Homework No. 03

Due: 23:59, 15 April, 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, `homework03_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 03
## 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.

Incorrect or incomplete type hinting **will** result in penalties if trivial type hints are used incorrectly (for example, `list` is used instead of `float`, where the variable is obviously a number).

Problem 1 [25 points]

Create an abstract base class `Expression` that represents a mathematical expression. This class should require methods for evaluating the expression (given a dictionary of variable values), for computing its symbolic derivative with respect to a variable. Also, it should require pretty printing and formatting for expressions. Then, implement concrete classes for constants, variables and binary operations (addition, subtraction, multiplication and division).

Example and testing

```
from abc import ABC

class Expression(ABC):
    pass


class Constant(Expression):
    pass


class Variable(Expression):
    pass


class Add(Expression):
    pass


class Subtract(Expression):
    pass


class Multiply(Expression):
    pass
```

```python
class Divide(Expression):
    pass

expr = Divide(
    Multiply(
        Add(Multiply(Constant(2), Variable("x")), Constant(3)),
        Subtract(Variable("x"), Constant(5))
    ),
    Subtract(Constant(1), Variable("x"))
)

print(expr) # (2 * x + 3) * (x - 5) / (1 - x)

assert expr.evaluate({"x": 3}) == 9.0

derivative_expr = expr.derivative("x")

assert derivative_expr.evaluate({"x": 3}) == -7.0
```

Problem 2 [25 points]

A Binary Search Tree (BST) is a data structure that consists of nodes, where each node contains a value and references to two child nodes: left and right. The key property of a BST is that for every node:

- The value of the left child node is less than the value of the parent node.
- The value of the right child node is greater than the value of the parent node.

Implement a generic BST that supports any comparable type (integers, floats, strings, custom objects). It should provide standard `insert` and `search` operations as defined below:

- **Insertion:** Insert a new node while maintaining the BST property.
- **Search:** Find a node in the tree based on its value.

Example and testing

```python
class BSTNode(...):
    pass


class BST(...):
    pass


"""
```

```
        10
       /  \
      5    15
     / \   /  \
    3   7 12   20
"""
tree = BST[int]()
tree.insert(10)
tree.insert(5)
tree.insert(15)
tree.insert(3)
tree.insert(7)
tree.insert(12)
tree.insert(20)

assert tree.search(7) == True
assert tree.search(15) == True
assert tree.search(42) == False
```

Problem 3 [25 points]

a. Implement a descriptor that tracks the history of all values assigned to an attribute.

Example and testing

```
class HistoryTracking:
    pass


class StockPrice:
    price = HistoryTracking()

    ...

stock = StockPrice("AAPL", 200)

assert stock.price == 200

stock.price = 210

assert stock.price == 210

stock.price = 225

assert stock.price == 225
```

```
assert stock.get_price_history() == [200, 210]

stock.price = 215

assert stock.price == 215
assert stock.get_price_history() == [200, 210, 225]
```

b. Implement a descriptor that logs the access to an attribute, along with the time and value, into a log file.

Example and testing

```
class FileLoggingDescriptor:
    ...


class Person:
    name = FileLoggingDescriptor(log_file="access_log.txt")


obj = Person("John Doe")

print(obj.name) # "John Doe"

obj.name = "Jane Dane"

print(obj.name) # "Jane Dane"

with open("access_log.txt", "r") as f:
    print(f.read())

"""
Sat Mar 29 20:23:56 2025: Modified value from 'no value' to 'John Doe'
Sat Mar 29 20:23:56 2025: Accessed value 'John Doe'
Sat Mar 29 20:23:56 2025: Modified value from 'John Doe' to 'Jane Dane'
Sat Mar 29 20:23:56 2025: Accessed value 'Jane Dane'
"""
```

Problem 4 [25 points]

Write a function `build_logging_class` that dynamically generates a new class which wraps all public methods of a given base class with logging behavior.

The logging should:

- Print a log message before the method is called, showing its name and arguments.
- Print a log message after the method is called, showing the result.

- Exclude dunder methods (`__init__`, `__str__`, etc.).
- Allow injection of a custom logging function (defaults to print).

(Optional) Bonus requirements

- Support a logging level prefix like [ERROR], [INFO], [DEBUG].
- Allow logging to be written to a file.
- Support toggling logging on/off.

Example and testing

```
from typing import Callable

def build_logging_class(name: str, base_class: type, log_func: Callable = print) -> type:
    ...


class Calculator:
    def add(self, x, y): return x + y
    def mul(self, x, y): return x * y

LoggingCalculator = build_logging_class("LoggingCalculator", Calculator)

calc = LoggingCalculator()
calc.add(2, 3)
calc.mul(4, 5)

"""
Expected output

[LOG] Calling: `add` with args=(2, 3) kwargs={}
[LOG] Result of `add`: 5
[LOG] Calling: `mul` with args=(4, 5) kwargs={}
[LOG] Result of `mul`: 20
"""
```