Finger Exercises 2

Objects, Expressions & Variables

≈ 25 minutes

Comp 102 — Programming Fundamentals

What this is. A short warm-up to lock in what we just covered in class.

How to do it. On **paper**, **no computer**, in one sitting. If an item takes you more than a minute, skip it, finish the rest, then come back. The answer key is on the last page — do not peek until you are done.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Write **one example value** of each scalar type:

`int` _____ `float` _____ `bool` _____ `NoneType` _____

A2. Two operators are both called "division". `/` always gives a _____ , while `//` on two `int`s gives an _____ and throws away the _____ .

A3. Circle the right ending. `int(2.8)` works by *rounding to nearest* / *chopping off the fraction*, whereas `round(2.8)` works by *rounding to nearest* / *chopping off the fraction*.

A4. The function `sub` lives in the `operator` module. Write the single line that must run before you can call it:

_____

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

(a) `type(2.0)` reports `float`, not `int`. True / False

(b) In `total = price * 3`, Python first works out the right-hand side, then binds `total` to the result. True / False

(c) `min(4, 9, 2)` and `min(2, 9, 4)` give the same value. True / False

(d) `sub(5, 2)` and `sub(2, 5)` give the same value. True / False

(e) Once `x = 8` has run, no later line can change what `x` is bound to. True / False

(f) `None` is just another way of writing `0`. True / False

(g) `sqrt` needs an `import` line before you can use it, but `pow` does not. True / False

Part B • Value and Type

TRACE ≈ 5 min

Write what Python prints, and (where asked) the **type** of that value.

B1.

```
1 x = 10 / 5
2 print(x)
```

Prints: _____

Type: _____

B2. Whole part and leftover.

```
1 print(23 // 4)
2 print(23 % 4)
```

Output:

_____

_____

B3. Three ways to lose a fraction — watch the sign on the last one.

```
1 print(int(3.7))
2 print(round(3.7))
3 print(int(-3.7))
```

Output:

_____

_____

_____

B4. An `int` meets a `float`. Write line 3's output *exactly* as Python shows it.

```
1 print(8 - 3)
2 print(8 - 3.0)
3 print(type(8 - 3.0))
```

Output:

_____

_____

_____

B5. One star or two.

```
1 print(5 ** 2)
2 print(5 * 2)
3 print(5 ** 2 - 5 * 2)
```

Output:

_____

_____

_____

B6. Casts can be chained — work from the inside out.

```
1 print(int(float(9)))
2 print(float(int(9.99)))
```

Output:

_____

_____

B7. Same symbols, one pair of brackets.

```
1 print((8 + 4) / 2 * 3)
2 print(8 + 4 / 2 * 3)
```

Output:

_____

_____

Part C • Follow the Evaluation

TRACE ≈ 7 min

C1. Two inner calls, then one subtraction.

```
1 print(max(2, 9, 4) - min(2, 9, 4))
```

Inner calls give: ____ and ____

Prints: _____

C2. Inner calls decide which is the base and which is the power.

```
1 print(pow(min(4, 2), max(1, 3)))
```

Becomes: `pow(____, ____)`

Prints: _____

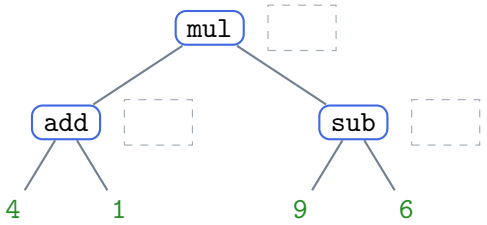
C3. Operators, dressed up as functions.

```
1 from operator import add, sub, mul
2 print(sub(mul(3, 4), add(1, 2)))
```

Becomes: `sub(____, ____)`

Prints: _____

C4. Here is the **evaluation tree** for `mul(add(4, 1), sub(9, 6))`. Work from the bottom up and write the value each call produces in the dashed box beside it. Then write the same computation using only operators.



Prints: _____

With operators only:

_____

C5. A call inside a call of the same function.

```
1 from math import sqrt
2 print(sqrt(sqrt(81)))
```

Inner call gives: _____

Prints: _____

C6. Trace table. Fill in what each name is bound to *after* each line runs. Write "–" if the name does not exist yet. Hint for line 3: right-hand side first, using the *old* value of `a`; only then re-bind `a`.

```
1 a = 3
2 b = a + 4
3 a = a * 2
4 c = a + b
```

after line	a	b	c
1	_____	_____	_____
2	_____	_____	_____
3	_____	_____	_____
4	_____	_____	_____

C7. One name is re-bound; two lines print.

```
1 rate = 4
2 pay = rate * 10
3 rate = 5
4 print(pay)
5 print(rate * 10)
```

Output:

_____

_____

Part D • Allowed, or Not Allowed?

CLASSIFY ≈ 2 min

Tick one column for each line. "Not allowed" means Python refuses it.

