# MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 6

DATE: 22/08/2025

INSTRUCTOR: ORANGZAIB RAJPOOT

# DIRECTING THE FLOW: CONDITIONAL LOGIC & ERROR HANDLING

We've learned to store and organize data. Today, we give our programs a brain. We'll learn how to make decisions with if/elif/else, creating dynamic programs that respond to different situations. Then, we'll give our programs a safety net, learning how to handle unexpected errors gracefully with try...except instead of crashing.

# TODAY'S AGENDA

- **The Crossroads - if and else Statements**
  - The Concept of Control Flow
  - The if Statement: The Basic Decision
  - Python's "Truthiness": Beyond True and False
  - The else Statement: The Alternative Path
  - *Interactive Exercises on basic decisions*
- **Handling Multiple Paths & Complex Logic**
  - The elif Statement: Avoiding Messy Nesting
  - Building Complex Conditions with and, or, not
  - Nested Conditionals: When a Decision Depends on Another
  - *Interactive Exercises on grading and access control*
- **Pythonic Conditionals, Error Handling & The Lab**
  - The Ternary Operator: A Concise if/else
  - **The Problem: When Programs Crash**
  - **The Solution: Handling Errors with try...except**
  - **Hands-On Lab: The Number Guessing Game**
  - Q&A and Wrap-up

# THE CROSSROADS - IF AND ELSE

## THE CONCEPT OF CONTROL FLOW

- Think of a program as a recipe. So far, we've only written recipes that are a straight list of steps. But most real-world instructions involve decisions: "If the mix is too dry, add more water, otherwise, continue."
- Control flow statements allow us to direct the path of execution in our code based on certain conditions. The if statement is the most fundamental tool for this.

# THE IF STATEMENT: THE BASIC DECISION

The if statement evaluates a condition. If that condition is True, the indented block of code below it is executed. If it's False, the block is skipped.

```
# Syntax: if condition:
#             indented_code_block

temperature = 35

if temperature > 30:
    print("It's a hot day!")
    print("Remember to stay hydrated.")

print("The program continues here, regardless of the temperature.")
```

# PYTHON'S "TRUTHINESS"

In Python, you don't need an explicit == True in your condition. Many values are considered "truthy" or "falsy" on their own. This leads to cleaner, more readable code.

**Everything else is Truthy!**

**Falsy Values (behave like False):**

- The number 0 or 0.0
- An empty string "“
- An empty list [], tuple (),
- dictionary {}, or set set()
- The special value None

```
# Un-Pythonic way
user_list = []
if len(user_list) == 0:
    print("No users found.")

# Pythonic way using Truthiness
if not user_list: # "if the list is not truthy (i.e., it's empty)"
    print("No users found.")

name = input("Enter your name: ")
if name: # "if the name is truthy (i.e., not an empty string)"
    print(f"Welcome, {name}!")
```

# THE ELSE STATEMENT: THE ALTERNATIVE PATH

The else statement provides a block of code to execute if the if condition is False.

```
age = 17

if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not yet eligible to vote.")
    years_left = 18 - age
    print(f"You can vote in {years_left} year(s).")
```

# IN-CLASS EXERCISE: LOGIN CHECK

- Create a variable password = "python123".
- Ask the user to enter a password using input().
- Write an if/else statement that checks if the user's input matches the stored password.
- Print "Access Granted" or "Access Denied"

# HANDLING MULTIPLE PATHS & COMPLEX LOGIC

## THE ELIF STATEMENT: THE "ELSE IF"

- What if you have more than two possible outcomes? You could nest if/else statements, but that gets messy fast. The elif (else if) statement is the clean solution. Python checks each condition in order and executes the *first* one that is True.

```
# Messy nested version
score = 85
if score >= 90:
    print("Grade: A")
else:
    if score >= 80:
        print("Grade: B")
    else:
        if score >= 70:
            print("Grade: C")
        # ...and so on
```

```
# Clean, readable elif version
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
elif score >= 70:
    print("Grade: C")
elif score >= 60:
    print("Grade: D")
else:
    print("Grade: F")
```

# SAFE ACCESS WITH .GET()

The .get() method is the preferred way to access data safely. It returns the value if the key exists, or None (a special null value) if it doesn't. You can also provide a default value.

```
# Safe access
location = user.get("location")
print(location) # Output: None

# Safe access with a default value
location = user.get("location", "Location not specified")
print(location) # Output: Location not specified
```

# BUILDING COMPLEX CONDITIONS WITH AND, OR, NOT

You can combine checks to create powerful, specific rules.

```
age = 25
has_license = True

# 'and' requires ALL conditions to be True
if age >= 25 and has_license:
    print("You are eligible to rent a car.")

day = "Sunday"
is_holiday = False

# 'or' requires AT LEAST ONE condition to be True
if day == "Saturday" or day == "Sunday" or is_holiday:
    print("Enjoy your day off!")
```

# IN-CLASS EXERCISE: THEME PARK RIDE ACCESS

A ride has the following rules:

- You must be at least 140cm tall.
- If you are between 120cm and 140cm, you must be accompanied by an adult.
- If you are under 120cm, you cannot ride.

