Finger Exercises 9

Abstraction, Decomposition, 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: *when does a function's body actually run, what comes back from a call, and which names exist where?*

How to do it. On **paper**, **no computer**, in one sitting. Whenever you meet a call, write its return value above it in the margin and read the line again with the call replaced by that value. If an item takes you more than a minute, skip it, finish the rest, then come back. Parts A–H are for *everyone*. Part I is a **stretch**: skip it freely, or attack it if A–H 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. A definition starts with the keyword _____, then the function's _____, then its _____ in brackets, then a _____.

A2. No `return` means a function hands back _____. To include `b`, write `range(a, _____)`.

A3. Hiding how something works, showing only what you need: _____. Breaking a big problem into small self-contained parts: _____.

A4. The value you write in a call, e.g. the `5` in `square(5)`, is an _____; the name that receives it in the `def` line is a _____. A function with no parameters is still called with _____.

A5. The text in triple quotes right under the `def` line is the _____. It should say what the _____ are and what the function _____. Python _____ it as code.

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

(a) A function's body runs as soon as Python reads its `def` line. True / False

(b) `print` inside a function sends a value back to the caller. True / False

(c) Default parameters must go at the end of the parameter list. True / False

(d) `is_even` and `is_even(4)` mean the same thing. True / False

(e) After `square(4)` runs, you can `print(x)` outside the function to see 4. True / False

(f) Once a `return` line runs, nothing else in that function runs. True / False

(g) A function may be called many times; each call runs the body again from the top. True / False

(h) `y = power(2)` stores the function itself in `y`. True / False

(i) A function can call another 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.

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

Output: _____

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

Output: _____

B2.

```
def power(x, n=2):
    return x ** n
print(power(2, 3))
print(power(3, 2))
```

Output:

_____

_____

B3. In what *order* do the four lines appear?

```
print("start")
def f():
    print("in f")
print("middle")
f()
print("end")
```

Output:

_____

_____

_____

_____

B4. `return` ends the call at once.

```
def check(n):
    if n > 10:
        return "big"
    return "small"
print(check(50))
print(check(3))
```

Output: _____

```
def first_even(a, b):
    for i in range(a, b + 1):
        if i % 2 == 0:
            return i
    return None
print(first_even(3, 9))
print(first_even(5, 5))
```

Output: _____

B5. A call is replaced by its return value — so calls can nest.

```
def square(x):
    return x * x
def add(a, b):
    return a + b
print(square(add(1, 2)))
print(add(square(2), square(3)))
print(square(2) == add(2, 2))
```

Output:

_____

_____

_____

B6. Same name inside and outside.

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

Output: _____

```
rate = 10
def cost(n):
    return n * rate
print(cost(3))
print(n)
```

Output: _____

B7. A `print` inside, and a name without brackets.

```
def add(a, b):
    print("adding")
    return a + b
print(add(1, 2))
```

Output: _____

```
def hello():
    print("hi")
hello()
hello
hello()
```

Output: _____

B8. A function inside a condition inside a loop.

```
def is_even(i):
    return i % 2 == 0
count = 0
for i in range(1, 8):
    if is_even(i):
        count += 1
print(count)
```

Output: _____

How many times is `is_even` called?

_____

Part C • Functions Calling Functions

TRACE ≈ 5 min

C1. Write what **Code A** (left) and **Code B** (right) print.

```
def is_even(i):
    return i % 2 == 0
def is_odd(i):
    return not is_even(i)
print(is_odd(7))
```

Output: _____

```
def square(x):
    return x * x
y = square(3) + square(4)
print(y)
```

Output: _____

C2. Using `sum_odd` from the slides (below), fill in each result. Do the small ones on paper first — that is the whole point.

```
def sum_odd(a, b):
    total = 0
    for i in range(a, b + 1):
        if i % 2 != 0:
            total += i
    return total
```

`sum_odd(2, 7)` → _____

`sum_odd(1, 3)` → _____

`sum_odd(5, 5)` → _____

`sum_odd(4, 4)` → _____

`sum_odd(7, 2)` → _____

C3. Three levels deep.

```
def is_even(i):
    return i % 2 == 0
def is_odd(i):
    return not is_even(i)
def count_odd(a, b):
    c = 0
    for i in range(a, b + 1):
        if is_odd(i):
            c += 1
    return c
print(count_odd(1, 6))
```

Output: _____

How many times does `is_even` run?

