MODULE 1: CORE PYTHON & DATA

WEEK: 1 LECTURE: 7

DATE: 25/08/2025

INSTRUCTOR: ORANGZAIB RAJPOOT

THE POWER OF REPETITION:
FOR AND WHILE LOOPS

We've learned how to make our programs smart with conditional logic. Today, we'll learn how to make them powerful and efficient by teaching them how to repeat tasks. Loops are the workhorses of programming, allowing us to process thousands of data points, wait for user input, or run a game until it's over, all with just a few lines of code.

TODAY'S AGENDA

- **The for Loop & Pythonic Iteration**
  - The "Why": Automating Repetitive Tasks
  - The for Loop: Python's "For Each" Powerhouse
  - The range() Function: Looping a Specific Number of Times
  - Looping with an Index: The enumerate() Function
  - Looping Over Multiple Lists: The zip() Function
  - Interactive Exercises: FizzBuzz and Data Processing
- **The while Loop & Controlling the Flow**
  - The while Loop: The Conditional Loop
  - The Danger of Infinite Loops (and How to Avoid Them)
  - Controlling Loops: break (The Emergency Exit)
  - Controlling Loops: continue (Skip to the Next Round)
  - *Interactive Exercises: Input Validation and Filtering*
- **Advanced Loop Patterns & The Hands-On Lab**
  - The else Block in Loops: A Unique Python Feature
  - Choosing the Right Loop: for vs. while
  - Hands-On Lab: Interactive Data Aggregator
  - Q&A and Wrap-up

THE FOR LOOP - ITERATING OVER COLLECTIONS

THE "WHY": AUTOMATING REPETITIVE TASKS

- Imagine you have a list of 100 names and you need to print each one. You could write 100 print() statements, but that's tedious and impossible to maintain. A loop lets you define a block of code and tells the program to run it once for every item in a collection.
- **Syntax:** for variable_name in sequence:

```
# The 'variable_name' (fruit) will hold one item from the list on each iteration.
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(f"Current fruit is: {fruit}")
```

```
# Iterating over a string
for char in "Python":
    print(char)

# Iterating over a dictionary's keys, values, or items
user = {"name": "Alice", "id": 123}
for key, value in user.items(): # Remember .items() from our dictionary lesson!
    print(f"{key}: {value}")
```

THE RANGE() FUNCTION: LOOPING A SPECIFIC NUMBER OF TIMES

- What if you don't have a list, but you just want to do something 5 times? The range() function generates a sequence of numbers.
- **Syntax:** range(stop), range(start, stop), range(start, stop, step)

```python
# Loop 5 times (0 to 4)
for i in range(5):
    print(f"Looping, round {i}")

# Loop from 2 to 6
for i in range(2, 7):
    print(i) # Prints 2, 3, 4, 5, 6

# Count down from 10 to 1 by 2s
for i in range(10, 0, -2):
    print(i) # Prints 10, 8, 6, 4, 2
```

LOOPING WITH AN INDEX: THE ENUMERATE() FUNCTION

- Sometimes you need both the item *and* its index. A common but un-Pythonic way is to use range(len(list)).

```
# The C-style, less Pythonic way
fruits = ["apple", "banana", "cherry"]
for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")
```

- The **Pythonic** solution is the enumerate() function. It wraps around any iterable and, on each iteration, yields a tuple containing the index and the item.

```
# The Pythonic way with enumerate()
for index, fruit in enumerate(fruits): # Uses tuple unpacking!
    print(f"Index {index}: {fruit}")


# You can even start the index at a different number
for index, fruit in enumerate(fruits, start=1):
    print(f"Item #{index}: {fruit}")
```

LOOPING OVER MULTIPLE LISTS: THE ZIP() FUNCTION

- What if you have two or more lists that correspond to each other? For example, a list of students and a list of their grades. The zip() function pairs them up. It creates an iterator that aggregates elements from each of the iterables.

- zip() stops as soon as one of the lists runs out of items.

- students = ["Alice", "Bob", "Charlie"]
- grades = [95, 88, 92]
- ids = [101, 102, 103]

- for student, grade, student_id in zip(students, grades, ids):
-     print(f"ID: {student_id} - Student: {student} - Grade: {grade}")

IN-CLASS EXERCISE: THE FIZZBUZZ CHALLENGE

This is a classic programming interview question. Write a program that loops through the numbers from 1 to 100.

- For multiples of three, print "Fizz" instead of the number.
- For multiples of five, print "Buzz".
- For numbers which are multiples of both three and five, print "FizzBuzz".
- Otherwise, print the number.
  (Hint: You'll need a for loop with range() and if/elif/else with the modulus % operator).

- Create a list of numbers from 1 to 30 using list(range(1, 31)).
- Use a for loop with enumerate(numbers, start=1) to iterate through the list.
- For each number, check the FizzBuzz rules (multiples of 3, 5, or both).
- Print the output in the format: Item #X - FizzBuzz, Item #Y - Fizz, Item #Z - Buzz, or Item #N - N.

