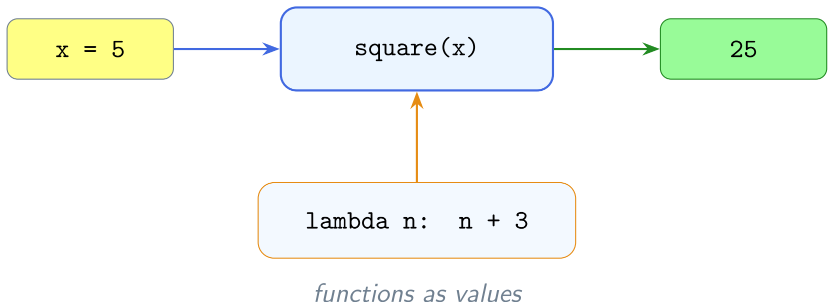
Programming Fundamentals
Lab 05

Functions, Scope, & Lambdas



functions as values

Comp 102 — Forman Christian University

Spring 2026

Estimated Time: ∼**3 hours** — Based on Lectures 9 & 10

How to Use This Lab

This lab is a **walkthrough tutorial**—it is designed to guide you through learning, not to test you. Work through each section at your own pace:

- **Predict** — Write down what you think will happen *before* touching the computer.
- **Type** — Enter the code in Thonny's **Code Editor** (top pane) and press **Run** (or **F5**).
- **Verify** — Compare your prediction with the actual result. If they differ, figure out *why*.
- Exercises are graded by difficulty:
  - ★ Easy — Immediate practice. Follow the pattern you just saw.
  - ★★ Medium — Requires some thinking. Combines concepts.
  - ★★★ Hard — Stretch goal. It is OK to need help!
- **Ask for help** whenever you are stuck for more than 5 minutes.

Prerequisites

This lab assumes you completed **Lab 04**. You should be comfortable with variables, `for`/`while` loops, counters/accumulators, and simple string iteration.

1 Functions as Black Boxes (∼30 min)

Ref: Lecture 9 — Abstraction, Decomposition, Functions

Functions let us hide details behind a name. The caller only needs to know the inputs and the promised return value (the docstring/spec). This section focuses on parameters vs. arguments and keeping the contract honest.

1. ★ Easy Predict the output, then type it in the **Code Editor** and run:

```
def cube(n):
    """Return n cubed."""
    return n * n * n

print(cube(2))
print(cube(5))
```

Your prediction:

The docstring is a promise about the return value. Does the function ever print inside it?

2. ★ Easy Predict the output, then identify the parameters and arguments:

```
def repeat(word, times):
    """Print word, times times."""
    for _ in range(times):
        print(word)

repeat("hi", 3)
```

Your prediction:

Parameters: ______________________ Arguments in the call: ______________________

Parameters are the names in the function header; arguments are the actual values in the call.

3. ★★ Medium Predict the output, then check whether the behavior matches the docstring:

```
def is_even(n):
    """Return True if n is even."""
    return n % 2 == 1

print(is_even(4))
print(is_even(5))
```

Your prediction:

Compare the docstring to the return expression. One of them is lying.

4. ★★ Medium The code works, but the docstring is misleading. Rewrite the docstring so it matches the code. Then run it to confirm behavior:

```
def clamp(x, low, high):
    return max(low, min(x, high))

print(clamp(5, 0, 10))
print(clamp(-3, 0, 10))
```

Rewrite the docstring here:

Describe the return value in one sentence.

5. ★★★ Hard Write a function that follows this spec, then test it with the calls below:

```
def shipping_cost(weight):
    """
    Return the shipping cost in rupees.
    Rules:
      - 200 if weight <= 1.0
      - 350 if 1.0 < weight <= 3.0
      - 500 otherwise
    """
    # TODO: your code here
    pass

print(shipping_cost(0.8))
print(shipping_cost(2.5))
print(shipping_cost(4.2))
```

Write your code here (replace the TODO and pass):

The docstring is your contract. Use it to decide the conditions.

2 Designing Helper Functions with Loops (~30 min)

Ref: Lecture 9 — Abstraction & Decomposition

Helpers let you keep the main function short and readable. When a loop gets long, extract the repeating logic into a helper that returns a value.

6. ★ Easy

Paper-First

Before you code, sketch the helper function on paper: inputs → loop steps → return value. Then type it.

Predict the output, then type it in the **Code Editor** and run:

```
def count_stars(text):
    total = 0
    for ch in text:
        if ch == "*":
            total += 1
    return total

print(count_stars("a*b*c*"))
print(count_stars("***"))
```

Your prediction:

Trace the loop on paper first. How many times does the condition pass?

