Module 11 of 20

Iterators, generators and itertools

The iteration protocol with iter, next and StopIteration, generators with yield and yield from, generator expressions and the short-circuit consumers, the itertools toolkit, functools and operator, and lazy pipelines of generator stages.

Every loop, comprehension and aggregate in Python runs on one protocol — `iter()`, then `next()` until `StopIteration` — and generators are the way that protocol is written in practice: a function with `yield` whose locals are its state. Once that is clear, laziness stops being a curiosity and becomes the default design: values are produced on demand, infinite sequences are ordinary, a million records cost the memory of one, and a chain of small stages processes a stream in a single pass. This module builds that understanding from the protocol up and ends with a real pipeline.

The iteration protocol defines iterable and iterator, desugars `for`, and writes an iterator class by hand. Generators covers `yield`, suspension, infinite generators bounded by `islice`, `yield from`, and `send` / `close` in outline. Generator expressions covers the lazy comprehension and the short-circuit consumers `any`, `all` and `next`. itertools catalogues the toolkit, with `groupby`'s consecutive-run rule and the combinatoric generators. functools and operator covers `cache`, `partial`, `reduce`, `wraps`, `singledispatch` and the operator factories. Lazy pipelines assembles the parts into a log processor with one job per stage.

The exercises are whole programs: a hand-written range iterator that shows one-shot exhaustion, a sentinel reader with `iter(callable, sentinel)`, an infinite Fibonacci generator sliced three ways, a tree walk with `yield from`, short-circuit drills, a trace that shows lazy versus eager evaluation, run-length encoding with `groupby`, combinatoric counts, cache statistics, a `singledispatch` describer, a purchase pipeline and a sliding-window stream. The checkpoint adds a run-length codec, a subset-sum search over `combinations` that stops at the first hit, and a log pipeline of generator stages.

CONTENTS

1	The iteration protocol — iter, next and StopIteration	13 min
2	Generators — functions that yield	14 min

3	Generator expressions — lazy comprehensions	12 min
4	itertools — the iterator toolkit	15 min
5	functools and operator — the function toolkit	13 min
6	Lazy pipelines — a worked log-processing example	14 min
✓	Checkpoint — Iterators, generators and itertools	25 min

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

LESSON 1 · 13 MIN

The iteration protocol — iter, next and StopIteration

Every `for` loop, comprehension, `sum`, `list()` and `zip` in Python runs on one small protocol: ask an object for an *iterator* with `iter()`, call `next()` on it until it raises `StopIteration`. Understanding that protocol explains why a file can be looped over only once, why `zip` and `enumerate` are lazy, why `range` can be iterated twice but `map(...)` cannot, and how to write a class that `for` understands. This lesson defines iterable and iterator precisely, desugars the `for` loop, shows the built-in iterators and what exhaustion means, writes an iterator class by hand, and covers the two-argument `iter(callable, sentinel)` form.

Iterable versus iterator

An **iterable** is anything `iter()` accepts: it has `__iter__` (returning an iterator) or `__getitem__` with integer indexes. Lists, strings, dicts, sets, ranges, files, generators.

An **iterator** is the object that produces the values: it has `__next__` (returning the next value or raising `StopIteration`) and `__iter__` returning itself, so that an iterator is also an iterable.

```
xs = [1, 2, 3]
it = iter(xs)          # a list_iterator
next(it)               # 1
next(it)               # 2
next(it)               # 3
next(it)               # StopIteration
next(it, "done")       # 'done' — a default instead of the exception
```

A list is iterable but not an iterator: `next(xs)` is `TypeError: 'list' object is not an iterator`. Calling `iter(xs)` again gives a *fresh* iterator starting at the beginning; the list can be looped over any number of times. The iterator itself is one-shot: once it has raised `StopIteration`, it raises it forever.

The for loop, desugared

```
for x in xs:
    body(x)

# is exactly:
_it = iter(xs)
while True:
    try:
        x = next(_it)
    except StopIteration:
        break
    body(x)
```

So `for` calls `iter()` once and `next()` per iteration, and stops on `StopIteration`. Every other consumer — `list(xs)`, `sum(xs)`, `"".join(xs)`, `max(xs)`, `x in xs` without `__contains__`, unpacking `a, b = xs` — does the same. That is why the protocol is the *only* thing a class needs to implement to work everywhere.

Lazy built-ins

`zip`, `enumerate`, `map`, `filter`, `reversed`, `range` iterators, dict views' iterators, file objects and generators all produce values on demand and are **iterators** (or produce one), not lists:

```
pairs = zip([1, 2], "ab")
list(pairs)            # [(1, 'a'), (2, 'b')]
list(pairs)            # [] — exhausted; zip returned a one-shot iterator

r = range(3)
list(r), list(r)       # [0, 1, 2] twice — range is an iterable, iter(r) makes

with open(path) as f:
    lines = list(f)    # a file is its own iterator: after this it is at the e
    more = list(f)     # []
```

The practical rule: if you need to go over the values twice, materialise them (`list(...)`) or make sure you have an iterable rather than an iterator. `itertools.tee` splits an iterator into several (Module 11 lesson 4), at the cost of buffering.

Writing an iterator class

```python
class Countdown:
    def __init__(self, start):
        self.current = start

    def __iter__(self):
        return self                          # an iterator returns itself

    def __next__(self):
        if self.current <= 0:
            raise StopIteration
        self.current -= 1
        return self.current + 1

list(Countdown(3))          # [3, 2, 1]
c = Countdown(2)
list(c), list(c)            # [2, 1], [] — one-shot, like every iterator
```

The state lives in the object; `__next__` advances it and raises when done. To make a *re-iterable* container, give the container an `__iter__` that returns a *new* iterator each time — a separate class, or (next lesson) a generator, which is what almost every real `__iter__` is:

```python
class Playlist:
    def __init__(self, tracks):
        self._tracks = list(tracks)

    def __iter__(self):
        return iter(self._tracks)            # a fresh iterator per loop
```

---

iter with a sentinel

`iter(callable, sentinel)` calls `callable()` repeatedly until it returns `sentinel`:

```python
with open(path, "rb") as f:
    for chunk in iter(lambda: f.read(4096), b""):
        process(chunk)

for line in iter(input, "END"):           # read until a line equals "END"
    handle(line)
```

The second form is the cleanest "read until sentinel" loop in the language — no `while True`, no `break`.

The protocol elsewhere

`in` falls back on iteration when there is no `__contains__`; `reversed` needs `__reversed__` or the sequence protocol; `len` is *not* part of iteration — an iterator has no length, which is why `len(zip(...))` is a `TypeError` and why `sum(1 for _ in it)` is how you count one. Unpacking `a, *rest = it` consumes the iterator entirely. `bool(it)` is always `True` for an iterator — emptiness cannot be tested without consuming.

Pitfalls

Iterating an iterator twice and getting nothing the second time.

`len()` on an iterator.

`next()` on an iterable (a list) instead of `iter(list)`.

A `__next__` that never raises `StopIteration` — every loop over it is infinite.

Testing `if it:` for emptiness.

Returning a list from `__iter__` (must be an iterator; `iter(list)` fixes it).

Key takeaways

Iterable: has `__iter__` (or `__getitem__`); iterator: has `__next__` and `__iter__` returning itself; `for` calls `iter()` once and `next()` until `StopIteration`.

Iterators are one-shot; containers give a fresh iterator each time; `zip`, `map`, files and generators are one-shot iterators.

Write `__next__` to advance state and raise `StopIteration` when done; give a container an `__iter__` that returns a new iterator.

`next(it, default)` avoids the exception; `iter(callable, sentinel)` loops until a value.

Iterators have no `len`, no emptiness test and cannot be rewound — materialise when you need those.

EXERCISE 1

A range of your own

Implement `Span(start, stop, step)` as an **iterator class** with `__iter__` returning `self` and `__next__` producing `start, start + step, …` while below `stop` (`step` is positive), then raising `StopIteration`. Read `start stop step`, print the values from a first pass, then try a second pass over the **same object** and print `exhausted` if it yields nothing. Finally print the values from `Span(...)` consumed with `next` and a default, three times.

