Advanced Python Functional Programming Workbook

Created using Gemini. By Muhammed Shafin P.

This workbook is designed for experienced Python developers seeking to master advanced functional programming techniques, including closures, higher-order functions, and complex comprehensions.

1️⃣ Lambda Functions

Lambda functions (anonymous functions) are powerful for quick, single-expression operations, especially when used with built-in functions like map, filter, and sorted.

Simple (S)

1. S1: Squaring Lambda
Define a lambda function that takes one argument x and returns its square. Use it to compute the square of 12.

```
square = ???(x: x * x)
result_s1 = square(12)
# Expected: 144
```

2. S2: String Case Check
Define a lambda function that checks if a string starts with an uppercase letter.

```
check_uppercase_start = ???(s: s[0].isupper() if s else False)
result_s2a = check_uppercase_start("Hello")
result_s2b = check_uppercase_start("world")
# Expected: True, False
```

Medium (M)

3. M1: Sorting List of Tuples
Sort the following list of tuples based on the second element of each tuple using a lambda as the key.

```
data_m1 = [('alpha', 3), ('beta', 1), ('gamma', 5), ('delta', 2)]
sorted_m1 = sorted(data_m1, key=???)
# Expected: [('beta', 1), ('delta', 2), ('alpha', 3), ('gamma', 5)]
```

4. M2: Filtering by Value
Use the filter() function and a lambda to keep only the numbers greater than 10 in the list.

```
numbers_m2 = [5, 12, 8, 25, 3, 15]
filtered_m2 = list(filter(???, numbers_m2))
# Expected: [12, 25, 15]
```

Hard (H)

5. H1: Dictionary Key Transformation
Use the map() function and a lambda to transform a list of dictionaries. Each dictionary must be transformed to contain only the key-value pair where the key is 'name' and the value is capitalized.

```
data_h1 = [{'id': 101, 'name': 'alice'}, {'id': 102, 'name': 'bob'}, {'id': 103, 'name': 'charlie'}]
transformed_h1 = list(map(???, data_h1))
# Expected: [{'name': 'ALICE'}, {'name': 'BOB'}, {'name': 'CHARLIE'}]
```

6. H2: Nested Dictionary Sorting
Sort the list of dictionaries by the value of the nested key 'score' inside the 'metrics' dictionary.

```
data_h2 = [
   {'user': 'a', 'metrics': {'score': 95, 'time': 120}},
   {'user': 'b', 'metrics': {'score': 88, 'time': 90}},
   {'user': 'c', 'metrics': {'score': 92, 'time': 150}}
]
sorted_h2 = sorted(data_h2, key=???)
# Expected: [{'user': 'b', 'metrics': {'score': 88, ...}}, {'user': 'c', ...}, {'user': 'a', ...}]
```

2️⃣ Closures and Freezing Variables

Closures occur when a nested function remembers the values from its enclosing scope, even after the enclosing function has finished execution. Understanding how loop variables (n=n trick) are handled is critical.

Simple (S)

7. S1: Basic Multiplier Closure
Create a function make_multiplier(x) that returns a function. The returned function should take one argument y and return x * y.

```
def make_multiplier(x):
    return ???

times_five = make_multiplier(5)
result_s7 = times_five(10)
# Expected: 50
```

8. S2: Static Message Generator
Create a closure greet_person(name) that returns a function which, when called, prints a fixed greeting for that person.

```
def greet_person(name):
    fixed_greeting = "Hello, " + name
    return lambda: ???

greet_bob = greet_person("Bob")
# Call should print "Hello, Bob"
```

Medium (M)

9. M3: The Late-Binding Trap
The following code demonstrates the classic late-binding issue. What will f() return for each function in the list, and why?

```
def create_functions():
    return [lambda: i for i in range(3)]

functions_m9 = create_functions()
results_m9 = [f() for f in functions_m9]
# Explain the result.
```

10. M4: Fixing Late-Binding with Default Arguments
Fix the create_functions example (M3) by using a default argument in the lambda definition to freeze the value of i at the time of creation.

```
def create_functions_fixed():
    # Use default argument (i=i) to freeze the value
    return [lambda i=i: ??? for i in range(3)]
```

```
functions_m10 = create_functions_fixed()
results_m10 = [f() for f in functions_m10]
# Expected: [0, 1, 2]
```

Hard (H)

