Module 2 of 20

Values, types and operators

Arbitrary-precision integers and floor division, floating point and the rules for comparing and printing it, truthiness and what and/or return, names bound to objects, explicit conversions and their errors, and operator precedence with the bitwise idioms.

Python's values are objects and its variables are names, and the whole module is a consequence of taking those two facts seriously. An integer never overflows because the object grows; a float is a binary double and `0.1 + 0.2` shows it; every object is truthy or falsy and `and` / `or` hand back an operand rather than a boolean; assignment binds a name and never copies, so whether a change is shared depends on whether the object is mutable; and nothing converts between text and numbers unless you call the type. Learners who arrive from C or Java carry a box model and a truncating division, and this module replaces both.

The six lessons take the built-in scalar types in turn. Numbers fixes the semantics of `/`, `//`, `%` and `**`, the half-to-even `round`, and the `math` integer functions. Floating point explains representation error once and gives the three rules — `math.isclose` instead of `==`, format on output, `fsum` or integer cents when accumulating — plus `Decimal` and `Fraction` for exactness. Booleans and truthiness gives the table, the operand-returning `and` / `or`, `None` and `is`. Names and binding builds the rebinding-versus-mutation model that Module 6 relies on. Conversions lists what `int`, `float`, `str` and the container constructors accept and raise. Operators and precedence gives the table and the bitwise idioms on unbounded integers.

The exercises are whole programs that make each rule observable: floor versus truncating division across every sign combination, a big-power report, a three-way sum that shows `sum`, `fsum` and `Decimal` disagreeing, money kept in cents, a truthiness classifier over Python literals, chained comparisons with missing values, a tiny binding machine that distinguishes aliases from copies, a walrus-driven reader, a token classifier, a base converter, a permission mask, and an evaluator that contrasts precedence with left-to-right reading. The checkpoint adds a digit-and-bit report, a formatted temperature table, and an exact-change calculator.

CONTENTS

1 Numbers — int, float and the arithmetic that surprises 13 min

2	Floating point — why 0.1 + 0.2 is not 0.3, and what to do about it	14 min
3	Booleans, truthiness, None, and is versus ==	13 min
4	Names, binding and mutability — there are no boxes	14 min
5	Conversions — between text, numbers and containers	12 min
6	Operators and precedence	13 min
✓	Checkpoint — Values, types and operators	25 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 13 MIN

Numbers — int, float and the arithmetic that surprises

Python has three numeric types and one of them is unlike anything in C or Java: `int` has no upper limit. `2 ** 1000` is an ordinary integer, factorials of hundreds do not overflow, and the questions that fill C++ interviews — "does this product fit in 64 bits?" — simply do not arise. What does arise is the *semantics* of division and remainder, which Python defines carefully and differently from most languages, and the difference between the exact `int` and the approximate `float`. This lesson settles the integer side; the next one is about floating point.

The integer type

An `int` literal is a run of digits, optionally with underscores for readability and a prefix for another base:

```
million = 1_000_000
mask = 0b1010_1010      # binary
perm = 0o755            # octal
colour = 0xFF_80_00     # hexadecimal
print(million, mask, perm, colour)   # 1000000 170 493 16744448
```

Integers are arbitrary-precision: the object grows as the value needs. `2 ** 64` is `18446744073709551616`, `10 ** 100` has a hundred zeros, and arithmetic on them is exact — slower than machine integers (roughly proportional to the number of digits), but correct. `sys.maxsize` exists (the largest size a container can have, $2^{63} - 1$ on 64-bit builds), but it is not a limit on `int`.

Division, three ways

Expression	Result	Type	Rule
`7 / 2`	`3.5`	`float`	true division, always a float
`7 // 2`	`3`	`int`	floor division: round **toward negative infinity**
`7 % 2`	`1`	`int`	remainder, with the **sign of the divisor**
`-7 // 2`	`-4`	`int`	floor of −3.5 is −4
`-7 % 2`	`1`	`int`	−7 = 2 × (−4) + 1
`7 % -2`	`-1`	`int`	7 = (−2) × (−4) + (−1)

Two things here differ from C, Java and JavaScript. `/` on two integers is a `float` — `6 / 3` is `2.0`, not `2` — so an index or a count computed with `/` is a bug; use `//`. And `//` floors rather than truncating, which makes `%` take the divisor's sign. The invariant `a == b * (a // b) + a % b` holds for every sign combination, and the practical payoff is that `x % n` for positive `n` is always in `0 … n-1`: wrapping an index with `(i - 1) % len(xs)` gives `len(xs) - 1` for `i == 0`, exactly the "previous element, cyclically" that a truncating language needs a special case for. `divmod(a, b)` returns both `(a // b, a % b)` in one call.

Powers and the rest

`**` is exponentiation and binds tighter than unary minus: `-2 ** 2` is `-4` (it parses as `-(2 ** 2)`); write `(-2) ** 2` for 4. `2 ** -1` is `0.5` — a negative exponent makes a float. `pow(base, exp, mod)` computes `base ** exp % mod` efficiently by modular exponentiation and is the right tool whenever a result is wanted modulo something: `pow(2, 10 ** 6, 10 ** 9 + 7)` finishes instantly where `2 ** 10 ** 6 % m` would first build a 300 000-digit number.

`abs(x)`, `min`, `max`, `sum` work as expected; `round(x, n)` rounds to `n` decimals and, with no `n`, returns an `int`. Its tie rule surprises people: `round(2.5)` is `2` and `round(3.5)` is `4` — **round half to even** ("banker's rounding"), which avoids the upward bias of always rounding halves up. `int(x)` on a float truncates toward zero (`int(-3.7)` is `-3`); `math.floor` and `math.ceil` go down and up and return ints.

The `math` module supplies the integer functions interviews use — `math.gcd(a, b)`, `math.lcm(a, b)`, `math.isqrt(n)` (exact integer square root, never a float), `math.comb(n, k)`, `math.perm(n, k)`, `math.factorial(n)` — and `math.sqrt`, `math.log`, `math.log2`,

`math.log10` return floats. `math.isqrt(n) ** 2 == n` is the exact test for a perfect square; `int(math.sqrt(n))` is not, for large `n`.

Mixed arithmetic

Arithmetic between an `int` and a `float` produces a `float`; the int is converted first. `3 + 0.5` is `3.5`; `10 * 1.0` is `10.0`. A `bool` is a subtype of `int` (`True` is 1), so `True + True` is `2` and `sum(flags)` counts true values — an idiom, not a trick. `complex` completes the tower: `1j` is the imaginary unit, `(1 + 2j) * (3 - 1j)` is `(5+5j)`, and `abs` of a complex number is its magnitude. Between numbers Python converts up the tower `int → float → complex` and never the other way without an explicit call.

Reading and printing numbers

`int("42")` parses decimal digits with optional sign and surrounding whitespace, and accepts underscores; `int("42", 16)` parses in another base; `int("0x2a", 0)` picks the base from the prefix. `int("3.0")` is a `ValueError` — parse the float and truncate if that is what you mean. `str(n)` and `f"{n}"` print in decimal; `bin`, `oct`, `hex` print with prefixes (`bin(10)` is `'0b1010'`); the format specs `{n:b}`, `{n:o}`, `{n:x}`, `{n:08b}` print without. Digit counting is `len(str(n))` for non-negative `n`; `bin(n).count("1")` or `n.bit_count()` (3.10) counts set bits; `n.bit_length()` is the number of bits needed.

Pitfalls

`len(xs) / 2` as an index. It is a float; `xs[len(xs) / 2]` is a `TypeError`. Use `//`.

Expecting `-7 // 2` to be `-3` and `-7 % 2` to be `-1`. Python floors.

`-2 ** 2` meaning 4. It is −4.

`round(2.5)` being 3. Half-to-even gives 2.

`int(math.sqrt(n))` for a perfect-square test on large numbers. Use `math.isqrt`.

Building `a ** b` and then `% m` for large exponents. `pow(a, b, m)`.

Key takeaways

`int` is arbitrary-precision; overflow does not exist, only slowness for huge values.

`/` is always a float; `//` floors toward negative infinity; `%` takes the divisor's sign; `divmod` gives both.

`**` binds tighter than unary minus; `pow(a, b, m)` is modular exponentiation.

`round` is half-to-even; `int()` truncates; `math.floor` / `ceil` round down/up to ints.

`math.gcd`, `lcm`, `isqrt`, `comb`, `factorial` are the integer tools; `bin` / `hex` and the `b` / `x` format specs print other bases.

EXERCISE 1

Floor versus truncation

Show the difference between Python's floor division and the truncating division of C and Java. For each pair `a b` (with `b` never zero) print Python's `a // b` and `a % b`, then the truncating quotient (rounded toward zero) and the remainder that goes with it, `a - b * tq`. Compute the truncating quotient with integer arithmetic on absolute values — no floats.

