Python Programming
Unit 02 – Lecture 05 Notes
Advanced Functions and Functional Patterns

Tofik Ali

February 14, 2026

Contents

1 Lecture Overview

This lecture moves from basic function syntax to advanced usage patterns that appear in real codebases:

- passing collections safely (mutation vs copying),
- recursion (and how to avoid infinite recursion),
- functional tools: `map`, `filter`, and `lambda`,
- closures (functions defined inside functions), and
- pure functions vs side effects.

2 Passing Collections: Mutation and Side Effects

Lists and dictionaries are mutable. When you pass them into a function, you pass *a reference to the same object*. That means the function can change the caller-visible object.

2.1 In-place Mutation (Sometimes Useful)

```
def add_item(lst, item):
    lst.append(item)

data = [1, 2]
add_item(data, 99)
print(data) # [1, 2, 99]
```

This can be efficient, but it is also a side effect. If a function mutates its inputs, it should be clearly documented (and ideally reflected in the name).

2.2 Avoiding Unintended Mutation (Copying)

If you want a function that behaves like "input in, output out" without changing input, copy first:

```
def sort_copy(items):
    tmp = items.copy()
    tmp.sort()
    return tmp
```

This pattern is a stepping stone to writing **pure functions**.

3 Recursion

3.1 What Is Recursion?

A recursive function calls itself to solve a smaller version of the same problem. Two required parts:

- **Base case:** stops recursion.
- **Progress step:** reduces the problem size each call.

3.2 Example: Factorial

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)
```

If the base case is missing or never reached, Python raises a `RecursionError` after hitting the recursion depth limit.

3.3 When to Avoid Recursion in Python

Recursion is sometimes the clearest solution (trees, divide-and-conquer), but for simple loops Python iteration is usually:

- faster (less call overhead),
- safer (no recursion depth risk), and
- easier to debug.

4 Functional Tools: `map`, `filter`, `lambda`

4.1 `map`: Transform Each Item

`map(f, iterable)` applies `f` to each item.

```
nums = [1, 2, 3, 4]
squares = list(map(lambda x: x * x, nums))
```

4.2 `filter`: Keep Items That Match a Condition

`filter(pred, iterable)` keeps items where `pred(item)` is true.

```
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
```

4.3 Pitfall: Iterators in Python 3

In Python 3, `map` and `filter` return **iterators**, not lists. That is why we wrap them in `list(...)` when printing or reusing.

4.4 `lambda`: Small Anonymous Functions

`lambda` is useful for short functions used once, often as a `key` for sorting or as a mapping function. Example (sorting by length):

```
words = ["pear", "banana", "fig"]
print(sorted(words, key=lambda w: len(w)))
```

If the lambda becomes complex, define a normal named function instead.

4.5 Inner Functions and Closures

An inner function can capture variables from the outer function.

```
def make_multiplier(k):
    def mult(x):
        return x * k
    return mult

triple = make_multiplier(3)
print(triple(10)) # 30
```

This is a closure: `mult` "remembers" `k`.

5 Pure Functions vs Side Effects

Pure function:

- same input always gives same output, and
- no changes to external state (no mutation of input, no global writes).

Pure functions are easier to test and reason about. Side effects are sometimes necessary (writing a file, printing, updating a database), but should be kept controlled.

6 Demo Walkthrough: Cleaning Data with `map/filter`

Script: `demo/functional_tools.py`

6.1 What the Demo Is Teaching

- filter invalid values before processing,
- map a transformation across data,
- compare recursion vs iteration for summation.

6.2 Suggested Run Steps

```
python demo/functional_tools.py
```

Extension idea: rewrite the `map/filter` part using list comprehensions and compare which version reads better.

7 Quiz and Solutions

7.1 Checkpoint 1 (Base Case)

Question. Identify the base case in a recursive function. Why is it necessary?

Solution. The base case is the condition that stops recursion (e.g., `if n == 1: return 1` in factorial). Without it, the function would call itself forever (until Python raises `RecursionError`).

7.2 Checkpoint 2 (Replace Loop with `map`)

Question. Replace this loop with `map`:

```
nums = [1,2,3,4,5]
squares = []
for n in nums:
    squares.append(n*n)
```

Solution.

```
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x * x, nums))
```

7.3 Think-Pair-Share (Avoiding Recursion)

Prompt. When should you avoid recursion in Python? Provide one reason and one example.

Sample response. Avoid recursion for simple counting/accumulation loops because recursion is slower and can hit recursion depth limits. Example: summing a list is clearer and safer with a `for` loop than with recursion.

7.4 Exit Question (Mutation Visibility)

Question. If a function appends to a list passed as an argument, does the caller see the change? Why?

Solution. Yes. The function receives a reference to the same list object, so mutating it (append/remove) changes the object the caller holds.

8 Practice Exercises (With Solutions)

8.1 Exercise 1: Pure vs In-place

Task. Write two functions: one that sorts a list in place, and one that returns a sorted copy.

Solution.

```
def sort_in_place(items):
    items.sort()
    return items

def sorted_copy(items):
    return sorted(items)
```

8.2 Exercise 2: Recursive Sum

Task. Write a recursive function `sum_list(nums)`.

Solution.

```
def sum_list(nums):
    if not nums:
        return 0
    return nums[0] + sum_list(nums[1:])
```

Note: slicing creates new lists; for very large lists, an iterative approach is more efficient.

8.3 Exercise 3: Rewrite `map`/`filter` as a Comprehension

Task. Given `nums`, build a list of squares of even numbers.

Solution.

```
nums = [1, 2, 3, 4, 5]
even_squares = [n*n for n in nums if n % 2 == 0]
```

8.4 Exercise 4: Closure Practice

Task. Use `make_multiplier` to create a function that multiplies by 5.

Solution.

```
five_x = make_multiplier(5)
print(five_x(7)) # 35
```

9 Further Reading

- `https://docs.python.org/3/library/functions.html#map`
- `https://docs.python.org/3/library/functions.html#filter`