YZM1022

Advanced Programming

Week 10: Advanced Functional Programming

Instructor: Ekrem Çetinkaya

Date: 29.04.2026

Today's Agenda

We know the tools, we know the basics of functional programming. Today we learn the **non-obvious patterns** that make professionals reach for them.

The questions we'll answer

- What happens inside `for x in obj` at the protocol level?
- How do generators *receive* values, not just produce them?
- How do you branch an iterator without consuming it twice?
- When does `reduce` beat a `for` loop - and when doesn't it?

What we'll build

A complete **lazy data pipeline** - read, parse, filter, group, and aggregate a dataset with no mutable accumulators, no intermediate lists, and no nested loops.

Each section adds one tool. The final section assembles them.

Recap - The Four Properties of Functional Code

Functional programming is not a framework or a library; it is a **discipline** applied to any language. Python supports all four properties.

Pure functions
Same inputs -> same outputs. No hidden state, no side effects. Call it twice, get the same result.

Immutability
Don't modify existing data -> produce new values. Eliminates a whole class of bugs caused by shared mutable state.

Composability
Small, focused functions snap together into pipelines. Each function does one thing; the chain is the program.

Laziness
Compute only what is needed, only when it is needed. Generators, `itertools`, and coroutines all express this property.

Generators Are a Functional Tool

A generator is a **lazy sequence**. It produces one value per `next()` call and nothing is computed or stored ahead of time. That is functional programming's **laziness** property in action.

```
# Eager: all 1M squares built in memory before any filtering begins
big_list = [x**2 for x in range(10**6)]
filtered = [s for s in big_list if s % 3 == 0]  # second pass, full copy

# Lazy: zero allocation. Values flow through one at a time.
pipeline = (s for s in (x**2 for x in range(10**6)) if s % 3 == 0)
next(pipeline)   # 0  - only computes as far as the first match
```

`itertools` extends this: `filter`, `map`, `compress`, `groupby`, `starmap` are all lazy. Chain them and the entire pipeline runs in a **single pass** over the data - one element at a time, regardless of input size. That is what makes these tools functional.

Advanced Generator Patterns

The Iterator Protocol

Every `for` loop in Python calls two methods. Here is what Python actually does when you write: for x in obj:

```
iterator = iter(obj)          # calls obj.__iter__()
while True:
    try:
        x = next(iterator)    # calls iterator.__next__()
        # loop body
    except StopIteration:
        break                 # loop ends cleanly
```

Key rules of the iterator protocol:

- `__next__()` returns the next value or raises `StopIteration`
- Iterators are **one-way** - no rewind, no previous, no copy
- Calling `next()` after `StopIteration` keeps raising `StopIteration`
- An iterator is also iterable: `__iter__` returns `self`

```
L = [1, 2]
it = iter(L)
print(next(it))  # 1
print(next(it))  # 2
print(next(it))  # StopIteration!
```

`yield from` - Sub-Generator Delegation

`yield from iterable` forwards every value the sub-generator produces, and does three things a manual `for` loop cannot: propagates `.send()` into the sub-generator, propagates `.throw()`, and makes the sub-generator's `StopIteration.value` the result of the whole expression.

```
# Without yield from — manual forwarding (misses send/throw propagation)
def flatten_manual(nested):
    for sublist in nested:
        for item in sublist:
            yield item

# With yield from — correct, cleaner, and implemented in C
def flatten(nested):
    for sublist in nested:
        yield from sublist

print(list(flatten([[1, 2], [3, 4], [5]])))  # [1, 2, 3, 4, 5]
```

`yield from` - Recursive Delegation

`yield from` composes naturally with recursion - each level delegates to the next, and values bubble all the way up to the top-level caller without any manual forwarding code.

```
class Node:
    def __init__(self, label, left=None, right=None):
        self.label, self.left, self.right = label, left, right

def inorder(node):
    """In-order traversal: left -> root -> right."""
    if node:
        yield from inorder(node.left)    # delegate left subtree
        yield node.label                 # emit this node
        yield from inorder(node.right)   # delegate right subtree

tree = Node(4, Node(2, Node(1), Node(3)), Node(6, Node(5), Node(7)))
print(list(inorder(tree)))  # [1, 2, 3, 4, 5, 6, 7]
```

Without `yield from` you'd need to loop over `inorder(node.left)` and re-yield each element - one extra line per level, and send/throw wouldn't propagate.

`itertools.tee` - Branching One Iterator into Many

You can't rewind an iterator but `tee` lets you branch it into N independent copies.

```
import itertools

def source_gen():
    """Simulates a one-shot stream (socket, file, API response…)."""
    yield from range(5)

a, b = itertools.tee(source_gen(), 2)  # branch into two independent iterators

print(list(a))  # [0, 1, 2, 3, 4]
print(list(b))  # [0, 1, 2, 3, 4]   - no re-generation
```

The memory cost warning:

```
# If one iterator races far ahead of the other,
# tee must buffer ALL unread values in memory.
a, b = itertools.tee(range(10**6))
next(a)  # advance a by 1
list(b)  # forces tee to buffer 999,999 values!
```

> **Rule:** use `tee` only when both consumers advance at roughly the same pace. If one will consume the whole stream before the other starts, use `data = list(source)` instead, simpler and no hidden buffering.

