Lecture 1-14: MS Python Programming Course Lectures

Programming Languages and Compilers
Eötvös Loránd University (ELTE), Budapest, Hungary

Lectures

Lectures (cont.)

12. Lecture 8-1: Python Modules
13. Lecture 8-2: Python Environment manager
14. Lecture 9: File handling in Python
15. Lecture 10: Testing, Debugging, Exceptions, Assertions
16. Lecture 11-1: Pythonic idioms
17. Lecture 11-2: Managing Database with Python
18. Lecture 12: Managing API with Python
19. Lecture 13-1: Using Python in Data Science
20. Lecture 13-2: Implementing simple Machine learning projects with Python
21. Lecture 14: Python Recap & Summary

Lecture 1-1: Course overview

Programming Languages and Compilers
Eötvös Loránd University (ELTE), Budapest, Hungary

Course Overview

Python Foundations

- Core programming concepts
- Data structures
- Functions and OOP
- Files and exceptions

Applying Python

- Web APIs
- Databases
- Data science and ML

First learn the fundamentals, then apply them in real-world domains.

Course Structure

- **Lectures**
  - Focus on developing an understanding of key concepts - KNOWLEDGE
- **Practice Labs**
  - Build programming skills through hands-on exercises - SKILL
- **Assignments**
  - Enhance problem-solving abilities by applying knowledge and skills - PROBLEM SOLVING

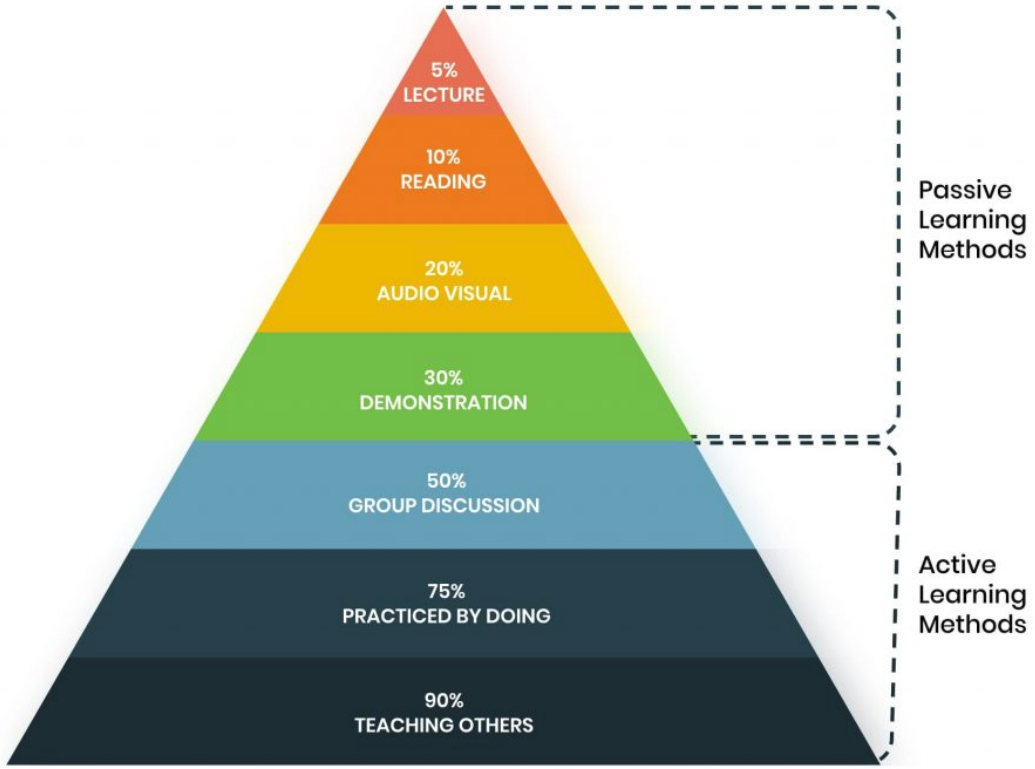
Purpose of the Course

The goal is to guide you in becoming a Python programmer.

Knowledge + Skill + Problem Solving

Programming success requires understanding concepts, practicing skills, and applying them to solve problems.

Learning Pyramid



Learning programming is not passive.

You learn by doing, testing, correcting, and trying again.
Use lectures, labs, assignments, and exams to improve your skills.

Mastering Programming: The Importance of Practice

New to Programming?

Practice. Practice. Practice.

- Programming cannot be absorbed passively
- It requires consistent and active engagement
- Experiment with Python
- Make mistakes and learn from them

Develop Your Learning Methodology

To truly understand programming concepts and build lasting skills, you need to develop your own learning strategy.

- Review examples actively
- Practice beyond the minimum requirement
- Use mistakes as learning opportunities
- Reflect on assignments, tasks, and exams
- Improve your approach step by step

Each task is an opportunity to strengthen both your skills and your learning method.

Your Role in the Learning Process

Your goal is to become proficient in Python.

You are the key player in this process.

The teacher guides, supports, and challenges you, but

the journey is yours.

A Successful Collaboration

Together, we are a team working toward your success.

Success is not about perfection.

It is about consistent progress.

Lecture 1-2: Introduction to Programming with Python

Course Roadmap

- **Foundations of Python**
  - Introduction to Programming with Python
  - Data Types (Language Objects) in Python
  - Basic Language Constructs
  - File Handling & Management
  - Python Package Management
  - Object-Oriented Programming (OOP)
  - Exception Handling
  - Pythonic Expressions
  - Testing & Debugging
- **Using Python in Various Domains**
  - Python in Web Development – API Design
  - Python with Databases
  - Python in Data Science
  - Python in Machine Learning

Lecture 1 Overview

Programming Fundamentals

Course administration

- Structure
- Evaluation
- Learning Tips

Fundamentals of Programming

- **What Does a Computer Do?**
  - Performs instructions
    - A billion instructions per second!
  - Remembers results
    - Hundreds of gigabytes of storage!
- **What kinds of instructions?**
  - **Built-in instructions:** basic operations like storing information, manipulating data, or performing arithmetic operations.
  - **Custom instructions:** Ones that you define as the programmer

Computers only know what you tell them.

Creating custom instructions

1. **Goal(Declarative)**: *WHAT WE WANT TO GET FROM COMPUTER.*
   - Example: Determine if a given number is a palindrome.

2. **Steps to reach goal(Imperative)**: TELLS THE COMPUTER HOW TO DO THINGS.

What Makes a Language?

Natural Language

- **Alphabet:**
  - Letters (A-Z), numerals, punctuation
- **Lexis:**
  - Words, idioms, expressions
- **Syntax:**
  - Grammar rules
- **Semantics:**
  - Meaning of sentences, context

Programming Language

- **Alphabet:**
  - Characters (A-Z, 0-9, special symbols like `#` `$` `_` `{` `}` `;` `:` etc.)
- **Lexis:**
  - Keywords, identifiers, operators
- **Syntax:**
  - Rules for writing valid code (e.g., loops, conditionals)
- **Semantics:**
  - What the code does when executed

What is algorithm?

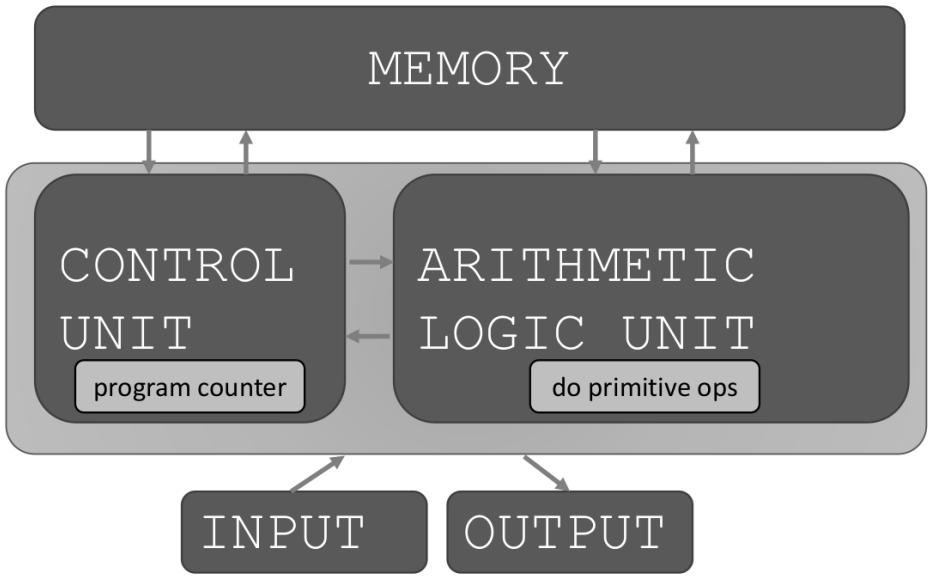
Algorithm = Step 1 + Step 2 + Step 3 ...

Does the computer understand this algorithm?

- Algorithms must be written in a high-level programming language like Python. computer understands and executes the code with the help of a special program called an interpreter or a compiler.
- These programs convert high-level code into machine code.

How Instructions are Executed in a Computer

- **Programs** are stored in the computer's memory.
- The **CPU fetches** instructions from memory one by one.
- Instructions are then **decoded and executed sequentially**.
- Results are stored back in memory or registers.



Compiler vs Interpreter

Compiler

- Translates entire source code into machine code at once. Produces an independent executable.
- Faster execution (pre-translated).
- Errors detected before execution.
- C, C++, Java, Rust.

Interpreter

- Translates and runs code line by line.
- Requires source code for execution.
- Slower execution (translates at runtime).
- Errors detected during execution.
- Python, JavaScript, PHP

Summary

- Source Code -> Compiler ->Executable File -> Execution
- Source Code -> Interpreter -> Line-by-line execution

Programming with Python

- How to write instructions in Python.
- The Python instructions can be categorized into:
  - **Definitions**: Define structures like variables, functions, or classes. These are evaluated when the program runs.
    - Example: `x = 10`, `def my_function():`
  - **Expressions**: Produce values when evaluated.
    - Example: `x + y`, `len("Python")`
  - **Commands**: Execute actions such as controlling the flow of the program or interacting with the user.
    - Example: `print("Hello, World!")`, `if x > 10:`

Summary

- Computers execute instructions and store data
- Programming defines custom instructions
- Algorithms describe solutions
- Programming languages implement algorithms
- Translators convert code into machine instructions

What's Next?

Why Python?

- Focus on solving real problems
- Easy to learn, powerful to use

Your Journey in This Course:

- Understand Python **syntax** and **semantics**
- Learn core programming concepts
- Build programs:
  - from simple scripts
  - to complex applications

Thank You for Your Attention!

A strong foundation leads to clear thinking.
Clear thinking leads to good programs.

Lecture 2-1: The Programming Pipeline: From Data to Computation

Lecture 2-1 Overview

- Programming Pipeline
- From Data to Computations

Example code

```
a = 10
b = 20
sum = a + b
average = (a + b) / 2
print(sum)
print(average)
```

Can you divide this program into three logical parts?

The Programming Pipeline

Input–Processing–Output (IPO)

1. Input – Receiving data
2. Processing – Performing computations or transformations
3. Output – Producing results

```
# Input
a = 10
b = 20

# Processing
sum = a + b
average = (a + b) / 2

# Output
print(sum)
print(average)
```

Input stage

Common Input Sources:

- Hard-coded values
- Command-line interface (CLI)
- Files
- Databases
- Networks
- Other applications
- Sensors / external systems

Key Insight

Input defines:

- Data source
- Data type

These directly influence how the program processes data.

Processing Stage

The processing stage transforms input data into meaningful output.

Processing may involve:

- Arithmetic computations
- Logical evaluations
- Data transformations
- Algorithmic manipulation
- Iterative refinement

Key Insight

The processing stage embodies the **algorithmic logic** of the program, defining how input is systematically transformed into output.

