Module 3 of 20

Control flow

if/elif/else and guard clauses, while with break, continue and else, for over any iterable with range, enumerate and zip, structural pattern matching with match, and the loop patterns interviews are built from.

Python has one conditional statement and two loops, and the craft is in the shapes: a guard clause that returns early instead of nesting, a `while True` whose exit is in the middle, a `for` that walks the elements rather than the indices, a `match` that takes a command line apart by its structure. This module teaches the statements and, with each one, the shape a fluent Python programmer reaches for and the built-in that replaces the loop altogether — `sum`, `max` with a key, `any`, `in`, `enumerate`, `zip`.

Conditionals covers truth testing in a condition, guard clauses, membership and identity tests, the conditional expression and the dictionary that replaces an `elif` chain. While loops covers the sentinel and EOF loops, `break` and `continue`, and the loop `else` that only Python has. For loops fixes `range`'s half-open rule, `enumerate`, `zip` with `strict=True`, dictionary iteration and the rule against modifying a collection you are iterating. Match statements gives the pattern kinds, guards, and the capture-versus-value rule. Loop patterns catalogues accumulate, count, extreme, search, running state, two pointers, prefix sums and building output.

The exercises are whole programs: a leap-year rule, a triangle classifier with guard clauses, the Collatz sequence, a `while … else` search, a multiplication table, a ranked list built with `enumerate` over a strict `zip`, a command dispatcher and a shape calculator written with `match`, a longest-streak finder and a two-pointer pair search. The checkpoint adds a configurable FizzBuzz, a prime lister on trial division with `for … else`, and a bank account whose commands are dispatched with patterns and guards.

CONTENTS

1	Conditionals — if, elif, else and the shapes that read well	12 min
2	While loops — repeat until a condition changes	12 min
3	For loops, range, enumerate and zip	14 min
4	Match statements — structural pattern matching	14 min

5	The loop patterns	14 min
✓	Checkpoint — Control flow	22 min

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

LESSON 1 · 12 MIN

Conditionals — if, elif, else and the shapes that read well

Every branch in a Python program is an `if`, and the language has exactly one form of it: a condition, a colon, an indented block, optionally `elif` blocks and an `else`. What makes conditional code good or bad is not the syntax but the shape — flat rather than nested, guard clauses that return early, conditions that read as sentences — and this lesson is about those shapes as much as the keywords. It covers the statement, the truth testing behind it, the conditional expression, membership and identity tests as conditions, and the refactorings that turn a five-level nest into a flat list of cases.

The statement

```
if score ⩾ 90:
    grade = "A"
elif score ⩾ 80:
    grade = "B"
elif score ⩾ 70:
    grade = "C"
else:
    grade = "F"
```

Conditions are tried top to bottom and the first true one wins; `elif` is `else if` written so that the blocks do not nest deeper and deeper. `else` is optional and catches everything the earlier conditions rejected. There are no parentheses around the condition (they are allowed and idle), and no braces: the block is the indented lines.

The condition is any expression; it is truth-tested by the rules of Module 2 — `if xs:` means non-empty, `if n:` means non-zero, `if s.strip():` means "has something other than whitespace". Writing `if xs ≠ []:` or `if len(xs) > 0:` is legal and slightly foreign; `if flag == True:` is a habit to drop, because `if flag:` says the same thing and also works for any truthy value.

Guard clauses

Nesting grows when every condition wraps the rest of the function. Returning early on the exceptional cases leaves the main path unindented:

```
def price(item, qty):                       # nested
    if item is not None:
        if qty > 0:
            if item.in_stock:
                return item.unit * qty
            else:
                return None
        else:
            return None
    else:
        return None

def price(item, qty):                       # guarded
    if item is None or qty ⩽ 0 or not item.in_stock:
        return None
    return item.unit * qty
```

A guard says "if this is not a case I handle, leave now", and the remaining code can assume the happy path. The same shape works with `continue` inside a loop body and with `raise` for invalid input (Module 10). The rule of thumb: handle the *exits* first, the *work* last.

Chained conditions

`and`, `or` and `not` combine conditions and short-circuit, which lets a later condition depend on an earlier one being true: `if xs and xs[0] > 0:` never indexes an empty list. Comparison chaining (`lo ⩽ x ⩽ hi`) expresses a range without repeating `x`. Membership tests replace runs of `==`:

```
if cmd == "quit" or cmd == "exit" or cmd == "q":     # verbose
if cmd in ("quit", "exit", "q"):                     # one test
if ch in "aeiou":                                    # substring/character tes
if key in table:                                     # dictionary key test
```

A tuple literal for the options is conventional; a set `{...}` is equally fine and faster for many options. `x is None` and `x is not None` are the identity conditions; `isinstance(x, int)` is the type condition (and it is true for `bool` too, because `bool` subclasses `int`).

The conditional expression

`a if condition else b` is an expression, so it can sit where a statement cannot — in an argument, an f-string, a return, a comprehension:

```
label = "even" if n % 2 == 0 else "odd"
print(f"{count} item{'s' if count != 1 else ''}")
return xs[0] if xs else None
```

The condition reads in the middle, which mirrors English ("the first element, if there is one, else nothing"). Use it for a choice between two *values*; when the branches are actions or the choice has three or more arms, write the statement — `a if c1 else b if c2 else d` is legal and a reviewer will ask you to rewrite it.

Dispatch tables

A long `elif` chain that maps a key to a value or a function is often a dictionary:

```
if op == "+": result = a + b
elif op == "-": result = a - b
elif op == "*": result = a * b

import operator
OPS = {"+": operator.add, "-": operator.sub, "*": operator.mul}
result = OPS[op](a, b)          # KeyError on an unknown op — or OPS.get(op) (
```

The dictionary is data: adding an operator is one line, and the mapping can be tested and reused. Module 3 lesson 4 introduces `match`, which handles the cases a dictionary cannot — patterns over structure rather than equality on one key.

Conditions that are not booleans

Beware three conditions that are true when you think they are false. A non-empty string is always true, so `if input():` is true for `"no"`, `"0"` and `"False"`; compare against the value you mean. A function object is always true, so `if is_ready:` (forgetting the call) never takes the else branch; write `if is_ready():`. And a comparison written like mathematics can be a chain in disguise: `if a < b > c` is legal, means `a < b and b > c`, and surprises the reader.

Pitfalls

`if x == True:` and `if x == None:`. Write `if x:` and `if x is None:`.

Assignment where a comparison was meant: `if x = 5:` is a syntax error (a deliberate design), but `if (x := 5):` is legal and always true.

Deep nesting that guard clauses would flatten.

An `elif` chain where the order matters and the general case comes first, shadowing the specific ones.

`if s.isdigit():` before `int(s)` rejecting negative numbers.

Relying on truthiness when zero or empty is a legitimate value.

Key takeaways

`if` / `elif` / `else` tries conditions in order; the condition is any expression, truth-tested.

Guard clauses — return, continue or raise early on the exceptional cases — keep the main path flat.

`and` / `or` short-circuit, comparisons chain, `in` tests membership, `is` tests identity, `isinstance` tests type.

`a if c else b` chooses between two values; use the statement for actions or more arms.

A dictionary can replace an `elif` chain that maps keys to values or functions.

