Module 13 of 20

The standard library in depth

math, statistics, Fraction, Decimal and seeded random; datetime arithmetic, parsing and time zones with zoneinfo; deque, Counter, OrderedDict, ChainMap and namedtuple in depth; textwrap, difflib, Template and the deeper re; and enums with auto, IntEnum, StrEnum and Flag.

The map of Module 12 said where things are; this module goes deep on the five areas of the standard library that every program touches and every interview probes: numbers beyond the basics, dates and times done correctly, the `collections` types in full, the second tier of text tools, and enumerations. Each lesson is organised around the guarantees a tool gives — exactness, calendar correctness, O(1) at both ends, reproducibility — because choosing the right tool is choosing the guarantee the problem needs.

Math and numbers covers `math`'s integer and float functions, `statistics` with sample versus population deviation, `Fraction` and `Decimal` with `quantize`, and `random.Random(seed)` versus `secrets`. Dates and times covers the four types, calendar arithmetic, ISO and `strftime` / `strptime`, naive versus aware, `zoneinfo`, timestamps and the store-UTC rule. Collections in depth covers `deque.rotate` / `maxlen`, `Counter` arithmetic, `OrderedDict.move_to_end` as an LRU cache, `ChainMap`, `namedtuple`'s helpers and the `User*` classes. Text-processing tools covers `textwrap`, `difflib`, `string.Template`, `unicodedata` and regex named groups, lookarounds, `sub` with a function and `VERBOSE`. Enums covers `Enum`, `auto`, lookup, methods, `IntEnum` / `StrEnum`, `Flag` and enums in `match`.

The exercises are whole programs: an integer and statistics report, seeded dice that are reproducible, calendar arithmetic across leap years, time-zone conversion with daylight saving, multiset operations on `Counter`, an LRU cache on `OrderedDict`, wrapping and fuzzy matching, regex extraction with named groups and lookarounds, a status enum with transitions, and typed permission flags. The checkpoint adds a statistics report with exact totals, a schedule converted between zones, and an LRU cache driven by an enum of operations.

CONTENTS

1	math, statistics, fractions, decimal and random	14 min
2	Dates and times — datetime, timedelta and zoneinfo	14 min

3	collections in depth — deque, Counter, OrderedDict, ChainMap and the User classes	13 min
4	Text-processing tools — textwrap, difflib, string.Template and re in depth	14 min
5	Enums — Enum, IntEnum, StrEnum, Flag and auto	13 min
✓	Checkpoint — The standard library in depth	22 min

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

LESSON 1 · 14 MIN

math, statistics, fractions, decimal and random

The numeric modules divide the work cleanly: `math` for functions on ints and floats, `statistics` for descriptive statistics done correctly, `fractions` and `decimal` for the two kinds of exact arithmetic, `random` for reproducible pseudo-randomness and `secrets` for the unpredictable kind. This lesson goes through each with the functions that matter, the precision rules that decide which to use, and the two facts about `random` that judged and tested programs depend on: seeding makes it deterministic, and the module-level functions share one global generator.

math

```
import math

math.sqrt(2), math.isqrt(17)          # 1.4142135623730951, 4  — float root, e
math.gcd(12, 18), math.lcm(4, 6)      # 6, 12  (gcd takes any number of argume
math.comb(5, 2), math.perm(5, 2)      # 10, 20
math.factorial(20)                    # 2432902008176640000 — exact int
math.floor(-2.5), math.ceil(-2.5)     # -3, -2  — ints
math.trunc(-2.5)                      # -2 — toward zero, like int()
math.log(100, 10), math.log2(1024), math.log10(1000), math.exp(1)
math.prod([2, 3, 4])                  # 24 — the multiplicative sum
math.fsum([0.1] * 10)                 # 1.0 — accurate float summation
math.hypot(3, 4), math.dist((0, 0), (3, 4))   # 5.0, 5.0
math.isclose(0.1 + 0.2, 0.3)          # True
math.pi, math.e, math.tau, math.inf, math.nan
math.radians(180), math.degrees(math.pi), math.sin, math.cos, math.atan2(y, x)
```

Integer functions (`isqrt` , `gcd` , `lcm` , `comb` , `perm` , `factorial`) return exact ints of any size; the float functions return floats and raise `ValueError` on domain errors (`math.sqrt(-1)`) and `OverflowError` on results too large. `math.pow` is always float — use `**` for integer powers. `math.isqrt(n) ** 2 == n` is the perfect-square test; `math.comb` replaces the factorial formula and never overflows.

statistics

```python
import statistics as st

st.mean([1, 2, 3, 4])           # 2.5
st.fmean([1, 2, 3, 4])          # 2.5 as a float, faster
st.median([3, 1, 2])            # 2
st.median([1, 2, 3, 4])         # 2.5 — the mean of the middle two
st.mode("aabbbc")               # 'b'
st.multimode([1, 1, 2, 2])      # [1, 2]
st.stdev([2, 4, 4, 4, 5, 5, 7, 9])    # sample standard deviation (n - 1)
st.pstdev(...)                  # population standard deviation (n)
st.variance, st.pvariance
st.quantiles([1, 2, 3, 4, 5], n=4)    # quartile cut points
```

`mean` of ints may return a `Fraction`-exact result converted to float; `median` of an even count averages the two middle values; `stdev` is the sample form — the one with `n - 1` — and `pstdev` the population form; interviews ask which is which. All raise `StatisticsError` on empty input. These functions are correct on the edge cases a hand-written loop misses (mixed `int` / `Fraction` / `Decimal` input, catastrophic cancellation in variance).

---

fractions

```python
from fractions import Fraction

Fraction(1, 3) + Fraction(1, 6)       # Fraction(1, 2)
Fraction("0.1") + Fraction("0.2")     # Fraction(3, 10) — exact
Fraction(0.1)                         # Fraction(3602879701896397, 3602879701
Fraction(3, 4).numerator, Fraction(3, 4).denominator
float(Fraction(1, 3)), round(Fraction(7, 3))       # 0.333…, 2
Fraction(5, 10)                       # Fraction(1, 2) — always reduced
Fraction(1, 3).limit_denominator(100) # the closest fraction with denominator
```

Exact rational arithmetic: sums of fractions are fractions, comparisons are exact, and probability and ratio computations come out as `3/8` rather than `0.375000001`. Construct from strings or int pairs, not floats. It is slow relative to floats and is used where exactness matters more than speed.

---

decimal

```python
from decimal import Decimal, getcontext, ROUND_HALF_UP, ROUND_HALF_EVEN

getcontext().prec = 28                        # significant digits (default 28
Decimal("0.1") + Decimal("0.2")               # Decimal('0.3')
Decimal("1") / Decimal("3")                   # Decimal('0.33333333333333333
Decimal("2.675").quantize(Decimal("0.01"))    # Decimal('2.68') — ROUND_HALF_E
Decimal("2.665").quantize(Decimal("0.01"), rounding=ROUND_HALF_UP)   # Decimal
Decimal("1.10") + Decimal("2.20")             # Decimal('3.30') — trailing zer
Decimal("100").sqrt(), Decimal("2").ln()
```

Decimal floating point with a *context* (precision and rounding mode) — the arithmetic finance and invoicing require. `quantize` rounds to a pattern (`Decimal("0.01")` for cents) with a stated rounding rule; the default is banker's rounding, and `ROUND_HALF_UP` is what accounting usually wants. `localcontext()` sets precision for a block. Construct from strings; `Decimal(0.1)` carries the float's error.

---

random

```python
import random

rng = random.Random(42)              # a private generator with a fixed seed -
rng.random()                         # a float in [0, 1)
rng.randint(1, 6)                    # inclusive both ends
rng.randrange(0, 10, 2)              # like range
rng.choice(["a", "b", "c"])
rng.choices(items, weights=[5, 1], k=3)   # with replacement, weighted
rng.sample(items, k=3)               # without replacement
rng.shuffle(xs)                      # in place
rng.uniform(1.5, 2.5), rng.gauss(0, 1)
random.seed(42); random.randint(1, 6)     # the module-level functions share (
```

Two rules. **Seed it and it is deterministic**: the same seed gives the same sequence on every run and every platform for a given algorithm version, which is what tests and judged exercises need. **Prefer a `Random(seed)` instance** to the module functions: it cannot be disturbed by another part of the program calling `random.random()` . `random` is a Mersenne Twister — statistically fine, cryptographically useless: for tokens, passwords and anything an attacker may guess, `secrets.token_hex(16)` , `secrets.choice` , `secrets.randbelow` .

