# Repetition Structures (Loops)

while, for, range, running totals, sentinels, validation, nested loops, turtle designs

# Learning Outcomes

After this lecture, you will be able to:

- Explain **why** repetition structures (loops) are needed in programs.
- Distinguish between **condition-controlled** (`while`) and **count-controlled** (`for`) loops.
- Describe what an **iteration** is and how loop execution flows.
- Use `range()` correctly (start, stop, step; ascending and descending).
- Build programs using **accumulators** (running totals) and **augmented assignments**.
- Use **sentinels** and **input validation loops** safely.
- Understand **nested loops** and compute total iterations.
- Use loops with **turtle** to create shapes and designs.

# What is a Repetition Structure?

**Concept:** A repetition structure causes a statement or set of statements to execute repeatedly.

**Why we need loops:**

- Writing the same code again and again is **slow** and **error-prone**.
- If requirements change, duplicated code must be edited **many times**.
- Loops let us write the logic **once** and repeat it **as needed**.

# Motivation Example: Duplicated Code (Bad Design)

```
sales = float(input("Enter sales: "))       # Read sales for person 1
rate  = float(input("Enter rate: "))        # Read commission rate
commission = sales * rate                   # Compute commission
print("Commission:", commission)            # Print commission

sales = float(input("Enter sales: "))       # Same code repeated (
    person 2)
rate  = float(input("Enter rate: "))        # Same code repeated
commission = sales * rate                   # Same computation
    repeated
print("Commission:", commission)            # Same print repeated
```

- This grows into a long program with repeated blocks.
- We want: **repeat the block using a loop**.

# Two Big Loop Categories

## 1) Condition-controlled loops

- Repeat **as long as a condition is True**.
- You typically **do not know** exact number of repeats in advance.
- Python tool: `while`

## 2) Count-controlled loops

- Repeat a **specific number** of times.
- You often know the number of iterations (e.g., 5 inputs, 10 rows).
- Python tool: `for` (usually with `range`)

# while Loop: How It Works

**Meaning:** *While a condition is True, keep repeating the body.*

**General form:**

- `while condition:`
- `    statements` (indented block)

**Key idea:** The condition is checked **before every iteration**.

# while is a Pretest Loop

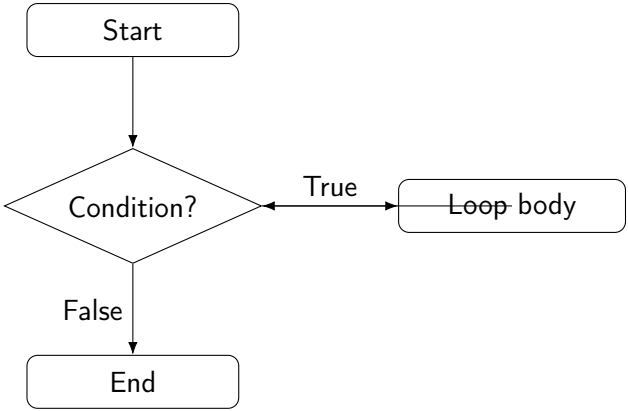
**Pretest loop** means:

- Python tests the condition **first**.
- If condition is False initially, loop runs **zero times**.

**Practical consequence:**

- You often need initialization (a starting value) before the loop.

# Flow of Execution: while Loop (Visual)




- Each time the body runs once = **one iteration**.

# Example: Commission Calculator with while

```
keep_going = "y"                                    # Initialize to
    force first entry

while keep_going == "y":                            # Loop continues
    while user types 'y'
    sales = float(input("Enter the amount of sales: "))    # Read
        sales
    rate  = float(input("Enter the commission rate: "))    # Read
        commission rate

    commission = sales * rate                       # Compute
        commission
    print("The commission is $", format(commission, ",.2f"), sep="")
          # Display formatted

    keep_going = input("Another commission? (y for yes): ")  #
        Update control variable
```

# What is an Iteration?

**Iteration = one complete pass through the loop body.**

In the commission example:

- User enters sales/rate $\rightarrow$ commission prints $\rightarrow$ asked again.
- If user says y three times, the loop iterates **3 times**.

