CS1010X: Programming Methodology I

Chapter 8: Working with Sequences, Functional Programming and Trees

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

---

8 Working with Sequences, Functional Programming and Trees **3**
8.1 Sequences and Collections . . . 3
8.1.1 Comparing Programming Paradigms . . . 4
8.1.2 Functional Programming as a Pipeline . . . 4
8.2 Recursive Problem Solving with Tuples . . . 5
8.3 Reversing a Sequence . . . 6
8.3.1 Recursive version (first & rest) . . . 6
8.3.2 Recursive version (last & rest) . . . 7
8.3.3 Iterative version (prepending) . . . 8
8.3.4 Iterative version (appending) . . . 9
8.3.5 Additional Method (Not in Lecture): Built-in slicing: `seq[::-1]` . . . 10
8.3.6 Summary of the Reverse Variants . . . 12
8.4 Abstracting Generic Tuple Operations with `accumulate` . . . 13
8.4.1 Recursive `accumulate` (Right Fold) . . . 13
8.4.2 Iterative `accumulate` . . . 17
8.5 Standard Functional Programming Combinators . . . 19
8.5.1 `enumerate_interval`: Generating indices . . . 19
8.5.2 `map`: Pointwise transformation . . . 19
8.5.3 `filter`: Selecting elements . . . 19
8.5.4 `accumulate`: Cumulative combinations . . . 20
8.6 Functional Programming: Chaining Higher-Order Functions . . . 20
8.6.1 Cleaning and analysing a data stream . . . 20
8.6.2 One-liner Expressions . . . 21
8.7 Python's Built-In `map`, `filter`, and `accumulate` . . . 22
8.7.1 Iterables and Iterators . . . 22
8.7.2 `map`: Lazy Element-wise Transformation . . . 22
8.7.3 `filter`: Lazy Selection . . . 23
8.7.4 `accumulate`: Fold Over Iterables . . . 23
8.8 Trees as Nested Sequences . . . 25
8.8.1 What is a Tree? Non-linear, Recursive Data Structure . . . 25
8.8.2 A Recursive Pattern for Tree Algorithms . . . 26
8.8.3 Counting Leaves . . . 26
8.8.4 Flattening a Tree to a Sequence of Leaves . . . 27

8.8.5 Using Flattening to Count Leaves . . . 27
8.8.6 Sum of Squares of Odd Leaves . . . 28
8.8.7 Scaling a Tree . . . 28
8.8.8 Deep Copying a Tree and Aliasing . . . 30
8.8.9 Tree Map: Generalising Tree Scaling . . . 31
8.8.10 Tree Accumulate: Combining Leaves with `accumulate` . . . 31
8.8.11 Filtering a Tree . . . 32
8.8.12 Recursion vs. Iteration on Trees . . . 33
8.9 Reference: Methods of Common Collections . . . 35
8.9.1 Generic built-in functions on iterables . . . 35
8.9.2 Tuple methods (immutable sequence) . . . 35
8.9.3 List methods (mutable sequence) . . . 36
8.9.4 Dictionary methods (mutable mapping) . . . 37
8.9.5 Set methods (mutable collection of unique items) . . . 38

Chapter 8: Working with Sequences, Functional Programming and Trees

Learning Objectives

By the end of this chapter, students should be able to:

- **Distinguish** between *sequences* and more general *collections* in Python.
- **Use** core tuple operations (construction, indexing, slicing, concatenation, repetition) and **state** their asymptotic time complexities.
- **Design**, **trace**, and **compare** recursive and iterative algorithms on tuples using "first & rest" and "last & rest" patterns.
- **Analyse** the time and space complexity of different `reverse` implementations, explaining why naive concatenation-based versions are $O(n^2)$.
- **Define** and **use** `accumulate` (right fold), both recursively and iteratively, to express generic tuple computations such as reversal and summation.
- **Relate** the cost of `accumulate` to the cost of its combining function, and **reason** (using series such as $\sum i^p$) about the time and space complexity of a function.
- **Use** the standard higher-order combinators `enumerate_interval`, `map`, `filter`, and `accumulate` to build and read functional pipelines on sequences, including compact one-line expressions.
- **Explain** the "signal processing" view of functional programming, where data flows through stages that transform, filter, and aggregate data.
- **Understand** what iterables and iterators are, **describe** the single-use and lazy nature of iterator-based pipelines, and **use** Python's built-in `map`, `filter`, and `itertools.accumulate` in relation to our tuple-based versions.
- **Understand** the recursive structure of trees, and leverage its recursive structure to **write** recursive algorithms for operations trees using functional styles.
- **Recall** common methods on tuples, lists, dictionaries, and sets; and **estimate** their rough time complexities.

8.1 Sequences and Collections

In Python you will encounter many types that store multiple values: tuples, lists, strings, dictionaries, sets, and so on. It is useful to separate two ideas:

- **Sequence** An *ordered* collection of items. Every element has a position (index) $0, 1, 2, \ldots$ Examples: tuples, lists, strings.
- **Collection** A more general term for a group of items, ordered or unordered. Examples: sets, dictionaries.

In this chapter we focus on working with tuples, as ideas used in this chapter will be applicable to other sequences too.

8.1.1 Comparing Programming Paradigms

There are three main programming paradigms that you may come across.

Imperative programming

- Main idea: *tell the computer how to do it*, step by step.
- Focus on changing variables and program state: `x = 0; x = x + 1;`.
- Typical tools: loops (`for`, `while`), assignments, mutable data.
- Good mental model: a recipe that updates a shared workspace.

Object-Oriented programming (OOP)

- Main idea: *bundle data together with the operations* on that data.
- Programs are organised around *objects* with attributes (data) and methods (behaviour).
- Emphasises concepts such as encapsulation, inheritance, and polymorphism.
- Example: a `Student` object with fields like `name`, `courses`, and methods like `add_course` or `compute_gpa`.

Functional programming (FP)

- Main idea: *describe what to compute*, not how to update state.
- Treats functions as *values*: you can pass them around, compose them, and build pipelines like `accumulate(f, 0, map(g, seq))`.
- Prefers immutable data and avoids side effects where possible.
- Good mental model: signal-processing style pipelines where a sequence flows through independent stages (`map`, `filter`, `accumulate`).

Python supports *all three* styles.

- Most first-year code is written in an imperative style (variables + loops).
- OOP becomes important when designing large systems with many interacting entities.
- FP ideas (higher-order functions, composition, immutability) are the key focus in this chapter: they give us powerful abstractions for reasoning about sequences and building reusable "processing blocks".

8.1.2 Functional Programming as a Pipeline

Functional programming encourages us to think of programs as *transformations of data* rather than as step-by-step updates to mutable variables. When we work with sequences, a very relevant mental model is:

> Data flows through a chain of simple components, each of which transforms it slightly and passes it on.

This is exactly how many *signal processing* systems are described:

- A signal is a sequence of values over time.
- Each block (filter, amplifier, detector, ...) transforms the signal.
- Blocks can be connected into a pipeline or network.

8.2 Recursive Problem Solving with Tuples

We would like to perform operations on sequences, using tuples as a starting point.

Recap: Tuple Operations

The following tuple operations are reproduced here from the previous chapter for ease of reference.

Operation	Syntax / Example	Time	Notes
Constructors	`()`	$O(1)$	Create the unique empty tuple.
	`(1, 'ab'), (1, 2, 3)`	$O(n)$	Builds a new tuple of length n.
	`(42,)`	$O(1)$	Trailing comma is required; `(42)` is just the integer 42.
Indexing	`triple[0], triple[2]`	$O(1)$	Random access by position; indices start at 0.
Length	`len(triple)`	$O(1)$	Length is stored with the tuple, so lookup is constant-time.
Concatenation	`a + b` `(1,2) + (3,4) # (1, 2, 3, 4)`	$O(\|a\| + \|b\|)$	Creates a *new* tuple containing all elements of a followed by all elements of b.
Repetition	`t * k` `('hi',) * 3 # ('hi', 'hi', 'hi')`	$O(n \cdot k)$	Returns a new tuple with the original contents repeated k times; the original tuple t is unchanged.

Table 8.1: Basic tuple operations: syntax, complexity, and behaviour.

For recursive algorithms, four slice operations are especially important:

Expression	Meaning	Time Complexity
`seq[0]`	first element	$O(1)$
`seq[1:]`	all but first	$O(n)$
`seq[-1]`	last element	$O(1)$
`seq[:-1]`	all but last	$O(n)$

These support two fundamental ways to traverse any sequence:

- **First & Rest** (*move forward*): operate on `seq[0]`, then recurse or iterate on `seq[1:]`.
- **Last & Rest** (*move backward*): operate on `seq[-1]`, then recurse or iterate on `seq[:-1]`.

