Seed Programming

Python Fundamentals

Lecture 04: Grader, Min/Max, and Calculator

1 Learning Objectives

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

- Combine conditions with the `and` and `or` operators.
- Insert variable values into formatted strings.
- Distinguish indentation errors from logical output errors.
- Develop an efficient running maximum and minimum algorithm.
- Explain why `elif` can be preferable to separate `if` statements.
- Create reusable grader and calculator functions.
- Match function arguments with parameters in the correct order.
- Combine several tasks in a menu-driven program.

2 The `and` and `or` Operators

The `and` and `or` operators combine multiple conditions into one Boolean result.

- **`and`:** the complete expression is `True` only when every individual condition is `True`.
- **`or`:** the complete expression is `True` when at least one individual condition is `True`.

```
A = 10
B = 5
C = 20

print(A > B and A > C)
print(A > B or A > C)
```

Output

```
False
True
```

In the first expression, `A > B` is true but `A > C` is false. The `and` operator requires both conditions to be true, so the result is false. The `or` operator needs only one true condition, so the second result is true.

2.1 Truth Table

P	Q	P and Q	P or Q
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Use `and` when every condition must hold at the same time, such as checking whether one value is greater than all the other values. Use `or` when any one condition is sufficient, such as detecting an

invalid mark:

```
if Marks > 100 or Marks < 0:
    return "I"
```

A mark is invalid when it is too high *or* too low. Using `and` would be incorrect because no number can be greater than `100` and less than `0` at the same time.

3 Formatted Strings and Indentation

An *f-string* inserts variable values or expressions directly into a string. Place the letter `f` before the opening quotation mark and place each expression inside curly braces.

```
Max = 25
print(f"The Max is: {Max}")
```

Output

```
The Max is:  25
```

Python evaluates the expression inside the braces, converts its value to text, and inserts it at that position.

3.1 Indentation Error

The following program contains a different problem: the statements below the two `if` conditions are not indented.

```
A = int(input("N1: "))
B = int(input("N2: "))

Max = 0
if A > B:
Max = A
if B > A:
Max = B

print(f"The Max is: {Max}")
```

Python raises an `IndentationError` because every `if` statement requires at least one indented statement in its body. The corrected version is:

```
A = int(input("N1: "))
B = int(input("N2: "))

Max = 0
if A > B:
    Max = A
if B > A:
    Max = B

print(f"The Max is: {Max}")
```

3.2 Missing Curly Braces

Forgetting the braces causes a logical output error rather than a crash:

```
Max = 25
print(f"The Max is: Max")
```

Output

```
The Max is:  Max
```

Without braces, Python treats `Max` as ordinary characters. This mistake can be harder to notice because the program runs successfully but displays a misleading result.

Remark. *Indentation errors stop the program immediately. Missing braces in an f-string usually do not cause an exception, so formatted output should always be checked visually.*

4 Finding the Maximum

The maximum-finding program can be improved in several stages. Each stage reveals a general programming idea: reducing comparisons, removing repetition, and selecting a safe initial value.

Before diving into code, think about how a person would naturally find the largest value in a long list of numbers. We wouldn't compare every number against every other number — instead, we'd remember the first number as the "largest so far," then look at each new number and only update our answer if it's bigger. This means only **one value** is kept in memory at any time, and only **one comparison** is needed per new number — the same method works whether there are 5 numbers or 5,000. This is exactly the logic we want our code to follow.

4.1 Version 1: Brute-Force Comparison

One direct approach compares every candidate with every other candidate:

```
A = int(input("N1: "))
B = int(input("N2: "))
C = int(input("N3: "))
D = int(input("N4: "))
E = int(input("N5: "))

Max = 0
if A > B and A > C and A > D and A > E:
    Max = A
if B > A and B > C and B > D and B > E:
    Max = B
if C > A and C > B and C > D and C > E:
    Max = C
if D > A and D > B and D > C and D > E:
    Max = D
else:
    Max = E

print(f"The Max is: {Max}")
```

This approach is lengthy and inefficient. Every candidate is repeatedly compared with the other values, and the amount of code grows quickly as more inputs are added.