Choosing

Need	Use
Integer arithmetic, combinatorics, exact roots	`int` and `math`'s integer functions
Scientific/geometric computation	`float` and `math`
Money, invoices, rounding rules	`Decimal` from strings with `quantize`, or integer cents
Ratios and probabilities that must be exact	`Fraction`
Descriptive statistics	`statistics`
Reproducible randomness	`random.Random(seed)`
Security-relevant randomness	`secrets`

Pitfalls

`math.sqrt` for integer roots (`isqrt`); `math.pow` for integer powers (`**`).

`stdev` where `pstdev` was meant, or the reverse.

`Decimal(0.1)` and `Fraction(0.1)`.

`random` for anything security-related.

Module-level `random.seed` in a library, resetting the caller's generator.

Expecting `round(2.675, 2)` to give `2.68` — use `Decimal` and `quantize`.

Key takeaways

`math`: exact integer functions (`isqrt`, `gcd`, `lcm`, `comb`, `factorial`), float functions that raise on domain errors, `fsum`, `isclose`, `prod`.

`statistics`: `mean` / `fmean`, `median`, `mode`, sample `stdev` versus population `pstdev`, `quantiles`; empty input raises.

`Fraction` for exact ratios, `Decimal` with `quantize` and a rounding mode for money — both built from strings.

`random.Random(seed)` is reproducible and isolated; the module functions share one global state; `secrets` for anything guessable.

Pick the numeric type by the guarantee you need, not by habit.

EXERCISE 1

Number report

Read a line of positive integers and print: the gcd and lcm of all of them (`math.gcd` / `math.lcm` accept many arguments), the integer square root of the largest, which values are perfect squares (`isqrt(n) ** 2 == n`), `math.comb(n, 2)` for `n` = the count of values, and the sample and population standard deviations from `statistics` with three decimals (`n/a` for the sample deviation when there is only one value).

Input: one line of positive integers (at least one). **Output:** `gcd <g> lcm <l>`, `isqrt <r>`, `squares <values or none>`, `pairs <c>`, `stdev <s> pstdev <p>`.

```
12 18 36
```

prints

```
gcd 6 lcm 36
isqrt 6
squares 36
pairs 3
stdev 12.490 pstdev 10.198
```

INPUT 1

```
12 18 36
```

OUTPUT 1

```
gcd 6 lcm 36
isqrt 6
squares 36
pairs 3
stdev 12.490 pstdev 10.198
```

REFERENCE SOLUTION

```
import math
import statistics

xs = list(map(int, input().split()))
print(f"gcd {math.gcd(*xs)} lcm {math.lcm(*xs)}")
print(f"isqrt {math.isqrt(max(xs))}")
squares = [x for x in xs if math.isqrt(x) ** 2 == x]
print("squares", " ".join(map(str, squares)) if squares else "none")
print(f"pairs {math.comb(len(xs), 2)}")
sample = f"{statistics.stdev(xs):.3f}" if len(xs) > 1 else "n/a"
print(f"stdev {sample} pstdev {statistics.pstdev(xs):.3f}")
```

EXERCISE 2

Seeded dice

Randomness must be reproducible in a judged program. Read a seed and `n`; create `random.Random(seed)`; print `n` rolls of `randint(1, 6)`, then a shuffled copy of `[1, 2, 3, 4, 5]` (`shuffle` in place on a copy), then `sample(range(100), 3)`, then a `choice` from `"abc"`. Every value comes from the **same** generator in that order.

Input: `seed n`. **Output:** `rolls …`, `shuffled …`, `sample …`, `choice <c>`.

```
42 5
```

prints

```
rolls 6 1 1 6 3
shuffled 4 3 1 5 2
sample 86 94 69
choice a
```

INPUT 1

```
42 5
```

OUTPUT 1

```
rolls 6 1 1 6 3
shuffled 4 3 1 5 2
sample 86 94 69
choice a
```

REFERENCE SOLUTION

```python
import random

seed, n = map(int, input().split())
rng = random.Random(seed)
print("rolls", " ".join(str(rng.randint(1, 6)) for _ in range(n)))
deck = [1, 2, 3, 4, 5]
rng.shuffle(deck)
print("shuffled", " ".join(map(str, deck)))
print("sample", " ".join(map(str, rng.sample(range(100), 3))))
print("choice", rng.choice("abc"))
```

QUIZ

1. Which is the exact integer square root?

A. `int(math.sqrt(n))`

B. `math.isqrt(n)`

C. `round(n ** 0.5)`

D. `math.floor(math.sqrt(n))`

2. Which `statistics` function divides by n − 1?

A. `pstdev`

B. `stdev`

C. `variance` and `pvariance` both

D. `mean`

3. What does `Decimal("2.675").quantize(Decimal("0.01"))` give by default?

A. `2.68`

B. `2.67`

C. `2.7`

D. `2.675`

4. Why prefer `random.Random(seed)` to `random.seed(seed)`?

A. It is faster

B. The instance has its own state, isolated from any other code calling the module functions

C. The module functions cannot be seeded

D. There is no difference

5. Which module generates a password-reset token?

A. `random`

B. `secrets`

C. `hashlib`

D. `uuid` only

ANSWER KEY · MATH, STATISTICS, FRACTIONS, DECIMAL AND RANDOM

1. B `isqrt` works on integers of any size; the float forms lose precision above 2^{53}.

2. B `stdev` / `variance` are the sample forms; `pstdev` / `pvariance` the population forms.

3. A The default context rounds half to even, and 2.675 exactly is a tie whose even neighbour is 2.68. (The float `2.675` is below the tie, which is why `round(2.675, 2)` is `2.67`.)

4. B The module functions share one global generator that any library call can advance.

5. B `random` is predictable by design; `secrets` uses the OS's cryptographic source.

LESSON 2 · 14 MIN

Dates and times — datetime, timedelta and zoneinfo

Dates are where programs go wrong quietly: a month with 30 days, a year that is not a leap year, a clock that jumps an hour, a timestamp that is in one zone on the server and another in the browser. The `datetime` module handles the calendar arithmetic correctly if you use its types rather than strings and integers, and `zoneinfo` (3.9) gives it real time zones from the IANA database. This lesson covers `date`, `time`, `datetime` and `timedelta`, parsing and formatting with `strptime` / `strftime` and the ISO methods, arithmetic and comparison, naive versus aware datetimes, converting between zones, and the rule that keeps a system sane: store UTC, display local.

The types

```
from datetime import date, time, datetime, timedelta

d = date(2024, 5, 1)                 # a calendar date
t = time(9, 30)                      # a wall-clock time, no date
dt = datetime(2024, 5, 1, 9, 30)     # both
delta = timedelta(days=3, hours=4)   # a duration

d.year, d.month, d.day, d.weekday()  # 2024 5 1 2   (Monday is 0; isoweekday()
dt.date(), dt.time()                 # split
datetime.combine(d, t)               # join
d.replace(year=2025)                 # a new date with one field changed — dat
date.today(), datetime.now()         # the clock — never in a judged program (
```

`timedelta` stores days, seconds and microseconds; it has no months or years because those are not fixed lengths. `delta.total_seconds()` converts; `delta.days` is the whole-day part.

Arithmetic and comparison

```
d + timedelta(days=30)                # 2024-05-31 — calendar-correct across mo
date(2024, 3, 1) - date(2024, 2, 1)   # timedelta(days=29): 2024 is a leap year
dt2 - dt1                             # a timedelta; (dt2 - dt1).total_seconds(
d1 < d2, d1 == d2                     # compare directly
timedelta(hours=36) // timedelta(hours=1)    # 36 — how many hours
sorted(dates)                         # chronological
min(dates), max(dates)
```

Adding a month is not a `timedelta` operation. The honest options: `replace(month=…)` with the day clipped by hand, or the third-party `dateutil.relativedelta`. Next-day and end-of-month logic is `timedelta(days=1)` and `replace(day=1) + timedelta(days=32)` then `replace(day=1) - timedelta(days=1)` — or `calendar.monthrange(year, month)[1]` for the length of a month.

