Finger Exercises 10

Environment Diagrams, Lambda Functions

≈ 30 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Every program here is new — none are from the slides — so you have to *trace* them, not remember them. The skills being drilled: *which frame does a name live in, what happens to that frame, and what does it mean to hand a function to another function?*

How to do it. On **paper**, **no computer**, in one sitting. For every call, sketch its frame in the margin — a box with the parameters written inside — and cross it out when the call returns. If an item takes you more than a minute, skip it, finish the rest, then come back. Parts A–I are for *everyone*. Part J is a **stretch**: skip it freely, or attack it if A–I felt smooth. The Console Quests at the end are for tonight, *at* a computer. The answer key is on the last page — do not peek until you are done.

Predict first, then check. Write down what you think *before* you look anything up or type anything in — even for the Console Quests. A guess that turns out wrong teaches you far more than an answer you only watched the computer produce.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Every Python program starts with one frame, called the _____ Frame. Every time a function is *called*, Python makes a _____.

A2. Fill in the shape of a lambda: `lambda` _____ : _____

A3. A function is **higher-order** if it does either of two things: 1. _____ or 2. _____

A4. When a call starts, the new frame already contains the _____, each holding the matching _____. When the call ends, the frame is _____ and only the _____ goes back to the caller.

A5. When Python meets a name inside a function body it looks in the _____ frame first, then in the _____ Frame. If it is in neither, the result is a _____.

A6. Circle **True** or **False**:

(a) A `def` statement runs the code inside the function straight away. True / False

(b) A local variable is still there after its function has returned. True / False

(c) The body of a lambda may contain the word `return`. True / False

(d) A function can be stored in a variable, just like a number can. True / False

(e) Two calls to the same function share one frame. True / False

(f) Assigning to `x` inside a function changes a global `x` of the same name. True / False

(g) A lambda can take two parameters. True / False

(h) `apply(is_even, 10)` and `apply(is_even(), 10)` mean the same thing. True / False

(i) Calling a lambda creates a frame, just like calling a `def` function. True / False

Part B • What Does It Print?

TRACE ≈ 9 min

Write the output exactly, one value per line, in order. If a line crashes, write the error name instead.

B1. Version (b) has one extra line. Give the output of each.

(a)

```
x = 5
def f(y):
    return x + y
print(f(3))
```

Output: _____

(b)

```
x = 5
def f(y):
    x = 100
    return x + y
print(f(3))
print(x)
```

Output: _____ , _____

B2.

```
def bump(n):
    n = n + 1
    return n
a = 10
bump(a)
print(a)
```

Output: _____

B3. A parameter with a global's name; two calls, one function.

```
x = 1
def f(x):
    return x + 1
print(f(10), x)
```

Output: _____

```
def count():
    n = 0
    n += 1
    return n
print(count(), count())
```

Output: _____

B4. Names are looked up when the body *runs*, not when it is defined.

```
def show():
    return msg
msg = "hi"
print(show())
```

Output: _____

```
def a():
    t = 5
    return t
def b():
    return t
print(a())
print(b())
```

Output: _____

B5. Lambdas in three positions.

```
f = lambda x: x * 3
print(f(4))
print((lambda x, y: x * y)(3, 4))
g = lambda x: x % 2 == 0
print(g(7))
```

Output:

_____

_____

_____

B6. Two names for one function — and a name stolen.

```
def double(x):
    return x * 2
d = double
print(d(4))
print(double(d(1)))
```

Output: _____

```
def double(x):
    return x * 2
double = 5
print(double(2))
```

Output: _____

B7. A higher-order function, called twice with different lambdas.

```
def apply_twice(func, n):
    return func(func(n))
print(apply_twice(lambda x: x + 10, 1))
print(apply_twice(lambda s: s + "!", "hi"))
```

Output:

_____

_____

B8. A lambda that prints.

```
f = lambda x: print(x)
y = f(3)
print(y)
```

Output:

_____

_____

Part C • Frames and Higher-Order Functions

TRACE ≈ 6 min

C1.

```
def triple(n):
    return n * 3
a = triple(2)
b = triple(a)
```

`b` holds: _____

Frames created in total (count the Global Frame): _____

Most frames alive at any one moment: _____

C2.

```
def do_twice(n, fn):
    return fn(fn(n))
print(do_twice(2, lambda x: x * x))
```

Output: _____

Frames created in total: _____

Most alive at one moment: _____

C3. Using `apply` from the slides, fill in each result. Remember the range is `0` to `n` *inclusive*.

```
def apply(criteria, n):
    count = 0
    for i in range(n + 1):
        if criteria(i):
            count += 1
    return count
def is_even(x):
    return x % 2 == 0
```

`apply(is_even, 10)` → _____

`apply(lambda x: x > 5, 10)` → _____

`apply(lambda x: x % 3 == 0, 10)` → _____

`apply(is_even, 0)` → _____

`apply(lambda x: x < 0, 10)` → _____

C4. A function in a loop, supplied from outside.

```
1 def square(x):
2     return x * x
3 def sum_fn(fn, a, b):
4     t = 0
5     for i in range(a, b + 1):
6         t += fn(i)
7     return t
8 print(sum_fn(square, 1, 3))
9 print(sum_fn(lambda x: x, 1, 4))
```

Output:

_____

_____

How many `square` frames does line 8 create? _____

Part D • Which Frame Does It Live In?

CLASSIFY ≈ 3 min

Tick the frame that each name is created in. Then, for the moment just before `return b` runs, write the value stored under each name.

```
1 x = 10
2 def f(a):
3     b = a + x
4     return b
5 y = f(3)
6 z = f(y)
```

Name	Global	`f` frame	Value (2nd call)
`x`	☐	☐	_____
`f`	☐	☐	_____
`a`	☐	☐	_____
`b`	☐	☐	_____
`y`	☐	☐	_____
`z`	☐	☐	_____

Part E • Draw the Frames

TRACE ≈ 7 min

Fill in every box exactly as the slides draw it: one `name : value` row per name, a function's value written as `func`. Leave spare rows empty. If a frame does not exist at the marked moment, cross the whole box out.

E1. The program pauses at the comment marked `*`, inside the call, just before the `return` runs. Draw that moment on the left. Then draw the state after line 6 has finished, on the right.

```
1 base = 100
2 def add_tax(price):
3     tax = price * 0.1
4     # * pause here
5     return price + tax
6 cost = add_tax(50)
```

at *

Global Frame
_____ : _____
_____ : _____
_____ : _____

add_tax frame
_____ : _____
_____ : _____

after line 6

Global Frame
_____ : _____
_____ : _____
_____ : _____

add_tax frame
_____ : _____
_____ : _____

E2. A higher-order call. Draw the moment the *second* call to `sq` is executing its `return` line. Three frames are alive; one has already been destroyed.

```
1 def sq(x):
2     return x * x
3 def twice(fn, n):
4     return fn(fn(n))
5 r = twice(sq, 3)
```

Global Frame

_____ : _____
_____ : _____
_____ : _____

sq frame (2nd call)

_____ : _____

twice frame

_____ : _____
_____ : _____

The destroyed frame belonged to _____, held `x` : ____, and returned ____.

After line 5, `r` : _____

E3. Two names, one function object — then one name is taken away. Draw the Global Frame after the whole program has run. For a name that holds a function, draw an arrow to the function-object box instead of writing a value.

```
1 def greet(name):
2     return "Hi " + name
3 hello = greet
4 greet = 7
5 s = hello("Ali")
```

Global Frame

_____ : _____
_____ : _____
_____ : _____

function object

`greet(name)`
`  "Hi " + name`

Frames created by line 5: _____

Does `greet("Ali")` still work?

_____

Part F • Lambda or Not?

CLASSIFY ≈ 3 min

F1. Tick each line that Python accepts as a lambda. For the rejected ones, say why in three words.

