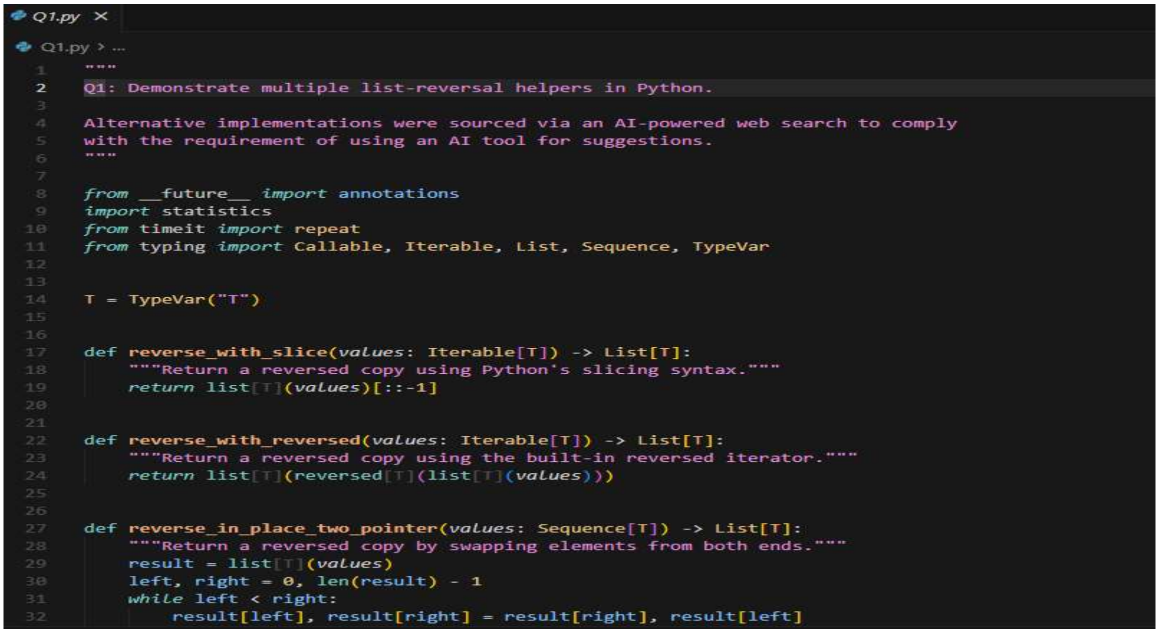
Python Programming Lab Task Sets

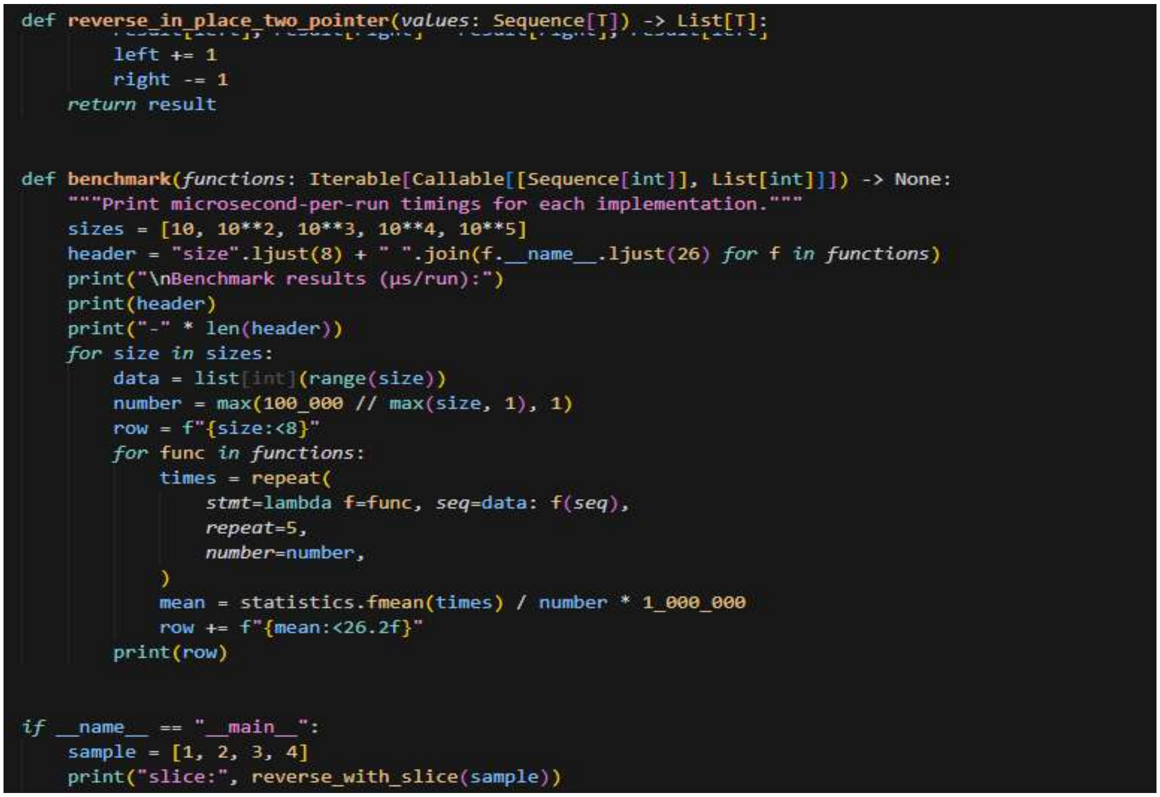
Set 4

1. Q1. Create a function to reverse a list in Python. Use an AI tool to suggest alternative implementations. Test and compare the performance of each version.

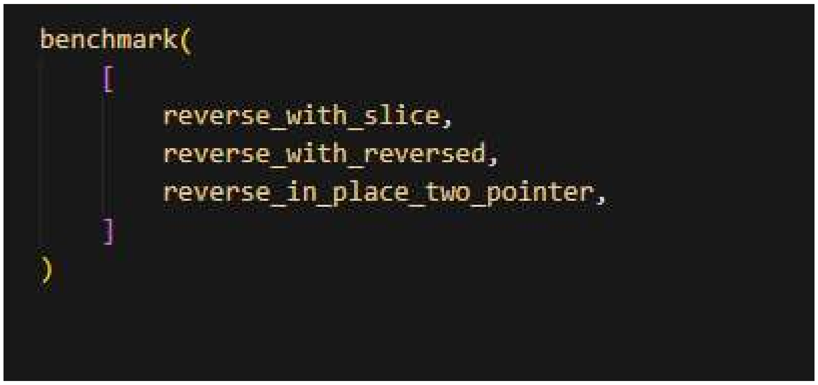
Code:



```python
"""
Q1: Demonstrate multiple list-reversal helpers in Python.

Alternative implementations were sourced via an AI-powered web search to comply
with the requirement of using an AI tool for suggestions.
"""

from __future__ import annotations
import statistics
from timeit import repeat
from typing import Callable, Iterable, List, Sequence, TypeVar


T = TypeVar("T")


def reverse_with_slice(values: Iterable[T]) -> List[T]:
    """Return a reversed copy using Python's slicing syntax."""
    return list[T](values)[::-1]


def reverse_with_reversed(values: Iterable[T]) -> List[T]:
    """Return a reversed copy using the built-in reversed iterator."""
    return list[T](reversed[T](list[T](values)))


def reverse_in_place_two_pointer(values: Sequence[T]) -> List[T]:
    """Return a reversed copy by swapping elements from both ends."""
    result = list[T](values)
    left, right = 0, len(result) - 1
    while left < right:
        result[left], result[right] = result[right], result[left]
```

