Python Programming Bootcamp
Week 7: Advanced Python Functions & Techniques

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

Learning Objectives

By the end of this week, you will be able to:

- Master functional programming concepts
- Use generators for memory-efficient programming
- Create and apply custom decorators
- Implement advanced data processing patterns
- Work with iterators and iterables effectively
- Apply functional programming paradigms
- Build efficient, elegant Python solutions
- Understand and use closures

Week 7 Overview

What We'll Cover Today:

- Lambda Functions (Anonymous Functions)
- Advanced Built-in Functions: map(), filter(), reduce()
- Generators and the yield Statement
- Decorators and Function Modification
- Context Managers and the with Statement
- Functional Programming Patterns
- Practical Applications

Quick Recap: Previous Weeks

Weeks 1-4:

- Python fundamentals, data structures, functions, file I/O, error handling

Weeks 5-6: OOP

- Classes, objects, inheritance, polymorphism
- Encapsulation, abstraction, magic methods

This week: Advanced functional programming techniques to write more elegant, efficient code!

Lambda Functions

Lambda Functions: Anonymous Functions i

What are Lambda Functions?

- Small, anonymous functions defined with `lambda` keyword
- Can have multiple arguments but only one expression
- Return value is the result of the expression
- Useful for short, simple operations

Syntax:

```python
# lambda arguments: expression

# Regular function
def add(x, y):
    return x + y

```

Lambda Functions: Anonymous Functions ii

```python
# Equivalent lambda
add_lambda = lambda x, y: x + y

print(add(3, 5))          # 8
print(add_lambda(3, 5))  # 8
```

Lambda Functions: Anonymous Functions iii

Common use cases:

```
# Single argument
square = lambda x: x ** 2
print(square(5))  # 25

# Multiple arguments
multiply = lambda x, y: x * y
print(multiply(3, 4))  # 12

# With default arguments
greet = lambda name, msg="Hello": f"{msg}, {name}!"
print(greet("Alice"))             # "Hello, Alice!"
print(greet("Bob", "Hi"))         # "Hi, Bob!"

# With conditional expression
max_value = lambda a, b: a if a > b else b
```

Lambda Functions: Anonymous Functions iv

```python
print(max_value(10, 20))  # 20

# Multiple operations (use tuple)
calculate = lambda x: (x + 1, x * 2, x ** 2)
print(calculate(5))  # (6, 10, 25)
```

Lambda with Sorting i

Custom sort keys:

```
# Sort by second element of tuple
points = [(1, 5), (3, 2), (2, 8), (4, 1)]
sorted_points = sorted(points, key=lambda x: x[1])
print(sorted_points)  # [(4, 1), (3, 2), (1, 5), (2,
    8)]

# Sort strings by length
words = ["python", "is", "awesome", "!"]
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # ['!', 'is', 'python', 'awesome
    ']

# Sort dictionaries by value
students = [
```

Lambda with Sorting ii

```
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 92},
    {"name": "Charlie", "grade": 78}
]
sorted_students = sorted(students, key=lambda x: x["
    grade"], reverse=True)
for student in sorted_students:
    print(f"{student['name']}: {student['grade']}")
# Bob: 92
# Alice: 85
# Charlie: 78
```

Advanced Built-in Functions

map() Function i

What is map()?

- Applies a function to every item in an iterable
- Returns a map object (iterator)
- Syntax: `map(function, iterable)`
- More efficient than list comprehensions for simple operations

```
# Square all numbers
numbers = [1, 2, 3, 4, 5]
squared = map(lambda x: x ** 2, numbers)
print(list(squared))  # [1, 4, 9, 16, 25]

# Convert to uppercase
words = ["hello", "world", "python"]
uppercase = map(str.upper, words)
```

map() Function ii

```
print(list(uppercase))  # ['HELLO', 'WORLD', 'PYTHON']

# Convert strings to integers
str_numbers = ["1", "2", "3", "4", "5"]
int_numbers = map(int, str_numbers)
print(list(int_numbers))  # [1, 2, 3, 4, 5]
```