Input: `n`, then `n` lines `a b`. **Output:** one line per pair: `<a> <b>: floor <q> <r> | trunc <tq> <tr>`.

```
2
7 2
-7 2
```

prints

```
7 2: floor 3 1 | trunc 3 1
-7 2: floor -4 1 | trunc -3 -1
```

INPUT 1

```
2
7 2
-7 2
```

OUTPUT 1

```
7 2: floor 3 1 | trunc 3 1
-7 2: floor -4 1 | trunc -3 -1
```

INPUT 2

```
2
7 -2
-7 -2
```

OUTPUT 2

```
7 -2: floor -4 -1 | trunc -3 1
-7 -2: floor 3 -1 | trunc 3 -1
```

REFERENCE SOLUTION

```
n = int(input())
for _ in range(n):
    a, b = map(int, input().split())
    q, r = divmod(a, b)
    tq = abs(a) // abs(b)
    if (a < 0) != (b < 0):
        tq = -tq
    tr = a - b * tq
    print(f"{a} {b}: floor {q} {r} | trunc {tq} {tr}")
```

EXERCISE 2

Big powers

Integers do not overflow. Read `n` and `k` and compute `n ** k` exactly, then report the number of decimal digits, the last four digits (zero-padded), the number of bits needed (`int.bit_length`) and the number of set bits (`int.bit_count`).

Input: one line `n k` with `n ⩾ 1` , `k ⩾ 0` . **Output:** four lines: `digits <d>` , `last4 <dddd>` , `bits <b>` , `ones <c>` .

```
2 100
```

prints

```
digits 31
last4 5376
bits 101
ones 1
```

INPUT 1

```
2 100
```

OUTPUT 1

```
digits 31
last4 5376
bits 101
ones 1
```

INPUT 2

```
7 1
```

OUTPUT 2

```
digits 1
last4 0007
bits 3
ones 3
```

REFERENCE SOLUTION

```python
n, k = map(int, input().split())
value = n ** k
print(f"digits {len(str(value))}")
print(f"last4 {value % 10_000:04d}")
print(f"bits {value.bit_length()}")
print(f"ones {value.bit_count()}")
```

QUIZ

1. What does `print(-7 // 2, -7 % 2)` print?

A. `-3 -1`

B. `-4 1`

C. `-3 1`

D. `-4 -1`

2. What is `type(6 / 3)`?

A. `int`

B. `float`

C. It depends on whether the division is exact

D. `Fraction`

3. What does `print(round(2.5), round(3.5))` print?

A. `3 4`

B. `2 4`

C. `2 3`

D. `3 3`

4. Which expression computes `3 ** 100000 % 1000000007` fastest?

A. `3 ** 100000 % 1000000007`

B. `pow(3, 100000, 1000000007)`

C. `(3 % 1000000007) ** 100000`

D. `math.pow(3, 100000) % 1000000007`

5. What is `-2 ** 2`?

A. `4`

B. `-4`

C. `SyntaxError`

D. `0`

6. Which is the exact test for whether a large integer `n` is a perfect square?

A. `int(math.sqrt(n)) ** 2 == n`

B. `math.isqrt(n) ** 2 == n`

C. `math.sqrt(n) == int(math.sqrt(n))`

D. `n ** 0.5 == int(n ** 0.5)`

ANSWER KEY · NUMBERS — INT, FLOAT AND THE ARITHMETIC THAT SURPRISES

1. B Floor division rounds toward negative infinity, so −3.5 becomes −4, and the remainder takes the sign of the divisor so that `2 * -4 + 1 == -7`. `-3 -1` is what a truncating language gives.

2. B True division always returns a float in Python 3, exact or not: `6 / 3` is `2.0`. Use `//` when an integer is wanted.

3. B `round` uses round-half-to-even: 2.5 goes to the even neighbour 2, 3.5 to the even neighbour 4. Always rounding halves up would bias sums upward.

4. B Three-argument `pow` does modular exponentiation without ever building the 47 000-digit intermediate. The first option is correct but builds it; the third still builds it; `math.pow` returns a float and overflows.

5. B `**` binds tighter than unary minus, so the expression is `-(2 ** 2)`. Parenthesise `(-2) ** 2` to square a negative number.

6. B `math.isqrt` computes the integer square root exactly for any size of int. The float-based tests lose precision above 2^{53} and can report a near-square as a square.

LESSON 2 · 14 MIN

Floating point — why 0.1 + 0.2 is not 0.3, and what to do about it

A Python `float` is an IEEE 754 double: 64 bits, of which 53 are the significand, giving about 15–17 significant decimal digits. Every language with doubles has the same behaviour, but Python makes it more visible than most because `print` shows the *shortest decimal that round-trips to the same double* rather than rounding to six places the way C's `printf("%f")` does. `0.1 + 0.2` therefore prints `0.30000000000000004` and a beginner concludes Python cannot add. This lesson explains the representation once, gives the three rules for comparing and printing floats, and names the two exact alternatives for the cases — money, mostly — where approximation is not acceptable.

Representation error

A double stores a value as a sign, a 53-bit binary fraction and a power of two. Numbers that are sums of powers of two — `0.5`, `0.25`, `3.75`, every integer below 2^{53} — are exact. `0.1` is not: in binary it is `0.0001100110011…` repeating, and the nearest double is `0.1000000000000000055511151231257827…`. `0.2` and `0.3` are likewise slightly off, and the sum of the two rounded inputs is not the same double as the rounded `0.3`:

```
print(0.1 + 0.2 == 0.3)          # False
print(0.1 + 0.2)                 # 0.30000000000000004
print(f"{0.1:.20f}")             # 0.10000000000000000555
print((0.1).hex())               # 0x1.999999999999ap-4
```

The error is about one part in 10^{16} per operation. It becomes visible when it is *compared* (`==`), *accumulated* (a million additions), or *magnified* (subtracting two nearly equal numbers). It never becomes visible if you format to a sensible number of decimals.

Rule 1: never compare floats with ==

Test closeness instead. `math.isclose(a, b)` uses a relative tolerance of 10^{-9} by default and accepts an absolute one for comparisons near zero:

```
import math
print(math.isclose(0.1 + 0.2, 0.3))                    # True
print(math.isclose(1e-10, 0.0, abs_tol=1e-9))          # True — relative tole
```

Or avoid floats altogether where the quantities are really integers: compare `3 * x == y` rather than `x == y / 3`, work in cents rather than in pounds.

Rule 2: format on output

`print(x)` shows the shortest round-tripping decimal, which is exact but ugly. Every judged exercise expects a fixed precision, and `f"{x:.2f}"` rounds *correctly* to the nearest representable two-decimal value:

```
x = 2 / 3
print(x)              # 0.6666666666666666
print(f"{x:.2f}")     # 0.67
print(f"{x:.0f}")     # 1
print(f"{1e21:.0f}") # 1000000000000000000000
print(f"{x:e}")       # 6.666667e-01
print(f"{x:g}")       # 0.666667
```

`round(x, 2)` returns a float that is again only *nearly* `0.67`; use it for arithmetic, use the format spec for display. One trap: `.2f` rounds the *binary* value, so `f"{2.675:.2f}"` is `2.67`, because 2.675 is stored as 2.67499999…; if a value must round half up at two decimals, it must not be a float.

Rule 3: accumulate carefully

Adding many floats accumulates error, and the order matters. `sum` adds left to right; `math.fsum` tracks the lost bits and returns the correctly rounded sum:

```
import math
xs = [0.1] * 10
print(sum(xs))          # 0.9999999999999999
print(math.fsum(xs))    # 1.0
```

For means and statistics, `statistics.fmean` and `statistics.mean` are similarly careful. When the sum is of decimal strings — prices, percentages — the better answer is not a

cleverer float sum but no floats at all.

Special values

`float("inf")`, `float("-inf")` and `float("nan")` exist (`math.inf`, `math.nan`). Infinity compares greater than everything and is a useful initial value for a running minimum. NaN compares unequal to *everything, including itself*: `nan == nan` is `False`, so `x != x` is the classic NaN test and `math.isnan` is the readable one. Division by zero on floats raises `ZeroDivisionError` in Python (it does not produce infinity as in C), and overflow in `**` raises `OverflowError`, while `1e308 * 10` gives `inf`.

When you need exactness

`decimal.Decimal` is decimal floating point with a configurable precision (28 significant digits by default) and explicit rounding modes. Construct it from a *string* — `Decimal("0.1")` is exactly a tenth, while `Decimal(0.1)` faithfully copies the float's error:

```
from decimal import Decimal, ROUND_HALF_UP
total = Decimal("0.10") + Decimal("0.20")
print(total)                                                    # 0.30
print(Decimal("2.675").quantize(Decimal("0.01"), rounding=ROUND_HALF_UP))   #
```

`quantize` rounds to a given number of places with the rounding you name — the correct tool for money. **`fractions.Fraction`** is exact rational arithmetic: `Fraction(1, 3) + Fraction(1, 6)` is `Fraction(1, 2)`, and `Fraction("0.1")` is one tenth. It is the right type for probabilities and ratios that must be exact, and slow enough that it is never the default.

The other exact tool is the plain **`int`**: represent money as integer cents, percentages as basis points, and convert only at the edges. `int(s.replace(".", ""))` for a two-decimal price string, `f"{cents // 100}.{cents % 100:02d}"` to print it back. Most "floating-point bugs" in real code are money held in floats, and this is the fix.