4.2 Version 2: Using `elif` — Removing Unnecessary Conditions

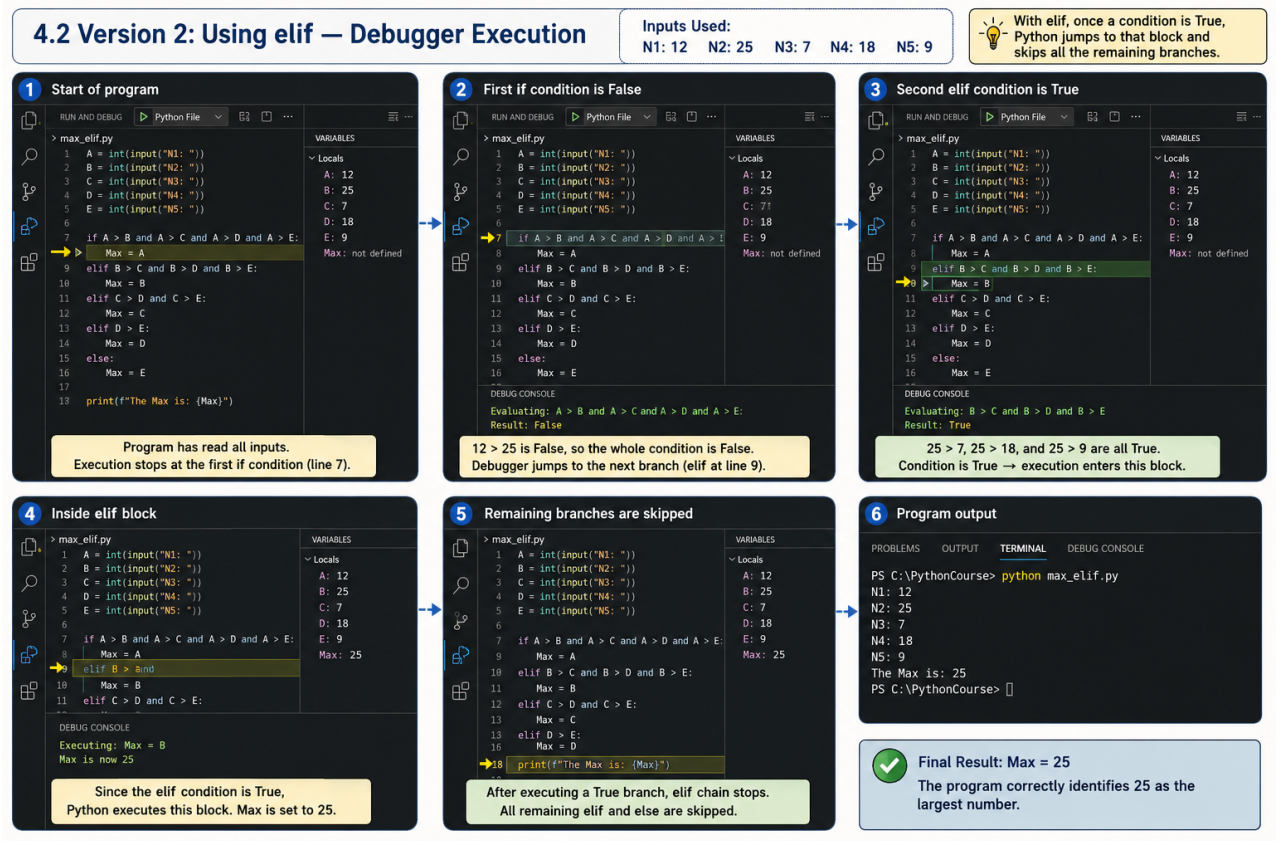
An `elif` chain stops as soon as one condition becomes true, so comparisons already ruled out by earlier failed branches can be removed entirely:

```
A = int(input("N1: "))
B = int(input("N2: "))
C = int(input("N3: "))
D = int(input("N4: "))
E = int(input("N5: "))

if A > B and A > C and A > D and A > E:
    Max = A
elif B > C and B > D and B > E:
    Max = B
elif C > D and C > E:
    Max = C
elif D > E:
    Max = D
else:
    Max = E

print(f"The Max is: {Max}")
```

Separate `if` statements all have to be checked. With `elif`, Python skips the remaining branches after finding a true condition. Comparisons that have already been implied by earlier failed branches can also be removed — for example, the `B` branch needs to compare `B` only with `C`, `D`, and `E`, since we already know `A` is not bigger than `B` (otherwise the first branch would have run).



4.3 The Negative-Numbers Bug

Now try running Version 2 with **all negative inputs**: `N1: -5`, `N2: -10`, `N3: -3`, `N4: -8`, `N5: -1`.