_____

Which function does the main program call directly? _____

Part D • Which Calls Give 125? CLASSIFY ≈ 3 min

Using `power(x, n=2)` from **B2**: we want $5^3 = 125$. Tick one column per call.

The call	Gives 125	Error	Runs, but not 125
`power(5, 3)`	☐	☐	☐
`power(5, n=3)`	☐	☐	☐
`power(n=3, x=5)`	☐	☐	☐
`power(x=5, n=3)`	☐	☐	☐
`power(n=3, 5)`	☐	☐	☐
`power(3, 5)`	☐	☐	☐
`power(5)`	☐	☐	☐
`power()`	☐	☐	☐
`power(5, 3, 1)`	☐	☐	☐

Part E • What Lives in Program Scope? CLASSIFY ≈ 3 min

After this program has run to the end, tick every name that exists in **program scope**, and write its value. Cross out the rest.

```
def area(w, h):
    result = w * h
    return result

a = area(2, 3)
b = area(4, 5)
```

☐ `area`: _____ ☐ `w`: _____

☐ `a`: _____ ☐ `h`: _____

☐ `b`: _____ ☐ `result`: _____

Part F • Fill In / Translate TRANSLATE ≈ 5 min

F1. Write only the `def` line (header) for each description.

(a) `average`: takes two numbers. _____

(b) `greet`: takes a name, and a greeting that is `"Hello"` unless the caller says otherwise. _____

(c) `roll`: takes nothing. _____

F2. Rewrite each function so that it *returns* the answer instead, in a **single** line of body.

```
def is_pos(n):
    print(n > 0)
```

Body: _____

```
def is_even(i):
    if i % 2 == 0:
        return True
    else:
        return False
```

Body: _____

F3. Complete `sum_even` so that it returns the sum of the even numbers from `a` to `b` *inclusive*, reusing `is_even`. `sum_even(1, 6)` must give 12.

```
def is_even(i):
    return i % 2 == 0
def sum_even(a, b):
    total = ________
    for i in range(________________):
        if ________________:
            ____________________
    ____________________
```

F4. `def greet(name, greeting="Hello"):` prints the greeting, a comma, then the name — so `greet("Ali")` prints `Hello, Ali`. Write a call that prints `Salaam, Sara`:

_____

Part G • Spot the Bug

FIX ≈ 8 min

Each program has exactly one bug of a kind we saw in class. Say what actually happens (wrong output, or the error name), then write the fix.

G1. Should print 6. Fix: _____

```
def triple(x):
    x * 3

print(triple(2))
```

G2. Should print 25. Fix: _____

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

print(square)
```

G3. `sum_to(1, 4)` should print 10. It prints _____. Fix: _____

```
def sum_to(a, b):
    total = 0
    for i in range(a, b):
        total += i
    return total
print(sum_to(1, 4))
```

G4. Should print 7. It crashes. Error: _____ Fix: _____

```
def double(n):
    print(n * 2)
y = double(3) + 1
print(y)
```

G5. Should print `done` and then 6. It prints _____. Fix: _____

```
def area(w, h):
    return w * h
    print("done")
print(area(2, 3))
```

G6. `count_a("banana")` should print 3. It prints _____. Fix: _____

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

G7. Refused before anything runs. Error: _____ Fix: _____

```
def power(n=2, x):
    return x ** n
```

G8. Should print 16. Error: _____ Fix: _____

```
def square(x):
    return x * x
square(4)
print(x)
```

G9. Should print 9. Error: _____ Fix: _____

```
print(square(3))

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

G10. Should print 5. Error: _____ Fix: _____

```
def add(a, b):
    return a + b
print(add(3))
```

G11. `sign(0)` should print `zero`. It prints _____. Fix: _____

```
def sign(n):
    if n > 0:
        return "positive"
    elif n < 0:
        return "negative"
print(sign(0))
```

Part H • One Sentence Each

EXPLAIN ≈ 4 min

H1. A friend's function uses `print`, not `return`. Why can you not store its result?