Pitfalls

- `x == 0.3` for a computed `x`. Use `math.isclose` or restructure.
- Printing a raw float when two decimals were asked for.

`round(2.675, 2)` expecting `2.68`. The binary value is below the half.

`Decimal(0.1)` instead of `Decimal("0.1")`.

`sum` of a long list of decimals when `math.fsum` or integer cents would be exact.

Using `float` for currency.

Key takeaways

A `float` is a binary double; most decimal fractions are approximations and `print` shows the exact shortest form.

Never `==` on computed floats; `math.isclose` with an `abs_tol` near zero.

Format for display with `:.2f`; `round` is for arithmetic and is still a float.

`math.fsum` sums accurately; NaN is unequal to itself; float division by zero raises.

`Decimal` from strings with `quantize` for money, `Fraction` for exact ratios, integer cents when possible.

EXERCISE 1

Three ways to add

Read one line of decimal numbers written as text and add them three ways: as floats with the built-in `sum`, as floats with `math.fsum`, and as `decimal.Decimal` values constructed **from the strings**. Print each result exactly as Python prints it (`print(x)` — no format spec), so that the representation error is visible where it exists.

Input: one line of decimal numbers. **Output:** three lines: `float <sum>`, `fsum <fsum>`, `decimal <sum>`.

```
0.1 0.2 0.3
```

prints

```
float 0.6000000000000001
fsum 0.6
decimal 0.6
```

INPUT 1

```
0.1 0.2 0.3
```

OUTPUT 1

```
float 0.6000000000000001
fsum 0.6
decimal 0.6
```

INPUT 2

```
0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
```

OUTPUT 2

```
float 0.9999999999999999
fsum 1.0
decimal 1.0
```

REFERENCE SOLUTION

```python
import math
from decimal import Decimal

tokens = input().split()
floats = [float(t) for t in tokens]
print(f"float {sum(floats)}")
print(f"fsum {math.fsum(floats)}")
print(f"decimal {sum(Decimal(t) for t in tokens)}")
```

EXERCISE 2

Money in cents

Prices are text with exactly two decimals. Add up an order **without ever creating a float**: convert each price to integer cents by splitting on the dot, multiply by the quantity, sum, and print the total formatted back as pounds and pence.

Input: `n`, then `n` lines `price qty`. **Output:** `total <pounds>.<pence>` with two digits of pence.

```
3
0.10 3
0.20 3
19.99 1
```

prints

```
total 20.89
```

INPUT 1

```
3
0.10 3
0.20 3
19.99 1
```

OUTPUT 1

```
total 20.89
```

REFERENCE SOLUTION

```python
n = int(input())
cents = 0
for _ in range(n):
    price, qty = input().split()
    pounds, pence = price.split(".")
    cents += (int(pounds) * 100 + int(pence)) * int(qty)
print(f"total {cents // 100}.{cents % 100:02d}")
```

QUIZ

1. Why is `0.1 + 0.2 == 0.3` false?

A. Python's addition is imprecise for small numbers

B. None of 0.1, 0.2 and 0.3 is exactly representable in binary, and the rounded sum of the first two is a different double from the rounded 0.3

C. The `==` operator compares floats by identity

D. Because `0.3` is parsed as a `Decimal`

2. What does `f"{2.675:.2f}"` evaluate to?

A. `'2.68'`

B. `'2.67'`

C. `'2.675'`

D. `'2.7'`

3. Which expression is exactly one tenth?

A. `Decimal(0.1)`

B. `Decimal("0.1")`

C. `float("0.1")`

D. `round(0.1, 1)`

4. What does `print(sum([0.1] * 10), math.fsum([0.1] * 10))` print?

A. `1.0 1.0`

B. `0.9999999999999999 1.0`

C. `1.0 0.9999999999999999`

D. `0.9999999999999999 0.9999999999999999`

5. What is `float("nan") == float("nan")`?

A. `True`

B. `False`

C. It raises `ValueError`

D. `None`

6. What happens on `1.0 / 0`?

A. `inf`

B. `nan`

C. `ZeroDivisionError`

D. `OverflowError`

ANSWER KEY · FLOATING POINT — WHY 0.1 + 0.2 IS NOT 0.3, AND WHAT TO DO ABOUT IT

1. B Every language with IEEE doubles has the same result; it is representation error in the inputs, not a flaw in addition. `math.isclose` is the comparison to use.

2. B The double nearest 2.675 is slightly below it (2.67499999…), so correct rounding of the stored value gives 2.67. When half-up rounding of a decimal is required, use `Decimal("2.675").quantize(...)`.

3. B Constructing a `Decimal` from the string parses the decimal digits exactly. `Decimal(0.1)` faithfully copies the float's binary approximation, and the two float options are the approximation itself.

4. B Left-to-right float addition accumulates rounding error; `fsum` tracks the lost low-order bits and returns the correctly rounded result, which is exactly 1.0.

5. B NaN compares unequal to everything, including itself, by the IEEE rules. `math.isnan(x)` is the readable test; `x ≠ x` is the classic one.

6. C Unlike C, Python raises on float division by zero rather than producing infinity. Overflowing multiplication (`1e308 * 10`) does give `inf`, and `**` overflow raises `OverflowError`.

LESSON 3 · 13 MIN

Booleans, truthiness, None, and is versus ==

Every Python object is either *truthy* or *falsy*, and the language's conditionals, `and`, `or`, `not`, `any` and `all` are defined over that property rather than over `bool` alone. That one rule makes `if xs:` the idiom for "if the list is non-empty" and `x or default` the idiom for a fallback — and makes `if count:` a bug when zero is a legitimate count. This lesson fixes the truthiness table, explains what `and` and `or` actually return, settles `None` and the difference between identity (`is`) and equality (`==`), and shows how comparison chaining reads.

---

bool is an int

`True` and `False` are the two instances of `bool`, and `bool` is a subclass of `int` with values 1 and 0. Consequences that are used on purpose: `True + True == 2`, `sum(x > 0 for x in xs)` counts the positives, `xs[flag]` picks the second element when `flag` is true. Consequences to avoid: `True == 1` is `True`, `{1: "a", True: "b"}` has one key, and `isinstance(True, int)` is `True` — code that checks `type(x) is int` treats booleans as integers.

Comparison operators return `bool`: `3 < 5` is `True`, `"a" == "a"` is `True`. `bool(x)` converts anything using the truthiness rules below.

The truthiness table

Falsy	Everything else is truthy
`False`, `None`	`True`
`0`, `0.0`, `0j`, `Decimal("0")`, `Fraction(0)`	any non-zero number, including `-1` and `0.001`
`""` (empty string)	any non-empty string — including `"False"`, `"0"` and `" "`
`[]`, `()`, `{}`, `set()`, `range(0)`	any non-empty container — including `[0]`, `[None]` and `[[]]`
objects whose `__bool__` returns `False` or whose `__len__` returns 0	objects with neither method

"Empty and zero are false; everything else is true" covers it. A user-defined class is truthy unless it defines `__bool__` or `__len__` (Module 8). The classic bug is the string `"False"`, which is true because it is non-empty; the second classic is `if n:` where `n` may legitimately be `0`.

```
def describe(xs):
    if xs:                      # non-empty
        return f"{len(xs)} items"
    return "empty"

def find(xs, target):
    idx = xs.index(target) if target in xs else None
    if idx is not None:         # NOT `if idx:` — index 0 is a real answer
        return idx
```

---

and, or, not — what they return

`not x` returns a `bool`. `and` and `or` do **not**: they return one of their operands, and they short-circuit.

- `x and y` evaluates `x`; if it is falsy the result is `x` (and `y` is never evaluated); otherwise the result is `y`.
- `x or y` evaluates `x`; if it is truthy the result is `x`; otherwise the result is `y`.

```
print(0 or "default")      # 'default'
print("name" or "default") # 'name'
print("" and 5)            # ''   — the falsy left operand
print(3 and 0)             # 0    — the right operand, whatever it is
print(3 and 7)             # 7
```

The short-circuit is the guard idiom: `xs and xs[0]` never indexes an empty list; `d.get("k") or fallback` supplies a default when the key is missing *or its value is falsy*. That last clause is the trap: `count or 10` replaces a genuine `0` with 10; when zero is meaningful, write `count if count is not None else 10`. In a condition the returned operand is truth-tested anyway, so the distinction only shows when the result is stored or printed.

`any(iterable)` is `True` if any element is truthy; `all(iterable)` if every one is (and `all([])` is `True` — vacuous truth). Both short-circuit and take a generator, so `any(x < 0 for x in xs)` stops at the first negative.

None

`None` is the single instance of `NoneType`: the value of a function that does not `return`, the default for an optional argument, the "no result" of `dict.get` and `re.match`. It is falsy, but the test for it is identity, `x is None` / `x is not None`, never `x == None` — a class could define `__eq__` to say it equals `None`, and `is` asks the only question you mean: "is this *the* None object?".

