Introduction

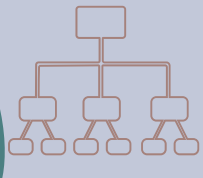
Programming: is the act of designing, implementing, and testing computer programs.

Program instructions and data (such as text, numbers, audio, or video) are stored in digital format.

pseudocode: a simple, human-readable description of an algorithm's steps that looks like programming code but isn't meant to be executed.

Core Programming Concepts:

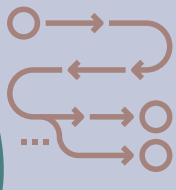
1



Data structures:

Ways to store related data (like lists or sequences).

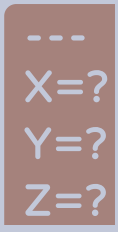
2



Flow control:

Controls how a program runs (using if statements and loops).

3



Variables:

Names that store data values.

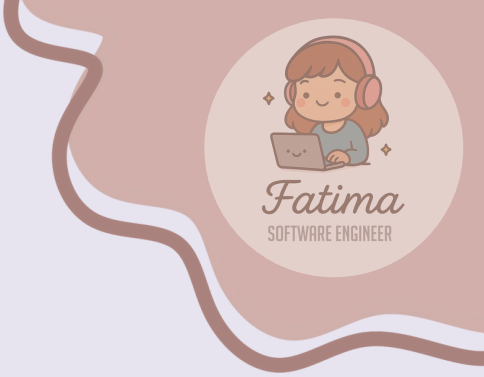
4

var(X)

Functions:

Reusable blocks of code that perform specific tasks.

Python



Why Python?

- Open-source, powerful, and easy to learn.
- Used for AI, machine learning, web development, data analysis, games, etc.
- Focuses on programming logic rather than complex syntax.
- Has simple and readable syntax, ideal for beginners.

What is Python?

- A high-level, object-oriented, interpreted, and dynamically typed language.
- It provides abstraction — you focus on logic, not machine details.
- The Python Virtual Machine (PVM) executes bytecode produced from .py files.

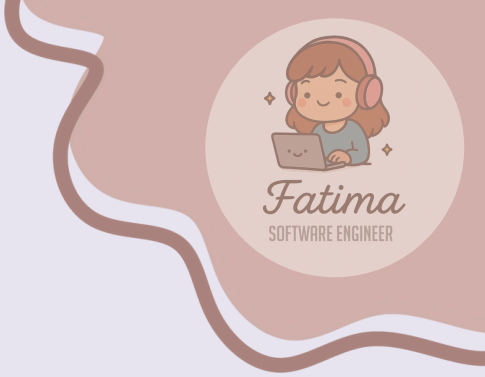
How Python Runs?

- Source code (.py) → compiled to bytecode (.pyc).
- Bytecode is executed by the Python Virtual Machine (PVM).
- Built-in functions like print() and input() are always available.

Getting Python

- Programs are saved as .py files (e.g., hello.py).
- Use Spyder IDE or Anaconda Distribution for development.
- Anaconda includes Python, libraries, and the interpreter.

Python



Types of Programming Errors:

1

Syntax Error:

- Code violates grammar rules.
- Example: Missing : or parentheses

2

Exception Error

- Code runs but encounters a runtime error
- Example: Division by zero

3

Logic Error:

- Code runs but gives wrong output
- Example: Wrong formula or condition

Areas of Computer Science

Systems Areas: Algorithms, Programming Languages, Architecture, OS, Software Engineering, HCI.

Application Areas: AI and Robotics, Databases, Graphics, Bioinformatics, Data Processing, etc.

Algorithm

it is a sequence of actions to take to accomplish the given task.

Software developers usually write an algorithm before starting to write a program.

Algorithmic thinking involves four main steps:

1

Problem Identification:

Define the main problem, data, and assumptions.

2

Decomposition

Break a large problem into smaller, manageable sub-problems.

3

Pattern Recognition

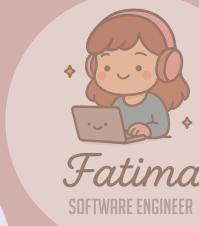
Identify similarities with previously solved problems.