Input: `start stop step`. **Output:** `first: <values>` (or `first: (empty)`), `second: <values>` or `second: exhausted`, then three lines from `next(it, "done")`.

```
1 8 3
```

prints

```
first: 1 4 7
second: exhausted
1
4
7
```

INPUT 1

```
1 8 3
```

OUTPUT 1

```
first: 1 4 7
second: exhausted
1
4
7
```

REFERENCE SOLUTION

```
class Span:
    def __init__(self, start, stop, step):
        self.current = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        return self

    def __next__(self):
        if self.current ⩾ self.stop:
            raise StopIteration
        value = self.current
        self.current += self.step
        return value


start, stop, step = map(int, input().split())
span = Span(start, stop, step)
first = list(span)
print("first:", " ".join(map(str, first)) if first else "(empty)")
second = list(span)
print("second:", " ".join(map(str, second)) if second else "exhausted")
it = Span(start, stop, step)
for _ in range(3):
    print(next(it, "done"))
```

EXERCISE 2

Until the sentinel

Use the two-argument form `iter(input, "END")` to read integer lines until a line equal to `END`, and print their sum and count. Then read the rest of the input with a plain `for line in sys.stdin` loop and print how many lines follow the sentinel.

Input: integer lines, `END`, then any further lines. **Output:** `sum <s> count <n>`, then `after <k>`.

```
3
4
END
x
y
```

prints

```
sum 7 count 2
after 2
```

INPUT 1

```
3
4
END
x
y
```

OUTPUT 1

```
sum 7 count 2
after 2
```

REFERENCE SOLUTION

```python
import sys

total = count = 0
for line in iter(input, "END"):
    total += int(line)
    count += 1
print(f"sum {total} count {count}")
after = sum(1 for _ in sys.stdin)
print(f"after {after}")
```

QUIZ

1. Which statement is correct?

A. A list is an iterator

B. A list is an iterable; `iter(list)` returns a fresh iterator each time

C. `next(list)` returns the first element

D. An iterator can be rewound with `reset()`

2. What does `for x in xs:` call?

A. `xs.__next__()` repeatedly

B. `iter(xs)` once, then `next()` on the result until `StopIteration`

C. `xs.__getitem__` only

D. `len(xs)` then indexing

3. After `p = zip(a, b); list(p)`, what is `list(p)`?

A. The same pairs again

B. `[]` — `zip` returns a one-shot iterator

C. `TypeError`

D. `None`

4. What must `__next__` do when there are no more values?

A. Return `None`

B. Raise `StopIteration`

C. Return `False`

D. Raise `IndexError`

5. What does `iter(f.readline, "")` do?

A. Reads the whole file at once

B. Calls `f.readline()` repeatedly until it returns the empty string

C. Raises `TypeError`

D. Reads one line

ANSWER KEY · THE ITERATION PROTOCOL — ITER, NEXT AND STOPITERATION

1. B Containers are iterables; the iterator is a separate one-shot object. `next` on a list is a `TypeError`.

2. B That desugaring is the whole protocol; every consumer does the same.

3. B `zip`, `map`, `filter`, `enumerate` and generators are exhausted after one pass; materialise if you need two.

4. B `StopIteration` is the signal the protocol defines; returning `None` would make `None` a value.

5. B The two-argument form loops a callable until the sentinel value appears — here, end of file.

LESSON 2 · 14 MIN

Generators — functions that yield

A generator is an iterator written as a function: instead of a class with state in attributes and a `__next__` that reads it, you write a function with `yield`, and Python keeps the state — every local variable and the position in the code — between calls. The previous lesson's `Countdown` becomes three lines. Generators are how most iterators in Python are written, how large or infinite sequences are produced without memory, and how pipelines of transformations are composed. This lesson covers the mechanics of `yield`, laziness and suspension, infinite generators with `islice`, `yield from`, `return` in a generator, and the `send` / `close` protocol in outline.

---

yield

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1

list(countdown(3))          # [3, 2, 1]
g = countdown(2)
type(g)                     # <class 'generator'>
next(g), next(g)            # 2, 1
next(g)                     # StopIteration
```

A function containing `yield` does not run when called: it returns a *generator object*. Each `next()` runs the body until the next `yield`, hands out the yielded value, and **suspends** — locals intact, position remembered. The next `next()` resumes right after the `yield`. When the function returns (falls off the end or hits `return`), the generator raises `StopIteration`. The generator object is an iterator: `for` works, `list()` works, it is one-shot.

Laziness

```python
def squares(n):
    for i in range(n):
        print(f"computing {i}")
        yield i * i

g = squares(3)              # nothing printed yet
next(g)                     # computing 0 → 0
for v in g:                 # computing 1, computing 2
    pass
```

Work happens only when a value is asked for. Three consequences: a generator over a million items uses the memory of one item; a consumer that stops early (`next`, `any`, `break`) never triggers the rest; and an exception inside a generator surfaces at the `next()` that runs the failing code, not at the call that created the generator.

Infinite generators

Because values are produced on demand, a generator may never end, and a consumer takes what it needs:

```python
import itertools

def naturals():
    n = 0
    while True:
        yield n
        n += 1

def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

list(itertools.islice(fibonacci(), 10))        # [0, 1, 1, 2, 3, 5, 8, 13, 21,
next(x for x in naturals() if x * x > 500)     # 23
list(itertools.takewhile(lambda x: x < 100, fibonacci()))
```

`islice(g, n)` takes the first `n`; `takewhile` takes while a condition holds; `next(...)` takes the first matching. Never call `list()`, `sum()` or `sorted()` on an infinite generator without a bound.

Generators as iter

```
class Tree:
    def __init__(self, value, children=()):
        self.value, self.children = value, list(children)

    def __iter__(self):                          # depth-first, pre-order
        yield self.value
        for child in self.children:
            yield from child                     # delegate to the child's iter
```

Writing `__iter__` as a generator is the idiomatic way to make a class iterable — and, because each call creates a fresh generator, the object is re-iterable. `yield from iterable` yields every value of a sub-iterable, which is what makes recursive traversals a few lines: flattening nested lists, walking a tree, walking a JSON document (Module 7).

return and StopIteration

A `return value` inside a generator ends it; the value is attached to the `StopIteration` (`e.value`) and is what `yield from` evaluates to. Raising `StopIteration` *manually* inside a generator is an error since 3.7 (it becomes `RuntimeError`) — use `return`. `next(g, default)` handles exhaustion without a `try`.

Generators are one-shot

```
g = countdown(3)
sum(g)          # 6
sum(g)          # 0 — exhausted
```

A generator *object* cannot be rewound. To iterate again, call the generator *function* again. Store the function call in a helper (`def data(): return countdown(3)`) or materialise (`list(...)`) when two passes are needed.

Pipelines

Generators compose: each stage consumes the previous one lazily, and only the final consumer pulls values through the chain:

```
def read_lines(text):
    for line in text.splitlines():
        yield line.strip()

def non_empty(lines):
    for line in lines:
        if line:
            yield line

def parse(lines):
    for line in lines:
        key, _, value = line.partition("=")
        yield key, int(value)

total = sum(v for _, v in parse(non_empty(read_lines(text))))
```

No intermediate lists; each line flows through all stages before the next line is read. Module 11 lesson 6 builds a full pipeline; the shape is stage functions that take an iterable and `yield`.

---

send, throw and close

A generator is also a *coroutine* in the original sense: `g.send(value)` resumes it with `value` as the result of the `yield` expression (`received = yield`), `g.throw(exc)` raises inside it at the `yield`, and `g.close()` raises `GeneratorExit` there so `finally` blocks run. These are the mechanism `@contextmanager` uses (Module 10) and the ancestor of `async` / `await` (Module 17); ordinary iteration code never needs them, and a generator with `finally` should be closed (or fully consumed) so its cleanup runs.

Pitfalls

Calling a generator function and expecting the body to run — it runs on `next`.

`list()` or `sum()` on an infinite generator.

Iterating a generator object twice.

`raise StopIteration` inside a generator (use `return`).

A generator that mutates shared state as a side effect of being consumed — surprising when the consumer is lazy.

Forgetting `yield from` and writing `for x in sub: yield x` (fine, but `yield from` also forwards `send` / `throw`).

Key takeaways

A function with `yield` returns a generator: an iterator whose state is its locals; `next` runs to the next `yield` and suspends.

Values are produced lazily and never stored; infinite generators are bounded by `islice`, `takewhile` or `next`.

Write `__iter__` as a generator; `yield from` delegates and makes recursive traversals short.

`return` ends a generator (its value rides on `StopIteration`); generator objects are one-shot — call the function again.

Stages that take an iterable and `yield` compose into pipelines with no intermediate lists.

EXERCISE 1

Fibonacci, sliced three ways

Write an infinite generator `fib()` yielding `0, 1, 1, 2, …`. Read `n` and `limit` and print: the first `n` values (`itertools.islice`); the values below `limit` (`itertools.takewhile`); and the first value greater than `limit` (`next` over a filtered generator expression). Each call to `fib()` starts fresh.

Input: `n limit`. **Output:** `first: …`, `below: …`, `next: <value>`.