11. H1: Closure for Dynamic Suffixing
Create a function suffix_generator(base_word) that returns a function. The returned function accepts a list of suffixes and generates a new list where the base_word is appended to each suffix, but only if the suffix is longer than 3 characters.

```
def suffix_generator(base_word):
    # This must be a closure using a lambda for mapping
    return lambda suffixes: [
        ???
        for suffix in suffixes if len(suffix) > 3
    ]

gen_report_name = suffix_generator("Analysis")
suffixes_h1 = ["Draft", "Final", "Summary", "Metrics", "Prelim"]
result_h1 = gen_report_name(suffixes_h1)
# Expected: ['FinalAnalysis', 'SummaryAnalysis', 'MetricsAnalysis', 'PrelimAnalysis']
```

12. H2: Function Generator for Custom Filters
Write a function filter_factory(min_length) that returns a filter function. The returned function takes a list of strings and returns only those strings whose length is greater than or equal to min_length.

```
def filter_factory(min_length):
    return lambda words: list(filter(???, words))

long_word_filter = filter_factory(7)
words_h2 = ["apple", "banana", "kiwi", "grapefruit", "orange"]
result_h2 = long_word_filter(words_h2)
# Expected: ['grapefruit']
```

3️ Conditional / Ternary Expressions

Ternary operators (conditional expressions) are a concise way to handle simple if-else logic in

a single line, often making comprehensions and lambdas more expressive.

Simple (S)

13. S1: Ternary Parity Check
Use a ternary expression to assign the string "Even" if number_s1 is even, and "Odd" otherwise.
number_s1 = 45
parity_s1 = ???
# Expected: "Odd"

14. S2: Status Assignment
Assign "Active" if is_logged_in_s2 is True, and "Inactive" otherwise.
is_logged_in_s2 = True
status_s2 = ???
# Expected: "Active"

Medium (M)

15. M3: Ternary in a Lambda
Define a lambda function that takes a list of numbers and returns the square of the number if it is positive, and the number itself if it is negative or zero.
transform_m3 = lambda x: ???
result_m3a = transform_m3(4)
result_m3b = transform_m3(-5)
# Expected: 16, -5

16. M4: Nested Ternary (Three States)
Use a single, nested ternary expression to classify a temperature: "Cold" (<= 0), "Warm" (> 0 and <= 25), or "Hot" (> 25).
temp_m4 = 30
classification_m4 = ???
# Expected: "Hot"

Hard (H)

17. H1: Dynamic Function Selection

Use a ternary expression to select either the built-in max function or the built-in min function based on the boolean use_max. Apply the selected function to data_h17.

```
data_h17 = [10, 5, 20, 8]
use_max = False

selected_func = ???
result_h17 = selected_func(data_h17)
# Expected: 5 (since use_max is False, it should select min)
```

18. H2: Combining Ternary, Filter, and Map

Use a single line combining filter, map, and a lambda with a ternary expression to:

1. Filter out numbers that are less than 5.
2. For the remaining numbers, if the number is even, square it; otherwise, keep it as is.

```
numbers_h18 = [2, 7, 4, 11, 8, 3]
processed_h18 = list(map(lambda x: ???, filter(lambda x: x >= 5, numbers_h18)))
# Expected: [7, 11, 64]
```

4️⃣ List Comprehensions

Mastering list comprehensions is vital for concise, readable Python. We cover single, nested, and filtered forms.

Simple (S)

19. S1: String Lengths

Create a list comprehension that calculates the length of each string in the provided list.

```
words_s19 = ["functional", "python", "closure", "lambda"]
lengths_s19 = [??? for word in words_s19]
# Expected: [10, 6, 7, 6]
```

20. S2: Filtering Vowels

Create a list comprehension that filters out all vowels (a, e, i, o, u, case-insensitive) from the string text_s20.

```
text_s20 = "Advanced Programming Exercise"
consonants_s20 = [char for char in text_s20 if ???]
# Expected: ['A', 'd', 'v', 'n', 'c', 'd', ' ', 'P', 'r', 'g', 'r', 'm', 'm', 'n', 'g', ' ', 'E', 'x', 'r', 'c', 's'] (Case kept, only vowels filtered)
```

Medium (M)

21. M3: Filter and Map
Create a list comprehension that only includes numbers greater than 10, and then cubes those numbers.

```
numbers_m21 = [2, 15, 7, 20, 5, 12]
cubes_m21 = [??? for num in numbers_m21 if ???]
# Expected: [3375, 8000, 1728]
```

22. M4: Flattening a 2D List
Use a nested list comprehension to flatten the matrix_m22 into a single, one-dimensional list.

```
matrix_m22 = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
flat_m22 = [??? for row in matrix_m22 for item in row]
# Expected: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Hard (H)

23. H1: Nested Comprehension with Filter (Coordinated Iteration)
Generate a list of tuples (i, j) where i is from range(5) and j is from range(5), but only include the pair if their sum i + j is a prime number (2, 3, 5, 7).