H2. `sum_odd` calls `is_odd`, which calls `is_even` — what do we gain?

H3. `square(x)` and `cube(x)` both name their parameter `x`. Why is that not a clash?

H4. `power(3, 5)` gives no error, yet the slides call it wrong. What does "wrong" mean here?

H5. Before coding `sum_odd`, the slides worked out `sum_odd(2, 7) = 15` by hand. What is that number for?

Part I • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full, with a one-line docstring. Work out the sample result on paper before you call it done.

I1. `is_vowel(c)`: `c` is one lowercase character. Return `True` if it is a vowel. Then `count_vowels(s)`, which *uses* `is_vowel`: `count_vowels("function")` is `3`.

I2. `max_of(a, b)`: return the larger of two numbers. `max_of(3, 9)` and `max_of(9, 3)` are both `9`; `max_of(4, 4)` is `4`.

I3. `repeat(s, n)`: return `s` joined to itself `n` times, with `n` defaulting to 2. `repeat("ab")` is `"abab"`; `repeat("ab", 3)` is `"ababab"`.

I4. `sum_squares(a, b)`: the sum of i^2 for every `i` from `a` to `b` inclusive, *using* a `square` function. `sum_squares(1, 3)` is `14`.

I5. `is_prime(n)`: `n` is an integer ≥ 2. Return `True` if no number from 2 up to `n - 1` divides it, *using* `div_by(n, d)` from class. *Hint: the moment you find a divisor, you already know the answer.*

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 square(x): return x * x`, then type just `square` and Enter. Then `type(square)`. Then `x`. Then `square(4) + square`. Read every message.
2. Type `x = print("hi")` and then `x`. Then `print(print("a"))`. Explain the second line's output to yourself using B1.
3. With `def power(x, n=2): return x ** n`, try `power(n=3, 5)`, `power(5, 3, 1)`, `power(5, x=3)` and `power()`. Four different messages — match each to a rule from the slides.
4. Give `power` a docstring on the line after the `def`, then type `help(power)`. Now you know who reads docstrings.

Stuck on more than two items? Re-read the Lecture 9 slides on `print` versus `return`, on program scope, and on default parameters before the next class — then redo Parts B and G cold, writing each call's return value in the margin. Functions are the unit everything from here on is built from.

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. `def` ; **name** ; **parameters** ; **colon** `:`. A docstring and the body come next, both indented.

A2. `None` — every Python function returns something. Write `range(a, b+1)`; without the `+1` the loop stops one short.

A3. **abstraction** (the black box); **decomposition**.

A4. **argument**; **parameter**; empty brackets `()`. Without the brackets nothing runs (see B7).

A5. **docstring**; the **inputs** (parameters and their types); **returns**; does **not run** it — it is documentation for the person using the black box, and it is exactly what you read instead of the body.

A6. (a) **False** — the body runs only when you *call* the function. (b) **False** — `print` writes to the screen; only `return` hands a value back. (c) **True**. (d) **False** — without brackets you are naming the function object, not calling it. (e) **False** — `x` lives only inside the call; outside it is a `NameError`. (f) **True** — `return` ends the call at once, even inside a loop. (g) **True**. (h) **False** — the call is replaced by its return value, so `y` is 4. (i) **True** — that is decomposition.

