Session 3.1: For Loops - Automation Begins!

Repeat Tasks Automatically: Week 3, Day 1

Today's Journey

- Part 1: For Loops - Automation Power!
- Part 2: Workshop - Automate Your First Tasks
- Break
- Part 3: Git Branching - Parallel Development
- Part 4: Workshop - Create Your First Branch
- Part 5: Quiz & Wrap-up

Part 1: For Loops

Automation = Programming Power!

Welcome to Week 3! 🚀

Last two weeks you learned:

- ✓ Variables, strings, operators
- ✓ Lists and list comprehensions
- ✓ Git and GitHub

This week: AUTOMATION!

Loops = Make computer do repetitive work

- Process thousands of items
- No copy-paste needed
- Computer never gets tired

This is where programming gets POWERFUL! 💪

What Are Loops? (Simple Explanation)

Loop = repeat code automatically

Real life examples:

- Check each email in inbox
- Grade each student's homework
- Process each row in Excel
- Send message to each customer

Without loops: Copy-paste code 1000 times 😱

With loops: Write code once, repeat automatically! 😎

DRY Principle: Don't Repeat Yourself

Loops make you a 10x programmer!

For Loop Syntax (Memorize This!)

Format:

- for ITEM in COLLECTION:
    - # code to repeat
    - # gets executed for each item

Parts:

- for = keyword (starts loop)
- ITEM = variable name (your choice)
- in = keyword
- COLLECTION = list, string, range, etc.
    - : = colon (required!)
    - Indentation = what repeats (4 spaces)

Read it: 'For each ITEM in COLLECTION, do this...'

Your First For Loop!

```python
# Loop through a list
fruits = ['apple', 'banana', 'cherry', 'orange']
for fruit in fruits:
    print(fruit)

# Output:
# apple
# banana
# cherry
# orange

# Python automatically:
# 1. Takes first item ('apple')
# 2. Runs the code (print)
# 3. Repeats for EACH item!
# It’s magic!!! ✨
```

For Loops with Actions

```
# Process each number
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    squared = num ** 2
    print(f'{num} squared is {squared}')

# Output:
# 1 squared is 1
# 2 squared is 4
# 3 squared is 9
# 4 squared is 16
# 5 squared is 25

# Each iteration = one execution
# 5 items = loop runs 5 times
```

range() Function - Counting

range() = create sequence of numbers

Perfect for counting!

Three ways to use range():

- range(stop) → 0 to stop-1
- range(start, stop) → start to stop-1
- range(start, stop, step) → with increment

Examples:

- range(5) → 0, 1, 2, 3, 4
- range(1, 6) → 1, 2, 3, 4, 5
- range(0, 10, 2) → 0, 2, 4, 6, 8

Note: Always stops BEFORE the end number!

