Module 7 - Functional Programming

Welcome to Module 7! In this module, we'll explore concepts from the **functional programming paradigm** that Python supports. This includes thinking about functions in a new way, writing concise "anonymous" functions, and understanding how to process data efficiently using iterators and generators.

Chapter 1: Introduction to Functional Programming

Python is a multi-paradigm language, meaning it supports several programming styles. While you've primarily been using an imperative/procedural style so far, understanding functional programming (FP) can help you write more elegant, maintainable, and often more performant code.

1.1 What is Functional Programming?

Functional Programming (FP) is a programming paradigm where programs are constructed by applying and composing functions. It treats computation as the evaluation of mathematical functions and avoids changing state and mutable data.

Core Concepts of Functional Programming:

1. **Immutability:**
    - Data structures are generally not modified after they are created. Instead of changing an existing object, you create new objects with the desired changes.
    - In Python, strings and tuples are immutable, while lists and dictionaries are mutable. Functional programming often encourages minimizing mutable data.
2. **Pure Functions:**
    - A cornerstone of FP. We'll dive deeper into this next.
3. **First-Class and Higher-Order Functions:**
    - Functions are treated like any other variable. They can be passed as arguments, returned from other functions, and assigned to variables.
4. **No Side Effects:**
    - Pure functions don't interact with the outside world beyond their inputs and outputs. This means no modifying global variables, no I/O operations (like printing or file writing) that aren't explicitly part of the return value, etc.

Contrast with Imperative/Procedural Programming:

- **Imperative/Procedural:** Focuses on *how* to achieve a result by explicitly describing a sequence of steps (e.g., "first do this, then do that, then change this variable"). It heavily relies on changing state.

  Python

  ```
  # Imperative example:
  my_list = [1, 2, 3]
  ```

```
new_list = []
for x in my_list:
    new_list.append(x * 2)
print(new_list) # State (my_list, new_list, x) changes
```

- **Functional:** Focuses on *what* needs to be computed by defining transformations on data. It minimizes state changes.

  Python

  ```
  # Functional example using map (we'll learn this soon):
  my_list = [1, 2, 3]
  new_list = list(map(lambda x: x * 2, my_list)) # No explicit loop, transformation defined
  print(new_list) # my_list remains unchanged
  ```

Benefits of Functional Programming:

- **Predictability:** Pure functions are easier to reason about because their output only depends on their inputs.
- **Easier Testing:** Pure functions are isolated and stateless, making them simple to test.
- **Concurrency:** Absence of side effects makes it easier to write programs that run in parallel without race conditions.
- **Modularity:** Promotes breaking down problems into small, reusable functions.

Python's Stance:

Python is a multi-paradigm language. It doesn't enforce a purely functional style but provides features that allow you to write code in a functional way, such as:

- First-class functions
- `lambda` functions
- `map()`, `filter()`, `reduce()` (from `functools` module)
- List, dictionary, and set comprehensions (which are often more "Pythonic" than `map`/`filter` for simple cases).
- Generators and iterators for lazy evaluation.

1.2 Pure Functions

A **pure function** is a function that satisfies two conditions:

1. **Determinism:** Given the same input arguments, it will always return the exact same output.
2. **No Side Effects:** It does not cause any observable changes outside its local scope. This means it doesn't:
   - Modify global variables.
   - Modify its input arguments (if they are mutable).
   - Perform I/O operations (like printing to console, reading/writing files, making network requests).

Examples of Pure Functions:

Python
```
# Pure Function
def add(x, y):
    """Adds two numbers."""
    return x + y

print(add(2, 3)) # Always 5
print(add(2, 3)) # Still 5 (no change in external state)

# Pure Function (list is not modified)
def square_list(numbers):
    """Returns a new list with each number squared."""
    new_list = [n * n for n in numbers] # Creates a new list
    return new_list

original = [1, 2, 3]
squared = square_list(original)
print(f"Original list: {original}") # Original list remains unchanged
print(f"Squared list: {squared}")
```

Examples of Impure Functions (with Side Effects):