```
8 20
```

prints

```
first: 0 1 1 2 3 5 8 13
below: 0 1 1 2 3 5 8 13
next: 21
```

INPUT 1

```
8 20
```

OUTPUT 1

```
first: 0 1 1 2 3 5 8 13
below: 0 1 1 2 3 5 8 13
next: 21
```

REFERENCE SOLUTION

```
from itertools import islice, takewhile


def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b


n, limit = map(int, input().split())
print("first:", " ".join(map(str, islice(fib(), n))))
print("below:", " ".join(map(str, takewhile(lambda x: x < limit, fib()))))
print("next:", next(x for x in fib() if x > limit))
```

EXERCISE 2

Walk a tree with yield from

A tree arrives as a nested list literal: a list is a node whose first element is its value and whose remaining elements are child nodes. Write a generator `walk(node, depth=0)` that yields `(depth, value)` pairs in pre-order, delegating to children with `yield from`. Print each pair as `<depth>:<value>`, then the maximum depth.

Input: one nested list literal (parse with `ast.literal_eval`). **Output:** one line per node, then `max depth <d>`.

```
["root", ["a", ["a1"]], ["b"]]
```

prints

```
0:root
1:a
2:a1
1:b
max depth 2
```

INPUT 1

```
["root", ["a", ["a1"]], ["b"]]
```

OUTPUT 1

```
0:root
1:a
2:a1
1:b
max depth 2
```

REFERENCE SOLUTION

```
import ast


def walk(node, depth=0):
    yield depth, node[0]
    for child in node[1:]:
        yield from walk(child, depth + 1)


tree = ast.literal_eval(input())
deepest = 0
for depth, value in walk(tree):
    print(f"{depth}:{value}")
    deepest = max(deepest, depth)
print(f"max depth {deepest}")
```

QUIZ

1. What does calling a generator function do?

A. Runs the body to the first `yield`

B. Returns a generator object without running the body

C. Returns a list

D. Runs the whole body

2. What does `yield from sub` do?

A. Yields the iterable `sub` as one value

B. Yields each value of `sub` in turn, and forwards `send` / `throw`

C. Returns `sub`

D. Raises `StopIteration`

3. What happens with `list(count_forever())` where the generator never returns?

A. An empty list

B. The program never finishes (or runs out of memory)

C. `StopIteration`

D. A list of `None`

4. How is a generator ended from inside?

A. `raise StopIteration`

B. `return`, optionally with a value

C. `yield None`

D. `break`

5. Why is writing `__iter__` as a generator a good way to make a class iterable?

A. It is faster than a list

B. Each call creates a fresh generator, so the object can be iterated repeatedly and the state is the generator's locals

C. It makes the class hashable

D. It avoids `StopIteration`

ANSWER KEY · GENERATORS — FUNCTIONS THAT YIELD

1. B The body runs only when `next()` is called; that is what makes generators lazy.

2. B It delegates to a sub-iterator; recursive traversals are its main use.

3. B `list` pulls until `StopIteration`, which never comes; bound infinite generators with `islice` or `next`.

4. B `return` ends it; a manual `raise StopIteration` inside a generator becomes `RuntimeError` since 3.7.

5. B The container stays re-iterable and the traversal logic reads top to bottom.

LESSON 3 · 12 MIN

Generator expressions — lazy comprehensions

A generator expression is a comprehension in parentheses that produces a generator instead of a list: `(x * x for x in xs)`. It is the same lazy machinery as a generator function, written inline for the one-line cases — the argument to `sum`, `max`, `any`, `all`, `"".join`, `sorted`, `set`, `dict`, or a `for` loop. Choosing it over a list comprehension is a memory decision (O(1) instead of O(n)) and sometimes a time decision (short-circuit consumers stop early). This lesson covers the syntax and its one-shot nature, when it beats a list comprehension and when it does not, the short-circuit consumers, the `next(gen, default)` idiom, and the scoping detail that makes the first `for` clause evaluate eagerly.

The syntax

```
squares = (x * x for x in range(5))     # a generator object; nothing computed
type(squares)                            # <class 'generator'>
list(squares)                            # [0, 1, 4, 9, 16]
list(squares)                            # [] — one-shot, like every generator

total = sum(x * x for x in range(5))     # the parentheses of the call are enough
big = sum((x for x in xs), start=0)      # with a second argument, the genera
```

Everything a list comprehension can express — a filtering `if`, nested `for` clauses, a conditional expression — works identically. The only difference is that the result is produced lazily and cannot be indexed, sliced, measured with `len` or iterated twice.

Generator versus list comprehension

Situation	Use
Feeding a single consumer once (`sum` , `max` , `any` , `join` , `for`)	generator expression
The values are needed more than once, indexed, sliced or `len` -ed	list comprehension
The data is large or the source is itself lazy (a file, another generator)	generator expression
The consumer may stop early (`any` , `all` , `next` , `in`)	generator expression — work stops at the answer
You want to print or debug the intermediate values	list comprehension (a generator shows as `<generator object …>`)

```
sum([x * x for x in range(10 ** 6)])      # builds a million-element list, ther
sum(x * x for x in range(10 ** 6))        # sums as it goes; constant memory
```

For `sum` and friends the generator is usually also faster, because no list is allocated and grown. For `sorted` and `set` , which must see everything, a list comprehension and a generator expression cost about the same — the generator saves memory, not time.

Short-circuit consumers

`any` , `all` , `in` (on a generator) and `next` stop at the first value that decides the answer, and a generator expression feeds them exactly as many values as they need:

```
any(x < 0 for x in xs)                    # stops at the first negative
all(len(w) < 10 for w in words)           # stops at the first long word
next((x for x in xs if x % 7 == 0), None)   # the first multiple of 7, or None
first_long = next((w for w in words if len(w) > 8), "")
```

With a list comprehension, `any([x < 0 for x in xs])` computes every element before `any` looks at the first. The generator version is the idiom for "does one exist?" and "give me the first that…", and `next(gen, default)` is how "first that…" handles the case where nothing matches.

In other calls

```
", ".join(str(n) for n in nums)          # join needs strings; the generator
max(words, key=len)                      # not a generator: a key function
dict((k, len(k)) for k in keys)          # a dict from pairs; the dict comprel
set(w.lower() for w in words)            # a set; {w.lower() for w in words} i
sorted(x for x in xs if x)               # fine: sorted materialises anyway
tuple(x for x in xs)                     # there is no tuple comprehension; th
```

When a set, dict or list is the *goal*, use the matching comprehension; when the goal is a *computation over* the values, pass a generator expression to it.

The eager first clause

The iterable of the *first* `for` clause is evaluated immediately, when the generator expression is created; everything else is evaluated lazily as values are pulled:

```
def source():
    print("source evaluated")
    return [1, 2, 3]

g = (x for x in source())                # prints "source evaluated" now
values = list(g)                          # the loop body runs here
```

This means an error in the first iterable surfaces at creation, and that later clauses (`for y in f(x)`) run lazily. It also means a generator expression captures its outer variables by reference, like a lambda — the late-binding trap of Module 4 applies to a generator created in a loop and consumed later.

Nesting and readability

A generator expression inside another call inside another call is hard to read; give the inner one a name:

```
lengths = (len(line) for line in lines)
print(f"longest: {max(lengths)}")
```

Two clauses is fine; three, or a conditional expression plus a filter, wants a generator *function* with a name and a docstring.

Pitfalls

Iterating a generator expression twice.

`len(gen)`, `gen[0]`, `gen[:3]` — `TypeError`.

`sum([... for ...])` when a generator would do.

`any([...])` losing the short-circuit.

Printing a generator expression and seeing `<generator object>`.

A generator expression capturing a loop variable that changes before it is consumed.

Key takeaways

`(expr for x in it if cond)` is a lazy, one-shot comprehension; the call's parentheses suffice when it is the only argument.

Use it to feed `sum`, `max`, `any`, `all`, `join`, `next` and `for`; use a list comprehension when values are reused, indexed or measured.

`any` / `all` / `next` short-circuit, so a generator does only the work the answer needs; `next(gen, default)` is "first that…".

The first `for` iterable is evaluated at creation; the rest lazily; outer names are captured by reference.

Prefer the set/dict comprehension when a set or dict is the goal; name a generator expression when it gets long.

EXERCISE 1

Short-circuit drills

Read a threshold and a line of integers. With generator expressions only (no lists), print: whether any value exceeds the threshold; whether all values are positive; the first value exceeding the threshold or `none` (using `next` with a default); how many exceed it (`sum`); and the values that exceed it joined by commas (`join` over a generator of strings).

Input: the threshold, then a line of integers. **Output:** `any <bool>`, `all-positive <bool>`, `first <v|none>`, `count <n>`, `above <a,b,…>` (or `above none`).