Write an if/elif/else structure that takes a height and a boolean is_with_adult and prints "Access Granted", "Must be with an adult", or "Access Denied"

# PYTHONIC CONDITIONALS

## THE TERNARY OPERATOR: A CONCISE IF/ELSE

- For simple assignments that depend on a condition, Python offers a clean one-line syntax.
- **Syntax:** value_if_true if condition else value_if_false
- Guideline: Use this for clarity on simple assignments, not for complex logic with multiple steps.

```
# Standard if/else
age = 22
if age >= 18:
    status = "Adult"
else:
    status = "Minor"

# Ternary operator version
status = "Adult" if age >= 18
else "Minor"

print(f"The person is a {status}.")
```

# THE PROBLEM: WHEN PROGRAMS CRASH

- So far, we've assumed the user will always give us the input we expect. What happens when they don't?
- When this code runs, the program halts and displays a ValueError.

```
user_input = input("Please enter your age:
")
# User types "twenty" instead of 20

age = int(user_input) # This line will
CRASH!

print("This line will never be reached.")
```

# THE SOLUTION: HANDLING ERRORS WITH TRY...EXCEPT

The try...except block is Python's mechanism for error handling. It's another form of control flow.

**Analogy:** You "try" to run some risky code. If an error occurs, the program doesn't crash. Instead, it jumps to the "except" block, which acts as a safety net.

# SYNTAX

```
try:
    # --- Risky code goes here ---
    # This is the code that might cause an error.
except ErrorType:
    # --- Fallback code goes here ---
    # This block only runs if the specific ErrorType occurs.
```

```
user_input = input("Please enter your age: ")

try:
    age = int(user_input)
    print(f"In five years, you will be {age + 5} years old.")
except ValueError:
    print("Invalid input! Please enter a number using digits,
not words.")

print("Program continues gracefully.")
```

# IN-CLASS EXERCISE: SAFE DIVISION CALCULATOR

- Ask the user for a numerator and a denominator.
- Convert them to floats.
- The division operation will crash if the denominator is 0 (a ZeroDivisionError).
- Wrap your division and print statement in a try...except block to catch the ZeroDivisionError and print a friendly message like "Cannot divide by zero."

# HANDS-ON LAB: THE NUMBER GUESSING GAME

- This project will combine everything: variables, operators, user input, type casting, conditional logic, loops, and error handling.
- **The Goal:** The program will think of a secret number between 1 and 100. The user has to guess the number. The program will provide feedback ("too high" or "too low") until the user guesses correctly.
- Create a new file guessing_game.py.

# PART 1: SETUP

- Import the random library at the top of your file.
- Generate a secret_number between 1 and 100 and store it.
- Print a welcome message to the user, explaining the game.

# PART 2: THE GAME LOOP

- Start an infinite loop with while True:. All the following steps should be indented inside this loop.
- Prompt the user for their guess using input().

# PART 3: MAKING IT ROBUST WITH TRY...EXCEPT

- Inside the loop, start a try block.
- Inside the try block, convert the user's input string to an integer using int().
- After the try block, add an except ValueError: block.
- Inside the except block, print a message like "That's not a valid number! Please try again." and use the continue keyword to restart the loop immediately.

# PART 4: THE LOGIC

- **After** the try...except block (but still inside the loop), write your if/elif/else block to compare the user_guess to the secret_number.
- **if** the guess is less than the secret number, print "Too low! Guess again."
- **elif** the guess is greater than the secret number, print "Too high! Guess again."
- **else** (this means they must have guessed correctly), print a congratulatory message like "You got it! The number was X."
- After the congratulatory message, use the break keyword to exit the while loop and end the game.

# CHALLENGE / BONUS FEATURES:

- **Turn Counter:** Create a variable guess_count before the loop starts. Increment it by 1 every time the user makes a *valid* guess. When they win, tell them how many guesses it took.
- **Limit the Number of Guesses:** Give the user only 7 tries.
  If guess_count reaches 7 and they still haven't guessed the number, end the game and reveal the secret number.