Two-Way Generators: `send()`

A generator paused at `yield` is a suspended stack frame. `send(value)` resumes that frame and makes `value` the result of the `yield` expression inside the generator. This is fundamentally different from calling a function as no new frame is created, and all local variables persist from where execution stopped.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)     # yield i OUT, receive val IN
        if val is not None:
            i = val         # caller injected a new value
        else:
            i += 1

it = counter(10)
print(next(it))    # 0  - must call next() first to reach the first yield
print(next(it))    # 1
print(it.send(8))  # 8  - jump the counter to 8
print(next(it))    # 9
```

Two rules for `send()`:

1. Always call `next(gen)` (or `gen.send(None)`) once before sending a real value - the generator must be paused at a `yield` first
2. Always wrap the `yield` in parentheses when using its return value: `val = (yield x)`

`throw()` - Exception Injection

`throw(ExcType, value)` raises an exception inside a suspended generator at the point where it is paused. The generator can `except` it like any other exception and continue running.

```
def managed_producer():
    i = 0
    while True:
        try:
            val = (yield i)
            i = val if val is not None else i + 1
        except ValueError as e:
            print(f"  caught ValueError: {e}, resetting to 0")
            i = 0

gen = managed_producer()
print(next(gen))            # 0
print(next(gen))            # 1
gen.throw(ValueError, "bad input")
# caught ValueError: bad input, resetting to 0
print(next(gen))            # 0  (counter reset)
```

`close()` - Generator Cleanup

`close()` sends `GeneratorExit` into the suspended generator. Code in a `finally` block runs unconditionally - whether the generator finishes normally, is `close()`-d, or is garbage-collected.

```
def managed_producer():
    try:
        i = 0
        while True:
            val = (yield i)
            i = val if val is not None else i + 1
    finally:
        print("  generator cleaned up")   # runs on close() or GC

gen = managed_producer()
print(next(gen))   # 0
gen.close()        # -> generator cleaned up
```

> **Why `finally`, not `except GeneratorExit`?** `GeneratorExit` inherits from `BaseException`, not `Exception`. Catching and suppressing it raises a `RuntimeError`. Always use `finally` for unconditional cleanup.

Coroutine Pattern: Running Accumulator

A coroutine is a two-way channel: push values in via `send()`, pull results back via the `yield` expression. Unlike a generator (one-way producer), a coroutine can both receive and emit on every cycle.

```
def running_total():
    """Coroutine: send() a number, receive the running sum."""
    total = 0
    while True:
        value = (yield total)   # yield current total OUT, receive next value IN
        if value is None:
            break
        total += value

acc = running_total()
next(acc)             # prime: advance to the first yield

print(acc.send(10))   # 10
print(acc.send(20))   # 30
print(acc.send(5))    # 35
acc.close()
```

The `next(acc)` priming step is the main ergonomic pain point - callers must remember it.

The `@coroutine` Priming Decorator

A small decorator hides the priming step so callers never need to call `next()` manually.

```
from functools import wraps

def coroutine(func):
    """Automatically prime a coroutine on creation."""
    @wraps(func)
    def start(*args, **kwargs):
        gen = func(*args, **kwargs)
        next(gen)   # advance to first yield automatically
        return gen
    return start

# Now callers just create and use - no next() needed
@coroutine
def running_total():
    total = 0
    while True:
        value = (yield total)
        if value is None: break
        total += value

acc = running_total()        # primed automatically
print(acc.send(10))   # 10
print(acc.send(20))   # 30
```

Practice - Coroutine: Running Average

Write a coroutine `running_average()` that:

- Is primed automatically (use the `@coroutine` decorator above)
- Accepts numbers via `send()`
- Yields the running average after each received value

```
avg = running_average()

print(avg.send(10))   # 10.0      (only one value so far)
print(avg.send(20))   # 15.0      (10+20)/2
print(avg.send(30))   # 20.0      (10+20+30)/3
print(avg.send(40))   # 25.0      (10+20+30+40)/4
```

Solution

The coroutine yields `average` (initially `None`) at the first `yield`, which `@coroutine` discards during priming. Each `send(value)` resumes execution: `value` is bound, `total` and `count` update, and the new average is yielded back to the caller.