Line of code	Allowed	Not allowed
`total_marks = 88`	☐	☐
`88 = total_marks`	☐	☐
`score2 = 14`	☐	☐
`2nd_score = 14`	☐	☐
`my score = 14`	☐	☐
`netPay = 900`	☐	☐
`net-pay = 900`	☐	☐
`a + b = 9`	☐	☐
`ab = a + b`	☐	☐

Part E • Int or Float?

CLASSIFY ≈ 2 min

Each expression evaluates to a single value. Tick the **type** of that value — writing the value itself in the margin is good practice.

Expression	int	float
`12 / 4`	☐	☐
`12 // 4`	☐	☐
`12 % 4`	☐	☐
`3 ** 3`	☐	☐
`2.5 * 2`	☐	☐
`int(2.5 * 2)`	☐	☐
`round(6.1)`	☐	☐
`float(round(6.1))`	☐	☐
`7 - 7.0`	☐	☐

Part F • Same Value, or Different?

CLASSIFY ≈ 2 min

For each pair, tick whether the two expressions evaluate to the **same value** or **different values**.

Left	Right	Same	Different
`pow(2, 4)`	`pow(4, 2)`	☐	☐
`2 ** 3`	`3 ** 2`	☐	☐
`max(7, 1)`	`max(1, 7)`	☐	☐
`sub(9, 4)`	`sub(4, 9)`	☐	☐
`10 / 4`	`10 // 4`	☐	☐
`int(6.9)`	`round(6.9)`	☐	☐

Part G • Find and Fix

FIX ≈ 4 min

G1. This was meant to **swap** `a` and `b` using a spare name, but both end up as 5. Circle the wrong line and write its corrected version.

```
1 a = 5
2 b = 9
3 temp = b
4 b = a
5 a = b
```

Wrong line: ____

Corrected: _____

G2. This prints `12`, but the programmer wanted the perimeter for a height of `6`. Fix it by **moving one line** — do not add or repeat anything.

```
1 width = 4
2 height = 2
3 perimeter = 2 * (width + height)
4 height = 6
5 print(perimeter)
```

Move line ____

to just before line ____

New output: _____

G3. Python stops with an error when it reaches line 1. Circle the two characters that cause it, then rewrite the line correctly and give the value it prints.

```
1 average = {70 + 85 + 91} / 3
2 print(average)
```

`average` = _____

Prints: _____

G4. The import line is there, but Python still refuses to run this. Rewrite line 1 correctly, then give the output.

```
1 import sqrt from math
2 print(sqrt(144))
```

Line 1: _____

Prints: _____

G5. Two of these three names break the naming rule. Rewrite both so the code runs (the meaning must stay obvious to a reader).

```
1 1st_half = 45
2 2nd_half = 38
3 total = 1st_half + 2nd_half
```

_____

_____

Part H • One Sentence Each

EXPLAIN ≈ 2 min

H1. A friend looks at `count = count + 1` and says "that is impossible, nothing equals itself plus one". What have they misunderstood?

_____

H2. Why does `7 / 7` print `1.0` rather than `1`?

_____

H3. After `area = 3.14 * radius ** 2` runs, is `area` bound to the *formula* or to a *number*? What follows if `radius` is changed on the next line?

_____

Stuck on more than two items? Re-read the Lecture 2 summary slide before the next class — every item above rehearses one idea from it, just with different numbers.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides.

A1. Any whole number such as `-3`; any number with a decimal point such as `2.5` (even `2.0` counts); `True` or `False`; and `None` — the only value `NoneType` has.

A2. `/` always gives a **float**; `//` on two `int`s gives an **int** and throws away the **fraction** (remainder). That is why `7 / 7` is `1.0` but `7 // 7` is `1`.

A3. `int()` **chops off the fraction** (truncates), so `int(2.8)` is `2`; `round()` **rounds to the nearest** whole number, so `round(2.8)` is `3`.

A4. `from operator import sub`. Pattern: `from` *module* `import` *function*. The module comes first, the function last.

A5. (a) **True** — the decimal point makes it a `float`. (b) **True** — right side first, then bind. (c) **True** — `min` treats all its objects alike, so their order does not matter. (d) **False** — `sub(5, 2)` is `3`, `sub(2, 5)` is `-3`; for `sub` (and `pow`) the order matters. (e) **False** — a later assignment re-binds `x` to something else. (f) **False** — `None` is a different type (`NoneType`) meaning "no value at all"; `0` is an `int`. (g) **True** — `pow`, `max`, `min`, `round` are available straight away; `sqrt` has to be imported from `math`.

B1. `2.0`, a `float`. The division is exact, but `/` gives a `float` regardless.

B2. `5` then `3`. Four goes into 23 five whole times with 3 left over: $23 = 5 \times 4 + 3$. Both are `int`s.

B3. `3`, `4`, `-3`. `int()` chops the fraction off, `round()` goes to the nearest whole number. Chopping `-3.7` leaves `-3` — truncation removes the `.7`, it does not "round down" to `-4`.

B4. `5`, then `5.0`, then `<class 'float'>`. One `float` in the arithmetic makes the whole result a `float`, and that is how `type()` reports it on screen.