Output Stage

The output stage communicates the results of a program to the user or to other systems.

Output may be directed to:

- Files
- Databases
- Network sockets
- Graphical user interfaces (GUI)
- Other applications

Programming Pipeline

Input → Processing → Output

Input	**Processing**	**Output**
`a = 10` `b = 20`	`sum = a + b` `average = (a + b) / 2`	`print(sum)` `print(average)`

Input	**Processing**	**Output**
Raw materials (flour, sugar, eggs)	Mixing and baking	Final product (cake)

Programs transform data just like factories transform raw materials into finished products.

Factory Assembly Line vs Programming Pipeline

Factory Assembly Line

$$\text{Steel} \rightarrow \text{Cutting} \rightarrow \text{Welding} \rightarrow \text{Painting} \rightarrow \text{Final Car}$$

Programming Pipeline

$$\text{Data/Data Object} \rightarrow \text{Input} \rightarrow \text{Processing (Algorithms)} \rightarrow \text{Output}$$

Key Insight:

- Both systems transform raw input into a final product
- Each stage performs a specific operation
- Output of one stage becomes input to the next

Data vs Data Object

Data

- Raw value only
- No behavior
- Examples:
  - 42
  - "hello"
  - 3.14

Analogy:
Think of it as
content on paper

Data Object

- Value + Type + Behavior
- Examples:
  - 42 (int $\rightarrow$ supports +, -, `.bit_length()`)
  - "hello" (str $\rightarrow$ supports concatenation, `.upper()`)

Analogy:
Think of it as
a smartphone app
(can copy, edit, translate, delete)

From Raw Data to Intelligent Objects

Data (Raw Fact)

Data Object (Value + Type + Behavior)

Why Transform Data?

- Easier to manage
- Easier to manipulate
- Easier to interpret

What is the Disadvantage?

- Slower execution
- More memory usage
- Compared to low-level languages (e.g., C)

Python Perspective

Does Python use Data or Data Objects?

- Python uses **Data Objects**

Python Philosophy:

- Simplicity and readability
- Trades performance for ease of use

From Code to Hardware: `a + b`

Example:

`a = 10` `b = 11` $\Rightarrow$ $a + b$

Hardware Level

- CPU instruction: `ADD`
- Built into processor
- Operates on binary in registers

```
ADD R1, R2
```

C Language Level

- `+` is built-in operator
- Compiler translates to `ADD`
- Direct hardware mapping

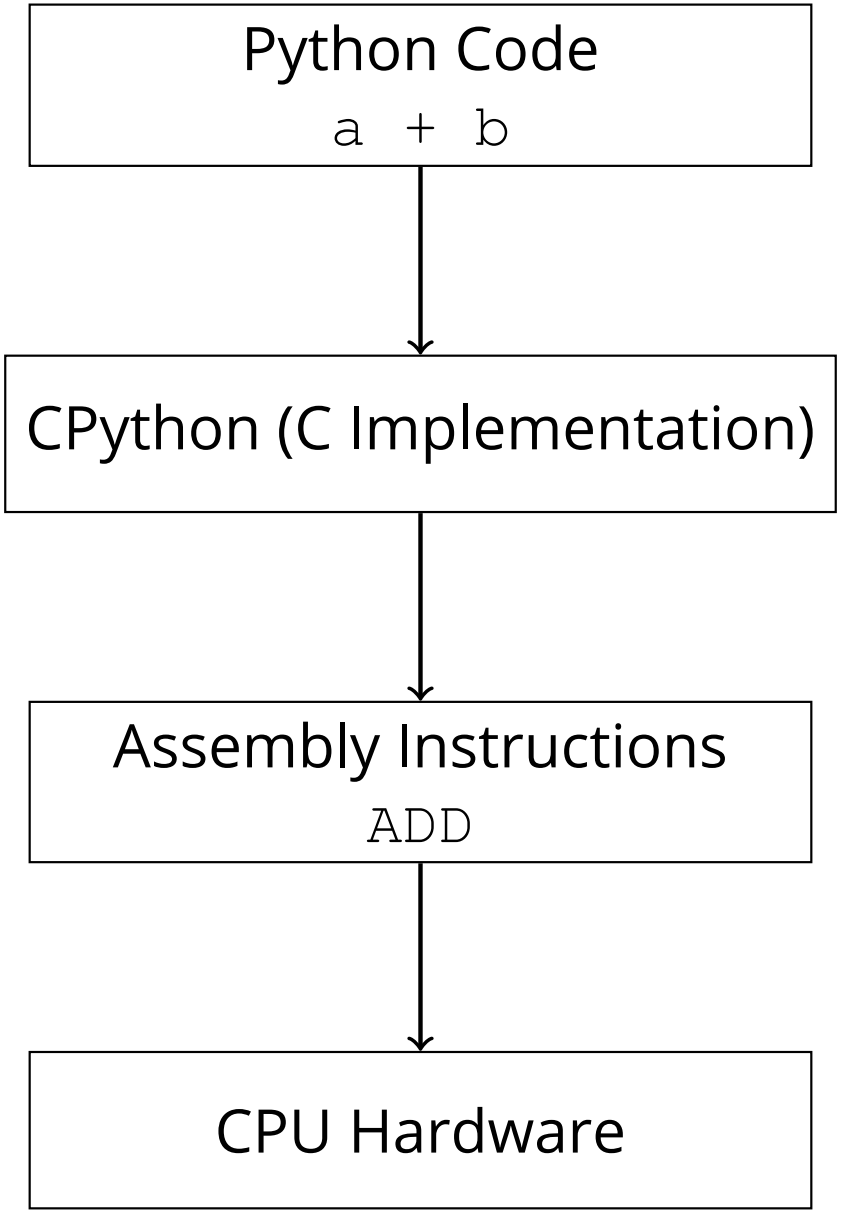
```
int c = a + b;
```

Python Level

- `+` calls object behavior
- `a + b` $\rightarrow$ `a.__add__(b)`
- Implemented in C (CPython)

```
print(a + b)
```

Abstraction Stack



Next Topic: From Data to computation

- Inside Processing
  - Data Types – What kind of data do we have?
  - Type Casting – How do we convert data if needed?
  - Expressions & Operators – How do we compute with data?

Same Operation, Different Levels

Hardware

- Numbers added using CPU instruction: `ADD`

C

- `a + b` → compiler → `ADD`

Python

- `a + b` → interpreter → `__add__()` → `ADD`

Key Insight:

Extra abstraction layer ⇒ slower execution but higher flexibility

C vs Python: How Addition Works

C

- Works with data types (`int`, `float`, `char`)
- Operators map directly to hardware

```
int a = 5, b = 7;
int c = a + b;
```

Direct addition

Python

- Works with data objects
- Operators are method calls

```
a = 5; b = 7
print(a + b)
```

```
a.__add__(b)
```

Object behavior

Key Idea:

C: direct operation vs Python: object-oriented operation

C vs Python: Big Picture

C

- Close to hardware
- Compiled to machine instructions
- High performance
- Manual memory management

Fast but requires more control

Python

- High-level abstraction
- Interpreted language
- Automatic memory management
- Uses objects and dynamic typing

Simple and expressive but slower

Key Trade-off:

$$\text{Performance (C)} \longleftrightarrow \text{Simplicity (Python)}$$

Everything is an Object in Python

Key Idea:

- Every piece of data in Python is an object
- Even simple values like numbers and strings

Each object has:

- Type (what it is)
- Value (data it holds)
- Behavior (methods it supports)

Important:

Type determines possible behavior

Analogy:

- Human - can walk, speak
- Cat - can walk, meow

Objects behave based on their type

Types of Data Objects

1. Scalar Objects

- Represent a single value
- No internal structure (from a programming perspective)
- Examples: `int`, `float`, `bool`

2. Non-Scalar Objects

- Represent collections or sequences of values
- Have internal structure
- Examples: `list`, `str`, `dict`

Next:

We will explore these types in detail

Thank You for Your Attention!

You now know the pipeline and the raw material—data.
Programming is about transforming it into meaningful results.