EXERCISE 1

Leap years

A year is a leap year when it is divisible by 4, except that years divisible by 100 are not — unless they are also divisible by 400. Read `n` years and classify each with one boolean expression built from `and`, `or` and `not` (no nested `if`s).

Input: `n`, then `n` years. **Output:** `<year> leap` or `<year> common` per year.

```
3
2000
1900
2024
```

prints

```
2000 leap
1900 common
2024 leap
```

INPUT 1

```
3
2000
1900
2024
```

OUTPUT 1

```
2000 leap
1900 common
2024 leap
```

REFERENCE SOLUTION

```python
def is_leap(year):
    return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)


n = int(input())
for _ in range(n):
    year = int(input())
    print(year, "leap" if is_leap(year) else "common")
```

EXERCISE 2

Triangle classifier

Read three integer side lengths and classify the triangle. Use guard clauses: if any side is not positive, or the two shorter sides do not add up to more than the longest, print `invalid` and stop. Otherwise print `equilateral`, `isosceles` or `scalene`, followed by `right` when the sides satisfy `a² + b² = c²` with `c` the longest.

Input: one line `a b c`. **Output:** one line.

```
3 4 5
```

prints

```
scalene right
```

INPUT 1

```
3 4 5
```

OUTPUT 1

```
scalene right
```

INPUT 2

```
2 2 2
```

OUTPUT 2

```
equilateral
```

REFERENCE SOLUTION

```
a, b, c = sorted(map(int, input().split()))
if a ⩽ 0 or a + b ⩽ c:
    print("invalid")
else:
    if a == b == c:
        kind = "equilateral"
    elif a == b or b == c:
        kind = "isosceles"
    else:
        kind = "scalene"
    if a * a + b * b == c * c:
        kind += " right"
    print(kind)
```

QUIZ

1. What does this print?

```
xs = []
if xs:
    print("some")
elif xs == []:
    print("empty list")
else:
    print("other")
```

A. `some`

B. `empty list`

C. `other`

D. Nothing

2. Which rewrite of `if cmd == "q" or cmd == "quit" or cmd == "exit":` is idiomatic?

A. `if cmd in ("q", "quit", "exit"):`

B. `if cmd == ("q" or "quit" or "exit"):`

C. `if cmd == "q" | "quit" | "exit":`

D. `if cmd is "q" or "quit" or "exit":`

3. What is a guard clause?

A. A `try` block around risky code

B. An early `return`, `continue` or `raise` that handles an exceptional case so the main path stays unindented

C. An `else` at the end of an `if` chain

D. A condition wrapped in parentheses

4. What does `print("yes" if 0 else "no")` print?

A. `yes`

B. `no`

C. `0`

D. `SyntaxError`

5. What is wrong with `if is_valid:` where `is_valid` is a function?

A. Nothing — Python calls it automatically

B. A function object is always truthy, so the branch always runs; the call `is_valid()` was meant

C. It raises `TypeError`

D. It is a syntax error

ANSWER KEY · CONDITIONALS — IF, ELIF, ELSE AND THE SHAPES THAT READ WELL

1. B An empty list is falsy, so the first branch is skipped; the `elif` compares by value and matches. Only the first true branch runs.

2. A Membership in a tuple tests each alternative. `"q" or "quit"` evaluates to `"q"` alone, `|` on strings is a `TypeError`, and `cmd is "q" or "quit"` is always truthy because the second operand is a non-empty string.

3. B Handling the exits first means the rest of the function can assume the happy path, which flattens nesting. The other options are unrelated constructs.

4. B The conditional expression tests its middle operand; `0` is falsy so the `else` value is chosen.

5. B Without parentheses the name refers to the function itself, which is truthy like any object without `__bool__`. The bug is silent, which is what makes it worth remembering.

LESSON 2 · 12 MIN

While loops — repeat until a condition changes

A `while` loop runs its body as long as its condition is true, and that is the whole statement — Python has no `do … while` and no C-style `for (init; test; step)`. The consequence is that `while` is reserved for the loops where you *do not know in advance how many times* you will go round: reading until a sentinel, iterating a numerical process to convergence, walking a linked structure, retrying until success. Everything with a known count or a known collection is a `for` (next lesson). This lesson covers the statement, `break` and `continue`, the `else` clause that only `while` and `for` have, and the three loop shapes that come up again and again.

The statement

```
n = 27
steps = 0
while n ≠ 1:
    n = n // 2 if n % 2 == 0 else 3 * n + 1
    steps += 1
print(steps)          # 111
```

The condition is tested *before* each iteration, including the first — a `while` whose condition is false at the start runs zero times. Something in the body must change what the condition depends on, or the loop never ends; on the judge that is a time-limit verdict, and locally it is Ctrl-C.

break, continue and the infinite loop

`break` leaves the loop immediately; `continue` skips to the next test. With `while True:` and `break` you get the loop whose exit is in the middle, which is the Python spelling of `do … while`:

```
while True:
    line = input()
    if line == "END":
        break
    process(line)
```

The body runs at least once, the exit condition is tested after reading, and the reading is written only once — the alternative, priming the loop with a read before it and repeating the read at the bottom, is longer and error-prone. The walrus operator collapses the common case (Module 2): `while (line := input()) ≠ "END":`.

`continue` is the loop's guard clause: reject this iteration and move on, leaving the rest of the body flat.

```
while queue:
    item = queue.pop()
    if item.done:
        continue
    handle(item)
```

The else clause

`while … else` runs the `else` block when the loop ends *because the condition became false* — not when it was left by `break`. It is the natural place for "we searched and did not find":

```
i = 0
while i < len(xs):
    if xs[i] == target:
        print(f"found at {i}")
        break
    i += 1
else:
    print("not found")
```

Without `else` you would need a `found` flag set inside the loop and tested after it. The clause is rarely seen, which is the main argument against it; readers who know it find it clearer than the flag.

Reading until a sentinel or EOF

A **sentinel** is a value that marks the end — `0`, `-1`, `END`, an empty line. **EOF** is the end of the stream itself, signalled by `EOFError` from `input()` or by `sys.stdin` running out:

```
total = 0
while True:                             # sentinel 0
    n = int(input())
    if n == 0:
        break
    total += n

try:                                    # until EOF
    while True:
        line = input()
        total += int(line)
except EOFError:
    pass
```

For EOF the `for line in sys.stdin:` loop is shorter and is what the later modules use; the `while` form is here because it shows where `EOFError` comes from.

Numerical loops

Iterate until the change is small enough — the loop whose count is genuinely unknown:

```
x = n / 2
while abs(x * x - n) > 1e-9:          # Newton's method for a square root
    x = (x + n / x) / 2
```

Two guards belong in a real version: a maximum iteration count (`for _ in range(100)` with a `break` is often the cleaner spelling), and a tolerance chosen with Module 2's floating-point rules in mind.

Counting loops — and why they are usually for loops

`i = 0; while i < n: …; i += 1` works, and `for i in range(n):` is what Python programmers write, because the count is known. The `while` version has two failure modes the `for` version cannot have: forgetting the increment (an infinite loop) and mutating `i` inside the body (a skipped or repeated element). Reserve `while` for the loops where the step is not a fixed increment — a search that jumps by variable amounts, a two-pointer sweep where either pointer may move, a queue drained until empty.