**Important:** Count iterations by counting how many times the body executes.

# Tracing Iterations (Step-by-Step)

```
count = 1                                   # Start at 1

while count <= 3:                           # Condition checked before
    each iteration
    print("Iteration:", count)              # Body runs if condition is
        True
    count = count + 1                       # Update: moves count toward
        stopping
```

**How it runs:**

- Test: 1<=3 True → print 1 → count=2
- Test: 2<=3 True → print 2 → count=3
- Test: 3<=3 True → print 3 → count=4
- Test: 4<=3 False → exit

# Infinite Loops

**Infinite loop** = loop that never stops.

Common cause:

- Condition never becomes False.
- Update step is missing or incorrect.

If your program "hangs" printing forever, suspect an infinite loop.

# Infinite Loop Example (What NOT to do)

```
keep_going = "y"                          # Control variable set to 'y'

while keep_going == "y":                  # Condition always True
    print("This will run forever!")       # No update to keep_going
        inside loop
```

- Stop manually with `Ctrl+C` in most terminals.

# Example: Temperature Technician (while loop)

```
MAX_TEMP = 102.5                                    # Maximum
    acceptable temperature

temperature = float(input("Enter Celsius temperature: "))  # Priming
     read (first input)

while temperature > MAX_TEMP:                       # Repeat
    while too hot
     print("Too high. Turn thermostat down and wait 5 minutes.")  #
         Instructions
     temperature = float(input("Enter new Celsius temperature: "))  #
          Read again

print("Temperature acceptable. Check again in 15 minutes.")       #
    After loop ends
```

**Why while is perfect here:** If temperature is already OK, loop runs **0 times**.

# for Loop: The Big Idea

**Meaning:** Repeat once for each item in a sequence.

**General form:**

- `for target_variable in sequence:`
- `    statements` (indented block)

**Target variable:**

- Receives a new value at the start of each iteration.

# for Loop with a List (Simple Example)

```
print("I will display the numbers 1 through 5.")    # Explanation
    message

for num in [1, 2, 3, 4, 5]:                          # num takes each
     list value
     print(num)                                      # Print current
         num each iteration
```

**Iterations:**

- 1st iteration: num=1
- 2nd iteration: num=2
- ...
- 5th iteration: num=5

# for Loop with Strings (Sequence of Names)

```
for name in ["Winken", "Blinken", "Nod"]:    # name becomes each
    string
     print(name)                              # Print the current
         name
```

- The loop iterates **3 times** because list has 3 items.

# range(): The Most Important Helper for for Loops

`range()` generates a sequence of integers you can iterate over.

**Three common forms:**

- `range(stop)` → 0,1,2,...,stop-1
- `range(start, stop)` → start,...,stop-1
- `range(start, stop, step)` → step increments (or decrements)

**Golden rule:** `stop` is **not included**.

# range(stop): Example

```
for x in range(5):                    # Generates 0,1,2,3,4
    print("Hello world")              # Prints message 5 times
```

**Iterations:** exactly 5 (because range(5) has 5 values).

# range(start, stop): Example

```
for num in range(1, 5):                    # Generates 1,2,3,4 (stop=5 not
    included)
    print(num)                              # Prints 1 to 4
```

- If you want 1..5, use `range(1, 6)`.

# range(start, stop, step): Example

```
for num in range(1, 10, 2):          # Generates 1,3,5,7,9
    print(num)                        # Prints odd numbers
```

- Step tells how much to add each time.

# Descending range(): Highest to Lowest

```
for num in range(5, 0, -1):          # Generates 5,4,3,2,1
    print(num)                        # Prints countdown
```

**Important:**

- For descending, step must be negative.

# Using the Target Variable in Calculations: Squares Table

```
print("Number\tSquare")              # Print headings with tab
print("--------------")              # Separator line

for number in range(1, 11):          # number goes 1..10
    square = number ** 2             # Compute square
    print(number, "\t", square)      # Print aligned columns using tab
```

**Iterations:** 10 (one per number).

# Example: KPH to MPH Table (Count-Controlled)

```
START_SPEED = 60                          # Start KPH
END_SPEED = 131                           # Stop limit (so last is 130)
INCREMENT = 10                            # Step size
FACTOR = 0.6214                           # Conversion factor KPH -> MPH

print("KPH\tMPH")                         # Table header
print("--------------")                   # Separator

for kph in range(START_SPEED, END_SPEED, INCREMENT):  # 60..130 step
    10
    mph = kph * FACTOR                    # Convert current kph to mph
    print(kph, "\t", format(mph, ".1f"))  # Print with 1 decimal
```

