Day2 - OOP, Generators, Iterators & Decorators

Problem 1: Vehicle Rental System (OOP - Beginner)

Learning Objectives:

- Class creation and inheritance
- Method overriding
- Instance and class attributes
- Basic encapsulation

Difficulty: Beginner

Estimated Time: 35 minutes

Problem Description:

Create a vehicle rental system with the following requirements:

Base Class - Vehicle:

- Attributes: `vehicle_id`, `brand`, `model`, `rental_price_per_day`, `is_rented`
- Methods:
  - `rent()` - Mark vehicle as rented
  - `return_vehicle()` - Mark vehicle as available
  - `calculate_rental_cost(days)` - Calculate total cost
  - `get_details()` - Return vehicle information

Derived Classes:

- **Car:** Additional attribute `num_doors`, rental price multiplier: 1.0
- **Motorcycle:** Additional attribute `engine_cc`, rental price multiplier: 0.7
- **Truck:** Additional attribute `cargo_capacity_tons`, rental price multiplier: 1.5

Class Variable: `total_vehicles` to track all vehicles created

Expected Output:

```python
# Create vehicles
car = Car("V001", "Toyota", "Camry", 50, 4)
motorcycle = Motorcycle("V002", "Harley", "Street 750", 40, 750)
truck = Truck("V003", "Ford", "F-150", 80, 2.5)

# Rent a vehicle
car.rent()
print(car.is_rented)  # Output: True

# Calculate rental cost
cost = car.calculate_rental_cost(5)
print(f"Rental cost for 5 days: ${cost}")
# Output: Rental cost for 5 days: $250.0

# Get details
print(motorcycle.get_details())
# Output: Motorcycle - Harley Street 750 (750cc) -
#         ID: V002 - $40/day - Available

# Try to rent already rented vehicle car.rent()
# Output: Vehicle is already rented

# Return vehicle
car.return_vehicle()
print(car.is_rented)  # Output: False

# Check total vehicles
print(f"Total vehicles: {Vehicle.total_vehicles}")
# Output: Total vehicles: 3
```

Hints:

- Use `super()` to call parent class constructor
- Implement validation in `rent()` method
- Use f-strings for formatted output
- Use class variable with incrementation in `__init__`

Bonus Challenge:

Add a `RentalAgency` class that manages multiple vehicles and tracks rental history.

Problem 2: Custom Range Iterator (Iterators - Beginner)

Learning Objectives:

- Implement `__iter__()` and `__next__()` methods
- Understand iterator protocol
- Handle StopIteration exception
- Create flexible iterators with parameters

Difficulty: Beginner

Estimated Time: 25 minutes

Problem Description:

Create a custom iterator class called `CustomRange` that works like Python's built-in `range()` but with additional features:

Requirements:

1. Support start, stop, and step parameters
2. Support negative step (counting down)
3. Support float step values
4. Implement `__iter__()` and `__next__()` methods
5. Add a `reset()` method to restart iteration

Additional Feature:
Create another iterator `EvenNumbers` that yields only even numbers in a given range.

Expected Output:

```
# Basic usage
print("Custom Range 0 to 10:")
for num in CustomRange(10):
    print(num, end=" ")
# Output: 0 1 2 3 4 5 6 7 8 9
```

```
print("\n\nCustom Range 5 to 15, step 2:")
for num in CustomRange(5, 15, 2):
    print(num, end=" ")
# Output: 5 7 9 11 13

print("\n\nCustom Range 10 to 0, step -2:")
for num in CustomRange(10, 0, -2):
    print(num, end=" ")
# Output: 10 8 6 4 2

print("\n\nFloat step - 0 to 2, step 0.5:")
for num in CustomRange(0, 2, 0.5):
    print(num, end=" ")
# Output: 0 0.5 1.0 1.5

print("\n\nEven Numbers 1 to 20:")
for num in EvenNumbers(1, 20):
    print(num, end=" ")
# Output: 2 4 6 8 10 12 14 16 18

# Test reset
print("\n\nTest Reset:")
my_range = CustomRange(3)
for num in my_range:
    print(num, end=" ")
print()
my_range.reset()
for num in my_range:
    print(num, end=" ")
# Output: 0 1 2
#         0 1 2
```

Hints:

- Store start, stop, step as instance variables
- Keep current value as instance variable
- Check conditions in `__next__()` before returning value
- Raise `StopIteration` when done
- `__iter__()` should return `self`

Bonus Challenge:

Add methods `to_list()` and `__len__()` to convert iterator to list and get count.

