Finger Exercises 7

Iteration

≈ 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 skill being drilled: *how many times does the body run, what does the loop variable hold each time, and what is left over when the loop ends?*

How to do it. On **paper**, **no computer**, in one sitting. For every loop, write the loop variable's value at the start of each round in the margin — that one habit catches almost every loop bug. 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 `while` loop handles its loop variable in three places:

_____ it *before* the loop _____ it in the condition _____ it inside the body

A2. Write out the numbers each of these produces:

`range(4)`: _____ `range(1,4)`: _____ `range(4,0,-1)`: _____

A3. Press _____ to stop a program stuck in an infinite loop. The keyword that ends a loop from *inside* is _____.

A4. Write each shorthand out in full, as an ordinary assignment:

`count += 1` is _____ `f *= i` is _____ `n -= 2` is _____

A5. `in` has two jobs. Name each one:

`for c in s:` _____ `"a" in s` _____

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

(a) `range(5)` includes the number 5. True / False

(b) A `while` loop can run its body zero times. True / False

(c) `for c in "hi":` gives `c` one character at a time. True / False

(d) Leaving out `step` in `range` defaults it to 1. True / False

(e) A `while` condition is checked once, at the start, and never again. True / False

(f) `break` skips the rest of the body *and* leaves the loop for good. True / False

(g) After `for i in range(3):` finishes, `i` is 3. True / False

(h) `range(3, 1)` produces `3 2`. True / False

(i) `while True:` always means an infinite loop. True / False

Part B • What Does It Print?

TRACE ≈ 8 min

Write the output exactly, one value per line, in order. Write *nothing* if the loop prints nothing.

B1.

```
1 for i in range(3):
2     print(i)
```

Output: _____

```
1 for i in range(1, 3):
2     print(i)
```

Output: _____

B2.

```
n = 3
while n > 0:
    print(n)
    n = n - 1
print(n)
```

Output:

_____

_____

B3. Steps other than 1.

```
for i in range(0, 10, 3):
    print(i)
```

Output: _____

```
for i in range(10, 0, -3):
    print(i)
```

Output: _____

B4. Order of the lines inside the body matters.

```
for i in range(3):
    print("hi")
```

Output: _____

```
n = 1
while n <= 3:
    n = n + 1
    print(n)
```

Output: _____

B5. `break` in a `for` loop.

```
for i in range(10):
    if i == 3:
        break
    print(i)
print("done", i)
```

Output:

_____

_____

B6. A condition with `and`.

```
n = 10
while n > 0 and n != 4:
    print(n)
    n = n - 3
print("end", n)
```

Output:

_____

_____

B7. Indexes from `range`, characters from `s`.

```
s = "python"
for i in range(2, 5):
    print(s[i])
```

Output: _____

```
s = "python"
for i in range(0, len(s), 2):
    print(s[i])
```

Output: _____

B8. A `for` over the characters, with an `if`/`else`.

```
for c in "a1b":
    if c == "1":
        print("digit")
    else:
        print(c)
```

Output:

_____

_____

B9. What is left when a `for` loop ends?

```
total = 0
for i in range(1, 4):
    total = total + i
print(i, total)
```

Output: _____

Part C • Running Totals and Counters

TRACE ≈ 7 min

Every loop below keeps something *alive* between rounds: a total, a count, or a string being built up. Track that variable round by round.

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

```
total = 0
i = 1
while i <= 3:
    total = total + i
    i = i + 1
print(total, i)
```

Output: _____

```
s = "loop"
count = 0
for c in s:
    if c == "o":
        count = count + 1
print(count)
```

Output: _____

C2. Multiplying, and building a string.

```
p = 1
for i in range(2, 5):
    p *= i
print(p)
```

Output: _____

```
s = "abc"
r = ""
for c in s:
    r = c + r
print(r)
```

Output: _____

C3. Counting with a condition.