Using range() in Loops

```
# Count to 5
for i in range(5): # This starts a for-loop
    print(i) #
# Output: 0, 1, 2, 3, 4

# Count 1 to 10
for i in range(1, 11):
    print(i)
# Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

# Count by 2's (even numbers)
for i in range(0, 11, 2):
    print(i)
# Output: 0, 2, 4, 6, 8, 10

# Count by 2's (odd numbers)
for i in range(1, 11, 2):
print(i)
# Output: 1, 3, 5, 7, 9

# Count backwards!
for i in range(10, 0, -1):
    print(i)
# 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Why range() Matters for ML

ML = training over multiple epochs

Typical ML training loop:

- for epoch in range(100):
    - train_model()
    - calculate_loss()
    - update_weights()

This runs 100 times automatically!

Processes thousands of data samples

You'll write loops like this constantly!

range() is essential! 🎯

Accumulator Pattern (Super Important!)

Accumulator = build result during loop

Pattern:

1. Start with empty/zero
2. Loop through data
3. Add to result each time
4. Return final result

Used for:

- Building lists
- Calculating sums
- Counting items
- Filtering data

Master this pattern - it's everywhere! 📊

Accumulator: Building a List

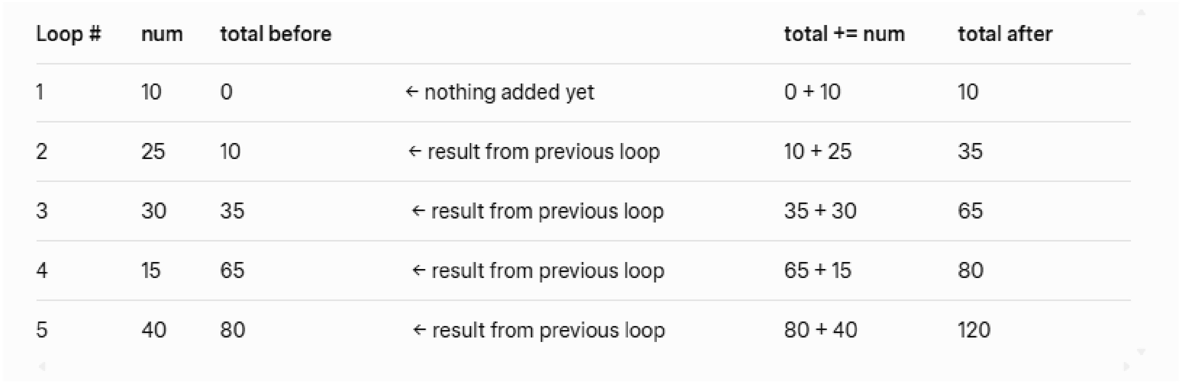
```
# Build list of squares (accumulator pattern)
squares = []  # 1. Initialize empty list to store results
for i in range(10):  # 2. Loop through data (from 0 to 9)
    squares.append(i ** 2)  # 3. Square i and add to list
print(squares)
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Same as comprehension:
# squares = [i**2 for i in range(10)]
# But accumulator pattern works for
# more complex logic!
# You'll use this CONSTANTLY!
```

Accumulator: Calculating Sum

```
# Calculate total (accumulator pattern)
numbers = [10, 25, 30, 15, 40]
total = 0  # 1. Initialize accumulator at zero

for num in numbers:  # 2. Loop through each number in list
    total += num  # 3. Add current number to running total
print(f'Total: {total}')  # 120

# Average = total / count
average = total / len(numbers) # Divide total by number of items
print(f'Average: {average}')  # 24.0
```

Accumulator: Calculating Sum

```
# Calculate total (accumulator pattern)
numbers = [10, 25, 30, 15, 40]
total = 0  # 1. Start at zero
for num in numbers:  # 2. Loop
    total += num  # 3. Add each
print(f'Total: {total}')  # 120
# Or use sum() function:
# total = sum(numbers)

# Below is how the calculation works:
```



Loop #	num	total before		total += num	total after
1	10	0	← nothing added yet	0 + 10	10
2	25	10	← result from previous loop	10 + 25	35
3	30	35	← result from previous loop	35 + 30	65
4	15	65	← result from previous loop	65 + 15	80
5	40	80	← result from previous loop	80 + 40	120

```
# Average = total / count
average = total / len(numbers)
print(f'Average: {average}')  # 24.0
```

enumerate() - Track Position!

enumerate() = loop with index counter

Returns: (index, value) pairs

Format: enumerate(list)

- for index, item in enumerate(list):

Why use it?

- Track position while looping
- Number items automatically
- Access index AND value

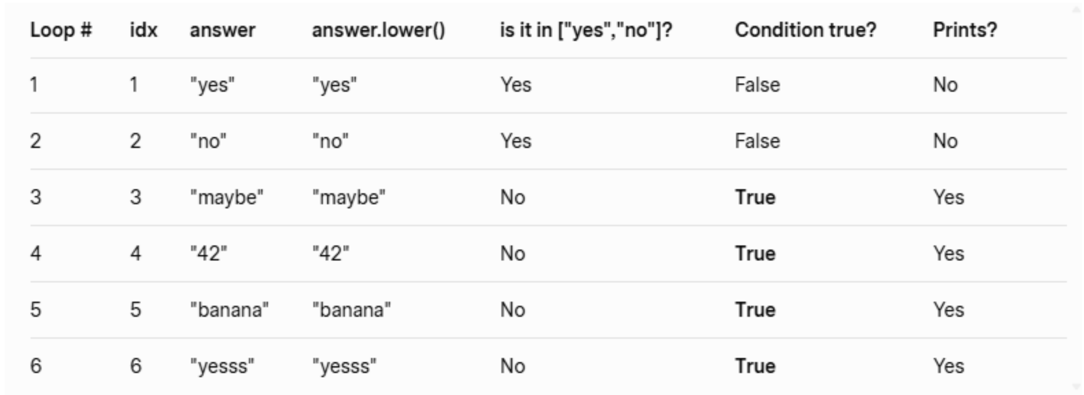
Super useful in data processing! 🔢

Using enumerate()