Python
```
# Impure Function (modifies a global variable)
global_counter = 0

def increment_counter():
    """Increments a global counter."""
    global global_counter
    global_counter += 1
    return global_counter

print(increment_counter()) # 1 (first call)
print(increment_counter()) # 2 (second call, output changed due to previous
call's side effect)
print(f"Final counter: {global_counter}") # global_counter was modified

# Impure Function (modifies its mutable input argument)
def append_item(my_list, item):
    """Appends an item to the input list (modifies in-place)."""
    my_list.append(item) # Modifies the original list
    return my_list

data = [1, 2, 3]
modified_data = append_item(data, 4)
print(f"Original list after call: {data}") # Original list was modified!
print(f"Returned list: {modified_data}")

# Impure Function (performs I/O)
def greet(name):
    """Prints a greeting to the console."""
    print(f"Hello, {name}!") # Side effect: prints to console
    return "Greeting complete"

greet("Alice") # Prints "Hello, Alice!" to the console
```

1.3 First-Class and Higher-Order Functions

Python treats functions as "first-class citizens," which enables "higher-order functions."

1. First-Class Functions:

This means functions can be:

- Assigned to variables.
- Passed as arguments to other functions.
- Returned as values from other functions.
- Stored in data structures (lists, dictionaries).

Python
```python
# Assign function to a variable
def multiply(a, b):
    return a * b

op = multiply
print(op(5, 4)) # Calls the function through the variable 'op'
# Output: 20

# Pass function as an argument
def apply_operation(func, x, y):
    return func(x, y)

def add_numbers(a, b):
    return a + b

print(apply_operation(add_numbers, 10, 5)) # Pass add_numbers function
# Output: 15
print(apply_operation(multiply, 10, 5)) # Pass multiply function
# Output: 50

# Return function from another function
def get_math_operation(operation_type):
    if operation_type == "add":
        def add(a, b):
            return a + b
        return add
    elif operation_type == "subtract":
        def subtract(a, b):
            return a - b
        return subtract
    else:
        return None

add_func = get_math_operation("add")
print(add_func(8, 2))
# Output: 10
```

2. Higher-Order Functions:

A higher-order function is a function that either:

- Takes one or more functions as arguments.
- Returns a function as its result.

The `apply_operation` and `get_math_operation` examples above are both higher-order functions. Other common higher-order functions in Python include `map()`, `filter()`, `sorted()`, and `min()`/`max()` (when used with the `key` argument).

Understanding these concepts is foundational for effective functional programming in Python.

Chapter 2: Lambda Functions - Anonymous Functions

Lambda functions are small, anonymous functions defined without a name. They are typically used for short, throw-away functions that you need only once.

2.1 What are Lambda Functions?

- **Definition:** An anonymous function is a function that is not bound to a name. In Python, lambda functions are a way to create such functions.
- **Syntax:**

  Python

  ```
  lambda arguments: expression
  ```

  - `lambda`: The keyword used to define a lambda function.
  - `arguments`: A comma-separated list of arguments.
  - `expression`: A single expression that is evaluated and returned.
- **Key Characteristics and Limitations:**
  - **Single Expression:** A lambda function can only contain a single expression. It cannot contain statements like `if`, `for`, `while`, `return` (explicitly), assignments, etc.
  - **No Name:** They don't have a name, which is why they're called "anonymous."
  - **Implicit Return:** The result of the expression is implicitly returned. You don't use the `return` keyword.

Basic Example:

Python
```
# A regular function
def add_five(x):
    return x + 5

# The equivalent lambda function
lambda_add_five = lambda x: x + 5

print(add_five(10))
# Output: 15
print(lambda_add_five(10))
# Output: 15
```

2.2 When to Use Lambda Functions

Lambda functions are best used in situations where:

1. **You need a simple function for a short period:** The function's logic is concise and needed only in a specific context.
2. **As arguments to higher-order functions:** This is their most common use case, especially with `map()`, `filter()`, `sorted()`, `min()`, `max()`, and other functions that expect a function as an argument.

- **When NOT to use Lambda Functions:**
  - If your function requires multiple statements or complex logic. Use a regular `def` function for clarity.
  - If the function needs to be reused multiple times throughout your code.
  - If the function's name would add clarity to your code.