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

Output: _____

```
count = 0
total = 0
for c in "895244254":
    if c == "2":
        count = count + 1
        total = total + int(c)
print(count, total)
```

Output: _____

C4. The user types 4, then 7, then 0.

```
total = 0
n = int(input("Next? "))
while n != 0:
    total = total + n
    n = int(input("Next? "))
print(total)
```

Prints: _____

How many times did the body run?

_____

C5. The Lost Forest, with a counter. For each script of what the user types, write what is printed and the final value of `count`.

```
count = 0
where = input("Go left or right? ")
while where == "right":
    count = count + 1
    where = input("Go left or right? ")
if count > 2:
    print(":(")
print("You got out!")
```

(a) types `right`, `right`, `left` prints: _____ count = _____

(b) types `right`, `right`, `right`, `left` prints: _____ count = _____

(c) types `left` prints: _____ count = _____

(d) types `RIGHT` prints: _____ count = _____

Part D • How Many Times Does the Body Run? CLASSIFY ≈ 4 min

Write the number of times the body runs. Write ∞ if it never stops. Assume nothing else in the body touches the loop variable.

The loop	Times	The loop	Times
`for i in range(1, 4):`	_____	`for i in range(3, 1):`	_____
`for c in "abc":`	_____	`for i in range(3, 3):`	_____
`for c in "a b":`	_____	`for i in range(0, 10, 3):`	_____
`for c in "":`	_____	`for i in range(10, 0, -5):`	_____
`while True:` (no `break`)	_____	`while True:` whose body starts with `break`	_____
`n = 0;` `while n > 0:`; body `n = n - 1`	_____	`n = 0;` `while n < 3:`; body `n = n + 1`	_____
`n = 3;` `while n > 0:`; body leaves `n` alone	_____	`n = 3;` `while n > 0:`; body `n = n + 1`	_____

Part E • The `in` Operator CLASSIFY ≈ 3 min

E1. Circle the value of each expression.

(a) `"a" in "banana"` True / False

(b) `"ana" in "banana"` True / False

(c) `"A" in "banana"` True / False

(d) `"q" in "comp102"` True / False

(e) `"102" in "comp102"` True / False

(f) `"an" in "nab"` True / False

(g) `"o" in "aeiou"` True / False

(h) `"aeiou" in "o"` True / False

E2. In each line, is `in` doing **iteration** (I) or a **membership** test (M)? Write I or M next to each `in`.

```
for c in word:                  # (a)
    if c in "aeiou":            # (b)
        print(c)
if "e" in word:                 # (c)
    print("has an e")
```

Part F • Fill In / Translate

TRANSLATE ≈ 5 min

F1. Write a `range(...)` call that produces exactly each sequence:

5 6 7: _____ 10 8 6 4 2: _____

0 5 10 15: _____ 3 2 1 0: _____

F2. Rewrite the `for` loop as a `while` loop that prints the same thing. Fill in the three blanks.

```
for i in range(2, 8, 2):
    print(i)
```

```
i = ________
while _________________:
    print(i)
    i = _________________
```

F3. Now the other way: rewrite this `while` loop as a single `for` loop with `range`.

```
n = 5
while n >= 1:
    print(n)
    n = n - 1
```

```
for n in range(_____________________):
    print(n)
```

F4. Complete the program so it prints how many times the character `"x"` appears in `s`.

```
count = ________
for c in ___________:
    if _________________:
        _____________________
print(count)
```

F5. `s` is a string with `len(s)` equal to 7. In `for i in range(len(s)):`

the first value of `i` is _____ the last value of `i` is _____

on the last round, `s[i]` is the _____ character of `s`.

Part G • Spot the Bug

FIX ≈ 7 min

Each program has exactly one bug of a kind we saw in class. Say what actually happens (wrong output, never stops, or the error name), then write the fix.

