Finger Exercises 4

Boolean Expressions

≈ 45 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**. If an item takes you more than a minute, skip it, finish the rest, then come back. Parts **A–E** are the core; Parts **F–K** go deeper into the places where two ideas meet — and those are exactly the places where next week's `if` statements go wrong. Two sittings is fine: A–E now, F–K before the next class. The Console Quests 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. `type(3 < 4)` evaluates to _____ . That type has exactly two values: _____ and _____ .

A2. Write the comparison operator that means each phrase:

"is not equal to" _____ "is at least" _____ "is at most" _____ "is equal to" _____

A3. In each pair, circle the operator Python applies **first**:

`not` / `and` `and` / `or` `>` / `+` `==` / `not`

A4. A value that `bool()` turns into `False` is called _____ . Apart from `False` itself, the three such values we met are _____ , _____ and _____ .

A5. Write the **opposite** of each operator — the one that is `True` exactly when the first is `False`:

opposite of `>` is _____ opposite of `<=` is _____ opposite of `==` is _____ opposite of `>=` is _____

A6. `input()` always returns a value of type _____ , no matter what the user types. To compare it with a number you must first _____ .

A7. Circle **True** or **False**:

(a) `7 >= 7` evaluates to `True`. True / False

(b) `false` (small `f`) is just another way to write `False`. True / False

(c) `"Zebra" < "apple"` evaluates to `True`. True / False

(d) `not 3 == 3` evaluates to `False`. True / False

(e) Running `x == 5` changes what `x` is bound to. True / False

(f) `bool("0")` evaluates to `False`. True / False

(g) `0.1 + 0.2 == 0.3` evaluates to `True`. True / False

(h) `-3 > -5` evaluates to `True`. True / False

(i) `not (5 > 5)` and `not (5 < 5)` are *both* `True`. True / False

(j) `"20" > "9"` evaluates to `True`. True / False

Part B • True or False?

TRACE ≈ 5 min

Write what each line prints. Apply the order of operations, not left-to-right reading.

	Line	Prints
B1.	`print(not False and False)`	
B2.	`print(not (False and False))` (only the brackets changed)	
B3.	`print(True or True and False)`	
B4.	`print(4 + 1 > 3 * 2)`	
B5.	`print(10 - 4 == 2 * 3 and not 1 > 2)`	
B6.	`print(len("comp") > 4 or 2 ** 3 == 8)`	
B7.	`print(-3 > -5 and -5 < 0)`	
B8.	`print(not -1 > 0)`	
B9.	`print(abs(-6) % 3 == 0 or -6 > 0)`	

Part C • One Step at a Time

TRACE ≈ 6 min

C1. Fill in each step of `8 // 3 < 3 or not 2 ** 2 == 4`:

Step 1 (arithmetic): _____

Step 2 (comparisons): _____

Step 3 (`not`): _____

Step 4 (`or`): _____

C2. Let `m = 9` and `n = 4`. Write `True` or `False` for each:

`m % n == 1` _____

`m > 8 and n > 4` _____

`not (m == n) or n > m` _____

`m - n >= 5 and m / n > 2` _____

C3. Complete the truth table. The two right-hand columns differ only by a pair of brackets.

A	B	not A and B	not (A and B)
True	True		
True	False		
False	True		
False	False		

C4. Write one operator in each gap so that the line has the value shown on the right:

`9 _____ 12` is `True` _____

`3 * 4 _____ 12` is `True` _____

`False _____ True` is `True` _____

`8 _____ 8` is `False` _____

C5. Function calls are expressions too, so they can sit on either side of a comparison. `True` or `False`?

`max(3, 8) == min(8, 12)` _____

`abs(2 - 9) > 7` _____

`max(2, 9) - min(2, 9) == 7` _____

C6. Add brackets to `not 2 > 3 or 1 == 1 and False` so they show exactly how Python groups it, then give its value.