```
5
3 8 1 9
```

prints

```
any True
all-positive True
first 8
count 2
above 8,9
```

INPUT 1

```
5
3 8 1 9
```

OUTPUT 1

```
any True
all-positive True
first 8
count 2
above 8,9
```

REFERENCE SOLUTION

```python
t = int(input())
xs = list(map(int, input().split()))
print(f"any {any(x > t for x in xs)}")
print(f"all-positive {all(x > 0 for x in xs)}")
print(f"first {next((x for x in xs if x > t), 'none')}")
print(f"count {sum(1 for x in xs if x > t)}")
above = ",".join(str(x) for x in xs if x > t)
print(f"above {above if above else 'none'}")
```

EXERCISE 2

Lazy versus eager, traced

`compute(i)` prints `compute <i>` and returns `i * i`. Read `n` and `target`. First evaluate `any(compute(i) > target for i in range(n))` — a generator expression, which stops at the first hit — then `any([compute(i) > target for i in range(n)])` — a list comprehension, which computes everything first. Print the trace lines as they happen and the two results as `lazy <bool>` and `eager <bool>`.

Input: `n target`. **Output:** the trace and the two result lines.

```
4 3
```

prints

```
compute 0
compute 1
compute 2
lazy True
compute 0
compute 1
compute 2
compute 3
eager True
```

INPUT 1

```
4 3
```

OUTPUT 1

```
compute 0
compute 1
compute 2
lazy True
compute 0
compute 1
compute 2
compute 3
eager True
```

REFERENCE SOLUTION

```
def compute(i):
    print(f"compute {i}")
    return i * i


n, target = map(int, input().split())
print(f"lazy {any(compute(i) > target for i in range(n))}")
print(f"eager {any([compute(i) > target for i in range(n)])}")
```

QUIZ

1. What is `type(x for x in range(3))`?

A. `list`

B. `generator`

C. `tuple`

D. `range`

2. Which is the memory-frugal spelling?

A. `sum([x * x for x in big])`

B. `sum(x * x for x in big)`

C. `sum(list(map(sq, big)))`

D. They are identical

3. What does `next((x for x in xs if p(x)), None)` return when nothing matches?

A. `StopIteration`

B. `None`

C. `[]`

D. `False`

4. Why does `any([f(x) for x in xs])` lose the short-circuit?

A. `any` does not short-circuit

B. The list comprehension evaluates every `f(x)` before `any` runs

C. Lists are always evaluated twice

D. It does not lose it

5. When is a generator expression created, and when does its body run?

A. Both immediately

B. The first `for` iterable is evaluated immediately; the body runs as values are pulled

C. Both when consumed

D. The body runs immediately; the iterable lazily

ANSWER KEY · GENERATOR EXPRESSIONS — LAZY COMPREHENSIONS

1. B Parentheses make a generator expression — lazy and one-shot.

2. B The generator feeds `sum` one value at a time; the list version allocates all of them first.

3. B The default replaces the exception — the 'first that matches' idiom.

4. B Only a lazy generator lets `any` stop at the first true value.

5. B That is why an error in the source surfaces at creation while later clauses are lazy.

LESSON 4 · 15 MIN

itertools — the iterator toolkit

`itertools` is a set of small, fast, lazy building blocks for iterators: infinite counters, chaining, slicing, running totals, grouping, and the combinatoric generators that interviews keep asking for. Each function takes iterables and returns an iterator, so they compose without intermediate lists, and each is written in C, so a chain of them is usually faster than the equivalent Python loop. This lesson catalogues the ones that matter, with the two that everyone misuses — `groupby`, which groups only consecutive runs, and `tee`, which buffers — called out.

Infinite iterators

```
from itertools import count, cycle, repeat, islice

list(islice(count(10, 5), 4))          # [10, 15, 20, 25] — count(start, step)
list(islice(cycle("ab"), 5))           # ['a', 'b', 'a', 'b', 'a']
list(repeat("x", 3))                   # ['x', 'x', 'x']; repeat(x) alone is i
```

`count` pairs with `zip` to number things without `enumerate`'s start limits; `cycle` alternates values; `repeat` supplies a constant to `map` (`map(pow, xs, repeat(2))`). Each is bounded by the consumer — `islice`, `zip` with a finite partner, or `takewhile`.

Slicing and terminating

```python
from itertools import islice, takewhile, dropwhile, chain, pairwise, accumulat

islice(it, 5)                          # first five
islice(it, 2, 8, 2)                    # like a slice, on any iterator
takewhile(lambda x: x < 5, xs)         # values until the first failure
dropwhile(lambda x: x < 5, xs)         # skip until the first failure, then e
chain(a, b, c)                         # one iterator over all of them
chain.from_iterable(list_of_lists)     # flatten one level, lazily
pairwise(xs)                           # (x0, x1), (x1, x2), … (3.10)
accumulate(xs)                         # running totals; accumulate(xs, max)
compress(xs, mask)                     # the xs where the mask is truthy
zip_longest(a, b, fillvalue=None)      # zip that pads instead of truncating
```

`chain.from_iterable` is the lazy flatten; `accumulate` is prefix sums (Module 3) as an iterator, and with `operator.mul` it is running products; `pairwise` replaces `zip(xs, xs[1:])` and works on any iterator.

---

groupby — consecutive runs

```python
from itertools import groupby

data = "aaabbcaa"
[(k, len(list(g))) for k, g in groupby(data)]          # [('a', 3), ('b', 2),

rows = [("eng", "ada"), ("eng", "cy"), ("ops", "bob")]
for dept, group in groupby(sorted(rows), key=lambda r: r[0]):
    print(dept, [name for _, name in group])           # eng ['ada', 'cy'] /
```

`groupby(iterable, key)` yields `(key, group)` pairs for each *run* of consecutive elements with the same key. On unsorted data it produces one group per run, not per key — the second `'a'` above is a separate group. To group whole data, sort by the same key first; to group without sorting, use `defaultdict(list)` (Module 7). Two more rules: each `group` is an iterator that is invalidated when you advance to the next pair, so consume it (`list(group)`) before moving on; and the run-length behaviour is a *feature* — run-length encoding is `groupby` in one line.

Combinatorics

```
from itertools import product, permutations, combinations, combinations_with_r

list(product("ab", [1, 2]))                 # [('a', 1), ('a', 2), ('b', 1), (
list(product(range(2), repeat=3))           # every 3-bit tuple
list(permutations("abc", 2))                # ('a','b'), ('a','c'), ('b','a'),
list(combinations("abc", 2))                # ('a','b'), ('a','c'), ('b','c')
list(combinations_with_replacement("ab", 2))   # ('a','a'), ('a','b'), ('b','b
```

`product` replaces nested loops (and gives a single `break`); `permutations(xs, r)` is the ordered arrangements, `combinations(xs, r)` the subsets of size `r` in input order. Sizes grow fast — `permutations` of 10 is 3.6 million — so filter lazily and never materialise more than you need. The subsets of every size are `chain.from_iterable(combinations(xs, r) for r in range(len(xs) + 1))`, the *powerset* recipe.

---

tee, starmap, filterfalse

```
from itertools import tee, starmap, filterfalse
import operator

a, b = tee(gen, 2)                      # two independent iterators over one s
list(starmap(operator.mul, [(2, 3), (4, 5)]))   # [6, 20] — apply f(*args) to
list(filterfalse(lambda x: x % 2, range(6)))    # [0, 2, 4] — the complement
```

`tee` is the way to look at a stream twice without a list, but it stores everything between the two readers, so a slow second reader costs the memory a list would have anyway; do not use the original iterator after tee-ing it.

Recipes worth knowing