G1. Should print `1 2 3`, but never stops. Fix: _____

```
i = 1
while i <= 3:
    print(i)
```

G2. Should print `1 2 3 4 5`. Fix: _____

```
for i in range(1, 5):
    print(i)
```

G3. Should print 6, the sum $1+2+3$. It prints _____. Fix: _____

```
for i in range(1, 4):
    total = 0
    total = total + i
print(total)
```

G4. Should print `1 2 3`, but never stops. Fix: _____

```
i = 1
while i <= 3:
    i = 1
    print(i)
    i = i + 1
```

G5. Should count down `3 2 1`, but never stops. Fix: _____

```
n = 3
while n > 0:
    print(n)
    n = n + 1
```

G6. Should print 3. It prints _____. Fix: _____

```
count = 0
for c in "aaa":
    if c == "a":
        count + 1
print(count)
```

G7. Should print the sum 6 once. It prints _____. Fix: _____

```
total = 0
for i in range(1, 4):
    total = total + i
    print(total)
```

G8. Should count the a's. It crashes. Error: _____ Fix: _____

```
s = "banana"
count = 0
for i in range(len(s)):
    if c == "a":
        count = count + 1
print(count)
```

G9. Same goal, the opposite mix-up. Error: _____ Fix: _____

```
s = "banana"
count = 0
for i in s:
    if s[i] == "a":
        count = count + 1
print(count)
```

G10. Should print each letter, then crashes. Error: _____ Fix: _____

```
s = "banana"
for i in range(len(s) + 1):
    print(s[i])
```

G11. Should print `1`, the number of `b`'s. It prints _____. Fix: _____

```
s = "banana"
count = 0
for i in range(1, len(s)):
    if s[i] == "b":
        count = count + 1
print(count)
```

Part H • One Sentence Each

EXPLAIN ≈ 3 min

H1. Why must something inside a `while` body change the loop variable?

_____

H2. In `for i in range(5):` and `for c in "hello":`, what does the loop variable hold each time?

_____

H3. The Lost Forest program uses `while`, the "print 1 to 10" program uses `for`. What decides which one you reach for?

_____

H4. `for c in s:` and `for i in range(len(s)):` both visit every character. When would you *need* the second one?

_____

H5. Why does `range(1, 4)` give three numbers and not four?

_____

Part I • Write the Code

STRETCH ≈ 10 min, optional

Write each program in full. Predict its output for the sample before you call it done. Use only what Lecture 7 covered: `while`, `for`, `range`, `len`, indexing, `in`, `break`.

I1. Print the squares of 1 to 5, one per line: `1 4 9 16 25`.

I2. Ask the user for a whole number `n` and print `n!`, i.e. $1 \times 2 \times \cdots \times n$. For `5` it prints `120`.

I3. Count the vowels in a sentence stored in `s`. For `"the quick brown fox"` it prints `5`.

I4. Keep asking `Password?` until the user types `open`, then print `Welcome` and how many attempts it took. Typing `no`, `nope`, `open` should end with `Welcome after 3 attempts`.

I5. `s` holds lowercase letters. Print how many *different* letters it contains: for `"abca"` print `3`. *Hint: build up a string of the letters seen so far, and use* ***`in`*** *before adding.*

I6. Print `s` backwards, using a loop over its *indexes* (not the trick from C2). For `"loop"` print `pool`.

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. Type `while True: print("x")` and hit Enter twice. Watch it run away, then press `Ctrl + c`. Read the name of the error Python reports — you will meet it again.
2. In a *fresh* shell type `for i in range(0): print(i)` (Enter twice), then `print(i)`. Now do the same with `range(3)` and `print(i)` again. What does the first case tell you about a loop that never ran?
3. Try `for i in range(1, 10, 0): print(i)` and `for i in range(-3): print(i)`. One complains, one is silent — which, and why?
4. Type `"" in "hello"`, then `"h" in ""`. One of them surprises almost everyone. Then `"a" in "banana" == True` — the answer is not what you expect; leave the mystery for now, and just never write `== True` after an `in`.

Stuck on more than two items? Re-read the Lecture 7 slides on the `while` loop structure (set, test, change) and on `range` before the next class — then redo Parts B and C cold, writing the loop variable's value in the margin every round. Everything from here on is loops inside other loops; this is the skill to lock in.