```
primes = {2, 3, 5, 7}
prime_pairs_h23 = [
    (i, j) for i in range(5) for j in range(5) if ???
]
# Expected length: 14 pairs. Example: (0, 2), (1, 2), (2, 3), etc.
```

24. H2: Dictionary Comprehension with Ternary
Create a dictionary comprehension from the list of words. The key should be the word itself. The value should be 'Long' if the word length is greater than 6, and 'Short' otherwise.

```
words_h24 = ["apple", "banana", "kiwi", "grapefruit", "date"]
classification_h24 = {??? for word in words_h24}
# Expected: {'apple': 'Short', 'banana': 'Long', 'kiwi': 'Short', 'grapefruit': 'Long', 'date': 'Short'}
```

5️⃣ Higher-Order Functions (HOF)

HOFs are functions that either take other functions as arguments or return a function as a result.

Simple (S)

25. S1: apply_to_list
Write a HOF apply_to_list(func, data) that takes a function and a list, and returns a new list with the function applied to every element. (This is a simplified map).

```
def apply_to_list(func, data):
    return [???]

def add_one(x):
    return x + 1

result_s25 = apply_to_list(add_one, [1, 2, 3])
# Expected: [2, 3, 4]
```

26. S2: Basic Function Composition
Write a function compose_two(f, g) that takes two functions, f and g, and returns a new function h(x) such that h(x) = f(g(x)).

```
def compose_two(f, g):
    return lambda x: ???

def double(x): return x * 2
def add_three(x): return x + 3

h = compose_two(double, add_three) # f(g(x)) = double(x + 3)
result_s26 = h(5)
# Expected: 16 (double(8))
```

Medium (M)

27. M3: make_power
Create a HOF make_power(p) that takes an exponent p and returns a function that raises its input to the power of p.

```
def make_power(p):
    return lambda x: ???
```

```
square = make_power(2)
cube = make_power(3)

result_m27a = square(4)
result_m27b = cube(2)
# Expected: 16, 8
```

28. M4: Generalized apply_twice

Write a HOF apply_twice(func) that takes one function func and returns a new function that applies func to its input two times.

```
def apply_twice(func):
    return lambda x: ???

def inc(x): return x + 1

double_inc = apply_twice(inc)
result_m28 = double_inc(5)
# Expected: 7 (5 -> 6 -> 7)
```

Hard (H)

29. H1: Functional reduce Implementation

Implement your own simplified version of functools.reduce called my_reduce(func, iterable, initializer=None). The func is a two-argument function that must be applied cumulatively to the items of the iterable.

```
def my_reduce(func, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        value = next(it)
    else:
        value = initializer
    for element in it:
        value = ???
    return value

result_h29 = my_reduce(lambda x, y: x + y, [10, 20, 30], 100)
# Expected: 160
```

30. H2: Generalized compose
Write a HOF compose(*functions) that accepts an arbitrary number of functions and returns a new function that represents their left-to-right composition.
$$ (f \circ g \circ h)(x) = f(g(h(x))) $$

```
def compose(*functions):
    # Hint: Use my_reduce (or the actual functools.reduce)
    # The accumulator should be the composed function so far.
    # The starting function is the identity function (lambda x: x)
    def inner(x):
        # Apply functions right to left
        result = x
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

# Example functions
def f(x): return x * 10 # 1st to execute
def g(x): return x + 1  # 2nd to execute
def h(x): return x - 2  # 3rd to execute

composed_func = compose(f, g, h) # f(g(h(x)))
result_h30 = composed_func(5)
# Expected: 40 (h(5)=3, g(3)=4, f(4)=40)
```

6 Mixed Advanced Exercises

These exercises require combining multiple functional concepts.

31. M1: Closure-Filtered Mapping
Create a closure transform_if_short(max_len) that returns a function. The returned function accepts a list of strings and uses a list comprehension to return the string's length if it is shorter than max_len, and the string itself otherwise.

```
def transform_if_short(max_len):
    return lambda words: [
        ???
        for word in words
    ]

transformer_m31 = transform_if_short(5)
```

```
result_m31 = transformer_m31(["cat", "dog", "elephant", "kite"])
# Expected: [3, 3, 'elephant', 4]
```

32. M2: Nested Comprehension with HOF and Lambda

Use a list comprehension to create a list of tuples (i, j) where i ranges from 0 to 2. The value of j should be the result of applying a lambda to i (e.g., squaring i) only if the result is even. Use a filter within the list comprehension.