```
@coroutine
def running_average():
    total = 0.0
    count = 0
    average = None
    while True:
        value = (yield average)
        if value is None:
            break
        total += value
        count += 1
        average = total / count

avg = running_average()
print(avg.send(10))   # 10.0
print(avg.send(20))   # 15.0
print(avg.send(30))   # 20.0
print(avg.send(40))   # 25.0
```

From Generators to `itertools`

Generators give you the **mechanism** for lazy evaluation. `itertools` gives you the **vocabulary** - a set of composable, lazy building blocks that express common patterns without writing them from scratch.

Generators

- You define the *shape* of the sequence
- `yield` controls when each value is produced
- Full control, but also full responsibility: you write the loop, the state, the logic

itertools

- Pre-built lazy operations: filter, merge, group, combine
- Each one does exactly one thing, in C, with no overhead
- **Composable**: the output of one is the input of the next

The principle stays the same: **data flows element by element, never all at once.**

itertools

`compress` - Mask-Based Filtering

`compress(data, selectors)` keeps elements from `data` where the corresponding selector is truthy - Python's lazy equivalent of NumPy/pandas boolean indexing (`df[mask]`). No intermediate list, works on any iterable.

```
from itertools import compress

columns = ["name", "dept", "salary", "age", "email"]
data    = ["Alice", "Engineering", 95000, 32, "alice@co.com"]

# Keep name, dept, salary — drop age and email
mask = [True, True, True, False, False]

selected = list(compress(data, mask))
print(selected)   # ['Alice', 'Engineering', 95000]
```

The selector list can be shorter than `data` - `compress` stops at whichever runs out first.

`compress` - Config-Driven Column Selection

Build the mask from a set of wanted column names; apply the same mask to every row. This pattern adapts to schema changes - add a column to `WANTED` and the mask updates automatically.

```
from itertools import compress

WANTED  = {"name", "dept", "salary"}
headers = ["name", "dept", "salary", "age", "email"]

mask = [col in WANTED for col in headers]  # [True, True, True, False, False]

rows = [
    ["Alice", "Eng",  95000, 32, "alice@co.com"],
    ["Bob",   "HR",   72000, 45, "bob@co.com"],
]
for row in rows:
    print(list(compress(row, mask)))
# ['Alice', 'Eng', 95000]
# ['Bob', 'HR', 72000]
```

`filterfalse` - The Complement of `filter`

`filterfalse(pred, iter)` keeps elements where the predicate returns **False** - the exact opposite of `filter`. Use both together when you need to view a stream from two angles simultaneously.

```
from itertools import filterfalse

numbers = range(10)

evens = list(filter    (lambda x: x % 2 == 0, numbers))  # [0, 2, 4, 6, 8]
odds  = list(filterfalse(lambda x: x % 2 == 0, numbers)) # [1, 3, 5, 7, 9]
```

`filter` scans the iterable once and discards non-matching elements. `filterfalse` does the same but with the opposite outcome - there is no way to recover the discarded half from either one alone.

`filterfalse` - The `partition` Pattern

`tee` + `filter` + `filterfalse` splits one stream into two at zero extra scan cost. This recipe appears in the Python docs and is the idiomatic way to split a stream without consuming it twice.

```
from itertools import tee, filterfalse

def partition(pred, iterable):
    """Return (pass, fail) - elements where pred is True, then False."""
    t1, t2 = tee(iterable)
    return filter(pred, t1), filterfalse(pred, t2)

records = [
    {"name": "Alice", "salary": 95000},
    {"name": "Bob",   "salary": 45000},
    {"name": "Carol", "salary": 110000},
    {"name": "Dave",  "salary": 38000},
]

senior, junior = partition(lambda r: r["salary"] >= 80000, records)
print([r["name"] for r in senior])  # ['Alice', 'Carol']
print([r["name"] for r in junior])  # ['Bob', 'Dave']
```

`takewhile` and `dropwhile` - Stream Sectioning

These two tools split a stream at the first point where a predicate changes state. Unlike `filter`, `takewhile` stops **permanently** at the first failure - it never resumes, even if later elements would pass.

```
from itertools import takewhile, dropwhile

data = [1, 3, 5, 2, 7, 9]   # odd…odd…odd…EVEN (flip!)

# takewhile: keep elements UNTIL predicate fails (then stop forever)
odds_prefix = list(takewhile(lambda x: x % 2 != 0, data))
print(odds_prefix)   # [1, 3, 5]   - stops at 2, never looks back

# dropwhile: skip elements UNTIL predicate fails, then yield the rest
after_even = list(dropwhile(lambda x: x % 2 != 0, data))
print(after_even)    # [2, 7, 9]   - starts at 2
```

This makes them unsuitable for general filtering but perfect for **prefix-structured** data: comment headers, leading whitespace, sorted monotone runs.

`takewhile` / `dropwhile` - Parsing File Headers

Split a text file into its comment header and its data section in a single lazy pass - no line-number tracking, no state variable.