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. **set** (initialise) it before the loop; **test** it in the condition; **change** it (increment) inside the body. Miss the third one and the loop never ends.

A2. `0 1 2 3` ; `1 2 3` ; `4 3 2 1`. The `stop` value is *never* included, in either direction.

A3. `Ctrl + c` ; `break`.

A4. `count = count + 1` ; `f = f * i` ; `n = n - 2`. Same meaning, just shorter; the old value is used on the right, then overwritten.

A5. In a `for` header, `in` **iterates**: `c` takes each character of `s` in turn. On its own, `in` is the **membership** operator: `"a" in s` is an expression that gives `True` or `False`.

A6. (a) **False** — `range(5)` stops at 4. (b) **True** — if the condition is false the very first time, the body never runs at all. (c) **True**. (d) **True**, and `start` defaults to 0. (e) **False** — it is re-checked before *every* round; that is how the loop knows when to stop. (f) **True**. (g) **False** — `i` keeps the *last* value it was given, which is 2. (h) **False** — with the default step of +1 you cannot count from 3 up to 1, so the range is *empty*; counting down needs `range(3, 1, -1)`. (i) **False** — a `break` inside the body ends it whenever you like.

B1. Left: `0 1 2`. Right: `1 2` — only *two* numbers. `range(1, 3)` starts at 1 and stops *before* 3, so 3 never appears.

B2. `3 2 1 0`, one per line. The loop prints 3, 2, 1; then `n` becomes `0`, the test fails and the loop ends — but `n` *keeps* that value, so the final `print(n)` shows `0`.

B3. Left: `0 3 6 9` — 9 is still below the stop of 10; the next value, 12, is not. Right: `10 7 4 1` — 1 is still above the stop of 0; the next value, −2, is not. The stop is a *wall*, not a target: the loop never has to land on it exactly.

B4. Left: `hi` three times. The loop variable does not have to be *used* — `range(3)` simply makes the body run 3 times. Right: `2 3 4`, *not* `1 2 3`: `n` is increased *before* it is printed each round, so the first thing printed is 2 and the last is 4 (printed right after `n` became 4, before the test `4 <= 3` fails).

B5. `0 1 2` then `done 3`. On the round where `i` is 3, `break` fires *before* the `print(i)`, so 3 is never printed — but `i` still holds 3 afterwards. The remaining values 4 ... 9 are abandoned; `range(10)` was a promise the loop did not keep.

B6. `10 7 4` is *wrong*. Trace: prints 10, $n = 7$; prints 7, $n = 4$; now `n != 4` is False, so the whole `and` is False and the loop stops *without* printing 4. Output: `10`, `7`, `end 4`. Either half of an `and` going False ends the loop.

B7. Left: `t h o` — `i` is 2, 3, 4 and `s[2]`, `s[3]`, `s[4]` are those letters (the first index is 0, so `p` is `s[0]`). Right: `p t o` — `len(s)` is 6, so `i` is 0, 2, 4: every second letter. Neither loop prints numbers; the numbers only choose *which* letter to look at.

B8. `a`, `digit`, `b`. Three characters, three rounds, and *exactly one* of the two branches runs each round. Note the middle one: `"1"` is a one-character *string*, so `c == "1"` is True for it.

B9. `3 6`. `total` is $1+2+3$. Unlike the `while` loop in B2, a `for` loop variable is *not* pushed one step past the end: it stops at the last value handed out, 3.

C1. **Code A** prints `6 4`. `total` is $1 + 2 + 3 = 6$. Watch `i`: it is **4**, not 3 — the loop only stops *after* `i` has grown past the test. **Code B** prints `2`. The loop looks at `l`, `o`, `o`, `p` one character at a time, and two of them are `"o"`.

C2. Left: `24`. `p` goes $1 \to 2 \to 6 \to 24$ ($2 \times 3 \times 4$). It starts at 1, not 0 — a running *product* started at 0 would stay 0 forever. Right: `cba`. `r` goes `""`, then `"a"`, then `"ba"`, then `"cba"`: each new character is put *in front*, so the string comes out reversed. Written as `r = r + c` it would give `abc`.

