

Haskell Meets Python

Exploring functional recursion across these two languages

Frame 1 · Quicksort in Haskell

The Code

```
1 qs :: [Int] -> [Int]
2 qs [] = []
3 qs (x:xs) =
4   qs smaller ++ [x] ++ qs bigger
5   where
6     smaller =
7       [a | a <= xs, a <= x]
8     bigger =
9       [b | b <= xs, b > x]
```

Script

Step 1. `qs (x:xs)` pattern-matches: `x` is the *pivot*; `xs` is the rest.

Step 2. Everything $\leq$ pivot goes **left**; everything $>$ pivot goes **right**.

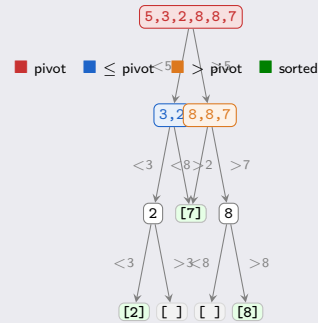
Step 3. Sort each half *recursively* — same rule, smaller list.

Step 4. Concatenate: `sorted_left ++ [pivot] ++ sorted_right`.

Base case. Empty list `[]` returns `[]` — recursion stops.

Pivot role: anchors its final sorted position each call.

Recursion Tree for [5,3,2,8,8,7]



Result: [2, 3, 5, 7, 8, 8] ✓

Each pivot lands in its *final* position after one partition pass.

Frame 2 · Python Quicksort — Head, Tail & Big-O

Why this recursion in Python?

Quicksort averages $\mathcal{O}(n \log n)$ comparisons. The recursive structure maps *directly* onto divide-and-conquer: each call shrinks the problem by roughly half.

Case	Complexity
Average	$\mathcal{O}(n \log n)$
Worst	$\mathcal{O}(n^2)$ (sorted list)
Space	$\mathcal{O}(\log n)$ call stack

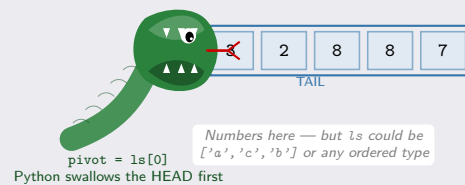
The Haskell idiom of *pattern-matching* on `(x:xs)` translates cleanly to `pivot = ls[0], tail = ls[1:]` — making the algorithm's intent immediately readable.

Python — mirroring Haskell

```
1 def qs(ls):
2     if len(ls) == 0:
3         return ls
4     pivot = ls[0]           # x
5     tail = ls[1:]           # xs
6     small = qs([a for a in tail
7                 if a <= pivot])
8     large = qs([b for b in tail
9                 if b > pivot])
10    return small + [pivot] + large
```

`pivot = ls[0]` mirrors `x`; `tail = ls[1:]` mirrors `xs`. List comprehensions replace Haskell's guards. Structure is identical.

Python masticating Head & Tail



Frame 3 · When Haskell & Python Get Real Close

Recursion patterns that look almost identical

haskellPurple!20

<pre>-- Sum a list mySum [] = 0 mySum (x:xs) = x + mySum xs</pre>	<pre># Sum a list def my_sum(ls): if not ls: return 0 return ls[0]+my_sum(ls[1:])</pre>
<pre>-- Fibonacci fib 0 = 0 fib 1 = 1 fib n = fib(n-1)+fib(n-2)</pre>	<pre># Fibonacci def fib(n): if n<=1: return n return fib(n-1)+fib(n-2)</pre>
<pre>-- Map (double each) dbl [] = [] dbl (x:xs) = (x*2) : dbl xs</pre>	<pre># Map (double each) def dbl(ls): if not ls: return [] return [ls[0]*2]+dbl(ls[1:])</pre>
<pre>-- Flatten nested flat [] = [] flat ([]:xs) = flat xs flat ((y:ys):xs) = y : flat (ys:xs)</pre>	<pre># Flatten nested def flat(ls): if not ls: return [] h = ls[0] if isinstance(h,list): return flat(h+ls[1:]) return [h]+flat(ls[1:])</pre>

Key insight: Haskell's `(x:xs)` pattern is Python's `head = ls[0]; tail = ls[1:]` written more compactly. Both languages share the same structural recursion idea: *solve the head, recurse on the tail, combine*. The main practical difference is that Python lacks *tail-call optimisation* — deep recursion will hit Python's default stack limit of ≈ 1000 frames, while Haskell handles it natively. For production Python, `functools.lru.cache` or iterative rewrites are preferred; for exploration and clarity, the Haskell-style recursive form is beautifully readable in both languages.