Homework No. 02

Due: 23:59, 19 March, 2025

Max points: 100

Rules

- **No late homeworks.** A penalty of 10 points is applied for each day.
- **No plagiarism.** Collaboration is encouraged, but copying someone else's work without proper attribution is not admitted and invalidates the submission. A penalty is applied to all parties included.
- **Responsible AI Use.** AI assistants (e.g. ChatGPT) may be used as a learning tool, but the primary goal of homework is to develop your own problem-solving and coding skills. AI should be used minimally and responsibly with proper understanding and attribution. Submissions that rely excessively on AI without demonstrating personal effort may receive penalties.

Submission procedure

- A single Jupyter Notebook file can be used. The following naming convention should be used: `homework{number}_{student name}.ipynb`. For example, `homework02_Jane_Dane.ipynb`.
- At the start of the file, homework number and student full name should be mentioned. Problem solutions should be clearly separated by problem numbers. For example:

```
"""
# Homework 02
## Name: Jane Dane

### Problem 1

...

### Problem 2

...
"""
```

- Solution files should be uploaded to YSU Moodle. Alternatively, you can commit your solutions to a Git repository and provide the repository URL on Moodle.

Instructions

Type Hinting

Please use type hints wherever possible. This will help improve the readability and maintainability of your code.

While we encourage using type hints, incorrect or incomplete type hinting **will not** result in penalties. The goal is to familiarize yourself with their usage, so feel free to use them to the best of your ability.

Problem 1 [12 points]

Note: Dataclasses are not allowed to be used in this problem.

Write a `BankAccount` class in Python, which models a bank account of a customer. The class should consist of the following components:

- **Properties**
  - `id`: A unique identifier for an account.
  - `name`: The full name of a customer.
  - `balance`: The balance of an account, which should be 0 by default.

  Getters and setters should be defined for the properties.

- **Methods**
  - It should be possible to initialize an instance either with initial balance or without initial balance.
  - Friendly string representation for an account should be implemented.
  - `deposit(amount)`: adds the given amount to the current balance.
  - `withdraw(amount)`: subtracts the given amount from the current balance. If there are insufficient funds, it should raise an error (`ValueError` can be used).
  - `transfer_to(another_account, amount)`: transfers the given amount from the current account to the given account. If there are insufficient funds, it should raise an error (`ValueError` can be used).

Example

```python
class BankAccount:
    pass

account_1 = BankAccount(1, "John Doe")
account_2 = BankAccount(2, "Jane Dane", 1000)

print(account_1) # BankAccount(id=1, name="John Doe", balance=0)
print(account_2) # BankAccount(id=2, name="Jane Dane", balance=1000)

account_1.deposit(500)
print(account_1) # BankAccount(id=1, name="John Doe", balance=500)
account_1.withdraw(600) # raises an error

account_2.transfer_to(account_1, 250)
print(account_1) # BankAccount(id=1, name="John Doe", balance=750)
print(account_2) # BankAccount(id=2, name="Jane Dane", balance=750)
account_2.transfer_to(account_1, 800) # raises an error
```

Problem 2 [12 points]

Write a superclass `Shape` and its subclasses `Circle` and `Rectangle` in Python.

- The `Shape` class should consist of the following components:
  - **Properties**
    - `color`: A color that indicates the color of a shape.
    - `is_filled`: A boolean flag that indicates if a shape is filled or not.

    Getters and setters should be defined for the properties.
  - **Methods**
    - It should be possible to initialize an instance by providing a color and whether the shape is filled or not.
    - Friendly string representation for a shape should be implemented.
    - `calculate_area()`: It should raise an error (`NotImplementedError` can be used).
    - `calculate_perimeter()`: It should raise an error (`NotImplementedError` can be used).
- The `Circle` class derives from the `Shape` class and should consist of the following components:
  - **Properties**
    - `radius`: The circle's radius.

    Getters and setters should be defined for the properties.
  - **Methods**
    - It should be possible to initialize an instance by providing a radius, a color, and whether the circle is filled or not.
    - Friendly string representation for a circle should be implemented.
    - `calculate_area()`: It should return the area of a circle.
    - `calculate_perimeter()`: It should return the perimeter of a circle.
- The `Rectangle` class derives from the `Shape` class and should consist of the following components:
  - **Properties**
    - `width`: The rectangle's width.
    - `length`: The rectangle's length.

    Getters and setters should be defined for the properties.
  - **Methods**
    - It should be possible to initialize an instance by providing a width, length, a color, and whether the circle is filled or not.
    - Friendly string representation for a rectangle should be implemented.
    - `calculate_area()`: It should return the area of a rectangle.
    - `calculate_perimeter()`: It should return the perimeter of a rectangle.