Problem 3: File Line Generator (Generators - Beginner)

Learning Objectives:

- Create generator functions using `yield`
- Understand lazy evaluation
- Process files efficiently
- Chain multiple generators

Difficulty: Beginner

Estimated Time: 30 minutes

Problem Description:

Create several generator functions for efficient file processing:

Generator Functions to Implement:

1. **`read_file_lines(filename)`** : Read file line by line (memory efficient)
2. **`filter_lines(lines, keyword)`** : Filter lines containing a keyword
3. **`strip_lines(lines)`** : Remove leading/trailing whitespace
4. **`number_lines(lines)`** : Add line numbers
5. **`chunk_lines(lines, chunk_size)`** : Group lines into chunks

Expected Output:

```
# Create a sample file for testing
sample_content = """
Hello World
Python Programming
This is a test
Python is awesome
Final line
"""

# Write sample file
```

```
with open('sample.txt', 'w') as f:
    f.write(sample_content)

# Read file using generator
print("All lines:")
for line in read_file_lines('sample.txt'):
    print(repr(line))
# Output:
# '\n'
# 'Hello World\n'
# 'Python Programming\n'
# 'This is a test\n'
# 'Python is awesome\n'
# 'Final line\n'

# Chain generators - filter and strip
print("\nFiltered lines (containing 'Python'):")
lines = read_file_lines('sample.txt')
filtered = filter_lines(lines, 'Python')
stripped = strip_lines(filtered)
for line in stripped:
    print(line)
# Output:
# Python Programming
# Python is awesome

# Number lines
print("\nNumbered lines:")
lines = read_file_lines('sample.txt')
stripped = strip_lines(lines)
numbered = number_lines(stripped)
for line in numbered:
    print(line)
# Output:
# 1:
# 2: Hello World
# 3: Python Programming
# 4: This is a test
# 5: Python is awesome
# 6: Final line

# Chunk lines
print("\nChunked lines (size 2):")
lines = read_file_lines('sample.txt')
```

```
stripped = strip_lines(lines)
chunked = chunk_lines(stripped, 2)
for chunk in chunked:
    print(chunk)
# Output:
# ['', 'Hello World']
# ['Python Programming', 'This is a test']
# ['Python is awesome', 'Final line']
```

Hints:

- Use `with open()` in generator
- Use `yield` to return values one at a time
- Generators can take other generators as input
- Use `if keyword in line` for filtering
- Collect lines in a list for chunking

Bonus Challenge:

Create a `reverse_lines()` generator that yields lines in reverse order (from end of file).

Problem 4: Performance Timer & Logger Decorators (Decorators - Beginner)

Learning Objectives:

- Create function decorators
- Understand wrapper functions
- Use `functools.wraps`
- Pass arguments through decorators

Difficulty: Beginner

Estimated Time: 30 minutes

Problem Description:

Create useful decorators for debugging and monitoring functions:

Decorators to Implement:

1. `@timer` : Measure and print execution time
2. `@logger` : Log function calls with arguments and return values
3. `@count_calls` : Count how many times a function is called
4. `@debug` : Print function name, arguments, and return value in detail

Expected Output:

```python
import time
import functools

# Timer decorator
@timer
def slow_function(n):
    time.sleep(n)
    return f"Slept for {n} seconds"

result = slow_function(2)
# Output: [TIMER] Function 'slow_function' executed in 2.0023 seconds
print(result)
# Output: Slept for 2 seconds

# Logger decorator
@logger
def add(a, b):
    return a + b

result = add(5, 3)
# Output: [LOGGER] Calling function 'add' with args=(5, 3), kwargs={}
# Output: [LOGGER] Function 'add' returned: 8

# Count calls decorator
@count_calls
def greet(name):
    return f"Hello, {name}!"

greet("Alice")
greet("Bob")
greet("Charlie")
print(f"Total calls: {greet.call_count}")
```

```
# Output: Total calls: 3

# Debug decorator
@debug
def calculate_average(numbers):
    return sum(numbers) / len(numbers)

result = calculate_average([10, 20, 30, 40])
# Output: [DEBUG] Calling 'calculate_average'
# Output: [DEBUG]   args: ([10, 20, 30, 40],)
# Output: [DEBUG]   kwargs: {}
# Output: [DEBUG] 'calculate_average' returned: 25.0

# Stack multiple decorators
@timer
@logger
def multiply(x, y):
    return x * y

result = multiply(4, 5)
# Output: [LOGGER] Calling function 'multiply' with args=(4, 5), kwargs={
# Output: [LOGGER] Function 'multiply' returned: 20
# Output: [TIMER] Function 'multiply' executed in 0.0001 seconds
```

Hints:

- Import `time` module for timing
- Use `time.time()` or `time.perf_counter()`
- Use `functools.wraps(func)` to preserve function metadata
- Store call count as function attribute
- Decorator structure: `def decorator(func): def wrapper(*args, **kwargs): ... return wrapper`

Bonus Challenge:

Create a `@slow_down(seconds)` decorator that adds a delay before function execution.

