Lesson 2 – Decisions & Loops: Searching for the Best

From Decisions to Optimization

To optimize, we need to make **choices under rules**. In Python, we do this with:

- **Conditions** (if/else) to decide what is allowed or best.
- **Loops** to test many options and keep the best one.

Big Idea

Optimization = *Check* → *Compare* → *Repeat* → *Keep the Best.*

Conditions: Teaching the Computer to Decide

Comparison & Boolean Operators

- Comparisons: `>`, `<`, `>=`, `<=`, `==`, `!=`
- Logic: `and`, `or`, `not`

Conditions are **questions** that return `True` or `False`.

Example: Quick checks

```
speed = 18
limit = 20
print(speed < limit) # True
print(limit - speed >= 5) # False (20-18=2, not >= 5)
print((speed <= limit) and (speed % 2 == 0)) # True, % is remainder
```

Output

```
True
False
True
```

Example: Feasibility check (budget vs cost)

```
budget = 14
cost = 15
print(cost <= budget) # False, not feasible within budget
```

Output

```
False
```

If, Elif, Else: Transport rule

```
distance_miles = 3.1
if distance_miles < 1:
    print("Walk")
elif distance_miles < 2.5:
    print("Bike")
else:
    print("Bus")
```

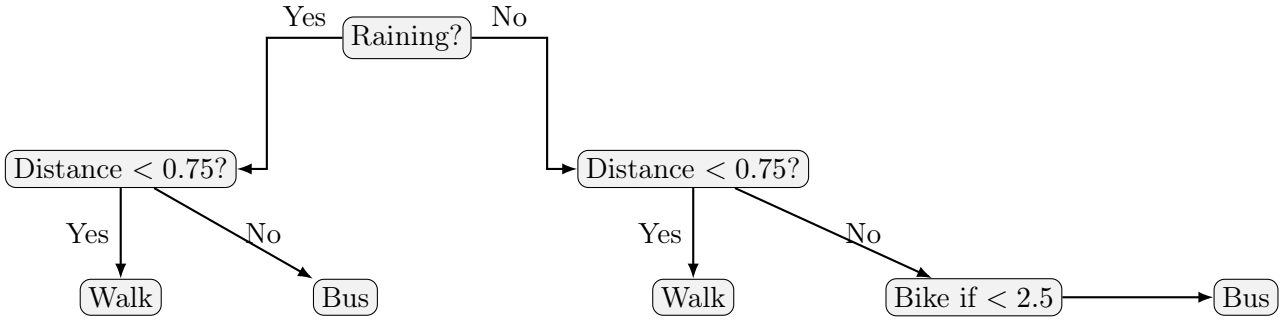
Nested conditions: Weather + Distance

```
raining = True
distance = 1.8

if raining:
    if distance < 0.75:
        print("Walk")
    else:
        print("Bus")
else:
    if distance < 0.75:
        print("Walk")
    elif distance < 2.5:
        print("Bike")
    else:
        print("Bus")
```

Optimization Lens

Rules are **constraints**. Conditions help pick the best *feasible* option.



Decision tree example (weather + distance)

Loops: Repeating to Explore Options

What does a loop do?

A **loop** repeats instructions. We use loops to

1. **visit each item** in a sequence (numbers or list elements),
2. **do something** with that item (print, add, compare), and
3. optionally **remember** a result (running sum, best-so-far, count).

A. The `range()` generator (numbers to loop over)

Example A1: `range(n)` (0 to n-1)

```
for i in range(3):
    print(i)
```

Output

```
0
1
2
```

Example A2: `range(a, b)` (a to b-1)

```
for i in range(2, 7):
    print(i)
```

Output

```
2
3
4
5
6
```

Example A3: `range(a, b, step)` (count by a step)

```
for i in range(10, 0, -2): # 10 down to 2 by 2
    print(i)
```

Output

```
10
8
6
4
2
```

A loop visits each item and does the same steps

Loop idea: same action repeated for each item

B. Looping over lists (real data)

Sometimes we don't want numbers, we want to visit the *items* in a list.

```
times = [18, 13, 16, 11] # minutes of four routes
for t in times:
    print("Route time:", t)
```