These two views are usually interchangeable if the operations are stateless (that is, it does not depend on the position of the element in the sequence). To see why, for a stateless operation, you should be able to perform the same algorithm on the reversed tuple to obtain the same result!

8.3 Reversing a Sequence

Given a tuple `(1, 2, 3, 4)`, we want the reversed tuple `(4, 3, 2, 1)`. This is a basic but important example of how to use recursion (and iteration) to manipulate sequences, and how to reason about *time* and *space* costs of different implementations.

8.3.1 Recursive version (first & rest)

```python
def reverse(seq):
    if seq == ():
        return ()
    else:
        return reverse(seq[1:]) + (seq[0],)
```

- **Base case:** the empty tuple reverses to itself.
- **Recursive case:** reverse the tail `seq[1:]`, then append the first element at the end.

Concretely, on the tuple $(1,2,3,4)$:

$$\begin{aligned}\text{reverse}((1,2,3,4)) &= \text{reverse}((2,3,4)) + (1,)\\ &= \big(\text{reverse}((3,4)) + (2,)\big) + (1,)\\ &= \big(\big(\text{reverse}((4,)) + (3,)\big) + (2,)\big) + (1,)\\ &= ((4,) + (3,) + (2,) + (1,)) = (4,3,2,1).\end{aligned}$$

Time complexity.

Let $n =$ `len(seq)` for the initial call.

- Each call creates a slice `seq[1:]` and performs a concatenation `... + (seq[0],)`.
- For a tuple of length k, concatenating with a 1-element tuple takes $O(k)$ time (we must copy all k elements into a new tuple).
- The cost of slicing `seq[1:]` is also $O(k)$ for a tuple of length k.
- Therefore, the total cost for the original sequence is $O(n)$, then for `seq[1:]`, it is $O(n-1)$, ..., down to $O(1)$.

So, $T(n) = T(n-1) + O(n) = O(1+2+\cdots+n) = O(n^2)$. The time complexity is $O(n^2)$.

Space complexity.

There are two kinds of extra space needed in this recursion:

- **Call stack:** The recursion depth is n, so there are $O(n)$ stack frames.
- **Tuple storage:** Each frame stores its own sliced tuple of lengths $n, n-1, \ldots, 1$. At the deepest point, all of these slices are still reachable, so the total number of elements stored across them is
$$n + (n-1) + \cdots + 1 = O(n^2).$$

Thus, the *peak* additional memory required is $O(n^2)$. The space complexity is $O(n^2)$.

8.3.2 Recursive version (last & rest)

An alternative way of writing `reverse` is to peel off the *last* element at each step:

```
def reverse(seq):
    if seq == ():
        return ()
    else:
        return (seq[-1],) + reverse(seq[:-1])
```

- **Base case:** the empty tuple reverses to itself.
- **Recursive case:** take the last element `seq[-1]`, put it at the front, then reverse the prefix `seq[:-1]`.

Concretely, on the tuple $(1,2,3,4)$:

$$\begin{aligned}\text{reverse}((1,2,3,4)) &= (4,) + \text{reverse}((1,2,3)) \\ &= (4,) + \big((3,) + \text{reverse}((1,2))\big) \\ &= (4,) + \big((3,) + \big((2,) + \text{reverse}((1,))\big)\big) \\ &= (4,) + (3,) + (2,) + (1,) \\ &= (4,3,2,1).\end{aligned}$$

Time complexity.

Again let $n =$ `len(seq)` for the initial call.

- Each call creates a slice `seq[:-1]` of length $k-1$ when the current tuple has length k.
- Each call also performs a concatenation `(seq[-1],)+ reverse(seq[:-1])`. Here the left tuple has length 1, and the right tuple has length roughly $k-1$, so the concatenation cost is $O(k)$ (we must copy all elements of the right-hand side into a new tuple).
- Thus, when the current length is k, the work per call is $O(k)$ (slice) $+ O(k)$ (concatenate) $= O(k)$ up to a constant.
- Over all recursive calls, the lengths are $n, n-1, \ldots, 1$.

So, $T(n) = T(n-1) + O(n) = O(1+2+\cdots+n) = O(n^2)$. The time complexity is $O(n^2)$.

Space complexity.

As before, there are two main contributors:

- **Call stack:** Recursion depth is n, so there are $O(n)$ stack frames.
- **Tuple storage:** At call depth i, the slice `seq[:-1]` has length $n-i$. At the deepest point in the recursion, all these slices are still reachable with lengths $n, n-1, \ldots, 1$. The total number of elements held across all slices is

$$n + (n-1) + \cdots + 1 = O(n^2).$$

Thus the *peak* additional memory is again $O(n^2)$, so the space complexity is also $O(n^2)$.

8.3.3 Iterative version (prepending)

We can also reverse iteratively, using a loop and building a new tuple `result`:

```
def reverse(seq):
    result = ()
    for item in seq:
        result = (item,) + result   # prepend
    return result
```

So, on the tuple `(1, 2, 3, 4)`:

$$\begin{aligned} &\texttt{result} = () \\ \text{process 1}: \quad &\texttt{result} = (1,) + () = (1,) \\ \text{process 2}: \quad &\texttt{result} = (2,) + (1,) = (2,1) \\ \text{process 3}: \quad &\texttt{result} = (3,) + (2,1) = (3,2,1) \\ \text{process 4}: \quad &\texttt{result} = (4,) + (3,2,1) = (4,3,2,1). \end{aligned}$$

Time complexity.

At the i-th iteration, `result` has length i. The concatenation `(item,)+ result` must copy i elements, thus taking $O(i)$ time.

Therefore, the total time taken is:

$$O(0 + 1 + 2 + \cdots + (n-1)) = O(n^2).$$

So even though we avoided recursion, the use of repeated tuple concatenation still costs $O(n^2)$ time, so time complexity is still $O(n^2)$.

Space complexity.

- There is no deep call stack; stack usage is $O(1)$.
- At iteration i, Python must create a new tuple of length $i+1$ and temporarily keep the old tuple of length i until it can be discarded.
- So, at any particular moment, the *largest* tuple is of length n, so the peak extra space for the final result is $O(n)$.
- Across the whole run, we *allocate* a total of $1 + 2 + \cdots + n = O(n^2)$ element slots, but many of these are short-lived.

Considering the peak memory, we say that the space complexity is $O(n)$.

8.3.4 Iterative version (appending)

Now consider a very similar-looking loop, but with *append* instead of *prepend*:

```
def reverse(seq):
    result = ()
    for item in seq:
        result = result + (item,)   # append
    return result
```

Starting from the tuple `(1, 2, 3, 4)`, the updates to `result` look like this:

$$\begin{aligned} &\texttt{result} = () \\ \text{process 1}: \quad &\texttt{result} = () + (1,) = (1,) \\ \text{process 2}: \quad &\texttt{result} = (1,) + (2,) = (1,2) \\ \text{process 3}: \quad &\texttt{result} = (1,2) + (3,) = (1,2,3) \\ \text{process 4}: \quad &\texttt{result} = (1,2,3) + (4,) = (1,2,3,4). \end{aligned}$$

So we end up reconstructing the *original* sequence rather than reversing it.

Time complexity. At the i-th iteration (starting from 0), `result` has length i. The operation

```
result = result + (item,)
```

must copy all i existing elements into a new tuple and then append the new item. Thus each concatenation costs $O(i)$ time.

Summing over all iterations:

$$O(0 + 1 + 2 + \cdots + (n-1)) = O(n^2).$$

So, just like the prepending loop, this appending loop also has quadratic time complexity.

Space complexity.

- There is no recursion, so the call stack is still $O(1)$.
- At iteration i, Python creates a new tuple of length $i+1$ and keeps the old tuple of length i alive until `result` is rebound. The largest tuple ever held is the final one of length n.
- Thus, if we care about *peak* additional memory at any moment, it is $O(n)$.
- As with the prepending version, over the entire execution we *allocate* a total of $1 + 2 + \cdots + n = O(n^2)$ element slots, but many of these intermediate tuples are short-lived.

So the space complexity (in terms of maximum live data at once) is $O(n)$.

Semantics and Cost. Comparing the two iterative versions:

- **Semantics:** With concatenation, the order in which you place the pieces matters:

  `(item,) + result` vs. `result + (item,)`.

  The former *prepends* and produces the reversed sequence; the latter *appends* and simply rebuilds the original sequence.

- **Cost:** In both cases, each concatenation copies a tuple whose length grows from 0 up to n, so both versions still take $O(n^2)$ time. Changing the order of concatenation changes the *result*, not the asymptotic time complexity.