```
from itertools import takewhile, dropwhile

lines = [
    "# config file",
    "# generated 2026-04-29",
    "# ----------------",
    "Alice,Engineering,95000",
    "Bob,HR,72000",
]

is_comment = lambda line: line.startswith("#")

header = list(takewhile(is_comment, iter(lines)))
data   = list(dropwhile(is_comment, iter(lines)))

print(header)
# ['# config file', '# generated 2026-04-29', '# ----------------']
print(data)
# ['Alice,Engineering,95000', 'Bob,HR,72000']
```

`starmap` - Multi-Argument Map

`starmap(f, it)` is shorthand for `(f(*args) for args in it)`. Prefer it when `f` is already named and writing tuple-unpacking in a comprehension would add noise.

```
from itertools import starmap

pairs = [(2, 3), (4, 5), (6, 7)]

# map can't call pow(a, b) directly — it passes one argument at a time
results = list(starmap(pow, pairs))
print(results)   # [8, 125, 279936]   (2³, 4⁵, 6⁷)

# Equivalent comprehension — more verbose but explicit:
results = [pow(a, b) for a, b in pairs]
```

Prefer `starmap` when the function is already defined and named. Prefer the comprehension when the unpacking itself makes the logic clearer.

`starmap` + `product` - Parameter Grid Evaluation

`product` generates all combinations of parameters; `starmap` evaluates a function at each point. Together they replace nested `for` loops with a declarative, lazy expression.

```
from itertools import product, starmap

learning_rates = [0.01, 0.001]
batch_sizes    = [32, 64, 128]

def train_score(lr, batch):
    return round(1 - lr * (batch / 1000), 4)

combos = list(product(learning_rates, batch_sizes))  # 6 combinations
scores = list(starmap(train_score, combos))

for (lr, bs), score in zip(combos, scores):
    print(f"lr={lr}  batch={bs}  -> {score}")
# lr=0.01  batch=32   -> 0.9997
# lr=0.01  batch=64   -> 0.9994
# lr=0.001 batch=32   -> 0.9999
```

`accumulate` - Running Results

`accumulate(iter, func)` is like `reduce` but yields **every intermediate result**, not just the final one. The default operation is addition; pass any two-argument function to change it.

```
from itertools import accumulate
import operator

values = [1, 2, 3, 4, 5]

# Running sum (default)
print(list(accumulate(values)))
# [1, 3, 6, 10, 15]

# Running product — yields all factorials up to n
print(list(accumulate(values, operator.mul)))
# [1, 2, 6, 24, 120]

# High-water mark — running max at each position
data = [3, 1, 4, 1, 5, 9, 2, 6]
print(list(accumulate(data, max)))
# [3, 3, 4, 4, 5, 9, 9, 9]
```

`accumulate` - Prefix Sums

A prefix-sum array enables O(1) range-sum queries over any slice - a classic algorithmic trick powered by `accumulate`.

```
from itertools import accumulate

prices = [10, 20, 5, 30, 15]

# Python 3.8+: initial=0 prepends the identity element automatically
prefix = list(accumulate(prices, initial=0))
# [0, 10, 30, 35, 65, 80]

# Sum of prices[i:j] in O(1) — no loop, no slice copy
def range_sum(i, j): return prefix[j] - prefix[i]

print(range_sum(1, 4))   # 55  ->  prices[1]+prices[2]+prices[3] = 20+5+30
print(range_sum(0, 5))   # 80  ->  total of all prices
```

Without `initial=0` you'd need `[0] + list(accumulate(prices))` - `initial` is the cleaner alternative and was added precisely for this pattern.

`groupby`

`groupby(iter, key)` groups **consecutive** elements with the same key. The key rule is **the input must be sorted by the key first**, otherwise the same key appears in multiple groups.

```
from itertools import groupby
from operator import itemgetter

records = [
    {"name": "Alice", "dept": "Engineering", "salary": 95000},
    {"name": "Bob",   "dept": "HR",          "salary": 72000},
    {"name": "Eve",   "dept": "Sales",       "salary": 58000},
]
# Must sort first — groupby only groups consecutive equal keys
records.sort(key=itemgetter("dept"))

for dept, group in groupby(records, key=itemgetter("dept")):
    members  = list(group)   # consume NOW — the sub-iterator shares the stream
    salaries = [m["salary"] for m in members]
    print(f"{dept}: count={len(members)}, avg=${sum(salaries)/len(salaries):,.0f}")

# Engineering: count=1, avg=$95,000
# HR: count=1, avg=$72,000
# Sales: count=1, avg=$58,000
```

Consume `group` immediately with `list(group)`. If you advance to the next key first, the current group is silently exhausted.

For unsorted/streaming data: use `collections.defaultdict(list)` instead.

`product` + `starmap` Together

`product` generates the cartesian product of multiple iterables; `starmap` maps a function over each tuple. Together they express 'evaluate f over all parameter combinations' declaratively - the equivalent of nested `for` loops but composable and lazy.