B1. Left: `10` then `None` — the function prints 10 on its own, but hands back nothing, so `x` holds `None`. Right: `10` only — `return` hands 10 back, `x` holds 10, and one `print` runs.

B2. `8` then `9`. The two calls use the same two numbers but are *not* the same: the first argument always lands in `x` and the second in `n`, so we get 2^3 and then 3^2. Argument order matters.

B3. `start`, `middle`, `in f`, `end`. Python reads the `def` and moves on without running the body; `in f` only appears when `f()` is reached on line 5.

B4. Left: `big` then `small`. No `else` is needed: if the `if` fires, its `return` leaves the function and the last line is never reached. Right: `4` then `None`. The loop is abandoned on the first even number — 5, 6, ... are never visited. For `(5, 5)` the loop finds nothing, falls out, and reaches `return None`.

B5. `9`, `13`, `True`. Work inside out: `add(1, 2)` becomes 3, then `square(3)` is 9. Next $4+9$. Last, $4 == 4$. A call can sit anywhere a value can — inside another call, in arithmetic, in a comparison.

B6. Left: `5 10`. The `x` inside `double` is the parameter — a separate box that lives only during the call. Doubling it does not touch the program-scope `x`, which is still 5. Right: `30`, then `NameError`. A function *can* read a program-scope name like `rate`, but the parameter `n` does not exist outside the call.

B7. Left: `adding` then `3`. The inner `print` runs *during* the call, before the call has a value; the outer `print` then shows the returned 3. This is the "always debug" print from the slides. Right: `hi` twice. The bare `hello` on line 4 is not a call: Python evaluates the function object and throws it away. No error, no output.

B8. `3` (the evens 2, 4, 6); `is_even` is called **7** times, once per loop round. The call becomes `True` or `False`, so `if is_even(i):` is exactly `if i % 2 == 0:` — just easier to read.

C1. **Code A** prints `True`. `is_odd` does no work of its own — it calls `is_even` and flips the answer with `not`. That is decomposition. **Code B** prints `25`. Each call is *replaced by its return value*, so the line becomes $9 + 16$.

C2. `15` ($3 + 5 + 7$); `4` ($1 + 3$); `5` — a one-number range, and 5 is odd; `0` — the only number, 4, is even, so nothing is added and the starting `total` comes back; `0` again — `range(7, 3)` is empty, the loop never runs. No crash in any case: an empty loop just leaves `total` alone.

C3. `3` (1, 3, 5). `is_even` runs **6** times: once per loop round, because every `is_odd(i)` call passes the question down to `is_even`. The main program calls only `count_odd`; it never needs to know that `is_odd` or `is_even` exist — that is the black box at work.

D. **125, 125, 125, 125, error, not 125, not 125, error, error.** Naming the parameters lets you put them in any order, so both `power(n=3, x=5)` and `power(x=5, n=3)` are fine. `power(n=3, 5)` is a **SyntaxError**: a plain argument may never follow a named one. `power()` is a **TypeError**: `x` has no default, so it is required. `power(5, 3, 1)` is a **TypeError**: at most two arguments fit. The two traps both run happily: `power(3, 5)` computes $3^5 = 243$ because order decides which value lands in `x`, and `power(5)` quietly uses the default `n=2` to give 25. Running without an error does not mean correct.

E. In program scope: `area` (the function object), `a` = 6, `b` = 20. Not there: `w`, `h`, `result`. Parameters and any variable created inside the body exist only while a call is running, and are thrown away when

it returns — twice here, once per call. `print(result)` at the end would be a `NameError`.

F1. (a) `def average(a, b):` (b) `def greet(name, greeting="Hello"):` — the default one goes *last* (c) `def roll():` — empty brackets and the colon are still required.

F2. Left: `return n > 0`. Right: `return i % 2 == 0`. A comparison is *already* a `True`/`False` value; the `if`/`else` only copies it out. (For `is_pos(0)` both versions give `False`.)