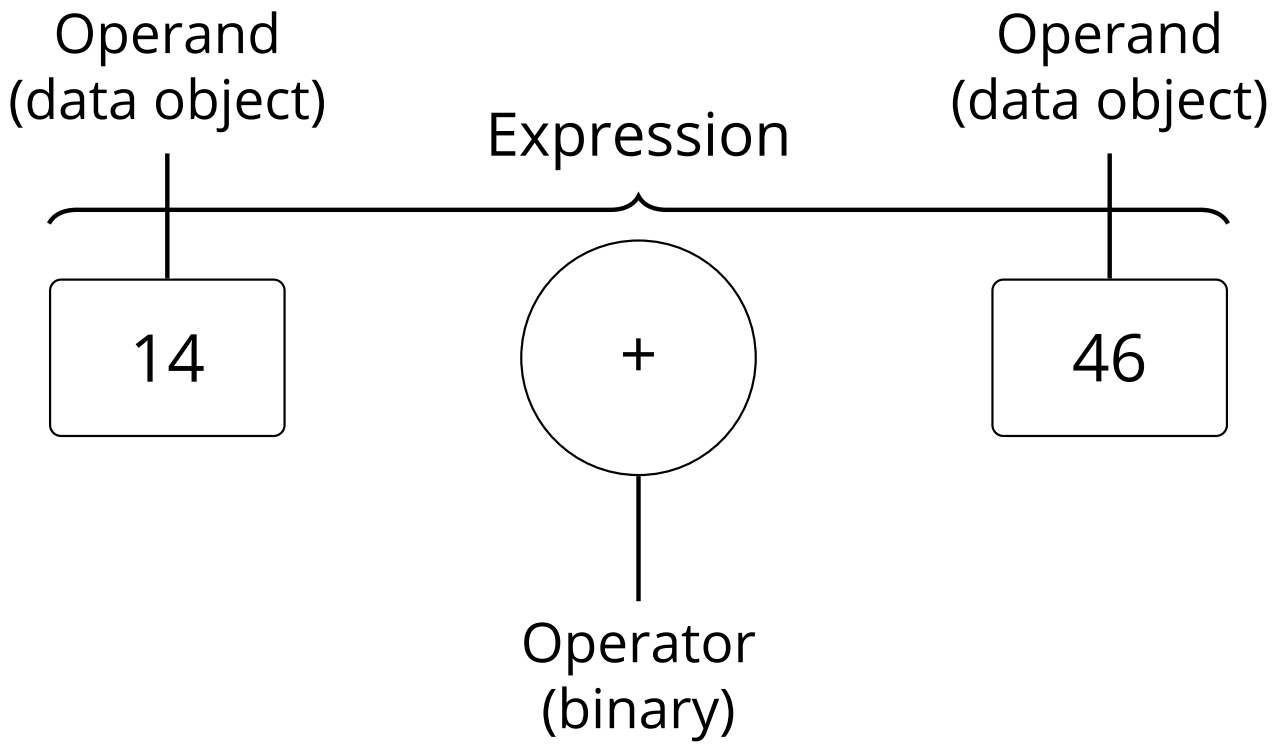
Lecture 2-2: From Data to Computation

Types of Python Instructions

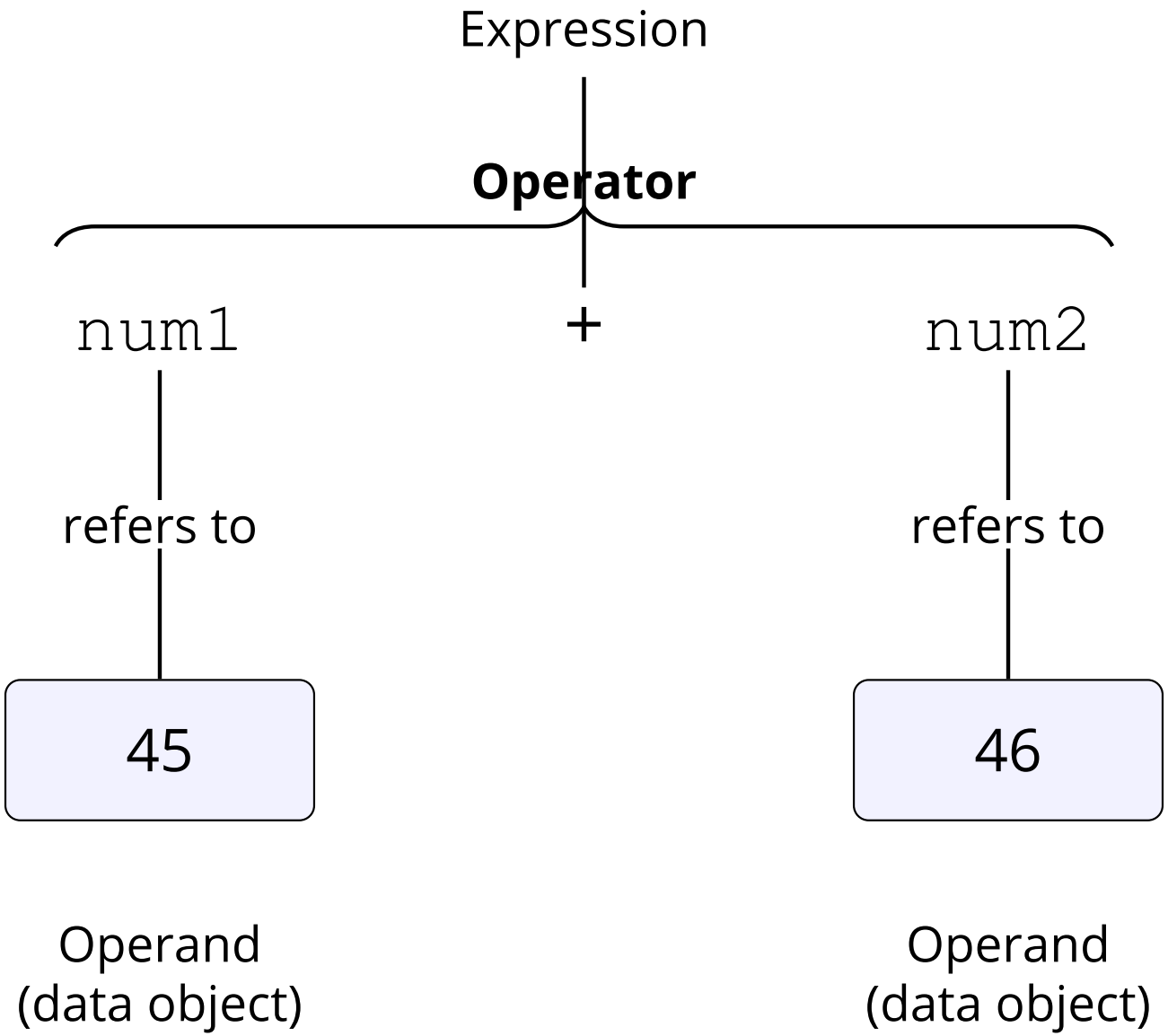
In Python, instructions can be categorized into three main types:

- **Definitions**
  - Create or define structures
  - Examples: variables, functions, classes
  - `x = 10`, `def my_function():`
- **Expressions**
  - Evaluate to produce values
  - `x + y`, `len("Python")`
- **Commands (Statements)**
  - Perform actions or control program flow
  - `print("Hello")`, `if x > 10:`

Programming Expression



Programming Expression: Big Picture



Data Objects(Recap)

- **Data objects** are the fundamental entities used to represent and manipulate data in Python.
- In Python, we do not work with raw data directly — we work with **objects**.
- Every object has:
  - a **value** (the actual data)
  - a **type** (how the data behaves)
- The **type** of an object determines:
  - what operations are allowed
  - how the object behaves in computations

- **Analogy:**
  - *Anna* is a **human** $\rightarrow$ can walk, speak, think
  - *A cat* is an **animal** $\rightarrow$ can walk, meow, hunt
- $\Rightarrow$ **Type defines behavior**

Data Types in Python

Data objects can be categorized into two broad groups:

- **1. Scalar Objects**
  - Represent a **single, atomic value**
  - Do not contain smaller accessible components
  - Typically immutable
  - *Examples*: `int`, `float`, `bool`
- **2. Non-Scalar Objects (Composite Objects)**
  - Represent **collections or structured data**
  - Contain multiple elements that can be accessed or modified
  - Support indexing, iteration, or key-based access
  - *Examples*: `list`, `str`, `dict`, `tuple`
- $\Rightarrow$ **Key Idea:**
  - Scalar $\rightarrow$ single value
  - Non-scalar $\rightarrow$ multiple values (structure)

Scalar Objects

- **Scalar objects** represent a **single, atomic value**
- Common scalar types in Python:
  - **int –** integers, e.g., `5`, `-2`
  - **float –** real numbers, e.g., `3.27`, `-0.5`
  - **bool –** logical values: `True`, `False`
  - **NoneType –** special type with a single value: `None`
- You can inspect the type of an object using:
  - `type(5)` → `int`
  - `type(3.27)` → `float`
- **Key idea:**
  - Scalar types are the **fundamental building blocks** of all computations in Python

Type Conversion (Casting)

Type conversion is the process of changing an object from one type to another.

- **1. Explicit Type Conversion (Casting)**
  - Performed **manually** using built-in functions
  - Gives the programmer **full control** over the conversion
  - **Examples:**
    - `float(3)` $\rightarrow$ `3.0`
    - `int(3.9)` $\rightarrow$ `3` (decimal part is truncated)

Type Conversion (Casting)

- **2. Implicit Type Conversion**
  - Performed **automatically** by Python during operations
  - Ensures **type consistency** in expressions
  - **Example:**
    - `5 + 3.2` $\rightarrow$ `8.2`
    - (the integer `5` is implicitly converted to `5.0`)

- $\Rightarrow$ **Key Idea:**
  - Explicit $\rightarrow$ you control the conversion
  - Implicit $\rightarrow$ Python decides automatically

Operators on `int` and `float`

- **Arithmetic Operators:**
  - `i + j` → addition
  - `i - j` → subtraction
  - `i * j` → multiplication
- **Division Variants:**
  - `i / j` → true division (always returns `float`)
  - `i // j` → floor division
  - `i % j` → remainder (modulus)
- **Exponentiation:**
  - `i ** j` → power operation
- ⇒ **Key Ideas:**
  - Operators define how values are combined
  - If both operands are `int`, the result is usually `int`
  - If at least one operand is `float`, the result becomes `float`
  - True division (`/`) always returns `float`

Expressions

- **Expressions** are combinations of values (objects) and operators that produce a result.
- Every expression:
  - evaluates to a **value**
  - has a corresponding **type**

- **Basic structure:**
  - `<operand>  <operator>  <operand>`

- **Examples:**
  - `3 + 5` $\rightarrow$ `8` (`int`)
  - `7.5 * 2` $\rightarrow$ `15.0` (`float`)
  - `10 / 4` $\rightarrow$ `2.5` (`float`)

- **Result Type Rules:**
  - Operations with `float` produce a `float`
  - `/` always produces a `float`
  - Other operators (e.g., `+`, `*`) may produce `int` or `float`

Question: What is the result of `5 % 3`?

- **Compute:** `5 % 3`
- What do you think the result is?

Question: What is the result type of the following expressions?

- **Expressions:**
  - `type(3 + 5)`
  - `type(7.5 - 5)`
- **What are the result types of these expressions?**

Operator Precedence and Execution Order

- Parentheses are used to tell Python to do these operations first.
- Operator precedence without parentheses:
  - $**$
  - $*$
  - $/$
  - $+$ and $-$ (executed left to right, as they appear in the expression)

Operator Precedence Examples

- Expression 1:

```
3 * 2 ** 2
```

  Result: `12`

- Expression 2:

```
2 + 3 * 4
```

  Result: `14`

- Expression 3:

```
(2 + 3) * 4
```

  Result: `20`

Variables

- A **variable** is a name that refers to an object stored in memory.
- **Assignment (Binding):**
  - A variable is created when a name is **bound** to an object
  - The assignment operator = is used for binding
  - **Examples:**
    - `pi = 3.14159`
    - `pi_approx = 22 / 7`
- **Key idea:**
  - The object (value) is stored in memory
  - The variable name is a **reference** to that object
- You can access the value by using the variable name:
  - `pi`
- **Good practice:**
  - Use meaningful variable names to improve readability and maintainability

Why give names to values?

- To reuse names instead of values.
- Makes it easier to change code later.

```
pi = 3.14159
radius = 2.2
# Calculate the area of a circle
area = pi * (radius**2)
```

The Importance of Meaningful Variable Names

Why does naming matter?

- Code is read not only by the computer, but also by people.
- Clear names make code easier to read, understand, and maintain
- Good naming improves collaboration and reduces errors

Example:

```
# Poor naming
a = 3.14159
b = 2.2
c = a * (b**2)
```

```
# Meaningful naming
pi = 3.14159
radius = 2.2
area = pi * (radius**2)
```

Key Idea: Descriptive names make the code self-explanatory

Variable naming rules

- Can include uppercase and lowercase letters, digits, and the underscore _.
- Cannot start with a digit.
- Variable names are case-sensitive.
  - Example: `Romeo` and `romeo` are different variables.
- Cannot use Python keywords or reserved words.
  - Example: `if`, `while`, `return` are not allowed as variable names.

Which one is valid variable names?

1. `my_variable`, `_count1`, `TotalSum`
2. `1stPlace`, `for`

Multiple binding (assignment)

- Python allows multiple assignments. The statement

```
x, y = 2, 3
```

What will it print?

```
x, y = 2, 3
x, y = y, x
print('x =', x)
print('y =', y)
```

Multiple binding (assignment)

- Python allows multiple assignments. The statement

```
x, y = 2, 3
```

What will it print?

```
x, y = 2, 3
x, y = y, x
print('x =', x)
print('y =', y)
```