2.3 Examples of Lambda Functions

1. Basic Arithmetic:

Python
```
>>> # Add two numbers
>>> sum_two = lambda a, b: a + b
>>> print(sum_two(7, 3))
10

>>> # Multiply by a factor
>>> multiply_by = lambda x, factor: x * factor
>>> print(multiply_by(5, 2))
10
```

2. Using Lambda with `sorted()`:

The `sorted()` built-in function (and list's `sort()` method) accepts an optional `key` argument, which is a function that specifies a sorting key for each item. Lambda functions are perfect for this.

- **Sorting a list of tuples by the second element:**

  Python
  ```
  >>> pairs = [(1, 'b'), (3, 'a'), (2, 'c')]
  >>> # Sort based on the second element of each tuple
  >>> sorted_pairs = sorted(pairs, key=lambda pair: pair[1])
  >>> print(sorted_pairs)
  [(3, 'a'), (1, 'b'), (2, 'c')]
  ```

- **Sorting a list of dictionaries by a specific key:**

  Python
  ```
  >>> students = [
  ...     {'name': 'Alice', 'age': 30, 'grade': 'A'},
  ...     {'name': 'Bob', 'age': 25, 'grade': 'C'},
  ...     {'name': 'Charlie', 'age': 28, 'grade': 'B'}
  ```

```
... ]

>>> # Sort by age
>>> sorted_by_age = sorted(students, key=lambda student:
student['age'])
>>> print(sorted_by_age)
[{'name': 'Bob', 'age': 25, 'grade': 'C'}, {'name': 'Charlie', 'age':
28, 'grade': 'B'}, {'name': 'Alice', 'age': 30, 'grade': 'A'}]

>>> # Sort by grade (descending)
>>> sorted_by_grade_desc = sorted(students, key=lambda student:
student['grade'], reverse=True)
>>> print(sorted_by_grade_desc)
[{'name': 'C', 'age': 25, 'grade': 'C'}, {'name': 'B', 'age': 28,
'grade': 'B'}, {'name': 'A', 'age': 30, 'grade': 'A'}] # Edited for
clarity of names
```

3. Using Lambda for Conditional Logic (limited):

While you can't use `if/else` statements, you can use the ternary operator (`value_if_true if condition else value_if_false`) within a lambda's single expression.

Python
```
>>> check_even_odd = lambda num: "Even" if num % 2 == 0 else "Odd"
>>> print(check_even_odd(4))
Even
>>> print(check_even_odd(7))
Odd
```

Lambda functions provide a concise way to define small, single-use functions, often in conjunction with higher-order functions.

Chapter 3: `filter()` - Filtering Elements

The `filter()` function is a built-in higher-order function that provides an elegant way to select elements from an iterable based on a specific condition.

3.1 Introduction to `filter()`

- **Purpose:** `filter()` constructs an **iterator** from elements of an iterable for which a function returns `True` (or a truthy value). It's essentially a way to "filter out" unwanted items.
- **Syntax:**

  Python

  ```
  filter(function, iterable)
  ```

  - `function`: A function that takes one argument and returns a boolean value (`True` or `False`). This function will be applied to each item in the `iterable`.

    - `iterable`: Any sequence or collection that can be iterated over (e.g., list, tuple, set, string, range).
- **Return Value:** `filter()` returns an **iterator**. This means it generates elements on demand and doesn't create a new list in memory immediately. To see the results as a list, you typically convert it using `list()`.

3.2 Examples of `filter()`

1. Filtering Numbers (using a regular function):

Let's filter out even numbers from a list.

Python
```
# filter_even.py

def is_even(num):
    """Returns True if the number is even, False otherwise."""
    return num % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Apply filter()
even_numbers_iterator = filter(is_even, numbers)

# Convert the iterator to a list to view results
even_numbers_list = list(even_numbers_iterator)

print(f"Original numbers: {numbers}")
print(f"Even numbers: {even_numbers_list}") # Output: [2, 4, 6, 8, 10]
```

2. Filtering Strings (using `None` as function):

If the `function` argument is `None`, `filter()` will remove elements that evaluate to `False` (or `None`, `0`, `""`, `[]`, `{}`, `set()`).