Formatting and parsing

```
dt.isoformat()                                # '2024-05-01T09:30:00'
d.isoformat()                                 # '2024-05-01'
datetime.fromisoformat("2024-05-01T09:30:00")  # parses ISO 8601 (3.11 accepts
date.fromisoformat("2024-05-01")

dt.strftime("%d/%m/%Y %H:%M")                 # '01/05/2024 09:30'
datetime.strptime("01/05/2024 09:30", "%d/%m/%Y %H:%M")
dt.strftime("%A %d %B %Y")                    # 'Wednesday 01 May 2024' (Engl
f"{dt:%Y-%m-%d}"                              # format specs work in f-string
```

The ISO methods are the right default: unambiguous, sortable as strings, zone-capable. `strftime` / `strptime` use the C directives — `%Y` year, `%m` month, `%d` day, `%H` hour (24), `%M` minute, `%S` second, `%f` microseconds, `%z` offset, `%Z` zone name, `%a` / `%A` weekday, `%b` / `%B` month name, `%j` day of year, `%U` / `%W` week — and `strptime` raises `ValueError` on a mismatch, which is the validation. Weekday and month *names* depend on the locale; the judge and most servers run in the C locale, where they are English.

Naive and aware

A **naive** datetime has no zone; it is a wall-clock reading with no claim about where. An **aware** one carries a `tzinfo` and represents an actual instant. The two do not mix: comparing or

subtracting a naive and an aware datetime raises `TypeError`.

```
from zoneinfo import ZoneInfo
from datetime import timezone

utc = datetime(2024, 5, 1, 4, 0, tzinfo=timezone.utc)
kolkata = utc.astimezone(ZoneInfo("Asia/Kolkata"))    # 2024-05-01 09:30+05:3
kolkata.isoformat()                                   # '2024-05-01T09:30:00+
local = datetime(2024, 5, 1, 9, 30, tzinfo=ZoneInfo("Asia/Kolkata"))
local.astimezone(timezone.utc)                        # back to 04:00+00:00
local.utcoffset()                                     # timedelta(seconds=198
naive.replace(tzinfo=ZoneInfo("Europe/London"))       # *attach* a zone to a
```

`astimezone` converts an instant between zones; `replace(tzinfo=…)` asserts which zone a naive reading was in — different operations, easily confused. `ZoneInfo` handles daylight-saving rules from the IANA database (`Europe/London` is +01:00 in summer and +00:00 in winter); a fixed `timezone(timedelta(hours=5, minutes=30))` does not, and is only right for zones without DST. The `utcnow()` and `utcfromtimestamp()` functions are deprecated because they return naive values; use `datetime.now(timezone.utc)`.

Timestamps

```
dt.timestamp()                                  # seconds since the epoch, fl
datetime.fromtimestamp(1714536000, tz=timezone.utc)
```

A Unix timestamp is an instant; converting it to a datetime without a `tz` uses the machine's local zone, which is the classic source of off-by-hours bugs. Always pass `tz=`.

The rule

Store and compute in UTC (aware, `timezone.utc`); convert to a user's zone with `astimezone` only to display; parse user input in the user's zone and convert to UTC immediately. Keep durations as `timedelta`, never as floats of hours. And keep the clock out of the logic: a function that takes `now` as a parameter can be tested with a fixed value (Module 18), while one that calls `datetime.now()` inside cannot.

Pitfalls

Adding "a month" as 30 days.

Comparing naive with aware (`TypeError`), or a naive `utcnow()` with an aware value.

`replace(tzinfo=…)` where `astimezone` was meant (it relabels; it does not convert).

`fromtimestamp` without `tz`.

Fixed offsets for zones that observe daylight saving.

Calling `now()` inside logic that should be testable or judged.

Key takeaways

`date`, `time`, `datetime`, `timedelta`: immutable, comparable, with calendar-correct arithmetic; no month/year deltas.

ISO methods for storage and exchange; `strftime` / `strptime` for human formats, with `ValueError` as validation.

Aware datetimes carry a zone and are instants; naive ones are readings; never mix them.

`ZoneInfo("Asia/Kolkata")` for real zones with DST; `astimezone` converts, `replace(tzinfo=)` labels; `timestamp()` and `fromtimestamp(ts, tz=)` for epoch seconds.

Store UTC, display local, pass `now` in as a parameter.

EXERCISE 1

Calendar arithmetic

Read pairs of ISO dates `a b`. For each print the number of days from `a` to `b` (negative if `b` is earlier), the weekday name of `a` (`%A`), the date 30 days after `a`, and whether `a`'s year is a leap year (`calendar.isleap`).

Input: lines `a b`. **Output:** `days <n> weekday <name> plus30 <date> leap <bool>` per line.

```
2024-02-28 2024-03-01
```

prints

```
days 2 weekday Wednesday plus30 2024-03-29 leap True
```

INPUT 1

```
2024-02-28 2024-03-01
```

OUTPUT 1

```
days 2 weekday Wednesday plus30 2024-03-29 leap True
```

REFERENCE SOLUTION

```
import calendar
import sys
from datetime import date, timedelta

for line in sys.stdin:
    a, b = (date.fromisoformat(t) for t in line.split())
    days = (b - a).days
    print(f"days {days} weekday {a.strftime('%A')} plus30 {(a + timedelta(days=30)).
```

EXERCISE 2

Across time zones

The first line names a zone (IANA name such as `Asia/Kolkata`). Each following line is either `utc <ISO datetime>` — an instant in UTC to convert into the zone — or `local <ISO datetime>` — a wall-clock reading in the zone to convert to UTC. Print the converted value in ISO format with its offset, using aware datetimes only (`timezone.utc` and `ZoneInfo`).

Input: the zone, then lines. **Output:** one ISO datetime per line.

```
Europe/London
utc 2024-07-01T12:00:00
local 2024-01-15T09:00:00
```

prints

```
2024-07-01T13:00:00+01:00
2024-01-15T09:00:00+00:00
```

INPUT 1

```
Europe/London
utc 2024-07-01T12:00:00
local 2024-01-15T09:00:00
```

OUTPUT 1

```
2024-07-01T13:00:00+01:00
2024-01-15T09:00:00+00:00
```

REFERENCE SOLUTION

```python
import sys
from datetime import datetime, timezone
from zoneinfo import ZoneInfo

zone = ZoneInfo(input().strip())
for line in sys.stdin:
    kind, text = line.split()
    naive = datetime.fromisoformat(text)
    if kind == "utc":
        print(naive.replace(tzinfo=timezone.utc).astimezone(zone).isoformat())
    else:
        print(naive.replace(tzinfo=zone).astimezone(timezone.utc).isoformat())
```

QUIZ

1. Why is there no `timedelta(months=1)`?

A. It was forgotten

B. A month has no fixed length, so it cannot be a duration

C. Months are handled by `time`

D. It exists

2. What happens when a naive and an aware datetime are compared?

A. The naive one is assumed UTC

B. `TypeError`

C. They compare by wall-clock

D. `False`

3. Which converts an instant from one zone to another?

A. `dt.replace(tzinfo=z)`

B. `dt.astimezone(z)`

C. `dt.tzinfo = z`

D. `dt + z`

4. What does `datetime.fromtimestamp(ts)` without `tz` use?

A. UTC

B. The machine's local zone — a classic source of off-by-hours bugs

C. The zone in `ts`

D. No zone; it raises

5. What is the recommended storage form for instants?

A. Local naive datetimes

B. Aware UTC datetimes, converted to the user's zone only for display

C. Strings in `%d/%m/%Y`

D. Floats of hours

ANSWER KEY · DATES AND TIMES — DATETIME, TIMEDELTA AND ZONEINFO

1. B `timedelta` holds days, seconds and microseconds; month arithmetic needs `replace` or `dateutil`.

2. B The two are different kinds of value; attach a zone first with `replace(tzinfo=…)`.

3. B `replace` only relabels; `astimezone` adjusts the clock reading to the new zone.

4. B Always pass `tz=timezone.utc` (or the intended zone).

5. B One reference frame for all computation removes DST and zone ambiguity from the logic.

LESSON 3 · 13 MIN

collections in depth — deque, Counter, OrderedDict, ChainMap and the User classes

`collections` has appeared in five modules already: `Counter` and `defaultdict` for counting and grouping, `deque` for queues and windows, `namedtuple` for records, `UserDict` for subclassing a mapping. This lesson gathers the rest of what each can do — `deque.rotate` and `maxlen`, `Counter` arithmetic and `most_common` semantics, `OrderedDict.move_to_end` as the basis of an LRU cache, `ChainMap` for layered configuration, `namedtuple`'s `_replace` / `_asdict` / `_make` and the `User*` classes — so that when a problem is "a bounded history", "a cache with eviction" or "settings with defaults and overrides", the answer is one class rather than a hand-written structure.

deque

```
from collections import deque

d = deque([1, 2, 3], maxlen=5)
d.append(4); d.appendleft(0)          # both ends, O(1)
d.pop(); d.popleft()
d.rotate(1)                           # [3, 1, 2] → the last element moves to t
d.rotate(-1)                          # the opposite
d.extend([7, 8]); d.extendleft([9])   # extendleft reverses the order it inser
d[0], d[-1], len(d), 2 in d           # indexing the ends is O(1); the middle i
list(d)
```

`maxlen` makes a bounded buffer that discards from the far end — the last-`n`-lines log, the moving window. `rotate` gives circular behaviour: a round-robin scheduler is `d.rotate(-1); d[-1]`, a Caesar-style shift of a sequence is a rotation. A deque has no slicing; convert when you need it.