4

Abstraction

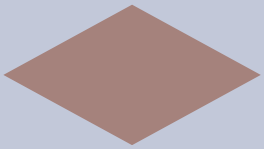
Focus only on the essential parts of the problem and ignore irrelevant details

Flowchart



A flowchart is a type of diagram that represents a workflow or process.

A flowchart is a diagrammatic representation of an algorithm.

Symbol	Name	Function
	Start / End	An oval represents a start or end point.
	Arrows	A line is a connector that shows relationships between the representative shapes.
	Input / Output	A parallelogram represents input or output.
	Process	A rectangle represents a process.
	Decision	A diamond indicates a decision.

Variables

A variable is a named storage location in a computer program.

You use an assignment statement to place a value into a variable.

- Use "=" to assign a new value to a variable.
- Left side (LHS): the variable name.
- Right side (RHS): a value or expression.
- The new value replaces the old one.
- "=" is not used for comparison.

```
x = 5          # Assign 5 to variable x
x = x + 3      # Update: Add 3 to x
print(x)       # Output: 8
```

Variable Naming Rules:

- Must start with a letter or underscore (_)
- No symbols or spaces allowed.
- Use camelCase for multi-word names.
- Case-sensitive — age ≠ Age.
- Don't use Python reserved words like print, list, or import.

Constants:

is a variable whose value should not be changed after it is assigned an initial value.

It is customary to use all UPPER_CASE letters for constants to distinguish them from variables.

Three different types of data:

Datatypes

Three different types of data:

1 Integer (int):

- Definition: A whole number without any fractional part.
- Example: x = 7

2 Float (float):

- Definition: A number that includes a fractional (decimal) part.
- Example: y = 3.14

3 String (str):

- Definition: A sequence of characters enclosed in quotes.
- Example: name = "Ali"

When a value such as 6 or 0.355 occurs in a Python program, it is called a number literal.

If you use a variable and it has an unexpected type an error will occur in your program.

Python comments:

Use comments to explain your code for humans. they help others understand your logic.

Single-line comment: using the # symbol at the beginning of a line.

Multi-line comment: Written between triple quotes (''' or """)

Arithmetic operations

The symbols + - * / for the arithmetic operations are called operators.

The combination of variables, literals, operators, and parentheses is called expression.

- The / operator performs a division yielding a value that may have a fractional value.
- The // operator performs a division, the remainder is discarded.
- The % operator computes the remainder of a floor division.

Precedence : PEMDAS

Precedence	Operator	Description
Higher ↓ Lower	( )	Parenthesis, Function Call
	+ -	Positive Negative
	**	Exponent
	* / // %	Multiplication Real Division Floor Division Modulus (remainder)
	+ -	Addition Subtraction
	=	Assignment

Strings


```python
#   Definition: Strings are sequences of characters written in quotes
message1 = "This is a string"
print(message1)

#   Length of a string using len()
word = "World!"
print("Length of word:", len(word))  # Output: 6

#   Empty string has length 0
empty = ""
print("Length of empty string:", len(empty))  # Output: 0

#   String concatenation (+)
firstName = "Fatima"
lastName = "Al-Amri"
fullName = firstName + " " + lastName
print("Full Name:", fullName)

#   String repetition (*)
dashes = "-" * 10
print(dashes)  # Output: ----------

#   Converting numbers and strings
num = 25
numStr = str(num)     # Convert number to string
print("Number as string:", numStr)

textNum = "3.5"
floatNum = float(textNum)  # Convert string to float
print("Converted to float:", floatNum)

#   Accessing individual characters by index
name = "Python"
print("First letter:", name[0])  # P
print("Last letter:", name[5])   # n

#  Escape sequences for special characters
print("He said \"Hi\"")     # Prints: He said "Hi"
print("C:\\Temp")           # Prints: C:\Temp
print("Line1\nLine2\nLine3")  # Prints on multiple lines

#  Unicode and characters
# Every character has a Unicode code number (ord) and can be converted back (chr)
print("Unicode of 'A':", ord('A'))   # Output: 65
print("Character of 66:", chr(66))   # Output: B
```

Strings as an object

- An object represents a value with certain behavior (e.g., string, window, file, etc.)
- Methods are specific to a type of object
- Functions are general and can accept arguments of different types
- A class describes a set of objects with the same behavior.

