ICC/Programming — Week 11 Exercises

4 May 2026

Goal. In these exercises you will practise the three pillars of functional programming in Python introduced in today's lecture: list comprehensions, higher-order functions, and lambda expressions. You will start with small, self-contained examples and finish by combining all three ideas to write concise, expressive code.

These exercises are self-contained. No additional libraries are required beyond the Python standard library.

Setup. Create a new file **`functional.py`** and work through each exercise by adding code to the bottom of the file. Run it at any point with:

```
python3 functional.py
```

Exercise 1. List Comprehensions

A list comprehension builds a new list by applying an expression to each element of an existing sequence, optionally filtering elements with **`if`**.

(a) **Basic transformation.** The loop below converts a list of temperatures from Celsius to Fahrenheit. Rewrite it as a single list comprehension:

```python
celsius = [0, 20, 37, 100]
fahrenheit = []
for c in celsius:
    fahrenheit.append(c * 9 / 5 + 32)
print(fahrenheit)  # [32.0, 68.0, 98.6, 212.0]

# Your one-line version:
fahrenheit = [... for c in celsius]
print(fahrenheit)
```

(b) **Filtering.** Using a comprehension with an **`if`** clause, extract only the words from the list that have more than four characters:

```python
words = ["cat", "elephant", "dog", "butterfly", "ox", "penguin"]

# Expected: ['elephant', 'butterfly', 'penguin']
long_words = [... for word in words if ...]
print(long_words)
```

(c) **Nested comprehension.** The multiplication table below is built with nested loops. Replace it with a nested list comprehension (a comprehension that produces a list of lists):

```python
table = []
for x in range(1, 6):
    row = []
    for y in range(1, 6):
        row.append(x * y)
    table.append(row)

for row in table:
    print(row)
# [1, 2, 3, 4, 5]
# [2, 4, 6, 8, 10]
# ...

# Your version:
table = [[... for y in range(1, 6)] for x in range(1, 6)]
```

(d) **Dictionary comprehension.** Comprehensions also work with dictionaries using **`{key: value for ...}`**. Build a dictionary that maps each word to its length:

```python
words = ["apple", "banana", "kiwi", "strawberry"]

# Expected: {'apple': 5, 'banana': 6, 'kiwi': 4, 'strawberry': 10}
lengths = {word: ... for word in words}
print(lengths)
```

(e) **Question.** Consider the following comprehension:

```
result = [x * y for x in range(1, 4) for y in range(1, 4) if x != y]
print(result)
```

Before running the code, predict what **`result`** will contain. Then run it and verify. How many elements does the list have, and why?

Exercise 2. Higher-Order Functions

A higher-order function is one that takes another function as an argument or returns a function as its result. Python's built-in **`map`**, **`filter`**, and **`sorted`** are common examples.

(a) **Functions as values.** In Python, a function is an object like any other. Run the following and make sure you understand each line:

```
def double(x: int) -> int:
    return x * 2

def triple(x: int) -> int:
    return x * 3

# Store functions in a list and call them in a loop
operations = [double, triple]
for f in operations:
    print(f(10))   # prints 20, then 30
```

What is the type of the variable **`f`** inside the loop?

(b) **Writing a higher-order function.** Complete the function **`apply_to_all`** below. It receives a function **`f`** and a list, and returns a new list containing the result of applying **`f`** to each element:

```
from typing import Callable

def apply_to_all(f: Callable[[int], int], values: list[int]) -> list[int]:
    result = []
    for v in values:
        result.append(...)   # fill this in
    return result

print(apply_to_all(double, [1, 2, 3, 4]))   # [2, 4, 6, 8]
print(apply_to_all(triple, [1, 2, 3, 4]))   # [3, 6, 9, 12]
```

Notice that **`apply_to_all`** does exactly what the built-in **`map`** does — **`list(map(double, [1, 2, 3, 4]))`** gives the same result.

(c) **Returning a function.** The function **`make_multiplier`** below takes a number **`n`** and returns a new function that multiplies its argument by **`n`**. Fill in the body:

```
def make_multiplier(n: int) -> Callable[[int], int]:
    def multiply(x: int) -> int:
        return ...   # fill this in
    return multiply

times5 = make_multiplier(5)
times7 = make_multiplier(7)

print(times5(3))    # 15
print(times7(3))    # 21
print(times5(times7(2)))   # what does this print?
```

(d) **Sorting with a key.** The built-in **`sorted`** accepts a **`key`** argument — a function that extracts a comparison value from each element. Sort the list of words first by length (shortest first), then sort it by length in descending order:

```
words = ["banana", "fig", "apple", "kiwi", "strawberry", "plum"]

by_length = sorted(words, key=...)
print(by_length)   # ['fig', 'kiwi', 'plum', 'apple', 'banana', 'strawberry']

by_length_desc = sorted(words, key=..., reverse=True)
print(by_length_desc)
```

(e) **Question.** Consider this function:

```
def apply_twice(f: Callable[[int], int], x: int) -> int:
    return f(f(x))

print(apply_twice(double, 3))
```

What does **`apply_twice(double, 3)`** return? What would **`apply_twice(apply_twice(double, ...), 1)`** mean — is this even valid Python? Try it and explain the result.