7. ★ Easy Predict the output, then run. (Helper function with a loop.)

```
def sum_to(n):
    total = 0
    for i in range(1, n + 1):
        total += i
    return total

def average_to(n):
    return sum_to(n) / n

print(average_to(4))
```

Your prediction:

Compute the helper result first, then divide.

8. ★★ Medium Fill in the blank to complete the helper. Predict the output, then type and run:

```
def count_digits(text):
    total = 0
    for ch in text:
        if ch.isdigit():
            total += 1
    return total
```

```
7
8 def has_enough_digits(text, needed):
9     return count_digits(text) ____ needed
10
11 print(has_enough_digits("a1b2c3", 3))
12 print(has_enough_digits("room7", 2))
```

Your prediction and the missing operator:

The helper returns a number; the main function compares it to `needed`*.*

9. ★★ Medium Write the helper function so the program works. Then run the tests.

```
1 def sum_of_squares(n):
2     # TODO: return 1^2 + 2^2 + ... + n^2
3     pass
4
5 def average_square(n):
6     return sum_of_squares(n) / n
7
8 print(average_square(3))
9 print(average_square(5))
```

Write your helper code here:

Use a loop inside the helper; keep the main function short.

10. ★★★ Hard Design two helpers to keep the main program clean. Requirements:

- `count_vowels(text)` returns how many vowels appear in `text`.
- `vowel_ratio(text)` returns vowels divided by total characters.

Test with:

```
1 print(vowel_ratio("banana"))
2 print(vowel_ratio("rhythm"))
```

Write both helper functions here:

Plan on paper: what does each helper return? Then use one inside the other.

3 Default Parameters & Common Function Mistakes (∼20 min)

Ref: Lecture 9 — Default Parameters

Default parameters are used when an argument is missing. Also, keep a clear mental model of the difference between printing and returning.

11. ★ Easy Predict the output, then run:

```
def greet(name, times=2):
    for _ in range(times):
        print("Hello", name)

greet("Ayesha")
greet("Bilal", 3)
```

Your prediction:

If an argument is missing, the default parameter is used.

12. ★★ Medium Predict the output, then run. Watch the difference between printing and returning:

```
def add(a, b=0):
    print(a + b)

result = add(5, 7)
print("result =", result)
```

Your prediction:

A function that only prints usually returns `None`*.*

13. ★★ Medium Bug hunt (intentionally broken): fix the mistake, then run.

```
def wrap(text, left="[", right="]"):
    """Return text wrapped by left and right."""
    print(left + text + right)

print(wrap("hi"))
```

What is the mistake? Rewrite the function header/body:

The docstring says "Return". The current code only prints.

14. ★★★ Hard Write a function with defaults and a docstring:

```
def banner(message, width=20, fill="-"):
    """
    Return a string of length width where message is
        centered.
    Use the fill character on both sides.
    """
    # TODO: your code here
    pass

print(banner("Sale"))
print(banner("Hi", 8, "*"))
```

Write your code here:

Start by computing how many fill characters are needed.

4 Environment Diagrams & Scope (~30 min)

Ref: Lecture 10 — Environment Diagrams

Environment diagrams show which names live in which frame. Pay close attention to what is local versus global, and when functions read global values.

15. ★ Easy Predict the output, then draw a simple environment diagram:

```
x = 10

def bump(x):
    x = x + 1
    return x

print(bump(5))
print(x)
```

Your prediction and diagram (show global and local x):

Parameters create local names. Does the global x change?

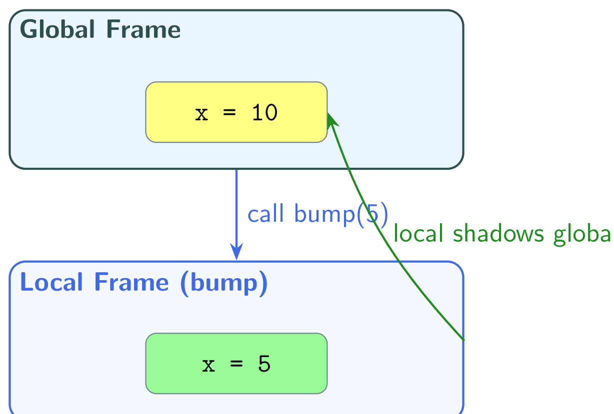


Figure: A local name can shadow a global name with the same spelling.