Pitfalls

A condition that nothing in the body changes.

`while x = 5:` is a syntax error; `while x == 5:` is the test.

Off-by-one in `while i <= len(xs):` — the last valid index is `len(xs) - 1`.

Expecting `else` to run after a `break`. It runs only after a normal exit.

A `try` around the whole loop swallowing the exception you wanted to see, so the loop spins on `EOFError` forever.

Floating-point equality as an exit test (`while x != 0.3`). Use a tolerance.

Key takeaways

`while` tests before each iteration and runs zero or more times; use it when the count is unknown.

`while True:` + `break` is the loop with its exit in the middle; `continue` skips an iteration.

`else` on a loop runs only when the loop ended without `break` — the "not found" branch.

Sentinel loops break on the marker value; EOF loops catch `EOFError` (or use `for line in sys.stdin`).

Known counts and collections are `for` loops; `while` is for searches, convergence and queues.

EXERCISE 1

Collatz

Starting from `n`, repeatedly replace it with `n // 2` if it is even and `3 * n + 1` if it is odd, until it reaches 1. Count the steps and track the largest value seen (including the start). The count is unknown in advance — this is a `while` loop.

Input: one integer `n >= 1`. **Output:** `steps <count>` then `peak <max>`.

```
6
```

prints

```
steps 8
peak 16
```

INPUT 1

```
6
```

OUTPUT 1

```
steps 8
peak 16
```

INPUT 2

```
27
```

OUTPUT 2

```
steps 111
peak 9232
```

REFERENCE SOLUTION

```python
n = int(input())
steps = 0
peak = n
while n != 1:
    n = n // 2 if n % 2 == 0 else 3 * n + 1
    steps += 1
    peak = max(peak, n)
print(f"steps {steps}")
print(f"peak {peak}")
```

EXERCISE 2

Search with while–else

Find the first position of a target in a list using an index-driven `while` loop that `break`s when it finds it, and a loop `else` that reports the failure — no flag variable.

Input: the target on the first line, then a line of integers (possibly empty). **Output:** `found at <index>` or `not found`.

```
4
1 3 4 4 9
```

prints

```
found at 2
```

INPUT 1

```
4
1 3 4 4 9
```

OUTPUT 1

```
found at 2
```

INPUT 2

```
7
1 2
```

OUTPUT 2

```
not found
```

REFERENCE SOLUTION

```python
target = int(input())
xs = list(map(int, input().split()))
i = 0
while i < len(xs):
    if xs[i] == target:
        print(f"found at {i}")
        break
    i += 1
else:
    print("not found")
```

QUIZ

1. How many times does the body run?

```python
n = 0
while n > 0:
    n -= 1
```

A. Once

B. Zero times

C. Forever

D. It is a syntax error

2. When does the `else` block of a `while` loop run?

A. After every iteration

B. When the loop exits because its condition became false — not after a `break`

C. Only after a `break`

D. When the condition was false the first time it was tested

3. What does this print?

```
i = 0
while True:
    i += 1
    if i % 2:
        continue
    if i > 6:
        break
    print(i, end=" ")
```

A. `2 4 6`

B. `1 2 3 4 5 6`

C. `2 4 6 8`

D. `1 3 5`

4. A judged program loops with `while True: line = input()` inside a `try/except EOFError` that wraps only the `input()` call and does nothing. What happens at end of input?

A. The loop ends normally

B. It spins forever raising and swallowing `EOFError`, and the judge reports a time limit

C. `input()` returns `""` and the loop continues with empty lines

D. The program exits with status 1

5. Which loop is the idiomatic spelling of 'repeat exactly n times'?

A. `i = 0\nwhile i < n:\n ...\n i += 1`

B. `for _ in range(n):\n ...`

C. `while n:\n ...\n n -= 1`

D. `for i in n:\n ...`

ANSWER KEY · WHILE LOOPS — REPEAT UNTIL A CONDITION CHANGES

1. B The condition is tested before the first iteration and `0 > 0` is false, so the body never runs. Python has no `do … while` that would run it once.

2. B The `else` is the normal-completion branch. A `break` skips it, which is what makes it the natural place for "not found". It also runs when the loop ran zero times, since that is a normal exit too — so the last option is incomplete rather than wrong.

3. A Odd values are skipped by `continue`; even ones print until `i` exceeds 6, when the `break` fires before printing 8.

4. B Swallowing the exception inside the loop hides the only signal that input is exhausted, so the loop never terminates. The `try` must enclose the whole loop, or the handler must `break`.

5. B A known count is a `for` over `range`; the underscore says the value is unused. The `while` versions work but invite forgotten increments, and `for i in n` is a `TypeError` because an int is not iterable.

LESSON 3 · 14 MIN

For loops, range, enumerate and zip

Python's `for` is a *for-each*: it takes an iterable — a list, a string, a range, a dictionary, a file, a generator — and binds the loop variable to each element in turn. There is no index unless you ask for one, and asking is done with `enumerate`; walking two sequences together is `zip`; counting is `range`. Learners from C write `for i in range(len(xs)): x = xs[i]`, which works and marks them as learners. This lesson fixes the idioms, explains `range` precisely, and states the one rule about modifying a collection while iterating over it.

The statement

```
for ch in "abc":
    print(ch)

for name in ["ada", "grace"]:
    print(name.title())

for key in {"a": 1, "b": 2}:          # a dict iterates its keys
    print(key)
```

The loop variable is an ordinary name bound in the enclosing scope; it keeps its last value after the loop (`for i in range(3): pass; print(i)` prints `2`), and rebinding it inside the body does not affect the iteration — the next value comes from the iterable, not from the variable. `break`, `continue` and `else` work exactly as in `while`: `else` runs when the iterable is exhausted without a `break`.

```
for x in xs:
    if x < 0:
        print("has a negative")
        break
else:
    print("all non-negative")
```

range

`range(stop)`, `range(start, stop)`, `range(start, stop, step)` — half-open, like slicing: `start` included, `stop` excluded.

```
list(range(5))            # [0, 1, 2, 3, 4]
list(range(2, 5))         # [2, 3, 4]
list(range(0, 10, 3))     # [0, 3, 6, 9]
list(range(5, 0, -1))     # [5, 4, 3, 2, 1]
list(range(3, 3))         # []
```

Half-open ranges compose: `range(0, n)` then `range(n, m)` cover `0 … m-1` exactly once; the count of elements is `stop - start` for step 1; `range(len(xs))` is exactly the valid indices. A `range` is not a list — it is a small object that computes elements on demand, so `range(10 ** 9)` costs nothing until iterated, and `5 in range(0, 100, 5)` is answered arithmetically. Counting down is `range(n - 1, -1, -1)`, or `reversed(range(n))`, which reads better.

The loop `for _ in range(n):` is the idiom for "n times, the number does not matter"; the underscore is a convention meaning "unused".

---

enumerate: index and element

```
for i, name in enumerate(names):          # 0-based
    print(i, name)

for i, name in enumerate(names, start=1): # 1-based, for display
    print(f"{i}. {name}")
```

`enumerate` yields `(index, element)` pairs, unpacked into two names by the `for`. Reach for it whenever the body needs the position — to print a line number, to compare with a neighbour `xs[i + 1]`, to record where a match was found. `for i in range(len(xs))` is the older spelling and is right only when the body needs the index *and not* the element, or must write `xs[i] = …`.