```
from itertools import product, starmap

models   = ["linear", "tree"]
datasets = ["train", "test"]
lambdas  = [0.1, 1.0]

def experiment(model, dataset, lam):
    return f"{model}/{dataset}/λ={lam}"

results = list(starmap(experiment, product(models, datasets, lambdas)))
for r in results:
    print(r)
# linear/train/λ=0.1
# linear/train/λ=1.0
# linear/test/λ=0.1
# linear/test/λ=1.0
# tree/train/λ=0.1
# tree/train/λ=1.0
# tree/test/λ=0.1
# tree/test/λ=1.0
```

itertools Quick Reference

Infinite iterators

Tool	What it does
`count(n, step)`	`n, n+step, n+2*step, ...`
`cycle(it)`	loops `it` forever
`repeat(x, n)`	`x` exactly `n` times (or forever)

Selecting & slicing

Tool	What it does
`islice(it, stop)`	first N elements
`takewhile(pred, it)`	elements until pred fails
`dropwhile(pred, it)`	elements after pred fails
`compress(data, sel)`	mask-based filter
`filterfalse(pred, it)`	complement of filter

itertools Quick Reference

Combining & applying

Tool	What it does
`chain(*its)`	concatenate iterables
`tee(it, n)`	branch into N copies
`starmap(f, it)`	`f(*args)` for each tuple
`accumulate(it, f)`	running results
`groupby(it, key)`	group consecutive equal keys
`product(*its)`	cartesian product

From `itertools` to `operator`

`itertools` gave us lazy operations over sequences. But many of those operations take a **function as an argument** - a sort key, a predicate, a combining function. If that function is a one-liner lambda, writing it every time adds noise without adding meaning.

```
# You have been writing this:
sorted(records, key=lambda r: r["salary"])
reduce(lambda a, b: a * b, values, 1)
map(lambda x: x.lower(), words)

# The operator module lets you write this instead:
sorted(records, key=itemgetter("salary"))
reduce(operator.mul, values, 1)
map(methodcaller("lower"), words)
```

`operator` is functional programming's answer to **naming the obvious**: every built-in Python operation is already a first-class function, you just have to import it.

The `operator` Module

Why `operator` Exists

Every time you write `lambda a, b: a + b` or `lambda x: x["key"]`, you're naming something that already exists.

The `operator` module exposes Python's built-in operators as **first-class functions**.

```
import operator

# These pairs are equivalent:
lambda a, b: a + b      ↔   operator.add
lambda a, b: a * b      ↔   operator.mul
lambda a, b: a < b      ↔   operator.lt
lambda x: x[key]        ↔   operator.itemgetter(key)
lambda x: x.attr        ↔   operator.attrgetter("attr")
lambda x: x.method(arg) ↔   operator.methodcaller("method", arg)
```

```
from functools import reduce
import operator

# Self-documenting: "multiply 1 through 5"
factorial = reduce(operator.mul, range(1, 6), 1)   # 120
# vs: reduce(lambda a, b: a * b, range(1, 6), 1)  # same result, but why name it?
```

The `operator` version **signals intent** (`mul` = multiplication) and is **faster** as CPython calls it directly in C rather than dispatching through a lambda.

`itemgetter` - Accessing by Key

`itemgetter(key)` returns a callable that extracts `obj[key]` from any subscriptable object. Unlike a lambda, it is a named, reusable, C-implemented function - faster and more readable in sort keys.

```
from operator import itemgetter

records = [
    {"name": "Carol", "dept": "Engineering", "salary": 110000},
    {"name": "Alice", "dept": "Engineering", "salary": 95000},
    {"name": "Bob",   "dept": "HR",          "salary": 72000},
]

# Dictionary / sequence key access
by_salary = sorted(records, key=itemgetter("salary"))
top       = max(records, key=itemgetter("salary"))
names     = list(map(itemgetter("name"), records))

print(names)        # ['Carol', 'Alice', 'Bob']
print(top["name"])  # Carol
```

`attrgetter` and `methodcaller`

`attrgetter` does the same for object attributes (works with dataclasses, namedtuples, any object). `methodcaller` creates a callable that calls a named method - useful when you need `map(methodcaller('strip'), lines)` instead of `map(str.strip, lines)` (which requires an unbound method).

```
from operator import attrgetter, methodcaller
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int

employees = [Employee("Carol", "Eng", 110000), Employee("Alice", "Eng", 95000)]
by_name = sorted(employees, key=attrgetter("name"))
print([e.name for e in by_name])  # ['Alice', 'Carol']

# methodcaller — call a method on each element
words = ["hello", "WORLD", "Python"]
lowered = list(map(methodcaller("lower"), words))
print(lowered)  # ['hello', 'world', 'python']
```

`operator` with `reduce` and `accumulate`

`operator` functions are the natural glue for `reduce` / `accumulate` - they express *what* the reduction is (`mul` , `and_`) rather than *how* to do it (`lambda a, b: a * b`).