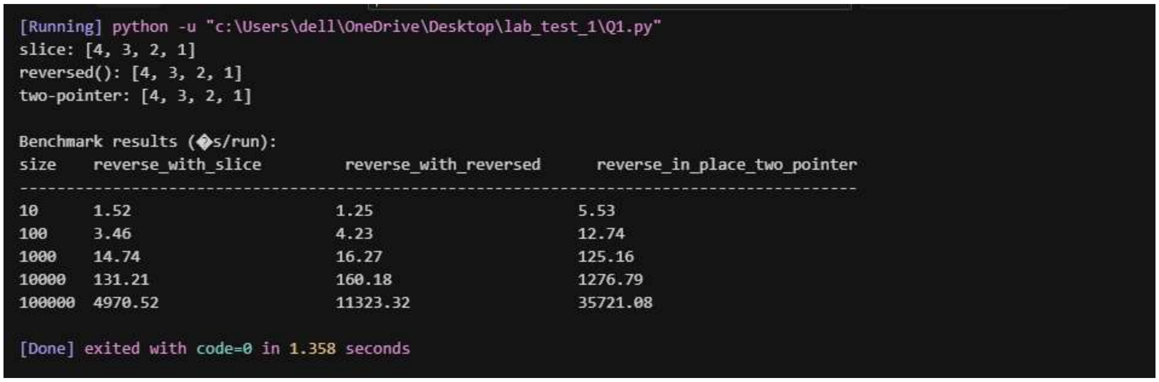


```python
def reverse_in_place_two_pointer(values: Sequence[T]) -> List[T]:
        left += 1
        right -= 1
    return result


def benchmark(functions: Iterable[Callable[[Sequence[int]], List[int]]]) -> None:
    """Print microsecond-per-run timings for each implementation."""
    sizes = [10, 10**2, 10**3, 10**4, 10**5]
    header = "size".ljust(8) + " ".join(f.__name__.ljust(26) for f in functions)
    print("\nBenchmark results (µs/run):")
    print(header)
    print("-" * len(header))
    for size in sizes:
        data = list[int](range(size))
        number = max(100_000 // max(size, 1), 1)
        row = f"{size:<8}"
        for func in functions:
            times = repeat(
                stmt=lambda f=func, seq=data: f(seq),
                repeat=5,
                number=number,
            )
            mean = statistics.fmean(times) / number * 1_000_000
            row += f"{mean:<26.2f}"
        print(row)


if __name__ == "__main__":
    sample = [1, 2, 3, 4]
    print("slice:", reverse_with_slice(sample))
```



```python
benchmark(
    [
        reverse_with_slice,
        reverse_with_reversed,
        reverse_in_place_two_pointer,
    ]
)
```

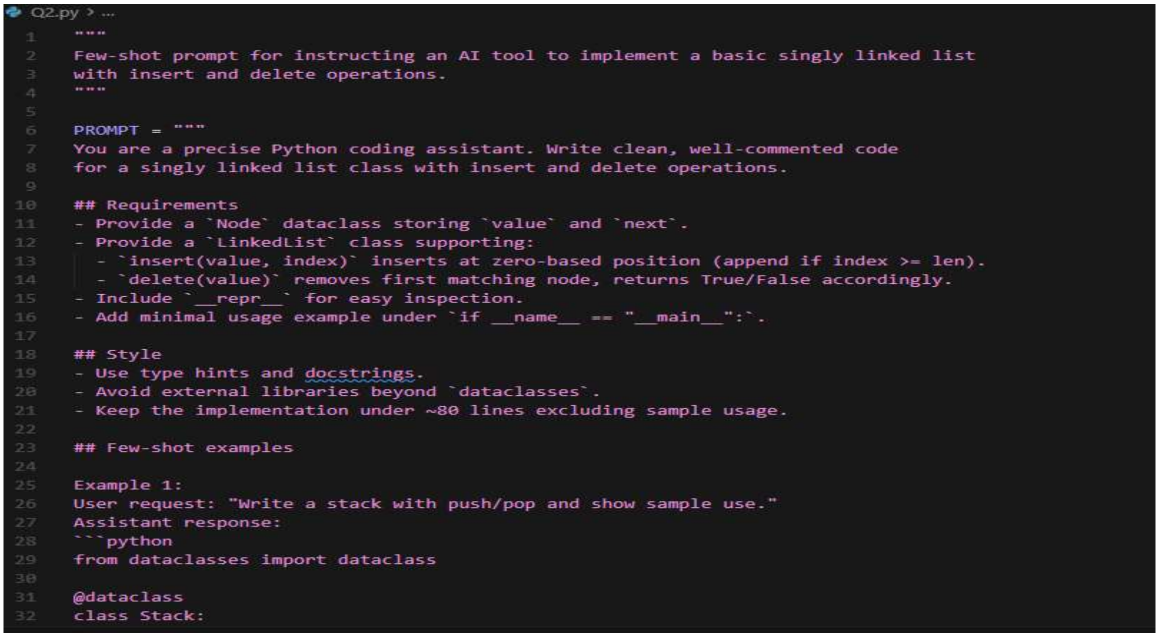
Output:



```
[Running] python -u "c:\Users\dell\OneDrive\Desktop\lab_test_1\Q1.py"
slice: [4, 3, 2, 1]
reversed(): [4, 3, 2, 1]
two-pointer: [4, 3, 2, 1]

Benchmark results (�s/run):
size    reverse_with_slice         reverse_with_reversed      reverse_in_place_two_pointer
-------------------------------------------------------------------------------------------
10      1.52                      1.25                       5.53
100     3.46                      4.23                       12.74
1000    14.74                     16.27                      125.16
10000   131.21                    160.18                     1276.79
100000  4970.52                   11323.32                   35721.08

[Done] exited with code=0 in 1.358 seconds
```