map() Function iii

map() with multiple iterables:

```python
# Add corresponding elements
list1 = [1, 2, 3, 4]
list2 = [10, 20, 30, 40]
sums = map(lambda x, y: x + y, list1, list2)
print(list(sums))  # [11, 22, 33, 44]

# Multiply corresponding elements
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
products = map(lambda x, y: x * y, numbers1, numbers2)
print(list(products))  # [4, 10, 18]

# Practical example: Calculate total prices
quantities = [2, 3, 1, 5]
prices = [10.50, 8.25, 15.00, 3.75]
```

map() Function iv

```
totals = map(lambda q, p: q * p, quantities, prices)
print(list(totals))  # [21.0, 24.75, 15.0, 18.75]
```

filter() Function i

What is filter()?

- Filters items based on a condition (function returns True/False)
- Returns a filter object (iterator)
- Syntax: `filter(function, iterable)`
- Keeps only items where function returns True

```
# Filter even numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = filter(lambda x: x % 2 == 0, numbers)
print(list(evens))  # [2, 4, 6, 8, 10]

# Filter positive numbers
numbers = [-5, -2, 0, 3, 7, -1, 9]
positives = filter(lambda x: x > 0, numbers)
```

filter() Function ii

```python
print(list(positives))  # [3, 7, 9]

# Filter long words (more than 5 characters)
words = ["hi", "python", "is", "awesome", "!"]
long_words = filter(lambda x: len(x) > 5, words)
print(list(long_words))  # ['python', 'awesome']
```

filter() Function iii

Practical filtering examples:

```
# Filter valid email addresses (simple check)
emails = ["user@example.com", "invalid", "test@test.
    org", "bad@"]
valid_emails = filter(lambda x: "@" in x and "." in x.
    split("@")[-1], emails)
print(list(valid_emails))  # ['user@example.com', '
    test@test.org']

# Filter students who passed (grade >= 60)
students = [
    {"name": "Alice", "grade": 85},
    {"name": "Bob", "grade": 55},
    {"name": "Charlie", "grade": 92},
    {"name": "Diana", "grade": 48}
]
```

filter() Function iv

```
passed = filter(lambda s: s["grade"] >= 60, students)
for student in passed:
    print(f"{student['name']}: {student['grade']}")
# Alice: 85
# Charlie: 92

# Filter non-empty strings
data = ["hello", "", "world", "", "python", ""]
non_empty = filter(lambda x: x != "", data)
# Or simply: filter(None, data)
print(list(non_empty))  # ['hello', 'world', 'python']
```

reduce() Function i

What is reduce()?

- Applies a function cumulatively to items in an iterable
- Reduces the iterable to a single value
- Must import from `functools` module
- Syntax: `reduce(function, iterable, initializer)`

```
from functools import reduce

# Sum all numbers
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers)
print(total)  # 15

# How it works:
```

reduce() Function ii

```
# Step 1: 1 + 2 = 3
# Step 2: 3 + 3 = 6
# Step 3: 6 + 4 = 10
# Step 4: 10 + 5 = 15

# Multiply all numbers
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)
print(product)  # 120 (factorial of 5)
```

reduce() Function iii

reduce() with initial value:

```
from functools import reduce

# Sum with initial value
numbers = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, numbers, 10)
print(total)  # 25 (10 + 1 + 2 + 3 + 4 + 5)

# Find maximum value
numbers = [5, 2, 9, 1, 7, 3]
maximum = reduce(lambda x, y: x if x > y else y,
    numbers)
print(maximum)  # 9

# Concatenate strings
words = ["Python", " ", "is", " ", "awesome"]
```

reduce() Function iv

```
sentence = reduce(lambda x, y: x + y, words)
print(sentence)  # "Python is awesome"

# Flatten nested list
nested = [[1, 2], [3, 4], [5, 6]]
flattened = reduce(lambda x, y: x + y, nested)
print(flattened)  # [1, 2, 3, 4, 5, 6]
```