```
square_if_even_m32 = [
    (i, j) for i in range(5)
    for j in [i * i] if ???
]
# Expected: [(0, 0), (2, 4), (4, 16)]
```

33. H1: Deep Filtering using Ternary and HOF

Given a list of records, use filter and a lambda with a nested ternary to select records based on the following logic:

- If 'status' is 'ERROR', keep the record.
- Otherwise, if 'priority' is 'HIGH' and 'time' is less than 50, keep the record.
- Otherwise, discard the record.

```
records_h33 = [
    {'status': 'OK', 'priority': 'LOW', 'time': 20},
    {'status': 'OK', 'priority': 'HIGH', 'time': 60},
    {'status': 'ERROR', 'priority': 'LOW', 'time': 100},
    {'status': 'OK', 'priority': 'HIGH', 'time': 40},
]

filtered_h33 = list(filter(lambda r: ???, records_h33))
# Expected: 2 records (Index 2 and 3)
```

34. H2: Late-Binding Fix in List Comprehension

Correct the late-binding issue directly within a list comprehension. Create a list of functions where the i-th function returns $i \times 10$.

```
# The functions should return 0, 10, 20, 30, 40 when called.
functions_h34 = [
    lambda i=i: ???
    for i in range(5)
]
```

```
results_h34 = [f() for f in functions_h34]
# Expected: [0, 10, 20, 30, 40]
```

35. H3: Closure for Conditional List Processing

Write a function conditional_processor(threshold) that returns a function. The returned function takes a list of numbers and uses a list comprehension to double the number if it is greater than the threshold (captured by the closure), and halves it otherwise.

```
def conditional_processor(threshold):
    # Use closure variable 'threshold' in the list comprehension with a ternary
    return lambda nums: [
        ???
        for n in nums
    ]

processor = conditional_processor(10)
result_h35 = processor([5, 12, 8, 20, 1])
# Expected: [2.5, 24, 4.0, 40, 0.5]
```

7️ Paragraphic/Scenario-based Hard Exercises

These scenarios require you to synthesize the functional programming requirements from the description and implement the solution without starter code.

36. P1: Inventory Price Calculator

You have an inventory list of parts: data = [{'name': 'A', 'stock': 10, 'price': 50.0}, {'name': 'B', 'stock': 0, 'price': 10.0}, {'name': 'C', 'stock': 5, 'price': 100.0}]. Use a list comprehension to calculate the total realizable value for each part. If the stock is 0, the value should be 0. Otherwise, the value is stock * price. The final output should be a list of these total values.

37. P2: Secure Log Filter

Write a higher-order function called create_security_filter(level) that takes a security level (an integer) and returns a filter function. The returned function accepts a list of log entries (dictionaries with keys 'severity' (int) and 'message' (str)). This filter function must use filter and a lambda to keep only log entries whose severity is greater than or equal to the level defined in the closure. Use the factory to create a filter for level 50 and apply it to logs = [{'severity': 20}, {'severity': 60}, {'severity': 40}, {'severity': 80}].

38. P3: Dynamic Suffix Generator with Late-Binding Fix

Create a function that generates a list of 5 functions. Each function must take a single string argument s and return s concatenated with a unique, static number from 1 to 5, ensuring the number is frozen at function creation time (i.e., fix the late-binding issue). The 3rd function in

the list, when called with "Item", should return "Item3".

39. P4: Composed Data Normalizer

Write three simple functions:

1. half(x): Divides x by 2.
2. add_ten(x): Adds 10 to x.
3. stringify(x): Converts x to a string with "Value: " prepended.
   Use your generalized compose HOF (from H2) to create a single function normalize_data that applies these three functions in the order: stringify(add_ten(half(x))). Apply normalize_data to the number 90.

40. P5: Employee Bonus Evaluator

You have a list of departments, each containing a list of employees: dept_data = [{'dept': 'HR', 'employees': [100000, 50000]}, {'dept': 'ENG', 'employees': [120000, 150000, 80000]}]. Use a nested list comprehension combined with a ternary expression to create a flat list of calculated bonuses. The bonus is 10% of the salary if the salary is over 100,000, and 5% otherwise.

41. P6: Custom Sort Utility

Define a HOF create_key_sorter(key_name) that takes a string key_name and returns a function suitable for use as the key argument in sorted(). The returned function should access the specified key in a dictionary. Use this factory to sort the following list of dictionaries by the 'age' key: people = [{'name': 'A', 'age': 30}, {'name': 'B', 'age': 25}, {'name': 'C', 'age': 35}].

42. P7: Conditional String Formatter

Use the map() function and a lambda with a ternary expression to process a list of names: names = ["alice", "bob", "charlie", "david"]. If a name starts with a vowel (A, E, I, O, U, case-insensitive), convert it to all uppercase; otherwise, title-case it (e.g., 'Bob').

43. P8: Recursive Closure for Sequences

Write a function fibonacci_generator() that returns a closure. The closure, when called repeatedly without arguments, should yield the next number in the Fibonacci sequence (starting 0, 1, 1, 2, 3, 5, ...). The closure must maintain the sequence state internally.

44. P9: Flatten and Filter Multi-Level Data

You have a list of data groups: groups = [[(1, 10), (2, 5)], [(3, 12), (4, 4)]]. Use a single, double-nested list comprehension to extract the second element of every tuple, but only if that second element (the value) is greater than 8. The final result must be a flat list of these values.

45. P10: Pipeline Function Creator

Write a function pipeline(*fns) that is similar to compose, but executes the functions in left-to-right order. So, pipeline(f, g, h) should return a function that calculates $h(g(f(x)))$. Apply pipeline(double, add_three) (from S2) to 5.

8️⃣ Solutions

Here are the complete solutions with explanations and commentary on tricky concepts.

1️⃣ Lambda Functions - Solutions

1. S1: Squaring Lambda