Counter, fully

```
from collections import Counter

c = Counter("mississippi")
c.most_common()             # [('i', 4), ('s', 4), ('p', 2), ('m', 1)] — count
c.most_common(2)
c.total()                   # 11 (3.10)
c.elements()                # an iterator: m, i, i, i, i, s, s, s, s, p, p (in
c["z"]                      # 0, and the key is NOT inserted
c.update("miss"); c.subtract("ss")
+c                          # a new Counter with only positive counts
Counter(a=3, b=1) - Counter(a=1, b=5)      # Counter({'a': 2}) — negatives dro
Counter(a=3) & Counter(a=1, b=2)           # min per key: Counter({'a': 1})
Counter(a=3) | Counter(a=1, b=2)           # max per key
Counter(a=1) == Counter(a=1, b=0)          # True (3.10): missing and zero cor
```

The arithmetic makes multiset problems one-liners: "can `word` be built from the letters of `tiles`" is `not (Counter(word) - Counter(tiles))`; the letters two words share is `Counter(a) & Counter(b)`.

OrderedDict and the LRU cache

A plain dict keeps insertion order, so `OrderedDict` survives for one operation it alone has: `move_to_end(key, last=True)`, which re-orders in O(1). That is exactly what a least-recently-used cache needs:

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self._data = OrderedDict()

    def get(self, key):
        if key not in self._data:
            return None
        self._data.move_to_end(key)            # most recently used → the end
        return self._data[key]

    def put(self, key, value):
        self._data[key] = value
        self._data.move_to_end(key)
        if len(self._data) > self.capacity:
            self._data.popitem(last=False)     # evict the least recently use
```

`popitem(last=False)` pops from the front; a plain dict's `popitem` pops only the last. This class is the standard interview answer to "design an LRU cache" (Module 20 implements it with a linked list as well); `functools.lru_cache` is the same idea applied to a function.

ChainMap

```python
from collections import ChainMap

defaults = {"colour": "blue", "size": 10}
config_file = {"size": 12}
command_line = {"colour": "red"}
settings = ChainMap(command_line, config_file, defaults)
settings["colour"], settings["size"]     # 'red', 12 — the first mapping that
settings["debug"] = True                 # writes go to the FIRST mapping only
settings.maps                            # the list of underlying dicts
settings.new_child({"tmp": 1})           # a new ChainMap with one more layer
```

A `ChainMap` is a view over several dicts searched in order — layered configuration, nested scopes in an interpreter — with no copying, so a change to an underlying dict is visible immediately. `dict(settings)` flattens it.

namedtuple, fully

```
from collections import namedtuple

Point = namedtuple("Point", "x y", defaults=[0])      # defaults apply to the
p = Point(3)                                            # Point(x=3, y=0)
p._replace(y=4)                                         # a new Point
p._asdict()                                             # {'x': 3, 'y': 0}
Point._make([1, 2])                                     # from an iterable
Point._fields                                           # ('x', 'y')
Point._field_defaults                                   # {'y': 0}
```

`typing.NamedTuple` adds hints and methods (Module 6). Both are tuples first: `p == (3, 0)` is `True`, which is the reason a `dataclass` is the better choice when that equality would be a bug.

UserDict, UserList, UserString

Written in Python so that every operation goes through the basic methods (`__getitem__` , `__setitem__` , `__delitem__` , `__len__` , `__iter__`), which is what makes an overridden method take effect in `update` , `pop` , slicing and the rest — the problem with subclassing the built-ins directly (Module 9). The wrapped object is `.data` . Use them for a mapping or sequence with a rule applied to every access; use `collections.abc.MutableMapping` when the storage is not a dict at all.

Choosing, again

- Bounded history / sliding window / queue / round-robin: `deque` .
- Multiset counting and comparison: `Counter` .
- Cache with eviction by recency: `OrderedDict` (or `lru_cache` for a function).
- Layered lookups without copying: `ChainMap` .
- Small immutable record: `namedtuple` ; with behaviour or defaults that are mutable: `dataclass` .
- A mapping with a rule on every access: `UserDict` .

Pitfalls

Slicing a `deque`.

`extendleft` reversing the order of what it inserts.

Reading `c["missing"]` and expecting the key to appear (it does not; `c["missing"] += 0` would insert it).

Relying on `most_common` tie order when the spec wants alphabetical.

Writing through a `ChainMap` and expecting it to update a lower layer.

`popitem()` on an `OrderedDict` without `last=False` for the LRU eviction.

Key takeaways

`deque`: O(1) at both ends, `rotate`, `maxlen`; no slicing.

`Counter`: `most_common`, `total`, `elements`, `update` / `subtract`, `+ - & |` as multiset operations.

`OrderedDict.move_to_end` and `popitem(last=False)` make an LRU cache; a plain dict cannot re-order.

`ChainMap` searches layers in order and writes to the first; `namedtuple` has `_replace`, `_asdict`, `_make`, `defaults`.

`UserDict` / `UserList` / `UserString` are the subclassable containers.

EXERCISE 1

Multisets

Two lines hold a word bank (`tiles`) and a target word. Using `Counter` arithmetic only, print whether the word can be built from the tiles (`Counter(word) - Counter(tiles)` is empty), the letters they share as a sorted string of `(Counter(word) & Counter(tiles)).elements()`, the tiles left over after building it (or after removing what can be matched, if it cannot be built), sorted, and the most common tile with its count (ties by first appearance in the tile line).

Input: two lines. **Output:** `buildable <bool>`, `shared <letters>`, `left <letters or none>`, `top <letter> <count>`.

```
aabbbc
abc
```

prints

```
buildable True
shared abc
left abb
top b 3
```

INPUT 1

```
aabbbc
abc
```

OUTPUT 1

```
buildable True
shared abc
left abb
top b 3
```

REFERENCE SOLUTION

```python
from collections import Counter

tiles = Counter(input().strip())
word = Counter(input().strip())
missing = word - tiles
print(f"buildable {not missing}")
print("shared", "".join(sorted((word & tiles).elements())))
left = tiles - word
print("left", "".join(sorted(left.elements())) if left else "none")
letter, count = tiles.most_common(1)[0]
print(f"top {letter} {count}")
```

EXERCISE 2

LRU cache on OrderedDict

Implement `LRUCache(capacity)` on an `OrderedDict`: `get(key)` returns the value and marks the key most recently used (`move_to_end`) or returns `-1`; `put(key, value)` inserts or updates, marks it most recent, and evicts the least recently used entry (`popitem(last=False)`) when over capacity. Read the capacity, then commands `get k` (print the result) and `put k v`; at the end print the keys from least to most recently used.

Input: the capacity, then commands. **Output:** one line per `get`, then `keys <k …>` (or `keys none`).