```python
def batched(iterable, n):                # groups of n; itertools.batched exist
    it = iter(iterable)
    while batch := tuple(islice(it, n)):
        yield batch

def sliding_window(iterable, n):         # consecutive windows of n
    it = iter(iterable)
    window = deque(islice(it, n), maxlen=n)
    if len(window) == n:
        yield tuple(window)
    for x in it:
        window.append(x)
        yield tuple(window)

def unique_everseen(iterable):           # deduplicate, keeping order, lazily
    seen = set()
    for x in iterable:
        if x not in seen:
            seen.add(x)
            yield x
```

The `itertools` documentation ends with a page of recipes like these (and the third-party `more-itertools` packages them); they are the idioms to reach for before writing an index loop.

Pitfalls

- `groupby` on unsorted data expecting one group per key.
- Holding a `groupby` group after advancing to the next one.
- `list(count())` or `list(cycle(...))` — infinite.
- `zip` where `zip_longest` was needed, or the reverse.
- Reusing the source after `tee`.
- Materialising `permutations` or `product` of a large input.

Key takeaways

- `count`, `cycle`, `repeat` are infinite; bound them with `islice`, `takewhile` or a finite `zip` partner.

`chain` / `chain.from_iterable`, `islice`, `pairwise`, `accumulate`, `compress`, `zip_longest` transform streams lazily.

`groupby` groups consecutive runs — sort first for whole-data grouping, consume each group before moving on; it is run-length encoding in one line.

`product` replaces nested loops; `permutations` are ordered, `combinations` unordered; the powerset is combinations of every size.

`tee` splits with buffering, `starmap` applies to tuples, the recipes (`batched`, windows, `unique_everseen`) are the idioms.

EXERCISE 1

Runs with groupby

Read a line of text. Print its run-length encoding built with `itertools.groupby` (each run as the character followed by its length, e.g. `aaab` → `a3b1`; an empty line prints `(empty)`). Then split the line into words, sort them, and group them by first letter with `groupby` and a key function — printing `<letter>: <words>` per group.

Input: one line. **Output:** `rle: <code>`, then one line per first-letter group (none for an empty line).

```
apple avocado banana
```

prints

```
rle: a1p2l1e1 1a1v1o1c1a1d1o1 1b1a1n1a1n1a1
a: apple avocado
b: banana
```

INPUT 1

```
apple avocado banana
```

OUTPUT 1

```
rle: a1p2l1e1 1a1v1o1c1a1d1o1
1b1a1n1a1n1a1
a: apple avocado
b: banana
```

REFERENCE SOLUTION

```
from itertools import groupby

line = input().rstrip("\n")
rle = "".join(f"{ch}{sum(1 for _ in run)}" for ch, run in groupby(line))
print("rle:", rle if rle else "(empty)")
for letter, words in groupby(sorted(line.split()), key=lambda w: w[0]):
    print(f"{letter}: {' '.join(words)}")
```

EXERCISE 2

Combinatoric counts

Read a string of distinct characters and `r`. Using `itertools`, print the number of permutations of length `r` and the first three (joined), the number of combinations and the first three, and how many tuples from `product((0, 1), repeat=len(items))` contain exactly `r` ones.

Input: `items r`. **Output:** `perms <n>: <first three>`, `combs <n>: <first three>`, `bits <n>`.

```
abc 2
```

prints

```
perms 6: ab ac ba
combs 3: ab ac bc
bits 3
```

INPUT 1

```
abc 2
```

OUTPUT 1

```
perms 6: ab ac ba
combs 3: ab ac bc
bits 3
```

REFERENCE SOLUTION

```
from itertools import permutations, combinations, product, islice

items, r = input().split()
r = int(r)
perms = list(permutations(items, r))
combs = list(combinations(items, r))
print(f"perms {len(perms)}:", " ".join("".join(p) for p in islice(perms, 3)))
print(f"combs {len(combs)}:", " ".join("".join(c) for c in islice(combs, 3)))
print(f"bits {sum(1 for t in product((0, 1), repeat=len(items)) if sum(t) == r)}")
```

QUIZ

1. What does `list(groupby("aabba"))` group?

A. Two groups: all `a`s and all `b`s

B. Three groups of consecutive runs: `aa`, `bb`, `a`

C. Five single-character groups

D. It raises

2. Which bounds an infinite `itertools.count()`?

A. `list(count())`

B. `islice(count(), 10)`

C. `len(count())`

D. `sorted(count())`

3. What is `list(product("ab", repeat=2))`?

A. `[('a', 'b')]`

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

C. `[('a','b'), ('b','a')]`

D. `['aa', 'ab', 'ba', 'bb']`

4. What does `chain.from_iterable([[1, 2], [3]])` produce?

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

B. `1, 2, 3` lazily — a one-level flatten

C. `[1, 2, 3]` as a list

D. `TypeError`

5. What does `accumulate([1, 2, 3])` yield?

A. `6`

B. `1, 3, 6` — running totals

C. `1, 2, 3`

D. `0, 1, 3, 6`

ANSWER KEY · ITERTOOLS — THE ITERATOR TOOLKIT

1. B `groupby` is a run detector; sort first for whole-data grouping.

2. B `islice`, `takewhile`, `next` or a finite `zip` partner stop an infinite iterator; the others consume forever.

3. B The cartesian product with itself — every ordered pair, as tuples.

4. B It yields the elements of each inner iterable in turn without building a list.

5. B Prefix sums as an iterator; `initial=0` would prepend the zero and `func=` changes the operation.

LESSON 5 · 13 MIN

functools and operator — the function toolkit

`functools` collects the tools for working *with* functions: caching results, fixing arguments, folding sequences, preserving metadata through wrappers, dispatching on a type, and deriving comparison methods. `operator` supplies every operator as a named function so that `map`, `sorted`, `reduce` and the itertools can be handed `add` or `itemgetter(1)` instead of a lambda. This lesson covers `cache` and `lru_cache`, `partial` and `partialmethod`, `reduce`, `wraps`, `cached_property`, `singledispatch`, `total_ordering`, and the `operator` functions and factories, with the rules for when each beats the plain alternative.

Caching

```
from functools import cache, lru_cache

@cache                                  # unbounded; arguments must be hashable
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

@lru_cache(maxsize=256)                 # bounded: evicts least-recently-used entr
def lookup(key):
    return expensive(key)

fib.cache_info()                        # CacheInfo(hits=…, misses=…, maxsize=None
fib.cache_clear()
```

The function must be *pure* (same arguments, same result, no effects that matter) and the arguments hashable — a list argument raises `TypeError`; pass a tuple. `cache` is right for recursion and for a fixed set of inputs; `lru_cache(maxsize)` for an open-ended set where memory must be bounded. A method decorated with `cache` keys on `self` too, and keeps every instance alive in the cache — for per-instance memoisation, `cached_property` (below) or an explicit dict on the instance is cleaner.

---

partial

```python
from functools import partial

power = lambda base, exp: base ** exp
square = partial(power, exp=2)
int_from_hex = partial(int, base=16)
list(map(int_from_hex, ["ff", "10"]))     # [255, 16]

square.func, square.keywords              # the original and the fixed argumen
```

`partial(f, *args, **kwargs)` fixes arguments *from the left* (positionally) or by name. Compared with `lambda x: power(x, 2)`: it is introspectable, picklable, slightly faster, and binds its arguments at creation — the fix for late binding in loops (Module 4). `partialmethod` is the same for methods defined in a class body.

---

reduce

```python
from functools import reduce
import operator

reduce(operator.mul, [1, 2, 3, 4], 1)        # 24
reduce(lambda acc, x: acc * 10 + x, [1, 2, 3], 0)   # 123 — digits to a numbe
reduce(operator.or_, sets, set())            # union of many sets
reduce(dict.__or__, dicts, {})               # merge many dicts, later winnin
```

`reduce(f, iterable, initial)` folds left: `f(f(f(initial, x0), x1), x2)`. Always pass `initial` — without it an empty iterable raises `TypeError` and the first element is used as the seed. Use it when the combining function has a name (`operator.mul`, `set.union`) and no built-in fold exists; `sum`, `min`, `max`, `any`, `all`, `math.prod`, `"".join` and `itertools.accumulate` cover the rest more readably.

---

wraps