```
square = lambda x: x * x
result_s1 = square(12) # 144
```

- *Explanation:* The simple lambda defines a function that executes the expression x * x on its argument x.

2. S2: String Case Check

```
check_uppercase_start = lambda s: s[0].isupper() if s else False
result_s2a = check_uppercase_start("Hello") # True
result_s2b = check_uppercase_start("world") # False
```

- *Explanation:* Uses a ternary expression (if s else False) for safety, checking if the string is non-empty before accessing the first element and checking its case.

3. M1: Sorting List of Tuples

```
data_m1 = [('alpha', 3), ('beta', 1), ('gamma', 5), ('delta', 2)]
sorted_m1 = sorted(data_m1, key=lambda item: item[1])
# [('beta', 1), ('delta', 2), ('alpha', 3), ('gamma', 5)]
```

- *Explanation:* The lambda item: item[1] extracts the second element of the tuple for comparison, driving the sort order.

4. M2: Filtering by Value

```
numbers_m2 = [5, 12, 8, 25, 3, 15]
filtered_m2 = list(filter(lambda x: x > 10, numbers_m2))
# [12, 25, 15]
```

- *Explanation:* filter() calls the lambda for every element; if the lambda returns True, the element is included in the output.

5. H1: Dictionary Key Transformation

```
data_h1 = [{'id': 101, 'name': 'alice'}, {'id': 102, 'name': 'bob'}, {'id': 103, 'name': 'charlie'}]
transformed_h1 = list(map(lambda d: {'name': d['name'].upper()}, data_h1))
```

```
# [{'name': 'ALICE'}, {'name': 'BOB'}, {'name': 'CHARLIE'}]
```

- *Explanation:* map() applies the lambda to each dictionary d. The lambda constructs a *new* dictionary using only the 'name' key, capitalized.

6. H2: Nested Dictionary Sorting

```
data_h2 = [
   {'user': 'a', 'metrics': {'score': 95, 'time': 120}},
   {'user': 'b', 'metrics': {'score': 88, 'time': 90}},
   {'user': 'c', 'metrics': {'score': 92, 'time': 150}}
]
sorted_h2 = sorted(data_h2, key=lambda d: d['metrics']['score'])
# Sorted by score: 88, 92, 95
```

- *Explanation:* The lambda accesses the dictionary d, then the nested dictionary 'metrics', and finally the key 'score', providing a deep key for sorting.

2️⃣ Closures and Freezing Variables - Solutions

7. S1: Basic Multiplier Closure

```
def make_multiplier(x):
   return lambda y: x * y

times_five = make_multiplier(5)
result_s7 = times_five(10) # 50
```

- *Explanation:* The inner lambda function forms a closure, remembering the value of x=5 from the make_multiplier scope even after the outer function finishes.

8. S2: Static Message Generator

```
def greet_person(name):
   fixed_greeting = "Hello, " + name
   return lambda: print(fixed_greeting)

greet_bob = greet_person("Bob")
# greet_bob() will print "Hello, Bob"
```

- *Explanation:* The closure captures the value of fixed_greeting computed in the outer scope, which includes the specific name.

9. M3: The Late-Binding Trap

```
def create_functions():
    return [lambda: i for i in range(3)]

functions_m9 = create_functions()
results_m9 = [f() for f in functions_m9]
# Result: [2, 2, 2]
```

- *Explanation (Tricky Part):* This is the classic late-binding trap. The variable i is not evaluated (frozen) when the lambda is created. Since i is a free variable, the functions look up its value *when they are executed*. By that time, the for loop has completed, and i has its final value, which is 2.

10. M4: Fixing Late-Binding with Default Arguments