Python
```
>>> data = [0, 1, "hello", "", None, True, [], 42]
>>> truthy_values = list(filter(None, data))
>>> print(truthy_values)
[1, 'hello', True, 42]
```

3. Using Lambda with `filter()` (most common use case):

Lambda functions are ideal for providing the simple filtering logic directly.

- **Filter numbers greater than 5:**

  Python

  ```
  >>> numbers = [1, 7, 3, 9, 2, 6]
  >>> greater_than_5 = list(filter(lambda x: x > 5, numbers))
  >>> print(greater_than_5)
  [7, 9, 6]
  ```

- **Filter strings that start with 'a':**

  Python

  ```
  >>> words = ["apple", "banana", "apricot", "cherry"]
  >>> a_words = list(filter(lambda word: word.startswith('a'), words))
  >>> print(a_words)
  ['apple', 'apricot']
  ```

- **Filter empty strings from a list:**

  Python

  ```
  >>> my_strings = ["apple", "", "banana", " ", "cherry", ""]
  >>> non_empty_strings = list(filter(lambda s: s.strip() != "",
  my_strings))
  >>> print(non_empty_strings)
  ['apple', 'banana', 'cherry']
  ```

Comparison with List Comprehension:

For many filtering tasks, list comprehensions can be just as readable and sometimes more performant if you need the entire list immediately.

Python
```
# Using filter()
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers_filter = list(filter(lambda x: x % 2 == 0, numbers))
print(f"Filter result: {even_numbers_filter}")

# Using List Comprehension
even_numbers_comprehension = [x for x in numbers if x % 2 == 0]
print(f"Comprehension result: {even_numbers_comprehension}")
```

Both produce `[2, 4, 6, 8, 10]`. The choice often comes down to personal preference or specific performance needs (e.g., `filter()` returns an iterator, which is memory efficient for very large datasets).

Chapter 4: `map()` - Transforming Elements

The `map()` function is another essential higher-order function that applies a given function to every item in an iterable and returns an iterator of the results.

4.1 Introduction to `map()`

- **Purpose:** `map()` takes a function and an iterable (or multiple iterables) and applies the function to each item, generating a new iterable with the results of the function calls. It's used for **transformation**.
- **Syntax:**

Python

```
map(function, iterable, ...)
```

  - `function`: The function to apply to each item. It should take the same number of arguments as there are iterables.
  - `iterable`: One or more sequences whose elements will be passed as arguments to the `function`.
- **Return Value:** `map()` returns an **iterator**. Like `filter()`, it generates results on demand. To get a list, you typically convert it using `list()`.

4.2 Examples of `map()`

1. Squaring Numbers (using a regular function):

Python
```
# map_square.py

def square(num):
    """Returns the square of a number."""
    return num * num

numbers = [1, 2, 3, 4, 5]

# Apply map()
squared_numbers_iterator = map(square, numbers)

# Convert the iterator to a list to view results
squared_numbers_list = list(squared_numbers_iterator)

print(f"Original numbers: {numbers}")
print(f"Squared numbers: {squared_numbers_list}") # Output: [1, 4, 9, 16,
25]
```

2. Converting Strings to Uppercase:

Python
```
>>> words = ["apple", "banana", "cherry"]
>>> uppercase_words = list(map(str.upper, words)) # str.upper is a method,
but callable here
>>> print(uppercase_words)
['APPLE', 'BANANA', 'CHERRY']
```

3. Using Lambda with `map()` (very common):

Lambda functions are frequently used with `map()` for concise transformations.

- **Add 10 to each number:**

  Python

  ```
  >>> numbers = [10, 20, 30]
  >>> new_numbers = list(map(lambda x: x + 10, numbers))
  >>> print(new_numbers)
  ```

```
[20, 30, 40]
```

- **Concatenate a prefix to each string:**

  Python

  ```
  >>> names = ["Alice", "Bob", "Charlie"]
  >>> greetings = list(map(lambda name: "Hello, " + name, names))
  >>> print(greetings)
  ['Hello, Alice', 'Hello, Bob', 'Hello, Charlie']
  ```

4. Using `map()` with Multiple Iterables:

The `function` argument to `map()` can accept multiple arguments, provided you pass multiple iterables. `map()` will take one item from each iterable at a time.