```python
from functools import wraps

def logged(fn):
    @wraps(fn)                      # copies __name__, __doc__, __module__, _
    def wrapper(*args, **kwargs):
        print(f"calling {fn.__name__}")
        return fn(*args, **kwargs)
    return wrapper
```

Without `@wraps`, every decorated function reports its name as `wrapper` in tracebacks, `help()` and test output. Module 16 covers decorators in full; the rule here is simply that a wrapper function always carries `@wraps(fn)`.

---

cached_property and total_ordering

```python
from functools import cached_property, total_ordering

class Report:
    def __init__(self, rows):
        self.rows = rows

    @cached_property                # computed once per instance, stored in _
    def total(self):
        return sum(r.amount for r in self.rows)

@total_ordering                     # __lt__ + __eq__ → all six comparisons
class Version:
    ...
```

Both appeared in Module 8; they live in `functools`.

---

singledispatch

```python
from functools import singledispatch

@singledispatch
def describe(value):                    # the default
    return f"object {value!r}"

@describe.register
def _(value: int):
    return f"int {value}"

@describe.register(list)
def _(value):
    return f"list of {len(value)}"

describe(3), describe([1, 2]), describe("s")    # 'int 3', 'list of 2', "objec
```

`singledispatch` picks an implementation by the *type of the first argument*, with subclasses matching their nearest registered base (and ABCs honoured), so `describe(True)` uses the `int` version. It is the honest form of the `isinstance` chain from Module 9 for operations that do not belong on the types themselves — formatting, serialising, converting external data — and it is extensible from outside: another module can register a new type without touching the original function. `singledispatchmethod` does the same for methods.

The operator module

```python
import operator
from operator import itemgetter, attrgetter, methodcaller

operator.add(2, 3)                          # 5;  also sub, mul, truediv, floor
operator.lt(1, 2), operator.eq(a, b)        # comparisons
operator.and_, operator.or_, operator.xor   # bitwise / set operations (traili
operator.contains(xs, 3), operator.getitem(d, "k")

sorted(rows, key=itemgetter(2))             # by column 2
sorted(rows, key=itemgetter(1, 0))          # by columns 1 then 0 — the key re
sorted(people, key=attrgetter("age", "name"))
sorted(words, key=methodcaller("lower"))
list(map(methodcaller("strip", "-"), lines))
```

`itemgetter` / `attrgetter` / `methodcaller` build key and mapping functions without lambdas, and `itemgetter(1)` is measurably faster than `lambda r: r[1]`. `operator.index`, `operator.length_hint` and the in-place variants (`iadd`, …) round out the set.

Pitfalls

`@cache` on a function with side effects or unhashable arguments.

`@cache` on a method, keeping every instance alive.

`reduce` without `initial` on a possibly-empty input.

A decorator without `@wraps`.

`singledispatch` with the type in the wrong place (it dispatches on the *first* argument only).

`operator.and` / `or` — the names have a trailing underscore.

Key takeaways

`@cache` / `@lru_cache(maxsize)` memoise pure functions with hashable arguments; `cache_info` / `cache_clear` inspect and reset.

`partial` fixes arguments at creation and is introspectable; prefer it to a lambda that only fixes arguments.

`reduce(f, it, initial)` folds left with a named function; use the built-in folds when they exist.

`@wraps` on every wrapper; `cached_property` and `total_ordering` for classes; `singledispatch` for type-based dispatch that lives outside the types.

`operator` names every operator; `itemgetter`, `attrgetter`, `methodcaller` build key functions.

EXERCISE 1

Cache statistics

Decorate a recursive `fib` with `functools.lru_cache(maxsize=None)`, read `n`, print `fib(n)` and the cache's hits and misses from `cache_info()`. Then build `double = partial(operator.mul, 2)` and print `double(fib(n))`. Finally use `reduce` with `operator.mul` and an initial value of 1 to print the product of `1..n`.

Input: `n`. **Output:** `fib <v>`, `hits <h> misses <m>`, `double <v>`, `factorial <v>`.

```
10
```

prints

```
fib 55
hits 8 misses 11
double 110
factorial 3628800
```

INPUT 1

```
10
```

OUTPUT 1

```
fib 55
hits 8 misses 11
double 110
factorial 3628800
```

REFERENCE SOLUTION

```python
import operator
from functools import lru_cache, partial, reduce


@lru_cache(maxsize=None)
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)


n = int(input())
value = fib(n)
info = fib.cache_info()
print(f"fib {value}")
print(f"hits {info.hits} misses {info.misses}")
double = partial(operator.mul, 2)
print(f"double {double(value)}")
print(f"factorial {reduce(operator.mul, range(1, n + 1), 1)}")
```

EXERCISE 2

Describe with singledispatch

Write `describe(value)` with `functools.singledispatch`: the default returns `other <type name>`; registrations for `int` (`int <v>`; note `bool` is an `int`), `str` (`str of <len>`), `list` (`list of`

`<len>`) and `dict` (`dict with keys <sorted keys>`). Read literals one per line (parse with `ast.literal_eval`) and print each description.

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

```
42
"hey"
[1, 2, 3]
{"b": 1, "a": 2}
3.5
```

prints

```
int 42
str of 3
list of 3
dict with keys a b
other float
```

INPUT 1

```
42
"hey"
[1, 2, 3]
{"b": 1, "a": 2}
3.5
```

OUTPUT 1

```
int 42
str of 3
list of 3
dict with keys a b
other float
```

REFERENCE SOLUTION

```
import ast
import sys
from functools import singledispatch


@singledispatch
def describe(value):
    return f"other {type(value).__name__}"


@describe.register
def _(value: int):
    return f"int {value}"


@describe.register
def _(value: str):
    return f"str of {len(value)}"


@describe.register
def _(value: list):
    return f"list of {len(value)}"


@describe.register
def _(value: dict):
    return f"dict with keys {' '.join(sorted(value))}"


for line in sys.stdin:
    print(describe(ast.literal_eval(line.strip())))
```

QUIZ

1. What does `@lru_cache(maxsize=128)` do that `@cache` does not?

A. Caches only integers

B. Bounds the cache, evicting the least recently used entries

C. Makes the function faster

D. Allows unhashable arguments

2. What is `partial(int, base=2)("101")`?

A. `"101"`

B. `5`

C. `101`

D. `TypeError`

3. Why pass an initial value to `reduce`?

A. It is required syntax

B. An empty iterable raises `TypeError` without it, and the fold has a defined start

C. It makes `reduce` lazy

D. To reverse the fold

4. What does `@wraps(fn)` preserve on a wrapper?

A. Its return type

B. `__name__`, `__doc__`, `__module__`, `__qualname__` and `__wrapped__`

C. Its performance

D. Its default arguments

5. On which argument does `singledispatch` dispatch?

A. All arguments

B. The type of the first argument

C. The return type

D. The keyword arguments

ANSWER KEY · FUNCTOOLS AND OPERATOR — THE FUNCTION TOOLKIT

1. B `cache` is unbounded; `lru_cache` with a size keeps memory bounded for open-ended inputs.

2. B The keyword `base=2` is fixed; the call supplies the string.

3. B Without `initial`, the first element seeds the fold and nothing seeds an empty one.

4. B Without it, decorated functions all report the wrapper's name in tracebacks and `help`.

5. B Single dispatch by definition; subclasses match their nearest registered base.

LESSON 6 · 14 MIN

Lazy pipelines — a worked log-processing example

The previous lessons gave the parts; this one assembles them into the shape that real data-processing code takes in Python: a chain of generator stages, each taking an iterable and yielding, that reads a source once and streams every record through parse, filter, transform and aggregate steps without ever holding the whole input. The worked example processes a web-server log — the kind of task that is a script in every company — and the design points are the ones that make a pipeline maintainable: one job per stage, stages that are independently testable, materialisation only at the end, and errors handled at the stage that can name them.

The source

```
import sys

def read_lines(stream):
    for line in stream:
        line = line.rstrip("\n")
        if line:
            yield line
```

The first stage adapts the input — a file, `sys.stdin`, a list of strings in a test — into a stream of clean lines. It does nothing else; in particular it does not parse. Because it yields, a file of a gigabyte is read line by line and never stored.

Parse