---

is versus ==

`==` asks whether two objects have equal *values* (each class decides, through `__eq__`). `is` asks whether two names refer to the *same object* — the same identity, the same `id()`. Two equal lists are not the same list:

```
a = [1, 2]
b = [1, 2]
c = a
print(a == b, a is b, a is c)    # True False True
```

Use `is` only for singletons: `None`, `True`, `False`, `NotImplemented`, `Ellipsis`, and sentinel objects you created yourself. Never for numbers or strings: CPython caches small integers (−5

to 256) and interns some strings, so `x is 5` may be `True` today and `False` for `x = 1000` — the interpreter warns `SyntaxWarning: "is" with a literal` for this exact mistake. `x == 5` is always the right question.

Comparison chaining

`a < b < c` means `a < b and b < c`, with `b` evaluated once. Any comparison operators chain: `0 <= i < len(xs)` is the bounds check, `a == b == c` tests three-way equality, `x < y > z` is legal but unreadable. This reads as mathematics, unlike C, where `a < b < c` compares a boolean with `c`. `in` and `is` are comparison operators too: `x in xs`, `x not in xs`, `a is not None` all chain by the same rule.

Conditional expressions

`value_if_true if condition else value_if_false` is Python's ternary; the condition sits in the middle, which reads naturally once seen a few times: `label = "even" if n % 2 == 0 else "odd"`. It is an expression, so it goes inside f-strings and arguments. Nesting more than one level is legal and discouraged.

Pitfalls

- `if count:` when `0` is a valid count; `if s:` when `"0"` should be false.
- `x or default` replacing a legitimate falsy value — `0`, `""`, `[]`.
- `x == None`. Use `is None`.
- `x is 5`, `s is "abc"`. Identity on values is an accident of caching.
- Believing `and` / `or` return booleans; `print(a and b)` prints an operand.
- `all([])` is `True` — check emptiness separately when it matters.

Key takeaways

- `bool` is an `int` subclass; comparisons return it; `sum` of booleans counts.
- Empty and zero are falsy, everything else truthy — including `"False"`, `[0]` and `-1`.
- `and` / `or` return an operand and short-circuit; `not`, `any`, `all` return booleans.

`is` is identity, for `None` and sentinels only; `==` is value equality.

Comparisons chain: `0 ⩽ i < n`; the conditional expression is `a if c else b`.

EXERCISE 1

Truthy or falsy

Read `n` Python literals, one per line — numbers, strings in quotes, lists, tuples, dicts, `None`, `True` / `False` — turn each into a value with `ast.literal_eval` (which parses literals without running code) and report its truth value.

Input: `n`, then `n` literals. **Output:** `<literal as written> is truthy` or `... is falsy`.

```
4
0
""
[0]
"False"
```

prints

```
0 is falsy
"" is falsy
[0] is truthy
"False" is truthy
```

INPUT 1

```
5
0
""
[]
[0]
None
```

OUTPUT 1

```
0 is falsy
"" is falsy
[] is falsy
[0] is truthy
None is falsy
```

INPUT 2

```
4
"False"
0.0
{}
-1
```

OUTPUT 2

```
"False" is truthy
0.0 is falsy
{} is falsy
-1 is truthy
```

REFERENCE SOLUTION

```python
import ast

n = int(input())
for _ in range(n):
    text = input()
    value = ast.literal_eval(text)
    word = "truthy" if value else "falsy"
    print(f"{text} is {word}")
```

EXERCISE 2

Chained comparison with missing values

Each line holds three values `a b c`, each an integer or `-` meaning missing. Decide whether `a < b < c` holds using a **chained comparison**; when any value is missing the answer is `unknown`. Represent a missing value as `None` and test for it with `is None`.

Input: `n`, then `n` lines. **Output:** `True`, `False` or `unknown` per line.

```
4
1 2 3
3 2 1
1 - 3
2 2 3
```

prints

```
True
False
unknown
False
```

INPUT 1

```
4
1 2 3
3 2 1
1 - 3
2 2 3
```

OUTPUT 1

```
True
False
unknown
False
```

REFERENCE SOLUTION

```
def parse(tok):
    return None if tok == "-" else int(tok)


n = int(input())
for _ in range(n):
    a, b, c = (parse(tok) for tok in input().split())
    if a is None or b is None or c is None:
        print("unknown")
    else:
        print(a < b < c)
```

QUIZ

1. Which of these values is truthy?

A. `""`

B. `[0]`

C. `0.0`

D. `None`

2. What does `print(0 or "x", 3 and 0, "" and 5)` print?

A. `True False False`

B. `x 0`

C. `x 0 5`

D. `0 3 5`

3. Why should `x is None` be used rather than `x == None`?

A. `==` cannot compare with `None`

B. `is` is faster and a class could define `__eq__` to say it equals `None`; identity asks the exact question

C. `x == None` is a syntax error

D. There is no difference

4. What does `0 ⩽ i < len(xs)` mean?

A. `(0 ⩽ i) < len(xs)`

B. `0 ⩽ i and i < len(xs)`, with `i` evaluated once

C. `0 ⩽ (i < len(xs))`

D. A syntax error

5. What does `print(True + True, sum([True, False, True]))` print?

A. `TypeError`

B. `2 2`

C. `True 2`

D. `2 True`

6. `count = 0` is a genuine result. What is wrong with `value = count or 10`?

A. Nothing — it gives 0

B. It gives 10, because 0 is falsy and `or` moves on to the fallback

C. It raises `TypeError`

D. It gives `True`

ANSWER KEY · BOOLEANS, TRUTHINESS, NONE, AND IS VERSUS ==

1. B A non-empty list is truthy regardless of what it contains — `[0]`, `[None]`, `[[]]` are all true. The empty string, zero of any numeric type and `None` are falsy.

2. B `or` returns the first truthy operand or the last one (`"x"`); `and` returns the first falsy operand or the last one — `0` for `3 and 0`, and the empty string for `"" and 5`. They return operands, not booleans.

3. B `None` is a singleton, so identity is exact and cannot be overridden by a custom `__eq__`. Equality works most of the time, which is what makes the rare surprise hard to find.

4. B Comparison operators chain: each adjacent pair is compared with `and` between them. This is the idiomatic bounds check; in C the same text would compare a boolean with a length.

5. B `bool` is a subclass of `int` with values 1 and 0, so booleans add and sum like integers. Counting matches with `sum(cond for ...)` relies on exactly this.

6. B `or` cannot tell a missing value from a falsy one. When zero, an empty string or an empty list is meaningful, write `count if count is not None else 10`.

LESSON 4 · 14 MIN

Names, binding and mutability — there are no boxes

The mental model that carries a Python programmer through every "why did that change too?" moment is this: a variable is a **name bound to an object**, not a box holding a value. Assignment binds; it never copies. Whether a later change through one name is visible through another depends on one question — is the object *mutable*? — and not on how the names were introduced. This lesson builds that model with the objects you already know, adds the assignment forms (multiple, augmented, unpacking, the walrus), and sets up the copying rules that Module 6 needs for lists.

Assignment binds a name

```
x = [1, 2, 3]
y = x
```

After these two lines there is *one* list object and two names bound to it. `id(x) == id(y)` and `x is y`. Nothing was copied, because assignment never copies: it evaluates the right side to an object and makes the left name refer to that object. Rebinding one name does not affect the other:

```
y = [9]        # y now labels a new list; x still labels the first
print(x)       # [1, 2, 3]
```

But *mutating* the shared object through either name is visible through both:

```
y = x
y.append(4)
print(x)       # [1, 2, 3, 4] — same object
```

The distinction is between **rebinding** (`y = …`, changes which object `y` labels) and **mutation** (`y.append(…)`, `y[0] = …`, changes the object itself). Only mutation is shared.

Mutable and immutable

Immutable — cannot be changed in place	Mutable — can be changed in place
`int`, `float`, `bool`, `complex`	`list`, `dict`, `set`
`str`, `bytes`, `tuple`, `frozenset`, `range`	`bytearray`, user-defined class instances (by default)
`None`	

Every operation on an immutable object that looks like a change produces a *new* object: `s = s + "!"`, `s = s.upper()`, `n += 1`. Two names bound to the same string can never observe each other changing, because a string never changes. This is why passing an `int` to a function "by value" and a `list` "by reference" is the wrong description — both are passed the same way (the object is bound to the parameter name); the difference is what the function can *do* to what it received.

```
def bump(n):
    n += 1            # rebinds the local name to a new int; the caller's obje

def extend(xs):
    xs.append(0)      # mutates the shared list; the caller sees it

def replace(xs):
    xs = [0]          # rebinds the local name; the caller's list is untouched
```

Augmented assignment

`x += y` calls `x.__iadd__(y)` if the type defines it, and falls back to `x = x + y`. Lists define `__iadd__` to extend in place, so `xs += [4]` mutates the shared list, while `xs = xs + [4]` creates a new one. Integers and strings have no in-place form, so `n += 1` always rebinds. The consequence for a shared list:

```
a = [1]
b = a
b += [2]          # in place: a is [1, 2]
b = b + [3]       # new list: a is still [1, 2], b is [1, 2, 3]
```

Multiple names, one line

Python binds several names at once from an iterable on the right — *unpacking*:

```
a, b = 1, 2                  # a tuple on the right, two targets on the left
a, b = b, a                  # swap: the right side is built before any name i
first, *rest = [1, 2, 3, 4]  # first = 1, rest = [2, 3, 4]
x = y = 0                    # both names bound to the same int object
```

The chained form `x = y = []` binds *both* names to the *same* list — the aliasing trap again. `x = y = 0` is fine because ints are immutable.