With brackets: _____

Value: _____

C7. Booleans can be stored in variables like any other value. Write the three lines this program prints.

```
1 sunny = True
2 free = False
3 go = sunny and free
4 print(go)
5 print(not go or sunny)
6 print(sunny != free)
```

Output:

_____

_____

_____

Part D • Truthy or Falsy? CLASSIFY ≈ 4 min

Tick what each `bool()` call returns. Work out the value inside the brackets first.

Expression	True	False
`bool(7)`	☐	☐
`bool(0 * 5)`	☐	☐
`bool(-0.5)`	☐	☐
`bool("0")`	☐	☐
`bool(len(""))`	☐	☐
`bool("None")`	☐	☐
`bool(3 > 5)`	☐	☐
`bool(1 - 1)`	☐	☐
`bool(0.0)`	☐	☐
`bool(" ")`	☐	☐

Now without the `bool()`. Python applies the same truthy/falsy rule whenever a value meets `not`, `and` or `or` — no cast needed. Write `True` or `False`.

D1. `not 0` _____

D2. `not 5` _____

D3. `not ""` _____

D4. `not "0"` _____

D5. `not None` _____

D6. `not not 7` _____

D7. `not len("")` _____

D8. `not (3 - 3)` _____

Part E • Comparing Strings

CLASSIFY ≈ 2 min

Strings compare **character by character**, and every capital letter sorts before every lowercase one. Write `True` or `False`.

E1. `"cat" == "Cat"` _____

E2. `"cat" != "cat "` _____

E3. `"bat" > "bay"` _____

E4. `"Zoo" < "ant"` _____

E5. `"10" < "9"` _____

E6. `len("abc") == "3"` _____

Part F • Type Traps

CLASSIFY ≈ 4 min

Comparison across two different types is where beginner programs die. Two rules, and they do *not* match: `==` between different types quietly answers `False`; `< > <= >=` between different types **crash**.

F1. Write the value, or the word **ERROR** if the line crashes:

`"5" == 5` _____

`5 == 5.0` _____

`"5" < 6` _____

`int("5") < 6` _____

F2. The user types **20** at the prompt. Write what each line does — a value, or **ERROR**.

```
1 age = input("Age: ")
2 print(age == 20)
3 print(age == "20")
4 print(age >= 18)
```

Line 2: _____

Line 3: _____

Line 4: _____

F3. `grade = "85"` (note the quotes). Write `True`, `False` or **ERROR**:

`grade == 85` _____

`grade > "9"` _____

`int(grade) > 9` _____

Part G • Where Is the Boundary?

TRACE ≈ 4 min

Knowing an expression is `True` for one value is easy. Finding the exact value where it *switches* is the skill that stops off-by-one bugs. Assume `x` is a whole number.

G1. Fill in the switching value:

`x >= 10` — smallest `x` that is `True`: _____

`x > 10` — smallest `x` that is `True`: _____

`x < 7` — largest `x` that is `True`: _____

`x <= 7` — largest `x` that is `True`: _____

G2. How many whole numbers make each one `True`?

`marks >= 50 and marks <= 59` _____

`x > 0 and x < 1` _____

`x > 3 and x < 5` _____

G3. `fee = age < 12 or age >= 60`. Tick every age that gets the discount:

☐ 11 ☐ 12 ☐ 59 ☐ 60 ☐ 61

Part H • Saying the Opposite

TRANSLATE ≈ 6 min

Every `if` you write next week has an `else`, and the `else` is the opposite of the condition. This part is that skill.

H1. Rewrite each condition as its opposite, **without using** `not`:

Condition	Its opposite
`x > 5`	_____
`x <= 0`	_____
`x == 7`	_____
`x >= 18`	_____
`name != "Ali"`	_____

H2. A student claims the opposite of `x > 5` is `x < 5`. Give one value of `x` for which *both* are `False`, and say why that settles it.

