Zadanie 1.

Napisz dekorator `Czasowy`, który będzie dodawał do dekorowanej metody pomiar czasu jej wykonania. Dekorator powinien mieć następującą sygnaturę:

```
def dekorator(funkcja):
    pass
```

Dekorator powinien działać w następujący sposób:

- Przed wywołaniem dekorowanej metody, dekorator powinien zapisać do zmiennej czas rozpoczęcia jej wykonania.
- Po zakończeniu wykonywania dekorowanej metody, dekorator powinien zapisać do zmiennej czas zakończenia jej wykonania.
- Na koniec, dekorator powinien zwrócić różnicę między czasem rozpoczęcia a czasem zakończenia wykonania dekorowanej metody.

Następnie wykorzystaj dekorator `Czasowy` do pomiaru czasu wykonania następujących metod:

```
def metoda1():
    for i in range(1000000):
        pass

def metoda2():
    for i in range(1000000):
        print(i)
```

Zadanie 2.

Napisz iterator `IteratorLiczb`, który będzie iterował po zbiorze liczb całkowitych od 0 do zadanego limitu. Iterator powinien mieć następującą sygnaturę:

```
class IteratorLiczb:
    def __init__(self, limit):
        pass

    def __next__(self):
        pass
```

Zadanie 3:

Napisz iterator `IteratorLiczb`, który będzie iterował po zbiorze liczb całkowitych od 0 do zadanego limitu. Iterator powinien mieć następującą sygnaturę:

```
class IteratorLiczb:
    def __init__(self, limit):
        pass

    def __next__(self):
        pass
```

Iterator powinien działać w następujący sposób:

- W konstruktorze iteratora należy ustawić zmienną `limit`, która będzie określać maksymalną wartość liczby, po której iterator powinien zakończyć iterację.
- Metoda `__next__()` powinna zwracać kolejną liczbę z zakresu od 0 do `limit`.

Następnie wykorzystaj iterator `IteratorLiczb` do wydrukowania na konsolę wszystkich liczb całkowitych od 0 do 100.

Zadanie 4.
Zdefiniuj klasę **`KolekcjaLiczb`**, która będzie przechowywała listę liczb. Dodaj dekorator **`@przechwytuj_zerowe`**, który będzie sprawdzał, czy dodawana liczba do kolekcji jest równa zeru. Jeśli tak, nie dodawaj jej do kolekcji, w przeciwnym razie dodaj. Następnie zaimplementuj iterator, który pozwoli na iterację tylko po dodatnich liczbach z kolekcji. Przetestuj działanie dekoratora i iteratora na instancji klasy.

Zadanie 5.
Zdefiniuj abstrakcyjną klasę `ZestawDanych` z metodą abstrakcyjną `generuj_dane`, która powinna być implementowana przez konkretne klasy dziedziczące. Następnie stwórz klasę dziedziczącą po `ZestawDanych` o nazwie `DaneLosowe`, która implementuje metodę `generuj_dane` zwracającą listę losowych liczb. Użyj generatora do wygenerowania danych w sposób leniwy (lazy), tzn. dane powinny być generowane na żądanie. Stwórz instancję klasy `DaneLosowe` i przetestuj generowanie danych w pętli.

Zaadanie 6.
Stwórz klasę `Fibonacci`, która będzie dziedziczyła po klasie `Iterable`. Klasa powinna mieć metodę magiczną `__iter__`, która będzie zwracała generator generujący liczby Fibonacciego. Wykorzystaj generatory do implementacji funkcji generującej ciąg Fibonacciego. Przetestuj działanie klasy `Fibonacci`, używając pętli `for` do iteracji przez kilka pierwszych liczb Fibonacciego.