---

zip: parallel sequences

```python
for name, score in zip(names, scores):
    print(f"{name}: {score}")

for i, (name, score) in enumerate(zip(names, scores), start=1):
    ...
```

`zip` stops at the *shortest* input, silently; `zip(a, b, strict=True)` (3.10) raises `ValueError` on a length mismatch, which is what you want when the sequences are supposed to correspond. `dict(zip(keys, values))` builds a mapping from two lists; `zip(*grid)` transposes a list of rows into columns (Module 6). `itertools.zip_longest` pads the shorter one instead of truncating.

Iterating dictionaries

`for k in d` gives keys; `for v in d.values()` values; `for k, v in d.items()` both. Insertion order is preserved (since 3.7), so a dictionary iterates in the order its keys were added. `for k in sorted(d)` iterates the keys in sorted order.

Nested loops

```python
for r in range(rows):
    for c in range(cols):
        visit(r, c)
```

Leaving both loops at once has no `break 2`. The options: put the loops in a function and `return`; set a flag and test it in the outer loop; or iterate the pairs with `itertools.product(range(rows), range(cols))`, which makes it one loop with one `break`. The function is the cleanest when the search *is* a unit of work.

Modifying while iterating

Do not add to or remove from a list, dict or set while a `for` loop is iterating over it. Removing shifts the elements and the loop skips one; adding can make it infinite; a dict raises `RuntimeError: dictionary changed size during iteration`. The fixes are to iterate over a copy (`for x in list(xs):`), to build a new collection (`kept = [x for x in xs if keep(x)]`

— the comprehension is almost always what you meant), or to collect the changes and apply them after the loop. Modifying *elements in place* (`xs[i] = …`) is fine.

Pitfalls

`for i in range(len(xs)): x = xs[i]` where `for x in xs` or `enumerate` was meant.

`range(1, n)` when `n` should be included: the stop is excluded.

`zip` truncating silently when the lists should have matched — `strict=True`.

`xs.remove(x)` inside `for x in xs`.

Expecting a `for` loop to have its own scope; the loop variable and everything assigned in the body are visible after it.

`for k, v in d:` — iterating a dict gives keys; `.items()` gives pairs.

Key takeaways

`for` iterates the elements of any iterable; `break`, `continue`, `else` as with `while`.

`range(start, stop, step)` is half-open and lazy; `range(len(xs))` is the valid indices; `for _ in range(n)` repeats.

`enumerate` for index and element, `zip` for parallel sequences (`strict=True` to check lengths), `.items()` for dict pairs.

Leave nested loops with a function `return` or `itertools.product`.

Never change a collection's size while iterating it; iterate a copy or build a new one.

EXERCISE 1

Multiplication table

Print the `n × n` multiplication table with nested `for` loops over `range(1, n + 1)`. Every entry is right-aligned in a field as wide as the largest product, and entries on a row are separated by one space.

Input: `n` (1–12). **Output:** `n` rows.

```
4
```

prints

```
 1  2  3  4
 2  4  6  8
 3  6  9 12
 4  8 12 16
```

INPUT 1

```
4
```

OUTPUT 1

```
 1  2  3  4
 2  4  6  8
 3  6  9 12
 4  8 12 16
```

INPUT 2

```
3
```

OUTPUT 2

```
1 2 3
2 4 6
3 6 9
```

REFERENCE SOLUTION

```python
n = int(input())
width = len(str(n * n))
for r in range(1, n + 1):
    row = [f"{r * c:{width}}" for c in range(1, n + 1)]
    print(" ".join(row))
```

EXERCISE 2

Ranked pairs

Two lines hold names and integer scores in matching order. Print them numbered from 1 using `enumerate` over `zip(names, scores, strict=True)`, then the name with the highest score (the first one on a tie). If the two lines have different lengths, `strict=True` raises `ValueError` — catch it and print `length mismatch` instead.

Input: a line of names, then a line of integers. **Output:** `<i>. <name> <score>` per pair, then `best: <name>`; or the single line `length mismatch`.

```
ada grace linus
90 95 88
```

prints

```
1. ada 90
2. grace 95
3. linus 88
best: grace
```

INPUT 1

```
ada grace linus
90 95 88
```

OUTPUT 1

```
1. ada 90
2. grace 95
3. linus 88
best: grace
```

REFERENCE SOLUTION

```python
names = input().split()
scores = list(map(int, input().split()))
try:
    pairs = list(zip(names, scores, strict=True))
except ValueError:
    print("length mismatch")
else:
    for i, (name, score) in enumerate(pairs, start=1):
        print(f"{i}. {name} {score}")
    best_name, _ = max(pairs, key=lambda p: p[1])
    print(f"best: {best_name}")
```

QUIZ

1. What is `list(range(2, 10, 3))`?

A. `[2, 5, 8]`

B. `[2, 5, 8, 11]`

C. `[3, 6, 9]`

D. `[2, 4, 6, 8, 10]`

2. What does this print?

```
for i in range(3):
    i = 10
print(i)
```

A. `10`

B. `2`

C. `3`

D. `NameError`

3. What does `list(zip([1, 2, 3], "ab"))` produce?

A. `[(1, 'a'), (2, 'b'), (3, None)]`

B. `[(1, 'a'), (2, 'b')]`

C. `ValueError`

D. `[(1, 'a', 'b')]`

4. Which loop prints each element with its 1-based position?

A. `for i, x in enumerate(xs, start=1):`

B. `for i, x in enumerate(xs) + 1:`

C. `for x, i in enumerate(xs):`

D. `for i in range(1, len(xs)):`

5. What happens here?

```
xs = [1, 2, 3, 4]
for x in xs:
    if x % 2 == 0:
        xs.remove(x)
print(xs)
```

A. `[1, 3]`

B. `[1, 3, 4]` or another surprising result — removing shifts the elements the loop is walking

C. `RuntimeError`

D. `[2, 4]`

6. What does `for k in {"a": 1, "b": 2}: print(k)` print?

A. `a` then `b`

B. `1` then `2`

C. `('a', 1)` then `('b', 2)`

D. `TypeError`

ANSWER KEY · FOR LOOPS, RANGE, ENUMERATE AND ZIP

1. A Start at 2, step by 3, stop before 10: 2, 5, 8. The stop value is never included.

2. A Rebinding the loop variable inside the body does not affect the iteration (the next value comes from the range), and the variable survives the loop with its last binding, which was `10` in the final iteration.

3. B `zip` stops at the shortest input silently. `strict=True` would raise; `itertools.zip_longest` would pad with `None`.

4. A `enumerate` yields `(index, element)` and `start` sets the first index. The second is a `TypeError`, the third swaps the names, and the last skips the first element.

5. B Removing an element during iteration shifts later elements down, so the loop skips the one that moved into the removed slot. Lists do not raise (dicts do). Build a new list with a comprehension instead.