8.3.5 Additional Method (Not in Lecture): Built-in slicing: `seq[::-1]`

Python provides a very compact way to reverse many built-in sequence types (strings, tuples, lists) using slicing with a negative step:

```
t = (1, 2, 3, 4)
print(t[::-1]) # (4, 3, 2, 1)
```

The general slicing form is `seq[start:stop:step]`. When we write `seq[::-1]`, we mean:

- `start`: omitted ⇒ start from the end,
- `stop`: omitted ⇒ go all the way to the beginning,
- `step`: `-1` ⇒ move one element at a time in reverse.

Conceptually, Python will:

1. Determine how many elements will be in the slice (here, all n elements).
2. Allocate a *single* new tuple of length n.
3. Traverse the original tuple once, copying each element into the new tuple.

Time complexity.

- Each element of `seq` is visited exactly once.
- Each element is copied exactly once.

The total work grows linearly with the length of the sequence, so the time complexity is $O(n)$.

Space complexity.

- Python allocates one new tuple of length n to hold the result.
- Apart from a few loop variables (constant size), there are no large intermediate tuples.

Therefore, the space complexity is also $O(n)$.

Deep Dive: Why `[::-1]` is $O(n)$

Python slicing for built-in sequences (such as tuples) is implemented in C inside CPython. For an *extended* slice such as `seq[::-1]` (negative step), CPython does:

1. Compute **start**, **step**, and the resulting **slice length**. Internally, CPython normalises these with helper functions so that it knows:
   - where to start (*start*),
   - by how much to move each step (*step*, which is -1 for `[::-1]`),
   - how many elements there will be (*slicelen*).
2. Allocate a result tuple of exactly the right size:
   `result = PyTuple_New(slicelen).`
3. Perform a simple loop to fill it:

```c
Py_ssize_t cur = start;
for (Py_ssize_t i = 0; i < slicelen; i++) {
    PyObject *v = PyTuple_GET_ITEM(a, cur);  // read source
    Py_INCREF(v);                             // adjust
        refcount
    PyTuple_SET_ITEM(result, i, v);          // store in
        result
    cur += step;                              // move by 'step
        ' (e.g. -1)
}
```

   Each iteration:
   - reads *one* element from the original tuple,
   - stores it at the next index in the new tuple,
   - advances `cur` by `step`.[a]

For a reverse slice `seq[::-1]` on a tuple of length n:

- `start` is normalised to $n - 1$,
- `step` is -1,
- `slicelen` is n.

The loop therefore executes exactly n iterations, each doing $O(1)$ work (pointer look-up, refcount increments, pointer stores). There are no nested loops and no repeated reallocation / concatenation.

Conclusion.

- `seq[::-1]` visits each element a constant number of times and allocates one new container of size n.
- Thus it runs in $O(n)$ time and uses $O(n)$ extra space for the new tuple or list.
- This is fundamentally more efficient than a naive Python implementation that repeatedly concatenates tuples (which leads to $O(n^2)$ total copying work).

[a] For more details, see the CPython source code for tuple slicing on GitHub: https://github.com/python/cpython/blob/main/Objects/tupleobject.c.

8.3.6 Summary of the Reverse Variants

Version	Style	Time	Peak extra space
Recursive (first & rest)	recursive	$O(n^2)$	up to $O(n^2)$
Recursive (last & rest)	recursive	$O(n^2)$	up to $O(n^2)$
Iterative (prepend)	iterative	$O(n^2)$	$O(n)$
Iterative (append, no reverse)	iterative	$O(n^2)$	$O(n)$
Built-in slice `seq[::-1]`	slicing	$O(n)$	$O(n)$

In practice, you should almost always use Python's built-in, linear-time mechanisms (such as `seq[::-1]` or `reversed(seq)` where appropriate) when you actually need a reversed sequence.

Our hand-written versions are valuable as *conceptual* tools:

- they illustrate how recursion and iteration operate on sequences,
- they make the cost of repeated concatenation visible,
- they provide concrete examples for time and space complexity analysis.

But when writing real programs, prefer the idiomatic $O(n)$ slicing or built-in tools.

8.4 Abstracting Generic Tuple Operations with `accumulate`

The recursive `reverse` function has the shape

```
def reverse(seq):
    if seq == ():
        return ()
    else:
        return reverse(seq[1:]) + (seq[0],)
```

This pattern is very close to a general **right fold** over a tuple: combine all elements into a single result using a binary function.

8.4.1 Recursive `accumulate` (Right Fold)

We define:

```
def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))
```

Conceptually,

$$\texttt{accumulate(fn, initial, } (x_0,\ldots,x_{n-1})\texttt{)} = \texttt{fn}(x_0,\ fn(x_1,\ \ldots\ \texttt{fn}(x_{n-1}, \texttt{initial})\ldots))$$

You can think of `accumulate` as a small *processing block*, similar to a signal-processing element:

- **Input:** a sequence `seq` and an **initial** value.
- **Output:** a single value, obtained by repeatedly combining elements of `seq` with the current "state".

We can implement `reverse` using `accumulate`:

```
def reverse(seq):
    return accumulate(lambda x, acc: acc + (x,), (), seq)
```

In this implementation,

- `fn` is `lambda x, acc: acc + (x,)`:
  - `x` is the current element from the front of `seq`;
  - `acc` is the "suffix" already processed (the result of folding the tail);
  - we append `x` at the *end* of that suffix.
- The initial accumulator is the empty tuple `()`.

On the tuple `(1, 2, 3, 4)`:

$$\begin{aligned}\texttt{accumulate(fn, (), (1,2,3,4))} &= \texttt{fn}(1, \texttt{ accumulate(fn, (), (2,3,4))}) \\ &= \texttt{fn}(1, \texttt{ fn}(2, \texttt{ accumulate(fn, (), (3,4))})) \\ &= \texttt{fn}(1, \texttt{ fn}(2, \texttt{ fn}(3, \texttt{ fn}(4, ())))).\end{aligned}$$

Unrolling:

$$\begin{aligned}\texttt{fn}(4, ()) &= () + (4,) = (4,) \\ \texttt{fn}(3, (4,)) &= (4,) + (3,) = (4,3) \\ \texttt{fn}(2, (4,3)) &= (4,3) + (2,) = (4,3,2) \\ \texttt{fn}(1, (4,3,2)) &= (4,3,2) + (1,) = (4,3,2,1).\end{aligned}$$

So the result is $(4,3,2,1)$, as desired.

We can also implement summation using accumulate:

```
def sum(seq):
    return accumulate(lambda x, acc: x + acc, 0, seq)
```

On `(1, 2, 3)`:

$$\begin{aligned}\texttt{accumulate(fn, 0, (1,2,3))} &= fn(1, fn(2, fn(3, 0))) \\ &= 1 + (2 + (3 + 0)) = 6.\end{aligned}$$

Here `fn` is just addition, which is $O(1)$ per call.

Time and Space Complexity of Recursive `accumulate`

Let $n = \texttt{len(seq)}$.

The recursive version above has two sources of cost:

1. **Slicing.** Each call computes `seq[1:]`, which for a tuple of length k costs $O(k)$ time and allocates a new tuple of length $k - 1$.
2. **Combining function fn.** The cost of `fn(x, acc)` depends on what it does.

Because the time and space complexity of `fn(x, acc)` is variable, the time complexity also depends on the function we use.

Case A: cheap `fn` (e.g. addition).

Suppose `fn` takes $O(1)$ time (e.g. adding numbers).

- At the top level, we slice off `seq[1:]` of length $n-1$: cost $O(n)$.
- Next call slices a length-$n-1$ tuple: cost $O(n-1)$, and so on.
- Final call slices a length-1 tuple: cost $O(1)$.

So time is $T(n) = T(n-1) + O(n) = O(1+2+\cdots+n) = O(n^2)$.

Even though `fn` itself is cheap, the repeated slicing makes this $O(n^2)$ in time.

For space, the slices `seq[1:]`, `seq[2:]`, ..., exist simultaneously (reachable on the call stack) at the deepest point, which will contain $(n-1)+(n-2)+\cdots+1 = O(n^2)$ elements across all slices. Thus peak additional memory can reach $O(n^2)$, plus $O(n)$ stack frames. So, the space complexity is $O(n^2)$.

Case B: expensive `fn` (e.g. tuple concatenation).

For reversing a tuple, we use the function:

```
lambda x, acc: acc + (x,)
```

If `acc` has length k, concatenation `acc + (x,)` takes $O(k)$ time to copy the k elements into a new tuple.

As the recursion unwinds, `acc` grows from length 0 up to $n-1$, so concatenation alone costs $O(1+2+\cdots+n) = O(n^2)$ time. Combined with slicing, the overall complexity is still $O(n^2)$ in time, so time complexity is $O(n^2)$.