The walrus operator

`:=` assigns as an expression (3.8), for the case where a value must be both tested and used:

```
import re
if (m := re.match(r"(\d+)", line)):
    print(m.group(1))

while (chunk := stream.read(1024)):
    process(chunk)

if (n := len(xs)) > 10:
    print(f"too many: {n}")
```

It binds in the enclosing function scope like ordinary assignment. It is not a general replacement for `=` — the parentheses are needed in most positions, and a reader expects it only where the value is reused immediately.

Names have no type; objects do

`x = 5` then `x = "five"` is legal. The name `x` has no type; `type(x)` reports the type of the object it currently labels. Type hints (`x: int = 5`) record intent for readers and tools (Module 15) and change nothing at run time. `del x` removes the binding — the object is freed if no other name refers to it — and a later `x` is a `NameError`.

Constants are a convention: `MAX_RETRIES = 3` is an ordinary name that PEP 8 asks you not to rebind. `Final` from `typing` lets a checker enforce it; the interpreter does not.

Copying, briefly

When you need an independent object, ask for one: `list(xs)`, `xs[:]`, `xs.copy()`, `dict(d)`, `set(s)` make a *shallow* copy — a new container whose elements are the same objects. For nested structures that is not enough (the inner lists are still shared); `copy.deepcopy` copies recursively. Module 6 returns to this with the `[[0] * 3] * 3` trap, which is the same aliasing written as a multiplication.

Pitfalls

- Expecting `b = a` to copy a list, dict or set.
- `x = y = []` sharing one list between two names.
- A function that mutates its argument when the caller expected a new value, or the reverse.
- `xs += [4]` on a list you meant to leave alone.
- `def f(xs=[])`: the default list is created once and shared by every call (Module 4).
- `is` for value comparison; two equal ints are not guaranteed to be the same object.

Key takeaways

- A name is a label bound to an object; assignment binds and never copies.
- Rebinding changes the label; mutation changes the object; only mutation is seen through other labels.
- Numbers, strings and tuples are immutable — every "change" is a new object; lists, dicts and sets are mutable.
- `+=` mutates a list in place and rebinds an int or string; `a, b = b, a` swaps; `*rest` gathers.
- `:=` assigns inside an expression; shallow copies share elements, `deepcopy` does not.

EXERCISE 1

A tiny binding machine

Simulate names bound to list objects. Read commands until the end of input: `new X` binds `X` to a fresh empty list; `alias Y X` binds `Y` to **the same object** as `X`; `copy Y X` binds `Y` to a **shallow**

copy of `X`'s list; `push X v` appends the integer `v` to `X`'s list; `show X` prints the list. The point is that a push through an alias is visible through the original, and a push through a copy is not.

Input: commands, one per line. **Output:** for every `show`: `<name>: <elements separated by spaces>` or `<name>: (empty)`.

```
new a
alias b a
push b 1
copy c a
push a 2
show a
show b
show c
```

prints

```
a: 1 2
b: 1 2
c: 1
```

INPUT 1

```
new a
alias b a
push b 1
copy c a
push a 2
show a
show b
show c
```

OUTPUT 1

```
a: 1 2
b: 1 2
c: 1
```

REFERENCE SOLUTION

```python
import sys

names = {}
for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    cmd = parts[0]
    if cmd == "new":
        names[parts[1]] = []
    elif cmd == "alias":
        names[parts[1]] = names[parts[2]]
    elif cmd == "copy":
        names[parts[1]] = list(names[parts[2]])
    elif cmd == "push":
        names[parts[1]].append(int(parts[2]))
    elif cmd == "show":
        items = names[parts[1]]
        body = " ".join(map(str, items)) if items else "(empty)"
        print(f"{parts[1]}: {body}")
```

EXERCISE 2

Walrus reads

Read lines until a line that is exactly `END` using a single `while` whose condition assigns and tests with the walrus operator: `while (line := input()) ≠ "END":`. Print the length of every line before `END`, then the number of lines read.

Input: lines, the last of which is `END` (anything after it is ignored). **Output:** one length per line, then `count: <n>`.

```
hello
hi
END
```

prints

```
5
2
count: 2
```

INPUT 1

```
hello
hi
END
```

OUTPUT 1

```
5
2
count: 2
```

REFERENCE SOLUTION

```python
count = 0
while (line := input()) != "END":
    print(len(line))
    count += 1
print(f"count: {count}")
```

QUIZ

1. What does this print?

```python
a = [1, 2]
b = a
b = b + [3]
print(a)
```

A. `[1, 2, 3]`

B. `[1, 2]`

C. `[3]`

D. `[1, 2, [3]]`

2. What does this print?

```
def bump(n):
    n += 1

x = 5
bump(x)
print(x)
```

A. `6`

B. `5`

C. `None`

D. `TypeError`

3. After `x = y = []` and `x.append(1)`, what is `y`?

A. `[]`

B. `[1]`

C. `None`

D. `NameError`

4. Which statement about mutability is correct?

A. Strings are mutable; `s.upper()` changes `s` in place

B. Tuples are immutable, but a list inside a tuple can still be mutated

C. Lists are immutable once passed to a function

D. Integers are mutable; `n += 1` changes the object

5. What does `if (n := len(xs)) > 3: print(n)` do?

A. Syntax error — assignment is not allowed in a condition

B. Binds `n` to the length and prints it when it exceeds 3

C. Compares `len(xs)` with 3 and prints `True`

D. Binds `n` to `True` or `False`

ANSWER KEY · NAMES, BINDING AND MUTABILITY — THERE ARE NO BOXES

1. B `b + [3]` builds a new list and the assignment rebinds `b` to it; `a` still labels the original. Had the line been `b += [3]`, the list would have been extended in place and `a` would show `[1, 2, 3]`.

2. B Integers are immutable; `n += 1` rebinds the local name `n` to a new int and the caller's binding is untouched. The function would need to `return n` and the caller to assign it.

3. B Chained assignment binds both names to the one list object created on the right. Mutating it through `x` is visible through `y`. For two independent lists write two assignments.

4. B A tuple cannot be rebound element-wise, but it holds references, and a mutable element such as a list can be changed through it: `t = ([],); t[0].append(1)` works. Strings and ints never change in place; every apparent change is a new object.

5. B The walrus operator is an assignment *expression*: it binds `n` and evaluates to the bound value, so the comparison uses the length and the body can reuse `n`. Ordinary `=` in that position would be the syntax error.

LESSON 5 · 12 MIN

Conversions — between text, numbers and containers

Python converts between types only when asked, and the asking is done by calling the target type: `int("42")`, `str(3.5)`, `list("abc")`, `float(7)`. Each conversion has precise rules about what it accepts and what it raises, and half the `ValueError`s in a beginner's program come from one of them being fed something it does not take. This lesson lists the conversions that matter, the failures each produces, the two implicit conversions that *do* exist (numeric widening and truth testing), and the cases — `"1" + 1`, `int("3.0")`, `float` to `int` — where the interpreter will not guess.

Strings to numbers

`int(s)` parses an optional sign, decimal digits and underscores, ignoring surrounding whitespace. Everything else is a `ValueError` with the offending text in the message:

```
int("42")        # 42
int("  -7\n")    # -7
int("1_000")     # 1000
int("3.0")       # ValueError: invalid literal for int() with base 10: '3.0'
int("")          # ValueError
int("0x1f")      # ValueError — base 10 does not know prefixes
int("0x1f", 16)  # 31 — with the base, a matching prefix is allowed
int("1f", 16)    # 31
int("101", 2)    # 5
int("0o17", 0)   # 15 — base 0 reads the prefix
```

`float(s)` parses decimal notation, exponents, `inf`, `nan`, with sign and whitespace: `float("2.5")`, `float("1e3")` → `1000.0`, `float("-inf")`. It also accepts what `int` accepts, so `float("7")` is `7.0`. A string that is a valid float but not a valid int is common input (`"3.0"`): read it with `float` and truncate with `int(float(s))` if an integer is wanted, or reject it, but do not pass it to `int`.

The recipe for classifying a token is to try the strict parse first:

```python
def classify(tok):
    try:
        return "int", int(tok)
    except ValueError:
        pass
    try:
        return "float", float(tok)
    except ValueError:
        return "text", tok
```

`str.isdigit()` is *not* a substitute — it is false for `"-7"` and true for `"²"` — and Module 5 lesson 4 explains the three `is*` methods it is often confused with.