```
2
put a 1
put b 2
get a
put c 3
get b
get c
```

prints

```
1
-1
3
keys a c
```

INPUT 1

```
2
put a 1
put b 2
get a
put c 3
get b
get c
```

OUTPUT 1

```
1
-1
3
keys a c
```

REFERENCE SOLUTION

```
import sys
from collections import OrderedDict


class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self._data = OrderedDict()

    def get(self, key):
        if key not in self._data:
            return -1
        self._data.move_to_end(key)
        return self._data[key]

    def put(self, key, value):
        self._data[key] = value
        self._data.move_to_end(key)
        if len(self._data) > self.capacity:
            self._data.popitem(last=False)

    def keys(self):
        return list(self._data)


cache = LRUCache(int(input()))
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "get":
        print(cache.get(args[0]))
    else:
        cache.put(args[0], int(args[1]))
print("keys", " ".join(cache.keys()) if cache.keys() else "none")
```

QUIZ

1. What does `deque([1, 2, 3]).rotate(1)` produce?

A. `[2, 3, 1]`

B. `[3, 1, 2]`

C. `[1, 2, 3]`

D. `[3, 2, 1]`

2. What is `Counter(a=3, b=1) - Counter(a=1, b=5)`?

A. `Counter({'a': 2, 'b': -4})`

B. `Counter({'a': 2})`

C. `Counter({'b': -4})`

D. `TypeError`

3. Which `OrderedDict` method does a plain dict lack, making LRU caches possible?

A. `pop`

B. `move_to_end`

C. `update`

D. `keys`

4. Where does a write through a `ChainMap` go?

A. Every layer

B. The first mapping only

C. The last mapping

D. A new layer

5. Why do `UserDict` overrides of `__setitem__` affect `update()` when `dict` overrides do not?

A. `UserDict` is faster

B. `UserDict` is written in Python and routes every operation through the basic item methods

C. `dict.update` is a class method

D. They behave the same

ANSWER KEY · COLLECTIONS IN DEPTH — DEQUE, COUNTER, ORDEREDDICT, CHAINMAP AND THE USER CLASSES

1. B A positive rotation moves elements from the right end to the front.

2. B Subtraction keeps only positive counts; `subtract()` would keep negatives.

3. B A plain dict preserves insertion order but cannot re-order an existing key in O(1).

4. B Reads search the layers in order; writes and deletes affect only `maps[0]`.

5. B The C implementation of `dict` calls its own internals; the Python one calls your overrides.

LESSON 4 · 14 MIN

Text-processing tools — textwrap, difflib, string.Template and re in depth

Beyond the string methods and the regular-expression basics of Module 5, the standard library has a second tier of text tools that turn common jobs into a call: wrapping and indenting paragraphs, computing what changed between two versions, fuzzy-matching a typed name against the valid ones, filling templates safely, and the parts of `re` — named groups, lookarounds, `sub` with a function, `VERBOSE` — that make a regex readable. This lesson covers `textwrap`, `difflib`, `string.Template` and `unicodedata`, then the regex features in depth.

textwrap

```
import textwrap

text = "The quick brown fox jumps over the lazy dog and keeps running."
textwrap.wrap(text, width=20)          # ['The quick brown fox', 'jumps over t
print(textwrap.fill(text, width=20))   # the same, joined with newlines
textwrap.shorten(text, width=25, placeholder="…")   # 'The quick brown fox…'
textwrap.indent("a\nb", "> ")          # '> a\n> b'
textwrap.dedent("""\
    def f():
        pass
""")                                   # removes the common leading whitespace
```

`wrap` breaks on whitespace and never splits a word unless it exceeds the width (`break_long_words=False` to forbid even that); `initial_indent` / `subsequent_indent` make hanging paragraphs. `dedent` is the fix for indented multi-line strings inside functions.

difflib

```
import difflib

a = "the cat sat on the mat".split()
b = "the cat lay on a mat".split()
difflib.SequenceMatcher(None, a, b).ratio()          # 0.8333… similarity in
for tag, i1, i2, j1, j2 in difflib.SequenceMatcher(None, a, b).get_opcodes():
    print(tag, a[i1:i2], b[j1:j2])                   # equal / replace / dele
list(difflib.unified_diff(old_lines, new_lines, lineterm=""))   # a patch-styl
difflib.get_close_matches("appel", ["apple", "ample", "apply"], n=2, cutoff=0.
```

`SequenceMatcher` works on any sequences (characters, words, lines) and its `ratio` is the standard "how similar" number; `get_close_matches` is the "did you mean?" helper; `unified_diff` and `ndiff` render changes for humans. `HtmlDiff` produces a side-by-side table.

---

string.Template

```
from string import Template

t = Template("Hello, $name! You owe $$${amount}.")
t.substitute(name="Ada", amount="12.50")         # 'Hello, Ada! You owe $12.50
t.safe_substitute(name="Ada")                     # leaves $amount in place if
```

`$identifier` placeholders and `$$` for a literal dollar — deliberately less powerful than f-strings and `str.format`, which is the point when the template comes from a user or a file: it cannot call methods or read attributes, so it cannot be abused. `string` also holds `ascii_letters`, `digits`, `punctuation`, `whitespace` and `capwords`.

---

unicodedata

```
import unicodedata

unicodedata.normalize("NFC", s)        # compose accents (compare and store in
unicodedata.normalize("NFKD", "ﬁ")     # 'fi' — compatibility decomposition
unicodedata.name("€")                  # 'EURO SIGN'
unicodedata.category("A")              # 'Lu' — letter, uppercase
"".join(c for c in unicodedata.normalize("NFKD", "café") if not unicodedata.co
```

The last line strips accents — the usual step before slugifying or comparing names loosely (Module 5).

---

re in depth

Named groups and groupdict.

```
m = re.search(r"(?P<key>\w+)\s*=\s*(?P<value>.*)", "port = 8080")
m["key"], m["value"], m.groupdict()          # 'port', '8080', {'key': 'port',
```

Lookarounds assert without consuming:

```
re.findall(r"\d+(?= kg)", "5 kg, 12 lb, 7 kg")       # ['5', '7'] — followed b
re.findall(r"(?<=\$)\d+", "cost $40 or $55")          # ['40', '55'] — precede
re.sub(r"(?<!\d)0+(?=\d)", "", "007 and 0042")        # '7 and 42' — strip lea
re.findall(r"\b(?!the\b)\w+", "the cat the dog")      # words that are not "th
```

`(?=…)` positive lookahead, `(?!…)` negative lookahead, `(?<=…)` positive lookbehind (fixed width), `(?<!…)` negative lookbehind. They let a pattern match *at a position* without including the context in the match — the tool for "a number followed by a unit, but only the number".

sub with a function transforms each match:

```
re.sub(r"\d+", lambda m: str(int(m.group()) * 2), "a1 b22")      # 'a2 b44'
re.sub(r"\b(\w)(\w*)", lambda m: m.group(1).upper() + m.group(2), "hello big w
```

VERBOSE makes a long pattern readable:

```
DATE = re.compile(r"""
    (?P<year>\d{4}) -      # year
    (?P<month>\d{2}) -     # month
    (?P<day>\d{2})         # day
""", re.VERBOSE)
```

Whitespace and `#` comments are ignored in the pattern (escape a literal space as `\ ` or `[ ]`).

Other tools. `re.split(r"[,;]\s*", text)` splits on a pattern; `re.finditer` yields `Match` objects with `.start()` / `.end()` / `.span()`; `re.escape(s)` quotes a literal for use inside a

pattern; `re.fullmatch` for validation; `re.compile(...).pattern` and `.flags` for introspection; `(?i)` inline flags; non-greedy `*?` / `+?` / `??` ; `{n,m}?` .

Backreferences `\1` and `(?P=name)` match a repeat of a group: `r"(\w+) \1"` finds a doubled word.

Choosing between them

The string methods stay the first choice: `split` , `partition` , `startswith` , `replace` and `strip` are faster and clearer than any regex for a fixed delimiter or prefix. A regex earns its place when the *shape* matters — digits followed by a unit, a key with optional spaces around `=` , several alternatives at once — and `re.VERBOSE` with comments keeps such a pattern maintainable. `difflib` is for comparing and suggesting, `textwrap` for presenting, `Template` for text that came from outside; none of them is a substitute for a parser when the input has nested structure.

Pitfalls

`textwrap.wrap` on text with existing newlines — it treats them as spaces unless `replace_whitespace=False` .

Using `ratio()` thresholds without `cutoff` in `get_close_matches` .

`Template.substitute` raising `KeyError` for a missing placeholder when `safe_substitute` was wanted.

Variable-width lookbehind (`(?<=\d+)`) — not allowed; fixed width only.

`re.VERBOSE` swallowing a significant space in the pattern.

Forgetting `re.escape` when building a pattern from user text.

Key takeaways

`textwrap` : `wrap` , `fill` , `shorten` , `indent` , `dedent` ; `difflib` : `SequenceMatcher.ratio` , `get_opcodes` , `unified_diff` , `get_close_matches` .

`string.Template` for user-supplied templates; `unicodedata` for normalisation and accent stripping.

Named groups with `groupdict`; lookarounds assert context without consuming; `sub` with a function transforms matches.

`re.VERBOSE` for readable patterns; `re.escape` for literals; backreferences for repeats.

Reach for these before writing a wrapper, a diff or a fuzzy match by hand.

EXERCISE 1

Wrap, shorten, suggest

The first line is a width; the second a paragraph; the third a vocabulary of words; the remaining lines are words to check. Print the paragraph wrapped to the width with `textwrap.fill`, then `shorten`ed to the width with the placeholder `[…]`, then for each remaining word either `ok` if it is in the vocabulary or `did you mean <best>` using `difflib.get_close_matches` (`n=1`, default cutoff), or `no suggestion`.

Input: width, paragraph, vocabulary, words. **Output:** the wrapped lines, the shortened line, then one line per word.