Problem 5: E-commerce Product System (OOP - Intermediate)

Learning Objectives:

- Complex class relationships
- Composition over inheritance
- Special methods (`__str__` , `__repr__` , `__eq__`)
- Property decorators (`@property`)

Difficulty: Intermediate

Estimated Time: 45 minutes

Problem Description:

Create an e-commerce product system with multiple interacting classes:

Classes to Implement:

1. **Product:**
   - Attributes: `product_id` , `name` , `price` , `stock` , `category`
   - Methods: `add_stock()` , `reduce_stock()` , `apply_discount(percentage)`
   - Properties: `is_available` (read-only)
   - Special methods: `__str__` , `__repr__` , `__eq__`
2. **Review:**
   - Attributes: `user` , `rating` (1-5), `comment`
   - Method: `is_positive()` (returns True if rating >= 4)
3. **ProductWithReviews (inherits Product):**
   - Additional: `reviews` list
   - Methods: `add_review()` , `average_rating()` , `get_review_summary()`
4. **ShoppingCart:**
   - Attributes: `items` (dict: product -> quantity)
   - Methods: `add_item()` , `remove_item()` , `update_quantity()` , `get_total()` , `clear()`
   - Special method: `__len__` (return total items)

Expected Output:

```python
# Create products
laptop = ProductWithReviews("P001", "Gaming Laptop",
         1200, 10, "Electronics")
mouse = ProductWithReviews("P002", "Wireless Mouse",
         25, 50, "Accessories")

print(laptop)
# Output: Gaming Laptop (P001) - $1200 - Stock: 10

# Check availability
print(f"Laptop available: {laptop.is_available}")
# Output: Laptop available: True

# Add reviews
laptop.add_review(Review("Alice", 5, "Excellent laptop!"))
laptop.add_review(Review("Bob", 4, "Great performance"))
laptop.add_review(Review("Charlie", 3, "Good but expensive"))

print(f"Average rating: {laptop.average_rating():.2f}")
# Output: Average rating: 4.00

print(laptop.get_review_summary())
# Output: 3 reviews - Average: 4.0/5 - 2 positive

# Shopping cart
cart = ShoppingCart()
cart.add_item(laptop, 1)
cart.add_item(mouse, 2)

print(f"Cart size: {len(cart)}")
# Output: Cart size: 3

print(f"Cart total: ${cart.get_total()}")
# Output: Cart total: $1250

# Apply discount
laptop.apply_discount(10)
print(f"Discounted price: ${laptop.price}")
# Output: Discounted price: $1080.0

print(f"New cart total: ${cart.get_total()}")
# Output: New cart total: $1130.0

# Reduce stock when purchasing
```

```
for product, quantity in cart.items.items():
    product.reduce_stock(quantity)

print(f"Laptop stock after purchase: {laptop.stock}")
# Output: Laptop stock after purchase: 9

# Test equality
laptop2 = ProductWithReviews("P001", "Gaming Laptop",
          1200, 10, "Electronics")
print(f"laptop == laptop2: {laptop == laptop2}")
# Output: laptop == laptop2: True
```

Hints:

- Use `@property` decorator for read-only attributes
- Implement `__eq__` to compare by `product_id`
- Store cart items as `{product: quantity}` dictionary
- Calculate total by iterating cart items
- Validate stock before reducing

Bonus Challenge:

Add a `Coupon` class that can apply different discount types (percentage, fixed amount, buy-one-get-one).

Problem 6: Infinite Sequence Generators (Generators - Intermediate)

Learning Objectives:

- Create infinite generators
- Use `itertools` module
- Generator expressions
- Yield from syntax

Difficulty: Intermediate

Estimated Time: 35 minutes

Problem Description:

Create several infinite sequence generators and utilities:

Generators to Implement:

1. `infinite_counter(start=0, step=1)` : Infinite counting sequence
2. `cycle_list(items)` : Cycle through a list infinitely
3. `fibonacci_infinite()` : Infinite Fibonacci sequence
4. `prime_generator_infinite()` : Infinite prime number generator
5. `alternating(*generators)` : Alternate between multiple generators
6. `take(n, generator)` : Take first n items from a generator
7. `skip(n, generator)` : Skip first n items from a generator