Example

```
class Shape:
    pass

class Circle(Shape):
    pass

class Rectangle(Shape):
    pass

shape = Shape("red", True)
print(shape) # Shape(color="red", is_filled=True)

shape.calculate_area() # raises an error
shape.calculate_perimeter() # raises an error

circle = Circle("black", False, 3)
print(circle) # Circle(color="black", is_filled=False, radius=3)
print(circle.calculate_area()) # 28.27
print(circle.calculate_perimeter()) # 18.85

rectangle = Rectangle("green", True, 3, 4)
print(rectangle) # Rectangle(color="green", is_filled=True, width=3, length=4)
print(rectangle.calculate_area()) # 12
print(rectangle.calculate_perimeter()) # 14
```

Problem 3 [15 points]

Write `Point` and `Triangle` classes in Python.

- The `Point` class should consist of the following components:
  - **Properties**
    - `x`: The x-coordinate of a point.
    - `y`: The y-coordinate of a point.

    Getters and setters should be defined for the properties.
  - **Methods**
    - It should be possible to initialize an instance by providing `x` and `y` coordinates.
    - Friendly string representation for a point should be implemented.
    - `get_xy()`: It should return a tuple of `x` and `y` coordinates.
    - `set_xy(x, y)`: It should change the `x` and `y` coordinates.
    - `distance_from_coordinates(x, y)`: It should return the Euclidean distance between the current point and the point at `(x, y)`.
    - `distance_from_point(another_point)`: It should return the Euclidean distance between the current point and the other point.
    - `abs()`: Point's absolute value should return the Euclidean distance of the point from the origin.
- The `Triangle` class should consist of the following components:
  - **Properties**
    - `vertex1`: The first vertex of a triangle modelled by `Point`.
    - `vertex2`: The second vertex of a triangle modelled by `Point`.
    - `vertex3`: The third vertex of a triangle modelled by `Point`.

    Getters and setters are not needed for the properties.
  - **Methods**
    - It should be possible to initialize an instance by providing `x` and `y` coordinates for all three vertices. Optionally, a feature to initialize an instance by three `Point` objects can be added.
    - Friendly string representation for a triangle should be implemented.
    - `calculate_perimeter()`: It should return the perimeter of a triangle.
    - `get_type()`: It should return the type of a triangle (equilateral, isosceles or scalene).

Example

```
class Point:
    pass

class Triangle:
    pass


point1 = Point(1, 2)
point2 = Point(1, 2)

print(point2) # Point(x=1, y=2)
print(point2.get_xy()) # (1, 2)

point2.set_xy(3, 4)
print(point2) # Point(x=3, y=4)

print(point1.distance_from_coordinates(3, 4)) # 2.83
print(point1.distance_from_point(point2)) # 2.83

print(abs(point1)) # 2.24
print(abs(point2)) # 5

triangle = Triangle(0, 0, 1, 1, 2, 2) # raises an error (No such triangle exists)

triangle = Triangle(0, 0, 0, 4, 2, 0)
print(triangle) # Triangle(vertex1=Point(0, 0), vertex2=Point(0, 4), vertex1=Point(2, 0))

print(triangle.calculate_perimeter()) # 10.47
print(triangle.get_type()) # scalene
```

Problem 4 [20 points]

Write a `Complex` class in Python. The class should consist of the following components:

- **Properties**
  - `real`: The real part of a complex number.
  - `imaginary`: The imaginary part of a complex number.

  Getters and setters are not required.

- **Methods**
  - It should be possible to initialize an instance by providing real and imaginary parts of a complex number.
  - Friendly string representation for a complex number should be implemented.
  - `+`: Addition of two complex numbers should be implemented. Also, it should be possible to add a scalar number to a complex number.
  - `-`: Subtraction of two complex numbers should be implemented. Also, it should be possible to subtract a scalar number from a complex number.
  - `*`: Multiplication two complex numbers should be implemented. Also, it should be possible to multiply a complex number by a scalar number.
  - `**`: Exponentiation of a complex number to an integer power should be implemented.
  - `/`: Division of two complex numbers should be implemented.
  - `==`: Equality checks if two complex numbers are equal.
  - `abs()`: Absolute value of a complex number should return the magnitude of a complex number.