F3. `total = 0`; `range(a, b + 1)`; `if is_even(i):`; `total += i` (or `total = total + i`); `return total`, indented one level — inside the function but *outside* the loop. Check: $2 + 4 + 6 = 12$.

F4. `greet("Sara", "Salaam")` or `greet("Sara", greeting="Salaam")` or `greet(greeting="Salaam", name="Sara")`. Not `greet("Salaam", "Sara")` — that runs, and prints `Sara, Salaam`.

G1. The `return` is missing, so the function works out 6 and then throws it away; it prints `None`. Write `return x * 3`.

G2. `square` without brackets is the function object itself, not a call, so Python prints something like `<function square at 0x...>`. Write `print(square(5))`.

G3. Prints `6`: `range(1, 4)` stops at 3, so the 4 is never added. Write `range(a, b + 1)`.

G4. Prints `6`, then `TypeError` — the call is replaced by `None`, and `None + 1` is meaningless. `print` inside a function cannot be reused in arithmetic; write `return n * 2`.

G5. Prints only `6`. The `print` sits *after* the `return`, so it is never reached — no error, it is simply dead code. Move it above the `return` line.

G6. Prints `0`. The `return` is indented into the loop body, so the function returns during the very first round — after looking only at `"b"`. Un-indent `return c` one level so it runs after the loop finishes.

G7. `SyntaxError` ("parameter without a default follows parameter with a default"). Defaults go at the end: `def power(x, n=2):`.

G8. `NameError: name 'x' is not defined`. The call ran and computed 16, but `x` vanished when it returned, and the 16 was never stored. Write `print(square(4))`, or `y = square(4)` then `print(y)`.

G9. `NameError: name 'square' is not defined`. Python runs top to bottom; on line 1 the `def` has not been read yet, so the name does not exist. Move the definition *above* the first call.

G10. `TypeError` ("missing 1 required positional argument: 'b'"). Two parameters, one argument. Either call `add(3, 2)`, or — if 2 is a sensible usual value — give `b` a default: `def add(a, b=2):`.

G11. Prints `None`. For 0 neither branch fires, the function falls off the end, and a function that reaches the end without a `return` hands back `None`. Add `else: return "zero"` (or a final `return "zero"` after the `elif`). Every path through a function should reach a `return`.

H1. Because `print` only puts characters on the screen and hands back `None`, so there is no value to store — only `return` gives one to the caller.

H2. Each small function is a black box we can write, test and *reuse* on its own, instead of copying the same code into every place that needs it.

H3. Each call gets its own fresh `x` that exists only while that call runs; the two functions never see each other's, and neither touches an `x` in program scope.

H4. Python cannot know you *meant* 5^3; it simply puts 3 in `x` and 5 in `n` and returns 243. The program is valid; the *answer* is not what you wanted. Only the docstring and a test on paper catch that.

H5. It is the *expected* answer: once the code runs, you compare against it. Without a hand-worked example you cannot tell a wrong result from a right one — code that runs is not code that works.

I1. `def is_vowel(c): return c in "aeiou"` — then `def count_vowels(s):` / `n = 0` / `for c in s:` / `if is_vowel(c):` / `n += 1` / `return n`. Two black boxes; the second never mentions `"aeiou"`.

I2. `if a > b:` / `return a` / `return b` (or `else: return b`). The equal case falls through to `return b`, which is fine because both are the same number.

I3. `def repeat(s, n=2):` / `return s * n`. String repetition from Lecture 3 does all the work; the default lives in the header, not in the body.

I4. `def square(x): return x * x` — then `total = 0` / `for i in range(a, b + 1):` / `total += square(i)` / `return total`. $1 + 4 + 9 = 14$.

I5. `for d in range(2, n):` / `if div_by(n, d):` / `return False` / then, after the loop, `return True`. The early `return False` leaves the loop at the first divisor; only a number that survives every round reaches `return True`. Check: 2, 3, 5, 7, 11 give `True`; 9 fails at $d = 3$.