(a) ☐ `lambda x:  x + 1`
(b) ☐ `lambda x, y:  x > y`
(c) ☐ `lambda x:  return x + 1`
(d) ☐ `lambda x x + 1`
(e) ☐ `lambda:  5`
(f) ☐ `lambda x:  y = x + 1`
(g) ☐ `lambda s:  s + "!"`
(h) ☐ `lambda x:  for i in range(x):  x`

F2. `def` or `lambda`? Tick the better choice.

(a) A one-line test passed to `apply` once, never used again. def ☐ lambda ☐
(b) A function that needs a loop and two `if`s. def ☐ lambda ☐
(c) A helper used in five places, which deserves a docstring. def ☐ lambda ☐
(d) `x * 3`, handed to `do_twice` on the spot. def ☐ lambda ☐

Part G • Translate

TRANSLATE ≈ 4 min

G1. Rewrite each `def` as a lambda stored under the same name, or write **cannot** if no lambda can do it.

```
def add(x, y):
    return x + y
```

add = _____

```
def last(s):
    return s[len(s) - 1]
```

last = _____

```
def count_a(s):
    c = 0
    for ch in s:
        if ch == "a":
            c += 1
    return c
```

count_a = _____

```
def is_big(n):
    if n > 100:
        return True
    else:
        return False
```

is_big = _____

G2. Now the other way. Write `def` versions of:

`half = lambda x: x / 2`

_____

_____

`both = lambda a, b: a and b`

_____

_____

G3. Complete the call so that it counts how many numbers from 0 to 20 end in the digit 7. Use `apply` from C3.

`print(apply(`_____`, _____`)) answer printed: _____

Part H • Spot the Bug

FIX ≈ 7 min

Each program has exactly one bug. Say what actually happens (wrong output, or the error name), then write the fix.

H1. Two broken lambdas — one bug each.

```
square = lambda x  x ** 2
half = lambda x: return x / 2
```

Fix 1: _____

Fix 2: _____

H2.

```
def add(x, y):
    print(x + y)
total = add(2, 3) + 1
```

Fix: _____

H3. Should print 6. Error: _____ Fix: _____

```
def is_even(x):
    return x % 2 == 0
print(apply(is_even(), 10))      # apply as in C3
```

H4. Should print 12. Error: _____ Fix: _____

```
def double(x):
    return x * 2
def apply_twice(func, n):
    return func(func(n))
print(apply_twice(double(3), 3))
```

H5. `apply(lambda x: x > 100, 10)` should print `0`. It prints _____. Fix: _____

```
def apply(criteria, n):
    count = 0
    for i in range(n + 1):
        if criteria:
            count += 1
    return count
```

H6. Should print `8`. It prints something strange: _____ Fix: _____

```
f = lambda x: x * 2
print(f)
```

H7. Should print `9`. Error: _____ Fix: _____

```
def apply_twice(func, n):
    return func(func(n))
print(apply_twice(lambda x, y: x + y, 3))
```

H8. Should print `5`. Error: _____ Fix: _____

```
def total():
    return n + 1
def setup():
    n = 4
setup()
print(total())
```

Part I • One Sentence Each

EXPLAIN ≈ 3 min

I1. Why does the `square` frame disappear once `square(4)` has finished?

_____

I2. Why can a `def` function be used again later in your program, while a lambda usually cannot?

_____

I3. `apply(criteria, n)` was written once, yet it counted evens, multiples of 3, and numbers above 5. What made that possible?

_____

I4. In an environment diagram, why do we draw `say_hi = greet` as an arrow to the *same* box rather than a copy of it?

_____

Part J • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full. Work out the sample result on paper before you call it done.

J1. `count_if(criteria, s)`: how many characters of the string `s` make `criteria` return `True`. Then call it with a lambda to count vowels in `"lambda"` (answer: 2).

J2. `first_match(criteria, n)`: return the *first* integer from 0 to `n` for which `criteria` is `True`, or `None` if there is none. `first_match(lambda x: x * x > 20, 10)` is `5`; `first_match(lambda x: x < 0, 10)` is `None`.