6. A Iterating a dictionary yields its keys, in insertion order. `.values()` gives values and `.items()` gives pairs.

LESSON 4 · 14 MIN

Match statements — structural pattern matching

`match` arrived in Python 3.10 and is not a `switch`. A `switch` compares one value against constants; `match` compares a value against *patterns* that describe its shape — a list of three elements whose first is `"move"`, a dictionary with a `"type"` key, an instance of `Point` whose `x` is zero — and binds the parts it names. It replaces the `if isinstance(...) and len(...) == 3 and cmd[0] == ...` staircases that parse commands, walk JSON-like data and dispatch on message types. This lesson gives the pattern kinds, the guard, the capture-versus-value rule that catches everyone once, and where `match` is and is not the right tool.

The statement

```
match command.split():
    case ["quit"]:
        return "bye"
    case ["go", direction]:
        return f"going {direction}"
    case ["drop", *items]:
        return f"dropping {', '.join(items)}"
    case _:
        return "unknown"
```

The subject (`command.split()`) is evaluated once; the `case` patterns are tried top to bottom; the first that matches runs its block and the statement ends — there is no fall-through and no `break`. `case _:` is the wildcard, matching anything, and it must be last. If nothing matches and there is no wildcard, nothing happens (no error).

The pattern kinds

Pattern	Matches	Binds
`42`, `"go"`, `None`, `True`	a value equal to the literal (`None` / `True` / `False` by identity)	—
`x`	anything	`x`
`_`	anything	—
`[a, b]`, `(a, b)`	a sequence of exactly two elements	`a`, `b`
`[first, *rest]`	a sequence of one or more	`first`, `rest` (a list)
`{"type": t}`	a mapping with key `"type"` (extra keys allowed)	`t`
`{"type": "add", **rest}`	a mapping whose `"type"` is `"add"`	`rest` (the other items)
`Point(x=0, y=y)`	an instance of `Point` with `x == 0`	`y`
`int()`, `str()`	an instance of that type	—
`"a" | "b"`	either alternative	—
`[x, y] as pair`	the sequence pattern, and also the whole	`pair`
`case p if cond:`	pattern `p`, then the guard must be true	as `p`

Sequence patterns match lists and tuples (but *not* strings, deliberately — a string would otherwise match as a sequence of characters). Mapping patterns match dicts, ignoring keys not mentioned. Class patterns use `isinstance` and then compare the named attributes; positional class patterns need `__match_args__`, which dataclasses define automatically (`case Point(0, y)`).

Capture versus value — the rule that bites

A bare name in a pattern **captures**; it does not compare. `case RED:` binds `RED` to whatever the subject is and always matches. To compare against a named constant, the name must be *dotted* — an attribute lookup:

```
class Colour:
    RED = 1

match code:
    case Colour.RED:      # value pattern: compared with ==
        ...
    case RED:             # capture pattern: matches anything, binds RED — a l
        ...
```

Literals, dotted names and the `|`-alternatives of those are value patterns; a plain identifier is a capture. A `case` with a capture pattern that makes later cases unreachable is a `SyntaxError` ("name capture makes remaining patterns unreachable"), which catches the worst version of the mistake.

Guards

`case [x, y] if x == y:` matches the shape, binds the names, then evaluates the guard; if the guard is false the next case is tried, with the bindings discarded. Guards express the conditions a pattern cannot — ranges, relations between captured values, membership:

```
match point:
    case (x, y) if x == 0 and y == 0:
        kind = "origin"
    case (x, 0) | (0, x):
        kind = "on an axis"
    case (x, y) if abs(x) == abs(y):
        kind = "diagonal"
    case _:
        kind = "elsewhere"
```

An or-pattern's alternatives must bind the same names (`x` in both), so that the body can use them.

Matching structured data

The strongest use is data that arrives as nested lists and dicts — parsed JSON, event records, tokens:

```
match event:
    case {"type": "click", "x": int(x), "y": int(y)}:
        handle_click(x, y)
    case {"type": "key", "key": str(key)} if len(key) == 1:
        handle_char(key)
    case {"type": "key", "key": key}:
        handle_special(key)
    case {"type": t}:
        raise ValueError(f"unknown event type {t!r}")
    case _:
        raise ValueError("not an event")
```

Each case reads as the shape it accepts, the type checks are inline (`int(x)` binds only if the value is an int), and the failure cases are explicit. The `if/elif` equivalent is three times longer and has the checks separated from the bindings.

Where match is not the tool

A plain value switch — `match op: case "+": … case "-": …` — works, but a dictionary of functions (Module 3 lesson 1) is shorter and reusable when the cases only map keys to results. Two or three simple conditions are still an `if`. And `match` on the *type* of a value alone (`case int(): … case str(): …`) is fine for a small closed set but is what polymorphism and `functools.singledispatch` (Module 11) exist for when the set is open.

Pitfalls

`case CONSTANT:` capturing instead of comparing. Use a dotted name or a literal.

Expecting a string to match `[a, b, c]`. Strings are excluded from sequence patterns.

Expecting fall-through, or writing `break`.

A `case _:` that is not last (`SyntaxError`).

Forgetting that a mapping pattern ignores extra keys — use `**rest` if the extras matter.

Different names in the alternatives of an or-pattern.

Key takeaways

`match` tests a subject against patterns in order, binds the parts they name, and runs the first case that fits — no fall-through.

Patterns: literals and dotted names compare, bare names capture, `_` ignores, `[...]` and `{...}` destructure, `Class(...)` checks type and attributes, `|` offers alternatives, `as` keeps the whole.

A guard `if` refines a case with any condition; a failed guard moves to the next case.

It shines on nested data and command parsing; for value-to-value maps use a dict, for two conditions use `if`.

A bare name in a `case` is a capture — the one rule to remember.

EXERCISE 1

Command dispatcher

Read commands until the end of input and dispatch each with a single `match` on `line.split()`: `move x y` (two integers) prints `moved to (x, y)`; `say` followed by one or more words prints `said: <the words>`; `color name` prints `color set to <name>`; `quit` prints `bye` and stops reading; anything else prints `unknown: <the line>`.

Input: commands, one per line. **Output:** one line per command, ending at `quit` if present.

```
move 3 4
say hello world
jump
quit
move 1 1
```

prints

```
moved to (3, 4)
said: hello world
unknown: jump
bye
```

INPUT 1

```
move 3 4
say hello world
jump
quit
move 1 1
```

OUTPUT 1

```
moved to (3, 4)
said: hello world
unknown: jump
bye
```

INPUT 2

```
color red
say hi
```

OUTPUT 2

```
color set to red
said: hi
```

REFERENCE SOLUTION

```python
import sys

for line in sys.stdin:
    line = line.rstrip("\n")
    match line.split():
        case ["move", x, y]:
            print(f"moved to ({int(x)}, {int(y)})")
        case ["say", first, *rest]:
            print("said:", " ".join([first, *rest]))
        case ["color", name]:
            print(f"color set to {name}")
        case ["quit"]:
            print("bye")
            break
        case _:
            print(f"unknown: {line}")
```

EXERCISE 2

Shapes by pattern

Each line describes a shape: `circle r`, `rect w h`, `square s` or `triangle a b c`, with decimal numbers. Print `<kind> area <value>` with two decimals. Use `match` with a **guard** so that a `rect` whose sides are equal is reported as a `square`, and a `triangle` whose sides cannot form a triangle prints `invalid triangle` (use Heron's formula for a valid one). Anything else prints `unsupported`.