Peak extra memory will also be $O(n^2)$, so space complexity is $O(n^2)$.

Case C: Very expensive `fn` (worse than $O(n^2)$).

In general, the total running time of `accumulate` is:

$$T(n) \approx \underbrace{\text{slicing cost}}_{O(n^2)} + \underbrace{\sum_{k=1}^{n} T_{\texttt{fn}}(k)}_{\text{cost of combiner}},$$

where $T_{\texttt{fn}}(k)$ is the cost of `fn` on the call where the remaining sequence has length k.

- In Case A, $T_{\texttt{fn}}(k) = O(1)$, so $\sum_{k=1}^{n} T_{\texttt{fn}}(k) = O(n)$ and slicing dominates: total $O(n^2)$.
- In Case B, $T_{\texttt{fn}}(k) = O(k)$, so $\sum_{k=1}^{n} T_{\texttt{fn}}(k) = O(n^2)$; both parts are $O(n^2)$.

But we *can* construct situations where `fn` itself is more expensive and dominates the slicing cost.

Example: Suppose `fn` does a quadratic amount of work in the size of the accumulator, e.g. it contains a nested loop over `acc` or builds a new tuple using multiple passes over `acc`:

```
def very_expensive_fn(x, acc): # Toy example: do O(len(acc)^2) work
    total = 0
    for i in range(len(acc)):
        for j in range(len(acc)):
            total += 1
    return acc + (x,)   # plus an O(len(acc)) concatenation
```

On the call where the accumulator has length k, the body runs in $T_{\texttt{fn}}(k) = O(k^2)$ time (the concatenation cost $O(k)$ is dominated by the k^2 loop).

During the recursion, the accumulator grows from length 0 up to $n-1$, so the total time spent *inside* `fn` is:

$$\sum_{k=1}^{n} T_{\texttt{fn}}(k) = \sum_{k=1}^{n} O(k^2) = O\big(1^2 + 2^2 + \cdots + n^2\big) = O(n^3).$$

Adding the slicing cost ($O(n^2)$) does not change the leading term:

$$T(n) = O(n^3) + O(n^2) = O(n^3).$$

So with a sufficiently expensive combiner, the overall complexity of `accumulate` can grow *worse than* quadratic.

General rule of thumb. If the cost of `fn` on an accumulator of "size" k behaves like $T_{\texttt{fn}}(k) = O(k^p)$ for some $p \geq 0$, then ignoring slicing we have:

$$\sum_{k=1}^{n} T_{\texttt{fn}}(k) = O\big(1^p + 2^p + \cdots + n^p\big) = O(n^{p+1}).$$

- $p = 0$: $T_{\texttt{fn}}(k) = O(1) \Rightarrow$ total $O(n)$.
- $p = 1$: $T_{\texttt{fn}}(k) = O(k) \Rightarrow$ total $O(n^2)$.
- $p = 2$: $T_{\texttt{fn}}(k) = O(k^2) \Rightarrow$ total $O(n^3)$.

In our teaching examples, we usually choose `fn` to be cheap (like addition) or at most linear (like tuple concatenation), so $O(n^2)$ is the dominant pattern. But it is important to remember that the *combiner* itself can be the main source of cost.

Deep Dive: Faulhaber's Formula

We asserted that $\sum_{i=1}^{n} i^p = O(n^{p+1})$. This is made precise by Faulhaber's formula,[a]

$$\sum_{i=1}^{n} i^p = \frac{1}{p+1} \sum_{j=0}^{p} \binom{p+1}{j} B_j n^{p+1-j},$$

where B_j are the Bernoulli numbers. This is clearly a polynomial of degree $p+1$, so $\sum_{i=1}^{n} i^p = O(n^{p+1})$.

[a] For an accessible derivation, see M. Paldridge, "Sums of powers".

8.4.2 Iterative `accumulate`

We can implement `accumulate` iteratively to avoid both deep recursion and repeated slicing. To preserve the *right-fold* behaviour, we iterate from the *end* of the sequence:

```
def accumulate_iter(fn, initial, seq):
    acc = initial
    for x in reversed(seq):    # process x_{n-1}, x_{n-2}, ..., x_0
        acc = fn(x, acc)
    return acc
```

Semantically, for a tuple $(x_0, x_1, \ldots, x_{n-1})$ we have:

`accumulate_iter(fn, initial, (`$x_0,\ldots,x_{n-1}$`)) = fn(`x_0`,` $fn(x_1$`, ... fn(`x_{n-1}`, initial)...))`

This is exactly the same right-associated structure as the recursive `accumulate`, but obtained via a loop.

Time and Space Complexity of Iterative `accumulate`

Let $n =$ `len(seq)` and let $T_{\texttt{fn}}(k)$ denote the cost of evaluating `fn(x, acc)` when the current accumulator has "size" (measured in elements, digits, etc.) k.

- Creating `reversed(seq)` for a built-in sequence (tuple, list) is $O(1)$: it produces a lightweight iterator that walks the same underlying storage backwards.
- The `for` loop executes exactly n iterations.
- In each iteration:
  - reading the next element from `reversed(seq)` is $O(1)$,
  - we pay the cost $T_{\texttt{fn}}(k)$ for the current accumulator size k.

Thus the total running time is

$$T(n) = \sum_{i=0}^{n-1} T_{\texttt{fn}}(k_i),$$

where k_i is the size of the accumulator on the i-th iteration.

Similarly, we will analyse two important special cases here:

Case A: `fn` is $O(1)$. Suppose `fn` does constant-time work, e.g. `lambda x, acc: x + acc`

Then $T_{\texttt{fn}}(k) = O(1)$ for all k, so

$$T(n) = \sum_{i=0}^{n-1} O(1) = O(n).$$

There is no hidden slicing cost, and each element is processed once.

Case B: `fn` is expensive. Suppose `fn` builds a larger tuple: `lambda x, acc: acc + (x,)`

If the accumulator currently has length k, then `acc + (x,)` must copy all k elements into a new tuple, so $T_{\texttt{fn}}(k) = O(k)$.

As the loop progresses, the accumulator lengths are $0, 1, 2, \ldots, n-1$. Therefore

$$T(n) = \sum_{k=0}^{n-1} O(k) = O(0 + 1 + \cdots + (n-1)) = O(n^2).$$

So even in the iterative version, a "growing" accumulator that is rebuilt on each step yields quadratic time.

For the space complexity,

- The call stack depth is constant ($O(1)$), since there is no recursion.
- We keep a single accumulator object `acc`, whose maximum size is the size of the final result. This contributes $O(n)$ space.
- When `fn` allocates new objects (e.g. via concatenation), old accumulators become unreachable and can be garbage-collected, so they do not all coexist. The *peak* additional memory is therefore $O(n)$, even though we may allocate $O(n^2)$ total element slots over the whole run in the concatenation case.

In summary:

- Iterative `accumulate` avoids the extra $O(n^2)$ cost due to slicing and deep recursion.
- The overall time complexity is $O(n \cdot T_{\texttt{fn}})$; with constant-time `fn` this is linear, but with concatenation-based `fn` it becomes quadratic.
- Peak extra space is $O(n)$ (for the final accumulator), in contrast to the recursive + slicing pattern, which can reach $O(n^2)$.

Recursive vs Iterative `accumulate`: Cost Comparison

Version	Implementation	Time (fn = $O(1)$)	Peak extra space
Recursive	Recursion + slicing on each call	$O(n^2)$	$O(n^2)$
Iterative	Single loop, no slicing	$O(n)$	$O(n)$
If `fn` itself does tuple concatenation			
Recursive	Recursion + slicing + concat in `fn`	$O(n^2)$	$O(n^2)$
Iterative	Loop + concat in `fn`	$O(n^2)$	$O(n)$

In summary, `accumulate` (fold) is a powerful abstraction for compressing an entire sequence into a single value. The recursive version is great for understanding the *shape* of such computations. The iterative version is closer to how efficient libraries implement folds in practice.

8.5 Standard Functional Programming Combinators

Recall that we first saw these functions in the chapter on higher-order functions. We shall now use them for functional programming.

8.5.1 `enumerate_interval`: Generating indices

```
def enumerate_interval(low, high):
    return tuple(range(low, high + 1))
```

This function manufactures a simple sequence of indices, useful for constructing test data or for feeding into `map` and `filter`.

8.5.2 `map`: Pointwise transformation

```
def map(fn, seq):
    if seq == ():
        return ()
    else:
        return (fn(seq[0]),) + map(fn, seq[1:])
```

Mathematically:

$$\text{map}(f, (x_0, \ldots, x_{n-1})) = \big(f(x_0), \ldots, f(x_{n-1})\big).$$

In the signal processing view, each element flows through the same transformer `fn`.