```
import re
from dataclasses import dataclass

LINE = re.compile(r'(?P<ip>\S+) \S+ \S+ \[(?P<ts>[^\]]+)\] "(?P<method>\w+) (

@dataclass(frozen=True)
class Hit:
    ip: str
    method: str
    path: str
    status: int
    size: int

def parse(lines):
    for n, line in enumerate(lines, start=1):
        m = LINE.match(line)
        if m is None:
            print(f"skip line {n}", file=sys.stderr)      # report, do not st
            continue
        size = m["size"]
        yield Hit(m["ip"], m["method"], m["path"], int(m["status"]), 0 if size
```

The parse stage turns text into typed records — a frozen dataclass, so downstream code cannot accidentally mutate a shared object — and decides what to do with malformed input *here*, where the line number is known. Reporting to stderr and continuing is the right default for a log processor; raising would be right for a config loader. Either way the decision is local to this stage.

Filter and transform

```
def only(hits, *, status=None, method=None):
    for hit in hits:
        if status is not None and hit.status != status:
            continue
        if method is not None and hit.method != method:
            continue
        yield hit

def without_query(hits):
    for hit in hits:
        yield hit if "?" not in hit.path else hit.__class__(hit.ip, hit.metho
```

Filters yield a subset; transforms yield a modified record (a *new* one, since the dataclass is frozen — `dataclasses.replace` is the cleaner spelling). Each is a few lines, has a name that says what it does, and can be dropped into or out of the chain. Keyword-only options keep the call sites readable.

Aggregate

```
from collections import Counter

def summarise(hits):
    by_status = Counter()
    by_path = Counter()
    bytes_total = 0
    for hit in hits:
        by_status[hit.status] += 1
        by_path[hit.path] += 1
        bytes_total += hit.size
    return by_status, by_path, bytes_total
```

The terminal stage is the one place the stream is consumed to a result. It is an ordinary function, not a generator: it pulls every record through every stage above it, and only its accumulators live in memory. Several aggregates in one pass — as here — beat several passes over a re-read file.

Assembling

```
def main():
    hits = parse(read_lines(sys.stdin))
    hits = only(hits, method="GET")
    hits = without_query(hits)
    by_status, by_path, total = summarise(hits)
    for status, n in sorted(by_status.items()):
        print(f"{status} {n}")
    for path, n in sorted(by_path.items(), key=lambda kv: (-kv[1], kv[0]))[:3]
        print(f"{n:5} {path}")
    print(f"bytes {total}")
```

Reads top to bottom as the data flows; every stage is lazy, so nothing runs until `summarise` starts pulling, and then each line goes through the whole chain before the next is read. Reassigning `hits` at each step keeps the chain flat. `main` owns I/O and output formatting; the stages know nothing of printing (the one stderr line in `parse` is a diagnostic, not output).

Testing a stage

Because each stage takes any iterable, a test feeds a list and checks the list it gets back:

```
sample = ['1.1.1.1 - - [x] "GET /a?q=1 HTTP/1.1" 200 10', 'garbage']
hits = list(parse(sample))
assert len(hits) == 1
assert list(without_query(hits))[0].path == "/a"
assert summarise(only(hits, status=404)) == (Counter(), Counter(), 0)
```

No files, no stdin, no mocking: the stage's contract is "iterable in, iterable out", and lists satisfy it. Module 18 turns these into a test suite.

Design points

- **One job per stage**, named for the job. A stage that parses *and* filters cannot be reused for the other.
- **Records over tuples** once there are more than two fields — `hit.status` reads, `hit[3]` does not.
- **Immutable records** (frozen dataclass) so that a transform must create a new one and a bug cannot corrupt an upstream stage's object.
- **Handle errors in the stage that has the context** — the parser knows the line number; the aggregator does not.
- **Materialise at the end**, or with `list()` at exactly the point where two passes are needed.
- **Keep `main` as the only stage that reads real input or prints.**

When the pipeline needs branching (one stream into two consumers), `itertools.tee` or a single pass that computes both, as `summarise` does, avoids re-reading. When it needs to run in parallel, the same stage functions map onto `concurrent.futures` or `asyncio` with no change to their contracts (Module 17).

Pitfalls

A stage that returns a list "for convenience", turning the whole chain eager.

Parsing and filtering in one stage.

Mutating records shared with an upstream stage.

Swallowing malformed lines silently, so a format change goes unnoticed.

Consuming the stream twice by accident (`len(list(hits))` then iterating `hits`).

Printing from the middle of the chain.

Key takeaways

A pipeline is a chain of generator stages — source, parse, filter, transform — ending in one aggregating consumer.

Each stage takes an iterable and yields; that contract makes stages composable and testable with plain lists.

Parse into frozen records; handle bad input where the context is; report or raise deliberately.

Nothing runs until the consumer pulls; one pass, constant memory; materialise only where a second pass is needed.

`main` reads and prints; stages compute.

EXERCISE 1

Purchase pipeline

Build a pipeline of generator stages over lines `user action amount`: `read_lines` strips and drops blank lines; `parse` yields `(user, action, int(amount))`, counting malformed lines (wrong field count or non-integer amount) in a mutable counter list instead of raising; `only_buys` keeps `action == "buy"`; `summarise` consumes the stream into a per-user total. Print each user's total in name order, then `skipped <n>`.

Input: lines. **Output:** `<user> <total>` lines, then `skipped <n>`.

```
ada buy 30
bob sell 10
ada buy 20
garbage
bob buy x
```

prints

```
ada 50
skipped 2
```

INPUT 1

```
ada buy 30
bob sell 10
ada buy 20
garbage
bob buy x
```

OUTPUT 1

```
ada 50
skipped 2
```

REFERENCE SOLUTION

```python
import sys


def read_lines(stream):
    for line in stream:
        line = line.strip()
        if line:
            yield line


def parse(lines, skipped):
    for line in lines:
        parts = line.split()
        if len(parts) != 3:
            skipped[0] += 1
            continue
        user, action, amount = parts
        try:
            yield user, action, int(amount)
        except ValueError:
            skipped[0] += 1


def only_buys(records):
    for record in records:
        if record[1] == "buy":
            yield record


def summarise(records):
    totals = {}
    for user, _, amount in records:
        totals[user] = totals.get(user, 0) + amount
    return totals


skipped = [0]
totals = summarise(only_buys(parse(read_lines(sys.stdin), skipped)))
for user in sorted(totals):
    print(user, totals[user])
print(f"skipped {skipped[0]}")
```

EXERCISE 2

Sliding windows over a stream

Implement the `sliding_window(iterable, n)` recipe with a bounded `deque` and `islice`, yielding each full window as a tuple. Read `n` and then integers **one per line** (a stream), pass the parsed stream through the generator, and print each window's mean with two decimals as it becomes available. Print `no window` if the stream never fills one.

Input: `n`, then integer lines. **Output:** one mean per full window, or `no window`.

```
3
1
2
3
4
```

prints

```
2.00
3.00
```

INPUT 1

```
3
1
2
3
4
```

OUTPUT 1

```
2.00
3.00
```

REFERENCE SOLUTION

```python
import sys
from collections import deque
from itertools import islice


def sliding_window(iterable, n):
    it = iter(iterable)
    window = deque(islice(it, n), maxlen=n)
    if len(window) == n:
        yield tuple(window)
    for x in it:
        window.append(x)
        yield tuple(window)


