Day 11: Functional Programming Mechanics (Pseudo Code)

Hint 11.1: The Anonymous Function

Task: `lambda x: x + 1`.
Pseudo Code Logic:

```
# Standard Definition (Heap allocation + Name binding)
FUNCTION named_add(a, b):
    RETURN a + b

# Lambda Definition (Heap allocation only)
lambda_obj = CONSTRUCT_FUNCTION(arguments=[a,b], code=[return
    a+b])
```

Detailed Explanation: A `lambda` is a function object created "on the fly." Unlike a standard `def`, it does not assign itself a name in the local namespace. It is an expression, meaning it evaluates to a value (the function object itself) immediately. This allows you to pass logic into other functions just like you pass integers or strings.

Hint 11.2: The Mapper

Task: `map(func, list)`.
Pseudo Code Logic:

```
FUNCTION map_logic(transformation_func, data_list):
    # Returns an Iterator (Lazy)
    FOR item IN data_list:
        new_val = transformation_func(item)
        YIELD new_val
```

Detailed Explanation: `map` applies a transformation function to every single item in an iterable. Crucially, it does not modify the original list (Immutability). It produces a stream of *new* values. In Python, this loop happens in C (compiled code), making it significantly faster than a manual Python `for` loop for simple operations.

Hint 11.3: The Filter

Task: `filter(func, list)`.
Pseudo Code Logic:

```
FUNCTION filter_logic(check_func, data_list):
    FOR item IN data_list:
        is_valid = check_func(item)

        IF is_valid IS True:
            YIELD item
        # Else: Discard item silently
```

Detailed Explanation: `filter` acts as a sieve. The function you pass must return a Boolean (`True/False`). If it returns `True`, the item passes through to the result stream. If `False`, it is dropped. Passing `None` as the function defaults to checking "Truthiness" (dropping 0, Empty Strings, None).

Hint 11.4: The Reducer

Task: `reduce(func, list)`.
Pseudo Code Logic:

```
FUNCTION reduce_logic(func, list):
    accumulator = list[0]  # Start with first item

    FOR i FROM 1 TO length(list):
        current = list[i]
        # Collapse Acc and Curr into new Acc
        accumulator = func(accumulator, current)

    RETURN accumulator
```

Detailed Explanation: Reduce transforms a list of many items into a single scalar value. It does this by progressively "eating" the list from left to right. Step 1: `func(Item1, Item2) -> ResultA` Step 2: `func(ResultA, Item3) -> ResultB` ...until only one value remains.

Hint 11.5: The Custom Sort Key

Task: `sorted(list, key=func)`.
Pseudo Code Logic:

```
FUNCTION sort_with_key(list, key_func):
    # Step 1: Create 'Shadow List' of (Key, Original)
    shadow = []
    FOR item IN list:
        sort_val = key_func(item)
```

```
        shadow.append( (sort_val, item) )

    # Step 2: Sort based on the Key (Index 0)
    shadow.sort()

    # Step 3: Extract original items
    RETURN [item FOR (val, item) IN shadow]
```

Detailed Explanation: This is the "Decorate-Sort-Undecorate" pattern (Schwartzian Transform). Python doesn't sort the actual data strings; it sorts the temporary values generated by your `key` function. This is extremely efficient because the key function runs exactly once per item, not every time a comparison happens.

Hint 11.6: The Zip Lock

Task: `zip(listA, listB)`.
Pseudo Code Logic:

```
FUNCTION zip_logic(iterA, iterB):
    WHILE True:
        TRY:
            valA = next(iterA)
            valB = next(iterB)
            YIELD (valA, valB) # Tuple
        CATCH StopIteration:
            BREAK # Stop when shortest list ends
```

Detailed Explanation: `zip` runs in parallel. It pulls one item from every iterator simultaneously and binds them into a Tuple. It is the fundamental tool for matrix transposition and dictionary construction (`keys` + `values`). Note that it stops as soon as the *shortest* input is exhausted.

Hint 11.7: List Comprehension Speed

Task: `[func(x) for x in data]` vs `map`.
Pseudo Code Logic:

```
# Map + Lambda overhead
FOR item IN list:
    CALL Python_Function_Frame(lambda, item) # Expensive!

# List Comprehension
```

```
FOR item IN list:
    EXECUTE_EXPRESSION(item) # Optimized Bytecode. Fast.
```

Detailed Explanation: While `map` is fast in C, using a `lambda` inside it forces the interpreter to create a generic function stack frame for every single item. List comprehensions have a special dedicated opcode (`LIST_APPEND`) that avoids this function-call overhead, often making them faster for simple math.

Hint 11.8: Any All

Task: Short-circuit Evaluation.
Pseudo Code Logic:

```
FUNCTION any_logic(iterable):
    FOR item IN iterable:
        IF item IS True:
            RETURN True # Stop immediately!
    RETURN False

FUNCTION all_logic(iterable):
    FOR item IN iterable:
        IF item IS False:
            RETURN False # Stop immediately!
    RETURN True
```

Detailed Explanation: These functions optimize boolean checks on lists. Instead of calculating `True/False` for 1 million items and then checking, they run the check lazily. As soon as the result is determined (e.g., one `False` found in `all`), they abort the loop. This changes best-case complexity from $O(N)$ to $O(1)$.

Hint 11.9: Partial Functions

Task: Locking arguments.
Pseudo Code Logic:

```
FUNCTION partial(func, *fixed_args):

    FUNCTION wrapper(*remaining_args):
        # Merge the locked args with new ones
        combined_args = fixed_args + remaining_args
        RETURN func(*combined_args)
```

```
    RETURN wrapper
```

Detailed Explanation: Partial application allows you to take a general function (like `power(base, exp)`) and "specialize" it into a specific function (like `square(x)` where `exp=2`). It creates a new callable that remembers the frozen arguments, useful for callbacks in GUIs or Event Listeners.

Hint 11.10: The Immutability Test

Task: Pure Functions.
Pseudo Code Logic:

```
# BAD (Impure / Side Effect)
FUNCTION impure(list):
    list[0] = 99  # Modifies external memory directly
    RETURN list

# GOOD (Pure)
FUNCTION pure(list):
    new_list = COPY(list) # Create new memory block
    new_list[0] = 99
    RETURN new_list       # Original list is safe
```

Detailed Explanation: Functional programming relies on the guarantee that data does not change under your feet. If you pass a list to a function, you must trust that the function won't delete items from it. Pure functions always return *new* data rather than mutating inputs, which eliminates "Race Conditions" in parallel processing.