Example. In order to square each element in a tuple, we can do

```
print(map(lambda x: x * x, (1, -2, 3)))   # (1, 4, 9)
```

8.5.3 `filter`: Selecting elements

```
def filter(pred, seq):
    if seq == ():
        return ()
    elif pred(seq[0]):
        return (seq[0],) + filter(pred, seq[1:])
    else:
        return filter(pred, seq[1:])
```

Mathematically:

$$\text{filter}(p, (x_0, \ldots, x_{n-1})) = (x_i \mid p(x_i) \text{ is True}).$$

In the signal processing view, the predicate `pred` decides whether each element passes through this stage or is dropped.

Example. In order to keep non-negative values in a tuple:

```
print(filter(lambda x: x >= 0, (-2, -1, 0, 3, -5, 4))) # (0, 3, 4)
```

8.5.4 accumulate: Cumulative combinations

```
def accumulate(fn, initial, seq):
    if seq == ():
        return initial
    else:
        return fn(seq[0], accumulate(fn, initial, seq[1:]))
```

This function allows us to combine all elements into a single result. This is usually a terminating function.

Example. To get the sum of squares of a tuple:

```
def sum_of_squares(seq):
    return accumulate(lambda x, acc: x*x + acc, 0, seq)
```

8.6 Functional Programming: Chaining Higher-Order Functions

Notice that most functions above return sequences too. Thus, we can *chain* these stages. The output of one function becomes the input of the next.

8.6.1 Cleaning and analysing a data stream

Suppose `raw` is a tuple of measurements:

```
raw = (0.1, -0.2, 0.9, 1.5, 2.1, -0.1, 1.2)
```

Suppose our goal is to:

1. Remove obviously invalid readings (outside 0–2).
2. Convert to a different unit (multiply by 100).
3. Compute a running sum (or you can even compute an average).

Using our functions, we can write:

```
def clean_and_analyse(raw):
    # Stage 1: keep only values in [0, 2]
    valid = filter(lambda x: 0.0 <= x <= 2.0, raw)
    # Stage 2: scale readings
    scaled = map(lambda x: x * 100.0, valid)
    # Stage 3: running sum
    running = accumulate(scaled, 0.0, lambda x, y: x + y)
    return running
```

Visually, we can express this as a data pipeline:

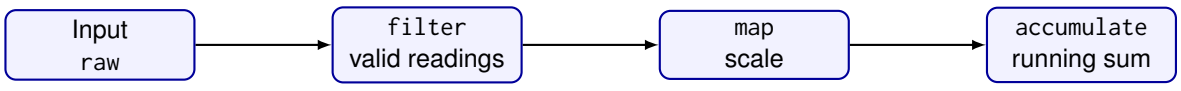

Figure 8.1: Functional sequence processing as a pipeline of stages.

Deep Dive: From Functional Pipelines to MapReduce and Modern ML

Google's **MapReduce** framework (and descendants like Hadoop and Spark) take the same ideas as our `map`/`accumulate` pipelines and scale them to clusters of machines. Conceptually, MapReduce consists of two steps:

- **Map step**: apply a function independently to every record in a huge dataset (often in parallel across many machines).

  $$\text{records} \xrightarrow{\texttt{map}} (\text{key}, \text{value}) \text{ pairs}$$

  This is closely related to our `map(fn, seq)`.
- **Reduce step**: group by key and *combine* all values for that key using an associative operation (sum, max, count, ...).

  $$(\text{key}, [v_1, \ldots, v_k]) \xrightarrow{\texttt{reduce}} (\text{key}, \text{summary})$$

  This is conceptually a large distributed `fold`/`accumulate`.

MapReduce finds a lot of use in data processing and AI / ML.

- **Data preprocessing:** cleaning logs, extracting features, normalising values across petabytes of raw data.
- **Statistics for training:** computing global means/variances, word counts, click-through rates, etc. via large-scale reductions.
- **Model training patterns:** many distributed training algorithms repeatedly:
  1. *map*: compute gradients on mini-batches of data in parallel workers,
  2. *reduce*: aggregate gradients (e.g. via a parameter server or all-reduce) before updating weights.

In summary, the functional style you see in Python: `seq` $\xrightarrow{\texttt{filter}}\xrightarrow{\texttt{map}}\xrightarrow{\texttt{accumulate}}$ `ans` is conceptually identical to large-scale MapReduce pipelines: local transforms followed by global aggregation. Learning to reason in this style on small sequences prepares you for thinking about data flows in real-world ML systems.

8.6.2 One-liner Expressions

We can also write everything in one line since the outputs of a function are inputs to the next.

```
def sum_even_squares(n): # Sum of squares of even numbers in [1, n]
    return accumulate(
        lambda x, acc: x + acc, 0,      # combiner, initial
        map(lambda k: k * k,            # square
            filter(lambda k: k % 2 == 0,   # keep even
                enumerate_interval(1, n)
            )
        )
    )
```

Reading from inside out:

1. `enumerate_interval(1, n)` generates the indices $1, \ldots, n$.
2. `filter` keeps only even indices.
3. `map` squares them.
4. `accumulate` adds the results.

8.7 Python's Built-In `map`, `filter`, and `accumulate`

So far, our own `map`, `filter`, and `accumulate` worked on tuples and returned tuples. Python's built-ins are more general: they work on any *iterable* and usually return a lazy *iterator*.

8.7.1 Iterables and Iterators

At a high level:

- An **iterable** is an object you can loop over in a `for` loop (e.g. tuples, lists, strings, sets, ranges, many user-defined classes).
- An **iterator** is the thing that actually produces values. It is single-use. That is, once the iterator is consumed, it cannot be reused.

Formally,

- An object is an *iterator* if it has a `__next__` method that either
  - returns the next element, or
  - raises `StopIteration` when there are no more.
- An object is *iterable* if it has an `__iter__` method that returns an iterator (or, historically, if it has `__getitem__` with integer indices starting at 0).

Thus, the Python `for` loop essentially makes use of this concept:

```
it = iter(some_iterable)   # calls some_iterable.__iter__()
while True:
    try:
        x = next(it)       # calls it.__next__()
    except StopIteration:
        break
    # use x
```

8.7.2 `map`: Lazy Element-wise Transformation

Python's built-in `map` does *not* return a tuple or list. Instead, it returns a **map object**, which is an iterator that computes each transformed element on demand.

```
seq = (1, 2)
seq_itr = map(lambda x: x * x, seq)  # built-in map -> iterator

next(seq_itr)  # 1
next(seq_itr)  # 4
next(seq_itr)  # StopIteration (no more values)
```

Some important key properties of Python's `map`:

- **Single-use:** Once `seq_itr` is exhausted, it cannot be reused. Any further `next` calls immediately raises `StopIteration`.
- **Lazy:** Values are only computed when requested (by `next` or a `for` loop). If you never consume them, they are never computed.

To *get* the results, you can convert the object to a tuple (or list):

```
seq = (1, 2)
seq_itr = map(lambda x: x * x, seq)
seq_tuple = tuple(seq_itr)

for x in seq_tuple:
    print(x)   # 1, 4
for x in seq_tuple:
    print(x)   # 1, 4 again
```

- The call `tuple(seq_itr)` consumes the iterator once, computes all results, and stores them in a new tuple.
- `seq_tuple` is now an ordinary tuple: you can iterate over it multiple times.

For time and space complexity, let f be the function passed to `map`, T_f be the cost of calling f.

- Creating the map object itself is $O(1)$.
- Iterating over n elements performs n calls to f, so it costs $O(n \cdot T_f)$ time.
- If you convert to a tuple or list, you also pay $O(n)$ for building the container.

8.7.3 `filter`: Lazy Selection

Similarly, built-in `filter(pred, iterable)` returns a **filter object** (an iterator). It yields only those elements for which `pred(element)` is true:

```
nums = (1, 2, 3, 4, 5)
evens_itr = filter(lambda x: x % 2 == 0, nums)

tuple(evens_itr)  # (2, 4)
```

- Internally, `filter` walks through the source iterable, calls `pred` for each element, and yields only the ones that pass.
- Time is $O(n \cdot T_{\text{pred}})$ where T_{pred} is the cost of one predicate call.
- Space is $O(1)$ while iterating lazily; $O(n)$ if you cast it into a concrete sequence.