Example

```
class Complex:
    pass


c1 = Complex(1, 2)
c2 = Complex(3, 4)
c3 = Complex(1, 2)

print(c1) # Complex(real=1, imaginary=2)
print(c1 + c2) # Complex(real=4, imaginary=6)
print(c1 + 5) # Complex(real=6, imaginary=2)
print(5 + c1) # Complex(real=6, imaginary=2)
print(c1 - c2) # Complex(real=-2, imaginary=-2)
print(c1 - 5) # Complex(real=-4, imaginary=2)
print(5 - c1) # Complex(real=4, imaginary=-2)
print(c1 * c2) # Complex(real=-5, imaginary=10)
print(c1 * 5) # Complex(real=5, imaginary=10)
print(5 * c1) # Complex(real=5, imaginary=10)
print(c1 / c2) # Complex(real=0.44, imaginary=0.08)
print(c1 ** 2) # Complex(real=-3, imaginary=4)
print(c1 == c2) # False
print(c1 == c3) # True
print(abs(c1)) # 2.2361
```

Problem 5 [25 points]

Write a `WordList` class in Python, which stores a list of words. The class should consist of the following components:

- **Properties**
    - `words`: A list containing words as strings.

  Getters and setters are not required.

- **Methods**
    - The class should allow initialization by providing a list of words.
    - A friendly string representation should be implemented that displays the words in a readable format.
    - `+`: Concatenation of two `WordList` objects should be implemented.
    - `+=`: In-place concatenation should be supported, where words from another `WordList` are appended to the current instance.
    - `*`: Overload the `*` operator for repetition.
    - `len()`: Overload `len()` to return the number of words in the `WordList`.
    - `in`: Overload the `in` operator to check if a word is present in the `WordList`.
    - `[]`: Overload indexing (`[]`) to access words by their index (0-based) and slicing.
        - Implement both getting and setting of words for indexing.
        - Return a new `WordList` object containing the sliced words for slicing.
    - `del`: Overload `del` to allow deletion of a word at a specific index.
    - `reversed()`: Overload the `reversed()` built-in function that yields words in the reverse order.
    - `sorted()`: Overload the necessary method to allow sorting of `WordList` objects lexicographically.
    - `iter()`: Make `WordList` iterable. It should be possible to iterate over the words of the list.

Example

```python
class WordList:
    pass

list1 = WordList(["hello", "world"])
list2 = WordList(["python", "programming"])

print(list1) # WordList(["hello", "world"])
print(list1 + list2) # WordList(["hello", "world", "python", "programming"])

list1 += list2
```

```
print(list1) # WordList(["hello", "world", "python", "programming"])

print(len(list1)) # 4
print("hello" in list1) # True
print(list1[1]) # "world"
print(list1[1:3]) # WordList(["world", "python"])

list1[0] = "hi"
del list1[1]

print(list1) # WordList(["hi", "python", "programming"])

for word in list1:
    print(word, end=" ") # hi python programming

for word in reversed(list1):
    print(word, end=" ") # programming python hi

for word_list in sorted([
    WordList(["def", "ghi"]),
    WordList(["abc", "123"])
]):
    print(word_list, end=" ") # WordList(["abc", "123"]) WordList(["def", "ghi"])
```

Problem 6 [8 points]

You want to automatically retry a block of code a certain number of times if it raises a specific exception (or any exception), with a delay between retries.

Create a context manager `Retry(max_retries=3, delay=1, exceptions=(Exception,))` that:

- Takes parameters for maximum retry attempts, delay in seconds between retries, and a tuple of exception types to catch and retry on (default to `Exception` for any exception).

- Enters a `with` block and executes the code inside.

- If an exception of a type in `exceptions` is raised within the block:
  - Catches the exception.

  - If the retry count is not exhausted, waits for `delay` seconds, increments the retry count, and re-executes the code block from the beginning of the `with` block.

  - If retries are exhausted, re-raises the last caught exception.

- If the code block completes successfully without exceptions, the context manager exits normally.

Example

```python
import random

def flaky_operation():
    if random.random() < 0.8:  # Simulate 80% chance of failure
        raise ValueError("Operation failed!")
    print("Operation successful.")

with Retry(max_retries=3, delay=2, exceptions=(ValueError,)):
    flaky_operation()  # Will retry up to 3 times if ValueError occurs

print("After Retry block.")
```

Problem 7 [8 points]

Given a recursive data structure representing a file system where directories contain files or subdirectories. Using structural pattern matching implement a function that lists all the file names at any depth in the directory tree.

```
class File:
    pass

class Directory:
    pass
```

- A `File` has a `name` attribute, representing the file's name.
- A `Directory` has a `name` and `contents`, which can be a mix of `File` objects and other `Directory` objects.

Example

```
def get_file_names(directory: Directory) -> list[str]:
    pass

root = Directory("root", [
    File("file1.txt"),
    Directory("subdir1", [
        File("file2.txt"),
        Directory("subdir2", [
            File("file3.txt")
        ])
    ]),
    File("file4.txt")
])

print(get_file_names(root)) # ['file1.txt', 'file2.txt', 'file3.txt', 'file4.txt']
```