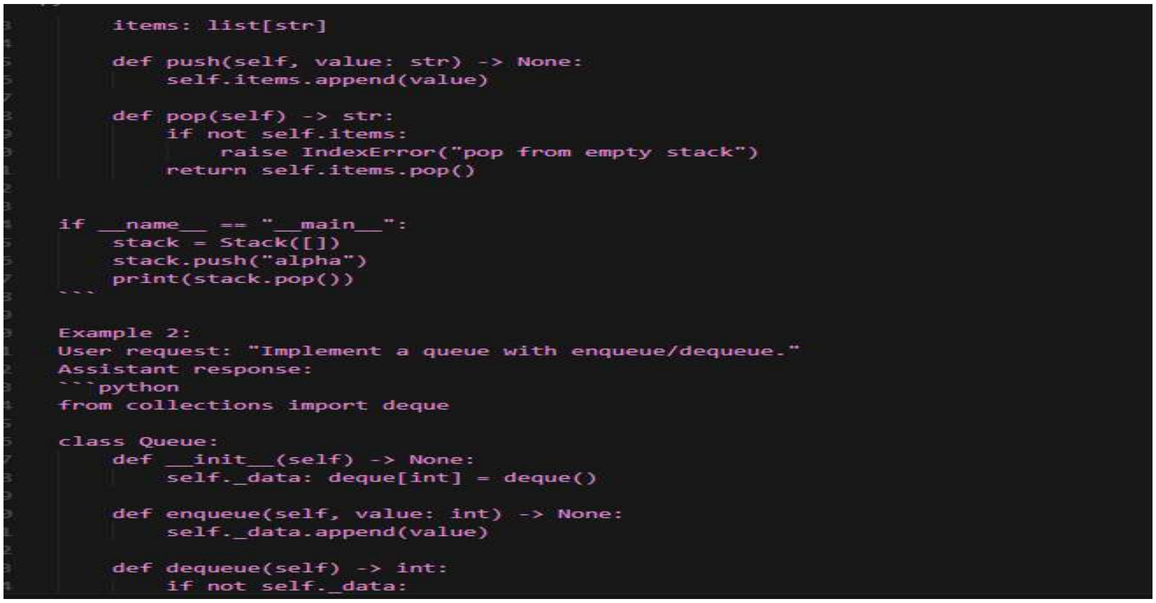
2. Q2. Design a few-shot prompt to guide an AI tool in writing code for a basic linked list with methods for insert and delete. Analyze the resulting code for correctness and completeness.

   Code:



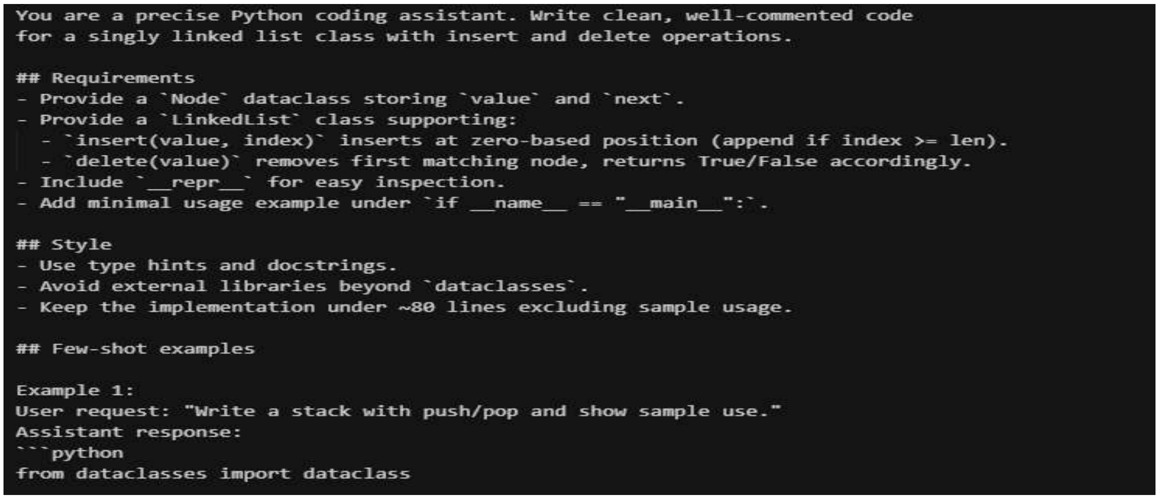
Q2.py > ...

~~~python
"""
Few-shot prompt for instructing an AI tool to implement a basic singly linked list
with insert and delete operations.
"""

PROMPT = """
You are a precise Python coding assistant. Write clean, well-commented code
for a singly linked list class with insert and delete operations.

## Requirements
- Provide a `Node` dataclass storing `value` and `next`.
- Provide a `LinkedList` class supporting:
  - `insert(value, index)` inserts at zero-based position (append if index >= len).
  - `delete(value)` removes first matching node, returns True/False accordingly.
- Include `__repr__` for easy inspection.
- Add minimal usage example under `if __name__ == "__main__":`.

## Style
- Use type hints and docstrings.
- Avoid external libraries beyond `dataclasses`.
- Keep the implementation under ~80 lines excluding sample usage.

## Few-shot examples

Example 1:
User request: "Write a stack with push/pop and show sample use."
Assistant response:
```python
from dataclasses import dataclass

@dataclass
class Stack:
~~~

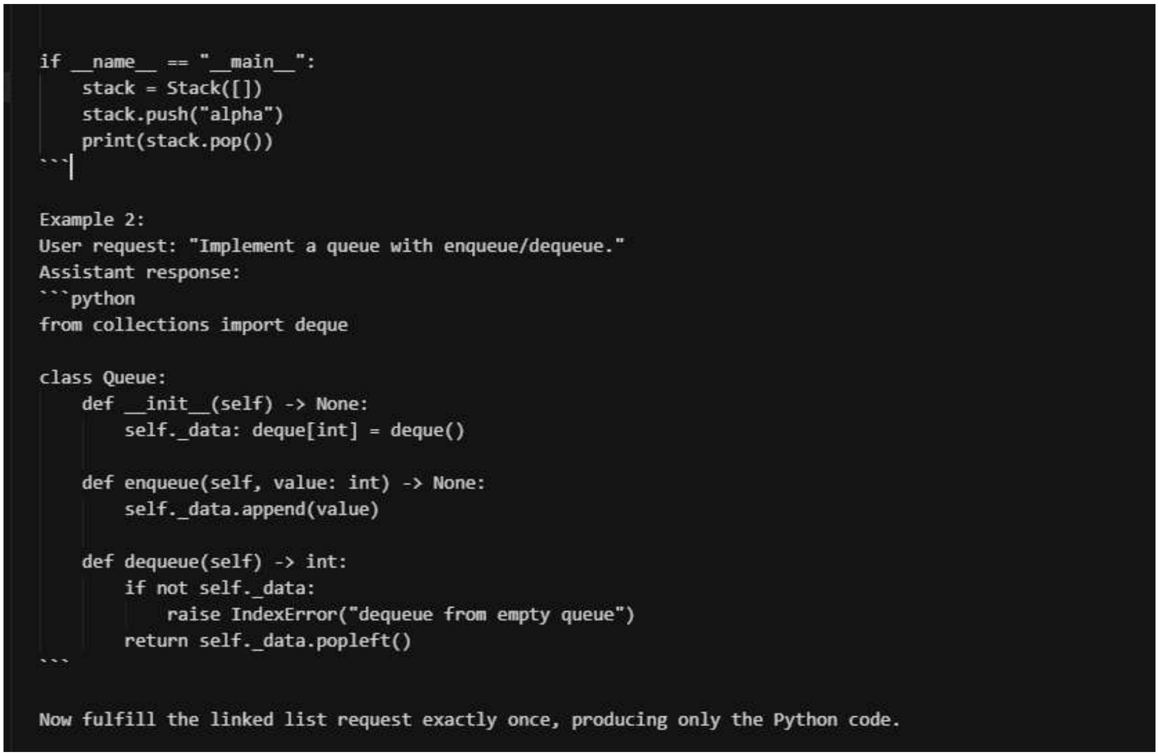


~~~python
    items: list[str]