n = int(input())
stream = (int(line) for line in sys.stdin if line.strip())
count = 0
for window in sliding_window(stream, n):
    print(f"{sum(window) / n:.2f}")
    count += 1
if count == 0:
    print("no window")
```

QUIZ

1. What is the contract of a pipeline stage?

A. Takes a list, returns a list

B. Takes any iterable, yields values lazily

C. Takes a file, prints results

D. Takes a string, returns a string

2. When does work in a generator pipeline actually happen?

A. When each stage is created

B. When the final consumer pulls values

C. At import time

D. When `main` returns

3. Where should a malformed input line be handled?

A. In the aggregator

B. In the parsing stage, which knows the line and its number

C. In `main` after the pipeline

D. Nowhere — let it crash

4. Why parse into frozen dataclass records rather than tuples or dicts?

A. They are faster

B. Named fields read clearly, and immutability stops a downstream stage corrupting an upstream object

C. Tuples cannot be yielded

D. Dicts are not iterable

5. What breaks a lazy pipeline?

A. Using `yield`

B. A stage that returns a list, forcing everything before it to be materialised

C. Using a dataclass

D. Sorting at the end

ANSWER KEY · LAZY PIPELINES — A WORKED LOG-PROCESSING EXAMPLE

1. B That contract makes stages composable and testable with plain lists.

2. B Nothing runs until something iterates the last stage; each record then flows through every stage in turn.

3. B The stage with the context can report or skip meaningfully; downstream stages only see records.

4. B `hit.status` beats `hit[3]`, and a transform must create a new record rather than mutate a shared one.

5. B One eager stage makes the whole upstream chain run to completion before anything downstream starts.

CHECKPOINT · 25 MIN

Checkpoint — Iterators, generators and itertools

This checkpoint covers the whole module: the iteration protocol with `iter`, `next` and `StopIteration`, generators with `yield`, `yield from` and laziness, generator expressions and the short-circuit consumers, the `itertools` toolkit with `groupby`'s consecutive-run rule and the combinatoric generators, `functools` and `operator`, and the design of lazy pipelines.

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 the difference between an iterable and an iterator, and which one is `zip(a, b)`?
- What happens the first time a generator function is called, and when does its body run?
- Why is `sum(x for x in xs)` preferable to `sum([x for x in xs])`, and when is a list comprehension still right?
- What does `groupby` do on unsorted input?
- What are `product`, `permutations` and `combinations`, and how do their sizes grow?
- What does `@cache` require of a function?
- In a pipeline, which stage should handle a malformed record, and why?

The three programs are a run-length codec built on `groupby`, a subset-sum search over `combinations` that stops at the first hit, and a log pipeline of generator stages that parses, filters and aggregates in one pass.

EXERCISE 1

Run-length codec with groupby

Each line is `E <text>` or `D <code>`. Encode with `itertools.groupby` — each run becomes the character followed by its count (`aaabcc` → `a3b1c2`). Decode by scanning the code: a character followed by one or more digits (a generator that yields the expanded characters, joined at the end). Text after the command is taken literally.

Input: lines. **Output:** one line per input line (an empty result prints `(empty)`).

```
E aaabcc
D a3b1c2
```

prints

```
a3b1c2
aaabcc
```

INPUT 1

```
E aaabcc
D a3b1c2
```

OUTPUT 1

```
a3b1c2
aaabcc
```

REFERENCE SOLUTION

```python
import sys
from itertools import groupby


def encode(text):
    return "".join(f"{ch}{sum(1 for _ in run)}" for ch, run in groupby(text))


def expand(code):
    i = 0
    while i < len(code):
        ch = code[i]
        j = i + 1
        while j < len(code) and code[j].isdigit():
            j += 1
        yield ch * int(code[i + 1:j])
        i = j


for line in sys.stdin:
    cmd, _, rest = line.rstrip("\n").partition(" ")
    result = encode(rest) if cmd == "E" else "".join(expand(rest))
    print(result if result else "(empty)")
```

EXERCISE 2

Subset sum, first hit

Read a target and a line of positive integers. Search the combinations of indexes by increasing size (1, 2, …, all) with `itertools.combinations`, in the order `combinations` produces them, and stop at

the **first** combination whose values sum to the target — using `next` over a generator expression chained across sizes with `itertools.chain.from_iterable`. Print the chosen values or `none`.

Input: the target, then a line of integers. **Output:** `found <values>` or `none`.

```
9
2 7 5 4
```

prints

```
found 2 7
```

INPUT 1

```
9
2 7 5 4
```

OUTPUT 1

```
found 2 7
```

REFERENCE SOLUTION

```python
from itertools import combinations, chain

target = int(input())
xs = list(map(int, input().split()))
candidates = chain.from_iterable(combinations(xs, r) for r in range(1, len(xs) + 1))
hit = next((c for c in candidates if sum(c) == target), None)
print("found", " ".join(map(str, hit))) if hit else print("none")
```

EXERCISE 3

Log pipeline

Process log lines `ip method path status size` through generator stages: `read_lines` (strip, drop blank), `parse` (yield a tuple with `status` and `size` as ints; skip lines with the wrong field count or non-integer fields), `only(method="GET")`, and a single-pass `summarise` that returns status counts, path counts and total bytes. Print `status <code>: <n>` in ascending order, `bytes <total>`, and `top <path>` (most requests, ties alphabetical) or `top none`.

Input: lines. **Output:** the summary.

```
1.1.1.1 GET /a 200 100
1.1.1.2 POST /a 200 50
1.1.1.1 GET /b 404 0
bad line
```

prints

```
status 200: 1
status 404: 1
bytes 100
top /a
```

INPUT 1

```
1.1.1.1 GET /a 200 100
1.1.1.2 POST /a 200 50
1.1.1.1 GET /b 404 0
bad line
```

OUTPUT 1

```
status 200: 1
status 404: 1
bytes 100
top /a
```

REFERENCE SOLUTION

```
import sys
from collections import Counter


def read_lines(stream):
    for line in stream:
        line = line.strip()
        if line:
            yield line


def parse(lines):
    for line in lines:
        parts = line.split()
        if len(parts) != 5:
            continue
        ip, method, path, status, size = parts
        try:
            yield ip, method, path, int(status), int(size)
        except ValueError:
            continue


def only(records, *, method):
    for record in records:
        if record[1] == method:
            yield record


def summarise(records):
    by_status, by_path, total = Counter(), Counter(), 0
    for _, _, path, status, size in records:
        by_status[status] += 1
        by_path[path] += 1
        total += size
    return by_status, by_path, total


by_status, by_path, total = summarise(only(parse(read_lines(sys.stdin)), method="GET
for status in sorted(by_status):
    print(f"status {status}: {by_status[status]}")
print(f"bytes {total}")
print("top", min(by_path, key=lambda p: (-by_path[p], p)) if by_path else "none")
```

CHECKPOINT QUESTIONS

1. Which object is an iterator?

A. `[1, 2, 3]`

B. `iter([1, 2, 3])`

C. `"abc"`

D. `range(3)`

2. What does `next(it)` do on an exhausted iterator?

A. Returns `None`

B. Raises `StopIteration`

C. Restarts

D. Returns the last value

3. What does this print?

```
def g():
    print("start")
    yield 1

x = g()
print("made")
next(x)
```

A. `start` then `made`

B. `made` then `start`

C. `start` only

D. `made` only

4. What is `sum(x for x in range(4))`?

A. `6`

B. `SyntaxError`

C. `[0, 1, 2, 3]`

D. `10`

5. Which `itertools` call yields `(1, 2), (2, 3), (3, 4)` from `[1, 2, 3, 4]`?

A. `combinations(xs, 2)`

B. `pairwise(xs)`

C. `product(xs, xs)`

D. `zip(xs)`

6. How many elements does `permutations("abcd", 2)` produce?

A. 6

B. 12

C. 16

D. 24

7. Why sort before `groupby` to group by a key?

A. `groupby` requires sorted input or raises

B. `groupby` groups consecutive runs only, so equal keys must be adjacent

C. Sorting is faster

D. It does not matter

8. What does `functools.cache` require of arguments?

A. They must be ints

B. They must be hashable

C. They must be positional

D. Nothing

9. What is `reduce(operator.add, [], 0)`?

A. `TypeError`

B. `0`

C. `None`

D. `[]`

10. What does `itemgetter(1, 0)` return when called on `("a", "b")`?

A. `"b"`

B. `("b", "a")`

C. `["b", "a"]`

D. `"a"`

11. In a pipeline, which stage is *not* a generator?

A. The parser

B. The filter

C. The final aggregator, which consumes the stream and returns a result

D. The source

12. What does `tee(it, 2)` cost?

A. Nothing — it duplicates the iterator

B. Buffering of every value one copy has consumed and the other has not

C. A full copy of the source list

D. It re-runs the source

ANSWER KEY · CHECKPOINT — ITERATORS, GENERATORS AND ITERTOOLS

1. B The others are iterables that produce iterators on demand.

2. B Pass a default as the second argument to avoid the exception.

3. B The body does not run until `next`; creating the generator prints nothing.

4. A A generator expression as the sole argument needs no extra parentheses; 0 + 1 + 2 + 3 = 6.

5. B `pairwise` gives consecutive pairs; `combinations` would include `(1, 3)` and `(1, 4)`.

6. B Ordered pairs without repetition: 4 × 3 = 12. `combinations` would give 6.

7. B Unsorted input gives one group per run, not per key.

8. B The cache is a dict keyed by the arguments; a list argument raises `TypeError`.

9. B With an initial value, an empty fold returns it; without one it raises.

10. B With several indexes, `itemgetter` returns a tuple — the multi-key sort key.

11. C Every stage yields except the terminal consumer, which pulls everything through.

12. B Two readers advancing at different rates keep the difference in memory; do not use the original after tee-ing.