It prints

x = 3

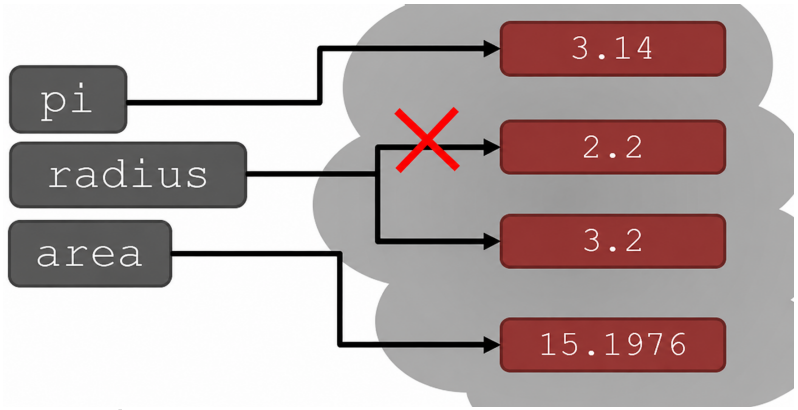
y = 2

Changing Bindings

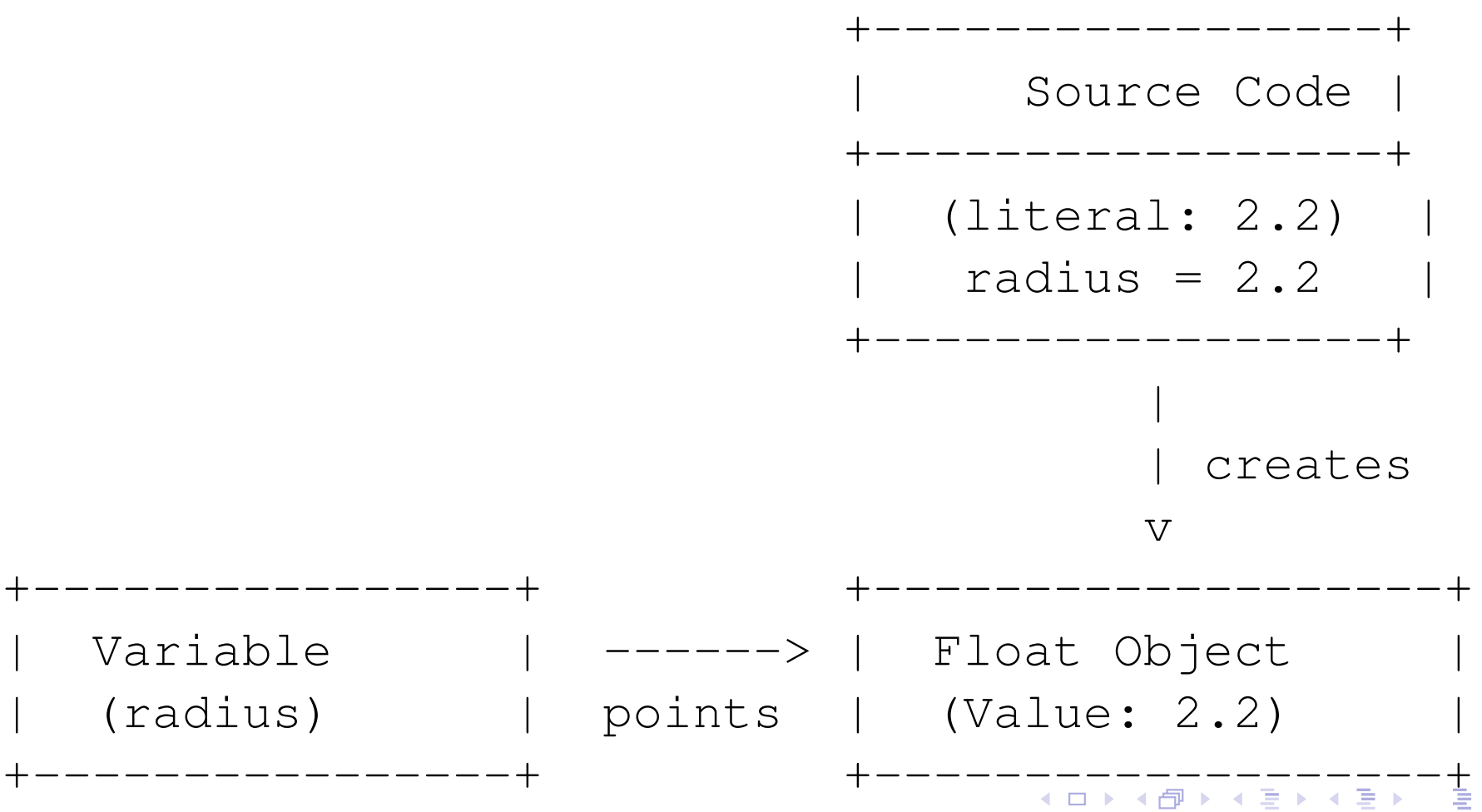
- Variables can be **re-bound** to new objects using assignment.
- Rebinding changes what the variable **refers to**, not the original object.
- The previous object may still exist in memory, but it is no longer accessible through that variable name.

- Expressions are **not automatically re-evaluated**
  - Changing `radius` does **not** update `area`
  - You must explicitly recompute the value

```
pi = 3.14159
radius = 2.2
area = pi * (radius**2)

radius = radius + 1
# area is still based on old radius!
```



Literal, Object, Variable



Non-Scalar Object: String in Python

- Letters, special characters, spaces, digits
- Enclose in double or single quotes:

```
hi = "hello there"
```

- Concatenate strings:

```
name = "Anna"
greet = hi + name
greeting = hi + " " + name
```

- Perform operations on strings as defined in Python docs:

```
silly = hi + " " + name * 3
```

Comparison Operators

- Comparison operators are used to evaluate expressions and return a Boolean result (True or False).
- Variables *i* and *j* can be of type `int`, `float`, or `string`.
- $i > j$
- $i \geq j$
- $i < j$
- $i \leq j$
- $i == j$ *equality test: True if i is the same as j.*
- $i \neq j$ *inequality test: True if i is not the same as j.*

Logic Operators on Booleans

a and *b* are variables with Boolean values.

- `not a`
  True if a is False, False if a is True.
- `a and b`
  True if both a and b are True.
- `a or b`
  True if either or both a and b are True.

a	*b*	*a* and *b*	*a* or *b*
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

Comparison Example

```
score = 85
passing_score = 70
print(score >= passing_score)

is_student = True
has_permission = False
can_enter = is_student or has_permission
print(can_enter)
```

Statically Typed Languages

- **Definition:**
  - Variable types are **explicitly declared**
  - Types are **checked at compile time**
- **Key Properties:**
  - Type of a variable is **fixed**
  - Type errors are detected **before execution**
- **Example (Java):**

```
int number = 5;
number = "Hello";  // compile-time error
```

- **Key Idea:** Safety and early error detection

Dynamically Typed Languages (Python)

- **Definition:**
  - Types are **determined at runtime**
  - No explicit type declaration is required
- **Key Properties:**
  - A variable can be **re-bound to different types**
  - Type checking happens **during execution**
- **Example (Python):**

```
number = 5
number = "Hello"    # valid in Python
```

- **Key Idea:** Flexibility and rapid development

From Data to Computation: Summary

- **Data Objects**
  - **Scalar**: single values (`int`, `float`, `bool`)
  - **Non-scalar**: structured collections (`list`, `str`, `dict`)
- **Variables and Binding**
  - Variables are names that **refer to objects**
  - Assignment creates a **binding** between name and object
- **Expressions and Operators**
  - Expressions combine values and operators to produce results
  - Operators define how computations are performed
- **Type System**
  - Python is **dynamically typed**
  - Type conversion can be **explicit** or **implicit**
- $\Rightarrow$ **Key Insight:**
  - Programs transform data through variables and expressions

Thank You for Your Attention!

You now have more ingredients.
What will you cook?

Lecture 3-1: From Computation to Control Flow

Last Time

- **Programming Pipeline:** Input $\rightarrow$ Processing $\rightarrow$ Output
- **Data and Variables:**
  - Data types (scalar and non-scalar)
  - Variables and binding
- **Computation:**
  - Expressions and operators
  - Type conversion (casting)

Overview

- From Computation to Control Flow
  - Branching computation
  - Iterating computation

Straight-Line Programs

```
hour = int(input("Starting time (hours): "))
mins = int(input("Starting time (minutes): "))
dura = int(input("Event duration (minutes): "))

# Calculate the end time
total_mins = hour * 60 + mins + dura
end_hour = (total_mins // 60) % 24
end_min = total_mins % 60

# Print the result
print(f"End time: {end_hour}:{end_min}")
```

- Execute one statement after another in order.
- Stop when they run out of statements.
- Limited in complexity and interest.

From Computation to Control

We know how to compute...

Perform calculations

Use expressions and operators

But how do we control execution?

When should a computation run?

Which computations should run?

From Straight-Line to Control Flow

- Real programs need more than sequential execution:
  - **Decision-making** $\Rightarrow$ execute code conditionally
  - **Repetition** $\Rightarrow$ execute code multiple times

- This is achieved using **control flow structures**

- **Two main types:**
  - **Branching (Conditionals):** `if`, `elif`, `else`
  - **Iteration (Loops):** `for`, `while`

Conditional Statements

- **Conditional statements** allow a program to make decisions
- **Basic idea:**
  - A **condition** (test) is evaluated
  - The condition results in `True` or `False`
- **Execution flow:**
  - If the condition is `True` $\rightarrow$ execute one block of code
  - If the condition is `False` $\rightarrow$ execute an optional alternative block
- $\Rightarrow$ **Key Idea:** Code execution depends on conditions

Conditional Statements in Python

`if` Statement:

```
if <condition>:
    <code block>
```

`if-else` Statement:

```
if <condition>:
    <code block>

else:
    <code block>
```

`if-elif-else` Statement:

```
if <condition>:
    <code block>

elif <condition>:
    <code block>

else:
    <code block>
```

Approach 1: Simple If-Elif-Else

```
num1 = 15
num2 = 20

if num1 > num2:
    print(f"{num1} is greater than {num2}")
elif num1 < num2:
    print(f"{num1} is less than {num2}")
else:
    print(f"{num1} is equal to {num2}")
```

Approach 2: If-Elif-Else with Different Comparison Order

```
num1 = 15
num2 = 20

if num1 > num2:
    print(f"{num1} is greater than {num2}")
elif num1 == num2:
    print("These are equal")
else:
    print(f"{num2} is greater than {num1}")
```

Approach 3: Nested If-Else within Else Block

```
num1 = 15
num2 = 20

if num1 == num2:
    print("They are equal")
else:
    if num1 > num2:
        print(f"{num1} is greater")
    else:
        print(f"{num2} is greater")
```

Importance of Indentation in Python

- **Python uses indentation to define code structure**
  - Unlike many languages, no braces ({ }) are used
  - Indentation determines **which statements belong together**

- **Why indentation matters:**
  - Defines blocks in `if`, `elif`, `else`, loops, etc.
  - Controls the **flow of execution**
  - Improves readability and maintainability

- **Important:**
  - Incorrect indentation leads to `IndentationError`

Importance of Indentation in Python

Example (C vs Python):

```
# C-style (uses braces)
if (num1 > num2) {
    printf("num1 is greater\n");
} else {
    printf("num2 is greater\n");
}

# Python (uses indentation)
if num1 > num2:
    print("num1 is greater")
else:
    print("num2 is greater")
```

Comparison of Three Code Snippets

- **Key Factors to Consider:**
  - Readability
  - Code structure
  - Number of comparisons made
- **Efficiency:**
  - First and Second Approaches:
    - Both are equally efficient regarding the number of comparisons and overall structure.
    - Utilize a straightforward `if-elif-else` chain.
  - Third Approach:
    - Introduces unnecessary nesting within the `else` block.
    - Slightly less efficient due to added complexity in structure.

Comparison of Three Code Snippets

- **Best Practice:**
  - The first or second approach is preferred due to their simplicity, clarity, and ease of maintenance. They promote clean, efficient code, which is crucial in both small and large projects.
- **Conclusion:**
  - Although the difference in efficiency is minimal, the first approach is generally the best practice. It is favored for its simplicity, readability, and clear logical structure.

Exercise

Write a program that examines three variables—`x`, `y`, and `z`—and performs the following:

- **Find the largest odd number** among `x`, `y`, and `z`.
- **If no odd numbers are present,** print the smallest value of the three.

Solution

```
# Initialize variables
x, y, z = 1, 4, 2

# Start with smallest value
answer = min(x, y, z)

# Check odd numbers
if x % 2 != 0:
    answer = x

if y % 2 != 0 and y > answer:
    answer = y

if z % 2 != 0 and z > answer:
    answer = z

# Print result
print(answer)
```

Control Flow: Loops in Python

- **Loops** allow a program to **repeat a block of code**
- **Why do we need loops?**
  - Avoid writing repetitive code
  - Perform tasks multiple times efficiently
- **Loops are used when:**
  - A condition is true (repeat until condition changes)
  - Iterating over a sequence of values
- **Two main types of loops in Python:**
  - **while loop** $\rightarrow$ repeats based on a condition
  - **for loop** $\rightarrow$ iterates over a sequence

Control Flow: `while` Loops

Repeats code execution as long as a specified condition is **True**

Syntax:

```
while <condition>:
    <expression>
    <expression>
    ...
```

How It Works:

- `<condition>` evaluates to a Boolean value.
- If `<condition>` is `True`, execute all the steps inside the `while` code block.
- Recheck `<condition>` after completing the block.
- Repeat the loop until `<condition>` evaluates to `False`.