Input: lines until the end of input. **Output:** one line per input line.

```
circle 1
rect 3 4
rect 2 2
triangle 3 4 5
hexagon 2
```

prints

```
circle area 3.14
rect area 12.00
square area 4.00
triangle area 6.00
unsupported
```

INPUT 1

```
circle 1
rect 3 4
rect 2 2
triangle 3 4 5
hexagon 2
```

OUTPUT 1

```
circle area 3.14
rect area 12.00
square area 4.00
triangle area 6.00
unsupported
```

REFERENCE SOLUTION

```python
import math
import sys


def is_triangle(a, b, c):
    a, b, c = sorted((a, b, c))
    return a > 0 and a + b > c


for line in sys.stdin:
    parts = line.split()
    nums = [float(p) for p in parts[1:]]
    match [parts[0], *nums] if parts else []:
        case ["circle", r]:
            print(f"circle area {math.pi * r * r:.2f}")
        case ["rect", w, h] if w == h:
            print(f"square area {w * w:.2f}")
        case ["rect", w, h]:
            print(f"rect area {w * h:.2f}")
        case ["square", s]:
            print(f"square area {s * s:.2f}")
        case ["triangle", a, b, c] if is_triangle(a, b, c):
            s = (a + b + c) / 2
            print(f"triangle area {math.sqrt(s * (s - a) * (s - b) * (s - c)):.2f}")
        case ["triangle", _, _, _]:
            print("invalid triangle")
        case _:
            print("unsupported")
```

QUIZ

1. What does this print?

```
RED = 1
code = 2
match code:
    case RED:
        print("red")
    case _:
        print("other")
```

A. `other`

B. `red`

C. `SyntaxError`

D. `NameError`

2. Which subject does the pattern `[first, *rest]` match?

A. Any list or tuple with at least one element

B. Any list with exactly two elements

C. Any string with at least one character

D. Only lists, never tuples

3. What is true of `case {"type": t}:` against the dict `{"type": "a", "size": 3}`?

A. It fails because `size` is not in the pattern

B. It matches and binds `t` to `"a"`; extra keys are ignored

C. It matches only if the dict has exactly one key

D. It raises `KeyError`

4. In `case (x, y) if x == y:`, what happens when the shape matches but `x ≠ y`?

A. The body runs with the bindings

B. The bindings are discarded and the next case is tried

C. A `ValueError` is raised

D. The `match` statement ends with no case run

5. Does `match` fall through to the next `case` after a matching body?

A. Yes, unless `break` is used

B. No — exactly one case runs and the statement ends

C. Only for literal patterns

D. Only when the body is empty

6. Which pattern matches an instance of `Point` whose `x` attribute is 0 and binds its `y`?

A. `case Point(x=0, y=y):`

B. `case Point[0, y]:`

C. `case Point(0, y) if Point.__match_args__ is None:`

D. `case {"x": 0, "y": y}:`

ANSWER KEY · MATCH STATEMENTS — STRUCTURAL PATTERN MATCHING

1. C A bare name in a pattern is a capture that matches anything, so `case RED:` would make the wildcard unreachable — and Python reports that as a `SyntaxError`. To compare with the constant, write a dotted name such as `Colour.RED`.

2. A A sequence pattern with a starred name matches sequences of at least the number of fixed elements; tuples and lists both count, strings are deliberately excluded from sequence patterns.

3. B Mapping patterns require the named keys and ignore the rest; `**rest` captures the extras if they are needed.

4. B A guard is evaluated after the pattern binds; a false guard means this case did not match, and matching continues downward.

5. B Unlike a C `switch`, there is no fall-through and `break` has no role in `match` (a `break` inside a case refers to an enclosing loop).

6. A A class pattern with keyword sub-patterns checks `isinstance` and compares attributes. Positional `Point(0, y)` also works when `__match_args__` is defined (dataclasses do it). The mapping pattern is for dicts.

LESSON 5 · 14 MIN

The loop patterns

Most loops are one of about eight patterns, and once you can name the pattern you can write the loop without thinking about it — and, more importantly, recognise when Python has already written it for you as a built-in. This lesson catalogues them: accumulate, count, extreme, search, running state, two pointers, prefix sums, and building output. Each comes with the built-in that replaces it when the body is simple, and the hand-written form for when it is not.

Accumulate, count, extreme

```
total = 0                       # accumulate
for x in xs:
    total += x
total = sum(xs)                 # the built-in

count = 0                       # count with a condition
for x in xs:
    if x % 2 == 0:
        count += 1
count = sum(1 for x in xs if x % 2 == 0)   # or sum(x % 2 == 0 for x in xs)

best = None                     # extreme with a key
for word in words:
    if best is None or len(word) > len(best):
        best = word
best = max(words, key=len)      # the built-in; raises ValueError on an empty i
```

`sum`, `min`, `max`, `len`, `any`, `all` are the accumulations; write the loop only when the update is not a plain fold — when it depends on the previous element, or when two things are tracked at once. The `best = None` start is the general form when no sensible initial value exists; `float("inf")` and `float("-inf")` are the starts for numeric minimum and maximum.

Search

```
def find(xs, target):
    for i, x in enumerate(xs):
        if x == target:
            return i
    return -1
```

Return from inside the loop the moment the answer is known; the code after the loop is the "not found" branch. In a loop that is not in its own function, `for … else` plays the same role (Module 3 lesson 2). The built-ins: `x in xs` for existence, `xs.index(x)` for the position (raising `ValueError` when absent), `next((x for x in xs if pred(x)), None)` for "first element satisfying", `any(pred(x) for x in xs)` for "does one exist".

Running state

A streak, a previous value, a state machine — the loop carries information from one iteration to the next:

```
best_len = cur_len = 0
prev = None
for x in xs:
    cur_len = cur_len + 1 if x == prev else 1
    best_len = max(best_len, cur_len)
    prev = x
```

The shape is always: initialise the state before the loop, update it from the old state and the current element, read the answer after. The classic mistake is forgetting the update at the end of the body (`prev = x`), or updating it before it has been used. `itertools.pairwise(xs)` (3.10) gives `(prev, x)` pairs directly for the two-element case, and `itertools.accumulate` for running totals and running maxima.

Two pointers

Two indices moving through one sorted sequence (or two sequences), each step advancing one of them by a rule:

```python
def pair_sum(xs, target):          # xs sorted ascending
    i, j = 0, len(xs) - 1
    while i < j:
        s = xs[i] + xs[j]
        if s == target:
            return i, j
        if s < target:
            i += 1
        else:
            j -= 1
    return None
```

This is a `while`, because which pointer moves depends on the data. The same shape merges two sorted lists, removes duplicates from a sorted list in place, and tests palindromes from both ends. The invariant worth writing as a comment is what has been ruled out: "no pair with an index below `i` or above `j` can sum to `target`".

Prefix sums and running totals

```python
prefix = [0]
for x in xs:
    prefix.append(prefix[-1] + x)
# sum(xs[a:b]) == prefix[b] - prefix[a], in O(1) per query
```

`prefix[i]` is the sum of the first `i` elements; the extra leading zero makes the subtraction uniform. `itertools.accumulate(xs, initial=0)` builds the same list. The pattern turns a sequence of range-sum questions from O(n) each into O(1) each after O(n) preparation, and is the first "precompute then answer" idea in most interview loops.