```
20
The quick brown fox jumps over the lazy dog
apple banana cherry
appel
banana
zzz
```

prints

```
The quick brown fox
jumps over the lazy
dog
The quick brown […]
did you mean apple
ok
no suggestion
```

INPUT 1

```
20
The quick brown fox jumps over the lazy
dog
apple banana cherry
appel
banana
zzz
```

OUTPUT 1

```
The quick brown fox
jumps over the lazy
dog
The quick brown […]
did you mean apple
ok
no suggestion
```

REFERENCE SOLUTION

```python
import difflib
import sys
import textwrap

width = int(input())
paragraph = input()
vocab = input().split()
print(textwrap.fill(paragraph, width=width))
print(textwrap.shorten(paragraph, width=width, placeholder=" […]"))
for line in sys.stdin:
    word = line.strip()
    if not word:
        continue
    if word in vocab:
        print("ok")
        continue
    matches = difflib.get_close_matches(word, vocab, n=1)
    print(f"did you mean {matches[0]}" if matches else "no suggestion")
```

EXERCISE 2

Regex in depth

Each line is a command. `pairs <text>` extracts every `key=value` pair with named groups and prints them as `key:value` space-separated in order (or `none`). `kg <text>` prints the numbers immediately followed by `kg` (a lookahead, so the unit is not part of the match), space-separated (or `none`). `double <text>` doubles every integer in the text with `re.sub` and a function.

Input: command lines. **Output:** one line per command.

```
pairs host=db port=8080 debug
kg 5 kg of rice, 12 lb of flour, 7 kg of sugar
double a1 b22 c
```

prints

```
host:db port:8080
5 7
a2 b44 c
```

INPUT 1

```
pairs host=db port=8080 debug
kg 5 kg of rice, 12 lb of flour, 7 kg of sugar
double a1 b22 c
```

OUTPUT 1

```
host:db port:8080
5 7
a2 b44 c
```

REFERENCE SOLUTION

```python
import re
import sys

PAIR = re.compile(r"(?P<key>\w+)=(?P<value>\S+)")
KG = re.compile(r"\d+(?= kg)")

for line in sys.stdin:
    cmd, _, text = line.rstrip("\n").partition(" ")
    if cmd == "pairs":
        found = [f"{m['key']}:{m['value']}" for m in PAIR.finditer(text)]
        print(" ".join(found) if found else "none")
    elif cmd == "kg":
        found = KG.findall(text)
        print(" ".join(found) if found else "none")
    elif cmd == "double":
        print(re.sub(r"\d+", lambda m: str(int(m.group()) * 2), text))
```

QUIZ

1. What does `textwrap.dedent` do?

A. Indents every line

B. Removes the common leading whitespace from every line

C. Wraps to a width

D. Removes all whitespace

2. What does `difflib.get_close_matches("appel", words)` return?

A. The single best match

B. A list of the closest matches above a cutoff, best first

C. A ratio

D. A diff

3. Why use `string.Template` for a template supplied by a user?

A. It is faster than f-strings

B. Its `$name` placeholders cannot call methods or read attributes, so it cannot be abused

C. It supports format specs

D. It is the only option

4. What does `(?<=\$)\d+` match in `"cost $40"`?

A. `$40`

B. `40` — preceded by `$`, which is not part of the match

C. `cost`

D. Nothing

5. What does `re.sub(pattern, func, text)` pass to `func`?

A. The matched string

B. A `Match` object for each match; its return value is the replacement

C. The whole text

D. The group list

ANSWER KEY · TEXT-PROCESSING TOOLS — TEXTWRAP, DIFFLIB, STRING.TEMPLATE AND RE IN DEPTH

1. B It is the fix for indented triple-quoted strings inside functions.

2. B `n` bounds the list and `cutoff` (default 0.6) the similarity; the list may be empty.

3. B `str.format` on untrusted templates can reach into objects; `Template` is deliberately limited.

4. B A lookbehind asserts context without consuming it.

5. B That is how a computed replacement (doubling numbers, title-casing) is applied per match.

LESSON 5 · 13 MIN

Enums — Enum, IntEnum, StrEnum, Flag and auto

A status that is one of `pending`, `active`, `closed`; a colour; a permission set; a day of the week. Written as strings or small integers, such values can be misspelled, compared with the wrong constant, or given a value that means nothing — and nothing catches it. An `Enum` is a class whose instances are a fixed set of named constants: they are singletons, they compare by identity, they iterate in definition order, they look themselves up by name or value, and they print as `Status.ACTIVE`. This lesson covers `Enum`, `auto`, lookup and iteration, `IntEnum` and `StrEnum` for values that must also be ints or strings, `Flag` for combinable bits, `unique`, methods on enums, and enums in `match`.

Enum

```
from enum import Enum, auto

class Status(Enum):
    PENDING = "pending"
    ACTIVE = "active"
    CLOSED = "closed"

s = Status.ACTIVE
s.name, s.value              # 'ACTIVE', 'active'
Status("active")             # Status.ACTIVE — lookup by value; ValueError if
Status["ACTIVE"]             # by name; KeyError if absent
s is Status.ACTIVE           # True — members are singletons
s == "active"                # False — an Enum is not its value
list(Status)                 # [Status.PENDING, Status.ACTIVE, Status.CLOSED]
len(Status), Status.ACTIVE in Status
str(s), repr(s)              # 'Status.ACTIVE', '<Status.ACTIVE: 'active'>'
```

Members are created from the class body's assignments; the value can be any object, but strings and ints are the norm. Two names with the same value are *aliases* for one member (`Status.OPEN = "active"` would be another name for `ACTIVE`); `@enum.unique` makes that a `ValueError`. `auto()` assigns values automatically (1, 2, 3 … for `Enum`; the lower-cased name for `StrEnum`):

```
class Direction(Enum):
    NORTH = auto()
    EAST = auto()
    SOUTH = auto()
    WEST = auto()
```

Why an enum

A closed set. `Status("archived")` is an immediate `ValueError` — a typo cannot survive.

Identity. `if status is Status.ACTIVE` cannot be confused with a similar string.

A type. Hints say `def close(s: Status)`; a checker rejects a string.

Iteration and reflection. A dropdown is `[m.value for m in Status]`; a parser is `Status(text)`.

A namespace for behaviour. Methods on the enum put the logic next to the values.

Methods and properties

```
class Planet(Enum):
    EARTH = (5.97e24, 6.371e6)
    MARS = (6.42e23, 3.390e6)

    def __init__(self, mass, radius):        # unpacks the tuple value
        self.mass, self.radius = mass, radius

    @property
    def gravity(self):
        return 6.674e-11 * self.mass / self.radius ** 2

    @classmethod
    def heaviest(cls):
        return max(cls, key=lambda p: p.mass)

Planet.MARS.gravity, Planet.heaviest()
```

An enum is a class: it can have methods, properties, class methods and a custom `__str__`. The `__init__` receives the value (unpacked if a tuple), which is how a member carries several attributes.

IntEnum and StrEnum