Exercise 3. Lambda Expressions

A lambda is an anonymous, single-expression function. It is useful when you need a short function as an argument and do not want to give it a name with **`def`**.

(a) **Basic syntax.** Run the following and make sure the output matches your expectation:

```
square    = lambda x: x ** 2
add       = lambda x, y: x + y
constant  = lambda: 42

print(square(5))       # 25
print(add(3, 7))       # 10
print(constant())      # 42
```

Each lambda expression is equivalent to a **`def`** definition. Rewrite **`square`** using a normal **`def`** and verify it produces the same result.

(b) **Lambda as an argument.** Rewrite the **`sorted`** calls from Exercise 2(d) using lambdas instead of a separately defined function:

```
words = ["banana", "fig", "apple", "kiwi", "strawberry", "plum"]

by_length      = sorted(words, key=lambda w: ...)
by_length_desc = sorted(words, key=lambda w: ..., reverse=True)
print(by_length)
print(by_length_desc)
```

(c) **Lambda with `apply_twice`.** Use the **`apply_twice`** function from Exercise 2 together with a lambda to compute the following without defining any named helper function:

```
# Add 3 to a number twice (i.e. add 6 in total)
result = apply_twice(lambda x: ..., 10)
print(result)   # 16

# Square a number twice (i.e. raise to the power 4)
result2 = apply_twice(lambda x: ..., 3)
print(result2)  # 81
```

(d) **Sorting a list of tuples.** Each tuple below represents a student as **`(name, grade)`**. Sort the list by grade in ascending order using a lambda key:

```
students = [
    ("Alice", 5.5),
    ("Bob",   4.0),
    ("Carol", 5.75),
    ("Dave",  4.5),
]

by_grade = sorted(students, key=lambda s: ...)
for name, grade in by_grade:
    print(f"{name}: {grade}")
# Bob: 4.0
# Dave: 4.5
# Alice: 5.5
# Carol: 5.75
```

(e) **What lambdas cannot do.** Try to predict whether each snippet below is valid Python. Then run each one and explain the error if it fails:

```python
# Snippet 1
f = lambda x: x + 1
print(f(4))

# Snippet 2
g = lambda x:
        x + 1
print(g(4))

# Snippet 3
h = lambda x: y = x + 1; return y
print(h(4))
```

Exercise 4. Putting It All Together

In this final exercise you will combine comprehensions, higher-order functions, and lambdas to solve a small data-processing task.

(a) **Setup.** Copy the dataset and helper function below into **`functional.py`**:

```python
transactions = [
    {"id": 1, "amount": 120.0,  "category": "food"},
    {"id": 2, "amount":  45.5,  "category": "transport"},
    {"id": 3, "amount": 200.0,  "category": "food"},
    {"id": 4, "amount":  15.0,  "category": "transport"},
    {"id": 5, "amount": 340.0,  "category": "electronics"},
    {"id": 6, "amount":  80.0,  "category": "food"},
    {"id": 7, "amount": 500.0,  "category": "electronics"},
    {"id": 8, "amount":  22.5,  "category": "transport"},
]
```

(b) **Filter.** Using a list comprehension, extract only the food transactions:

```python
food = [t for t in transactions if ...]
print(food)   # three dicts with category "food"
```

(c) **Transform.** Using a list comprehension, build a list of **`(id, amount)`** tuples for every transaction whose amount exceeds 100:

```python
expensive = [(t["id"], t["amount"]) for t in transactions if ...]
print(expensive)   # [(1, 120.0), (3, 200.0), (5, 340.0), (7, 500.0)]
```

(d) **Sort.** Sort all transactions by amount in descending order using **`sorted`** and a lambda key:

```python
by_amount = sorted(transactions, key=lambda t: ..., reverse=True)
for t in by_amount:
    print(t["id"], t["amount"])
```

(e) **Group totals.** Using a dictionary comprehension, compute the total amount spent per category. *Hint:* use **`sum()`** with a generator expression inside.

```python
categories = {"food", "transport", "electronics"}

totals = {
    cat: sum(t["amount"] for t in transactions if t["category"] == cat)
    for cat in categories
}
print(totals)
# {'food': 400.0, 'transport': 83.0, 'electronics': 840.0}
```

(f) **Pipeline.** Combine everything into a single expression: start from **`transactions`**, keep only those with amount above 50, sort them by amount descending, and extract just the **`id`** of each. Write this as a single line using a comprehension and **`sorted`**:

```python
result = [t["id"] for t in sorted(
    [...],          # filter here
    key=lambda t: ...,
    reverse=True
)]
print(result)   # [7, 5, 3, 6, 1]
```