0) Grund-Legende Zuordnung

Pseudocode-Idee	Python-Entsprechung (funktional gedacht)
`f(x) = …`	`def f(x): return …`
`x :: xs` (Kopf & Rest)	`head, *tail = xs`
`[]`	`[]`
`xs ++ ys`	`xs + ys` (liefert **neue** Liste)

`match xs with`
`| [] -> A`
`| y::ys -> B`
	`if not xs: A else: y, *ys = xs; B`

`map(f, xs)`	`list(map(f, xs))` **oder** `[f(x) for x in xs]`
`filter(p, xs)`	`list(filter(p, xs))` **oder** `[x for x in xs if p(x)]`
`fold(op, acc, xs)`	`functools.reduce(op, xs, acc)`
`g ∘ f`	`lambda x: g(f(x))`
„immutable Update“	neue Struktur konstruieren (keine In-Place-Mutation)
Sortieren „ohne Seiteneffekt“	`sorted(xs)` (nicht `xs.sort()`)
„Rekursion statt Schleife“	`def f(...): if basis: return …; return f(…kleiner…)`

> Faustregel: **Kein** `.append`, `.extend`, `.sort`, `.pop`, `.remove` im Kern — nutze Ausdrücke, Comprehensions, `map/filter/reduce`, `sorted`, „Kopie + Änderung“.

1) `fold` / Reduktionen

Pseudocode

```
fold(op, acc, xs) =
  match xs with
  | []    -> acc
  | y::ys -> fold(op, op(acc, y), ys)
```

Python

```python
from functools import reduce

def fold(op, acc, xs):
    return reduce(op, xs, acc)
```

Beispiele

```
summe     = fold(lambda a, b: a + b, 0, xs)
produkt   = fold(lambda a, b: a * b, 1, xs)
maximum   = fold(lambda m, b: b if b > m else m, float("-inf"), xs)
```

2) `map` / `filter`

Pseudocode

```
map(f, xs) =
  match xs with
  | []    -> []
  | y::ys -> f(y) :: map(f, ys)

filter(p, xs) =
  match xs with
  | []    -> []
  | y::ys -> if p(y) then y :: filter(p, ys) else filter(p, ys)
```

Python

```
mapped  = [f(x) for x in xs]
gefiltert = [x for x in xs if p(x)]
# oder:
mapped    = list(map(f, xs))
gefiltert = list(filter(p, xs))
```

3) Verschachtelte Summe (`sum_nested`)

Pseudocode

```
sum_nested(node) =
  match node with
  | Zahl z   -> z
  | Liste xs -> fold( (acc, e) -> acc + sum_nested(e), 0, xs )
```

Python

```
def sum_nested(node):
    if isinstance(node, (int, float)):
        return node
    if isinstance(node, list):
        return sum(sum_nested(e) for e in node)
    return 0  # Nicht-Zahlen ignorieren, wie im Pseudocode angedeutet
```

4) Zinseszins (rekursiv + als Fold)

Pseudocode (rekursiv)

```
compound(P, r, n) =
  if n == 0 then P
  else compound(P*(1+r), r, n-1)
```

Python (rekursiv)

```python
def compound(P, r, n):
    if n == 0:
        return P
    return compound(P * (1 + r), r, n - 1)
```

Pseudocode (Fold)

```
compound_fold(P, r, n) =
  fold( (A, _) -> A*(1+r), P, [1..n] )
```

Python (Fold)

```python
from functools import reduce

def compound_fold(P, r, n):
    return reduce(lambda A, _: A * (1 + r), range(n), P)
```

Mit jährlichem Beitrag `c` (purer Zustandstransfer)

Pseudocode

```
compound_with_contrib(P, r, n, c) =
  if n == 0 then P
  else compound_with_contrib(P*(1+r) + c, r, n-1, c)
```

Python

```python
def compound_with_contrib(P, r, n, c):
    if n == 0:
        return P
    return compound_with_contrib(P * (1 + r) + c, r, n - 1, c)
```

---

5) „Dateisuche“ funktional gedacht (Baum statt IO)

> Da echte IO Seiteneffekte hat, trenne **Kern** (Traversal als pure Funktion) und **Rand** (tatsächliche FS-Zugriffe).