Output

```
The Max is:  0
```

This is **wrong** — `0` was never one of the numbers entered, yet the program reports it as the maximum.

Dry run — why did this happen?

Step	Check	Result
Before checks	`Max` starts at `0`	—
`if Max < A:` (N1 = -5)	`0 < -5`	`False` — `Max` stays `0`
`if Max < A:` (N2 = -10)	`0 < -10`	`False` — `Max` stays `0`
`if Max < A:` (N3 = -3)	`0 < -3`	`False` — `Max` stays `0`
`if Max < A:` (N4 = -8)	`0 < -8`	`False` — `Max` stays `0`
`if Max < A:` (N5 = -1)	`0 < -1`	`False` — `Max` stays `0`

The issue: initializing `Max = 0` assumes `0` is a safe "low" starting point. But since every actual input is negative, and every negative number is smaller than `0`, the condition `Max < A` is **always false** — so `Max` never gets updated and incorrectly stays at its starting value of `0`, which was never even one of the numbers entered.

4.4 The Solution

The safe solution is to initialize the maximum with the **first real input**, instead of an arbitrary guess like `0`:

```
A = int(input("N1: "))
Max = A
```

Each later number is then compared with this running maximum:

```
A = int(input("N2: "))
if Max < A:
    Max = A

A = int(input("N3: "))
if Max < A:
    Max = A

A = int(input("N4: "))
if Max < A:
    Max = A

A = int(input("N5: "))
if Max < A:
    Max = A

print(f"The Max is: {Max}")
```

This is the same running "largest value so far" strategy used when scanning a list manually — and it now handles negative numbers correctly, since `Max` always starts as a genuine value from the input rather than an assumed `0`.

4.5 Using a `for` Loop — The Final Version

Notice the block above repeats the same two lines (`input` + `if Max < A: Max = A`) over and over — once for every number after the first. This repetition is a sign that we should use a **loop** instead of writing it out manually.

We use a `for` loop to remove this duplication, and we track the **minimum** during the same pass by applying the mirrored comparison (`Min > A` instead of `Max < A`). The whole thing is also wrapped in a function so it can be called from a menu:

```
def TaskMinMaxFinder():
    H = int(input("How many numbers: "))
    A = int(input("N1: "))

    Max = A
    Min = A

    for i in range(2, H + 1):
        A = int(input(f"N{i}: "))

        if Max < A:
            Max = A
        if Min > A:
            Min = A

    print(f"The Max is: {Max}")
    print(f"The Min is: {Min}")
```

Important details are:

- `H` lets the user decide how many numbers to enter — no longer hardcoded to 5.
- The first number initializes both `Max` and `Min`, so negative inputs are handled correctly.
- `range(2, H + 1)` begins at the second number because the first has already been processed.
- Both values are updated inside one loop, so the data is scanned only once.
- Wrapping the logic in `TaskMinMaxFinder()` makes it reusable from a menu.

5 The Grader Function

The grader receives a mark and returns a grade letter rather than printing directly. Returning a value makes the function reusable in different programs.

5.1 Version 1: Full Conditions with `and`

In the first version, every grade range is checked completely. Two conditions joined with `and` confirm that the mark is above the lower bound and at or below the upper bound.

```
def Grader(Marks):
    if Marks > 100 or Marks < 0:
        return "I"
    elif Marks <= 100 and Marks > 90:
        return "A+"
    elif Marks <= 90 and Marks > 80:
        return "A"
    elif Marks <= 80 and Marks > 70:
        return "B"
    elif Marks <= 70 and Marks > 60:
        return "C"
    elif Marks <= 60 and Marks > 50:
        return "D"
    else:
        return "R"
```

Each `elif` re-checks both ends of its range. For example, `Marks <= 100 and Marks > 90` confirms that the mark is not above `100` and is greater than `90`. However, the first invalid-input condition has already established that a valid mark cannot exceed `100`. Repeating that upper-bound check makes the later conditions longer than necessary.