Python
```
>>> list1 = [1, 2, 3]
>>> list2 = [10, 20, 30]

>>> # Add corresponding elements from two lists
>>> summed_lists = list(map(lambda x, y: x + y, list1, list2))
>>> print(summed_lists)
[11, 22, 33]

>>> # Combine elements into strings
>>> names = ["Alice", "Bob"]
>>> ages = [30, 25]
>>> combined_info = list(map(lambda name, age: f"{name} is {age} years
old.", names, ages))
>>> print(combined_info)
['Alice is 30 years old.', 'Bob is 25 years old.']
```

Comparison with List Comprehension:

Like `filter()`, `map()` can often be replaced by a list comprehension for single iterable transformations.

Python
```
# Using map()
numbers = [1, 2, 3, 4, 5]
squared_map = list(map(lambda x: x * x, numbers))
print(f"Map result: {squared_map}")

# Using List Comprehension
squared_comprehension = [x * x for x in numbers]
print(f"Comprehension result: {squared_comprehension}")
```

Both produce `[1, 4, 9, 16, 25]`. For simple transformations on a single iterable, list comprehensions are often preferred for their readability. `map()` shines when the transformation logic is already encapsulated in a named function, or when dealing with multiple iterables.

Chapter 5: Iterators and Generators

Understanding iterators and generators is key to writing memory-efficient and scalable Python code, especially when dealing with large datasets or infinite sequences.

5.1 Understanding Iterables and Iterators

These two terms are often used interchangeably but have distinct meanings:

1. **Iterable:**
   - An object that can be **iterated over**. This means you can loop through its elements one by one.
   - Examples: Lists, tuples, strings, dictionaries, sets, ranges, file objects.
   - Behind the scenes, an iterable is an object that has an `__iter__()` method (which returns an iterator) or a `__getitem__()` method (which allows indexing).
   - You can use an iterable directly in a `for` loop.

   Python

   ```
   >>> my_list = [1, 2, 3] # This is an iterable
   >>> for item in my_list: # The 'for' loop knows how to iterate over
   it
   ...     print(item)
   1
   2
   3
   ```

2. **Iterator:**
   - An object that **represents a stream of data**. It knows how to give you the *next* element in the sequence and remembers its state (where it currently is in the sequence).
   - An iterator must have two methods:
     - `__iter__()`: Returns the iterator object itself.
     - `__next__()`: Returns the next item from the sequence. If there are no more items, it raises a `StopIteration` exception.
   - You don't usually create iterators directly, but you get them from iterables.

How `for` loops work internally:

When you write `for item in iterable:`, Python does roughly this:

1. Calls `iter(iterable)` to get an **iterator** object.
2. Repeatedly calls `next(iterator)` to get the next item.
3. If `next()` raises `StopIteration`, the loop terminates.

Benefits of Iterators:

- **Memory Efficiency:** Iterators produce items one at a time. They don't load all elements into memory at once, which is crucial for very large or infinite sequences.

- **Lazy Evaluation:** Elements are generated only when requested, not all at once.

5.2 Creating Iterators (`iter()` and `next()`)

You can explicitly get an iterator from an iterable using the built-in `iter()` function and then manually retrieve elements using `next()`.

Python
```
>>> my_list = [10, 20, 30]

>>> # Get an iterator from the list
>>> my_iterator = iter(my_list)
>>> print(type(my_iterator))
<class 'list_iterator'>

>>> # Get the next item
>>> print(next(my_iterator))
10
>>> print(next(my_iterator))
20
>>> print(next(my_iterator))
30

>>> # No more items, calling next() again raises StopIteration
>>> print(next(my_iterator))
StopIteration
```

5.3 What are Generators?

Generators are a special type of iterator. They are functions that allow you to declare a function that behaves like an iterator, i.e., it can be iterated over.