Pseudocode (Datenmodell)

```
Tree := File(name) | Dir(name, children: List[Tree])

find_txt(tree) =
  match tree with
  | File(n)      -> if ends_with(".txt", n) then [n] else []
  | Dir(_, kids) -> fold( (acc, t) -> acc ++ find_txt(t), [], kids )
```

Python (Datenmodell + Traversal)

```python
from dataclasses import dataclass
from typing import List, Union

@dataclass(frozen=True)
class File:
    name: str

@dataclass(frozen=True)
class Dir:
    name: str
    children: List["Node"]

Node = Union[File, Dir]

def find_txt(node: Node) -> List[str]:
    if isinstance(node, File):
        return [node.name] if node.name.lower().endswith(".txt") else []
    if isinstance(node, Dir):
        # keine Mutation: Listen-Konkatenation in einem Ausdruck
        return [p for child in node.children for p in find_txt(child)]
    return []
```

> Falls du wirklich den echten Dateibaum brauchst: Schreibe **eine** separate IO-Funktion, die das Dateisystem in so einen `Dir/File`-Baum **einliest**; nutze dann **nur** `find_txt` für die Logik.

---

6) „Immutable Updates“ (keine In-Place-Mutation)

Pseudocode

```
update_record(rec, key, val) =
  copy = clone(rec)
  copy[key] = val
  return copy
```

Python

```python
def update_record(rec: dict, key, val):
    return {**rec, key: val}  # neue Dict-Kopie mit überschriebenem Key
```

Listen „anhängen/entfernen“ ohne Mutation

```
def add_item(xs, x):
    return xs + [x]             # neue Liste

def remove_if(pred, xs):
    return [e for e in xs if not pred(e)]
```

7) Komposition & Pipelines

Pseudocode

```
normalize = map( λx -> (x-μ)/σ )
positives = filter( λx -> x > 0 )
total     = fold( (a,b) -> a+b , 0 )

pipeline(xs) = (total ∘ positives ∘ normalize)(xs)
```

Python

```
def normalize(xs, mu, sigma):
    return [(x - mu) / sigma for x in xs]

def positives(xs):
    return [x for x in xs if x > 0]

def total(xs):
    from functools import reduce
    return reduce(lambda a, b: a + b, xs, 0)

def pipeline(xs, mu, sigma):
    return total(positives(normalize(xs, mu, sigma)))
```

8) Pattern-Matching simulieren

Python hat (noch) kein vollwertiges algebraisches Pattern-Matching für Listen wie `x :: xs`. Du simulierst es so:

```
def match_list(xs):
    if not xs:
        # []
        ...
    else:
        head, *tail = xs  # y :: ys
        ...
```

Für Summe, Produkt, etc. brauchst du in Python **keine** explizite Rekursion, wenn eine Reduktion/Comprehension eleganter ist — beides bleibt funktional (keine Mutation).

9) „Imperativ gesehen“ → „funktional übersetzen“ (mentale Schritte)

1. **Schleife** erspäht? → Ist das `map`, `filter` oder `fold`?

2. **Zähler/akkumulator**? → Das ist `fold` mit Anfangswert.
3. **Zweig mit `continue/if` in Schleife**? → Das ist `filter` vor dem `fold` oder innerhalb als Guard.
4. **Zwischenliste per `.append`**? → Ersetze durch Comprehension/`map`.
5. **Sortieren**? → `sorted(xs)` zurückgeben, Original nicht ändern.
6. **IO mitten drin**? → In eigene dünne Funktion auslagern, Kern pure halten.

---

10) Mini-Katalog „Pseudocode → Python“

- `sum_pos_squares(xs)` **Pseudocode:**

  ```
  xs |> filter(λx -> x>0) |> map(λx -> x*x) |> fold(+, 0)
  ```

  Python:

  ```python
  def sum_pos_squares(xs):
      return sum(x*x for x in xs if x > 0)
  ```

- `sum_nested(node)` → siehe Abschnitt 3.
- `compound(P, r, n)` → siehe Abschnitt 4.
- `find_txt(tree)` → siehe Abschnitt 5.

Do / Don’t (schnell merken)

Do

- `sorted`, `sum`, `all`, `any`, `min`, `max`, `map`, `filter`, `reduce`
- Comprehensions (`[f(x) for x in xs if cond]`)
- neue Objekte statt Mutation zurückgeben
- Rekursion/Reduce statt Zählschleifen

Don’t

- `.append`, `.extend`, `.insert`, `.remove`, `.pop`, `.sort` im Kern
- globale Variablen lesen/schreiben
- gemischte IO + Logik in einer Funktion