```
from functools import reduce
from itertools import accumulate
import operator

# Factorial: reduce over multiplication
def factorial(n):
    return reduce(operator.mul, range(1, n + 1), 1)

print(factorial(5))   # 120
print(factorial(0))   # 1  - initial handles the empty case

# All factorials 1..7 in one pass with accumulate
print(list(accumulate(range(1, 8), operator.mul)))
# [1, 2, 6, 24, 120, 720, 5040]

# Logical AND / OR across a boolean list
flags = [True, True, True, False]
print(reduce(operator.and_, flags))   # False
print(reduce(operator.or_,  flags))   # True
# Note: all(flags) and any(flags) are idiomatic for these specific cases
```

When Lambda Is Fine

The Python docs offer a practical guide for deciding when to convert a lambda to a named function.

> 1. Write a lambda function.
> 2. Write a comment explaining what the lambda does.
> 3. Study the comment for a while, and think of a name that captures the essence.
> 4. Convert the lambda to a `def` statement using that name.
> 5. Remove the comment.

```
# Step 1: lambda
total = reduce(lambda a, b: (0, a[1] + b[1]), items)[1]

# Step 2: comment
# sum the second elements of all (x, y) pairs, ignoring x

# Step 3: name it
def sum_second(a, b): return (0, a[1] + b[1])

# Step 4 + 5: convert and remove comment
total = reduce(sum_second, items)[1]

# Or even simpler: don't use reduce at all
total = sum(b for _, b in items)   # ← this is actually the best version
```

From `operator` to `functools`

`operator` solved the problem of *what* the function does. `functools` solves the problem of *how* functions are used - how they are **adapted**, **composed**, and **remembered**.

What `functools` adds

- `partial` - freeze some arguments of a function, produce a new one
- `reduce` - collapse a sequence to a single value with a binary function
- `total_ordering` - derive six comparison operators from two
- `lru_cache` - memoize a pure function: same inputs -> cached output

The common thread

All four tools treat functions as **values** - things you transform, store, wrap, and pass around.

That is the deepest property of functional programming: **functions are first-class citizens**, not special syntax.

functools

`reduce` - Edge Cases and When to Use It

`reduce` collapses a sequence to a single value. Always supply `initial` as it is the result for an empty sequence and the identity element for the operation (`0` for `+`, `1` for `*`, `True` for `and_`).

```
from functools import reduce
import operator

print(reduce(operator.add, [1, 2, 3, 4]))   # 10

# Without initial, empty sequence raises TypeError
print(reduce(operator.add, [], 0))   # 0  ← safe
print(reduce(operator.mul, [], 1))   # 1  ← safe

# Where reduce genuinely wins: building nested structures
path = ["users", "alice", "report.pdf"]
nested = reduce(lambda inner, k: {k: inner}, reversed(path), None)
print(nested)
# {'users': {'alice': {'report.pdf': None}}}
```

When to use a `for` loop instead: any time you need a comment to explain what `reduce` is doing just write the loop.

`partial` in Pipelines

`filter`, `map`, and `sorted` accept single-argument callables. When your predicate or transform naturally takes two arguments, `partial` freezes the invariant one and produces a unary function that fits the slot.

```
from functools import partial
import operator

def above(threshold, record):
    return record["salary"] > threshold

# Without partial — lambda needed to adapt the signature
seniors = list(filter(lambda r: above(80000, r), records))

# With partial — named, reusable, no lambda
is_senior = partial(above, 80000)
seniors   = list(filter(is_senior, records))

# Stack partials to build element-wise transforms
double = partial(operator.mul, 2)
triple = partial(operator.mul, 3)

print(list(map(double, [1, 2, 3])))   # [2, 4, 6]
print(list(map(triple, [1, 2, 3])))   # [3, 6, 9]
```

`total_ordering` - All Six Comparisons from Two

Define `__eq__` and one of the four ordering methods; `total_ordering` generates the other five.

```
from functools import total_ordering

@total_ordering
class Version:
    def __init__(self, major, minor, patch):
        self.major, self.minor, self.patch = major, minor, patch

    def _tuple(self):
        return (self.major, self.minor, self.patch)

    def __eq__(self, other):
        return self._tuple() == other._tuple()

    def __lt__(self, other):        # only this one + __eq__ required
        return self._tuple() < other._tuple()

    def __repr__(self): return f"v{self.major}.{self.minor}.{self.patch}"

versions = [Version(1, 2, 0), Version(2, 0, 1), Version(1, 10, 3), Version(1, 2, 1)]
print(sorted(versions))   # [v1.2.0, v1.2.1, v1.10.3, v2.0.1]
print(max(versions))      # v2.0.1
```

`@total_ordering` derived `>`, `<=`, `>=` from `__lt__` + `__eq__` - all six operators work without writing them manually.

`lru_cache` Internals

`lru_cache` memoizes with a Least-Recently-Used eviction policy.