`x =` _____ because _____

H3. Flip both pieces and the joiner. To negate an `and`/`or` expression, negate each side *and* swap `and` ↔ `or`. Rewrite without an outer `not`:

```
not (x >= 1 and x <= 5)
```
→ _____

```
not (day == "Sat" or day == "Sun")
```
→ _____

```
not (temp > 30 and raining)
```
→ _____

H4. Check the rule with a truth table. If the two right-hand columns match on all four rows, the rule holds.

A	B	not (A or B)	not A and not B
True	True	_____	_____
True	False	_____	_____
False	True	_____	_____
False	False	_____	_____

Part I • Find and Fix

FIX ≈ 6 min

I1. `passed` should be `True` only when `marks` is between 50 and 100 inclusive. With `marks = 30` it prints `True`. Circle the wrong operator and write its replacement.

```
1 marks = 30
2 passed = marks >= 50 or marks <= 100
3 print(passed)
```

Replace with:

_____

I2. Python stops with a `NameError` on line 1. Circle the culprit and write the corrected line.

```
1 ready = TRUE and 5 > 3
2 print(ready)
```

Fixed: _____

I3. This should print `True` because 7 is *not* between 1 and 5 — but it prints `False`. Add **one pair of brackets** to line 2 to fix it.

```
1 x = 7
2 outside = not x >= 1 and x <= 5
3 print(outside)
```

`outside =`
_____

I4. `night` should be `True` before 6 a.m. or after 10 p.m. — so for `hour = 3` it should print `True`. It prints `False`, and in fact it prints `False` for *every* hour. Circle the wrong operator and write its replacement.

```
1 hour = 3
2 night = hour < 6 and hour > 22
3 print(night)
```

Replace with:
_____

I5. The one that does not crash. `weekend` should be `True` only for day 6 or day 7. With `day = 3` you expect `False` — but it prints `7`. Rewrite line 2.

```
1 day = 3
2 weekend = day == 6 or 7
3 print(weekend)
```

`weekend =` _____

I6. The user types **25**, and the program stops with `TypeError` on line 2. The comparison is what you want — fix line 1.

```
1 age = input("Age: ")
2 adult = age >= 18
3 print(adult)
```

`age =` _____

Part J • English to Python

TRANSLATE ≈ 5 min

Let `temp = 30` and `raining = False`. For each phrase, write a boolean expression using those names, then its value.

J1. "temp is at least 25 and it is not raining"

_____ value: _____

J2. "temp is below 10 or above 35"

_____ value: _____

J3. "temp is an even number"

_____ value: _____

J4. "it is raining, or temp is exactly 30"

_____ value: _____

J5. "temp is NOT above 35"

_____ value: _____

J6. "temp is within 5 of 25"

_____ value: _____

J7. "temp is below freezing" (freezing is 0)

_____ value: _____

J8. "temp is NOT between 20 and 30, ends included"

_____ value: _____

J9. "it is not raining, and temp is not below 20"

_____ value: _____

J10. "temp is odd"

_____ value: _____

Part K • One Sentence Each

EXPLAIN ≈ 4 min

K1. Does Python read `not x > 5` as `(not x) > 5` or as `not (x > 5)`? Why?

K2. Why is `"100" < "20"` `True`, when 100 is obviously bigger than 20?

K3. A friend writes `score = 90 == 90` and is surprised that `score` is `True` rather than `90`. Explain.

K4. `0.1 + 0.2 == 0.3` is `False`, yet `1 + 2 == 3` is `True`. What is different about the first line?

K5. A friend writes `grade == "A" or "B"` and finds it is truthy for *every* grade. Explain in one sentence.

K6. Why does `"5" == 5` quietly give `False`, while `"5" < 6` stops the program?

K7. Why is the opposite of `x >= 18` written `x < 18` rather than `x <= 18`?