Expected Output:

```
from itertools import islice

# Infinite counter
print("First 10 numbers from counter:")
counter = infinite_counter(1, 2)
for num in take(10, counter):
    print(num, end=" ")
# Output: 1 3 5 7 9 11 13 15 17 19

print("\n\nCycle through list:")
colors = cycle_list(['red', 'green', 'blue'])
for color in take(7, colors):
    print(color, end=" ")
# Output: red green blue red green blue red

print("\n\nFirst 15 Fibonacci numbers:")
fib = fibonacci_infinite()
for num in take(15, fib):
    print(num, end=" ")
# Output: 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

print("\n\nFirst 10 prime numbers:")
primes = prime_generator_infinite()
for p in take(10, primes):
    print(p, end=" ")
# Output: 2 3 5 7 11 13 17 19 23 29
```

```
print("\n\nAlternating generators:")
gen1 = infinite_counter(1)
gen2 = infinite_counter(100, 100)
alt = alternating(gen1, gen2)
for num in take(8, alt):
    print(num, end=" ")
# Output: 1 100 2 200 3 300 4 400

print("\n\nSkip first 5 from counter:")
counter = infinite_counter()
skipped = skip(5, counter)
for num in take(5, skipped):
    print(num, end=" ")
# Output: 5 6 7 8 9

# Using generator with condition
print("\n\nFirst 10 even Fibonacci numbers:")
fib = fibonacci_infinite()
even_fib = (x for x in fib if x % 2 == 0)
for num in take(10, even_fib):
    print(num, end=" ")
# Output: 0 2 8 34 144 610 2584 10946 46368 196418
```

Hints:

- Use `while True:` for infinite loops
- Use `yield from` to delegate to another generator
- For primes, check divisibility up to sqrt(n)
- For alternating, use a list to store generators and index
- `take()` and `skip()` can use regular loops with `next()`

Bonus Challenge:

Create `merge_sorted(*generators)` that merges multiple sorted infinite generators into one sorted stream.

Problem 7: Advanced Decorator Patterns (Decorators - Intermediate)

Learning Objectives:

- Decorators with arguments
- Class-based decorators
- Decorator factory pattern
- Preserving function metadata

Difficulty: Intermediate

Estimated Time: 40 minutes

Problem Description:

Create advanced decorators with more complex functionality:

Decorators to Implement:

1. `@repeat(n)` : Execute function n times
2. `@validate_types(**type_hints)` : Validate argument types
3. `@rate_limit(max_calls, time_window)` : Limit function calls per time window
4. `@memoize_with_expiry(seconds)` : Cache with expiration time
5. `@singleton` : Ensure only one instance of a class exists (class decorator)

Expected Output:

```
import time
from datetime import datetime, timedelta

# Repeat decorator
@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("World")
# Output:
# Hello, World!
# Hello, World!
# Hello, World!

# Type validation decorator
@validate_types(a=int, b=int)
```

```
def add(a, b):
    return a + b

result = add(5, 3)
print(f"5 + 3 = {result}")
# Output: 5 + 3 = 8

try:
    add(5, "3")
except TypeError as e:
    print(f"Error: {e}")
# Output: Error: Argument 'b' must be of type <class 'int'>,
                 got <class 'str'>

# Rate limit decorator
@rate_limit(max_calls=3, time_window=5)
def api_call():
    return "API response"

print(api_call())  # Success
print(api_call())  # Success
print(api_call())  # Success
try:
    print(api_call())  # Should fail
except Exception as e:
    print(f"Rate limit: {e}")
# Output: Rate limit: Rate limit exceeded. Try again in X seconds.

# Wait and try again
time.sleep(5)
print(api_call())  # Success again
# Output: API response

# Memoize with expiry
@memoize_with_expiry(seconds=3)
def expensive_calculation(x):
    print(f"Computing for {x}...")
    time.sleep(1)
    return x ** 2

print(expensive_calculation(5))  # Computes
# Output: Computing for 5...
#         25
```

```
print(expensive_calculation(5))  # Returns cached
# Output: 25

time.sleep(3)
print(expensive_calculation(5))  # Cache expired, recomputes
# Output: Computing for 5...
#         25

# Singleton decorator
@singleton
class Database:
    def __init__(self):
        print("Initializing database connection...")
        self.connection = "Connected"

db1 = Database()
# Output: Initializing database connection...

db2 = Database()
# (No output - returns same instance)

print(f"db1 is db2: {db1 is db2}")
# Output: db1 is db2: True
```

Hints:

- Decorator with arguments: `def decorator(arg): def actual_decorator(func): def wrapper...`
- Use `inspect` module to get function signature
- For rate limit, store call times in a list
- For memoize with expiry, store (value, timestamp) tuples
- For singleton, store instance as decorator attribute

Bonus Challenge:

Create `@async_timeout(seconds)` decorator that raises TimeoutError if async function takes too long.

