Finger Exercises 3

Strings and I/O

≈ 30 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class.

How to do it. On **paper**, **no computer**, in one sitting. Parts A–G are for *everyone*. Part H is a **stretch**: skip it freely, or attack it if A–G 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 • Warm-Up Recall

RECALL ≈ 3 min

A1. Three casting functions so far: `int()`, `float()` and _____ . `str(25)` evaluates to _____ (write it exactly as Python would show it).

A2. `len("")` is _____ . For a string of length `n`, the highest legal index is _____ and the lowest legal *negative* index is _____ .

A3. In `t[a:b:c]`, a negative `c` means walk from _____ to _____ . The slice `t[::-1]` evaluates to _____ .

A4. `print("Score", 40)` works and puts a _____ between the two items. `print("Score" + 40)` is an error because _____ .

A5. Circle **True** or **False**:

(a) If the user types only digits, `input()` hands back an `int`. True / False

(b) For any non-empty `s`, `s[len(s) - 1]` and `s[-1]` are the same character. True / False

(c) `"7" * 2` evaluates to `14`. True / False

(d) `s[2] = "x"` changes the third character of `s`. True / False

(e) When a program is run from a *file*, only `print()` puts anything on the screen. True / False

(f) `f"{2 + 2}"` evaluates to the string `"4"`. True / False

(g) `'me'` and `"me"` are the same string. True / False

(h) `"Ali"` and `"ali"` are the same string. True / False

Part B • One Symbol, Two Jobs

CLASSIFY ≈ 5 min

The meaning of `+` and `*` depends entirely on the **types** beside them. Write each value exactly as Python would show it (keep the quotes if it is a string!), or **ERROR**.

Expression	Value / ERROR	Expression	Value / ERROR
`2 + 9`	_____	`"9" * 2`	_____
`"2" + "9"`	_____	`2 * "9"`	_____
`"2" + 9`	_____	`"2" * "9"`	_____
`"2" + str(9)`	_____	`"no" * 3`	_____
`int("2") + 9`	_____	`len("no" * 3)`	_____

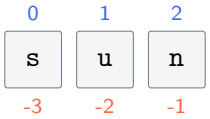
B1. Casts have edge cases too. Value, or **ERROR**?

Expression	Value / ERROR	Expression	Value / ERROR
`int("31")`	_____	`float("2")`	_____
`int(31.9)`	_____	`str(2.5)`	_____
`int("ten")`	_____	`len(str(2500))`	_____
`int("31.9")`	_____	`"no" * int("2")`	_____

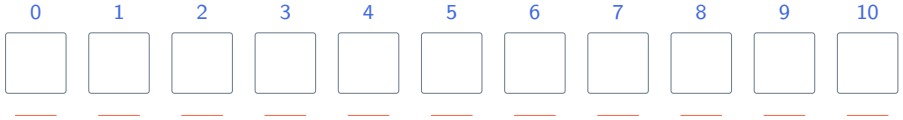
Part C • The Index Ruler

TRACE ≈ 4 min

Every string wears an invisible ruler. Here is `"sun"` with its ruler drawn — positive indices on top, negative underneath:



C1. Now let `s = "blue sky 42"`. Copy its characters into the boxes, **one per box** (spaces and digits get boxes too!), then write each character's negative index underneath:



C2. Read straight off your ruler (write strings with quotes):

`len(s)` → _____ `s[5]` → _____ `s[4]` → _____ `s[-2]` → _____ `s[10]` → _____

`s[11]` → _____ `s[-11]` → _____ `type(s[9])` → _____ `s[9] + s[10]` → _____

Part D • The Slice Machine

TRACE ≈ 6 min

Same string, same ruler: `s = "blue sky 42"`. Write what each slice evaluates to, with quotes. Look at the `step` first!

Slice	Value	Slice	Value
`s[0:4]`	_____	`s[2:7:2]`	_____
`s[5:8]`	_____	`s[8:4:-1]`	_____
`s[9:]`	_____	`s[7:3]`	_____
`s[-2:]`	_____	`s[::5]`	_____
`s[4:-1]`	_____	`s[2:6:-1]`	_____

D1. Still with `s = "blue sky 42"`. One of these does arithmetic and one does not:

`int(s[9:]) + 1` → _____ `s[9:] + "1"` → _____

Part E • The Input Pipeline

TRACE ≈ 8 min

E1. The user types `25` and hits Enter.

```
n = input("How many? ")
print(n + n)
print(int(n) + int(n))
print(n * 2)
```

Output:

_____

_____

_____

E2. Comma or plus?

```
team = "Lions"
goals = 3
print(team, goals)
print(team + str(goals))
print(team, "scored", goals)
```