**Why END_SPEED=131?** because stop is excluded.

# Letting the User Control Iterations (End Value)

```
print("This program prints numbers and their squares.")   # Explain

end = int(input("How high should I go? "))                 # User
    decides maximum

print("Number\tSquare")                                    # Header
print("--------------")

for number in range(1, end + 1):                           # end+1 to
    include end
     square = number ** 2                                  # Square
     print(number, "\t", square)                           # Print row
```

- Using end+1 fixes the common **off-by-one** error.

# Loop Mechanics: The 4-Step Mental Model

For most loops, think of this repeating cycle:

1. **Initialize** (set starting values, counters, accumulators)
2. **Test** (check condition for `while`, or get next value for `for`)
3. **Execute body** (do the repeated work)
4. **Update** (change something so progress happens)

If update is missing or wrong $\Rightarrow$ infinite loop or wrong result.

# Side-by-Side: while vs for Iterations

**while: you control the counter**

```
count = 1                          # Initialize counter
while count <= 3:                  # Test condition
    print(count)                   # Body
    count += 1                     # Update counter
```

**for: Python gives the next value automatically**

```
for count in range(1, 4):          # range provides 1,2,3
    automatically
    print(count)                   # Body
```

# Running Total and Accumulator

**Running total:** sum that grows each iteration.

**Accumulator:** variable that stores the running total.

**Critical rule:** Initialize accumulator properly (usually to 0).

# Example: Sum of 5 Numbers (Accumulator)

```
MAX = 5                                    # Number of inputs to read
total = 0.0                                # Accumulator starts at 0

print("This program sums", MAX, "numbers.")   # Explain

for _ in range(MAX):                       # Repeat exactly 5 times
    number = float(input("Enter a number: "))   # Read number
    total = total + number                  # Add current number to
        accumulator

print("The total is", total)               # Print final accumulated
    total
```

**Iterations:** 5; **accumulator updates each iteration**.

# Augmented Assignment Operators

Instead of writing:

```
total = total + number
```

Python allows:

```
total += number
```

Common forms:

- `x += 1` (add)
- `x -= 1` (subtract)
- `x *= 2` (multiply)
- `x /= 2` (divide)
- `x %= 3` (remainder)

# Accumulator with += (Cleaner)

```
MAX = 5                                 # Number of values
total = 0                               # Accumulator

for _ in range(MAX):                    # Iterate MAX times
    n = int(input("Enter a number: "))  # Read integer
    total += n                          # Add to running total (same as
        total = total + n)

print("Total:", total)                  # Final total after loop
```

# Sentinel Concept

**Sentinel** = special value that marks the end of input.

Use when:

- You **do not know** how many inputs will be entered.
- You want user to stop by typing a special value (e.g., 0).

**Sentinel rule:** choose a value that can never be a valid data item.

# Example: Property Tax with Sentinel (0 Ends)

```
TAX_FACTOR = 0.0065                                  # Tax factor (given)

print("Enter the property lot number")               # Instructions
print("or enter 0 to end.")
lot = int(input("Lot number: "))                     # Priming read (first lot
    number)

while lot != 0:                                      # Loop runs until sentinel (0)
    value = float(input("Enter the property value: "))  # Read property
        value
    tax = value * TAX_FACTOR                         # Compute tax
    print("Property tax: $", format(tax, ",.2f"), sep="")  # Display
        formatted

    print("Enter the next lot number or")            # Ask for next lot number
    print("enter 0 to end.")
    lot = int(input("Lot number: "))                 # Update lot (next value)
```

**Iteration meaning:** one iteration processes one property.

# Input Validation (GIGO)

**Garbage In, Garbage Out (GIGO):**

- If the user enters invalid input, program output becomes invalid.
- Computers do not automatically know what "reasonable" input means.