5.2 Version 2: Removing the Unnecessary Condition

In an `elif` chain, Python checks a later condition only when every earlier condition has failed. By the time execution reaches `elif Marks > 90`, Python already knows that `Marks > 100` was false. The upper bound can therefore be removed safely, leaving only the lower bound:

```
def Grader(Marks):
    if Marks > 100 or Marks < 0:
        return "I"
    elif Marks > 90:
        return "A+"
    elif Marks > 80:
        return "A"
    elif Marks > 70:
        return "B"
    elif Marks > 60:
        return "C"
    elif Marks > 50:
        return "D"
    else:
        return "R"
```

This version is more efficient because:

- Each grade condition uses one comparison instead of two.
- The logic is shorter and easier to read and maintain.
- Fewer comparison symbols reduce the chance of writing a typo such as `<=` instead of `<`.
- The grade boundaries and behavior remain exactly the same as Version 1; only redundant checks have been removed.

This is the same idea used by the maximum-finder `elif` chain in Section 4.3: once earlier conditions have failed, some comparisons are already known and do not need to be repeated.

The invalid test still uses `or`: a mark is invalid if it is above `100` or below `0`. Using `and` would be incorrect because no number can be greater than `100` and less than `0` at the same time.

5.3 Handling Invalid Input with `while`

```
def TaskGrader():
    Marks = int(input("Enter Marks: "))
    Grade = Grader(Marks)

    while Grade == "I":
        Marks = int(input("Invalid Input\nEnter Marks again: "))
        Grade = Grader(Marks)

    print(f"You have got {Grade} grade...")
```

The loop keeps requesting a new mark for as long as the returned grade is `"I"`. Unlike a single `if` statement, it can handle several invalid attempts in a row.

6 The Calculator Function

The calculator accepts two numbers and an operator. It selects one operation and returns the result.

```
def Calculator(N1, N2, Operation):
    Result = 0

    if Operation == "+":
        Result = N1 + N2
    elif Operation == "-":
        Result = N1 - N2
    elif Operation == "*":
        Result = N1 * N2
    elif Operation == "/":
        Result = N1 / N2
    elif Operation == "//":
        Result = N1 // N2
    elif Operation == "%":
        Result = N1 % N2

    return Result
```

The `elif` chain ensures that only the matching operation runs.

6.1 A Closer Look at Function Calls

```
N1 = 7
N2 = 2
Opr = "*"

answer = Calculator(N1, Opr, N2)
print(f"{N1}{Opr}{N2}={answer}")
```

