Chapter 3: Conditional Execution

Theory Lecture (Classroom)

Chapter Roadmap

In this chapter we study **conditional execution** in Python:

- Boolean expressions
- Logical operators
- Basic `if` statement (one-way decision)
- `if/else` (two-way decision)
- Chained conditionals (`if/elif/else`)
- Nested conditionals
- Handling errors using `try/except`
- Short-circuit evaluation and guardian pattern
- Debugging conditional code

Learning Outcomes

After this theory lecture, I should be able to:

- Explain what a **boolean expression** is and how comparison operators work.
- Use `and`, `or`, and `not` to combine conditions.
- Describe the structure and syntax of `if`, `if/else`, and `if/elif/else`.
- Explain the idea of **nested** conditionals and when to avoid deep nesting.
- Describe the purpose of `try/except` for handling runtime errors.
- Explain **short-circuit evaluation** and the **guardian pattern**.
- Read and interpret basic traceback messages related to conditionals.

3.1 Boolean Expressions

Boolean expression: an expression whose value is either `True` or `False`.

Examples of comparisons:

- `5 == 5` $\Rightarrow$ `True`
- `5 == 6` $\Rightarrow$ `False`

Important points:

- `True` and `False` are special values of type `bool`.
- They are **not** strings; no quotes around them.

Comparison Operators

Python comparison operators:

- `x == y` x is equal to y
- `x != y` x is not equal to y
- `x > y` x is greater than y
- `x < y` x is less than y
- `x >= y` x is greater than or equal to y
- `x <= y` x is less than or equal to y
- `x is y` x is the same object as y (identity)
- `x is not y` x is not the same object as y

Common beginner mistake:

- Using = instead of ==.
- = is **assignment**, == is **comparison**.
- There is no =< or => in Python.

Boolean Expressions (Small Example)

```
>>> 5 == 5
True
>>> 5 == 6
False
>>> type(True)
<class 'bool'>
```

Key idea:

- We will use such boolean expressions as **conditions** in `if` statements.

3.2 Logical Operators

Logical operators combine boolean expressions:

- `and` both conditions must be true
- `or` at least one condition must be true
- `not` negates a condition

Conceptual examples:

- `x > 0 and x < 10`
  True only if x is greater than 0 *and* less than 10.
- `n % 2 == 0 or n % 3 == 0`
  True if n is divisible by 2 *or* by 3 (or both).
- `not (x > y)`
  True when `x > y` is false.

Logical Operators (Small Example)

```
>>> x = 1
>>> y = 2
>>> x > y
False
>>> not (x > y)
True
```

Notes:

- Python allows non-boolean operands as well (any non-zero number is “true”), but
- For clarity in beginner code, we usually keep logical operators on boolean expressions.

3.3 Conditional Execution: if

Most useful programs need to **check conditions** and choose actions accordingly.

Basic form:

- `if` *condition* :
- one or more **indented** statements (the body)

Key ideas:

- The condition is a boolean expression.
- If the condition is `True`, Python executes the body.
- If the condition is `False`, Python skips the body.

Basic if (Example)

```
if x > 0:
    print("x is positive")
```

Conceptually:

- The header line ends with a colon :.
- The body is indented under the header (same indentation for all lines in body).
- This is a **compound statement** (header + indented body).

There must be at least one statement in the body; if we need a placeholder, we can use:

```
if x < 0:
    pass    # do nothing for now
```

Blocks and Indentation

Important details:

- Indentation in Python is **syntactic** (not just style).
- All statements in the body of an `if` must be indented the same amount.
- A block ends when indentation returns to the previous level.

Interactive mode note:

- In the Python shell, after typing an `if` header, the prompt changes to `...`
- You finish the block with a blank line.

3.4 Alternative Execution: if/else

Sometimes there are two alternative actions:

- One if the condition is true.
- Another if the condition is false.

Structure:

- `if` *condition* :
  body when condition is true
- `else:`
  body when condition is false

Exactly one of the two branches executes.

if/else (Example)

```
if x % 2 == 0:
    print("x is even")
else:
    print("x is odd")
```

Concept:

- If x has remainder 0 when divided by 2, it is even.
- Otherwise (else), x is odd.
- These are the two **branches** of the conditional.

3.5 Chained Conditionals: if/elif/else

Sometimes there are more than two possibilities.

General structure:

- `if` *condition1* : $\Rightarrow$ branch 1
- `elif` *condition2* : $\Rightarrow$ branch 2
- `elif` *condition3* : $\Rightarrow$ branch 3
- `else:` $\Rightarrow$ default branch (optional)

Properties:

- Conditions are checked **top to bottom**.
- As soon as one condition is true, its branch executes and the chain ends.
- At most one branch is executed.

Chained Conditional (Example)

```
if x < y:
    print("x is less than y")
elif x > y:
    print("x is greater than y")
else:
    print("x and y are equal")
```

Idea:

- Exactly one of the three messages will be printed.
- The `else` catches all remaining cases (here: equality).

3.6 Nested Conditionals

Nested conditional: an `if` (or `if/else`) that appears inside the body of another `if` or `else`.