```
def create_functions_fixed():
    # Fix: Use a default argument (i=i) to evaluate and freeze the value of i.
    return [lambda i_fixed=i: i_fixed for i in range(3)]

functions_m10 = create_functions_fixed()
results_m10 = [f() for f in functions_m10]
# [0, 1, 2]
```

- *Explanation (Tricky Part):* By setting a default argument i_fixed=i, the value of the loop variable i is evaluated at the time of the lambda's definition (i.e., *early binding* of the default argument), effectively freezing the intended value in the lambda's signature.

11. H1: Closure for Dynamic Suffixing

```
def suffix_generator(base_word):
    return lambda suffixes: [
        suffix + base_word
        for suffix in suffixes if len(suffix) > 3
    ]

gen_report_name = suffix_generator("Analysis")
```

```
suffixes_h1 = ["Draft", "Final", "Summary", "Metrics", "Prelim"]
result_h1 = gen_report_name(suffixes_h1)
# ['FinalAnalysis', 'SummaryAnalysis', 'MetricsAnalysis', 'PrelimAnalysis']
```

- *Explanation:* The closure captures base_word. The returned lambda uses a list comprehension with a filter to process the input suffixes list, only appending base_word if the condition is met.

12. H2: Function Generator for Custom Filters

```
def filter_factory(min_length):
    return lambda words: list(filter(lambda w: len(w) >= min_length, words))

long_word_filter = filter_factory(7)
words_h2 = ["apple", "banana", "kiwi", "grapefruit", "orange"]
result_h2 = long_word_filter(words_h2)
# ['grapefruit']
```

- *Explanation:* The factory captures min_length. The returned function takes words and uses filter with an *inner* lambda that checks the length against the closed-over min_length.

3️⃣ Conditional / Ternary Expressions - Solutions

13. S1: Ternary Parity Check

```
number_s1 = 45
parity_s1 = "Even" if number_s1 % 2 == 0 else "Odd"
# "Odd"
```

- *Format:* value_if_true if condition else value_if_false

14. S2: Status Assignment

```
is_logged_in_s2 = True
status_s2 = "Active" if is_logged_in_s2 else "Inactive"
# "Active"
```

15. M3: Ternary in a Lambda

```
transform_m3 = lambda x: x * x if x > 0 else x
result_m3a = transform_m3(4) # 16
result_m3b = transform_m3(-5) # -5
```

16. M4: Nested Ternary (Three States)

```
temp_m4 = 30
classification_m4 = "Cold" if temp_m4 <= 0 else ("Hot" if temp_m4 > 25 else "Warm")
# "Hot"
```

- *Explanation (Tricky Part):* The structure is true_val_A if cond_A else (true_val_B if cond_B else false_val_B). Note that the second ternary expression is enclosed in parentheses for clarity, though not strictly required by Python syntax.

17. H1: Dynamic Function Selection

```
data_h17 = [10, 5, 20, 8]
use_max = False

selected_func = max if use_max else min
result_h17 = selected_func(data_h17)
# 5
```

- *Explanation:* The ternary expression selects the function object itself (max or min), which is then called on the data.

18. H2: Combining Ternary, Filter, and Map

```
numbers_h18 = [2, 7, 4, 11, 8, 3]
processed_h18 = list(map(
   lambda x: x * x if x % 2 == 0 else x,
   filter(lambda x: x >= 5, numbers_h18)
))
# [7, 11, 64]
```

- *Explanation:* The filter runs first, reducing the list to [7, 11, 8]. Then, map applies the

lambda/ternary: 7 is odd (7), 11 is odd (11), 8 is even (64).

4️⃣ List Comprehensions - Solutions

19. S1: String Lengths

```
words_s19 = ["functional", "python", "closure", "lambda"]
lengths_s19 = [len(word) for word in words_s19]
# [10, 6, 7, 6]
```

20. S2: Filtering Vowels

```
text_s20 = "Advanced Programming Exercise"
vowels = "aeiouAEIOU"
consonants_s20 = [char for char in text_s20 if char not in vowels]
# ['A', 'd', 'v', 'n', 'c', 'd', ' ', 'P', 'r', 'g', 'r', 'm', 'm', 'n', 'g', ' ', 'E', 'x', 'r', 'c', 's']
```

21. M3: Filter and Map

```
numbers_m21 = [2, 15, 7, 20, 5, 12]
cubes_m21 = [num ** 3 for num in numbers_m21 if num > 10]
# [3375, 8000, 1728]
```

- *Order of execution:* The if num > 10 filter runs first, selecting the elements, and then the expression num ** 3 is evaluated.

22. M4: Flattening a 2D List

```
matrix_m22 = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
flat_m22 = [item for row in matrix_m22 for item in row]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- *Order of execution (Tricky Part):* The order of the for loops in a nested comprehension is the same as nested conventional loops: the outermost loop variable (row) changes slowest.