J3. `compose_call(f, g, x)`: return `f` applied to the result of `g` applied to `x`. With `f = lambda x: x + 1` and `g = lambda x: x * 10`, `compose_call(f, g, 2)` is `21`; swapping `f` and `g` gives `30`.

J4. Fix the recap slide's `is_triangular(n)` so it *returns* correctly (`n` is triangular if it equals $1 + 2 + \cdots + k$ for some k), then use it with `apply` to count the triangular numbers from 1 to 10. Expected: `4`.

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. Define `def double(x): return x * 2`. Type `double`, then `type(double)`, then `lambda x: x * 2` on its own. Compare the two function objects' names in what Python shows.
2. Type `(lambda x: x * 3)(4)`. Then `(lambda x: x * 3)`. Then `lambda x: x * 3 (4)` — no outer brackets. Which one is not a call at all?
3. Type `def f(): return w`, then `f()`. Now `w = 1` and `f()` again. Then `w = "a"` and `f()`. When does Python decide what `w` is?
4. Type `d = double`, then `double = 5`, then `d(4)`. The function object is still alive — why? Then try `double(4)`.

Stuck on more than two items? Re-read the Lecture 10 slides on frames ("Local vs Global Variables", "Multiple Function Calls") and on `apply` before the next class — then redo Parts B, D and E cold, drawing every frame. If you can draw the frames, you can predict any program in this course.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. the **Global** Frame; a **new frame** for that call (its own private box of names).

A2. `lambda` *parameters* `:` *one expression.* The value of that single expression is handed back automatically — you never write `return` inside a lambda.

A3. it **takes a function as an argument**, or it **returns a function** as its result.

A4. **parameters**; **argument** values; **destroyed**; **return value**. Locals made inside the body join the frame as the body runs, and die with it.

A5. the function's **own (local)** frame first; then the **Global** Frame; `NameError`. Reading a global from inside a function is fine — see B1(a).

A6. (a) **False** — `def` only creates the function object in the Global Frame; nothing runs yet. (b) **False** — the frame is destroyed on return, and the local names go with it. (c) **False** — a lambda body is a single expression; `return` is not allowed. (d) **True** — functions are first-class objects. (e) **False** — every call gets a fresh frame, even calls to the same function. (f) **False** — it creates a new *local* `x`; the global one is untouched. (g) **True** — `lambda x, y: x + y`. (h) **False** — the first passes the function object; the second *calls* it with no argument and crashes. (i) **True** — the slides drew two lambda frames for `do_twice`.

B1. (a) `8` — `y` is in the `f` frame, `x` is not, so Python looks `x` up in the Global Frame and finds `5`. (b) `103` then `5`. Assigning `x` inside `f` makes a *brand-new local* `x` in the `f` frame; the global `x` is never touched, so the last line still prints `5`.

B2. `10`. Two things go wrong for anyone expecting `11`: `n` lives only in the `bump` frame, so re-binding it cannot reach `a`; and the returned `11` is thrown away because nobody catches it.

B3. Left: `11 1`. Inside the call, `x` means the parameter (10) in the `f` frame; the global `x` is a different box and stays 1. Right: `1 1`. Each call starts a *fresh* frame with `n = 0`; a frame keeps no memory of earlier calls, so the second call does not see the first one's `n`.

B4. Left: `hi`. `msg` does not exist when the `def` is read, but that does not matter: the body only runs at the call on line 4, and by then `msg` is in the Global Frame. Right: `5`, then `NameError`. `t` was a local of the `a` frame, which was destroyed when `a` returned; `b` looks in its own frame, then Global, and finds no `t` anywhere.

B5. `12`, `12`, `False`. Line 1 gives the lambda a name, so `f(4)` is an ordinary call. Line 3 calls a lambda *on the spot*: the brackets around it make the function object, the `(3, 4)` calls it. Line 4's body is a comparison, so `g` returns a boolean — exactly the shape a `criteria` needs.