Combining map, filter, and reduce i

```python
from functools import reduce

# Example: Sum of squares of even numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Step 1: Filter even numbers
evens = filter(lambda x: x % 2 == 0, numbers)

# Step 2: Square them
squared = map(lambda x: x ** 2, evens)

# Step 3: Sum them
result = reduce(lambda x, y: x + y, squared)

print(result)  # 220 (2^2 + 4^2 + 6^2 + 8^2 + 10^2)
```

Combining map, filter, and reduce ii

```
# Or in one line (chained):
result = reduce(
    lambda x, y: x + y,
    map(lambda x: x ** 2, filter(lambda x: x % 2 == 0,
     numbers))
)
print(result)  # 220
```

Combining map, filter, and reduce iii

Practical example: Data processing pipeline:

```
from functools import reduce

# Sales data processing
sales = [
    {"product": "Laptop", "price": 1000, "quantity":
    2},
    {"product": "Mouse", "price": 25, "quantity": 5},
    {"product": "Keyboard", "price": 75, "quantity":
    3},
    {"product": "Monitor", "price": 300, "quantity":
    1}
]

# Calculate total revenue
# Step 1: Map to get totals for each product
```

Combining map, filter, and reduce iv

```
totals = map(lambda x: x["price"] * x["quantity"],
    sales)

# Step 2: Reduce to sum all totals
total_revenue = reduce(lambda x, y: x + y, totals)
print(f"Total Revenue: ${total_revenue}")  # $2550

# Filter expensive items (price > 50) and get total
expensive_items = filter(lambda x: x["price"] > 50,
    sales)
expensive_revenue = reduce(
    lambda x, y: x + y,
    map(lambda x: x["price"] * x["quantity"],
    expensive_items)
)
```

Combining map, filter, and reduce v

```
print(f"Expensive Items Revenue: ${expensive_revenue}"
    )  # $2525
```

Generators and yield

Generators: Memory-Efficient Iteration

What are Generators?

- Functions that return an iterator using `yield`
- Generate values on-the-fly (lazy evaluation)
- Memory efficient - don't store all values
- State is preserved between calls
- Can only iterate once

Why Use Generators?

- Handle large datasets without loading everything into memory
- Infinite sequences
- Improved performance for large iterations
- Cleaner code than custom iterators

Creating Generators with yield i

```
# Regular function returns list (all at once)
def get_numbers_list(n):
    result = []
    for i in range(n):
        result.append(i ** 2)
    return result

# Generator function yields values (one at a time)
def get_numbers_generator(n):
    for i in range(n):
        yield i ** 2

# Usage comparison
numbers_list = get_numbers_list(5)
print(numbers_list)  # [0, 1, 4, 9, 16] - all stored
    in memory
```

Creating Generators with yield ii

```
numbers_gen = get_numbers_generator(5)
print(numbers_gen)  # <generator object>

# Iterate through generator
for num in numbers_gen:
    print(num)  # 0, 1, 4, 9, 16 - one at a time
```

Creating Generators with yield iii

Memory efficiency demonstration:

```python
import sys

# List approach - stores everything
def large_list(n):
    return [i for i in range(n)]

# Generator approach - generates on demand
def large_generator(n):
    for i in range(n):
        yield i

# Memory comparison
big_list = large_list(1000000)
print(f"List size: {sys.getsizeof(big_list)} bytes")
# List size: ~8000000 bytes (8 MB)
```

Creating Generators with yield iv

```
big_gen = large_generator(1000000)
print(f"Generator size: {sys.getsizeof(big_gen)} bytes
    ")
# Generator size: ~120 bytes

# Both produce same results, but generator uses
    minimal memory!
```

Practical Generator Examples i

```
# Fibonacci sequence generator
def fibonacci(n):
    a, b = 0, 1
    count = 0
    while count < n:
        yield a
        a, b = b, a + b
        count += 1

# Generate first 10 Fibonacci numbers
for num in fibonacci(10):
    print(num, end=" ")
# 0 1 1 2 3 5 8 13 21 34

# Infinite Fibonacci (careful!)
def fibonacci_infinite():
```

Practical Generator Examples ii

```
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Use with caution - it's infinite!
fib = fibonacci_infinite()
print([next(fib) for _ in range(10)])
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Practical Generator Examples iii

Reading large files with generators:

```
# Memory-inefficient approach
def read_file_all(filename):
    with open(filename, 'r') as file:
        return file.readlines()  # Loads entire file!