8.7.4 `accumulate`: Fold Over Iterables

Python does not have a built-in named `accumulate` in the global namespace, but the `itertools` module provides `itertools.accumulate`, which behaves like a running fold:

```
from itertools import accumulate
nums = (1, 2, 3, 4)
print(list(accumulate(nums))) # default operation is +, [1, 3, 6, 10]
print(list(accumulate(nums, lambda x, y: x * y))) # [1, 2, 6, 24]
```

- **Default behaviour:** `accumulate(nums)` yields partial sums: $(1, 1+2, 1+2+3, \dots)$.
- **Custom operation:** you can pass any associative binary function `func`; then the k-th output is `func(... func(func(x0, x1), x2)..., xk)`.
- **Iterator-based:** Again, it returns an iterator; it does not allocate all results in advance.

Time and space complexity is similar to our `accumulate_iter`:

- One pass over the data: $O(n)$ iterations.
- Each iteration calls the combining function once, so total time is $O(n \cdot T_{\text{op}})$.
- Memory is $O(1)$ while streaming; $O(n)$ if collected into a concrete container.

Our `accumulate` vs Python's Versions

- Our `accumulate` returns a *single* final result, and is conceptually similar to Python's `functools.reduce`.
- `itertools.accumulate` returns an iterator over all partial results, behaving like a running-sum or running-product stage.

Looking Forward: Java Streams and Other Lazy Pipelines

Java's **Stream** API is conceptually very similar to Python's iterator-based `map/filter/accumulate` style:

- A `Stream<T>` is a *one-shot* pipeline over a source of data (collection, array, generator, I/O, ...).
- **Intermediate operations** (like `map`, `filter`, `sorted`) are lazy and return a new stream; they correspond to building up a sequence of transformations but do not actually process elements yet.
- A **terminal operation** (like `forEach`, `collect`, `reduce`) triggers evaluation of the pipeline, similar to how iterating over a Python iterator or converting it to a list forces computation.

Example in Java:

```
ints.stream()
    .filter(x -> x % 2 == 0)
    .map(x -> x * x)
    .reduce(0, (acc, x) -> acc + x);
```

This is analogous to the Python pipeline:

```
from itertools import accumulate
evens   = filter(lambda x: x % 2 == 0, ints)
squares = map(lambda x: x * x, evens)
total   = sum(squares)  # or: list(accumulate(squares))[-1]
```

In both languages:

- we build a pipeline of small, composable operations;
- evaluation is single-pass and can be done lazily;

Understanding Python's iterator-based functional tools therefore gives you a good conceptual foundation for working with streams and dataflow APIs in other languages and frameworks.

8.8 Trees as Nested Sequences

So far, our pipelines have worked on *flat* sequences (tuples). We can go one step further and build **hierarchical** data: *trees*.

8.8.1 What is a Tree? Non-linear, Recursive Data Structure

A **tree** is a *non-linear* data structure: instead of a single line of elements, we have branching.

In this chapter we represent a tree as a **sequence of sequences**:

- A **leaf** is any object that is *not* a tuple (e.g. `1`, `'hello'`).
- A **tree** is either
  - the empty tuple `()` (an empty forest), or
  - a tuple whose elements are themselves trees.

For example, the tuple

$$((1,2),(3,4),5)$$

represents the tree in Figure 8.2.

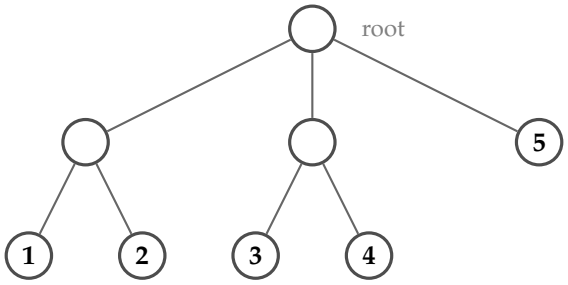

Figure 8.2: A tree represented as a sequence of sequences: $((1,2),(3,4),5)$.

Recursive Definition of a Tree

The definition of a tree is *recursive*:

- Leaves are trees.
- Tuples whose elements are trees are trees.

This closure property (a sequence may contain other sequences) lets us build trees of arbitrary depth.

We will use the following helper to detect leaves:

```
def is_leaf(tree):
    return type(tree) != tuple
```

8.8.2 A Recursive Pattern for Tree Algorithms

Most of our tree functions follow a common recursive pattern. That is, an algorithm that works on a subtree should work for a larger tree.

```
def tree_fn(tree):
    if tree == ():                # empty forest
        ... base case 1 ...
    elif is_leaf(tree):           # single leaf (not a tuple)
        ... base case 2 ...
    else:
        # tree[0] is first subtree, tree[1:] is the rest of the forest
        return combine(
            tree_fn(tree[0]),
            tree_fn(tree[1:])
        )
```

We will reuse this "first subtree & rest of tree" pattern for other tree operations.

8.8.3 Counting Leaves

We first write a function that **counts the number of leaves** in a tree.

```
def count_leaves(tree):
    if tree == ():
        return 0
    elif is_leaf(tree):
        return 1
    else:
        return count_leaves(tree[0]) + count_leaves(tree[1:])
```

On the tree `((1, 2), (3, 4), 5)` we obtain 5 leaves: `1, 2, 3, 4, 5`.

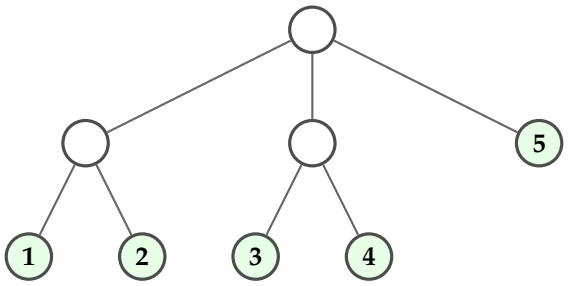

Figure 8.3: There are 5 leaves (highlighted) in this tree.

The key observations:

- An empty tree has 0 leaves.
- A leaf contributes 1 leaf.
- Otherwise, the total leaves in a non-empty forest is

$$\text{leaves of first subtree} + \text{leaves of rest of tree.}$$

8.8.4 Flattening a Tree to a Sequence of Leaves

Next, we write a function that **flattens** a tree into a single tuple consisting of its leaves, from left to right.

```
def tree_flatten(tree):
    """Return a tuple of all leaves in left-to-right order."""
    if tree == ():
        return ()
    elif is_leaf(tree):
        return (tree,)
    else:
        return tree_flatten(tree[0]) + tree_flatten(tree[1:])
```

On our example tree:

`tree_flatten(((1, 2), (3, 4), 5))` $\Rightarrow$ `(1, 2, 3, 4, 5)`.

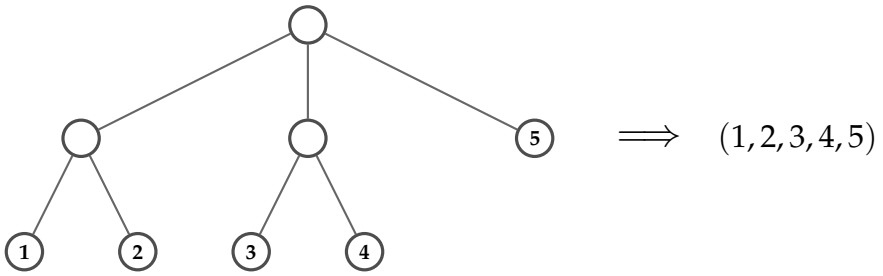

Figure 8.4: Flattening a tree to a tuple of leaves.

The structure of `tree_flatten` is almost identical to `count_leaves`; only the base cases and the combining operation change.

8.8.5 Using Flattening to Count Leaves

Once we have `tree_flatten`, counting leaves becomes trivial:

```
def count_leaves_via_flatten(tree):
    return len(tree_flatten(tree))
```

This uses our earlier fact that `len` on a tuple is $O(1)$.

In practice, the direct `count_leaves` is slightly more efficient (it does not build the whole flattened tuple), but `tree_flatten` is a powerful building block for other tasks.

8.8.6 Sum of Squares of Odd Leaves

We can now combine trees with our earlier `map`, `filter`, and `accumulate` to write high-level functional-style code.

```
def sum_odd_square(tree):
    """Sum the squares of odd leaves in a tree."""
    return accumulate(
        lambda x, acc: x + acc, 0,    # combiner, initial value
        map(lambda x: x * x,          # square each leaf
            filter(lambda x: x % 2 == 1,  # keep odd leaves
                   tree_flatten(tree)))
    )
```

Reading this from inside out:

1. `tree_flatten(tree)` produces all leaves.
2. `filter` keeps only odd numbers.
3. `map` squares those numbers.
4. `accumulate` adds everything up.