Numbers to strings

`str(x)` and `f"{x}"` produce the same text as `print` would: `str(42)` is `"42"`, `str(2.5)` is `"2.5"`, `str(1e21)` is `"1e+21"`, `str(0.1 + 0.2)` is `"0.30000000000000004"`. `repr(x)` is the same for numbers and differs for strings (`repr("a")` is `"'a'"`, with quotes). Format specs control the shape: `f"{x:.2f}"`, `f"{n:05d}"`, `f"{n:x}"`, `f"{n:,}"`. `bin(n)`, `oct(n)`, `hex(n)` give prefixed strings; `format(n, "b")` gives digits only; `int(bin(n), 2)` round-trips.

Between numbers

`float(n)` widens an int exactly up to 2^{53} and rounds above it. `int(x)` truncates a float toward zero — `int(3.9)` is 3, `int(-3.9)` is −3 — and raises `ValueError` on `nan` and `OverflowError` on `inf`. `round(x)` gives the nearest int (half to even); `math.floor` and `math.ceil` round down and up. `bool(x)` applies the truthiness table; `int(True)` is 1. `complex(2, 3)` is `(2+3j)`, and no conversion goes from complex back to real without `.real` / `.imag`.

Two implicit conversions exist. Arithmetic widens `int → float → complex` as needed. And any object is converted to a truth value when tested in `if`, `while`, `and`, `or`, `not`. There is no implicit `int ⇔ str`: `"1" + 1` is a `TypeError`, `"1" * 3` is `"111"` (repetition, not multiplication), and concatenating a number into text needs `str(n)` or an f-string.

Characters and code points

`ord("A")` is `65`; `chr(65)` is `"A"`. Arithmetic on code points is how you shift letters: `chr(ord("a") + 3)` is `"d"`, and `ord(c) - ord("0")` turns a digit character into its value (though `int(c)` is clearer). Module 5 lesson 4 covers the encoding side, `str.encode` and `bytes.decode`.

Containers

The container constructors take any iterable:

```
list("abc")            # ['a', 'b', 'c']
tuple([1, 2])          # (1, 2)
set([1, 1, 2])         # {1, 2}
list(range(3))         # [0, 1, 2]
dict([("a", 1)])       # {'a': 1}
dict(zip(keys, vals))  # pairs into a mapping
"".join(["a", "b"])    # 'ab' — the string constructor is join, not str()
str([1, 2])            # '[1, 2]' — the repr, not the elements
```

`str(container)` is almost never what you want in output; `" ".join(map(str, xs))` is. `list(x)` on a list makes a shallow copy; on a dict it gives the keys; on a string the characters; on a file the lines.

Conversions your own types can define

The built-in constructors ask the object: `int(x)` calls `x.__int__()` (or `__index__`), `float(x)` calls `__float__`, `str(x)` calls `__str__`, `bool(x)` calls `__bool__` then `__len__`, `list(x)` iterates `__iter__`. A class that implements these participates in the same conversions (Module 8 and Module 16). `Decimal("0.1")` and `Fraction("1/3")` are ordinary classes whose constructors parse strings — the pattern is not special to the built-ins.

Pitfalls

- `int("3.0")`. Parse with `float` first.
- `int(x)` truncating toward zero when you wanted the floor of a negative number.
- `"1" + 1` and `"total: " + n`. Use an f-string.
- `str(xs)` to print a list's elements.

`int(s)` on input with a stray non-digit — surrounding whitespace is fine, an inner space is not.

Checking `s.isdigit()` before `int(s)` and rejecting negative numbers.

Key takeaways

Conversion is explicit: call the target type; `int`, `float` and friends raise `ValueError` on text they cannot parse and `TypeError` on the wrong kind of object.

`int(s, base)` parses other bases; `bin` / `oct` / `hex` and format specs print them.

`int(x)` truncates a float; `round`, `floor`, `ceil` are the other roundings; numeric arithmetic widens implicitly, nothing else does.

`ord` / `chr` map characters to code points; container constructors take any iterable; `join` builds a string from parts.

Try the strict conversion first and catch `ValueError` to classify a token.

EXERCISE 1

Parse or reject

Classify every token on a line by trying the strict conversion first: if `int(tok)` succeeds it is an `int`; otherwise if `float(tok)` succeeds it is a `float`; otherwise it is `text`. Print the parsed value for numbers (a float prints as Python prints it) and the token itself for text.

Input: one line of whitespace-separated tokens. **Output:** one line per token: `int <v>`, `float <v>` or `text <tok>`.

```
42 3.5 1e3 abc 0x1f -7 inf
```

prints

```
int 42
float 3.5
float 1000.0
text abc
text 0x1f
int -7
float inf
```

INPUT 1

```
42 3.5 1e3 abc 0x1f -7 inf
```

OUTPUT 1

```
int 42
float 3.5
float 1000.0
text abc
text 0x1f
int -7
float inf
```

REFERENCE SOLUTION

```
def classify(tok):
    try:
        return "int", int(tok)
    except ValueError:
        pass
    try:
        return "float", float(tok)
    except ValueError:
        return "text", tok


for tok in input().split():
    kind, value = classify(tok)
    print(kind, value)
```

EXERCISE 2

Bases

Each line holds a number written in some base and the base (2–36). Parse it with `int(text, base)` and print its value in decimal, then with `bin`, `oct` and `hex` (which keep their prefixes and the sign).

Input: `n`, then `n` lines `text base`. **Output:** `<decimal> <bin> <oct> <hex>` per line.

```
2
ff 16
101 2
```

prints

```
255 0b11111111 0o377 0xff
5 0b101 0o5 0x5
```

INPUT 1

```
2
ff 16
101 2
```

OUTPUT 1

```
255 0b11111111 0o377 0xff
5 0b101 0o5 0x5
```

REFERENCE SOLUTION

```python
n = int(input())
for _ in range(n):
    text, base = input().split()
    value = int(text, int(base))
    print(value, bin(value), oct(value), hex(value))
```

QUIZ

1. What does `int("3.0")` do?

A. Returns `3`

B. Returns `3.0`

C. Raises `ValueError`

D. Raises `TypeError`

2. What does `"3" * 3` evaluate to?

A. `9`

B. `"9"`

C. `"333"`

D. `TypeError`

3. What does `int(-3.9)` return?

A. `-4`

B. `-3`

C. `-3.0`

D. `ValueError`

4. Which expression turns the text `"1f"` into the integer 31?

A. `int("1f")`

B. `int("1f", 16)`

C. `hex("1f")`

D. `int("0x1f")`

5. What does `str([1, 2, 3])` produce?

A. `'1 2 3'`

B. `'1, 2, 3'`

C. `'[1, 2, 3]'`

D. `'123'`

ANSWER KEY · CONVERSIONS — BETWEEN TEXT, NUMBERS AND CONTAINERS

1. C `int` parses integer literals only; a decimal point is not accepted. Parse with `float` first and truncate with `int(float(s))` if an integer is wanted. `TypeError` would need a non-string, non-number argument such as a list.

2. C A string times an integer is repetition. No implicit conversion to a number happens; `int("3") * 3` gives 9.

3. B `int()` on a float truncates toward zero. `math.floor(-3.9)` gives −4 and `round(-3.9)` gives −4; choose the rounding you mean.

4. B With an explicit base, `int` parses digits in that base. Base 10 rejects both `1f` and `0x1f`; `hex` goes the other way, from an int to a string.

5. C `str` of a list is its `repr`, brackets and commas included. To print elements space-separated use `" ".join(map(str, xs))` or `print(*xs)`.

LESSON 6 · 13 MIN

Operators and precedence

Python's operators are the usual set plus a few of its own — `//`, `**`, `in`, `is`, `@`, the walrus — and the precedence table is short enough to learn. What is worth learning precisely is the handful of places where the table produces something a reader does not expect: `-2 ** 2`, `not a == b`, `a & b == c`, and the bitwise operators applied to arbitrary-precision integers. This lesson gives the table, the semantics of the bitwise operators on unbounded ints, and the operator module that lets you pass an operator as a function.

The table, highest first

Precedence	Operators	Notes
1	`(…)`, `[…]`, `{…}`	grouping and literals
2	`x[i]`, `x[a:b]`, `f(…)`, `x.attr`	subscript, slice, call, attribute
3	`await x`	
4	`**`	right-associative; binds tighter than unary minus on its **left**
5	`+x`, `-x`, `~x`	unary
6	`*`, `@`, `/`, `//`, `%`	
7	`+`, `-`	binary
8	`<<`, `>>`	shifts
9	`&`	bitwise and
10	`^`	bitwise xor
11	`|`	bitwise or
12	`in`, `not in`, `is`, `is not`, `<`, `⩽`, `>`, `⩾`, `≠`, `==`	comparisons — all one level, and they chain
13	`not x`	
14	`and`	
15	`or`	
16	`x if c else y`	conditional expression
17	`lambda`	
18	`:=`	walrus

Three consequences are worth memorising. `**` is right-associative: `2 ** 3 ** 2` is `2 ** 9 = 512`, not `64`. `-2 ** 2` is `-(2 ** 2) = -4`, because the unary minus is *lower* than `**`; but `2 ** -1` is `0.5`, because the exponent position takes a unary expression. And the bitwise operators sit *above* comparisons, so `a & b == c` parses as `a & (b == c)` — parenthesise every bitwise sub-expression inside a comparison. Comparisons sit above `not`, so `not a ==`

`b` is `not (a == b)`, which is what you meant; `not a in xs` works the same way but `a not in xs` is the readable spelling.