Examples of Common String Methods :

```
text = "Let it be, let it be"

# Replace part of the text
print(text.replace("let", "sing"))  # Replace 'let' with 'sing'

# Find first and last occurrence
print(text.find("let"))   # Find the first index of 'let'
print(text.rfind("let"))  # Find the last index of 'let'

# Count how many times a word appears
print(text.count("let"))  # Count occurrences of 'let'

# Change letter case
print(text.upper())   # Convert all to uppercase
print(text.lower())   # Convert all to lowercase
print(text.title())   # Capitalize first letter of each word

# Check string type
print("abc".isalpha())  # True if all characters are letters
print("123".isdigit())  # True if all characters are digits

# Remove spaces from both ends
print("  Hello  ".strip())  # Remove leading/trailing spaces

# Check how the string starts or ends
print(text.startswith("Let"))  # True if starts with 'Let'
print(text.endswith("be"))     # True if ends with 'be'
```

Input and Output

Format Specifier:

Format String	Sample Output	Comments
"%d"	24	Use d with an integer.
"%5d"	␣␣␣24	Spaces are added so that the field width is 5.
"%05d"	00024	If you add 0 before the field width, zeroes are added instead of spaces.
"Quantity:%5d"	Quantity:␣␣␣24	Characters inside a format string but outside a format specifier appear in the output.
"%f"	1.219997	Use f with a floating-point number.
"%.2f"	1.22	Prints two digits after the decimal point.
"%7.2f"	␣␣␣1.22	Spaces are added so that the field width is 7.
"%s"	Hello	Use s with a string.
"%d %.2f"	24 1.22	You can format multiple values at once.
"%9s"	␣␣␣␣Hello	Strings are right-justified by default.
"%-9s"	Hello␣␣␣␣	Use a negative field width to left-justify.
"%d%%"	24%	To add a percent sign to the output, use %%.

Code:

```
# Read user's name (input always returns a string)
name = input("Enter your name: ")

# Read age and convert to integer
age = int(input("Enter your age: "))

#  Read item price and convert to float (decimal number)
price = float(input("Enter price: "))

# Read quantity and convert to integer

qty = int(input("Enter quantity: "))

#  Process step - calculate total cost
total = price * qty

#  Basic Output using print()
print("\nHello,", name)
print("Age next year:", age + 1)
print("Total =", total)

#  Formatted Output - using % for alignment
print("\n%-10s%8s%12s" % ("Item", "Qty", "Price"))       # Header row
print("%-10s%8d%12.2f" % ("Product", qty, price))        # Data row
print("%-10s%8.2f" % ("Total:", total))                  # Total line
```

if Statement

A computer program often needs to make decisions based on input, or circumstances

If and else statement together are called a compound statement.

An if statement may not need a 'False' (else) branch

Compound statements require a colon ":" at the end of the header.

Code:

```python
# 1  Basic if-else example
score = int(input("Enter your score: "))   # Example input → 75
if score >= 60:
    print("Pass")   # Output → Pass
else:
    print("Fail")   # Output → Fail (if score < 60)

# 2  Relational operators
a, b = 10, 20
print("\na == b:", a == b)   # Output → a == b: False
print("a != b:", a != b)     # Output → a != b: True
print("a > b :", a > b)      # Output → a > b : False
print("a < b :", a < b)      # Output → a < b : True

# 3 □String comparison
word1 = "Apple"
word2 = "banana"
print("\nword1 < word2:", word1 < word2)   # Output → word1 < word2: True

# 4  Floating-point comparison using EPSILON
a = 0.3 * 3 + 0.1
b = 1.0
EPSILON = 1e-14
if abs(a - b) < EPSILON:
    print("\nNumbers are almost equal!")   # Output → Numbers are almost equal!

# 5  Simple discount example
price = float(input("\nEnter price: "))    # Example input → 100
if price < 128:
    rate = 0.92
else:
    rate = 0.84
print("Discounted price:", price * rate)   # Output → Discounted price: 92.0
```

Relational Operators

Assignment: makes something true. It saves a value into a variable.