```
students = ['Alice', 'Bob', 'Carol', 'David']

# With enumerate
for index, name in enumerate(students):
    print(f'Student #{index + 1}: {name}')

# Output:
# Student #1: Alice
# Student #2: Bob
# Student #3: Carol
# Student #4: David

# Can also start from different number:
for index, name in enumerate(students, start=1):
    print(f'{index}. {name}')
# Automatic numbering!
```

Using enumerate()

```
# Imagine you're processing user input and want to number invalid entries for better
error messages:
responses = ["yes", "no", "maybe", "42", "banana", "yesss"]

print("Invalid answers (with line numbers):")
for idx, answer in enumerate(responses, start=1):
     if answer.lower() not in ["yes", "no"]:
          print(f" Line {idx:2d}: '{answer}' ← not allowed")
#Output:
Invalid answers (with line numbers):
  Line  3: 'maybe' ← not allowed
  Line  4: '42' ← not allowed
  Line  5: 'banana' ← not allowed
  Line  6: 'yesss' ← not allowed
```



Loop #	idx	answer	answer.lower()	is it in ["yes","no"]?	Condition true?	Prints?
1	1	"yes"	"yes"	Yes	False	No
2	2	"no"	"no"	Yes	False	No
3	3	"maybe"	"maybe"	No	**True**	Yes
4	4	"42"	"42"	No	**True**	Yes
5	5	"banana"	"banana"	No	**True**	Yes
6	6	"yesss"	"yesss"	No	**True**	Yes

Loops for Data Processing (1/3)

Common loop patterns in data science:

1. Read and process each item

```
for item in data:
    result = process(item)
```

2. Accumulate results

```
total = 0
for value in data:
    total += value
```

3. Filter data

```
filtered = []
for item in data:
    if condition: filtered.append(item)
```

These patterns = 90% of your code!

Three Common Loops for Data Processing (1/2)

```
# 1. Read and process each item (transformation / mapping) - Clean and standardize city names
cities = [" new york ", "Los angeles", "chicago ", "houston", "PHOENIX"]
cleaned_cities = []
for city in cities: # process each item: strip whitespace + title case
cleaned = city.strip().title()
cleaned_cities.append(cleaned)
print(cleaned_cities)
# Output: ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix’]
# Alternative one-liner style:
 cleaned_cities = [city.strip().title() for city in cities]

# 2: Accumulate results (sum, count, min/max, product, concatenation…)
sales = [120.50, 45.00, 890.75, 15.20, 320.00, 675.30]
total_sales = 0
high_value_count = 0
for amount in sales:
     total_sales += amount # accumulate sum
     if amount > 500:
          high_value_count += 1 # accumulate count
average_sale = total_sales / len(sales)
print(f"Total sales : ${total_sales:,.2f}")
print(f"Average sale : ${average_sale:,.2f}")
print(f"High-value sales (> $500): {high_value_count}")
# Output: # Total sales : $2,066.75
# Average sale : $344.46
# High-value sales (> $500): 2
```

Three Common Loops for Data Processing (2/2)