```
from enum import IntEnum, StrEnum

class Priority(IntEnum):
    LOW = 1
    HIGH = 3

Priority.HIGH > Priority.LOW        # True — ordered like ints
Priority.HIGH + 1, Priority(3)      # 4, Priority.HIGH
sorted([Priority.HIGH, Priority.LOW])

class Colour(StrEnum):               # 3.11
    RED = auto()                     # value 'red'
    BLUE = "blue"

Colour.RED == "red"                 # True — it IS a str
f"{Colour.RED}"                     # 'red'
json.dumps({"c": Colour.RED})       # '{"c": "red"}'
```

A plain `Enum` member is *not* comparable to its value, and that strictness is usually what you want. `IntEnum` and `StrEnum` are subclasses of `int` and `str` too, so they compare, sort, serialise and substitute where the raw value is expected — the right choice for values that cross a boundary (a database column, a JSON field, an HTTP status) or must be ordered. The cost is that `Priority.LOW == 1` is also true for a `1` that came from anywhere.

Flag

```python
from enum import Flag, auto

class Perm(Flag):
    READ = auto()        # 1
    WRITE = auto()       # 2
    EXEC = auto()        # 4

p = Perm.READ | Perm.EXEC
Perm.WRITE in p          # False
p & Perm.READ            # <Perm.READ: 1> — truthy
p | Perm.WRITE           # <Perm.READ|WRITE|EXEC: 7>
Perm(0), Perm(5)         # <Perm: 0>, <Perm.READ|EXEC: 5>
list(p)                  # [Perm.READ, Perm.EXEC] (3.11)
```

`Flag` members combine with `|`, `&`, `^` and `~`, test membership with `in`, and are the typed version of the bit masks of Module 2. `IntFlag` also behaves as an int.

Enums in match and as keys

```python
match status:
    case Status.ACTIVE:              # a dotted name — a value pattern
        ...
    case Status.PENDING | Status.CLOSED:
        ...

counts = {s: 0 for s in Status}      # hashable, so dict keys and set members
```

The `match` rule of Module 3 applies: dotted names compare, bare names capture — `case ACTIVE:` would be a capture and a bug.

Pitfalls

Comparing a plain `Enum` member with its value (`Status.ACTIVE == "active"` is `False`).

Two members with the same value silently becoming aliases — use `@unique`.

`case ACTIVE:` without the class prefix.

Using an `Enum` where JSON or a database needs the raw value — `.value`, or `StrEnum` / `IntEnum`.

Mutating enum members or adding members at run time (they are constants).

Choosing `IntEnum` for identity-only values, losing the strictness.

Key takeaways

An `Enum` is a closed set of named singleton constants with `.name`, `.value`, lookup by `Cls(value)` and `Cls[name]`, and iteration in definition order.

`auto()` numbers members; `@unique` forbids aliases; methods, properties and tuple-valued `__init__` put behaviour on members.

`IntEnum` / `StrEnum` are also ints/strings for ordering and serialisation; a plain `Enum` is strict.

`Flag` members combine with bitwise operators and test with `in`.

In `match`, use the dotted name; enums are hashable keys.

EXERCISE 1

A status machine

Define `Status(Enum)` with values `pending`, `active` and `closed`, and a method `next()` returning the next status in definition order (`closed` stays `closed`). Commands: `set <value>` (look up by value; `unknown status` on `ValueError`), `next` (advance and print the new status's name), `show` (print `<name>=<value>`), `list` (all members as `name=value` space-separated).

Input: commands (the status starts as `pending`). **Output:** one line per `next`, `show`, `list` and failed `set`.

```
list
set active
next
next
show
set archived
```

prints

```
PENDING=pending ACTIVE=active CLOSED=closed
CLOSED
CLOSED
CLOSED=closed
unknown status
```

INPUT 1

```
list
set active
next
next
show
set archived
```

OUTPUT 1

```
PENDING=pending ACTIVE=active
CLOSED=closed
CLOSED
CLOSED
CLOSED=closed
unknown status
```

REFERENCE SOLUTION

```python
import sys
from enum import Enum


class Status(Enum):
    PENDING = "pending"
    ACTIVE = "active"
    CLOSED = "closed"

    def next(self):
        members = list(type(self))
        i = members.index(self)
        return members[min(i + 1, len(members) - 1)]


status = Status.PENDING
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "set":
        try:
            status = Status(args[0])
        except ValueError:
            print("unknown status")
    elif cmd == "next":
        status = status.next()
        print(status.name)
    elif cmd == "show":
        print(f"{status.name}={status.value}")
    elif cmd == "list":
        print(" ".join(f"{m.name}={m.value}" for m in Status))
```

EXERCISE 2

Typed permission flags

Define `Perm(Flag)` with `READ`, `WRITE` and `EXEC` via `auto()`. The first line grants permissions by name (possibly none). Commands: `has NAME` (`True` / `False` via `in`), `add NAME`, `remove NAME`, `show` — printing the value and the names of the set members in definition order as `<value>: <names>` (or `0: none`).

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

```
READ EXEC
show
has WRITE
add WRITE
remove READ
show
```

prints

```
5: READ EXEC
False
6: WRITE EXEC
```

INPUT 1

```
READ EXEC
show
has WRITE
add WRITE
remove READ
show
```

OUTPUT 1

```
5: READ EXEC
False
6: WRITE EXEC
```

REFERENCE SOLUTION

```
import sys
from enum import Flag, auto


class Perm(Flag):
    READ = auto()
    WRITE = auto()
    EXEC = auto()


perms = Perm(0)
for name in input().split():
    perms |= Perm[name]
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "has":
        print(Perm[args[0]] in perms)
    elif cmd == "add":
        perms |= Perm[args[0]]
    elif cmd == "remove":
        perms &= ~Perm[args[0]]
    elif cmd == "show":
        names = [m.name for m in Perm if m in perms]
        print(f"{perms.value}: {' '.join(names) if names else 'none'}")
```

QUIZ

1. What is `Status.ACTIVE == "active"` for a plain `Enum` with value `"active"`?

A. `True`

B. `False` — a member is not its value

C. `TypeError`

D. Depends on `auto`

2. How is a member looked up by value?

A. `Status["active"]`

B. `Status("active")`

C. `Status.value("active")`

D. `Status.get("active")`

3. What does `@enum.unique` do?

A. Sorts the members

B. Raises `ValueError` if two members share a value (which would otherwise be aliases)

C. Makes values unique automatically

D. Freezes the enum

4. Which enum class makes members usable as ints in arithmetic and comparisons?

A. `Enum`

B. `IntEnum`

C. `Flag`

D. `StrEnum`

5. In `match`, which pattern compares against an enum member?

A. `case ACTIVE:`

B. `case Status.ACTIVE:`

C. `case "ACTIVE":`

D. `case Status:`

ANSWER KEY · ENUMS — ENUM, INTENUM, STRENUM, FLAG AND AUTO

1. B `StrEnum` would make it true; the plain `Enum` is deliberately strict.

2. B Calling the class looks up by value (`ValueError` if absent); subscripting looks up by name.

3. B Duplicate values silently create aliases; `unique` turns that into an error.

4. B `IntEnum` members are ints; `StrEnum` members are strings; plain `Enum` members are neither.

5. B The dotted name is a value pattern; a bare name would capture and match anything.

CHECKPOINT · 22 MIN

Checkpoint — The standard library in depth

This checkpoint covers the whole module: `math`, `statistics`, `Fraction`, `Decimal` and seeded `random`; `datetime` arithmetic, parsing and formatting, aware datetimes and `zoneinfo`; `deque`, `Counter` arithmetic, `OrderedDict` for an LRU cache, `ChainMap` and `namedtuple`; `textwrap`, `difflib`, `string.Template` and the deeper `re` features; and enums with `auto`, `IntEnum`, `StrEnum` and `Flag`.

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:

- Which `statistics` function is the sample standard deviation, and what does an empty input do?
- Why is `Decimal("0.1")` right and `Decimal(0.1)` wrong, and what does `quantize` take?
- What does seeding `random.Random(42)` guarantee, and why prefer an instance to the module functions?
- What is the difference between `astimezone` and `replace(tzinfo=…)`?
- Which `OrderedDict` method makes an LRU cache possible?
- What does `Counter(a) - Counter(b)` drop?
- Why is `Status.ACTIVE == "active"` false for a plain `Enum`, and which enum class makes it true?

The three programs are a statistics report built with exact and rounded arithmetic, a schedule calculator that converts between time zones and adds durations, and an LRU cache on `OrderedDict` driven by commands with an enum for the operation names.

EXERCISE 1

Statistics report

Read amounts as decimal strings with two decimals, one per line. Print the mean, median and population standard deviation computed by `statistics` over floats, each with three decimals; then the exact total as a `Decimal` sum of the strings; then the total rounded to whole units with

`quantize(Decimal("1"), rounding=ROUND_HALF_UP)`; then the mean as an exact `Fraction` of the total in cents over the count, reduced.

Input: at least one amount per line. **Output:** `mean <m>`, `median <d>`, `pstdev <s>`, `total <t>`, `rounded <r>`, `exact-mean <p>/<q>` (or `exact-mean <n>` when the denominator is 1).