16. ★ Easy Predict the output, then run:

```
y = 3

def add_y(n):
    return n + y

print(add_y(4))
y = 10
print(add_y(4))
```

Your prediction:

A function uses the current global value at call time.

17. ★★ Medium Trace on paper first. Predict the output and draw the environments.

```
def outer(a):
    b = a + 2
    def inner(c):
        return a + b + c
    return inner(5)

print(outer(3))
```

Prediction and environment diagram:

Track where `a` and `b` live when `inner` runs.

18. ★★ Medium Predict the output, then run. Identify which variable is local.

```
count = 1

def inc():
    count = 0
    count = count + 1
    return count

print(inc())
print(count)
```

Your prediction:

Assignment inside a function creates a local variable unless declared global.

19. ★★★ Hard Draw the environment diagram and predict the output:

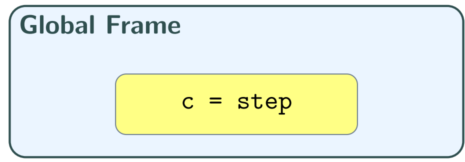
```
def make_counter(start):
    count = start
    def step():
        nonlocal count
        count += 1
        return count
    return step

c = make_counter(10)
```

```
10 print(c())
11 print(c())
```

Prediction and diagram (show the closure variable):

The inner function remembers `count` *even after* `make_counter` *ends.*



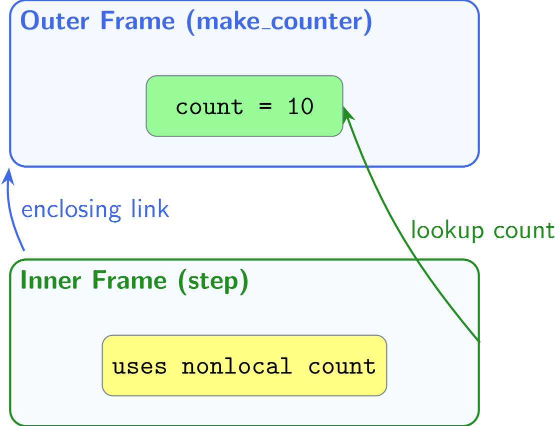


Figure: Closures keep a link to an enclosing frame for nonlocal names.

5 Functions as Data: Higher-Order Functions (∼25 min)

Ref: Lecture 10 — Higher Order Functions

Functions can be passed as arguments and returned from other functions. Focus on what each helper returns, not on how it is defined.

20. ★ Easy Predict the output, then run:

```
1 def apply_twice(f, x):
2     return f(f(x))
3
4 def add_one(n):
5     return n + 1
6
7 print(apply_twice(add_one, 3))
```

Your prediction:

The function is data: it can be passed in like a value.

21. ★★ Medium Predict the output, then run:

```
def keep_if(f, a, b):
    if f(a):
        return a
    else:
        return b

def is_long(word):
    return len(word) >= 5

print(keep_if(is_long, "kite", "stone"))
```

Your prediction:

Focus on what the helper returns, not how it prints.

22. ★★ Medium Predict the output, then run. (Closure practice.)

```
def make_multiplier(k):
    def times(n):
        return k * n
    return times

double = make_multiplier(2)
triple = make_multiplier(3)

print(double(7))
print(triple(7))
```

Your prediction:

Each returned function "remembers" its own `k`*.*

23. ★★★ Hard Write a higher-order function `compose(f, g)` that returns a new function. The returned function should compute `f(g(x))`. Test with:

```
def square(n):
    return n * n

def add_three(n):
    return n + 3

```

```
7 h = compose(square, add_three)
8 print(h(4))
```

Write your code here:

Return a function that takes one argument and applies two functions in order.

6 Anonymous Functions & Lambda Pitfalls (~15 min)

Ref: Lecture 10 — Lambda Functions

Lambdas are small anonymous functions. They are handy, but they can hide subtle bugs with variable binding. Predict carefully.

24. ★ Easy Predict the output, then run:

```
1 longest = max("ant", "elephant", "cat", "dog", key=lambda w
    : len(w))
2 print(longest)
```

Your prediction:

The key function returns the value used for sorting.