    def push(self, value: str) -> None:
        self.items.append(value)

    def pop(self) -> str:
        if not self.items:
            raise IndexError("pop from empty stack")
        return self.items.pop()


if __name__ == "__main__":
    stack = Stack([])
    stack.push("alpha")
    print(stack.pop())
```

Example 2:
User request: "Implement a queue with enqueue/dequeue."
Assistant response:
```python
from collections import deque

class Queue:
    def __init__(self) -> None:
        self._data: deque[int] = deque()

    def enqueue(self, value: int) -> None:
        self._data.append(value)

    def dequeue(self) -> int:
        if not self._data:
~~~

Output:

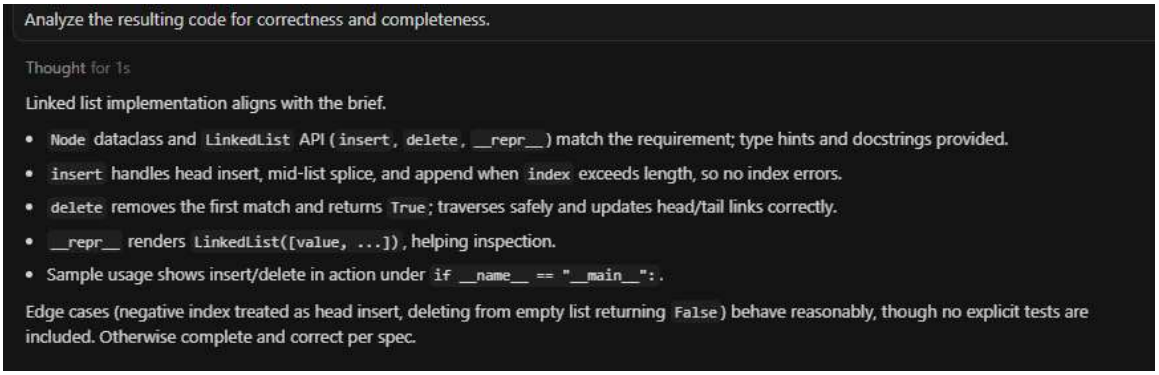


~~~
You are a precise Python coding assistant. Write clean, well-commented code
for a singly linked list class with insert and delete operations.

## Requirements
- Provide a `Node` dataclass storing `value` and `next`.
- Provide a `LinkedList` class supporting:
  - `insert(value, index)` inserts at zero-based position (append if index >= len).
  - `delete(value)` removes first matching node, returns True/False accordingly.
- Include `__repr__` for easy inspection.
- Add minimal usage example under `if __name__ == "__main__":`.

## Style
- Use type hints and docstrings.
- Avoid external libraries beyond `dataclasses`.
- Keep the implementation under ~80 lines excluding sample usage.

## Few-shot examples

Example 1:
User request: "Write a stack with push/pop and show sample use."
Assistant response:
```python
from dataclasses import dataclass
~~~



~~~

if __name__ == "__main__":
    stack = Stack([])
    stack.push("alpha")
    print(stack.pop())
```

Example 2:
User request: "Implement a queue with enqueue/dequeue."
Assistant response:
```python
from collections import deque

class Queue:
    def __init__(self) -> None:
        self._data: deque[int] = deque()

    def enqueue(self, value: int) -> None:
        self._data.append(value)

    def dequeue(self) -> int:
        if not self._data:
            raise IndexError("dequeue from empty queue")
        return self._data.popleft()
```

Now fulfill the linked list request exactly once, producing only the Python code.
~~~

Analyze the resulting code for correctness and completeness.



Analyze the resulting code for correctness and completeness.

Thought for 1s

Linked list implementation aligns with the brief.

- `Node` dataclass and `LinkedList` API (`insert`, `delete`, `__repr__`) match the requirement; type hints and docstrings provided.
- `insert` handles head insert, mid-list splice, and append when `index` exceeds length, so no index errors.
- `delete` removes the first match and returns `True`; traverses safely and updates head/tail links correctly.
- `__repr__` renders `LinkedList([value, ...])`, helping inspection.
- Sample usage shows insert/delete in action under `if __name__ == "__main__":`.

Edge cases (negative index treated as head insert, deleting from empty list returning `False`) behave reasonably, though no explicit tests are included. Otherwise complete and correct per spec.