Conceptually:

- Allows multi-level decision structures.
- Each inner conditional is only considered if the outer condition(s) allow it.

Example idea:

- First check if two values are equal.
- If not equal, check which one is smaller inside the `else` branch.

Nested Conditional (Example Shape)

```
if x == y:
    print("x and y are equal")
else:
    if x < y:
        print("x is less than y")
    else:
        print("x is greater than y")
```

Notes:

- Indentation shows the structure.
- Deep nesting quickly becomes hard to read.
- Often we can rewrite nested conditionals using logical operators or chained conditionals (`elif`).

3.7 Exceptions and try/except

Problem:

- Some operations may fail at runtime (e.g., converting text to number).
- In a script, such an error stops the program immediately.

Solution:

- Use `try/except` to **catch exceptions**.
- Allows us to handle errors more gracefully.

Concept:

- `try:` block: code that may cause an error.
- `except:` block: code that runs if an error occurs in the try block.

try/except (Temperature Example)

```
inp = input("Enter Fahrenheit Temperature: ")
try:
    fahr = float(inp)
    cel = (fahr - 32.0) * 5.0 / 9.0
    print(cel)
except:
    print("Please enter a number")
```

Explanation:

- If `inp` can be converted to `float`, we compute and print Celsius.
- If conversion fails (e.g., user types `"fred"`), the `except` block runs.
- The program does not crash; it prints a friendly message instead.

Catching Exceptions

Key ideas:

- **Catching an exception** means intercepting an error and handling it.
- We can:
  - Show a clear error message.
  - Ask the user to try again.
  - End the program gracefully.
- `try/except` acts like an **insurance policy** around risky code.

3.8 Short-Circuit Evaluation

When evaluating logical expressions such as:

$$x \geq 2 \texttt{ and } (x/y) > 2$$

Python:

- Evaluates from left to right.
- For `and`, if the left side is `False`, the whole expression is `False`.
- In that case, it **does not evaluate** the right-hand side.

This early stopping is called **short-circuiting** the evaluation.

Short-Circuit Example (Concept)

```
x >= 2 and y != 0 and (x / y) > 2
```

- If `x >= 2` is false, Python stops (result is false).
- If `x >= 2` is true but `y != 0` is false, Python again stops.
- Only if both are true, Python evaluates `(x / y) > 2`.

If we write conditions in a careful order, we can **avoid errors**, such as division by zero.

Guardian Pattern

Guardian pattern: a technique where we:

- Place a safe check (the "guardian") first in an `and` expression.
- Rely on short-circuit evaluation to prevent dangerous operations.

Example idea:

- Check `y != 0` before computing `x / y`.
- This ensures the division happens only when it is safe.

This pattern is especially useful when checking:

- Length of lists before indexing.
- Denominators before division.

3.9 Debugging Conditional Code

When a program fails, Python prints a **traceback**:

- Tells us what kind of error occurred.
- Shows where in the code it was detected.

Common issues in conditional code:

- **Syntax errors:** missing colon, wrong indentation.
- **Name errors:** typo in a variable name.
- **Logical errors:** condition is wrong, so wrong branch executes.

Indentation and Whitespace Errors

Whitespace errors are sometimes tricky:

- Indenting where we should not (unexpected indent).
- Mixing tabs and spaces.
- Error message may point at a line, but the real mistake is earlier.

General advice:

- Check the lines *before* the line mentioned in the error.
- Keep indentation consistent (e.g., 4 spaces).

Debugging Strategy

When an error or wrong behavior appears:

1. Read the traceback carefully (type of error and line number).
2. Check the relevant condition(s): are they what you really mean?
3. Add temporary `print` statements to see values and which branch executes.
4. Simplify complex or nested conditions if necessary.
5. Consider using the guardian pattern for risky operations.

Key Concepts (Glossary Style)

Boolean expression An expression whose value is `True` or `False`.

Conditional statement Controls the flow of execution depending on a condition.

Branch One of the alternative paths in a conditional.

Chained conditional Multiple branches using `if/elif/else`.

Nested conditional A conditional that appears inside another conditional.

Logical operator `and`, `or`, `not`.

Traceback The error report Python prints when an exception occurs.

Short circuit Stopping evaluation of a logical expression when the result is already known.

Guardian pattern Using a safe check at the start of a logical expression to prevent errors.

Where We Use Conditionals

After this chapter, you should see conditionals in many contexts:

- Validating user input (e.g., range checks).
- Grading schemes (e.g., A/B/C/D/F based on score).
- Payment calculations (e.g., overtime rules).
- Error handling with `try/except`.
- Safe use of division and indexing with guardian pattern.

These are foundational tools for all later programming topics.

End of Chapter 3 (Theory)

- We have focused on **concepts** and **structures**:
  - boolean expressions, logical operators,
  - `if`, `if/else`, `if/elif/else`, nested `if`,
  - `try/except`, short-circuit, guardian pattern,
  - terminology and debugging mindset.
- In lab, we will apply these ideas by:
  - implementing pay calculations,
  - input validation,
  - and small decision-making programs.

Questions?