```
# 3. Filter data (create a new list with only items that match a condition)
import datetime # use this if getting syntax error: from datetime import datetime

dates = [
    datetime.date(2025, 10, 13),    # Monday
    datetime.date(2025, 10, 14),    # Tuesday
    datetime.date(2025, 10, 15),    # Wednesday
    datetime.date(2025, 10, 18),    # Saturday
    datetime.date(2025, 10, 19),    # Sunday
    datetime.date(2025, 10, 20),    # Monday
]

weekdays = []
for dt in dates:
    if dt.weekday() < 5:            # 0-4 = Mon-Fri
        weekdays.append(dt)

print("Weekdays only:")
for d in weekdays:
    print(d.strftime("%A, %Y-%m-%d"))
# Output:
# Weekdays only:
# Monday, 2025-10-13
# Tuesday, 2025-10-14
# Wednesday, 2025-10-15
# Monday, 2025-10-20
```

Bonus: Combining patterns 1 + 3 (process + filter)

```
# Very frequent in real workflows:

# Keep only emails that look valid and convert to lowercase
raw_emails = ["alice@company.com", "bob@home", "charlie@gmail.com", "", "DAVE@WORK.COM  "]
valid_emails = []
for email in raw_emails:
    cleaned = email.strip().lower()
    if "@" in cleaned and "." in cleaned.split("@")[-1]:
        valid_emails.append(cleaned)

print(valid_emails)
# ['alice@company.com', 'charlie@gmail.com', 'dave@work.com']
```

Part 2: Workshop

Automate Your First Tasks!

💻 Exercise 1: Multiplication Table

Create file: multiplication_table.py

Build a multiplication table generator:

1. Ask user for a number (or use number = 7)

2. Use a for loop with range(1, 11)

3. Print: '7 × 1 = 7'

    '7 × 2 = 14'

    ... up to 10

4. Use f-strings for formatting

5. Make output look professional

This automates what would be 10 print statements!

That's the power of loops! 🔥

Exercise 2: Grade Statistics

Create file: grade_stats.py

Given list of grades:

grades = [95, 87, 78, 92, 88, 76, 94, 85, 90, 82]

Use loops to calculate:

1. Total of all grades (accumulator)
2. Average grade
3. Number of grades >= 90 (A's)
4. Number of grades < 70 (failing)
5. Highest grade (or use max())
6. Lowest grade (or use min())

Print each statistic with clear labels

This is real data analysis!

☕ Great Work!

You're automating tasks!

Part 3: Git Branching

Parallel Development

What is Branching? (Simple Explanation)

Branch = parallel version of your code

Think of it like:

- Trying on clothes without buying
- Draft email before sending
- Sketch before final drawing

Why branch?

- Experiment safely
- Work on features independently
- Multiple people, same project
- Keep main code stable

Professional teams use branches constantly! 🌳

Main Branch (Default)

main = default branch (previously 'master')

Main branch should be:

- Always working code
- Production-ready
- Tested and stable

Never experiment directly on main!

Instead:

1. Create new branch
2. Make changes there
3. Test everything
4. Merge back to main when ready

Protects your working code! 🛡️

Creating Your First Branch

```
# See current branch
git branch
# * main  (the * shows current branch)

# Create new branch
git branch feature-loops

# See all branches now
git branch
# * main
#   feature-loops
# Switch to new branch
git checkout feature-loops
# Switched to branch 'feature-loops'

# Shortcut: Create and switch in one command
git checkout -b feature-loops
# Now you're on the new branch!
```

Working on a Branch

When on a branch:

1. Make changes to your code
2. Commit as normal (git add, git commit)
3. Changes only affect this branch
4. Main branch unchanged

To switch branches:

- git checkout branch-name

Files literally change!

Python code shows branch version

This is mind-blowing first time you see it! 🤯

Branch Workflow Example