Building the output

Printing inside a loop is fine for a few lines. For many, collect and print once — it is faster (one write instead of thousands) and it keeps formatting in one place:

```
out = []
for r in range(rows):
    out.append(" ".join(str(v) for v in grid[r]))
print("\n".join(out))
```

Strings should be joined, not concatenated in a loop: `s += piece` copies the growing string each time (Module 19), while `"".join(pieces)` is one allocation. When each element maps to one line of output, the whole loop is `print("\n".join(f(x) for x in xs))`.

Nested loops and early exit

```
def has_duplicate_pair(grid):
    for row in grid:
        for a, b in itertools.pairwise(row):
            if a == b:
                return True
    return False
```

A `return` leaves every enclosing loop at once; that is the reason to put a nested search in its own function. When the loops are not a search but a full traversal, the nesting is fine as written, and `itertools.product` flattens two ranges into one loop when a single `break` is wanted.

Pitfalls

Writing the loop for `sum`, `max` or `any` by hand and getting the initial value wrong (`best = 0` for a list of negatives).

Forgetting `prev = x` at the end of a running-state body.

A two-pointer loop where neither pointer moves in some branch — an infinite loop.

An off-by-one in a prefix array without the leading zero.

`s += line` in a loop over thousands of lines.

Searching with a flag variable when `return` or `for … else` says it directly.

Key takeaways

Accumulate, count and extreme are `sum`, `sum(cond)`, `min` / `max` with `key`; write the loop only when the update is not a plain fold.

Search returns from inside the loop; after the loop is "not found"; `in`, `index`, `next(gen, None)` and `any` are the built-ins.

Running state: initialise, update from the previous state and the element, remember to advance the state.

Two pointers are a `while` with an invariant; prefix sums turn range queries into subtraction.

Collect output in a list and `join` once; never build a long string with `+=`.

EXERCISE 1

Longest streak

Find the longest run of equal consecutive values in a line of integers — the running-state pattern: carry the current value and its run length from one element to the next. On a tie, the earliest streak wins.

Input: one line of integers (possibly empty). **Output:** `value <v> length <n>`, or `empty` for no values.

```
1 1 2 2 2 3
```

prints

```
value 2 length 3
```

INPUT 1

```
1 1 2 2 2 3
```

OUTPUT 1

```
value 2 length 3
```

REFERENCE SOLUTION

```
xs = list(map(int, input().split()))
if not xs:
    print("empty")
else:
    best_val = cur_val = xs[0]
    best_len = cur_len = 1
    for x in xs[1:]:
        if x == cur_val:
            cur_len += 1
        else:
            cur_val, cur_len = x, 1
        if cur_len > best_len:
            best_val, best_len = cur_val, cur_len
    print(f"value {best_val} length {best_len}")
```

EXERCISE 2

Two-pointer pair sum

Given a target and a list of integers **sorted in ascending order**, find two indices `i < j` with `xs[i] + xs[j] == target` using the two-pointer sweep: one index from each end, moving the left one up when the sum is too small and the right one down when it is too large. Print the first pair the sweep finds.

Input: the target, then a line of sorted integers. **Output:** `<i> <j>` or `none`.

```
9
1 2 4 5 7 8
```

prints

```
0 5
```

INPUT 1

```
9
1 2 4 5 7 8
```

OUTPUT 1

```
0 5
```

INPUT 2

```
10
1 2 3
```

OUTPUT 2

```
none
```

REFERENCE SOLUTION

```
target = int(input())
xs = list(map(int, input().split()))
i, j = 0, len(xs) - 1
while i < j:
    s = xs[i] + xs[j]
    if s == target:
        print(i, j)
        break
    if s < target:
        i += 1
    else:
        j -= 1
else:
    print("none")
```

QUIZ

1. Which single expression counts the even numbers in `xs`?

A. `sum(x % 2 == 0 for x in xs)`

B. `len(x for x in xs if x % 2 == 0)`

C. `count(xs, even)`

D. `xs.count(x % 2 == 0)`

2. What is the bug?

```
best = 0
for x in xs:
    if x > best:
        best = x
```

A. `best` should start at `xs[0]`, `None` or `-inf`: for an all-negative list the answer is wrong

B. `>` should be `≥`

C. It should use `while`

D. There is no bug

3. What does `next((x for x in xs if x < 0), None)` return?

A. The first negative element, or `None` if there is none

B. A list of the negatives

C. `True` if any element is negative

D. `StopIteration`

4. With `prefix = [0, 3, 8, 10]` built from `xs = [3, 5, 2]`, what is `sum(xs[1:3])`?

A. `prefix[3] - prefix[1]` = 7

B. `prefix[2] - prefix[1]` = 5

C. `prefix[3]` = 10

D. `prefix[1] + prefix[3]` = 13

5. Why is `"\n".join(lines)` preferred to `s += line + "\n"` in a loop?

A. Because `+=` does not work on strings

B. Because each `+=` copies the whole growing string, while `join` allocates once

C. Because `join` sorts the lines

D. There is no difference

ANSWER KEY · THE LOOP PATTERNS

1. A Booleans sum as 0 and 1. `len` does not accept a generator, `count` is not a built-in, and `xs.count` looks for a specific value.

2. A Starting at 0 silently assumes a non-negative maximum exists. `max(xs)` raises on an empty list and is otherwise correct; a hand loop should start from the first element or negative infinity.

3. A `next` on a generator with a default returns the first produced value or the default when the generator is exhausted. `any` would give the boolean; a list comprehension the list.

4. A With the leading zero, the sum of `xs[a:b]` is `prefix[b] - prefix[a]`: 10 − 3 = 7 = 5 + 2.

5. B Strings are immutable, so repeated concatenation is quadratic in the total length; `join` computes the final size and copies each piece once.

CHECKPOINT · 22 MIN

Checkpoint — Control flow

This checkpoint covers the whole module: `if` / `elif` / `else` and the guard-clause shape, `while` with `break`, `continue` and the loop `else`, `for` over any iterable with `range`, `enumerate` and `zip`, structural pattern matching with `match`, and the loop patterns — accumulate, search, running state, two pointers, prefix sums and building output.

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 is a guard clause, and what does it do to the indentation of the rest of the function?
- When does the `else` of a `while` or `for` loop run, and when does it not?
- What does `range(2, 10, 3)` produce, and why is `range(len(xs))` exactly the valid indices?
- What does `zip` do when its inputs have different lengths, and how do you make that an error?
- In a `case` pattern, what is the difference between `RED` and `Colour.RED`?
- Why must a list not be modified while a `for` loop iterates over it?
- Which built-ins replace the accumulate, count and extreme loops?

The three programs are a FizzBuzz with configurable divisors, a prime lister built on a trial-division loop with `for … else`, and a bank account whose commands are dispatched with `match`.

EXERCISE 1

FizzBuzz, configured

Print the numbers from 1 to `n`, except that multiples of `a` print `Fizz`, multiples of `b` print `Buzz`, and multiples of both print `FizzBuzz`.