THE WHILE LOOP & CONTROLLING THE FLOW

THE WHILE LOOP: THE CONDITIONAL LOOP

- While a for loop runs for a known number of iterations (the length of the sequence), a while loop runs as long as a certain condition remains True. It's perfect for situations where you don't know when the loop will end, like waiting for user input.
- **Syntax:** while condition:

```
count = 0
while count < 5:
    print(f"Count is {count}")
    count += 1 # CRITICAL: Update the variable to avoid an infinite loop!

print("Loop finished.")
```

- **The Danger of Infinite Loops:** If the condition in a while loop never becomes False, the loop will run forever, and you'll have to manually stop your program. Always ensure something inside the loop can eventually change the condition.

CONTROLLING LOOPS: BREAK AND CONTINUE

Sometimes you need to exit a loop early or skip an iteration.

- break: Immediately terminates the **innermost** loop it's in. The program continues executing the code *after* the loop.
- continue: Immediately stops the *current iteration* and jumps to the top of the loop to start the next one.

```
# Example of 'break': Find the first number divisible by 7
numbers = [2, 5, 11, 14, 22, 21]
for num in numbers:
    if num % 7 == 0:
        print(f"Found a number divisible by 7: {num}")
        break # Exit the loop immediately

# Example of 'continue': Process only positive numbers
data = [10, 20, -5, 30, -1, 0]
for item in data:
    if item <= 0:
        continue # Skip this iteration and go to the next item
    print(f"Processing positive number: {item}")
```

IN-CLASS EXERCISE: INPUT VALIDATOR

Write a program that asks the user for their name.

- Use a while loop to keep asking until they enter a name that is at least 3 characters long.
- If they enter a short name, print an error and let the loop continue.
- Once they enter a valid name, print a welcome message and break the loop.

ADVANCED LOOP PATTERNS & THE HANDS-ON LAB

THE ELSE BLOCK IN LOOPS: A UNIQUE PYTHON FEATURE

- Python loops can have an else block. This is a special feature that often surprises programmers from other languages. The else block executes only if the loop completes its full course without being terminated by a break statement.

```
# Use case: Searching for an item
my_list = [1, 3, 5, 7]
search_for = 5

for item in my_list:
    if item == search_for:
        print("Found it!")
        break
else: # This only runs if the 'break' was never hit
    print("Item not found in the list.")
```

CHOOSING THE RIGHT LOOP: FOR VS. WHILE

- **Use a for loop when:** You have a definite collection of items to iterate over (a list, a string, a dictionary, a range). You know how many times you need to loop.
- **Use a while loop when:** You are looping based on a condition. You don't know how many iterations it will take (e.g., waiting for user input, running a game, processing a data stream until it ends).

INTERACTIVE DATA AGGREGATOR

- **The Goal:** Create a program that allows a user to enter a series of numbers. The program will then calculate and display the sum, average, count, maximum, and minimum of the numbers entered.
- **Step-by-Step Instructions:**
- Create a new file aggregator.py.

PART 1: SETUP

- Create an empty list called numbers to store the user's input.
- Print a welcome message explaining that the user can enter numbers one by one and should type "done" when they are finished.

PART 2: THE INPUT LOOP

- Start an infinite while True: loop.
- Inside the loop, prompt the user for input: "Enter a number or 'done': ".
- Check if the user's input is equal to "done". If it is, break the loop.

PART 3: INPUT VALIDATION AND STORAGE

- If the input was not "done", you need to convert it to a number.
- Use a try...except block to handle potential ValueErrors if the user enters text that isn't a number.
- Inside the try block, convert the input to a float and .append() it to your numbers list.
- Inside the except block, print an error message like "Invalid input. Please enter a number or 'done'." and use continue to skip to the next loop iteration.

PART 4: CALCULATION AND OUTPUT

- **After** the loop has finished, you need to perform the calculations.
- First, check if the numbers list is empty (in case the user typed "done" immediately). If it is, print "No numbers were entered." and exit.
- If the list is not empty:
  - Calculate the count of numbers (the length of the list).
  - Calculate the total_sum (you can use the built-in sum() function).
  - Calculate the average (total_sum / count).
  - Find the maximum and minimum values (you can use the max() and min() built-in functions).
- Print all the results in a nicely formatted summary.

CHALLENGE / BONUS FEATURES:

- **Store in a Dictionary:** Instead of just printing the results, create a dictionary called analysis_results with keys like "sum", "average", "count", "max", "min" and store your calculated values in it. Then, print this dictionary.
- **Filter the Data:** Before performing the calculations, create a *new list* that contains only the positive numbers from the user's input. Perform all the calculations on this new filtered list instead.
- **Median Calculation:** For a real challenge, calculate the median of the numbers. You will need to .sort() the list first. Then, if the number of items is odd, the median is the middle element. If it's even, it's the average of the two middle elements.