**Solution:** validate input **before using it**.

# Input Validation Loop Pattern

**Pattern:**

- Read input once (**priming read**).
- While input is invalid:
  - print error message
  - read input again

**Priming read:** first read done before entering the validation loop.

# Validation Example: Score Must Be 0..100

```
score = int(input("Enter a test score (0-100): "))   # Priming read

while score < 0 or score > 100:                      # Bad input
    condition
    print("ERROR: score must be between 0 and 100.") # Error message
    score = int(input("Enter the correct score: "))  # Read again (
        inside loop)

print("Accepted score:", score)                      # Use score
    safely
```

**Iteration meaning:** each iteration is another attempt to correct invalid input.

# Example: Retail Price with Validation + Loop Control

```
MARK_UP = 2.5                                    # Markup factor
another = "y"                                    # Loop control variable

while another == "y" or another == "Y":          # Repeat while user wants
    more
    wholesale = float(input("Wholesale cost: ")) # Read cost

    while wholesale < 0:                         # Validation loop for
        negative cost
        print("ERROR: cost cannot be negative.") # Error message
        wholesale = float(input("Enter correct cost: ")) # Re-read until
            valid

    retail = wholesale * MARK_UP                 # Compute retail price
    print("Retail price: $", format(retail, ",.2f"), sep="") # Display

    another = input("Do you have another item? (y for yes): ") # Update
        control
```

**Notice:** two loops: outer repeats items, inner validates input.

# Nested Loops

**Nested loop** = a loop inside another loop.

Key facts:

- Inner loop completes **all** its iterations for **each** outer iteration.
- Total iterations = product of iterations of all loops.

# Nested Loop Example: Digital Clock (Concept)

```
for hours in range(24):                          # Outer loop: 24 hours
    for minutes in range(60):                    # Middle loop: 60 minutes
        per hour
        for seconds in range(60):                # Inner loop: 60 seconds
            per minute
            print(hours, ":", minutes, ":", seconds)  # Print
                current time
```

**Total inner prints:** $24 \times 60 \times 60 = 86400$ iterations of the innermost body.

# Example: Average Test Scores (Nested + Accumulator)

```
num_students = int(input("How many students? "))          # Read number of
    students
num_tests = int(input("How many tests per student? "))    # Read tests per
    student

for student in range(num_students):                       # Outer loop per
    student
    total = 0.0                                           # Accumulator per
        student
    print("Student", student + 1)                         # Display student
        number

    for test_num in range(num_tests):                     # Inner loop per
        test
        score = float(input("Test " + str(test_num + 1) + ": "))  # Read
            score
        total += score                                    # Add score to
            total

    average = total / num_tests                           # Compute average
    print("Average:", average)                            # Print student's
        average
    print()                                               # Blank line
```

# Nested Loops for Patterns

Think in **rows** and **columns**:

- Outer loop controls rows.
- Inner loop controls columns (characters per row).

After each row, you usually print a newline with `print()`.

# Pattern 1: Rectangle of *

```
rows = int(input("How many rows? "))          # Read number of rows
cols = int(input("How many columns? "))       # Read number of columns

for r in range(rows):                         # Outer loop: each row
    for c in range(cols):                     # Inner loop: each column
        print("*", end="")                    # Print star without
            newline
    print()                                   # Newline after finishing
        a row
```

# Pattern 2: Triangle of *

```
BASE_SIZE = 8                                   # Total rows

for r in range(BASE_SIZE):                      # r = 0..7
    for c in range(r + 1):                      # Inner runs 1,2,3,...,8
        times
        print("*", end="")                      # Print stars on same
            line
    print()                                     # Newline after each row
```

**Iteration idea:**

- Row r has exactly r+1 stars.

# Pattern 3: Stair-Step with Spaces + #

```
NUM_STEPS = 6                                   # Number of rows

for r in range(NUM_STEPS):                      # Outer loop: row number
    for c in range(r):                          # Print r spaces before #
        print(" ", end="")                      # Spaces create shifting
    print("#")                                  # Print # and newline
```

**Row logic:**

- Row 0 prints 0 spaces then #
- Row 5 prints 5 spaces then #

# Example

You can still teach **loops + repetition** using:

- text patterns (stars / ASCII art)
- lists and accumulation
- simple graphics with `matplotlib` (plots)