Equality testing: checks if something is true.

Relational Operators

Python	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Code:

```
a = 10
b = 20

# 1  Equal to (==)
print("a == b:", a == b)    # Output → False (10 is not equal to 20)

# 2  Not equal (!=)
print("a != b:", a != b)    # Output → True (10 is not equal to 20)

# 3  Greater than (>)
print("a > b:", a > b)      # Output → False (10 is not greater than 20)

# 4  Less than (<)
print("a < b:", a < b)      # Output → True (10 is less than 20)

# 5  Greater than or equal to (>=)
print("a >= b:", a >= b)    # Output → False (10 is not ≥ 20)

# 6  Less than or equal to (<=)
print("a <= b:", a <= b)    # Output → True (10 is ≤ 20)

# 7  String comparison (Lexicographical order)
x = "Apple"
y = "Banana"
print("\nx == y:", x == y)   # Output → False
print("x != y:", x != y)     # Output → True
print("x < y:", x < y)       # Output → True (A < B alphabetically)
print("x > y:", x > y)       # Output → False
```

Nested Branches

- You can nest an if inside a branch of another if statement.
- When using multiple if statements, test the general conditions after the more specific conditions.
- An if statement may not need a 'False' (else) branch
- elif statement is short for Else if

Code:

```
# 1  Nested if example
num = int(input("Enter a number: "))  # Example → 9
if num > 0:
    if num % 3 == 0:
        print("Positive and divisible by 3")   # Output → Positive and divisible by 3
    else:
        print("Positive but not divisible by 3")
else:
    print("Zero or negative")

# 2  Multiple alternatives (elif)
richter = float(input("\nEnter Richter value: "))  # Example → 6.5
if richter >= 8.0:
    print("Most structures fall")                  # Output for ≥8.0
elif richter >= 7.0:
    print("Many buildings destroyed")              # Output for ≥7.0
elif richter >= 6.0:
    print("Some buildings collapse")               # Output for ≥6.0
elif richter >= 4.5:
    print("Damage to weak buildings")              # Output for ≥4.5
else:
    print("No damage")                             # Output for <4.5
```

Boolean Variables

A Boolean variable is often called a flag because it can either be up (true) or down (false).

boolean is a Python data type.

Boolean Operators:

and

Both sides of the and must be true for the result to be true.

```
age = 20
print(age > 18 and age < 30)
# True → because both are True
```

or

use or if only one of two conditions need to be true.

```
age = 15
print(age > 18 or age == 15)
# True → because the second condition is True
```

not

If you need to invert a boolean variable or comparison, precede it with not.

```
is_sunny = True
print(not is_sunny)
# False → because not True = False
```

Boolean Variables

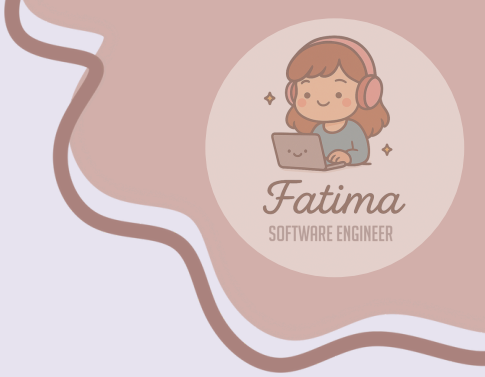
Operator Precedence:

Highest	Operator	Description
↓	()	Parentheses (grouping)
	f(args...)	Function call
	x[index:index]	Slicing
	x[index]	Subscription
	**	Exponentiation
	+x, -x	Positive, negative
	*, /, %	Multiplication, division, remainder
	+, -	Addition, subtraction
	in, not in, <, <=, >, >=, !=, ==	Comparisons, membership, identity
	not x	Boolean NOT
	and	Boolean AND
	or	Boolean OR
Lowest	=	Assignment operator