```
10.00
20.50
30.00
```

prints

```
mean 20.167
median 20.500
pstdev 8.168
total 60.50
rounded 61
exact-mean 6050/3
```

INPUT 1

```
10.00
20.50
30.00
```

OUTPUT 1

```
mean 20.167
median 20.500
pstdev 8.168
total 60.50
rounded 61
exact-mean 6050/3
```

REFERENCE SOLUTION

```python
import statistics
import sys
from decimal import Decimal, ROUND_HALF_UP
from fractions import Fraction

texts = [line.strip() for line in sys.stdin if line.strip()]
floats = [float(t) for t in texts]
print(f"mean {statistics.fmean(floats):.3f}")
print(f"median {statistics.median(floats):.3f}")
print(f"pstdev {statistics.pstdev(floats):.3f}")
total = sum(Decimal(t) for t in texts)
print(f"total {total}")
print(f"rounded {total.quantize(Decimal('1'), rounding=ROUND_HALF_UP)}")
cents = sum(int(t.replace(".", "")) for t in texts)
exact = Fraction(cents, len(texts))
print(f"exact-mean {exact.numerator}" if exact.denominator == 1 else f"exact-mean {e
```

EXERCISE 2

Schedule across zones

The first line is `<zone> <ISO start datetime>` — a meeting start as a wall-clock time in that zone. The following lines are either `add <minutes>` (shift the start by a `timedelta`), or `in <zone>` (print the current start converted to that zone in ISO format), or `utc` (print it in UTC). Use aware datetimes throughout.

Input: the first line, then commands. **Output:** one line per `in` and `utc`.

```
Asia/Kolkata 2024-03-10T09:00:00
utc
in Europe/London
add 90
in America/New_York
```

prints

```
2024-03-10T03:30:00+00:00
2024-03-10T03:30:00+00:00
2024-03-10T00:00:00-05:00
```

INPUT 1

```
Asia/Kolkata 2024-03-10T09:00:00
utc
in Europe/London
add 90
in America/New_York
```

OUTPUT 1

```
2024-03-10T03:30:00+00:00
2024-03-10T03:30:00+00:00
2024-03-10T00:00:00-05:00
```

REFERENCE SOLUTION

```python
import sys
from datetime import datetime, timedelta, timezone
from zoneinfo import ZoneInfo

zone, text = input().split()
start = datetime.fromisoformat(text).replace(tzinfo=ZoneInfo(zone))
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        start += timedelta(minutes=int(args[0]))
    elif cmd == "in":
        print(start.astimezone(ZoneInfo(args[0])).isoformat())
    elif cmd == "utc":
        print(start.astimezone(timezone.utc).isoformat())
```

EXERCISE 3

Cache operations as an enum

Define `Op(StrEnum)` with members `GET`, `PUT`, `DEL` and `SIZE` whose values are the lower-case words (`auto()`). Parse each command's first word with `Op(word)` — an unknown word prints `unknown op` — and drive an LRU cache on `OrderedDict` of the given capacity: `get k` prints the value or `-1`, `put k v`, `del k` (prints `deleted` or `missing`), `size` prints the entry count. At the end print the keys from least to most recently used.

Input: the capacity, then commands. **Output:** the command results, then `keys …` (or `keys none`).

```
2
put a 1
put b 2
get a
put c 3
del b
size
flush
```

prints

```
1
missing
2
unknown op
keys a c
```

INPUT 1

```
2
put a 1
put b 2
get a
put c 3
del b
size
flush
```

OUTPUT 1

```
1
missing
2
unknown op
keys a c
```

REFERENCE SOLUTION

```
import sys
from collections import OrderedDict
from enum import StrEnum, auto


class Op(StrEnum):
    GET = auto()
    PUT = auto()
    DEL = auto()
    SIZE = auto()


capacity = int(input())
cache = OrderedDict()
for line in sys.stdin:
    word, *args = line.split()
    try:
        op = Op(word)
    except ValueError:
        print("unknown op")
        continue
    match op:
        case Op.GET:
            if args[0] in cache:
                cache.move_to_end(args[0])
                print(cache[args[0]])
            else:
                print(-1)
        case Op.PUT:
            cache[args[0]] = int(args[1])
            cache.move_to_end(args[0])
            if len(cache) > capacity:
                cache.popitem(last=False)
        case Op.DEL:
            print("deleted" if cache.pop(args[0], None) is not None else "missing")
        case Op.SIZE:
            print(len(cache))
print("keys", " ".join(cache) if cache else "none")
```

CHECKPOINT QUESTIONS

1. What is `math.comb(5, 2)`?

A. `20`

B. `10`

C. `25`

D. `120`

2. What does `statistics.median([1, 2, 3, 4])` return?

A. `2`

B. `2.5`

C. `3`

D. `StatisticsError`

3. Which expression is exactly three tenths?

A. `0.1 + 0.2`

B. `Fraction("0.1") + Fraction("0.2")`

C. `Fraction(0.1) + Fraction(0.2)`

D. `round(0.1 + 0.2, 1)`

4. What does `date(2024, 3, 1) - date(2024, 2, 1)` give?

A. `29`

B. `timedelta(days=29)`

C. `timedelta(days=28)`

D. `TypeError`

5. What does `datetime.now()` return, and why avoid it in a judged program?

A. An aware UTC datetime; it is fine

B. A naive local datetime; it is nondeterministic

C. A timestamp float

D. A string

6. What does `ZoneInfo("Europe/London")` know that `timezone(timedelta(hours=1))` does not?

A. Nothing

B. The daylight-saving rules — the offset changes twice a year

C. The zone's name only

D. Leap seconds

7. What is `deque(maxlen=3)` after appending 1, 2, 3, 4?

A. `[1, 2, 3]`

B. `[2, 3, 4]`

C. `[1, 2, 3, 4]`

D. It raises

8. What does `Counter("aab") & Counter("abb")` give?

A. `Counter({'a': 2, 'b': 2})`

B. `Counter({'a': 1, 'b': 1})`

C. `Counter({'a': 3, 'b': 3})`

D. `Counter()`

9. What does `textwrap.shorten("a b c d", width=5, placeholder="…")` produce?

A. `'a b c'`

B. `'a b…'`

C. `'a b c…'`

D. `'…'`

10. Which regex construct matches digits only when followed by `px`, without including `px`?

A. `\d+px`

B. `\d+(?=px)`

C. `(?<=px)\d+`

D. `\d+(px)`

11. What is the value of the second member of `class C(Enum): A = auto(); B = auto()`?

A. `'B'`

B. `2`

C. `1`

D. `'b'`

12. What does `Perm.READ in (Perm.READ | Perm.EXEC)` evaluate to?

A. `False`

B. `True`

C. `TypeError`

D. `Perm.READ`

ANSWER KEY · CHECKPOINT — THE STANDARD LIBRARY IN DEPTH

1. B Combinations choose unordered pairs: 5 × 4 / 2 = 10. `perm(5, 2)` is 20.

2. B For an even count the median is the mean of the two middle values.

3. B Fractions built from strings are exact; from floats they inherit the binary error.

4. B Date subtraction yields a `timedelta`; 2024 is a leap year, so February has 29 days.

5. B The clock differs on every run; pass the time in as data instead.

6. B Fixed offsets are only right for zones without DST.

7. B A bounded deque discards from the opposite end when full.

8. B Intersection takes the minimum count per key.

9. B It fits whole words plus the placeholder within the width.

10. B A lookahead asserts the following text; the match itself stops before it.

11. B `auto()` numbers plain `Enum` members from 1; on a `StrEnum` it would give the lower-cased name.

12. B `in` on flags tests whether every bit of the left operand is set in the right.