Output:

_____

_____

_____

E3. Three near-identical lines. Write each output *exactly*.

```
k = 4
print(f"{k} times {k} is {k * k}")
print("{k} times {k} is {k * k}")
print(f"k times k is k * k")
```

Output:

_____

_____

_____

E4. This code is saved in a **file** and run. What appears in the output window?

```
"hello" * 2
print("hello" * 2)
7 - 2
print(7 - 2)
```

Output:

_____

_____

_____

E5. A chain of six bindings. Write the value of each name at the end of the program execution, with quotes where the value is a string.

```
a = 7
b = 2
c = a // b
d = str(c) * a
e = len(d)
f = d[0] + d[-1]
```

`c` → _____ `d` → _____

`e` → _____ `f` → _____

E6. One step of the square-root guesser from class. The user types `9`, then `4`. Write the line that is printed, and then answer the question underneath.

```
a = int(input("Number? "))
g = float(input("Guess? "))
next_g = 0.5 * (g + a / g)
print("Next guess =", next_g)
```

Printed:

_____

If line 2 used `int(...)` instead and the user typed `4.5`, what would happen?

_____

Part F • Bug Hunt

FIX ≈ 5 min

F1. There are **two** errors here. Circle both, then write the corrected lines.

```
price = input("Price? ")
total = price * 1.17
print("With tax: " + total)
```

_____

_____

F2. The programmer wanted `bake`. Python refuses line 2. Why — and what one line does the job instead? *(Hint: slice around the letter you are replacing.)*

```
word = "bike"
word[1] = "a"
print(word)
```

`word =` _____

F3. This should print `You got 45 out of 50`. What does it actually print, and what is the fixed line?

```
marks = 45
print("You got {marks} out of 50")
```

It prints:

_____

Fix:

_____

F4. Python stops with an error on line 2. Why, and give **two** different fixes for that line.

```
code = "FCCU"
last = code[4]
print(last)
```

`last =` _____

or `last =` _____

F5. No error, but the output is blank. The programmer wanted `hor`. What did line 2 actually print, and what is the fix?

```
city = "lahore"
print(city[5:2])
```

It printed: _____

Fix: `city[`_____`]`

F6. No error, but the output has an unwanted space: the programmer wanted `Speed:  30km/h`. What did it print, and give **two** different fixed lines.

```
speed = 30
print("Speed:", speed, "km/h")
```

It printed: _____

Fix 1: _____

Fix 2: _____

Part G • Say It in One Sentence

EXPLAIN ≈ 2 min

G1. Why does `print("Total:", 90)` work but `print("Total:" + 90)` crash?

_____

G2. A friend says that `s = s[::-1]` proves strings *can* be changed. What actually happened?

_____

G3. A program file contains the line `2 ** 10` on its own. Why does nothing appear on screen when the file runs?

_____

Part H • Write the Code

STRETCH ≈ 10 min, optional

Everything so far ran *forwards*: here is code, what does it do? These run *backwards*: here is the result — **you** write the code. This is where slicing starts to feel like a superpower. On paper, as always.

H1. Let `s = "university"`. Write a **slice of s** that evaluates to `"sit"`:

`s[_____]`

H2. Same `s`. Write **two different** slices that both give `"uni"`:

`s[_____]` and `s[_____]`

H3. One slice — no `+`, nothing else — turns `"stressed"` into `"desserts"`. Find it:

`"stressed"[_____]`

H4. Let `s = "comp102"`. Using only `s`, slicing, `+` and `int()`:

(a) build the string `"comp 102"`: _____

(b) build the number `103`: _____

H5. `w = 6` and `h = 3`. Write **one** `print` line, using an f-string, whose output is exactly

```
6 x 3 = 18 sq m
```

`print(_____)`

H6. Write a **two-line** program: ask the user for a word, then print it three times separated by dashes. If they type `go`, the output is `go-go-go`.

_____

_____

H7. Write a **four-line** program: ask the user for a word, then for a count; print the word repeated that many times with nothing in between; then print the word **reversed**. If they type `hey` and `3`, the output is `heyheyhey` then `yeh`.

_____

_____

_____

_____

Console Quests

tonight · 5 min · at a computer

The lecture's Big Idea: *when unsure, just try it in the console.* These three are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. `s = "sky"` — now try `s[-4]`, then `s[-4:]`, then `s[1:50]`. Which ones forgive you, and what do they give?
2. What is `"ab" * 0`? And `"ab" * -3`? And `len("" * 100)`?
3. Predict all three, then check: `int(" 7 ")` (with the spaces), `int("7.0")`, `float("7")`.