# Memory-efficient generator approach
def read_file_generator(filename):
    with open(filename, 'r') as file:
        for line in file:
            yield line.strip()

# Process large file without loading it all
for line in read_file_generator('large_file.txt'):
    # Process one line at a time
    if 'ERROR' in line:
```

Practical Generator Examples iv

```
        print(line)

# Generator for CSV processing
def read_csv_rows(filename):
    import csv
    with open(filename, 'r') as file:
        reader = csv.DictReader(file)
        for row in reader:
            yield row

# Process CSV one row at a time
for row in read_csv_rows('data.csv'):
    print(row['name'], row['age'])
```

Generator Expressions i

Generator Expressions

- Similar to list comprehensions but with parentheses
- Create generators in a concise way
- Syntax: `(expression for item in iterable if condition)`
- More memory-efficient than list comprehensions

```
# List comprehension - creates list immediately
squares_list = [x**2 for x in range(10)]
print(squares_list)  # [0, 1, 4, 9, 16, 25, 36, 49,
    64, 81]

# Generator expression - creates generator
squares_gen = (x**2 for x in range(10))
print(squares_gen)  # <generator object>
```

Generator Expressions ii

```
print(list(squares_gen))  # [0, 1, 4, 9, 16, 25, 36,
    49, 64, 81]

# With filtering
evens_gen = (x for x in range(20) if x % 2 == 0)
print(list(evens_gen))  # [0, 2, 4, 6, 8, 10, 12, 14,
    16, 18]

# Sum using generator (memory efficient)
total = sum(x**2 for x in range(1000000))
print(total)  # Large number, but uses minimal memory
```

Decorators

Decorators: Function Modifiers

What are Decorators?

- Functions that modify or enhance other functions
- Wrap existing functions with additional functionality
- Use `@decorator_name` syntax
- Common uses: logging, timing, authentication, caching

Key Concepts

- **First-class functions**: Functions are objects
- **Higher-order functions**: Functions that take/return functions
- **Closures**: Inner functions that remember outer scope
- **Wrapper pattern**: Decorating without modifying original

Understanding Closures First i

```
# Functions are first-class objects
def greet(name):
    return f"Hello, {name}!"

# Assign to variable
say_hello = greet
print(say_hello("Alice"))  # "Hello, Alice!"

# Pass as argument
def execute_function(func, value):
    return func(value)

print(execute_function(greet, "Bob"))  # "Hello, Bob!"

# Return from function
def get_greeting_function():
```

Understanding Closures First ii

```
    def greet_inner(name):
        return f"Hi, {name}!"
    return greet_inner

greet_func = get_greeting_function()
print(greet_func("Charlie"))  # "Hi, Charlie!"
```

Understanding Closures First iii

Closures - Inner functions remember outer scope:

```python
def make_multiplier(factor):
    def multiply(number):
        return number * factor  # Remembers 'factor'
    return multiply

# Create specialized functions
double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))  # 10
print(triple(5))  # 15

# Another closure example
def make_counter():
    count = 0
```

Understanding Closures First iv

```
    def increment():
        nonlocal count  # Access outer variable
        count += 1
        return count
    return increment

counter = make_counter()
print(counter())  # 1
print(counter())  # 2
print(counter())  # 3
```

Creating Simple Decorators i

```python
# Basic decorator structure
def my_decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

# Manual decoration
def say_hello():
    print("Hello!")

say_hello = my_decorator(say_hello)
say_hello()
# Before function call
# Hello!
```

Creating Simple Decorators ii

```python
# After function call

# Using @ syntax (syntactic sugar)
@my_decorator
def say_goodbye():
    print("Goodbye!")

say_goodbye()
# Before function call
# Goodbye!
# After function call
```

Creating Simple Decorators iii

Decorator with arguments:

```python
def my_decorator(func):
    def wrapper(*args, **kwargs):  # Accept any
   arguments
        print(f"Calling {func.__name__}")
        result = func(*args, **kwargs)
        print(f"Finished {func.__name__}")
        return result
    return wrapper

@my_decorator
def add(a, b):
    return a + b

@my_decorator
def greet(name, greeting="Hello"):
```

Creating Simple Decorators iv

```python
    return f"{greeting}, {name}!"

print(add(3, 5))
# Calling add
# Finished add
# 8

print(greet("Alice", greeting="Hi"))
# Calling greet
# Finished greet
# Hi, Alice!
```

Practical Decorator Examples i

```
import time

# Timing decorator
def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f}
 seconds")
        return result
    return wrapper

@timer
def slow_function():
    time.sleep(1)
```

Practical Decorator Examples ii

```
    return "Done!"

print(slow_function())
# slow_function took 1.0001 seconds
# Done!

@timer
def calculate_sum(n):
    return sum(range(n))

result = calculate_sum(1000000)
# calculate_sum took 0.0234 seconds
print(result)
```

Practical Decorator Examples iii

Logging decorator:

```python
def logger(func):
    def wrapper(*args, **kwargs):
        print(f"LOG: Calling {func.__name__}")
        print(f"LOG: Arguments: {args}, {kwargs}")
        result = func(*args, **kwargs)
        print(f"LOG: {func.__name__} returned {result}
   ")
        return result
    return wrapper

@logger
def divide(a, b):
    return a / b

result = divide(10, 2)
```

Practical Decorator Examples iv

```python
# LOG: Calling divide
# LOG: Arguments: (10, 2), {}
# LOG: divide returned 5.0

# Multiple decorators
@timer
@logger
def multiply(a, b):
    return a * b

result = multiply(3, 4)
# Applied bottom-to-top: logger first, then timer
```

Decorators with Parameters i

```python
# Decorator that takes arguments
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

@repeat(times=3)
def greet(name):
    print(f"Hello, {name}!")
    return f"Greeted {name}"

greet("Alice")
```

Decorators with Parameters ii

```
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!

# Another example: Cache with timeout
def cache_with_expiry(seconds):
    def decorator(func):
        cache = {}
        def wrapper(*args):
            import time
            key = args
            if key in cache:
                result, timestamp = cache[key]
                if time.time() - timestamp < seconds:
                    print("Returning cached result")
                    return result
```

Decorators with Parameters iii

```
            result = func(*args)
            cache[key] = (result, time.time())
            return result
        return wrapper
    return decorator
```

Context Managers

Context Managers with 'with' Statement i

What are Context Managers?

- Manage resources (files, connections, locks)
- Ensure proper setup and cleanup
- Use `with` statement
- Implement `__enter__` and `__exit__` methods

```
# Without context manager (manual cleanup)
file = open('data.txt', 'r')
try:
    content = file.read()
    print(content)
finally:
    file.close()  # Must remember to close!
```

Context Managers with 'with' Statement ii

```
# With context manager (automatic cleanup)
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)
# File automatically closed after block!
```

Creating Custom Context Managers i

```
# Using class
class Timer:
    def __enter__(self):
        import time
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        import time
        self.end = time.time()
        print(f"Elapsed time: {self.end - self.start
    :.4f} seconds")
        return False  # Don't suppress exceptions

# Usage
with Timer():
```

Creating Custom Context Managers ii

```
    # Code to time
    total = sum(range(1000000))
# Elapsed time: 0.0234 seconds

# Using contextlib
from contextlib import contextmanager

@contextmanager
def timer_context():
    import time
    start = time.time()
    yield  # Everything before yield is __enter__
    end = time.time()  # Everything after yield is
   __exit__
    print(f"Elapsed: {end - start:.4f} seconds")
```

Creating Custom Context Managers iii

```
with timer_context():
    total = sum(range(1000000))
# Elapsed: 0.0234 seconds
```

Creating Custom Context Managers iv

Practical context manager examples:

```python
from contextlib import contextmanager

# Database connection manager
@contextmanager
def database_connection(db_name):
    print(f"Opening connection to {db_name}")
    connection = {"db": db_name, "connected": True}
    try:
        yield connection
    finally:
        print(f"Closing connection to {db_name}")
        connection["connected"] = False

with database_connection("mydb") as conn:
    print(f"Working with {conn['db']}")
```

Creating Custom Context Managers v

```
# Opening connection to mydb
# Working with mydb
# Closing connection to mydb

# Temporary directory changer
import os

@contextmanager
def change_dir(path):
    old_dir = os.getcwd()
    os.chdir(path)
    try:
        yield
    finally:
        os.chdir(old_dir)
```

Creating Custom Context Managers vi

```
with change_dir('/tmp'):
    print(os.getcwd())  # /tmp
print(os.getcwd())  # Back to original directory
```

Functional Programming Patterns

Functional Programming Principles

Key Concepts

- **Pure Functions**: No side effects, same input same output
- **Immutability**: Don't modify data, create new data
- **Higher-Order Functions**: Functions as arguments/return values
- **Function Composition**: Combine simple functions

Benefits

- Easier to test and debug
- More predictable code
- Better for parallel processing
- Cleaner, more expressive code

Pure Functions vs Impure Functions i

```python
# Impure function (has side effects)
total = 0
def add_to_total(x):
    global total
    total += x  # Modifies external state!
    return total

# Pure function (no side effects)
def add(x, y):
    return x + y  # Only depends on inputs

# Impure (different output each time)
import random
def get_random():
    return random.randint(1, 10)  # Non-deterministic
```

Pure Functions vs Impure Functions ii

```python
# Pure (same input      same output)
def multiply(x, y):
    return x * y

print(multiply(3, 4))  # Always 12
print(multiply(3, 4))  # Always 12
```

Function Composition i

```python
# Simple functions
def add_tax(price):
    return price * 1.1

def apply_discount(price):
    return price * 0.9

def round_price(price):
    return round(price, 2)

# Manual composition
price = 100
price = apply_discount(price)
price = add_tax(price)
price = round_price(price)
print(price)  # 99.0
```

Function Composition ii

```
# Compose function helper
def compose(*functions):
    def composed(value):
        for func in reversed(functions):
            value = func(value)
        return value
    return composed

# Create composed function
calculate_final_price = compose(round_price, add_tax,
    apply_discount)
print(calculate_final_price(100))  # 99.0
```

Practical Examples

Complete Example: Data Processing Pipeline i

```
from functools import reduce

# Sample data: list of transactions
transactions = [
    {"id": 1, "amount": 100, "type": "debit", "
   category": "food"},
    {"id": 2, "amount": 50, "type": "credit", "
   category": "salary"},
    {"id": 3, "amount": 200, "type": "debit", "
   category": "rent"},
    {"id": 4, "amount": 30, "type": "debit", "category
   ": "food"},
    {"id": 5, "amount": 1000, "type": "credit", "
   category": "salary"},
    {"id": 6, "amount": 75, "type": "debit", "category
   ": "utilities"}
```

Complete Example: Data Processing Pipeline ii

```
]

# Task 1: Calculate total debits
debits = filter(lambda t: t["type"] == "debit",
    transactions)
total_debits = reduce(lambda x, y: x + y["amount"],
    debits, 0)
print(f"Total Debits: ${total_debits}")  # $405

# Task 2: Get all food expenses
food_expenses = filter(
    lambda t: t["category"] == "food" and t["type"] ==
     "debit",
    transactions
)
```

Complete Example: Data Processing Pipeline iii

```
food_amounts = map(lambda t: t["amount"],
    food_expenses)
total_food = sum(food_amounts)
print(f"Food Expenses: ${total_food}")  # $130
```

Complete Example: Data Processing Pipeline iv