C3. Left: `3` — the even numbers among 1 ... 7 are 2, 4, 6. Right: `2 4`. Look carefully: `895244254` has only *two* 2's (the 4's are not 2's), so `count` is 2 and the sum of those 2's is 4. `int(c)` is needed because `c` is the *string* `"2"`; `total + "2"` would crash.

C4. Prints `11`; the body ran **2** times. Round 1 adds 4, reads 7. Round 2 adds 7, reads 0. Then the test `0 != 0` fails. The `0` is a *stop signal* and is never added. Notice the *two* `input` calls: one before the loop to get the first value, one at the *end* of the body to get the next — exactly the Lost Forest pattern.

C5. (a) `You got out!`, count = 2 — the body runs once per `right`, and 2 is not `> 2`. (b) `:(` then `You got out!`, count = 3. (c) `You got out!`, count = 0 — the condition is false on the first test, so the

body never runs. (d) same as (c): `"RIGHT" == "right"` is False, capitals and all. In every case the last line prints, because it is *after* the loop and not inside the `if`.

D. Left column, top to bottom: **3, 3, 3, 0, ∞, 0, ∞**. Right column: **0, 0, 4, 2, 1, 3, ∞**. Traps: `range(1, 4)` is 3 rounds, not 4 — count the values `1 2 3`. The space in `"a b"` is a character, so 3. `range(3, 1)` and `range(3, 3)` are *empty* (start is not below stop, step is +1), so 0 — no error, the body is simply skipped. `range(0, 10, 3)` is `0 3 6 9`; `range(10, 0, -5)` is `10 5`. A `while` whose condition is false at the first test runs 0 times. `while True` with an immediate `break` runs exactly once. The last row moves `n` the *wrong way*: 4, 5, 6, ... is always `> 0`.

E1. (a) True (b) True — a whole *substring* can be tested, not just one character (c) **False** — case matters, `"A"` is not `"a"` (d) False (e) True (f) **False** — `"nab"` contains `n`, `a`, `b` but not `"an"` in that *order* (g) True (h) **False** — the left side must be found *inside* the right side; `"aeiou"` is not inside `"o"`. Swapping the sides changes the question.

E2. (a) I — it follows `for`, so `c` walks through `word`. (b) M — it follows `if` and gives True/False: "is `c` one of the vowels?" (c) M. Rule of thumb: `in` right after a `for` variable iterates; anywhere else it asks a yes/no question.

F1. `range(5, 8)`; `range(10, 0, -2)` (stop anywhere from 0 to 1 works); `range(0, 20, 5)` (stop anywhere from 16 to 20 works); `range(3, -1, -1)` — to *include* 0 when counting down, the stop must go one past it, to −1.

F2. `i = 2`; `while i < 8:`; `i = i + 2`. Both print `2 4 6`. The three blanks are exactly the three places from A1: set, test, change — a `for` loop does all three for you in one line.

F3. `range(5, 0, -1)`. Prints `5 4 3 2 1`. The `while` keeps going while `n >= 1`, so the last value printed is 1; the `range` stop must therefore be 0, one *past* 1 in the direction of travel.

F4. `count = 0`; `for c in s:`; `if c == "x":`; `count = count + 1` (or `count += 1`). The last line is *unindented* so it runs once, after the loop.

F5. 0; 6; the **last** character. `range(len(s))` is `0 1 2 3 4 5 6` — exactly the seven valid indexes. Index 7 would be `s[len(s)]`, which crashes with `IndexError`; the "never includes the stop" rule is what keeps this loop safe.

G1. The loop variable is never changed, so `i` stays 1 forever. Add `i = i + 1` indented inside the body, below `print(i)`.

G2. Off by one: `range(1, 5)` stops before 5, so it prints only `1 2 3 4`. Write `range(1, 6)`.

G3. Prints `3`. `total = 0` is *inside* the loop, so the total is wiped back to 0 every round and only the last `i` survives. Move `total = 0` *above* the `for` line. An accumulator is set up once, before the loop.