Console Quests

tonight · 8 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 `0.1 + 0.2 == 0.3`, then `0.1 + 0.2`, then `0.5 + 0.25 == 0.75`. Why might the last one behave differently from the first?
2. Strings of different lengths: `"apple" < "apple pie"`, `"" < "a"`, `"Z" < "a"`, `"z" < "A"`. What happens when one string runs out of characters first?
3. Mixed types: `5 == 5.0`, `"5" == 5`, `bool(-0.0)`, `bool("False")`. Which of these surprised you?
4. **Booleans are secretly numbers.** Try `0 == False`, `1 == True`, `2 == True`, and then `True + True`. What number does each boolean seem to stand for? (You will not need this for a while — but it explains a lot of odd output.)
5. **Python allows a shortcut we did not teach.** Try `1 <= 5 <= 10`, then `5 < 3 < 9`, then `3 < 5 > 4`. Does chaining behave like the `and` version you would have written by hand? Try to break it before you trust it.
6. **Does Python always look at both sides?** Type `True or (1 / 0 == 1)` — then `False or (1 / 0 == 1)`. One of them crashes and one does not. What does that tell you about when `or` bothers to read its right-hand side? Now predict which of `False and (1 / 0 == 1)` and `True and (1 / 0 == 1)` crashes, before typing them.

7. **The minus-sign trap.** Predict `-2 ** 2`, then type it. Now try `(-2) ** 2`. Which one did you mean?

Stuck on more than two items? Re-read the Lecture 4 order-of-operations slide and the truth tables before the next class — every item above rehearses one of them, with different numbers. **If Parts F–K were the hard ones**, that is expected and it is the part worth repeating: those cover the seams between two ideas — negating a condition, truthiness without `bool()`, comparing across types, and finding the exact value where a condition switches. Next week every `if` statement you write stands on them.

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. `bool`; `True` and `False` (capital first letter, no quotes). A comparison *is* an expression, so it has a type like any other.

A2. `!=`, `>=`, `<=`, `==`. "At least 18" includes 18, hence the `=` half of `>=`; "at most" likewise.

A3. `not` before `and`; `and` before `or`; `+` before `>`; `==` before `not`. Full order: arithmetic → comparison → `not` → `and` → `or`.

A4. **falsy**. `0` (and `0.0`), the empty string `""`, and `None`. Everything else — every other number, every non-empty string — is truthy.

A5. `<=`, `>`, `!=`, `<`. The opposite of `>` is `<=`, **not** `<` — the "equal" case has to land somewhere. Each operator pairs with the one that covers everything it leaves out.

A6. `str`. You must **cast** it, with `int()` or `float()`. Typing 20 at an `input()` prompt gives you the string `"20"`, not the number 20 — and `"20"` is not equal to `20`.

A7. (a) **True** — the "or equal" half of `>=` is satisfied. (b) **False** — `false` is an undefined name; Python raises a `NameError`. (c) **True** — every capital letter sorts before every lowercase letter, so `"Z"` comes before `"a"`. (d) **True** — the comparison runs first (`True`), then `not` flips it. (e) **False** — `==` only *asks*; it never assigns. (f) **False** — `"0"` is a non-empty *string*, so it is truthy. Only the number 0 is falsy. (g) **False** — `0.1 + 0.2` comes out as `0.30000000000000004`. Floats are stored approximately, so `==` between computed floats is a trap. (h) **True** — on the number line -3 sits to the *right* of -5. The bigger the digits after the minus sign, the *smaller* the number. (i) **True** — and this is why `<` is not the opposite of `>`. At `5`, both `5 > 5` and `5 < 5` are `False`, so `not` makes both `True` at once. The real opposite of `>` is `<=`. (j) **False** — these are *strings*. Comparison stops at the first character: `"2"` comes before `"9"`, so `"20"` is the smaller string. As numbers, `20 > 9` would be `True`.

B1. `False`. `not` binds first: `not False` is `True`, then `True and False` is `False`.