Example

Repeat/Print the character 'X' a specified number of times:

```
num_x = int(input("How many times should I print X? \n"))

to_print = ''

# Use a while loop to append 'X' num_x times
while num_x > 0:
    to_print += 'X'
    num_x -= 1

print(to_print)
```

Control Flow in Loops: `break` and `continue`

`break` Statement:

- Terminates the loop immediately.
- The code following the loop is executed.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

`continue` Statement:

- Skips the rest of the current loop iteration.
- Continues with the next iteration.

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Control Flow: `for` Loops

For Loop with Sequences:

- Iterates over a sequence (list, tuple, set, dictionary) or iterable objects such as strings
- Executes the loop body once for each element

Syntax:

```
for item in iterable:
    # code block
```

Example:

```
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(f"I have a {fruit}.")
```

Control Flow: `while` and `for` Loops

Iterating through numbers in a sequence
Using a `while` Loop:

```
# more complicated with while loop
n = 0
while n < 5:
    print(n)
    n = n + 1
```

Using a `for` Loop:

```
# shortcut with for loop
for n in range(5):
    print(n)
```

Control Flow: `for` Loops

Syntax:

```
for <variable> in range(<some_num>):
    <expression>
    <expression>
    ...
```

How It Works:

- Each time through the loop, `<variable>` takes a value from the sequence generated by `range`.
- On the first iteration, `<variable>` starts at the smallest value (0 by default).
- On subsequent iterations, `<variable>` gets the previous value plus 1.
- This continues until the loop has iterated through all values in the range.

Control Flow: `range` Function

Syntax:

```
range(start, stop, step)
```

Default Values:

- `start` = 0 (optional), `step` = 1 (optional)

Behavior:

- Loops until the value is `stop - 1`.

```
mysum = 0
for i in range(7, 10):
    mysum += i
print(mysum)
```

```
mysum = 0
for i in range(5, 11, 2):
    mysum += i
print(mysum)
```

`break` Statement in Python

- Immediately exits the loop it is in.
- Skips remaining expressions in the code block.
- Exits only the innermost loop.

Example:

```
while <condition_1>:
    while <condition_2>:
        <expression_a>
        break
        <expression_b>
    <expression_c>
```

`for` vs `while` Loops

`for` Loops

- Know number of iterations.
- Uses a counter.
- Can end early via `break`.
- Can be rewritten as a `while` loop.

`while` Loops

- Unbounded number of iterations.
- Can use a counter but must initialize before loop and increment inside the loop.
- Can end early via `break`.
- May not be easily rewritten as a `for` loop.

Thank You for Your Attention!

You now have advanced control over your programs.
The next step is to structure your ideas more clearly.

Programs, like us, make decisions, repeat actions,
and skip steps when needed.

Lecture 3-2: From Control Flow to Working with Strings

So far, we learned how to compute and control execution.

Now we apply these concepts to a very important type of data:

Strings

`STRINGS` in Python

- A string is a sequence of case-sensitive characters
- Strings are **iterable** (can loop through characters)
- Strings are **immutable** (cannot be changed after creation)
- Membership operators can be used:
  - `in`, `not in`
- Strings can be compared using ==, >, <
- `len()` returns the number of characters

Example:

```
s = "abc"
len(s)          # 3
"a" in s        # True
```

STRINGS in Python: Indexing

- Square brackets `[]` are used to perform indexing into a string to get the value at a certain index/position.

Example:

```
s = "abc"



s[0]    # evaluates to "a"
s[1]    # evaluates to "b"
s[2]    # evaluates to "c"

s[3]    # trying to index out of bounds, error

s[-1]   # evaluates to "c"
s[-2]   # evaluates to "b"
s[-3]   # evaluates to "a"
```

STRINGS in Python: Slicing

- You can slice strings using `[start:stop:step]`.
- If you give two numbers, `[start:stop]`, the step is `1` by default.
- You can also omit numbers and leave just colons.

Examples:

```
s = "abcdefgh"

s[3:6]      # evaluates to "def", same as s[3:6:1]
s[3:6:2]    # evaluates to "df"
s[::]       # evaluates to "abcdefgh",
                 same as s[0:len(s):1]
s[::-1]     # evaluates to "hgfedcba",
same as s[-1:-(len(s)+1):-1]
s[4:1:-2]   # evaluates to "ec"
```

STRINGS in Python: Immutability

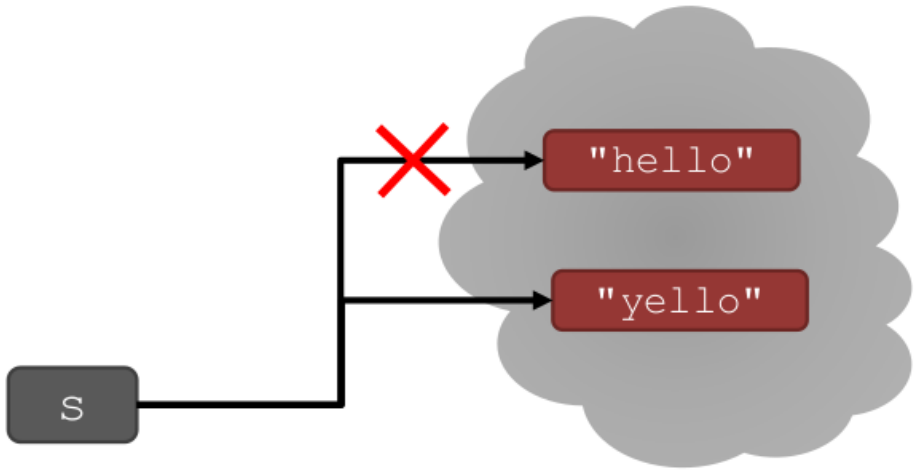
- Strings are **immutable** – they cannot be modified after they are created.

```
s = "hello"
s[0] = 'y'           # gives an error:
                  strings are immutable

s = 'y' + s[1:]      # is allowed,
              s is bound to a new object
```

STRINGS in Python: Immutability

- Strings are **immutable** – they cannot be modified after they are created.

```
s = "hello"
s[0] = 'y'              # gives an error:
                      strings are immutable

s = 'y' + s[1:]       # is allowed,
                 s is bound to a new object
```



Lists in Python: Mutability

- Lists are **mutable** – they can be modified after creation.
- Elements can be changed without creating a new object.

```
numbers = [1, 2, 3]

numbers[0] = 100    # allowed

print(numbers)
```

Output:

```
[100, 2, 3]
```

Unlike strings, lists can be modified in place.

`for` LOOPS RECAP

- `for` loops have a loop variable that iterates over a set of values.

```
for var in range(4):
    <expressions>
```

- `var` iterates over values `0, 1, 2, 3`.
- Expressions inside the loop are executed with each value for `var`.

```
for var in range(4,6):
    <expressions>
```

- `var` iterates over values `4, 5`.

- `range` is a way to iterate over numbers, but a `for` loop variable can iterate over any set of values, not just numbers!

Strings as Iterable Objects

- A string is a sequence of characters
- It is also an **iterable object**
- This means we can iterate over each character:

```
word = "hello"

for char in word:
    print(char)
```

- Each iteration returns one character at a time

Membership Operators in Python

- Membership operators are used to test whether a value exists in a sequence or, more generally, in an iterable object.
- Two main operators:
  - `in` returns `True` if value exists
  - `not in` returns `True` if value does not exist
- Example:

```
"e" in "hello"      # True
"x" in "hello"      # False
"a" not in "hello" # True
```

- Used frequently with strings and lists

CODE EXAMPLE: ROBOT CHEERLEADERS

```
an_letters = "aefhilmnorsxAEFHILMNORSX"

word = input("I will cheer for you! Enter a word: ")
times = int(input("Enthusiasm level (1-10): "))

    i = 0
    while i < len(word):
        char = word[i]

    for char in word:

        if char in an_letters:
            print("Give me an " + char + "! " + char)
        else:
            print("Give me a " + char + "! " + char)

            i += 1

print("What does that spell?")
```

Thank You for Your Attention!

Now that you can control execution,
you can process and transform real data—like text.

Lecture 4: From Processing to Abstraction: Functions

Which text is easier to read?

Unstructured Text

This is a long text without clear structure it contains several ideas it talks about readability and organization and also mentions how difficult it becomes when everything is written continuously without separating thoughts and the reader has to figure out where one idea ends and another begins

Hard to follow and mentally exhausting

Structured Text (Paragraphs)

This is a long text, but it is easier to read.
It is divided into paragraphs.
Each paragraph expresses a clear idea.

Clear, readable, and well-organized

Do you notice a problem?

```
pi = 3.14159

r1 = 2
area1 = pi * (r1 ** 2)

r2 = 5
area2 = pi * (r2 ** 2)

r3 = 10
area3 = pi * (r3 ** 2)

Same computation repeated multiple times
```

How Do We Manage Complexity?

- A program is a **sequence of instructions**
- As programs grow, they become:
  - harder to read
  - harder to understand
  - harder to maintain
- **Solution: Decomposition**
  - Break programs into smaller, manageable pieces
- $\Rightarrow$ We need structure in our programs

Code Organization in Python

How do we structure programs in Python?

- **Functions**
  - Reusable pieces of code that perform a specific task
- **Classes**
  - Group data and behavior together
- **Modules**
  - Organize related code into separate files for better structure
- $\Rightarrow$ **Key Idea:**
  - Break programs into smaller, organized, and reusable components

So far, we wrote programs step by step.

Now we learn how to **structure** and **reuse** code.

Functions

abstraction & decomposition

Functions

- A **function** is a reusable block of code that performs a specific task
- Functions are defined once and executed when called
- **Function Characteristics:**
  - Has a **name**
  - May take **parameters** (inputs)
  - Contains a **body** (instructions)
  - May **return** a result
- $\Rightarrow$ Functions reduce repetition and improve structure, just like paragraphs in writing.

Writing Better Code

- **More code does not mean better code**
- Good programmers are not measured by how much code they write, but by:
  - How much **functionality** they achieve
  - How **clear** and **maintainable** their code is
  - How effectively they **reuse** and **organize** code
- $\Rightarrow$ **Goal:** Write simple, efficient, and reusable code

HOW TO WRITE and CALL/INVOKE A FUNCTION

keyword

name

parameters or arguments

specification, docstring

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

body

later in the code, you call the function using its name and values for parameters

```
is_even(3)
```

Function body

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

Return vs Print

return

- Used **inside functions** only
- Ends the function execution
- Only one `return` is executed
- Code after `return` is **not executed**
- Sends a value back to the **caller**

print

- Can be used **anywhere**
- Does **not stop** execution
- Can have multiple `print` statements
- Code continues after `print`
- Outputs value to the **console**

- $\Rightarrow$ **Key Idea:**
  - `return` gives values back to the program
  - `print` displays values to the user

Important Note: Default Return Value

```
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print("inside is_even")
    return i%2 == 0
```

run some commands

keyword

expression to evaluate and return

- **Warning:** If a function has no `return` statement, Python automatically returns `None`
- `None` represents the **absence of a value**

Function Calls and Scope

- **Parameter Binding:** formal parameters are bound to the values of actual parameters when a function is called
- **Function Scope:** a new **scope (frame/environment)** is created when entering a function
  - Variables inside the function exist only within this scope
  - A scope is a mapping of **names** to **objects**
- **Key Idea:** Functions create their own local environment

```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

formal parameter

actual parameter

Function definition

Main program code
* initializes a variable x
* makes a function call f(x)
* assigns return of function to variable z

Program Execution and Scope

```
def f(x):
    x = x + 1
    print('function scope: x = ', x)
    return x

x = 3
print('Global scope: x = ', x)
z = f(x)
```

Program Execution and Scope

```
def f(x):
    x = x + 1
    print('function scope: x = ', x)
    return x

x = 3
print('Global scope: x = ', x)
z = f(x)
```

What happens when this runs?

Program Execution: What Happens

- Step 1: Function definition - function object created
- Step 2: `x = 3` - global `x` is bound to 3
- Step 3: Print global `x`
- Step 4: Call `f(x)`
  - New local frame is created
  - Parameter `x` is bound to 3
  - `x = x + 1` - local `x = 4`
  - Print local `x`
  - Return 4: assigned to `z`

Objects and Memory in Python

- Every value in Python is an **object** stored in memory
- Each object has a unique **identity** (memory address)
- The built-in function `id()` returns the identity of an object