This is a direct application of the signal-processing / pipeline mindset:

$$\text{tree} \xrightarrow{\texttt{flatten}} \xrightarrow{\texttt{filter odd}} \xrightarrow{\texttt{square}} \xrightarrow{\texttt{accumulate}} \text{total}.$$

8.8.7 Scaling a Tree

Suppose we want to multiply every leaf in the tree by a fixed **factor**, such as scaling by 2:

$$((1,2),(3,4),5) \longrightarrow ((2,4),(6,8),10).$$

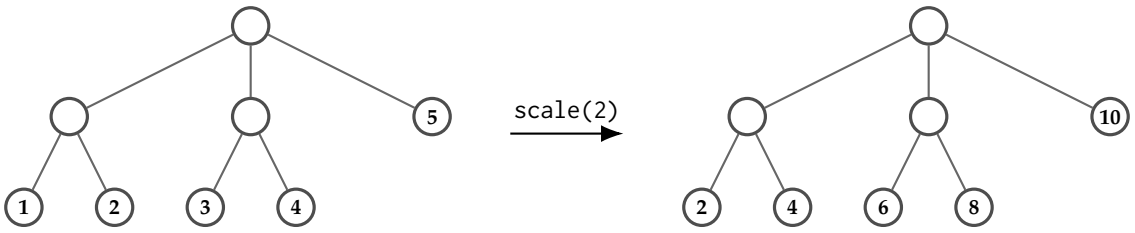

Figure 8.5: Scaling every leaf by 2 preserves structure but changes values.

We can write this directly using recursion, but it is more instructive to reuse our earlier `map` (the tuple-based version from this chapter) and treat the tree itself as a sequence:

```
def tree_scale(tree, factor):
    """Return a new tree with every leaf multiplied by factor."""
    def scale_fn(subtree):
        if is_leaf(subtree):            # at a leaf
            return factor * subtree
        else:                           # still a tree
            return tree_scale(subtree, factor)

    return map(scale_fn, tree)          # use our tuple-based map
```

We can also write it with recursion directly:

- If the tree is empty, return an empty tuple.
- If the tree is a leaf, return the scaled value.
- Otherwise, recursively scale the first element (the first subtree) and the rest of the forest, then combine them.

```
def tree_scale(tree, factor):
    """Return a new tree with every leaf multiplied by factor."""
    if tree == ():
        return ()
    elif is_leaf(tree):
        return tree * factor
    else:
        # Scale the first subtree (head) and the rest of the forest (
            tail)
        # Note the comma to make a singleton tuple for the head
        return (tree_scale(tree[0], factor),) + tree_scale(tree[1:],
            factor)
```

This function builds a entirely new tuple structure that mirrors the original, but with transformed values at the leaves.

8.8.8 Deep Copying a Tree and Aliasing

The function `tree_scale` *returns a new tree*. It never modifies its argument (although tuples are immutable anyway).

This is important when we care about **aliasing**: the same subtree might be referenced from multiple places. If we used a mutable representation (e.g. lists of lists), or the tree contained mutable sequences, and we modified the tree in place, all aliases would see the changes, which may not be what we want.

Instead, we often want a **deep copy**: a tree with the same structure and leaf values, but with entirely new container objects. Note that because a lot of our functions return new instances of the tree, we can directly use them to achieve this.

Examples include:

```
def tree_copy(tree):
    """Return a deep copy of tree (same shape and leaf values)."""
    return tree_scale(tree, 1)
```

```
def tree_copy(tree):
    """Return a deep copy of tree (same shape and leaf values)."""
    return tree_map(lambda x: x, tree)
```

The helper `tree_map` (defined below) applies a function to every leaf and rebuilds the whole tree, producing fresh tuples along the way. This respects the abstraction barrier: higher-level code does not need to know whether the underlying representation uses tuples, lists, or something else.

Aliasing and Deep Copies

- With immutable tuples, aliasing is relatively harmless: no code can change a shared subtree.
- If we later switch to mutable containers (lists, custom classes), in-place updates can unexpectedly affect other parts of the program that share the same object.
- Functions like `tree_copy` make aliasing explicit and help maintain clean invariants when working with more complex data structures.

8.8.9 Tree Map: Generalising Tree Scaling

The structure of `tree_scale` suggests a more general operation: apply an arbitrary function `fn` to *every leaf* in the tree.

```
def tree_map(fn, tree):
    """Apply fn to every leaf in tree; preserve the tree shape."""
    def mapping_fn(subtree):
        if is_leaf(subtree):
            return fn(subtree)
        else:
            return tree_map(fn, subtree)
    return map(mapping_fn, tree)   # our tuple-based map
```

Now `tree_scale` can be defined in one line:

```
def tree_scale(tree, factor):
    return tree_map(lambda x: x * factor, tree)
```

and we also get other useful operations "for free":

```
def tree_copy(tree):
    return tree_map(lambda x: x, tree)        # deep copy

def tree_square(tree):
    return tree_map(lambda x: x * x, tree)    # square all leaves
```

8.8.10 Tree Accumulate: Combining Leaves with `accumulate`

We can go one step further: sometimes we want to *transform* each leaf and then *combine* all results into a single value. One way to do this is to use `tree_flatten` followed by our `accumulate`:

```
def tree_accumulate(fn, initial, tree):
    """Apply fn to each leaf, then combine using accumulate."""
    return accumulate(
        lambda x, y: x + y,
        initial,
        map(fn, tree_flatten(tree))
    )
```

For example, to sum the squares of all leaves:

```
def sum_squares(tree):
    return tree_accumulate(
        lambda x, acc: x * x + acc,
        0,
        tree
    )
```

Here `tree_accumulate` is really "`map` over leaves + `accumulate`" fused into a single abstraction.

8.8.11 Filtering a Tree

Just as we can filter a flat sequence, we can **filter** a tree to keep only the leaves that satisfy a specific condition (a *predicate*). Specifically, when filtering a tree:

- If a leaf is removed, it disappears from its tuple.
- If a subtree (nested tuple) loses all its leaves, it becomes an empty tuple `()`.

For example, if we filter for **odd numbers**:

$$((1,2),3,(4,6)) \longrightarrow ((1,),3,()).$$

Note that `(1, 2)` became `(1,)`, and `(4, 6)` became `()` because no leaves remained there.

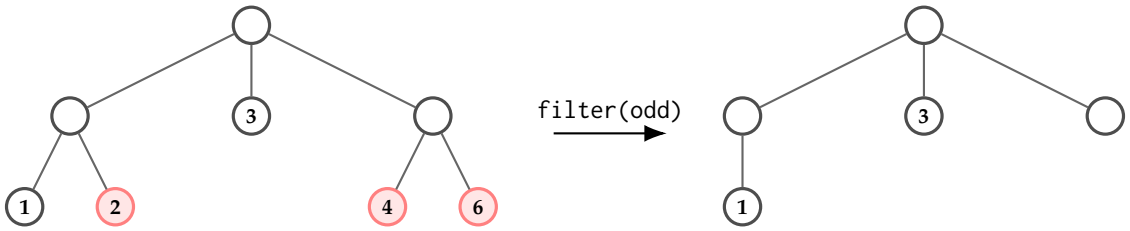

Figure 8.6: Filtering, even numbers are removed; rightmost subtree becomes empty.

We use the standard recursive pattern, but we must handle the head of the sequence carefully. If the head is a leaf, we check the predicate. If the head is a subtree, we recurse into it and wrap the result.

```
def tree_filter(pred, tree):
    """Return a new tree with only leaves that satisfy pred."""
    if tree == ():
        return ()
    # Deconstruct the tree
    first = tree[0]
    rest = tree[1:]
    if is_leaf(first):
        if pred(first):
            # Keep the leaf (wrap in tuple) + filter the rest
            return (first,) + tree_filter(pred, rest)
        else:
            # Drop the leaf + filter the rest
            return tree_filter(pred, rest)
    else:
        # The first element is a subtree (tuple).
        # Filter the subtree, wrap it to preserve nesting, and combine
        return (tree_filter(pred, first),) + tree_filter(pred, rest)
```

Structure Preservation

Notice that when `first` is a subtree, we must include `(tree_filter(pred, first),)` in the result. This preserves the tree's shape, even if the subtree becomes empty.

8.8.12 Recursion vs. Iteration on Trees

For *linear* sequences, we saw that recursion and iteration are often interchangeable. For trees, recursion is much more natural:

- Each recursive call focuses on a smaller subtree.
- The call stack automatically records where to return to after each subtree is processed.

Writing an **iterative** version usually means emulating the call stack explicitly with our own data structure, such as a tuple used as a stack.