B2. `True`. The brackets force `False and False` first, giving `False`, and `not False` is `True` — the opposite of B1, from two brackets.

B3. `True`. `and` binds tighter than `or`, so Python reads `True or (True and False)` = `True or False` → `True`. Reading left to right, `(True or True) and False`, would wrongly give `False`.

B4. `False`. Arithmetic first on *both* sides: `5 > 6`, which is `False`.

B5. `True`. Arithmetic: `6 == 6 and not 1 > 2`. Comparisons: `True and not False`. `not`: `True and True`. Finally `True`.

B6. `True`. `len("comp")` is 4, so `4 > 4` is `False`; `2 ** 3 == 8` is `True`; `False or True` is `True`. Call expressions and powers are worked out before any comparing.

B7. `True`. `-3 > -5` is `True` (further right on the number line) and `-5 < 0` is `True`, so `True and True`.

B8. `True`. No brackets needed: the comparison `-1 > 0` runs first and is `False`, then `not` flips it. The minus sign belongs to the *value* `-1`, not to the comparison.

B9. `True`. `abs(-6)` is 6 and `6 % 3` is 0, so the left side is `True`; `or` is satisfied and the right side (`False`) cannot undo it.

C1. Step 1: `2 < 3 or not 4 == 4`. Step 2: `True or not True`. Step 3: `True or False`. Step 4: `True`.

C2. `True` (9 % 4 is 1); `False` (`True and False` — 4 is not *greater than* 4); `True` (`m == n` is `False`, so `not` makes it `True`, and `or` needs only one side); `True` (`5 >= 5` and `2.25 > 2` are both `True`).

C3. `not A and B` (top to bottom): `False`, `False`, `True`, `False` — it is `True` only when A is off and B is on. `not (A and B)`: `False`, `True`, `True`, `True` — it is `False` only when both are on. Same words, one bracket pair, three rows different.

C4. `9 < 12` (also `<=` or `!=`); `3 * 4 == 12` (also `>=` or `<=`, since 12 is equal to 12); `False or True` (also `!=`); `8 != 8` (also `<` or `>`) — anything that denies 8 is 8.

C5. `True` (both sides are 8); `False` (`abs(-7)` is 7, and 7 is not *greater than* 7); `True` (`9 - 2 == 7`). Work out every call first, then compare — a call is just a value waiting to happen.

C6. `(not (2 > 3)) or ((1 == 1) and False)`. Comparisons first, then `not` wraps only the `2 > 3`, then `and` groups before `or`. Value: `True or False` → `True`.

C7. `False`, then `True`, then `True`. `go` is bound to the *result* of `True and False`, i.e. `False`. Then `not go` is `True`, and `or` needs only one side. Finally `True != False` is `True` — booleans can be compared

with == and != like any other values.

D. (with `bool()`) **True, False, True, True, False, True, False, False, False, True.** Numbers: only zero is falsy, so `0 * 5`, `1 - 1` and `0.0` are `False` while `7` and `-0.5` are `True`. `len("")` is the number 0 — `False`. Strings: any non-empty string is truthy, so `"0"` and `"None"` are both `True` — what the characters *spell* is irrelevant — and `" "` is `True` too: a space is a character, so the string is not empty. `bool(3 > 5)` is `bool(False)`, which is `False`.

D. (without `bool()`) **True, False, True, False, True, True, True, True.** Read `not x` as "is `x` falsy?". So `not 0` is `True` (0 is falsy) while `not 5` is `False`. `""` is falsy but `"0"` is a non-empty string, so `not "0"` is `False`. `not not 7` flips twice and lands back on `True` — it is `bool(7)` written the long way. `len("")` and `3 - 3` are both the number 0. **Why this matters:** next week you will write `if name:` and `if count:` with no `bool()` anywhere in sight — this is the rule that decides whether those run.