Problem 8: Custom Collection Classes (OOP + Iterators - Intermediate)

Learning Objectives:

- Implement collection protocols
- Special methods for container types
- Make custom classes iterable
- Context managers

Difficulty: Intermediate

Estimated Time: 50 minutes

Problem Description:

Create custom collection classes that behave like built-in Python collections:

Classes to Implement:

1. `Stack` : LIFO (Last In First Out) collection
   - Methods: `push()` , `pop()` , `peek()` , `is_empty()` , `size()`
   - Special methods: `__len__` , `__str__` , `__iter__`
2. `Queue` : FIFO (First In First Out) collection
   - Methods: `enqueue()` , `dequeue()` , `front()` , `is_empty()` , `size()`
   - Special methods: `__len__` , `__str__` , `__iter__`
3. `CircularBuffer` : Fixed-size buffer that overwrites oldest data
   - Methods: `append()` , `get_all()` , `is_full()` , `clear()`
   - Special methods: `__len__` , `__getitem__` , `__setitem__` , `__iter__`
4. `UniqueList` : List that automatically prevents duplicates
   - Methods: `append()` , `extend()` , `remove()` , `count()` , `index()`
   - Special methods: `__len__` , `__getitem__` , `__contains__` , `__iter__`

Expected Output:

```python
# Stack implementation
print("=== Stack Demo ===")
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)
print(f"Stack: {stack}")
# Output: Stack: [1, 2, 3]

print(f"Peek: {stack.peek()}")
# Output: Peek: 3

print(f"Pop: {stack.pop()}")
# Output: Pop: 3

print(f"Stack after pop: {stack}")
# Output: Stack after pop: [1, 2]

print("Iterate through stack:")
for item in stack:
    print(item, end=" ")
# Output: 1 2

# Queue implementation
print("\n\n=== Queue Demo ===")
queue = Queue()
queue.enqueue("First")
queue.enqueue("Second")
queue.enqueue("Third")
print(f"Queue: {queue}")
# Output: Queue: [First, Second, Third]

print(f"Front: {queue.front()}")
# Output: Front: First

print(f"Dequeue: {queue.dequeue()}")
# Output: Dequeue: First

print(f"Queue after dequeue: {queue}")
# Output: Queue after dequeue: [Second, Third]

# Circular Buffer
print("\n=== Circular Buffer Demo ===")
buffer = CircularBuffer(3)
```

```python
buffer.append(1)
buffer.append(2)
buffer.append(3)
print(f"Buffer: {buffer.get_all()}")
# Output: Buffer: [1, 2, 3]

print(f"Is full: {buffer.is_full()}")
# Output: Is full: True

buffer.append(4)  # Overwrites oldest (1)
print(f"Buffer after adding 4: {buffer.get_all()}")
# Output: Buffer after adding 4: [2, 3, 4]

print(f"Access by index - buffer[1]: {buffer[1]}")
# Output: Access by index - buffer[1]: 3

buffer[0] = 10
print(f"After setting buffer[0] = 10: {buffer.get_all()}")
# Output: After setting buffer[0] = 10: [10, 3, 4]

# Unique List
print("\n=== Unique List Demo ===")
unique = UniqueList()
unique.append(1)
unique.append(2)
unique.append(3)
unique.append(2)  # Duplicate, won't be added
print(f"Unique list: {list(unique)}")
# Output: Unique list: [1, 2, 3]

print(f"Length: {len(unique)}")
# Output: Length: 3

print(f"Contains 2: {2 in unique}")
# Output: Contains 2: True

unique.extend([4, 5, 3, 6])  # 3 is duplicate
print(f"After extend: {list(unique)}")
# Output: After extend: [1, 2, 3, 4, 5, 6]

print(f"Index of 4: {unique.index(4)}")
# Output: Index of 4: 3

unique.remove(2)
```

```
print(f"After removing 2: {list(unique)}")
# Output: After removing 2: [1, 3, 4, 5, 6]
```

Hints:

- Use internal list to store data
- Implement `__iter__` to make class iterable
- For CircularBuffer, use modulo operator for circular indexing
- For UniqueList, check existence before adding
- Raise appropriate exceptions (IndexError, ValueError)

Bonus Challenge:

Add context manager support (`__enter__` and `__exit__`) to automatically save collection state to a file.