Input: one line `n a b` with `n ⩾ 1` and `a, b ⩾ 1`. **Output:** `n` lines.

```
6 2 3
```

prints

```
1
Fizz
Buzz
Fizz
5
FizzBuzz
```

INPUT 1

```
6 2 3
```

OUTPUT 1

```
1
Fizz
Buzz
Fizz
5
FizzBuzz
```

REFERENCE SOLUTION

```
n, a, b = map(int, input().split())
for i in range(1, n + 1):
    word = ""
    if i % a == 0:
        word += "Fizz"
    if i % b == 0:
        word += "Buzz"
    print(word or i)
```

EXERCISE 2

Primes up to n

List every prime up to and including `n`. Test each candidate by trial division in a `for` loop over the possible divisors from 2 up to its square root, using the loop's `else` clause to record that no divisor was found.

Input: one integer `n ⩾ 0`. **Output:** `primes: <space-separated primes>` (or `primes: none`), then `count: <k>`.

```
10
```

prints

```
primes: 2 3 5 7
count: 4
```

INPUT 1

```
10
```

OUTPUT 1

```
primes: 2 3 5 7
count: 4
```

REFERENCE SOLUTION

```python
import math

n = int(input())
primes = []
for candidate in range(2, n + 1):
    for d in range(2, math.isqrt(candidate) + 1):
        if candidate % d == 0:
            break
    else:
        primes.append(candidate)
print("primes:", " ".join(map(str, primes)) if primes else "none")
print(f"count: {len(primes)}")
```

EXERCISE 3

Bank account

Run a bank account from commands read until the end of input, dispatching each with `match` on `line.split()`. `deposit n` adds `n` and prints `ok <balance>`; `withdraw n` subtracts `n` and prints `ok <balance>`, or `insufficient funds` if the balance would go negative; `balance` prints `balance <balance>`. A deposit or withdrawal whose amount is not positive prints `invalid amount` (use a guard). Anything else prints `unknown: <line>`. Amounts are integers.

Input: commands, one per line. **Output:** one line per command.

```
deposit 100
withdraw 30
balance
withdraw 100
deposit -5
foo
```

prints

```
ok 100
ok 70
balance 70
insufficient funds
invalid amount
unknown: foo
```

INPUT 1

```
deposit 100
withdraw 30
balance
withdraw 100
deposit -5
foo
```

OUTPUT 1

```
ok 100
ok 70
balance 70
insufficient funds
invalid amount
unknown: foo
```

REFERENCE SOLUTION

```python
import sys

balance = 0
for line in sys.stdin:
    line = line.rstrip("\n")
    parts = line.split()
    if len(parts) == 2 and parts[0] in ("deposit", "withdraw"):
        parts[1] = int(parts[1])
    match parts:
        case ["deposit" | "withdraw", amount] if amount <= 0:
            print("invalid amount")
        case ["deposit", amount]:
            balance += amount
            print(f"ok {balance}")
        case ["withdraw", amount] if amount > balance:
            print("insufficient funds")
        case ["withdraw", amount]:
            balance -= amount
            print(f"ok {balance}")
        case ["balance"]:
            print(f"balance {balance}")
        case _:
            print(f"unknown: {line}")
```

CHECKPOINT QUESTIONS

1. What does this print?

```
for i in range(5):
    if i == 3:
        break
else:
    print("done")
print(i)
```

A. `done` then `3`

B. `3`

C. `done` then `4`

D. `4`

2. What is `list(range(10, 0, -3))`?

A. `[10, 7, 4, 1]`

B. `[10, 7, 4, 1, 0]`

C. `[]`

D. `[9, 6, 3, 0]`

3. Which condition tests whether `x` lies in the closed range 1–10?

A. `1 ⩽ x ⩽ 10`

B. `1 ⩽ x and ⩽ 10`

C. `x in [1, 10]`

D. `x ⩾ 1 or x ⩽ 10`

4. What does this print?

```
n = 10
while n:
    n //= 2
    print(n, end=" ")
```

A. `5 2 1 0`

B. `5 2 1`

C. `10 5 2 1 0`

D. It loops forever

5. Which loop visits every element of `xs` together with its index?

A. `for i, x in enumerate(xs):`

B. `for x, i in xs:`

C. `for i in xs.index():`

D. `for x in xs[i]:`

6. In a `match`, which pattern compares the subject with the constant `Status.OK` rather than capturing it?

A. `case Status.OK:`

B. `case OK:`

C. `case "Status.OK":`

D. `case status_ok:`

7. What does `zip(names, scores, strict=True)` do when `names` has 3 elements and `scores` has 2?

A. Raises `ValueError` when the mismatch is reached

B. Pads with `None`

C. Truncates silently

D. Raises `IndexError` immediately

8. Which is the correct way to remove the even numbers from `xs`?

A. `xs = [x for x in xs if x % 2]`

B. `for x in xs: if x % 2 == 0: xs.remove(x)`

C. `for i in range(len(xs)): del xs[i]`

D. `xs.remove(x % 2 == 0)`

9. What does `" ".join(str(x) for x in [1, 2, 3])` produce?

A. `'1 2 3'`

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

C. `TypeError`

D. `'123'`

10. In the two-pointer pair-sum loop over a sorted list, why does moving the left pointer up when the sum is too small never miss an answer?

A. Because the right element is the largest remaining, so no pair using the current left element can reach the target

B. Because sums are always increasing

C. Because the list has no duplicates

D. It can miss answers; the algorithm is a heuristic

11. What does `case ["go", *rest] if rest:` require?

A. A list starting with `"go"` followed by at least one more element

B. Exactly `["go"]`

C. Any list containing `"go"`

D. A string starting with `go`

12. What does `sum(1 for _ in range(4))` evaluate to?

A. `4`

B. `6`

C. `10`

D. `TypeError`

ANSWER KEY · CHECKPOINT — CONTROL FLOW

1. B The `break` skips the loop's `else`, and the loop variable keeps the value it had when the loop was left.

2. A Counting down by 3 from 10 while staying above the excluded stop 0 gives 10, 7, 4, 1.

3. A Comparison chaining reads as mathematics. The second is a syntax error, the third tests membership in a two-element list, and the fourth is true for every number.

4. A The loop runs while `n` is truthy (non-zero); each pass halves and prints, including the final `0`, after which the condition is false.

5. A `enumerate` is the idiom; the others are errors or nonsense (`xs.index()` needs an argument; `xs[i]` needs `i` defined).

6. A A dotted name is a value pattern. Bare identifiers capture; the quoted form compares with a string, which is a different value.

7. A `strict=True` (3.10) raises `ValueError` once one iterable is exhausted before the others. The default truncates; `zip_longest` pads.

8. A Build a new list; removing while iterating skips elements, deleting by index shifts the rest and eventually raises `IndexError`, and the last removes the value `False`.

9. A `join` takes an iterable of strings; the generator converts each int. Joining ints directly would be the `TypeError`.

10. A With `xs[j]` the largest candidate partner and the sum still too small, every pair with `xs[i]` is too small; discarding `i` loses nothing. The symmetric argument justifies moving `j` down.

11. A The starred name captures the remainder as a list, and the guard requires it to be non-empty. Strings never match sequence patterns.

12. A The generator yields `1` four times; summing counts the iterations. `_` marks the unused loop variable.