E. **False, True, False, True, True, False.** (1) case matters. (2) the second string has a trailing space, so they differ. (3) `b`, `a` tie; then `t` comes before `y`, so `"bat"` is the *smaller* one. (4) capital `Z` sorts before lowercase `a`. (5) these are *strings*, not numbers: `"1"` comes before `"9"`, so `"10"` is smaller. (6) the left side is the *number* 3, the right side the *string* `"3"` — a number is never equal to a string, and Python does not convert one to the other for you.

F1. `False` — a string is never equal to a number, and Python will not convert for you. `True` — `int` and `float` are both numbers, so 5 and 5.0 really are equal. **ERROR** (`TypeError`) — Python refuses to say whether a string is "less than" a number; there is no sensible answer. `True` — once you cast, it is 5 against 6. Note the asymmetry: == across types is *safe but always False*, ordering across types is a *crash*.

F2. `False`, then `True`, then **ERROR** (`TypeError`). `input()` hands back the *string* `"20"`, never the number. So it is not equal to `20`, it *is* equal to `"20"`, and asking whether it is at least 18 crashes. The fix is on the input line, not the comparison line: `age = int(input("Age: "))`. Expect this one in your first `if` program.

F3. `False` (string versus number); `False` — both sides are strings, so this is legal, but comparison stops at the first character: `"8"` comes before `"9"`, so `"85"` is the *smaller* string; `True` — as numbers, 85 really is bigger than 9. A string comparison that runs is not the same as a string comparison that is *right*.

G1. **10**, **11**, **6**, **7**. The whole difference between > and >= is one single value — and that one value is exactly the mark, the age or the price a user is most likely to type.

G2. **10** (50 up to 59 inclusive — count the ends, that is the classic off-by-one); **0** — no whole number sits strictly between 0 and 1, so the test can never fire for an `int` (it is fine for a `float` such as 0.5); **1** — only 4.

G3. **11, 60 and 61.** 12 misses because < excludes it; 59 misses because the cutoff is 60. Always test a boundary from *both* sides — one value below and one value above — rather than picking a number in the middle where every version looks correct.

H1. `x <= 5`, `x > 0`, `x != 7`, `x < 18`, `name == "Ali"`. Each operator pairs with the one covering everything it leaves out. The opposite of > is <=, **never** < — see A7(i).

H2. `x = 5`. Then `5 > 5` is `False` and `5 < 5` is `False` too — so they cannot be opposites, because true opposites must disagree on *every* value. The missing case is exactly `x == 5`, which <= sweeps up.

H3. `x < 1 or x > 5` — "not inside the range" is "below it *or* above it". `day != "Sat" and day != "Sun"` — "not a weekend day" is "not Saturday *and* not Sunday". English uses "and" here, which is exactly why students write `or` and get it backwards. `temp <= 30 or not raining`. In all three: flip each piece, flip the joiner.

H4. Both columns read `False`, `False`, `False`, `True` — identical on every row, so the rule holds. `not (A or B)` is `True` only when *neither* is on. (The matching rule is `not (A and B)` = `not A or not B`; you filled that left-hand column in C3.)

I1. Replace `or` with `and`. Every number is either at least 50 *or* at most 100 (usually both), so the `or` version is `True` for any `marks`. Being *inside* a range needs both conditions at once.

I2. `ready = True and 5 > 3`. Only `True` and `False`, capital first letter, are booleans; `TRUE` is an unknown name. Fixed, it prints `True`.

I3. `outside = not (x >= 1 and x <= 5)`. As written, `not` grabs only `x >= 1`: that is `not True` = `False`, and `False and ...` is `False` whatever follows. The brackets make `not` apply to the whole "inside the range" test. (`x < 1 or x > 5` says the same thing without `not`.)

I4. Replace `and` with `or`. No number is below 6 *and* above 22 at the same time, so the `and` version can never be `True`. Being *outside* a range is one condition *or* the other — the mirror image of I1, where

`or` made "inside" always `True`.