```
print(id(d))    # identity of function object
print(id(x))    # identity of integer object
```

- **Function object**:
  - Represents executable code
- **Data object (e.g., int)**:
  - Represents stored data (a value)
- $\Rightarrow$ **Key Idea:** Functions and data are both objects in Python

Objects in Python(Recap)

- In Python, every value is an **object**
- Objects have:
  - a value
  - a type
  - an identity (location in memory)

- Examples of objects:
  - **Data objects** $\rightarrow$ represent values (e.g., `int`, `str`)
  - **Function objects** $\rightarrow$ represent computations (callable code)

```
x = 10          # data object
def f():        # function object
    return x

print(type(x))
print(type(f))
```

Objects in Python(Recap)

- In Python, every value is an **object**
- Two important types:
  - **Data objects** $\rightarrow$ represent values
  - **Function objects** $\rightarrow$ represent computations
- **Key Idea:**
  - Functions and data are both objects in Python

Functions as Arguments

- In Python, functions are **first-class objects**
  - Arguments can be of **any type**, including functions
  - Functions can **return any type**: numbers, strings, lists, even **other functions**

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```

call `func_a`, takes no parameters

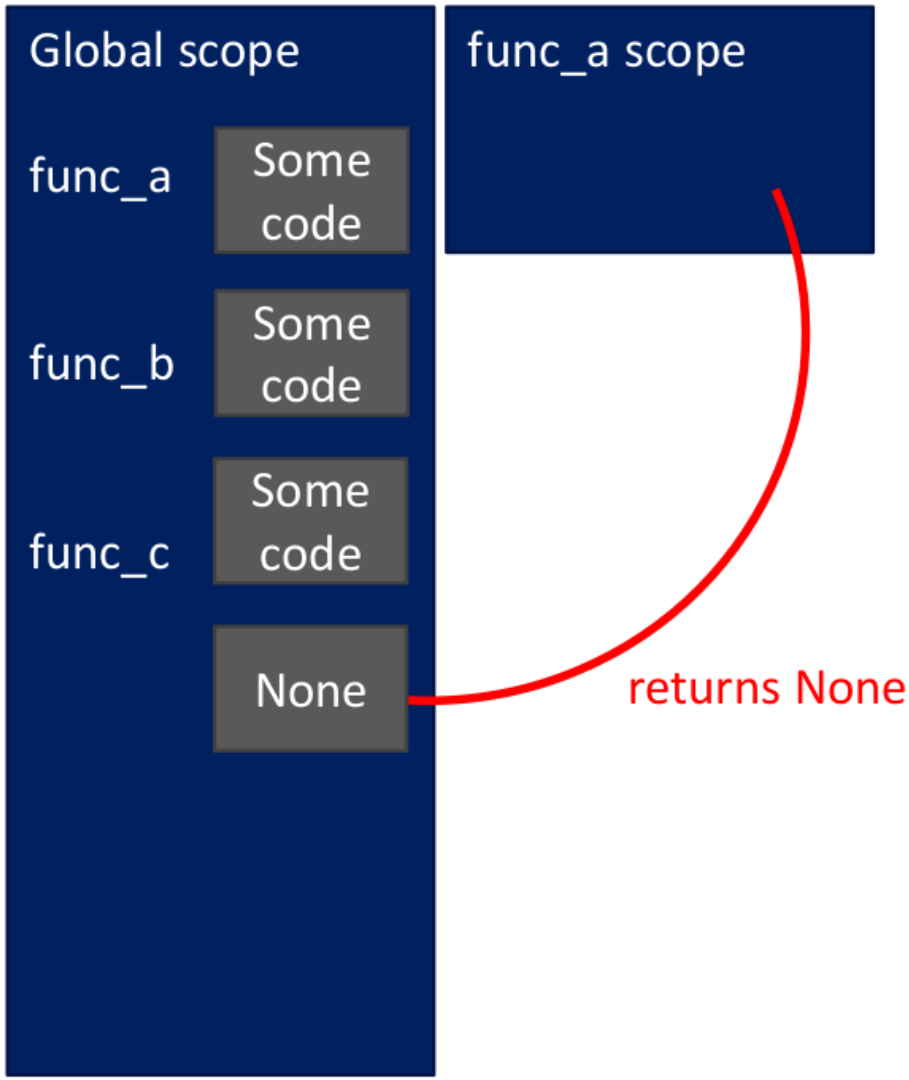
call `func_b`, takes one parameter

call `func_c`, takes one parameter, another function

$\Rightarrow$ Functions can be passed, returned, and used like any other value

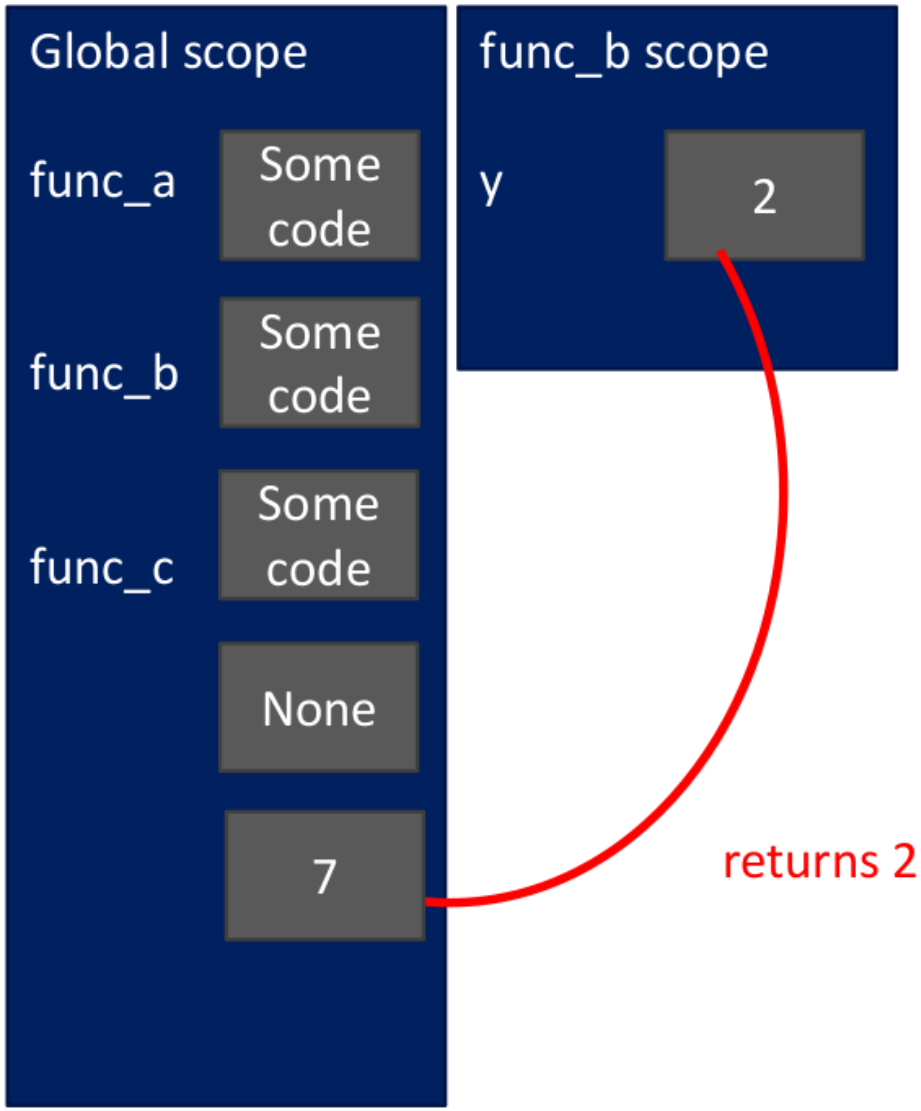
Function Calls and Execution Flow

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



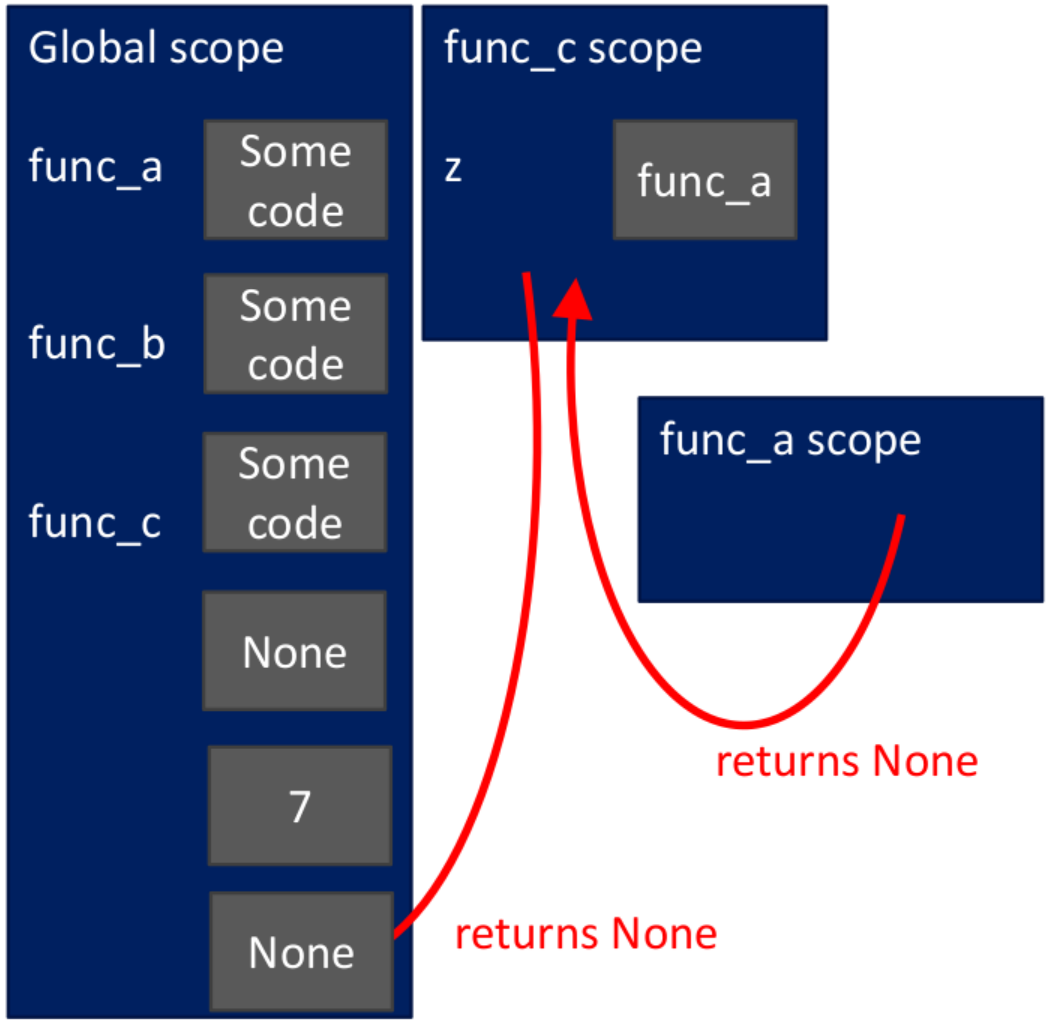
Function Calls and Execution Flow

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



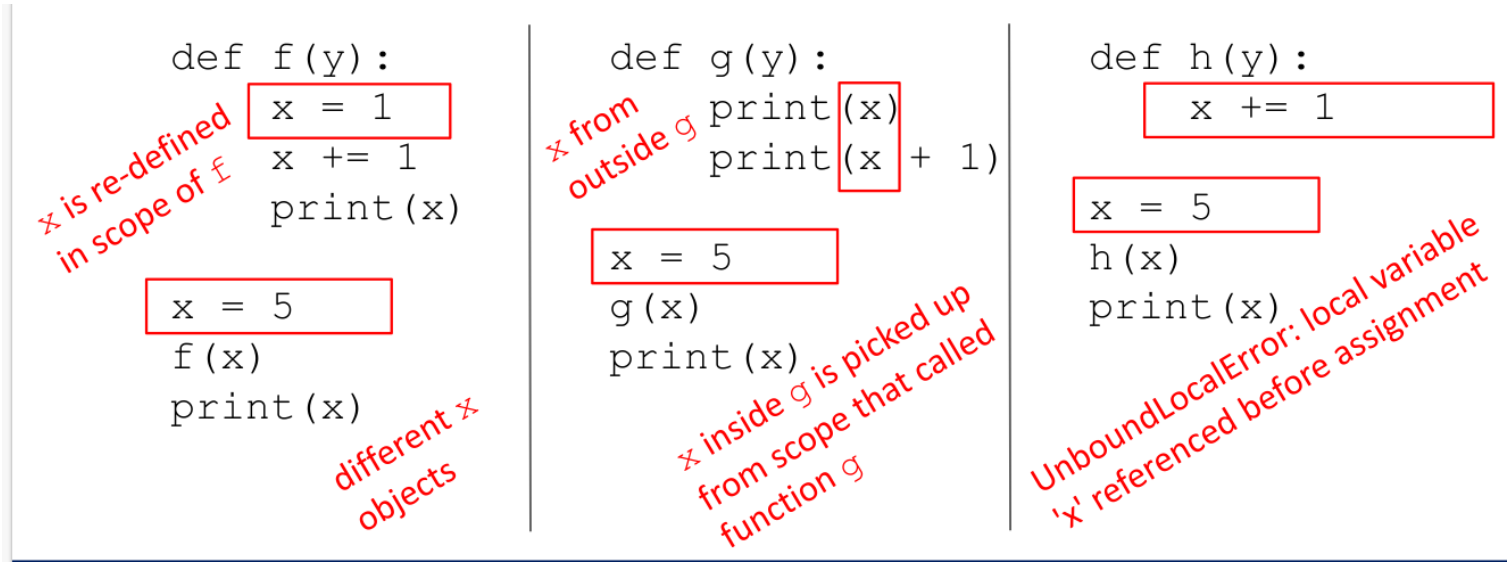
Function Calls and Execution Flow