Stuck on more than two items? Re-read the Lecture 3 slides on indexing and slicing before the next class — then redo Parts C and D cold with a *different* string of your own. They are the backbone of the next three weeks.

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. `str()`. `str(25)` is `"25"` — two characters in quotes, no longer a number you can do arithmetic with.

A2. `0` (the empty string has no characters — the quotes are never counted, they only mark where the string starts and ends); `n - 1`; `-n`. Both rulers cover exactly `n` characters — one starts at 0 and one ends at −1.

A3. From **right** to **left**. `t[::-1]` is the whole string **reversed** — both ends left open, step −1.

A4. A **space**. The `+` version fails because `+` between a `str` and an `int` is not defined — every piece being glued with `+` must be a string (`str(40)`).

A5. (a) **False** — `input()` *always* returns a `str`, digits or not. (b) **True** — same last character, two rulers. (c) **False** — it is `"77"`: string times int repeats. (d) **False** — strings are immutable; that line is an error. (e) **True** — a bare expression in a file is evaluated and thrown away; only the console echoes values. (f) **True** — the braces are evaluated and the result becomes part of the string. (g) **True** — single or double quotes, Python does not care; the quotes are not part of the string. (h) **False** — strings are *case sensitive*: `"A"` and `"a"` are two different characters.

B. Left column: `11`; `"29"` (join, not arithmetic!); **ERROR** (cannot glue a string to a number); `"29"` (now both sides are strings, so it joins); `11` (now both sides are numbers, so it adds).
Right column: `"99"`; `"99"` (the `int` may sit on either side of the `*`); **ERROR** (the repeat count must be an `int`, not a string); `"nonono"`; `6` (build `"nonono"` first, then count its characters).
Rows 1, 4 and 5 on the left are the whole lesson: same `+`, three outcomes — *the operator does whatever the types tell it to.*

B1. Left: `31`; `31` (`int()` of a float *truncates* — it chops off the fraction, it does not round); **ERROR** (`int()` only accepts a string that looks like a whole number); **ERROR** (looks like a number to you, but `int()` refuses the decimal point — `float("31.9")` would work).
Right: `2.0`; `"2.5"` (a string now, note the quotes); `4` (turn the number into text, then count its characters); `"nono"` (the cast turns `"2"` into a repeat count — without it, `"no" * "2"` is an error).

C1. Boxes: `b`, `l`, `u`, `e`, (space), `s`, `k`, `y`, (space), `4`, `2` — eleven characters, with the two spaces at indices 4 and 8. Negative indices: `-11` to `-1`, left to right.

C2. `len(s)` is `11`. `s[5]` is `"s"`. `s[4]` is `" "` — a space *is* a character. `s[-2]` is `"4"`. `s[10]` is `"2"`. `s[11]` is an **ERROR**: eleven characters means the highest legal index is 10, so `s[len(s)]` is always one step too far. `s[-11]` is `"b"` — the very first character, reached from the other end. `type(s[9])` is `str`: a digit *inside a string* is still a character, not a number. So `s[9] + s[10]` is `"42"`, a join, not 6.

D. `s[0:4]` is `"blue"` (indices 0–3; the stop index 4 is left out). `s[5:8]` is `"sky"`. `s[9:]` is `"42"` (missing stop means go to the end). `s[-2:]` is also `"42"` — same slice, reached by the negative ruler; handy when you do not know the length. `s[2:7:2]` is `"u k"` (indices 2, 4, 6 — the middle one is a space). `s[8:4:-1]` is `" yks"` (indices 8, 7, 6, 5, walking leftwards; it starts with the space at 8). `s[7:3]` is `""`, the *empty string* — walking right from 7 you never reach 3, and Python quietly gives you nothing rather than an error. `s[::5]` is `"bs2"` (indices 0, 5, 10). `s[4:-1]` is `" sky 4"` — you may mix the two rulers: start at 4, stop *before* the last character (−1 is index 10), so indices 4–9. `s[2:6:-1]` is `""` — the mirror image of `s[7:3]`: the step says walk *left* from 2, and 6 is to the right, so again you collect nothing.

D1. `43` and `"421"`. `s[9:]` is the *string* `"42"`; cast it and `+` adds, leave it and `+` joins.

E1. `2525`, then `50`, then `2525`. `n` holds the *string* `"25"`: joined with itself it is `"2525"`, repeated twice it is also `"2525"`. Only the cast line does arithmetic. Every input travels this pipeline: `str` first, number only after a cast. (The output window also shows the prompt with what you typed, `How many?  25`, above these three lines — `input()` prints its prompt before it waits.)