B5. `25`, `10`, `15`. `**` is power, `*` is multiply. On line 3 both products are worked out before the subtraction: $25 - 10$.

B6. `9` then `9.0`. Line 1: `float(9)` is `9.0`, then `int(9.0)` is `9`. Line 2: `int(9.99)` chops to `9`, then `float(9)` is `9.0`. The *outer* cast decides the final type.

B7. `18.0` then `14.0`. Line 1: brackets first, $12/2 = 6.0$, then 6.0×3. Line 2: `/` and `*` go left to right before `+`, so $4/2 = 2.0$, $2.0 \times 3 = 6.0$, then $8 + 6.0$. Both are `float`s because a `/` happened along the way.

C1. `max(2, 9, 4)` is `9`, `min(2, 9, 4)` is `2`, so the line prints $9 - 2 = 7$. Each call collapses to a single value before the `-` gets to work.

C2. `min(4, 2)` is `2` and `max(1, 3)` is `3`, so this is `pow(2, 3)` → `8`. Had the inner calls come out the other way round, `pow(3, 2)` would give `9` — order matters for `pow`.

C3. `mul(3, 4)` is `12`, `add(1, 2)` is `3`, so this is `sub(12, 3)` → `9`. Written with operators it is `3 * 4 - (1 + 2)`.

C4. `add(4, 1)` → `5`; `sub(9, 6)` → `3`; `mul(5, 3)` → `15`, which is what prints. With operators: `(4 + 1) * (9 - 6)`. The brackets are essential — without them `4 + 1 * 9 - 6` is `7`.

C5. `sqrt(81)` is `9.0`, then `sqrt(9.0)` is `3.0`. A `float` both times — `sqrt` never hands back an `int`, even for perfect squares.

C6. Row by row (`a`, `b`, `c`): after line 1: `3`, –, –; after line 2: `3`, `7`, –; after line 3: `6`, `7`, –; after line 4: `6`, `7`, `13`. Notice `b` stays `7` when `a` changes — it was bound to the *value* `7`, not to the formula `a + 4`.

C7. `40` then `50`. `pay` was worked out when `rate` was 4 and nobody asked Python to redo it, so it still holds `40`. Line 5 works out `rate * 10` *afresh*, with the current `rate` of 5.

D. **Allowed:** `total_marks = 88`, `score2 = 14` (a digit is fine as long as it is not *first*), `netPay = 900`, `ab = a + b` (assuming `a` and `b` were bound earlier; `ab` is just a two-letter name).
Not allowed: `88 = total_marks` (the name goes on the left), `2nd_score = 14` (starts with a digit), `my score = 14` (no spaces inside a name), `net-pay = 900` (Python reads it as `net`

minus `pay`), `a + b = 9` (the left side must be a single name, not an expression).

E. `int:` `12 // 4` (3), `12 % 4` (0), `3 ** 3` (27), `int(2.5 * 2)` (5), `round(6.1)` (6).
`float:` `12 / 4` (3.0 — `/` always gives a float), `2.5 * 2` (5.0 — the answer is whole, the type is not), `float(round(6.1))` (6.0 — the outer cast wins), `7 - 7.0` (0.0 — one float in the mix is enough).

F. **Same:** `pow(2, 4)` and `pow(4, 2)` — both 16! Order *usually* matters for `pow`, but this pair happens to coincide, which is why you should work it out rather than guess. `max(7, 1)` and `max(1, 7)` — both 7; order never matters for `max` or `min`.
Different: `2 ** 3` is 8, `3 ** 2` is 9. `sub(9, 4)` is 5, `sub(4, 9)` is −5. `10 / 4` is 2.5, `10 // 4` is 2. `int(6.9)` is 6, `round(6.9)` is 7.

G1. Line 5. By then `b` has already been re-bound to 5, so `a = b` just copies 5 back. The saved 9 is sitting in `temp`: the line should read `a = temp`.

G2. Move line 4 (`height = 6`) to just before line 3, so the perimeter is worked out *after* the height is 6. Output becomes `20`. (Simply deleting line 2 also works.) Python computes `perimeter` once, at the moment line 3 runs, and never revisits it.

G3. Circle the braces `{ }`. Only parentheses group an expression: `average = (70 + 85 + 91) / 3`, which prints `82.0` (a `float`, because of the `/`). Braces and square brackets build other kinds of object in Python, and those cannot be divided by 3.

G4. `from math import sqrt` — the module comes after `from`, the function after `import`. Then the code prints `12.0`.

G5. Names may not start with a digit. For example `first_half` and `second_half` (or `half_1` and `half_2`) — and the same new names must be used on line 3. `total` was already fine.

H1. They are reading `=` as mathematical equality. In Python it is an **assignment**: the right side is worked out first (the *old* `count` plus one) and the name `count` is then re-bound to that new value.

H2. Because `/` *always* produces a `float`, whether or not the division comes out exact. If you want the `int` 1, use `7 // 7`.

H3. To a **number** — the formula is evaluated once and only the result is kept. Changing `radius` afterwards leaves `area` untouched; to update it you must run the assignment again.