String Methods

```python
s = "Hello123"
t = "hello"
u = "   "
v = "WORLD"
text = "Python is fun!"

# 1  isalnum() → True if only letters or digits
print(s.isalnum())   # True → contains only letters and digits

# 2  isalpha() → True if only letters
print("Hello".isalpha())  # True → all are letters

# 3  isdigit() → True if only digits
print("1234".isdigit())  # True → all are digits

# 4  islower() → True if all letters are lowercase
print(t.islower())  # True → all letters are small

# 5  isupper() → True if all letters are uppercase
print(v.isupper())  # True → all letters are capital

# 6  isspace() → True if only spaces, tabs, or newlines
print(u.isspace())  # True → only spaces

# 7  substring in s → True if substring exists
print("fun" in text)  # True → "fun" found in text

# 8  count() → number of occurrences
print(text.count("n"))  # 2 → 'n' appears twice

# 9  endswith() → True if text ends with given substring
print(text.endswith("!"))  # True → ends with '!'

# 10  find() → returns index of first occurrence (or -1 if not found)
print(text.find("Python"))  # 0 → found at start

# 11  startswith() → True if text begins with substring
print(text.startswith("Py"))  # True → starts with "Py"
```

while Loop



A while loop repeats a block of code as long as a condition is True. When the condition becomes False, the loop stops.

1 Count-Controlled Loop

A loop that repeats a fixed number of times using a counter variable.

```
counter = 1
while counter <= 10:
    print(counter)
    counter = counter + 1
```

2 Event-Controlled Loop

Repeats until a specific event occurs, such as reaching a target value

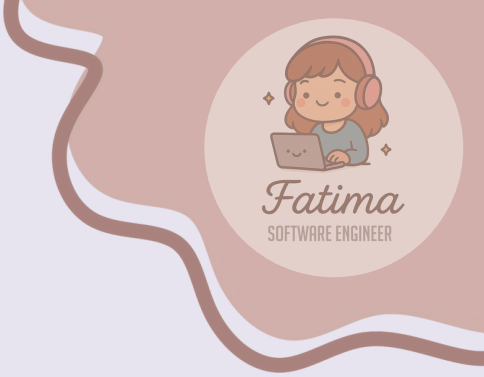
```
counter = 1
while counter <= 10:
    print(counter)
    counter = counter + 1
```

3 Sentinel-Controlled Loop

Uses a special sentinel value to signal the end of input, e.g., -1 means stop.

```
num = int(input("Enter number (-1 to stop): "))
while num != -1:
    print("You entered:", num)
    num = int(input("Enter number (-1 to stop): "))
print("Done!")
```

for Loop



is used to repeat a block of code a specific number of times, or to iterate through the elements of a sequence (like a list, string, or range).

1

Container-Controlled Loop

This type is used to iterate over all elements in a sequence (like a string, list, or tuple).

2

Count-Controlled Loop (Using range())

- Used when you know exactly how many times the loop should repeat.
- The range(start, stop)
- stop - 1

```
# Loop through each character in a string
country = "Oman"
for letter in country:          # Goes through each character automatically
    print(letter)               # Prints one character per line

#  Output:
        # O
        # m
        # a
        # n
```

```
# Loop a specific number of times using range(start, stop)
for i in range(1, 6):           # Loops from 1 up to 5 (not including 6)
    print(i)                    # Prints the current number

# Output:
        # 1
        # 2
        # 3
        # 4
        # 5
```

Nested Loops

- A nested loop is a loop placed inside another loop.
- The outer loop controls how many times the inner loop runs.
- The inner loop repeats completely for each repetition of the outer loop.
- In Python, indentation is used to show nesting.