Output

```
Route time: 18
Route time: 13
Route time: 16
Route time: 11
```

C. The Accumulation Pattern (sum, count, average)

Pattern recipe

Before the loop, create a **box** (variable) to store a running result (e.g., `total = 0`). Inside the loop, **update** that variable each time. After the loop, **use** it.

```
scores = [72, 88, 91, 79]

total = 0 # 1) start at 0
count = 0

for s in scores: # 2) visit each score
    total += s # add it to the running total
    count += 1 # how many times this loop runs

average = total / count # 3) use the result
print("Total:", total)
print("Average:", average)
```

Output

```
Total: 330
Average: 82.5
```

Trace of `total` *for the list* `[72, 88, 91, 79]`

Seen	Add	New total
(start)	–	0
72	+72	72
88	+88	160
91	+91	251
79	+79	330

D. Best-so-far (min/max) — core optimization move

We scan all options and **remember the best seen so far**.
Find the minimum (fastest route)

```
times = [18, 13, 16, 11] # lower is better

best_time = float('inf') # start impossibly large
best_idx = -1

for i, t in enumerate(times):
    if t < best_time: # found a better (smaller) time
```

```
        best_time = t
        best_idx = i

print("Best route index:", best_idx, "with", best_time, "minutes")
```

Output

```
Best route index: 3 with 11 minutes
```

Find the maximum (most fun)

```
fun = [6, 9, 7, 8] # higher is better

best_fun = -float('inf') # start impossibly small
best_idx = -1

for i, f in enumerate(fun):
    if f > best_fun:
        best_fun = f
        best_idx = i

print("Best activity index:", best_idx, "with fun =", best_fun)
```

Output

```
Best activity index: 1 with fun = 9
```

E. Filter + best-so-far (constraints matter)

Only consider options that satisfy a rule (*feasible*), then pick the best.
Most expensive you can afford (under a budget)

```
prices = [5, 3, 7, 2, 9]
budget = 6

min_price = -1
for p in prices:
    if p <= budget and p > min_price: # feasible AND better
        min_price = p

print("Best affordable price:", min_price)
```

Output

```
Best affordable price: 5
```

Fastest route when it is raining (skip bikes)

```
routes = [("walk", 14), ("bike", 9), ("bus", 8)]
raining = True

best_mode = None
best_time = float('inf')

for mode, t in routes:
    if raining and mode == "bike":
        continue # not feasible in the rain
    if t < best_time:
        best_time = t
        best_mode = mode

print("Take:", best_mode, "Time:", best_time)
```

Output

```
Take: bus Time: 8
```

F. Common pitfalls to avoid

- Forgetting that `range` *excludes* the end (off-by-one errors).
- Using `=` (assign) instead of `==` (compare) in `if` conditions.
- Resetting your running variable *inside* the loop by mistake.
- Missing colons `:` after `for`, `if`, `elif`, `else`.

Lesson 2 Practice Exercises

1. Write a program that prints `"Even"` if a number `n` is even, else prints `"Odd"`. Test with 7 and 22.
2. With `budget = 20` and `prices = [8, 12, 5, 15, 4]`, print `"buy"` if the price is within budget, otherwise `"skip"`.
3. Extend the transport rules: if `raining = True` → `"Bus"`; otherwise use distance: < 1 mile → Walk, < 2.5 miles → Bike, else → Bus.
4. Use a loop to print the numbers from 5 to 15 (inclusive).
5. Compute and print the average of `temps = [59, 63, 71, 68, 64]`.
6. For `miles = [0.6, 1.1, 2.3, 3.2]`, print the right mode (Walk/Bike/Bus) for each distance.

7. For `times = [14, 12, 17, 9, 13]`, find the index of the fastest route using a loop (do not use `min`).
8. For `ratings = [3, 5, 4, 5, 2]`, find the index of the highest rating (choose the first if there's a tie).
9. With `prices = [5, 7, 3, 6, 9]` and `budget = 8`, print the most expensive item you can afford.
10. **Challenge:** With items `[(3,4), (4,5), (6,6), (2,2)]` as (price, fun) and `budget = 10`, pick two different items that maximize fun without going over budget.