**Idea:** repeat a small action many times to build a bigger output.

# Example 1: ASCII Square Using Loops

```
SIZE = 6

for row in range(SIZE):                    # repeat for each row
    for col in range(SIZE):                # repeat for each column
        # draw border only
        if row == 0 or row == SIZE-1 or col == 0 or col == SIZE-1:
            print("#", end="")
        else:
            print(" ", end="")
    print()                                # new line after each row
```

**Output (example):**

```
######
#    #
#    #
#    #
#    #
```

# Example 2: Number Pattern (Triangle)

```
ROWS = 6

for i in range(1, ROWS + 1):
    for j in range(1, i + 1):
        print(j, end=" ")
    print()
```

**Output:**

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
```

# Example 3: Bar Chart with Matplotlib (Loop Drawing)

```
import matplotlib.pyplot as plt

values = [3, 7, 2, 6, 4]
labels = ["A", "B", "C", "D", "E"]

# Build x positions using a loop idea (range)
x = [i for i in range(len(values))]

plt.bar(x, values)
plt.xticks(x, labels)
plt.title("Simple Bar Chart")
plt.xlabel("Category")
plt.ylabel("Value")
plt.show()
```

**Iteration meaning:** the loop builds the x-axis positions so each bar is placed correctly.

# Example 4: Growing List (Accumulation with Loops)

```
nums = [2, 5, 1, 8, 3]
running_sum = 0
prefix_sums = []

for n in nums:
    running_sum += n           # repeated action: add current number
    prefix_sums.append(running_sum)

print(prefix_sums)
```

**Output:**

```
[2, 7, 8, 16, 19]
```

**Iteration meaning:** each loop step updates the total and stores it.

# Example 5: Simple Animation Effect in Console

```
import time

for i in range(1, 21):
    print("*" * i)                  # repeat: print more stars each time
    time.sleep(0.1)                 # small pause so it looks animated
```

**Iteration meaning:** each iteration increases the number of stars by 1.

# Common Loop Bugs

- **Infinite loop:** update missing or condition never becomes False.
- **Off-by-one error:** misunderstanding `range(stop)` exclusion.
- **Wrong step sign:** descending ranges need negative step.
- **Accumulator not initialized:** totals become wrong.
- **Validation missing:** garbage input produces garbage output.

# Debug Example 1: Off-by-One

```
for i in range(1, 5):                    # Generates 1,2,3,4 (NOT 5)
    print(i)
```

**Fix if you want 1..5:**

```
for i in range(1, 6):                    # Generates 1,2,3,4,5
    print(i)
```

# Debug Example 2: Accumulator Not Starting at 0

```
total = 100                          # WRONG: accumulator should start
    at 0
for i in range(5):
    total += i
print(total)                         # Result is shifted by 100 (wrong
    total)
```

**Fix:**

```
total = 0                            # Correct initialization
```

# Debug Example 3: Infinite while Loop

```
count = 1
while count <= 5:
    print(count)
    # count += 1                    # Missing update causes infinite
      loop
```

**Fix:**

```
count += 1                          # Update moves count toward
   stopping
```

# Checkpoint (Concepts)

Answer verbally or in notebook:

1. What is a repetition structure?
2. What is the difference between condition-controlled and count-controlled loops?
3. What is an iteration?
4. Why is `while` called a pretest loop?
5. What is an accumulator? Why must it start at 0?
6. What is a sentinel? Why must it be distinctive?
7. What is a priming read in input validation?
8. How do you compute total iterations in nested loops?

# Checkpoint (range Practice)

Predict the output:

1. `for n in range(6): print(n)`
2. `for n in range(2,6): print(n)`
3. `for n in range(0,501,100): print(n)`
4. `for n in range(10,5,-1): print(n)`

Rule reminder: stop is excluded; step controls direction.

# Summary

Today you learned:

- **while** loops: condition-controlled, pretest, can run 0 times.
- **for** loops: count-controlled, iterate over sequences (especially `range`).
- **Iterations**: one execution of loop body; track using initialization/test/update.
- **Accumulators**: running totals; initialize to 0; use +=.
- **Sentinels**: end-of-input markers.
- **Input validation**: reject invalid inputs using priming read + loop.
- **Nested loops**: inner loop completes for every outer iteration; total iterations multiply.
- **Turtle designs**: loops + small turns create complex patterns.

# End

Questions?