Example 1: Numeric Nested Loop

```python
#  Multiplication Table
for i in range(1, 6):              # Outer loop → controls rows (1 to 5)
    for j in range(1, 6):          # Inner loop → controls columns (1 to 5)
        print(i * j, end="\t")     # Print product with a tab space
    print()                        # Move to the next line after each row

# Output:
    # 1 2 3 4 5
    # 2 4 6 8 10
    # 3 6 9 12 15
    # 4 8 12 16 20
    # 5 10 15 20 25
```

Example 2: Pattern Nested Loop

```python
# Star Pyramid
rows = 5
for i in range(1, rows + 1):       # Outer loop → controls number of rows
    for j in range(i):             # Inner loop → prints stars per row
        print("*", end="")         # Print stars on the same line
    print()                        # Move to a new line after each row

# Output:
    # *
    # **
    # ***
    # ****
    # *****
```

Functions

- A function is a sequence of instructions with a name
- Functions can receive multiple arguments or it is also possible to have functions with no arguments
- Functions return only one value
- Function Definition: Starts with def and defines the function's name and parameters.
- Parameters: Names inside parentheses in the function definition. They act as placeholders for the values that will be passed later.
- Arguments: The actual values you send to the function when calling it.
- Function Body: Contains the code that performs the operation using the parameters.
- Return Statement: Sends the calculated value back to where the function was called.
- Function Call: Executes the function and provides the arguments.
- Output: Displays the returned result.

```python
# Function to calculate the area of a rectangle
def rectangleArea(length, width):      #  Function definition with PARAMETERS
    area = length * width              #  Function body
    return area                        #  Return value

# Main program
def main():
    l = 5                              # Actual value for length
    w = 3                              # Actual value for width
    result = rectangleArea(l, w)       #  Function call with ARGUMENTS
    print("The area is:", result)      #  Output

main()                                 #  Start the program
```

Functions variable scope

The scope of a variable is the part of the program where that variable is visible or can be used.

Variables can be:

- Local variables → Declared inside a function.
- Global variables → Declared outside any function.

Re-using Variable Names:

- Two functions can have variables with the same name, since their scopes don't overlap.
- Each one is separate and independent.

Local Variables

- Exist only inside the function where they are declared.
- Cannot be accessed by other functions.
- Parameter variables are also local to their function.

```
def main():
    sideLength = 10
    result = cubeVolume()
    print(result)

def cubeVolume():
    return sideLength ** 3
# ❌ ERROR - sideLength not visible here
```

Global Variables

- Declared outside any function.
- Visible to all functions defined after it.
- To modify them inside a function, use the keyword global.

```
balance = 10000     # Global variable

def withdraw(amount):
    global balance  # Access global variable
    if balance >= amount:
        balance -= amount

withdraw(350)
print("Balance =", balance)
```

Lists

A list is a container that stores multiple values in adjacent memory locations.

Defined using square brackets [].

- A list is a sequence of elements, each of which has an integer position or index.
- The first element in the list always has subscript 0.

differences between lists and strings:

- strings are immutable— you cannot change the characters in the sequence.
- Lists are mutable – you can replace one list element with another.

Code:

```
# 1  Create a list
numbers = [10, 20, 30, 40, 50]
# 2  Access and change elements
print(numbers[0])      # First element → 10
numbers[2] = 99        # Change value at index 2
print(numbers)         # [10, 20, 99, 40, 50]

# 3  Length of the list
print(len(numbers))    # 5 elements

# 4  Loop through list
for num in numbers:
    print(num)         # Prints each number

# 5  Negative index
print(numbers[-1])     # Last element → 50

# 6  Add or remove items
numbers.append(60)     # Add at end
numbers.remove(20)     # Remove by value
print(numbers)         # [10, 99, 40, 50, 60]

# 7  Combine or repeat lists
new_list = numbers + [70, 80]  # Concatenate lists
repeat = [1, 2] * 3            # Repeat 3 times
print(new_list)                # [10, 99, 40, 50, 60, 70, 80]
print(repeat)                  # [1, 2, 1, 2, 1, 2]

# 8  Membership test
print(99 in numbers)  # True

# 9  Copy vs alias
a = [1, 2, 3]
b = a                  # alias → same list
c = a[:]               # copy → new list
a[0] = 100
print(b)               # [100, 2, 3] same as a
print(c)               # [1, 2, 3] separate list
```

List Operations

List Methods

- append() → Adds an element to the end of the list.
- insert() → Adds an element at a specific position.
- index() → Returns the position of the first matching element.
- remove() → Deletes the first occurrence of a specific element.
- pop() → Removes and returns an element by its index.
- sort() → Sorts the list in ascending order.