B6. Left: `8` then `4`. `d = double` (no brackets!) points a second name at the same function object; both names call it. Right: `TypeError` ("'int' object is not callable"). Line 3 re-binds the name `double` to the number 5, so the function object is no longer reachable by that name, and `5(2)` is nonsense. A function name is just a variable.

B7. `21` then `hi!!`. The inner call gives `11` (or `"hi!"`), and that result is fed straight back into the same function. `apply_twice` never cares what `func` does, or even what type `n` is — that is why it is worth writing once.

B8. `3` then `None`. The lambda is legal — `print(x)` is an expression — but its value is what `print` returns, which is `None`. Same lesson as Lecture 9, in lambda clothing: printing is not returning.

C1. `b` is `18`. **3** frames in total: the Global Frame plus one `triple` frame per call. Only **2** ever exist at the same moment — each `triple` frame is destroyed the instant it returns, before the next call begins.

C2. `16` — the inner call gives 4, the outer squares that. **4** frames in total: Global, `do_twice`, and one lambda frame *per call.* At most **3** alive at once: while a lambda frame runs, the `do_twice` frame that called it is still waiting below it. The first lambda frame is gone before the second appears.

C3. `6` (0, 2, 4, 6, 8, 10 — zero is even and ten is included); `5` (6 to 10); `4` (0, 3, 6, 9 — zero again, `0 % 3` is 0); `1` — `range(1)` visits only 0, which is even; `0` — the criteria never fires, and the starting `count` comes back. Same loop every time; only the black box passed in changes.

C4. `14` ($1+4+9$) then `10` ($1+2+3+4$: the identity lambda hands each `i` straight back). Line 8 creates **3** `square` frames, one per loop round, each destroyed before the next. Note `square` is passed *without* brackets — see H4 for what happens with them.

D. `x` — **Global**, 10. `f` — **Global** (the `def` puts the function object there), a function object. `a` — `f`

frame, 13 in the second call (a parameter is a local name). `b` — `f` **frame**, 23. `y` — **Global**, 13. `z` — **Global**, not yet assigned at that moment; 23 once the call returns. The first call's `a` and `b` (3 and 13) were destroyed long before the second call made its own.

E1. **At `*`:** Global has `base : 100` and `add_tax : func` — *not* `cost`, which is only assigned once the call returns. The `add_tax` frame has `price : 50` and `tax : 5.0`. **After line 6:** Global gains `cost : 55.0`; the `add_tax` frame is crossed out — it was destroyed on return, taking `price` and `tax` with it.

E2. Global: `sq : func`, `twice : func`, and no `r` yet. `twice` frame: `fn : func` (the same object `sq` points at — an arrow to it is the best drawing) and `n : 3`. Second `sq` frame: `x : 9`. The destroyed frame was the *first* `sq` call, with `x : 3`; it returned 9, which became the argument of the second call. After line 5, `r : 81`. Frames stack: `twice` is still waiting underneath while each `sq` runs.

E3. Global: `greet : 7`, `hello` → the function object, `s : "Hi Ali"`. Line 3 gave the object a second name; line 4 re-pointed the *first* name at a number, which does not harm the object — `hello` still reaches it, so line 5 works and creates **1** frame (`name : "Ali"`, gone by the end). `greet("Ali")` would now be a `TypeError`: `7` is not callable.

F1. Accepted: (a), (b), (e), (g). (e) is a lambda with *no* parameters that always gives 5 — call it as `(lambda: 5)()`. Rejected: (c) `return` is a statement; (d) missing colon; (f) an assignment is a statement, not an expression; (h) a loop is a statement. The rule behind all four: the body must be *one expression* — something that has a value.

F2. (a) lambda (b) def (c) def (d) lambda. Lambda for a short, one-time, single-expression function passed as an argument; `def` for anything with statements, documentation, or reuse. Both create the same kind of object — the choice is about readability.

