Intermediate Python Programming – Lesson 5

Facilitated by Kent State University
Topic: Functional Programming in Python
Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Understand the principles behind functional programming
- Use `map()`, `filter()`, `reduce()`, and `lambda` functions effectively
- Compare functional tools with list comprehensions

Lesson 5: Functional Programming in Python

I. Introduction to Functional Programming (10 minutes)

Functional programming is a programming paradigm where computation is treated as the evaluation of mathematical functions and avoids changing state or mutable data.

Key principles:

- **Immutability**: Data is not changed; instead, new data is returned
- **Pure functions**: Functions that always return the same output for the same input, without side effects
- **Higher-order functions**: Functions that can take other functions as arguments or return them as results

Advantages of Functional Programming:

- **Fewer side effects**: Functions are easier to test and debug when they don't depend on or modify outside state.
- **Predictability**: Pure functions always return the same result for the same inputs.
- **Easier parallelization**: Because functions don't modify shared state, they can be run safely in parallel.
- **Improved readability and modularity**: Code is structured around small, reusable components.
- **Concise transformations**: Functional tools like `map`, `filter`, and `reduce` express intent clearly and avoid verbose loops.

Python isn't a purely functional language, but it supports functional constructs that integrate well with its other styles.

Multiple-Choice Question

Which of the following is a core idea of functional programming?

A. Using for-loops to modify lists directly
B. Avoiding functions entirely
C. Avoiding side effects and using pure functions
D. Relying on object-oriented inheritance

Answer: C. Avoiding side effects and using pure functions

Short Answer Question

What is a pure function?

Expected Answer: A function that returns the same output for the same input and does not modify external state.

II. Using `map()`, `filter()`, `reduce()`, and `lambda` (20 minutes)

Python includes several built-in functions that embody functional programming concepts:

`map(function, iterable)`

Applies a function to every item in the iterable.

```
numbers = [1, 2, 3, 4]
squares = list(map(lambda x: x**2, numbers))
print(squares)  # Output: [1, 4, 9, 16]
```

`filter(function, iterable)`

Filters elements for which the function returns True.

```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Output: [2, 4]
```

`reduce(function, iterable)` (from `functools`)

Reduces the iterable to a single value using a function.

```
from functools import reduce
product = reduce(lambda x, y: x * y, numbers)
print(product)  # Output: 24
```

`lambda` expressions

Anonymous inline functions, used for short, throwaway functions.

```
add = lambda x, y: x + y
print(add(3, 4))  # Output: 7
```

Exercise 1: Using `map`, `filter`, and `lambda`

Given a list of names:

```
names = ["alice", "BOB", "ChArLiE"]
```

Use `map()` and `lambda` to convert all names to lowercase.

Answer:

```
lower_names = list(map(lambda name: name.lower(), names))
print(lower_names)  # Output: ['alice', 'bob', 'charlie']
```

III. List Comprehensions vs. Functional Tools (15 minutes)

List comprehensions can often do the job of `map` and `filter`, and they are generally more readable for Python developers.

Example: Squaring numbers

Using `map()`:

```
squares = list(map(lambda x: x**2, range(5)))
```

Using list comprehension:

```
squares = [x**2 for x in range(5)]
```

Example: Filtering

Using `filter()`:

```
evens = list(filter(lambda x: x % 2 == 0, range(10)))
```

Using list comprehension:

```
evens = [x for x in range(10) if x % 2 == 0]
```

Discussion:

- `map()` and `filter()` can be combined with `lambda` to produce compact code
- Comprehensions are often clearer and more idiomatic in Python
- `reduce()` is powerful but often better replaced by loops or comprehensions unless the reduction is conceptually clean (e.g., sum, product)

Exercise 2:

Convert this comprehension to use `map()`:

```
words = ["Python", "is", "fun"]
lengths = [len(word) for word in words]
```

Answer:

```
lengths = list(map(len, words))
```

IV. Recap and Q&A (10 minutes)

We explored the basics of functional programming in Python:

- Pure functions and immutability
- How to use `map()`, `filter()`, `reduce()`, and `lambda`
- When to use comprehensions instead
- The advantages of a functional style for writing compact, reliable, and testable code

Final Exercise:

Use `filter()` to extract all positive numbers from the list:

```
nums = [-5, 3, 0, -2, 8, 1]
```

Expected Answer:

```
positives = list(filter(lambda x: x > 0, nums))
print(positives)  # Output: [3, 8, 1]
```