```
# Start on main
git checkout main

# Create feature branch
git checkout -b add-loops-exercise

# Make changes, add new file
# ... edit code ...

# Commit on this branch
git add loops_exercise.py
git commit -m 'Add loops practice exercise'

# Switch back to main
git checkout main
# loops_exercise.py disappears!

# Switch to feature branch
git checkout add-loops-exercise
# loops_exercise.py reappears!
# Magic!!! ✨
```

Merging Branches

Merge = combine branch back into main

Steps:

1. Switch to main: git checkout main
2. Merge feature: git merge feature-name
3. Delete branch: git branch -d feature-name

What happens:

- Changes from feature → main
- Main now has your new code
- Feature branch can be deleted

This is how teams work together! 🤝

Complete Branch & Merge Example

```
# 1. Create and switch to new branch
git checkout -b improve-calculator

# 2. Make changes and commit
# ... edit calculator.py ...
git add calculator.py
git commit -m 'Add advanced math functions'

# 3. Switch back to main
git checkout main

# 4. Merge feature into main
git merge improve-calculator
# Updating abc123..def456
# Fast-forward

# 5. Delete feature branch (optional)
git branch -d improve-calculator
# Done! Changes now in main
```

Branch Naming Conventions

Good branch names:

- ✓ feature-login
- ✓ fix-bug-123
- ✓ add-loops-exercises
- ✓ update-readme

Bad branch names:

- ✗ test
- ✗ branch1
- ✗ my-branch
- ✗ asdf

Name describes what you're doing

Use hyphens, not spaces

Keep it short but clear

Why Branches Matter

Professional development uses branches:

1. Feature development
    - Work on new feature without breaking main
2. Bug fixes
    - Fix bugs in isolation
3. Experimentation
    - Try ideas safely
4. Team collaboration
    - Multiple people, same project

Employers expect you to know branching!

Essential professional skill! 💼

Part 4: Workshop

Practice Branching!

💻 Exercise 4: Complete Branch Workflow

Practice the full branch workflow:

1. Check current branch: git branch

2. Create new branch: git checkout -b practice-loops

3. Create file: loop_practice.py

   Add a for loop that prints 1-10

   Add a loop that calculates sum of 1-100

4. Commit changes:

   git add loop_practice.py

   git commit -m 'Add loop practice exercises'

5. Switch to main: git checkout main

   (Notice file disappears!)

6. Switch back: git checkout practice-loops

   (File reappears!)

7. Merge into main:

   git checkout main

   git merge practice-loops

8. Push to GitHub: git push

You just did professional Git workflow! 🎉

Part 5: Quiz & Wrap-up

Session Complete!

📝 Session 3.1 Quiz

🎉 Session 3.1 Complete! Automation Unlocked!

What you mastered today:

- ✓ For loops (automation power!)
- ✓ range() function (counting & iteration)
- ✓ enumerate() (position tracking)
- ✓ Accumulator pattern (building results)
- ✓ Git branching (parallel development)
- ✓ Branch workflow (create, commit, merge)

You can now:

- Automate repetitive tasks
- Process thousands of items
- Work on features safely with branches

Major milestone! 🌟

Homework - Master Loops!

1. Complete the quiz (10 questions - required)
2. Project: Data Processor
    - Build: data_processor.py
    - Create list of 20 numbers (use range)
    - Calculate: sum, average, min, max
    - Count: evens, odds, divisible by 5
    - Build new list: only numbers > average
    - Use loops for everything!
3. Git Practice
    - Create branch: homework-week3
    - Commit your work
    - Merge to main
    - Push to GitHub
4. Prepare: Read about while loops (next topic)

Looking Ahead: Session 3.2

Next session: Advanced Loops & While

You'll learn:

- Nested loops (loop inside loop)
- while loops (condition-based)
- break and continue (loop control)
- Advanced iteration patterns
- Feature branches in depth

Why it matters:

- Process 2D data (matrices)
- Complex iteration patterns
- Conditional repetition
- Real-world data processing

Loops level up! 🚀

Week 3 Progress! 💪

Think about your growth:

Two weeks ago:

- 'What's a variable?'

Today:

- 'I write loops and use Git branches!'

You've learned:

- Core Python (variables, strings, lists)
- Automation (loops, comprehensions)
- Version control (Git, GitHub, branches)

This week: More automation, conditionals

You're becoming a real programmer! 🎓

Keep practicing! See you next session! 🚀