- **Key Feature: The `yield` keyword:**
    - A function becomes a generator function if it contains at least one `yield` statement.
    - When a `yield` statement is encountered, the generator *pauses* its execution, saves all its local state (variables, instruction pointer), and yields the value to the caller.
    - When `next()` is called on the generator again, it resumes execution from where it left off.
    - This "pause-and-resume" behavior makes generators incredibly memory-efficient.
- **`yield` vs. `return`:**
    - `return` terminates the function and returns a value.
    - `yield` pauses the function, yields a value, and allows the function to resume later from that point. A generator function can `yield` multiple times.

5.4 Creating Generators (using `yield`)

1. Simple Generator Function:

Python
```
# my_generator.py
```

```
def count_up_to(max_num):
    """A simple generator that counts up to a max number."""
    n = 1
    while n <= max_num:
        print(f"Yielding {n}...")
        yield n  # Pause here and return n
        n += 1

# Create a generator object (execution doesn't start yet)
counter_gen = count_up_to(3)
print(type(counter_gen)) # Output: <class 'generator'>

# Iterate through the generator
print("First next():")
print(next(counter_gen)) # Resumes, prints "Yielding 1...", returns 1

print("Second next():")
print(next(counter_gen)) # Resumes, prints "Yielding 2...", returns 2

print("Third next():")
print(next(counter_gen)) # Resumes, prints "Yielding 3...", returns 3

print("Fourth next() (will raise StopIteration):")
# print(next(counter_gen)) # This line would cause StopIteration

print("\n--- Looping through the generator (common usage) ---")
# Once exhausted, a generator cannot be reused. Create a new one.
new_counter_gen = count_up_to(5)
for num in new_counter_gen:
    print(f"Received {num}")
```

Interaction:

```
<class 'generator'>
First next():
Yielding 1...
1
Second next():
Yielding 2...
2
Third next():
Yielding 3...
3
Fourth next() (will raise StopIteration):

--- Looping through the generator (common usage) ---
Yielding 1...
Received 1
Yielding 2...
Received 2
Yielding 3...
Received 3
Yielding 4...
Received 4
Yielding 5...
Received 5
```

2. Fibonacci Sequence Generator:

Python
```
def fibonacci_sequence(limit):
    """Generates Fibonacci numbers up to a certain limit."""
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

print("Fibonacci sequence up to 10:")
for num in fibonacci_sequence(10):
    print(num)
# Output:
# 0
# 1
# 1
# 2
# 3
# 5
# 8
```

5.5 Generator Expressions

Generator expressions are similar to list comprehensions but use parentheses `()` instead of square brackets `[]`. They create a generator object without building the entire list in memory.

Syntax: **`(expression for item in iterable if condition)`**

Python
```
>>> # List comprehension (creates list in memory)
>>> list_of_squares = [x * x for x in range(1000000)] # Large list

>>> # Generator expression (creates generator object)
>>> generator_of_squares = (x * x for x in range(1000000)) # Small object
>>> print(type(generator_of_squares))
<class 'generator'>

>>> # Access elements one by one
>>> print(next(generator_of_squares))
0
>>> print(next(generator_of_squares))
1
>>> # ... or iterate over it
>>> for _ in range(5): # Take first 5 elements
...     print(next(generator_of_squares))
4
9
16
25
```

Generator expressions are extremely useful for performance-critical applications, especially with very large datasets, as they consume less memory.

5.6 When to Use Iterators/Generators

- **Large Datasets:** When you're processing files that are too large to fit into memory, or working with database query results row by row.

- **Infinite Sequences:** For sequences that conceptually never end (e.g., prime numbers, Fibonacci sequence), generators allow you to generate them on demand without running out of memory.
- **Performance:** When you only need to iterate over data once, or when memory is a concern.
- **Pipelining Operations:** You can chain multiple generator expressions or `map`/`filter` calls together, with each step processing data lazily.

Chapter 6: Coding Challenges for Functional Programming

Let's put these functional programming concepts into practice!

Challenge 1: Custom Sorting with Lambda

Goal: Sort a list of strings based on their length, and then by alphabetical order if lengths are the same.