```
def func_a():
    print 'inside func_a'
def func_b(y):
    print 'inside func_b'
    return y
def func_c(z):
    print 'inside func_c'
    return z()
print func_a()
print 5 + func_b(2)
print func_c(func_a)
```



Global vs Local Variables

- Inside a function, you can **access** variables defined outside
- Inside a function, you cannot modify variables defined outside (without global).
  - Using `global` is generally **not recommended**
  - It can make code harder to understand and maintain



⇒ **Key Idea:** Prefer passing values as parameters instead of using globals

Global vs Local Variables

- Inside a function, you can **access** variables defined outside
- Inside a function, you cannot modify variables defined outside (without global).
  - Using `global` is generally **not recommended**
  - It can make code harder to understand and maintain

```
def f(y):
    x = 1
    x += 1
    print(x)

x = 5
f(x)
print(x)
```

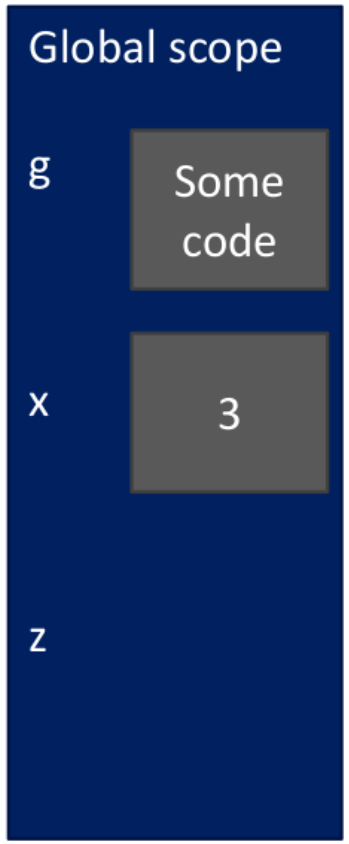
```
def g(y):
    print(x)


x = 5
g(x)
print(x)
```

```
def h(y):
    x += 1


x = 5
h(x)
print(x)
```

Scope Details

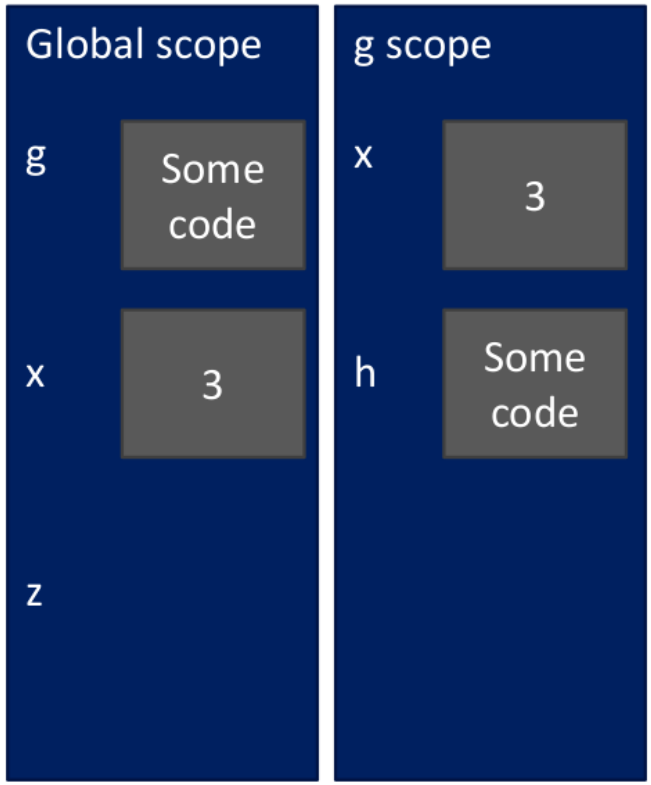
```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```

Some code



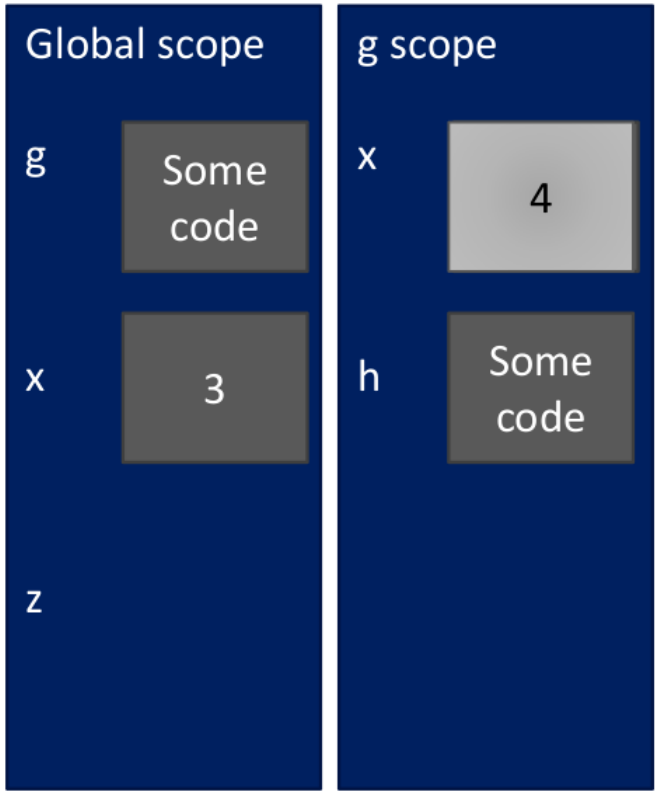
Scope Details

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



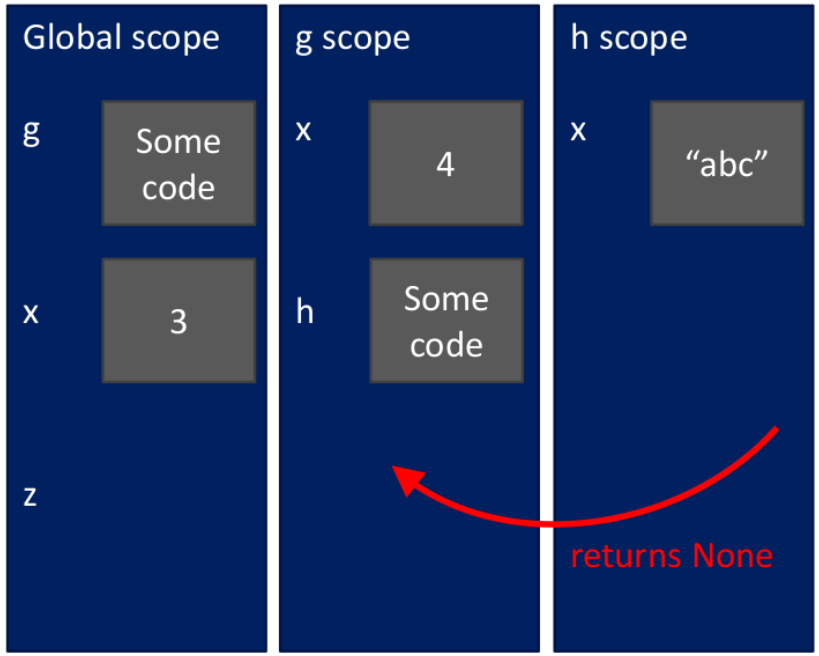
Scope Details

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



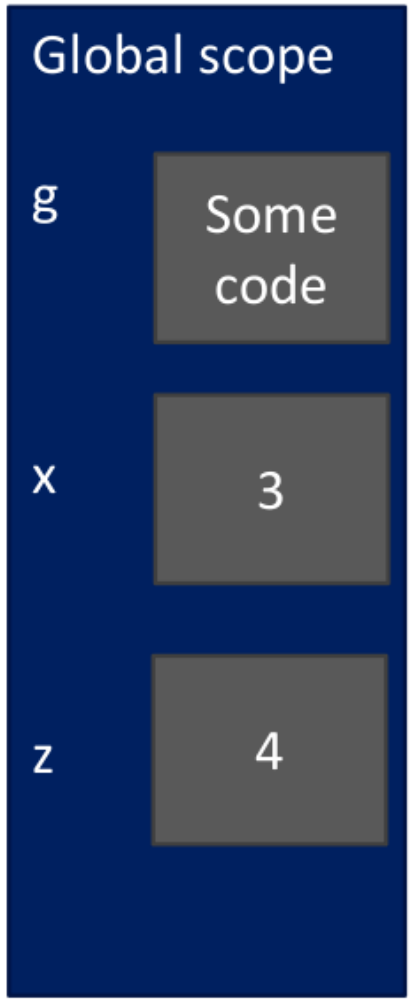
Scope Details

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Scope Details

```
def g(x):
    def h():
        x = 'abc'
    x = x + 1
    print('g: x =', x)
    h()
    return x

x = 3
z = g(x)
```



Summary: Decomposition and Abstraction

- **Decomposition**
  - Break problems into smaller, manageable parts
- **Abstraction**
  - Focus on **what** a function does, not **how** it works
- **Benefit**
  - Debug once, reuse many times with confidence
- $\Rightarrow$ **Key Idea:**
  - Decompose with functions
  - Abstract the details
  - Build reliable and reusable code

Like building with Lego blocks:
Each piece is a function — test it once, reuse it anywhere, and combine pieces to build larger systems.