```
from functools import lru_cache

@lru_cache(maxsize=128)
def fib(n):
    if n < 2: return n
    return fib(n - 1) + fib(n - 2)

print(fib(50))                # 12586269025 - instant
print(fib.cache_info())
# CacheInfo(hits=48, misses=51, maxsize=128, currsize=51)

fib.cache_clear()
print(fib.cache_info())
# CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)
```

Key points:

- Arguments must be **hashable** - no lists or dicts
- `maxsize=None` for unbounded; bounded domains only
- Practical sizing: `maxsize=128` suits most cases

Putting It All Together

The Scenario

We have a CSV file of employees:

```
name,dept,salary,level
Alice,Engineering,95000,senior
Bob,HR,45000,junior
Carol,Engineering,110000,senior
Dave,HR,72000,senior
Eve,Sales,58000,junior
Frank,Engineering,88000,senior
Grace,Sales,95000,senior
Henry,HR,38000,junior
```

We want to:

1. Read lazily, don't load the whole file into memory
2. Skip the header and any malformed rows
3. Keep only `senior` employees
4. Group by department
5. For each department: count, average salary, and top earner

The Imperative Version

```
import csv

results = {}
with open("employees.csv") as f:
    reader = csv.DictReader(f)
    for row in reader:
        try:
            salary = int(row["salary"])
        except (ValueError, KeyError):
            continue
        if row.get("level") != "senior":
            continue
        dept = row["dept"]
        if dept not in results:
            results[dept] = {"count": 0, "total": 0, "top": None}
        results[dept]["count"] += 1
        results[dept]["total"] += salary
        if results[dept]["top"] is None or salary > results[dept]["top"]["salary"]:
            results[dept]["top"] = {"name": row["name"], "salary": salary}

for dept, stats in sorted(results.items()):
    avg = stats["total"] / stats["count"]
    print(f"{dept}: count={stats['count']}, avg=${avg:,.0f}, top={stats['top']['name']}")
```

Mutable `results` dict, nested conditionals, logic tangled with I/O. Now let's rewrite it.

Functional Rewrite - Step 1: Lazy Reading

`read_employees` is a generator - it yields one dict per valid row without holding the whole file in memory. The `try/except` inside silently discards malformed rows so the rest of the pipeline never sees parse errors.

```
import csv
from itertools import groupby
from operator import itemgetter
from functools import partial

def read_employees(path):
    """Lazy generator: yield one parsed dict per valid row."""
    with open(path) as f:
        reader = csv.DictReader(f)
        for row in reader:
            try:
                yield {**row, "salary": int(row["salary"])}
            except (ValueError, KeyError):
                pass   # skip malformed rows silently
```

Functional Rewrite - Step 2: Filter with `partial`

`is_level` is a reusable two-argument predicate. `partial` freezes `"senior"`, producing a unary function that fits directly into `filter`. Nothing runs yet - `senior_stream` is still a lazy generator.

```
def is_level(level, record):
    return record.get("level") == level

is_senior = partial(is_level, "senior")

# Still lazy — no rows have been read from disk yet
senior_stream = filter(is_senior, read_employees("employees.csv"))
```

Each function has a single responsibility and can be tested with a plain list of dicts - no file system, no state, no surprises.

Functional Rewrite - Steps 3 and 4

Step 3: sort for `groupby` - the first point data is pulled through the pipeline. Only senior records are materialised; everything else stays on disk.

```
# groupby requires sorted input — this is the pipeline's one materialisation point
seniors = sorted(senior_stream, key=itemgetter("dept"))
```

Step 4: aggregate each group

```
def dept_stats(dept, group):
    members  = list(group)
    salaries = [m["salary"] for m in members]
    top      = max(members, key=itemgetter("salary"))
    return {
        "dept":  dept,
        "count": len(members),
        "avg":   sum(salaries) / len(salaries),
        "top":   top["name"],
    }
```

Functional Rewrite - Step 5: Compose and Print

Wire `groupby` and `dept_stats` together. The list comprehension collects one summary dict per department; the final `sorted` orders the output.