Iterative `count_leaves`

Here is an iterative version of `count_leaves` that uses a stack of work to be done:

```
def count_leaves_iter(tree):
    """Iterative count_leaves using an explicit stack."""
    if tree == ():
        return 0

    result = 0
    stack = (tree,)           # stack is a tuple of subtrees

    while stack != ():
        node = stack[0]       # pop the first element
        stack = stack[1:]

        if node == ():
            continue          # empty forest contributes nothing
        elif is_leaf(node):
            result += 1       # found a leaf
        else:
            items = node      # node is a tuple of children
            # push children onto the front of the stack in reverse
                order
            for child in items[::-1]:
                stack = (child,) + stack

    return result
```

Iterative `tree_flatten`

We can adapt the same pattern to flatten a tree iteratively:

```
def tree_flatten_iter(tree):
    """Iterative tree_flatten using an explicit stack."""
    if tree == ():
        return ()

    result = ()
    stack = (tree,)

    while stack != ():
        node = stack[0]
        stack = stack[1:]

        if node == ():
            continue
        elif is_leaf(node):
            result = result + (node,)       # append leaf to result
        else:
            items = node
            for child in items[::-1]:
                stack = (child,) + stack

    return result
```

Why Recursion is Often Preferred for Trees

- The recursive versions of `count_leaves` and `tree_flatten` directly mirror the *recursive definition* of a tree.
- The iterative versions must manually simulate the call stack (push / pop, reverse order, keep track of where we are in the structure).
- For tree-shaped data, thinking recursively is usually clearer and less error-prone. When performance or recursion depth is a concern, we can fall back to carefully written iterative versions with explicit stacks.

8.9 Reference: Methods of Common Collections

So far we have focused on *how* to think about sequences and pipelines. Python also provides many convenient *methods* on its built-in collections. This section is a brief reference; you do *not* need to memorise every method. Instead, focus on:

- which collections are **mutable** vs. **immutable**,
- which operations **create** new objects vs. **modify** existing ones,
- the **rough time complexity** of common methods.

If you are required to make use of any of these operations in examinations, you will be provided with their documentation.

Unless otherwise stated, the complexities below are *average case* for CPython, with n the size of the collection (or of the argument collection, where relevant).

8.9.1 Generic built-in functions on iterables

The following built-in functions work on many iterable types (tuples, lists, strings, sets, ranges, generators, etc.), subject to type-specific rules.

Function	Example	Time	Effect / Notes
`len(it)`	`len((1, 2, 3))`	$O(1)$	Return the number of elements. For built-in containers, the size is stored and can be read in constant time.
`max(it)`	`max((3, 1, 4))`	$O(n)$	Single pass over the iterable to find the largest element (according to the default order or a key function: `max(it, key=...)`. Raises `ValueError` on an empty iterable unless a `default=...` is given.
`min(it)`	`min((3, 1, 4))`	$O(n)$	Single pass to find the smallest element. Same remarks as `max`.
`sum(it, start)`	`sum((1, 2, 3))`	$O(n)$ (for numeric types)	Add up all elements from left to right, starting from `start` (default `0`). Intended for numbers; for concatenation of sequences, prefer `".join(...)` or `itertools.chain`.
`sorted(it)`	`sorted((3, 1, 4))`	$O(n \log n)$ time, $O(n)$ extra space	Return a *new list* containing all items from `it` in ascending order. Does not modify original iterable. Accepts `key=...` and `reverse=True`.

8.9.2 Tuple methods (immutable sequence)

Tuples are immutable, so their methods never modify the tuple; they only *inspect* it.

Method	Example	Time	Meaning
`count(x)`	`(1, 2, 1).count(1)`	$O(n)$	Return the number of occurrences of `x` in the tuple.
`index(x)`	`('a', 'b').index('b')`	$O(n)$	Return index of the first occurrence of `x`, or raise `ValueError` if `x` is not present.

8.9.3 List methods (mutable sequence)

Lists are ordered and *mutable*. Most methods update the list *in place* and return `None`.

Method	Example	Time	Effect
`append(x)`	`lst.append(10)`	$O(1)$ (amortised)	Add `x` at the end of the list.
`extend(it)`	`lst.extend(other)`	$O(k)$	Append all k elements from iterable `other` to the end.
`insert(i, x)`	`lst.insert(0, 'hi')`	$O(n)$	Insert `x` before position `i`, shifting later elements to the right.
`pop()` / `pop(i)`	`lst.pop()`, `lst.pop(0)`	$O(1)$ / $O(n)$	Remove and return the last element (`pop()`) or the element at index `i` (`pop(i)`). Removing from the front requires shifting elements.
`remove(x)`	`lst.remove(3)`	$O(n)$	Find and remove the first occurrence of `x`, or raise `ValueError` if not found.
`clear()`	`lst.clear()`	$O(n)$	Remove all elements from the list.
`reverse()`	`lst.reverse()`	$O(n)$	Reverse the list *in place*.
`sort()`	`lst.sort()`	$O(n \log n)$	Sort the list *in place*; uses Timsort (stable, adaptive).
`copy()`	`lst2 = lst.copy()`	$O(n)$	Return a shallow copy of the list (new list object; same element references).
`count(x)`	`lst.count(0)`	$O(n)$	Count occurrences of `x` in the list.
`index(x)`	`lst.index('a')`	$O(n)$	Return index of first `'a'`, or raise `ValueError` if not present.

In-place vs. New Object

- Methods like `append`, `sort`, `reverse` modify the *same* list object.
- Operators like `+` or slicing `lst[1:]` build a *new* list and leave the original unchanged.
- This matters when several variables alias the same list.

8.9.4 Dictionary methods (mutable mapping)

A dictionary maps *keys* to *values*. It is implemented as a hash table, so most key-based operations are $O(1)$ on average.

Method	Example	Time	Effect
`keys()`	`d.keys()`	$O(1)$ (view)	Return a dynamic view object over all keys. Iterating over the keys takes $O(n)$ time.
`values()`	`d.values()`	$O(1)$ (view)	Return a view object over all values.
`items()`	`d.items()`	$O(1)$ (view)	Return a view over `(key, value)` pairs.
`get(k, default)`	`d.get('id', 0)`	$O(1)$ avg.	Return value for `'id'` if present; otherwise return `0` (or a provided default) without raising an error.
`update(other)`	`x': 1d.update(')`	$O(m)$	Insert or overwrite m key–value pairs from `other` (another dict or iterable of pairs).
`pop(k)`	`d.pop('x')`	$O(1)$ avg.	Remove and return value associated with key `'x'`, or raise `KeyError` if missing (unless default given).
`popitem()`	`d.popitem()`	$O(1)$	Remove and return an arbitrary (in CPython 3.7+, last-inserted) key–value pair.
`setdefault(k, default)`	`d.setdefault('x', 0)`	$O(1)$ avg.	If key `'x'` exists, return its value; otherwise insert `('x', 0)` and return `0`.
`clear()`	`d.clear()`	$O(n)$	Remove all key–value pairs from the dictionary.
`copy()`	`d2 = d.copy()`	$O(n)$	Shallow copy: new dictionary object, but values are the same underlying objects.

8.9.5 Set methods (mutable collection of unique items)

A set is an unordered collection with no duplicates, also implemented as a hash table.

Method	Example	Time	Effect
`add(x)`	`s.add(10)`	$O(1)$ avg.	Insert element `x` into the set (no effect if already present).
`remove(x)`	`s.remove(10)`	$O(1)$ avg.	Remove `x`; raise `KeyError` if `x` not in the set.
`discard(x)`	`s.discard(10)`	$O(1)$ avg.	Remove `x` if present; do nothing if absent (no error).
`pop()`	`s.pop()`	$O(1)$ avg.	Remove and return an arbitrary element from the set.
`clear()`	`s.clear()`	$O(n)$	Remove all elements from the set.
`union(t)`	`s.union(t)`	$O(\|s\| + \|t\|)$	Return a new set with all elements that appear in `s`, `t`, or both.
`intersection(t)`	`s.intersection(t)`	$O(\|s\| + \|t\|)$	Return a new set containing only elements common to both sets.
`difference(t)`	`s.difference(t)`	$O(\|s\| + \|t\|)$	Return a new set of elements in `s` but not in `t`.
`symmetric_difference(t)`	`s.symmetric_difference(t)`	$O(\|s\| + \|t\|)$	Return a new set of elements that are in exactly one of `s` or `t` (but not both).
`issubset(t)`	`s.issubset(t)`	$O(\|s\|)$ to $O(\|s\|+\|t\|)$	Return `True` if every element of `s` is also in `t`.
`issuperset(t)`	`s.issuperset(t)`	$O(\|t\|)$ to $O(\|s\|+\|t\|)$	Return `True` if `s` contains every element of `t`.
`isdisjoint(t)`	`s.isdisjoint(t)`	$O(\min(\|s\|, \|t\|))$	Return `True` if the two sets have no elements in common.