E2. `Lions 3`, then `Lions3`, then `Lions scored 3`. Commas insert a space between items and accept any type; `+` inserts nothing and needs every piece to be a string — hence the `str(goals)`.

E3. Line 2: `4 times 4 is 16`. Line 3 has braces but no `f`, so it prints the characters as they are: `{k} times {k} is {k * k}`. Line 4 has the `f` but no braces, so again nothing is evaluated: `k times k is k * k`. You need *both* the `f` and the braces.

E4. Just two lines: `hellohello` and `5`. Lines 1 and 3 are evaluated and thrown away — in a file, only `print()` reaches the screen. (Typed one at a time in the console, all four would show something;

that echo is a console convenience, not program output.)

E5. `c` is `3` (`7 // 2` floors). `d` is `"3333333"` — the one-character string `"3"` repeated 7 times. `e` is `7`. `f` is `"33"`: first and last character, joined. Keep asking "*what type is this now?*" at every line.

E6. `Next guess = 3.125`. `a` is the int `9`, `g` is the float `4.0`; `a / g` is `2.25`, `4.0 + 2.25` is `6.25`, half of that is `3.125`. The comma in `print` adds the space and happily prints a float. With `int(...)` on line 2, typing `4.5` crashes the program: `int("4.5")` is an error (see B1). Casting with `float()` accepts both `4` and `4.5`, which is why the guess is read that way.

F1. Line 2: `price` is a *string*, and a string cannot be repeated `1.17` times — use `total = float(price) * 1.17` (or cast on line 1 with `float(input(...))`).
Line 3: `total` is now a `float`, and you cannot glue a string to a number — use `str(total)`, or an f-string: `print(f"With tax: {total}")`.

F2. Strings are **immutable** — you cannot assign into one. Build a new string from the pieces you keep, with the new letter in between: `word = word[0] + "a" + word[2:]` (or `word[:1] + "a" + word[2:4]`), and re-bind the name to it.

F3. It prints `You got {marks} out of 50`, braces and all. Without the `f` in front, a string is just characters — the braces are not special. Fix: `print(f"You got {marks} out of 50")`.

F4. `"FCCU"` has 4 characters at indices 0–3, so index 4 is out of range. Either `code[3]` or `code[-1]` — and the second one keeps working if the string later gets longer.

F5. It printed the *empty string* `""` — an empty line. With a positive step Python walks right from 5 and never reaches 2, so it gives nothing, silently. `h` is at 2, `r` at 4, so the fix is `city[2:5]`.

F6. It printed `Speed: 30 km/h` — every comma inserts a space, wanted or not. To glue the number to the unit you must make one string out of them: `print("Speed:", str(speed) + "km/h")` or, more simply, an f-string: `print(f"Speed: {speed}km/h")`. Inside an f-string *you* decide where every space goes.

G1. The comma lets `print` take any mix of types and adds the space itself; `+` is string concatenation and refuses to join a `str` to an `int`.

G2. Nothing changed the old string: the slice built a *new* reversed string and the name `s` was re-bound to it. Immutable objects, movable names.

G3. Because a bare expression is evaluated and discarded — in a program, only `print()` writes to the screen. The console showing `1024` is a shortcut the console offers, not something the program does.

H1. `s[6:9]`. Number the letters first: `s` is at 6, `t` at 8, so the stop is 9. (Counting before slicing is the whole trick.)

H2. Any two of `s[:3]`, `s[0:3]`, `s[-10:-7]`, `s[0:3:1]`. Same three characters, described from different rulers.

H3. `[::-1]` — `"desserts"` is `"stressed"` read backwards. If you spotted that before slicing letter by letter, well done: reading the *output* carefully is half of programming.

H4. (a) `s[:4] + " " + s[4:]` — cut at index 4 and put a space between the halves. (b) `int(s[4:]) + 1` — slice off `"102"`, cast it, then add. Without the cast, `s[4:] + 1` is an error and `s[4:] + "1"` is `"1021"`.

H5. `print(f"{w} x {h} = {w * h} sq m")`. Three brace pairs, and the arithmetic happens inside the third one.

H6. `w = input("Word? ")` then `print(w + "-" + w + "-" + w)`. A shorter second line: `print((w + "-") * 2 + w)` — repeat *word-plus-dash* twice, then add the last word without a dash.

H7. `w = input("Word? ")`, then `k = int(input("Count? "))`, then `print(w * k)`, then `print(w[::-1])`. The cast on line 2 is the whole point: without it `k` is the string `"3"` and `w * k` is a `str * str` error (Part B). The word needs no cast — it is meant to stay a string.