0.09 <= t.duration <= 0.2

**How to run:**

pytest test\_timer.py

**Output:**

**===================== test session starts =====================**

**collected 1 item**

**test\_timer.py . [100%]**

**====================== 1 passed in 0.11s ======================**

**Conclusion:**

This task taught me how to write modular, testable Python code using advanced features like **dataclasses**, dunder methods, and context managers. I also practiced applying clean code principles and used **pytest** to validate execution time accurately. These are essential skills for large-scale AI and software systems.

**References:**

* <https://docs.python.org/3/>
* <https://realpython.com/>
* Stack overflow
* Blog posts on dataclasses, context managers and SOLID design princlples