25. ★★ Medium Predict the output, then run. (Lambda pitfall: late binding.)

```
1 for i in range(3):
2     if i == 0:
3         f0 = lambda: i
4     elif i == 1:
5         f1 = lambda: i
6     else:
7         f2 = lambda: i
8
9 print(f0())
10 print(f1())
11 print(f2())
```

Your prediction:

All lambdas close over the same `i` *name. What is* `i` *after the loop?*

26. ★★ Medium Fix the pitfall by binding the value now. Predict the output, then run:

```
for i in range(3):
    if i == 0:
        f0 = lambda i=i: i
    elif i == 1:
        f1 = lambda i=i: i
    else:
        f2 = lambda i=i: i

print(f0())
print(f1())
print(f2())
```

Your prediction:

Default parameters can "freeze" a value inside a lambda.

27. ★★★ Hard Write a function that returns a lambda. Requirements:

- `make_offset(k)` returns a function that adds `k` to its input.
- Use a `lambda`, not a nested `def`.

Test with:

```
add5 = make_offset(5)
print(add5(10))
```

Write your code here:

A lambda is an expression that returns a function value.

7 Capstone Challenges (∼20 min)

These problems combine everything from this lab. Write your solutions in Thonny's code editor.

28. ★★ Medium **Capstone: "Receipt Builder"** Write functions to compute a receipt for three items with a default tax rate. Requirements:

1. `subtotal(a, b, c)` returns the sum of three item prices.
2. `total_with_tax(a, b, c, tax=0.05)` returns subtotal plus tax.

Sample run:

```
>>> print(total_with_tax(120, 80, 50))
262.5
```

Write your code here:

Build a helper first; then use the default parameter.

29. ★★ Medium **Capstone: “Name Filter”** Write a higher-order function `pick_name(test, a, b)` that returns `a` if `test(a)` is True, otherwise `b`. Then use a lambda to choose the longer name.

```
>>> print(pick_name(lambda s: len(s) >= 5, "Ali", "Ayesha"))
Ayesha
```

Write your code here:

Keep the function generic: it should work with any test.

30. ★★★ Hard **Capstone: “Discount Factory”** Create a closure that makes discount functions.

1. `make_discount(rate)` returns a function that applies the discount.
2. The returned function takes `price` and returns the new price.

Sample run:

```
>>> ten_off = make_discount(0.10)
>>> print(ten_off(500))
450.0
```

Write your code here:

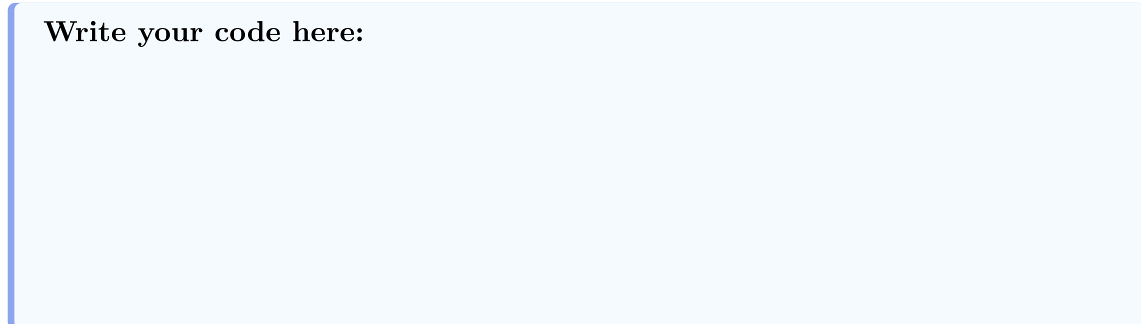
The inner function should "remember" `rate`.

31. ★★★ Hard **Capstone: "Score Summary"** Write a program with these functions:

1. `average(a, b, c, d)` returns the mean of four scores.
2. `summary(a, b, c, d, formatter)` returns a string using `formatter(avg)`.
3. Provide a `formatter` using a `lambda` that rounds to 1 decimal.

Sample run:

```
>>> print(summary(18, 22, 25, 19, lambda x: round(x, 1)))
Average: 21.0
```



Pass the formatter function as a value, just like a number.

Self-Assessment Checklist

Before you leave, check off what you can do:

- ☐ I can explain parameters vs. arguments using a simple function call.
- ☐ I can write a docstring that correctly describes a function's return value.
- ☐ I can design a helper function with a loop and use it inside another function.
- ☐ I can use default parameters and predict what happens when arguments are omitted.
- ☐ I can identify the difference between printing and returning in a function.
- ☐ I can draw a basic environment diagram with global and local frames.
- ☐ I can trace how closures keep access to nonlocal variables.
- ☐ I can pass a function as an argument to a higher-order function.
- ☐ I can create and use a lambda for simple one-line behavior.
- ☐ I can combine helpers, defaults, and higher-order functions in a small program.