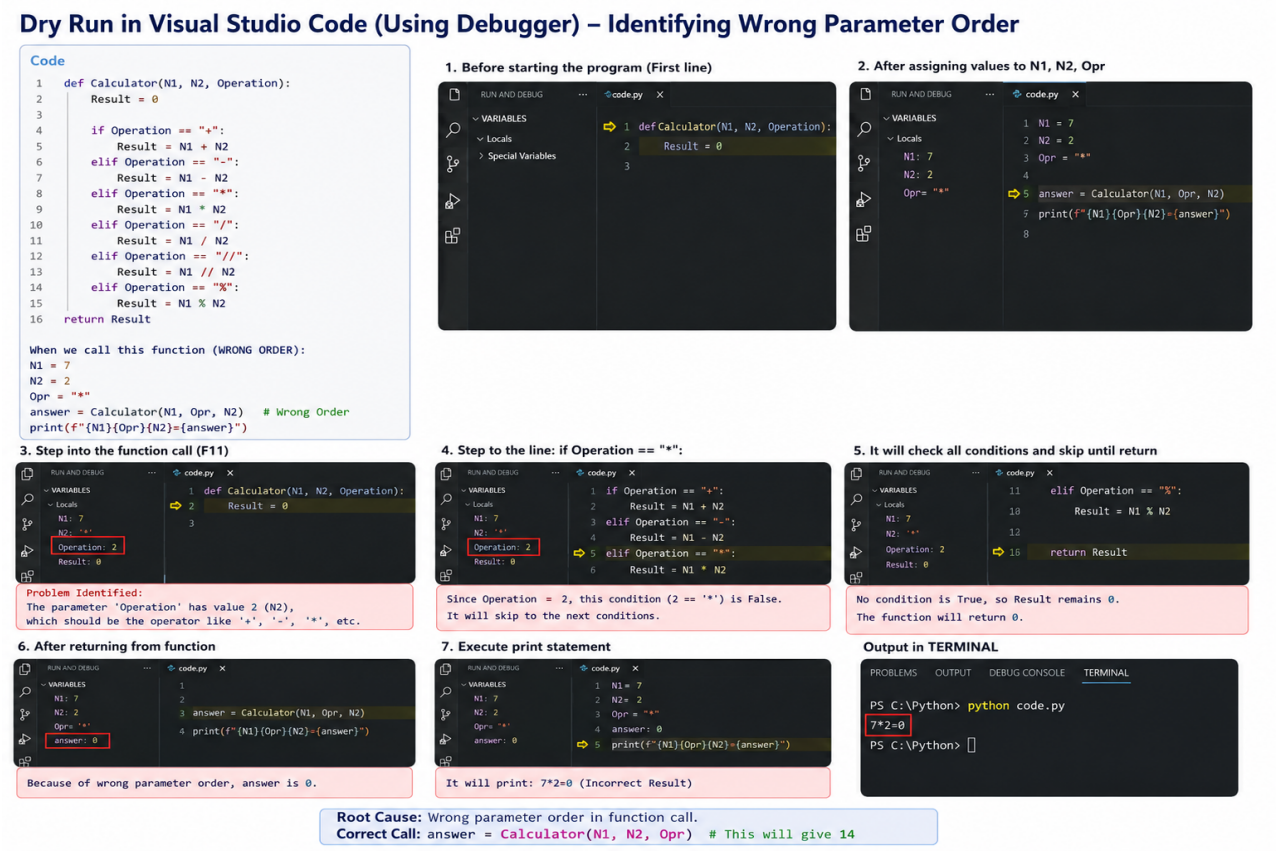
Output

```
7*2=0
```

This is **not correct**. The expression `7 * 2` should produce `14`, not `0`. The program ran without an

error message, so something inside the logic is silently wrong.

Let us find out with the help of the debugger.



Stepping through the call in the debugger lets us watch the value that each parameter receives as soon as `Calculator()` is entered:

Parameter in function	Value received	Expected
`N1`	7	7 (correct)
`N2`	`"*"`	2 (incorrect)
`Operation`	2	`"*"` (incorrect)

Debugging reveals the problem: the arguments were supplied in the wrong order. The call is written as `Calculator(N1, Opr, N2)`, but the function is defined as `Calculator(N1, N2, Operation)`. Python matches arguments to parameters purely by position. Therefore, `Opr` fills the `N2` parameter and `N2` fills the `Operation` parameter.

Because `Operation` now contains `2` instead of `"*"`, none of the `elif` conditions for `"+"`, `"-"`, `"*"`, `"/"`, `"//"`, or `"%"` matches. Every branch is skipped, and the function returns the initial value of `Result: 0`.

The program does not crash because this is a *logical error*, not a syntax error. It runs "successfully" while quietly producing the wrong answer.

6.2 The Solution

The solution is to pass the arguments in the same order expected by the function: `N1`, then `N2`, and then `Operation`.

```
answer = Calculator(N1, N2, Opr)  # Correct order
print(f"{N1}{Opr}{N2}={answer}")
```

Output

```
7*2=14
```

7 Menu-Based Program

The three tasks can be combined in a menu-driven loop. The program continues until the user selects the exit option.

```
from os import system
from time import sleep

while True:
    system("cls")

    print("1. Max/Min Finder")
    print("2. Grader")
    print("3. Calculator")
    print("E. Exit")

    choice = input("Choice: ")

    if choice == "1":
        TaskMinMaxFinder()
    elif choice == "2":
        TaskGrader()
    elif choice == "3":
        TaskCalculator()
    elif choice == "E" or choice == "e":
        break
    else:
        print("Wrong Choice...")

    sleep(2)
```

The menu works as follows:

- `while True` keeps the menu running until `break` is executed.
- `system("cls")` clears a Windows console before redrawing the menu.
- Each choice calls one task function.
- `choice == "E" or choice == "e"` accepts either uppercase or lowercase input for exiting.
- The `else` branch handles any unsupported choice without crashing.
- `sleep(2)` gives the user time to read the result before the screen clears.

This structure connects `TaskMinMaxFinder()`, `TaskGrader()`, and `TaskCalculator()` in one interactive program.

8 Summary

Students should now be able to:

- Select `and` when all conditions must be true and `or` when any one true condition is sufficient.
- Use curly braces to insert values into f-strings.
- Correct indentation errors and recognize silent output mistakes.
- Replace brute-force comparisons with a running maximum or minimum.
- Initialize a maximum or minimum from real input instead of assuming `0`.
- Return grades and calculator results from reusable functions.
- Match arguments with function parameters in the correct order.
- Build a menu-driven application from several task functions.