Concepts Covered: `sorted()`, `lambda` functions, string `len()`.

Requirements:

1. Create a list of strings (e.g., `words = ["banana", "apple", "grape", "kiwi", "date"]`).
2. Use the `sorted()` function with a `lambda` function as its `key` to sort the words.
3. The primary sorting criterion should be the **length of the word** (ascending).
4. The secondary sorting criterion (for words of the same length) should be **alphabetical order** (ascending).
5. Print the sorted list.

Example Output:

```
Original words: ['banana', 'apple', 'grape', 'kiwi', 'date']
Sorted words: ['kiwi', 'date', 'apple', 'grape', 'banana']
```

(Explanation: 'kiwi' (4), 'date' (4) - kiwi comes before date alphabetically. 'apple' (5), 'grape' (5), 'banana' (6) - apple before grape, grape before banana alphabetically.)

Challenge 2: Data Cleaning with `filter()`

Goal: Filter a list of user inputs to remove invalid entries (empty strings, strings containing only whitespace, or non-numeric strings if expecting numbers).

Concepts Covered: `filter()`, `lambda` functions, string methods (`strip()`, `isdigit()`), conditional logic.

Requirements:

1. Start with a list of raw user inputs: `raw_inputs = ["10", " ", "25", "abc", "", "30", "  ", "xyz", "5"]`
2. Use `filter()` and a `lambda` function to create a new list containing only valid numeric strings. A string is valid if:
   - It is not empty after stripping whitespace (`.strip()`).
   - It consists only of digits (`.isdigit()`).
3. Convert the filtered strings to integers using `map()`.
4. Print the cleaned list of integers.

Example Output:

```
Original inputs: ['10', '  ', '25', 'abc', '', '30', '   ', 'xyz', '5']
Cleaned numbers: [10, 25, 30, 5]
```

Challenge 3: Product Price Calculation with `map()`

Goal: Given a list of product dictionaries, calculate the total price for each product (quantity * price_per_unit).

Concepts Covered: `map()`, `lambda` functions, dictionaries, accessing dictionary values.

Requirements:

1. Define a list of product dictionaries:

   Python

   ```
   products = [
       {"name": "Laptop", "quantity": 2, "price_per_unit": 1200},
       {"name": "Mouse", "quantity": 5, "price_per_unit": 25},
       {"name": "Keyboard", "quantity": 3, "price_per_unit": 75}
   ]
   ```

2. Use `map()` and a `lambda` function to create a new list of strings, where each string represents the total cost for that product in the format: `"Product Name: $Total_Cost"`.
3. Print the list of formatted total costs.

Example Output:

```
['Laptop: $2400', 'Mouse: $125', 'Keyboard: $225']
```

Challenge 4: Countdown Generator

Goal: Create a generator function that yields numbers in a countdown sequence from a given start number down to 1.

Concepts Covered: Generator functions, `yield` keyword, `while` loop.

Requirements:

4. Define a generator function `countdown(start_num)`.
5. Inside the function, use a `while` loop to yield numbers, starting from `start_num` down to 1.
6. Demonstrate its use by iterating over the generator and printing each yielded number.
7. Also, show how to manually get `next()` values a few times.

Example Interaction:

```
Counting down from 5:
5
4
3
2
1

Manual countdown from 3:
Next: 3
Next: 2
Next: 1
```

Challenge 5: Chaining Functional Operations (Map, Filter, Lambda)

Goal: Given a list of numbers, first filter out all odd numbers, then square the remaining even numbers, and finally sum them up.

Concepts Covered: `filter()`, `map()`, `lambda` functions, `sum()`.

Requirements:

1. Start with a list of numbers: `data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
2. Use `filter()` with a `lambda` to get only the even numbers.
3. Chain the result to `map()` with a `lambda` to square each of these even numbers.
4. Use the `sum()` built-in function to calculate the total sum of the squared even numbers.
5. Print the final sum.

Example Output:

```
Original data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Sum of squared even numbers: 220
```

(Calculation: Even numbers are [2, 4, 6, 8, 10]. Squared: [4, 16, 36, 64, 100]. Sum: 4+16+36+64+100 = 220)