23. H1: Nested Comprehension with Filter (Coordinated Iteration)

```
primes = {2, 3, 5, 7}
```

```
prime_pairs_h23 = [
    (i, j) for i in range(5) for j in range(5) if i + j in primes
]
# Length: 14
```

24. H2: Dictionary Comprehension with Ternary

```
words_h24 = ["apple", "banana", "kiwi", "grapefruit", "date"]
classification_h24 = {
    word: "Long" if len(word) > 6 else "Short"
    for word in words_h24
}
# {'apple': 'Short', 'banana': 'Long', 'kiwi': 'Short', 'grapefruit': 'Long', 'date': 'Short'}
```

5️⃣ Higher-Order Functions - Solutions

25. S1: apply_to_list

```
def apply_to_list(func, data):
    return [func(item) for item in data]

def add_one(x):
    return x + 1

result_s25 = apply_to_list(add_one, [1, 2, 3])
# [2, 3, 4]
```

26. S2: Basic Function Composition

```
def compose_two(f, g):
    return lambda x: f(g(x))

def double(x): return x * 2
def add_three(x): return x + 3

h = compose_two(double, add_three)
result_s26 = h(5) # double(add_three(5)) -> double(8) -> 16
# 16
```

27. M3: make_power

```
def make_power(p):
    return lambda x: x ** p

square = make_power(2)
cube = make_power(3)

result_m27a = square(4) # 16
result_m27b = cube(2) # 8
```

28. M4: Generalized apply_twice

```
def apply_twice(func):
    return lambda x: func(func(x))

def inc(x): return x + 1

double_inc = apply_twice(inc)
result_m28 = double_inc(5) # 7
```

29. H1: Functional reduce Implementation

```
def my_reduce(func, iterable, initializer=None):
    it = iter(iterable)
    if initializer is None:
        try:
            value = next(it)
        except StopIteration:
            raise TypeError('reduce() of empty sequence with no initial value')
    else:
        value = initializer
    for element in it:
        value = func(value, element) # Cumulative application
    return value

result_h29 = my_reduce(lambda x, y: x + y, [10, 20, 30], 100)
```

```
# 160
```

30. H2: Generalized compose

```
def compose(*functions):
    def inner(x):
        # Applies functions right-to-left
        result = x
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

def f(x): return x * 10
def g(x): return x + 1
def h(x): return x - 2

composed_func = compose(f, g, h)
result_h30 = composed_func(5) # f(g(h(5))) = f(g(3)) = f(4) = 40
# 40
```

6️⃣ Mixed Advanced Exercises - Solutions

31. M1: Closure-Filtered Mapping

```
def transform_if_short(max_len):
    return lambda words: [
        len(word) if len(word) < max_len else word
        for word in words
    ]

transformer_m31 = transform_if_short(5)
result_m31 = transformer_m31(["cat", "dog", "elephant", "kite"])
# [3, 3, 'elephant', 4]
```

32. M2: Nested Comprehension with HOF and Lambda

```
square_if_even_m32 = [
```

```
    (i, j) for i in range(5)
    for j in [i * i] if j % 2 == 0 # j must be even
]
# [(0, 0), (2, 4), (4, 16)]
```

- *Explanation:* This uses a technique where the inner loop only iterates once (over [i * i]), effectively treating it as a local variable binding within the comprehension, allowing for filtering based on the computed value j.

33. H1: Deep Filtering using Ternary and HOF

```
records_h33 = [
    {'status': 'OK', 'priority': 'LOW', 'time': 20},
    {'status': 'OK', 'priority': 'HIGH', 'time': 60},
    {'status': 'ERROR', 'priority': 'LOW', 'time': 100},
    {'status': 'OK', 'priority': 'HIGH', 'time': 40},
]

filtered_h33 = list(filter(
    lambda r: r['status'] == 'ERROR' or (r['priority'] == 'HIGH' and r['time'] < 50),
    records_h33
))
# Records 2 and 4 are kept.
```

34. H2: Late-Binding Fix in List Comprehension

```
functions_h34 = [
    lambda i_fixed=i: i_fixed * 10
    for i in range(5)
]
results_h34 = [f() for f in functions_h34]
# [0, 10, 20, 30, 40]
```

- *Tricky Part:* The i_fixed=i sets the default argument's value during the definition phase, solving the late-binding issue.