```
# Task 3: Create summary by category
from collections import defaultdict

def summarize_by_category(transactions):
    summary = defaultdict(lambda: {"debit": 0, "credit
    ": 0})
    for t in transactions:
        summary[t["category"]][t["type"]] += t["amount
    "]
    return dict(summary)

summary = summarize_by_category(transactions)
for category, amounts in summary.items():
    print(f"{category}: Debit=${amounts['debit']},
    Credit=${amounts['credit']}")
```

Complete Example: Data Processing Pipeline v

```
# food: Debit=$130, Credit=$0
# salary: Debit=$0, Credit=$1050
# rent: Debit=$200, Credit=$0
# utilities: Debit=$75, Credit=$0

# Task 4: Calculate net balance
all_debits = sum(t["amount"] for t in transactions if
    t["type"] == "debit")
all_credits = sum(t["amount"] for t in transactions if
    t["type"] == "credit")
net_balance = all_credits - all_debits
print(f"Net Balance: ${net_balance}")  # $645
```

Best Practices

When to Use Each Technique

Lambda Functions

- Simple, one-line operations
- Short-lived, throwaway functions
- As arguments to map, filter, sorted, etc.
- L Avoid for complex logic

Generators

- Large datasets
- Infinite sequences
- Memory-constrained environments
- One-time iteration

Decorators

- Cross-cutting concerns (logging, timing, auth)
- Don't modify original function code
- Reusable functionality

Functional Programming Best Practices

Do:

- Write pure functions when possible
- Use immutable data structures
- Favor function composition
- Use built-in functions (map, filter, reduce)
- Leverage generators for large data
- Keep functions small and focused

Don't:

- Overuse lambda for complex operations
- Chain too many map/filter operations (use comprehensions)
- Ignore readability for cleverness
- Use decorators when simple function calls suffice

Practice Exercises

Exercise 1: Data Transformation Pipeline

Process a list of numbers using functional programming:

Task
Given: `numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Using map, filter, and reduce:

1. Filter out odd numbers
2. Square the remaining numbers
3. Sum the squared values
4. Do it in one chained expression

Expected result: 220 ($2 + 4 + 6 + 8 + 10$)

Exercise 2: Custom Decorator

Create a caching decorator:

Task
Implement a `@cache` decorator that:

- Stores results of function calls
- Returns cached result for same arguments
- Prints "Cache hit" or "Computing..." messages
- Works with Fibonacci function

Test with:

```
@cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Exercise 3: Generator for File Processing

Create a generator to process log files:

Task
Write a generator that:

- Reads a log file line by line
- Filters lines containing "ERROR"
- Extracts timestamp and error message
- Yields parsed error information

Bonus: Chain with map/filter to further process errors

Summary

Week 7 Summary

What We Learned:

Functional Programming

- Lambda functions for simple operations
- map(), filter(), reduce() for data transformation
- Pure functions and immutability
- Function composition patterns

Advanced Techniques

- Generators and yield for memory efficiency
- Decorators for function modification
- Context managers for resource management

Skills Checklist

You should now be able to:

- Write and use lambda functions effectively
- Apply map, filter, and reduce to data
- Create generators for large datasets
- Build custom decorators
- Implement context managers
- Apply functional programming patterns
- Write memory-efficient code
- Compose functions for clean pipelines

Next Steps

Continue Practicing

- Refactor existing code to use generators
- Create your own decorator library
- Build data processing pipelines
- Explore more functional programming concepts
- Study Python's itertools module
- Practice writing pure functions

Homework

- Complete all practice exercises
- Build a data processing application using generators
- Create a timing/logging decorator suite
- Refactor old code with functional patterns

Next Week Preview

Week 8: Testing, Best Practices & Project Design

Part 1: Testing & Debugging

- Unit testing with unittest and pytest
- Test-driven development (TDD)
- Debugging techniques and tools
- Code coverage

Part 2: Best Practices & Project Design

- Code organization and documentation
- Virtual environments and pip
- Project planning and design patterns
- Git basics for version control

Capstone Project
Apply everything you've learned in a complete Python application!

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials

Center for Artificial Intelligence & Emerging Technologies