Everything else associates left to right: `10 - 4 - 3` is `3`, `100 / 10 / 2` is `5.0`, `a % b % c` is `(a % b) % c`.

Arithmetic

`+ - * / // % **` are the previous lessons' subject; `@` is matrix multiplication, defined by NumPy arrays and by your own classes through `__matmul__`, and unused by the built-in types. `+` on sequences concatenates (`[1] + [2]`, `"a" + "b"`) and `*` with an int repeats (`[0] * 5`, `"-" * 20`). The augmented forms `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, `&=`, `|=`, `^=`, `<<=`, `>>=` exist for every binary operator; there is no `++` or `--` (`++n` parses as `+(+n)`, a no-op, silently).

Bitwise operators on unbounded integers

`&`, `|`, `^`, `~`, `<<`, `>>` treat an `int` as a two's-complement number with *infinitely many* bits. Positive numbers behave as in any language: `0b1100 & 0b1010` is `0b1000`, `1 << 40` is `1099511627776` (no overflow), `x >> 1` halves toward negative infinity. `~x` is `-x - 1`, so `~5` is `-6`, and a negative number has an infinite run of leading ones, which is why `-1 & 0xFF` is `255` — masking selects the low bits exactly as you would want.

The idioms interviews expect:

```
x & 1                # 1 if odd
x & (x - 1)          # clears the lowest set bit; zero iff x is a power of two
x & -x               # isolates the lowest set bit
x | (1 << k)         # set bit k
x & ~(1 << k)        # clear bit k
x ^ (1 << k)         # toggle bit k
(x >> k) & 1         # read bit k
x.bit_count()        # number of set bits (3.10); bin(x).count("1") before that
x.bit_length()       # bits needed: (255).bit_length() == 8
```

Flags are sets of bits: `READ, WRITE, EXEC = 1, 2, 4`; a permission set is `READ | EXEC`; testing is `perms & WRITE`, which is `0` or `WRITE` — truthy either way for `if`, but compare with

`≠ 0` or `== WRITE` when a boolean must be stored. `enum.Flag` gives these a type (Module 13).

Comparisons, membership, identity

`==`, `≠`, `<`, `⩽`, `>`, `⩾` compare values and chain (`0 ⩽ i < n`). `in` and `not in` test membership — of an element in a container, a substring in a string, a key in a dict. `is` and `is not` test identity (Module 2 lesson 3). Comparing different types with `<` raises `TypeError` (`3 < "3"`); `==` between different types is simply `False`. Sequences compare lexicographically: `[1, 2] < [1, 3]`, `"abc" < "abd"`, `(1, "b") < (1, "c")` — the rule that makes sorting tuples by several keys work (Module 6).

The operator module

Every operator has a named function in `operator`: `operator.add`, `mul`, `floordiv`, `neg`, `lt`, `eq`, `and_`, `or_`, `contains`, `getitem`. They are what you pass to `functools.reduce`, `map` or `sorted(key=…)` when a lambda would only wrap an operator: `reduce(operator.mul, xs, 1)` is the product. `operator.itemgetter(1)` and `attrgetter("name")` build key functions (Module 11).

Evaluating from text

An expression written as a string can be evaluated with `eval`, and a program that reads expressions from untrusted input must not — `eval("__import__('os').system('rm -rf /')")` is why. For arithmetic on trusted, small input it is a reasonable tool, and `ast.literal_eval` safely parses *literals only* (numbers, strings, tuples, lists, dicts) without evaluating code, which is the right way to turn `"[1, 2, 3]"` back into a list.

Pitfalls

- `-2 ** 2` is −4; `2 ** 3 ** 2` is 512.
- `a & mask == 0` compares first; write `(a & mask) == 0`.
- `++i` does nothing; `i += 1`.
- `3 < "3"` is a `TypeError`; parse first.

`x is not None and x > 0` needs no parentheses, but `not x in xs` should be `x not in xs`.

`eval` on input you did not write.

Key takeaways

`**` binds tightest among arithmetic and associates right; unary minus is below it; bitwise operators are above comparisons.

Comparisons chain and sit above `not`, `and`, `or`, which sit above the conditional expression.

Bitwise operators work on infinite two's-complement integers; `x & (x - 1)`, `1 << k`, `bit_count` are the idioms.

`==` across types is `False`, `<` across types is an error; sequences compare lexicographically.

`operator` names every operator as a function; `ast.literal_eval` parses literals safely.

EXERCISE 1

Permission flags

Model a permission set as bits: `read` is 1, `write` is 2, `exec` is 4. The first line lists the permissions initially granted (possibly none). Then `n` commands follow: `has <perm>` prints `yes` or `no` using `&`; `add <perm>` sets a bit with `|`; `remove <perm>` clears it with `& ~`; `show` prints the mask as three binary digits.

Input: a line of permissions, then `n`, then `n` commands. **Output:** one line per `has` and `show`.

```
read exec
4
show
has write
add write
show
```

prints

```
mask 101
no
mask 111
```

INPUT 1

```
read exec
4
show
has write
add write
show
```

OUTPUT 1

```
mask 101
no
mask 111
```

REFERENCE SOLUTION

```python
BITS = {"read": 1, "write": 2, "exec": 4}

mask = 0
for perm in input().split():
    mask |= BITS[perm]

n = int(input())
for _ in range(n):
    parts = input().split()
    cmd = parts[0]
    if cmd == "show":
        print(f"mask {mask:03b}")
    elif cmd == "has":
        print("yes" if mask & BITS[parts[1]] else "no")
    elif cmd == "add":
        mask |= BITS[parts[1]]
    elif cmd == "remove":
        mask &= ~BITS[parts[1]]
```

EXERCISE 2

Precedence versus left to right

Read integer expressions written with spaces between tokens, using the operators `+ - * // % **`. For each, print the value Python gives (evaluate it with `eval` — the input is trusted here) and the value a strictly left-to-right reading would give, applying each operator as it is met with no precedence at all.

Input: `n`, then `n` expressions. **Output:** `<expression>: precedence <p>, left-to-right <l>` per line.

```
2
2 + 3 * 4
2 ** 3 ** 2
```

prints

```
2 + 3 * 4: precedence 14, left-to-right 20
2 ** 3 ** 2: precedence 512, left-to-right 64
```

INPUT 1

```
2
2 + 3 * 4
2 ** 3 ** 2
```

OUTPUT 1

```
2 + 3 * 4: precedence 14, left-to-right
20
2 ** 3 ** 2: precedence 512, left-to-
right 64
```

REFERENCE SOLUTION

```python
import operator

OPS = {
    "+": operator.add,
    "-": operator.sub,
    "*": operator.mul,
    "//": operator.floordiv,
    "%": operator.mod,
    "**": operator.pow,
}


def left_to_right(tokens):
    value = int(tokens[0])
    for i in range(1, len(tokens), 2):
        fn = OPS[tokens[i]]
        value = fn(value, int(tokens[i + 1]))
    return value


n = int(input())
for _ in range(n):
    expr = input()
    tokens = expr.split()
    print(f"{expr}: precedence {eval(expr)}, left-to-right {left_to_right(tokens)}")
```

QUIZ

1. What is `2 ** 3 ** 2`?

A. `64`

B. `512`

C. `36`

D. `SyntaxError`

2. How does `x & 1 == 0` parse?

A. `(x & 1) == 0`

B. `x & (1 == 0)`

C. It is a syntax error

D. `x & 1` then compared with `0` left to right

3. What does `x & (x - 1) == 0` test for a positive integer `x` (once parenthesised as `(x & (x - 1)) == 0`)?

A. `x` is even

B. `x` is a power of two

C. `x` is odd

D. `x` has exactly two set bits

4. What does `~5` evaluate to?

A. `-5`

B. `-6`

C. `250`

D. `2`

5. What does `3 < "3"` do?

A. Returns `True`

B. Returns `False`

C. Raises `TypeError`

D. Compares the strings `"3"` and `"3"`

6. Which is the safe way to turn the text `"[1, 2, 3]"` into a list?

A. `eval(text)`

B. `ast.literal_eval(text)`

C. `list(text)`

D. `text.split(", ")`

ANSWER KEY · OPERATORS AND PRECEDENCE

1. B `**` is right-associative, so it is `2 ** (3 ** 2) = 2 ** 9`. Every other arithmetic operator associates left.

2. B Comparisons sit *below* the bitwise operators in Python's table, so `1 == 0` (which is `False`) is computed first and then anded with `x`. Always parenthesise a bitwise sub-expression inside a comparison.

3. B Subtracting one flips the lowest set bit and everything below it; anding with the original clears that lowest set bit. The result is zero only when there was exactly one set bit — a power of two.

