```
from functools import wraps
from typing import Callable, Generator

# 데코레이터 함수: 함수의 실행 시간을 측정하는 기능을 추가함
def timeit(func: Callable) -> Callable:
    import time

    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f"실행 시간: {end_time - start_time:.4f}초")
        return result
    return wrapper

# 제너레이터 함수: 메모리를 아끼기 위해 한 번에 하나의 값을 생성
def fibonacci(n: int) -> Generator[int, None, None]:
    """n번째 피보나치 수열까지 생성"""
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# 고차 함수: 다른 함수를 인자로 받아 처리
def apply_operation(data: list[int], operation: Callable[[int], int]) -> list[int]:
    """주어진 함수(operation)를 데이터 리스트에 적용"""
    return [operation(x) for x in data]

# 함수형 프로그래밍: 람다 함수 사용
square = lambda x: x ** 2
cube = lambda x: x ** 3

# 주요 실행 코드
@timeit  # 데코레이터 적용: 실행 시간을 측정
def main():
    fib_list = list(fibonacci(10))  # 10번째 피보나치 수열까지 생성
    print("피보나치 수열:", fib_list)

    # 제곱 함수 적용
    squared_list = apply_operation(fib_list, square)
    print("제곱된 값:", squared_list)

    # 세제곱 함수 적용
    cubed_list = apply_operation(fib_list, cube)
    print("세제곱된 값:", cubed_list)

# main 함수 실행
if __name__ == "__main__":
    main()
```

```python
# 메타클래스 정의: 클래스를 생성하는 클래스를 정의함
class SingletonMeta(type):
    """싱글톤 패턴 구현을 위한 메타클래스"""
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

# 싱글톤 클래스: 동일한 인스턴스만 생성
class SingletonClass(metaclass=SingletonMeta):
    def __init__(self, value):
        self.value = value

# 컨텍스트 관리자: 특정 자원의 사용을 관리
class FileManager:
    """파일을 안전하게 열고 닫는 컨텍스트 관리자"""
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        print(f"{self.filename} 파일을 엽니다.")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_value, traceback):
        print(f"{self.filename} 파일을 닫습니다.")
        if self.file:
            self.file.close()

# 데코레이터 체이닝: 여러 데코레이터를 체인 형태로 적용
def bold(func):
    """HTML 태그를 사용해 텍스트를 굵게 만듦"""
    def wrapper(*args, **kwargs):
        return f"<b>{func(*args, **kwargs)}</b>"
    return wrapper

def italic(func):
    """HTML 태그를 사용해 텍스트를 기울임"""
    def wrapper(*args, **kwargs):
        return f"<i>{func(*args, **kwargs)}</i>"
    return wrapper

@bold  # 먼저 bold 데코레이터 적용
@italic  # 그다음 italic 데코레이터 적용
def get_text():
    return "Python 고급 프로그래밍"

# 주요 실행 코드
if __name__ == "__main__":
    # 싱글톤 클래스 사용
    s1 = SingletonClass("첫 번째 인스턴스")
    s2 = SingletonClass("두 번째 인스턴스")

    print(f"s1과 s2는 같은 객체인가? {s1 is s2}")
    print(f"s1의 값: {s1.value}")
    print(f"s2의 값: {s2.value}")

    # 컨텍스트 관리자 사용
    with FileManager("sample.txt", "w") as file:
        file.write("파일에 텍스트를 씁니다.")

    # 데코레이터 체이닝 예시
    print(get_text())
```