G4. Every round starts by resetting `i` to 1, so `i = i + 1` can never get it past 3: it prints `1 1 1 ...` forever. Delete the `i = 1` inside the body; initialising belongs *before* the loop only.

G5. `n` *is* changed — but in the wrong direction: 4, 5, 6, ... are all `> 0`. Write `n = n - 1`. Changing the variable is not enough; it has to move *towards* making the condition false.

G6. Prints `0`. `count + 1` *computes* 1 and throws it away — nothing is stored. It must be `count = count + 1` (or `count += 1`).

G7. Prints `1 3 6`, one per line — the running total *every* round. The `print` is indented into the body. Un-indent it so it runs once, after the loop.

G8. `NameError: name 'c' is not defined`. The loop hands out *indexes* in `i`; there is no `c`. Either compare `s[i] == "a"`, or change the header to `for c in s:`. Pick one style and stay in it.

G9. `TypeError: string indices must be integers`. Here `i` is already a *character* (`"b"` on round one), and `s["b"]` makes no sense. Compare `i == "a"` directly — or rename `i` to `c` so the code reads honestly.

G10. `IndexError: string index out of range`. After printing all six letters, `i` becomes 6, and `s[6]` does not exist (the last index is 5). Use `range(len(s))` — the "+1" is a habit from wanting to include a stop value, and here it is wrong.

G11. Prints `0`. `range(1, len(s))` starts at index 1, so `s[0]` — the only `b` — is never looked at. No crash, just a silent wrong answer: the nastiest kind. Use `range(len(s))`, which starts at 0.

H1. Because the condition is re-tested every round using the same variable; if nothing changes it, the answer stays `True` and the loop runs forever.

H2. `i` takes the *numbers* 0, 1, 2, 3, 4, while `c` takes the *characters* h, e, l, l, o — `for` walks through whatever sequence you give it.

H3. Whether you know *in advance* how many rounds there will be. A fixed count (or a walk through a string) is a `for` with `range`/`in`; "keep going until something happens" — the user types `left`, a

number reaches 0 — is a `while`.

H4. When you need the *position* as well as the character — to print "found at index 3", or to look at the neighbour `s[i+1]`. If you only need each character, `for c in s` is shorter and cannot go out of range.

H5. Because the stop is excluded: it produces 1, 2, 3 and halts *before* 4. The count is stop minus start, $4 - 1 = 3$ — the same rule that made `s[1:4]` three characters long.

I1. `for i in range(1, 6):` / `print(i * i)`. Start at 1 and stop at 6, because 5 must be included.

I2. `n = int(input("n? "))` / `f = 1` / `for i in range(1, n + 1):` / `f = f * i` / `print(f)`. The product starts at 1, and the range stops at `n + 1` so that `n` itself is multiplied in. A `while` with `i <= n` is equally good.

I3. `count = 0` / `for c in s:` / `if c in "aeiou":` / `count += 1` / `print(count)`. The membership test replaces five `or`s. Spaces are simply not vowels, so they need no special case.

I4. `tries = 1` / `word = input("Password? ")` / `while word != "open":` / `word = input("Password? ")` / `tries = tries + 1` / `print("Welcome after", tries, "attempts")`. The first `input` is outside the loop and already counts as attempt 1; every extra `input` inside adds one. (Starting `tries` at 0 and counting *both* inputs also works.)

I5. `seen = ""` / `for c in s:` / `if not (c in seen):` / `seen = seen + c` / `print(len(seen))`. For `"abca"`, `seen` grows `a`, `ab`, `abc`, and the second `a` is already in it, so the length is 3. Two ideas from the lecture together: building a string round by round, and `in` as a test.

I6. `r = ""` / `for i in range(len(s) - 1, -1, -1):` / `r = r + s[i]` / `print(r)`. The first index is the last one, `len(s) - 1`; the stop is -1 so that index 0 is still included; the step is -1. (Printing `s[i]` one per line inside the loop is also fine.)