4. B Python integers behave as infinite two's-complement values, and `~x` is `-x - 1`. There is no fixed width, so `250` (an 8-bit complement) is not what you get.

5. C Ordering comparisons between unrelated types are not defined in Python 3 and raise `TypeError`. Only `==` / `≠` work across types, and they simply report inequality.

6. B `literal_eval` parses literals only and never runs code, unlike `eval`. `list(text)` gives the characters, and splitting leaves brackets and strings rather than integers.

CHECKPOINT · 25 MIN

Checkpoint — Values, types and operators

This checkpoint covers the whole module: arbitrary-precision integers and the floor semantics of `//` and `%`, floating point and the three rules for comparing, printing and summing floats, truthiness and what `and` / `or` return, names bound to objects and the difference between rebinding and mutation, the explicit conversions and their errors, and operator precedence with the bitwise idioms.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- What are `-7 // 2` and `-7 % 2`, and which invariant relates them to `-7`?
- Why is `0.1 + 0.2 == 0.3` false, and what do you write instead?
- What does `0 or "x"` evaluate to, and what does `3 and 0` evaluate to?
- After `a = [1]; b = a; b += [2]`, what is `a`? And after `b = b + [3]`?
- Which exception does `int("3.0")` raise, and what two-step conversion works?
- What is `-2 ** 2`, and why?
- What does `x & (x - 1) == 0` test, once parenthesised correctly?

The three programs are a digit and bit report that uses integer arithmetic and the base-conversion functions, a temperature table that formats floats to fixed widths and picks labels with chained comparisons, and an exact-change calculator that keeps money in integer cents.

EXERCISE 1

Digit and bit report

Read a non-negative integer (it may be very large) and print six facts about it: its number of decimal digits, the sum of its digits, its digital root (sum the digits repeatedly until one digit remains; the root of 0 is 0), its binary form without the `0b` prefix, the number of one bits, and the number of bits needed to hold it (`bit_length`, which is 0 for 0).

Input: one integer. **Output:** six lines: `digits`, `digit-sum`, `digital-root`, `binary`, `ones`, `bits`, each followed by a space and the value.

```
12345
```

prints

```
digits 5
digit-sum 15
digital-root 6
binary 11000000111001
ones 6
bits 14
```

INPUT 1

```
12345
```

OUTPUT 1

```
digits 5
digit-sum 15
digital-root 6
binary 11000000111001
ones 6
bits 14
```

REFERENCE SOLUTION

```python
n = int(input())
text = str(n)
digit_sum = sum(int(d) for d in text)
root = n
while root >= 10:
    root = sum(int(d) for d in str(root))
print(f"digits {len(text)}")
print(f"digit-sum {digit_sum}")
print(f"digital-root {root}")
print(f"binary {n:b}")
print(f"ones {n.bit_count()}")
print(f"bits {n.bit_length()}")
```

EXERCISE 2

Temperature table

Read Celsius temperatures, one per line, until the end of input, and print a table row for each: the Celsius value and the Fahrenheit value (`c * 9 / 5 + 32`) both right-aligned in 7 characters with

one decimal, then two spaces and a label — `freezing` when `c ⩽ 0`, `cold` when `0 < c < 15`, `warm` when `15 ⩽ c < 25`, `hot` otherwise. Use chained comparisons for the bands.

Input: zero or more numbers. **Output:** one row per number.

```
-3
22
```

prints

```
   -3.0    26.6  freezing
   22.0    71.6  warm
```

INPUT 1

```
-3
0
14.9
22
30
```

OUTPUT 1

```
   -3.0    26.6  freezing
    0.0    32.0  freezing
   14.9    58.8  cold
   22.0    71.6  warm
   30.0    86.0  hot
```

REFERENCE SOLUTION

```python
import sys


def label(c):
    if c ⩽ 0:
        return "freezing"
    if 0 < c < 15:
        return "cold"
    if 15 ⩽ c < 25:
        return "warm"
    return "hot"


for line in sys.stdin:
    c = float(line)
    f = c * 9 / 5 + 32
    print(f"{c:7.1f} {f:7.1f}  {label(c)}")
```

EXERCISE 3

Exact change

A customer pays for an item; compute the change in coins using integer cents throughout. Denominations, in cents: 2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1. Use the greedy method (largest coin first). If the payment is smaller than the price print `insufficient`.

Input: one line `paid price`, both with exactly two decimals. **Output:** `change <amount>` with two decimals, then one line per denomination used, largest first: `<coin with two decimals> x <count>`.

```
20.00 13.45
```

prints

```
change 6.55
5.00 x 1
1.00 x 1
0.50 x 1
0.05 x 1
```

INPUT 1

```
20.00 13.45
```

OUTPUT 1

```
change 6.55
5.00 x 1
1.00 x 1
0.50 x 1
0.05 x 1
```

REFERENCE SOLUTION

```
COINS = [2000, 1000, 500, 200, 100, 50, 20, 10, 5, 2, 1]


def to_cents(text):
    pounds, pence = text.split(".")
    return int(pounds) * 100 + int(pence)


def money(cents):
    return f"{cents // 100}.{cents % 100:02d}"


paid, price = input().split()
change = to_cents(paid) - to_cents(price)
if change < 0:
    print("insufficient")
else:
    print(f"change {money(change)}")
    for coin in COINS:
        count, change = divmod(change, coin)
        if count:
            print(f"{money(coin)} x {count}")
```

CHECKPOINT QUESTIONS

1. What does `print(7 // -2, 7 % -2)` print?

A. `-3 1`

B. `-4 -1`

C. `-3 -1`

D. `-4 1`

2. Which comparison of two computed floats is correct?

A. `a == b`

B. `abs(a - b) == 0`

C. `math.isclose(a, b)`

D. `str(a) == str(b)`

3. What does `print(0 or [], [] or 0, 1 and 2)` print?

A. `False False True`

B. `[] 0 2`

C. `0 [] 1`

D. `[] [] 2`

4. After `a = [1]; b = a; b += [2]; b = b + [3]`, what is `a`?

A. `[1]`

B. `[1, 2]`

C. `[1, 2, 3]`

D. `[1, 3]`

5. What does `int(" 42 ")` return?

A. `42`

B. `ValueError`

C. `" 42 "`

D. `4`

6. What is `2 ** -1`?

A. `0`

B. `0.5`

C. `-2`

D. `ValueError`

7. Which value is falsy?

A. `"0"`

B. `[None]`

C. `0.0`

D. `-1`

8. What does `x = 5; x is 5` do, and why is it discouraged?

A. It is `False` because `is` compares types

B. It happens to be `True` because CPython caches small ints, but identity is not the right question for values — use `==`

C. It raises `TypeError`

D. It is always `True` for every integer

9. Which produces the string `'0b101'`?

A. `format(5, "b")`

B. `bin(5)`

C. `f"{5:b}"`

D. `str(5, 2)`

10. What does `Decimal("0.1") + Decimal("0.2") == Decimal("0.3")` evaluate to?

A. `True`

B. `False`

C. `TypeError`

D. `Decimal('0.3')`

11. What does `(1 << 3) | 1` evaluate to?

A. `8`

B. `9`

C. `3`

D. `16`

12. A function receives a list and does `xs.append(1)`; another receives a string and does `s += "!"`. Which caller sees a change?

A. Both

B. Neither

C. Only the list's caller

D. Only the string's caller

ANSWER KEY · CHECKPOINT — VALUES, TYPES AND OPERATORS

1. B Floor of −3.5 is −4, and the remainder takes the divisor's sign: `-2 * -4 + (-1) == 7`.

2. C `isclose` allows a relative tolerance (and an absolute one near zero); the other three all reduce to exact equality of values that carry rounding error.

3. B `or` returns the last operand when the first is falsy: `[]` then `0`. `and` returns the second operand when the first is truthy: `2`.

4. B `+=` extends the shared list in place, so `a` sees `2`; `b + [3]` creates a new list bound only to `b`.

5. A Surrounding whitespace is allowed by `int`. Inner whitespace (`"4 2"`) and a decimal point are not.

6. B A negative exponent on integers yields a float reciprocal. Only non-negative integer exponents keep the result an int.

7. C Zero of any numeric type is falsy. A non-empty string or list is truthy whatever it contains, and every non-zero number is truthy.

8. B Small integers are cached so the identity test passes by accident; for `1000` it may not. The interpreter emits `SyntaxWarning` for `is` with a literal. `is` is for `None` and sentinels.

9. B `bin` includes the `0b` prefix; the format specs give bare digits `101`; `str(5, 2)` is a `TypeError` (that signature decodes bytes).

10. A Decimal arithmetic on values constructed from strings is exact at these precisions, so the sum is exactly three tenths. `==` returns a bool, not a Decimal.

11. B `1 << 3` is 8 (binary 1000); or-ing in bit 0 gives 1001, which is 9.

12. C `append` mutates the shared list object. `s += "!"` creates a new string and rebinds the local name only; strings are immutable and the caller's binding is untouched.

CodeKairo · Python study plan · Module 2: Values, types and operators. Solve the exercises online to track your progress and earn XP.