```
stats = [
    dept_stats(dept, group)
    for dept, group in groupby(seniors, key=itemgetter("dept"))
]

for s in sorted(stats, key=itemgetter("dept")):
    print(f"{s['dept']}: count={s['count']}, avg=${s['avg']:,.0f}, top={s['top']}")
```

Output:

```
Engineering: count=3, avg=$97,667, top=Carol
HR: count=1, avg=$72,000, top=Dave
Sales: count=1, avg=$95,000, top=Grace
```

Before and After

Imperative

- 1 large `with` block
- Mutable `results` dict updated in place
- Nested `if` for first-time key insertion
- Logic, I/O, and aggregation all mixed
- Hard to test any step in isolation

Functional

- `read_employees` - one job: yield parsed rows
- `filter(is_senior, ...)` - one job: discard juniors
- `sorted(...)` - one job: prepare for groupby
- `dept_stats(dept, group)` - one job: aggregate
- Each function is testable with a list of dicts - no file needed

The functional version is not shorter by line count, but every piece has a **single responsibility** and a **clear name**. Bugs are isolated. Steps are reusable.

The functional version is not a toy - this exact pattern (lazy source -> named predicate -> sort -> groupby -> aggregate) handles files with millions of rows without modification. The imperative version would require a complete rewrite to stream data.

> This is composability in practice: a personal library of small, tested functions that snap together for any new task.

The Complete Functional Pipeline

All five steps assembled. The entire solution — lazy I/O, filtering, grouping, and aggregation — in one place with no mutable state.

```
import csv
from itertools import groupby
from operator import itemgetter
from functools import partial

def read_employees(path):
    with open(path) as f:
        reader = csv.DictReader(f)
        for row in reader:
            try:
                yield {**row, "salary": int(row["salary"])}
            except (ValueError, KeyError):
                pass

def is_level(level, record):
    return record.get("level") == level

def dept_stats(dept, group):
    members  = list(group)
    salaries = [m["salary"] for m in members]
    top      = max(members, key=itemgetter("salary"))
    return {"dept": dept, "count": len(members),
            "avg": sum(salaries) / len(salaries), "top": top["name"]}

is_senior = partial(is_level, "senior")
seniors   = sorted(filter(is_senior, read_employees("employees.csv")),
                   key=itemgetter("dept"))
stats     = [dept_stats(d, g) for d, g in groupby(seniors, key=itemgetter("dept"))]

for s in sorted(stats, key=itemgetter("dept")):
    print(f"{s['dept']}: count={s['count']}, avg=${s['avg']:,.0f}, top={s['top']}")
```

Practice - Log Parsing Pipeline

Given a list of log entries in the format `"YYYY-MM-DD LEVEL message"`:

```
logs = [
    "2026-04-28 ERROR timeout in db",
    "2026-04-28 INFO server started",
    "2026-04-29 ERROR disk full",
    "2026-04-28 ERROR connection refused",
    "2026-04-29 INFO backup complete",
    "2026-04-29 ERROR segfault in worker",
    "2026-04-30 INFO maintenance",
]
```

Using a **functional pipeline** (no `for` loops with accumulators, no mutable dicts built by hand), produce:

```
2026-04-28: 2 errors
2026-04-29: 2 errors
```

Only ERROR lines, grouped by date, counted per day, sorted descending by count.

Solution

Each step is a pure transformation: parse -> filter -> sort -> group -> count -> sort. No mutable accumulator, no nested loop. The `groupby` step requires the prior `sorted` because `groupby` only groups consecutive equal keys.

```
from itertools import groupby
from operator import itemgetter

def parse_log(line):
    parts = line.split(" ", 2)
    return {"date": parts[0], "level": parts[1], "msg": parts[2]}

parsed = map(parse_log, logs)
errors = filter(lambda r: r["level"] == "ERROR", parsed)
sorted_errors = sorted(errors, key=itemgetter("date"))
counts = [{"date": d, "count": sum(1 for _ in g)}
          for d, g in groupby(sorted_errors, key=itemgetter("date"))]
result = sorted(counts, key=itemgetter("count"), reverse=True)

for r in result:
    print(f"{r['date']}: {r['count']} errors")
```

Summary

Advanced Generators

- Iterator protocol: `__next__`, `StopIteration`, one-way
- `yield from` - delegate to sub-generators
- `tee` - branch; watch the memory cost
- `send()` - two-way coroutines
- `throw()` / `close()` - exception injection, cleanup

Advanced itertools

- `compress` - boolean mask filter
- `filterfalse` + `tee` - split into pass/fail
- `takewhile` / `dropwhile` - stream sectioning
- `starmap` - multi-argument map
- `accumulate` - running results (not just final)
- `groupby` - sort first, consume groups immediately
- `product` + `starmap` - parameter grid evaluation

`operator` Module

- `itemgetter`, `attrgetter`, `methodcaller`
- Replace lambdas for single operations
- Composes with `reduce`, `accumulate`, `sorted`
- Lundh's rules: name your lambdas

`functools` Depth

- `reduce`: supply `initial`; know when a loop wins
- `partial`: fit multi-arg functions into pipelines
- `total_ordering`: define 2, get 6
- `lru_cache`: `cache_info()`, hashable args only

Thank You!

Contact Information

- **Email:** ekrem.cetinkaya@yildiz.edu.tr

- **Office Hours:** Wednesday 13:30-15:30 - Room C-120
- **Book a slot before coming:** Booking Link
- **Course Repository:** GitHub

Next Week

- **Week 11:** Recursion and Dynamic Programming