Code:

```
# Create a list of numbers
numbers = [10, 20, 30, 40]
print("Start:", numbers)                    # Original list

# 1  Append → adds an element to the end of the list
numbers.append(50)
print("append(50):", numbers)               # [10, 20, 30, 40, 50]

# 2  Insert → adds an element at a specific index
numbers.insert(2, 25)                       # insert 25 at index 2
print("insert(2,25):", numbers)             # [10, 20, 25, 30, 40, 50]

# 3  Find & Membership → locate an element or check if it exists
idx_40 = numbers.index(40)                  # find position of 40
print("index(40):", idx_40)                 # prints 4
print("30 in numbers?", 30 in numbers)      # True if found
print("99 not in numbers?", 99 not in numbers)  # True if not found

# 4  Remove / Pop / del → different ways to delete items
numbers.remove(20)                          # remove first 20
print("remove(20):", numbers)               # [10, 25, 30, 40, 50]
popped = numbers.pop(0)                     # remove by index and return it
print("pop(0):", popped, "| now:", numbers)  # 10 | [25, 30, 40, 50]
del numbers[1]                              # delete by index (no return)
print("del idx=1:", numbers)                # [25, 40, 50]

# 5  Concatenation (+) → joins two lists
a, b = [1, 2], [3, 4]
c = a + b
print("concat a+b:", c)                     # [1, 2, 3, 4]

# 6  Replication (*) → repeats the list multiple times
rep = [7, 8] * 3
print("replicate [7,8]*3:", rep)            # [7, 8, 7, 8, 7, 8]

# 7  Equality & Inequality → compares both values and order
print("[1,2] == [1,2] ->", [1,2] == [1,2])   # True (same order & values)
print("[1,2] != [2,1] ->", [1,2] != [2,1])   # True (different order)

# 8  Sum / Min / Max → work on numeric lists
nums2 = [5, 2, 9, 1]
print("sum/min/max:", sum(nums2), min(nums2), max(nums2))  # 17 1 9

# 9  Sorting → arrange elements in order
nums2.sort()                                # in-place sorting (ascending)
print("sort() asc:", nums2)                 # [1, 2, 5, 9]
print("sorted(desc):", sorted(nums2, reverse=True))  # [9, 5, 2, 1]
```

Lists With Functions

Lists are mutable, so functions can change their contents directly.

You can create and return an entire list from a function.

Code:

```python
# 1  Function that modifies a list (passed by reference)
def multiply(values, factor):
    """Multiply each element in the list by a given factor."""
    for i in range(len(values)):
        values[i] = values[i] * factor     # modifies original list

# Example:
numbers = [2, 4, 6]
print("Before multiply:", numbers)
multiply(numbers, 3)                       # function changes the original list
print("After multiply:", numbers)          # [6, 12, 18]

# 2  Function that creates and returns a new list
def squares(n):
    """Return a list of square numbers from 0 to n-1."""
    result = []                            # empty list
    for i in range(n):
        result.append(i * i)
    return result                          # return new list

# Example:
sq = squares(5)
print("List of squares:", sq)              # [0, 1, 4, 9, 16]

# 3  Demonstrating list passed by reference
def doubleList(values):
    """Double each value (affects original list)."""
    for i in range(len(values)):
        values[i] *= 2

nums = [1, 2, 3, 4]
print("Before double:", nums)
doubleList(nums)                           # modifies directly
print("After double:", nums)               # [2, 4, 6, 8]

# 4  Returning modified list from a function
def addTen(values):
    """Return a new list with 10 added to each value."""
    result = []
    for v in values:
        result.append(v + 10)
    return result

original = [5, 10, 15]
newList = addTen(original)
print("Original:", original)               # [5, 10, 15]
print("New list:", newList)                # [15, 20, 25]
```