G1. `add = lambda x, y: x + y`; `last = lambda s: s[len(s) - 1]`; `count_a`: **cannot** — three statements and a loop; keep the `def`; `is_big = lambda n: n > 100` — the `if`/`else` only copies out a comparison that is already a boolean, so it *can* be a lambda once you see that.

G2. `def half(x): / return x / 2` and `def both(a, b): / return a and b`. The lambda's parameters become the `def`'s parameters and its expression becomes the `return` line.

G3. `apply(lambda x: x % 10 == 7, 20)` prints 2 (7 and 17). The lambda must take *one* number and give back a boolean; `n` is 20 because `apply` already adds the 1.

H1. Line 1: the **colon is missing** — `square = lambda x: x ** 2`. Line 2: a lambda body is *just an expression*, so `return` is not allowed — `half = lambda x: x / 2`. Both are `SyntaxError`s: Python refuses the file before running anything.

H2. `add` *prints* but never *returns*, so it hands back `None` and the last line breaks with a `TypeError`. Change the body to `return x + y`. Printing and returning are not the same thing.

H3. `TypeError` ("`is_even()` missing 1 required positional argument"). The brackets *call* `is_even` right there, with nothing to give it. Pass the function object: `apply(is_even, 10)` — no brackets. (With `is_even(4)` instead you would get past this line but pass `True` to `apply`, which then fails trying to call a bool.)

H4. `TypeError` ("'int' object is not callable"). `double(3)` is replaced by 6 before `apply_twice` even starts, so `func` holds the number 6 and `6(3)` is meaningless. Write `apply_twice(double, 3)`. Brackets mean "call it now"; no brackets mean "hand it over".

H5. Prints `11`. `if criteria:` never calls the function — it just asks whether the function *object* counts as true, and every function object does. So all eleven numbers are counted. Write `if criteria(i):`.

H6. Prints `<function <lambda> at 0x...>` — the function object itself, which has no name of its own (`f` is only a variable pointing at it). It was never called. Write `print(f(4))`.

H7. `TypeError` ("missing 1 required positional argument: 'y"'). `apply_twice` calls `func` with *one* argument, so the function you hand it must take exactly one. Write `lambda x: x + 3` ($3 + 3 + 3 = 9$). Read the higher-order function's body to learn what shape of function it expects.

H8. `NameError: name 'n' is not defined`. `setup` created `n` in its *own* frame, which vanished when it returned; `total` searches its frame and then Global and finds nothing. Frames do not share locals. Make `setup` return 4 and pass it in: `def total(n): return n + 1`, `print(total(setup()))`.

I1. A frame exists only for the length of one call — once the function returns, Python throws the frame (and every local name in it) away, keeping only the returned value.

I2. `def` binds a **name** to the function object, so you can call it whenever you like; a lambda is anonymous, so unless you deliberately store it in a variable there is no name left to call it by.

I3. The *test* is a parameter: the caller hands in a function, and `apply` only knows it can call it on a number and get a boolean. The loop is written once; the varying part is passed in as data.

I4. Because assignment binds a name to an object, it does not duplicate the object; there is one function object and now two names in the Global Frame that point at it.

J1. `c = 0` / `for ch in s:` / `if criteria(ch):` / `c += 1` / `return c`. Call: `count_if(lambda ch: ch in "aeiou", "lambda")`. Same skeleton as `apply`, walking a string instead of a range.

J2. `for i in range(n + 1):` / `if criteria(i):` / `return i` / then after the loop `return None`. The early `return` leaves the loop at the first hit; only a search that finds nothing reaches the last line.

J3. `return f(g(x))`. One line: `g` runs first (inner), `f` second. Order matters: $(2 \times 10) + 1 = 21$ but $(2 + 1) \times 10 = 30$.

J4. `total = 0` / `for i in range(1, n + 1):` / `total += i` / `if total == n:` / `return True` / then `return False` after the loop. `apply(is_triangular, 10)` gives 4: 1, 3, 6, 10.