35. H3: Closure for Conditional List Processing

```
def conditional_processor(threshold):
```

```
    return lambda nums: [
        n * 2 if n > threshold else n / 2
        for n in nums
    ]

processor = conditional_processor(10)
result_h35 = processor([5, 12, 8, 20, 1])
# [2.5, 24, 4.0, 40, 0.5]
```

7️⃣ Paragraphic/Scenario-based Hard Exercises - Solutions

36. P1: Inventory Price Calculator

```
data_p1 = [{'name': 'A', 'stock': 10, 'price': 50.0}, {'name': 'B', 'stock': 0, 'price': 10.0}, {'name':
'C', 'stock': 5, 'price': 100.0}]
realizable_value = [
    d['stock'] * d['price'] if d['stock'] > 0 else 0
    for d in data_p1
]
# [500.0, 0, 500.0]
```

37. P2: Secure Log Filter

```
def create_security_filter(level):
    # The returned function uses the closed-over 'level'
    return lambda logs: list(filter(lambda log: log.get('severity', 0) >= level, logs))

logs_p2 = [{'severity': 20}, {'severity': 60}, {'severity': 40}, {'severity': 80}]
high_severity_filter = create_security_filter(50)
filtered_logs = high_severity_filter(logs_p2)
# [{'severity': 60}, {'severity': 80}]
```

38. P3: Dynamic Suffix Generator with Late-Binding Fix

```
functions_p3 = [
    lambda s, n_fixed=n: s + str(n_fixed)
    for n in range(1, 6)
]
```

```
result_p3 = functions_p3[2]("Item") # The 3rd function (index 2) should use n=3
# "Item3"
```

39. P4: Composed Data Normalizer

```
def compose(*functions):
    def inner(x):
        result = x
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

def half(x): return x / 2
def add_ten(x): return x + 10
def stringify(x): return f"Value: {x}"

normalize_data = compose(stringify, add_ten, half)
result_p4 = normalize_data(90) # stringify(add_ten(half(90))) -> stringify(add_ten(45)) ->
stringify(55)
# "Value: 55.0"
```

40. P5: Employee Bonus Evaluator

```
dept_data_p5 = [{'dept': 'HR', 'employees': [100000, 50000]}, {'dept': 'ENG', 'employees':
[120000, 150000, 80000]}]

bonuses = [
    salary * 0.10 if salary > 100000 else salary * 0.05
    for dept in dept_data_p5
    for salary in dept['employees']
]
# [5000.0, 2500.0, 12000.0, 15000.0, 4000.0]
```

41. P6: Custom Sort Utility

```
def create_key_sorter(key_name):
    # Returns a lambda suitable for the 'key' argument
```

```
    return lambda d: d[key_name]

people_p6 = [{'name': 'A', 'age': 30}, {'name': 'B', 'age': 25}, {'name': 'C', 'age': 35}]
age_sorter = create_key_sorter('age')
sorted_people = sorted(people_p6, key=age_sorter)
# Sorted by age: 25, 30, 35
```

42. P7: Conditional String Formatter

```
names_p7 = ["alice", "bob", "charlie", "david"]
vowels_p7 = "aeiou"

formatted_names = list(map(
    lambda name: name.upper() if name[0].lower() in vowels_p7 else name.title(),
    names_p7
))
# ['ALICE', 'Bob', 'Charlie', 'David']
```

43. P8: Recursive Closure for Sequences

```
def fibonacci_generator():
    a, b = 0, 1 # State maintained in the closure
    def next_fib():
        nonlocal a, b
        current = a
        a, b = b, a + b
        return current
    return next_fib

fib_next = fibonacci_generator()
sequence = [fib_next() for _ in range(8)]
# [0, 1, 1, 2, 3, 5, 8, 13]
```

- *Tricky Part:* Requires the use of the nonlocal keyword to modify the state variables (a and b) captured from the outer scope.

44. P9: Flatten and Filter Multi-Level Data

```
groups_p9 = [[(1, 10), (2, 5)], [(3, 12), (4, 4)]]
```

```
high_values = [
    val for group in groups_p9 for key, val in group if val > 8
]
# [10, 12]
```

45. P10: Pipeline Function Creator

```
def pipeline(*fns):
    def inner(x):
        # Applies functions left-to-right
        result = x
        for func in fns:
            result = func(result)
        return result
    return inner

# Functions from S2
def double(x): return x * 2
def add_three(x): return x + 3

piped_func = pipeline(double, add_three)
result_p10 = piped_func(5) # add_three(double(5)) -> add_three(10) -> 13
# 13
```