I5. `weekend = day == 6 or day == 7`. **Each side of `or` needs its own complete comparison** — you must repeat the variable. As written, Python groups it as `(day == 6) or 7`: the left side is `False`, so `or` hands back the right side, the bare number `7`. That is why it prints `7` instead of `True` or `False`. And `7` is *truthy* (Part E), so next week an `if` built on this would run for **every** day of the week. No error message, no crash, just silently always on — the most expensive bug on this sheet.

I6. `age = int(input("Age: "))`. `input()` returns the string `"25"`, and Python will not order a string against a number. Cast at the *source* — do not patch the comparison into `age >= "18"`, which would run but compare character by character and give wrong answers (try `"9" >= "18"`).

J1. `temp >= 25 and not raining` → `True`. Note `not raining`, not `raining == False` — both work, the first reads like English.

J2. `temp < 10 or temp > 35` → `False`. Each side needs its own `temp`; `temp < 10 or > 35` is not Python.

J3. `temp % 2 == 0` → `True`. Even means "remainder 0 when divided by 2" — remember this one, you will use it constantly.

J4. `raining or temp == 30` → `True`. `raining` is already a boolean, so it can stand alone; "exactly" means `==`, never a single `=`.

J5. `not (temp > 35)` → `True`. Equally good: `temp <= 35` — "not above" means "at most". If you write `not temp > 35` without brackets it still works (comparison first), but the brackets say what you mean.

J6. `abs(temp - 25) <= 5` → `True` (the gap is exactly 5, and "within" includes the edge). Also fine: `temp >= 20 and temp <= 30`. One call to `abs` replaces a two-sided range check.

J7. `temp < 0` → `False`. "Below freezing" means negative. Watch the direction: `-5 < 0` is `True`, so colder really is smaller.

J8. `temp < 20 or temp > 30` → `False` (30 *is* in the range, so "not between" is false). Equally correct: `not (temp >= 20 and temp <= 30)`. Going from the second form to the first is Part H's rule — flip both pieces, flip the joiner.

J9. `not raining and temp >= 20` → `True`. Also fine: `not raining and not (temp < 20)`. Two English "not"s joined by "and" stay an `and` — it is only when you negate the *whole* expression that the joiner flips.

J10. `temp % 2 == 1` → `False` (30 is even). Also fine: `temp % 2 != 0`, or `not (temp % 2 == 0)`. Three spellings of one idea — pick the one that reads most like the English.

K1. As `not (x > 5)`: comparisons come before `not` in the order of operations, so the comparison is worked out first and `not` flips its result.

K2. Because they are strings, not numbers: comparison goes character by character, and the first characters `"1"` and `"2"` already decide it. Cast with `int()` if you mean the numbers.

K3. The right-hand side is the boolean expression `90 == 90`, which evaluates to `True` first; the single `=` then binds `score` to that `True`. Assignment always stores the *value* of the whole right side.

K4. The first line uses *floats*, which the computer stores only approximately; `0.1 + 0.2` lands on `0.30000000000000004`, a hair away from `0.3`. Integers are exact, so `==` on them is always safe.

K5. Python reads it as `(grade == "A") or "B"`, and when the comparison is `False` the `or` hands back the bare string `"B"` — which is non-empty and therefore truthy, so the test never fails. Each side of `or` needs its own full comparison: `grade == "A" or grade == "B"`.

K6. "Are these the same value?" always has an answer, and for two different types that answer is simply no. But "does this string come before this number?" has no meaning at all, so instead of inventing one Python raises a `TypeError`. The crash is the friendlier of the two — the silent `False` is the one that hides in your program.

K7. Because `x = 18` already belongs to the original condition, so the opposite must *exclude* it; `x <= 18` would claim 18 for both sides at once. A condition and its opposite must divide every possible value between them — each value landing on exactly one side, never both and never neither.