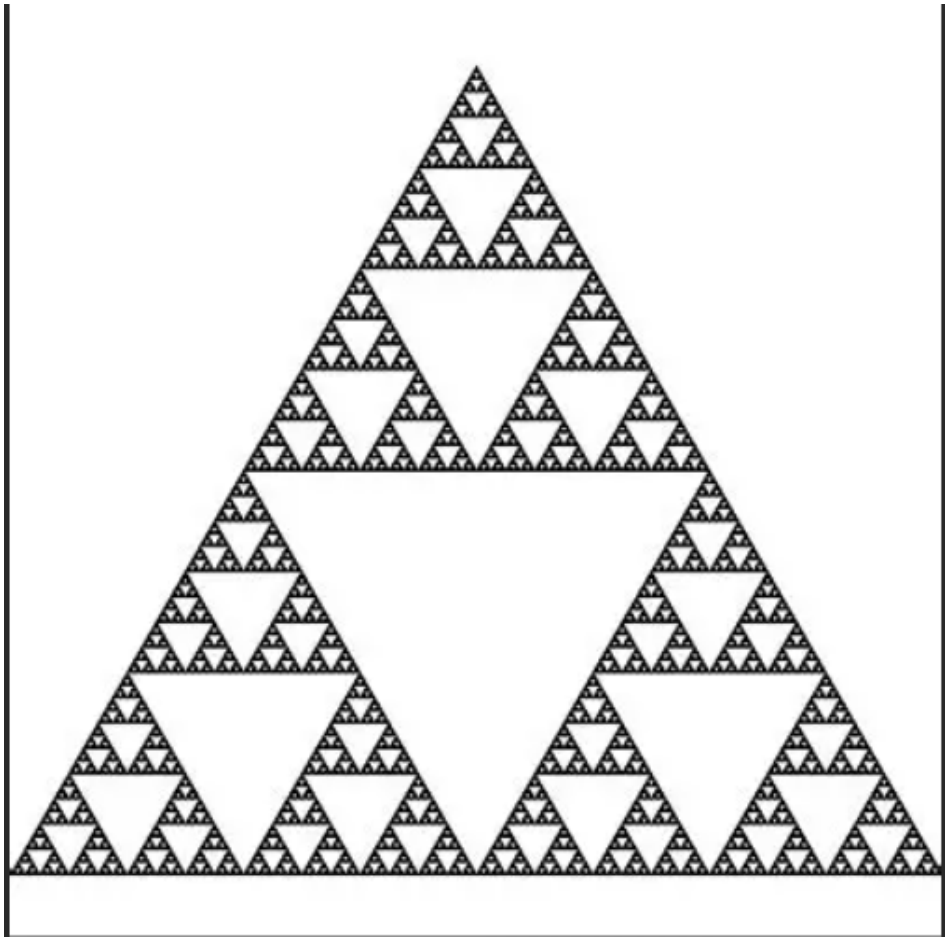
Thank You for Your Attention!

You are no longer just writing code—you are organizing ideas.

Lecture 5-1: Recursive Thinking in Programming

Recursion Concept

Recursion is the process of repeating items in a self-similar way.



Recursive Thinking in Programming

Semantically (in programming):

- A technique where a function calls itself

Algorithmically:

- A strategy to solve problems using:
  - Divide-and-Conquer
  - Decrease-and-Conquer

Relation to Loops (Iteration):

- Recursion is another way to perform repetition
- Similar to `for` / `while` loops
- Requires a **base case** (like a stopping condition)

Core Idea:

Reduce a big problem into smaller versions of the same problem

Key Components of Recursion

Base Case(s):

- The simplest situation that can be solved directly
- Stops the recursion and prevents infinite calls

Recursive Step:

- The function calls itself with a modified input
- Each step simplifies the problem
- Moves the computation closer to the base case

A correct recursive function must always reach a base case.

Repetition: Imperative vs Declarative Thinking

Imperative Thinking (Loops):

- Focus on **how** to compute
- Step-by-step state updates (`num`, `rev_num`)
- Control flow via loop

Declarative Thinking (Recursion):

- Focus on **what** the problem is
- Define solution in terms of smaller subproblems
- Avoid explicit state updates

```
num = 12321
orig_num = num
rev_num = 0

while(num > 0):
    rem = num % 10
    rev_num = rev_num * 10 + rem
    num = num // 10
```

Two Ways to Express Repetition: Iteration and Recursion

Iteration

- Uses `while` / `for` loops
- Relies on **state variables**
- State is updated on each iteration

Recursion

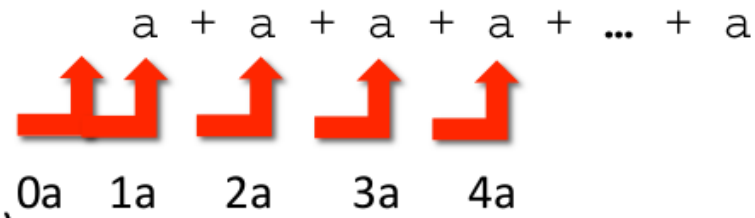
- Uses function calls instead of loops
- Relies on the **call stack**
- State is managed implicitly across calls

Both approaches can solve many of the same problems—just with different ways of thinking.
Anything you can write with iteration, you can also write with recursion (and vice versa).

Multiplication – Iterative solution

Recursion is the process of repeating items in a self-similar way.

- "multiply `a * b`" is equivalent to "add `a` to itself `b` times"
- capture **state** by
  - an **iteration** number (`i`) starts at b
    `i` ← `i - 1` and stop when 0
  - a current **value of computation** (`result`)
    `result` ← `result + a`




```
def mult_iter(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result
```

iteration

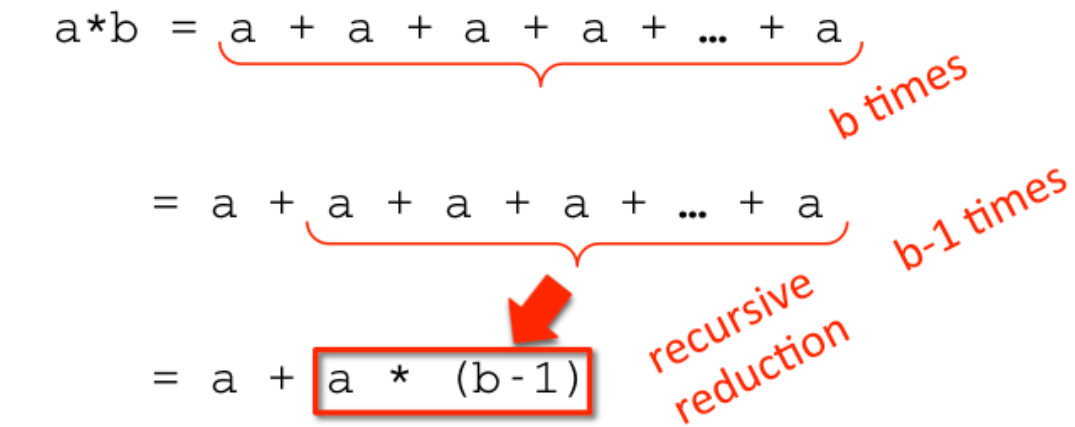
current value of computation, a running sum

current value of iteration variable

Multiplication – Recursive solution

Recursion is the process of repeating items in a self-similar way.

- **recursive step**
  - think how to reduce problem to a **simpler/ smaller version** of same problem
- **base case**
  - keep reducing problem until reach a simple case that can be **solved directly**
  - when b = 1, a*b = a




```
def mult(a, b):
    if b == 1:
        return a
    else:
        return a + mult(a, b-1)
```

base case

recursive step

Factorial and Recursion

Definition:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

For what n do we know the factorial directly?

$n = 1 \quad \Rightarrow \quad$ `if n == 1: return 1` (*basecase*)

How do we reduce the problem?

Rewrite in terms of a smaller problem to reach base case:

$$n! = n \times (n-1)!$$

`else: return n * factorial(n-1)` (*recursivestep*)

Recursion Example: Function Scope

Recursive Function:

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)

print(fact(4))
```

Understanding Function Scope:

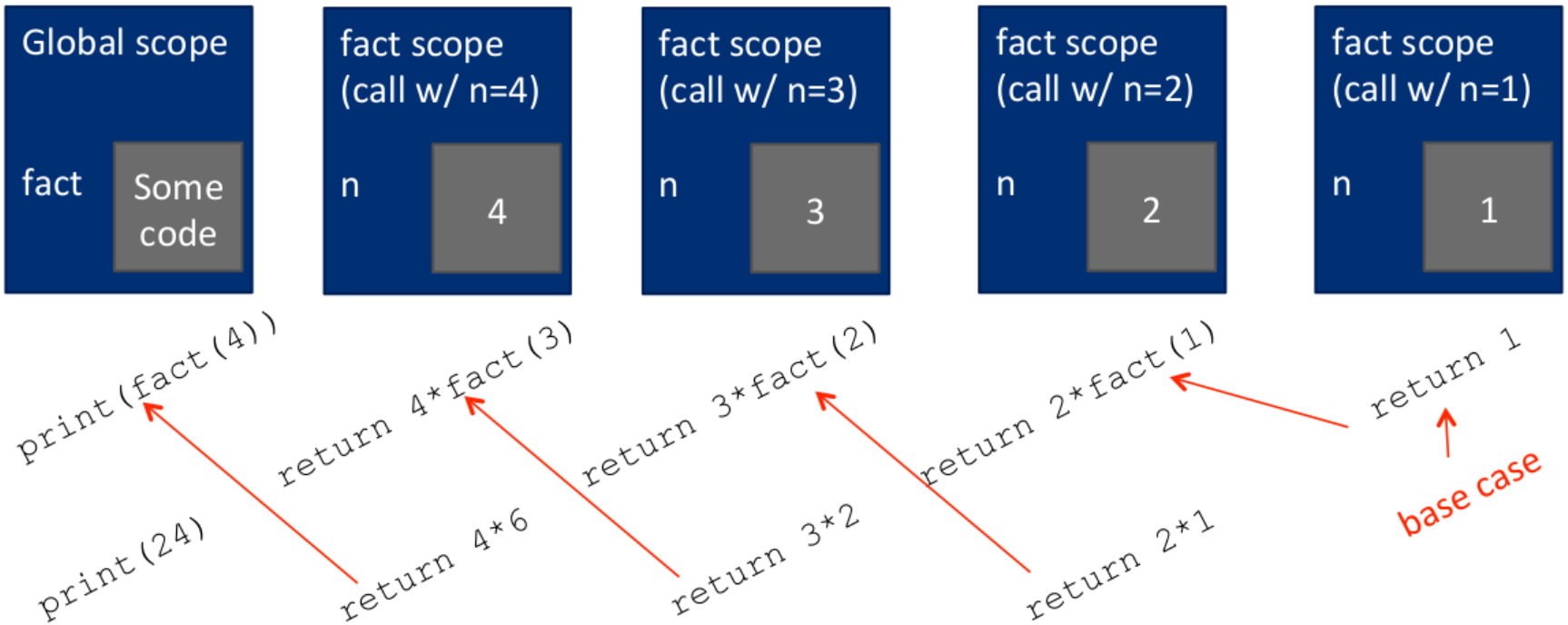
- Each function call creates a new **stack frame**
- Each frame has its own value of `n`
- Calls are stacked until the base case is reached
- Then results are returned step-by-step

Think of it as a stack of function calls

Recursion is the process of repeating items in a self-similar way.

RECURSIVE FUNCTION SCOPE EXAMPLE

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
```



Recursion is the process of repeating items in a self-similar way.

- each recursive call to a function creates its **own scope/environment**
- **bindings of variables** in a scope are not changed by recursive call
- flow of control passes back to **previous scope** once function call returns value

using the same variable names but they are different objects in separate scopes

Iteration vs Recursion

Iteration

- Uses `for` or `while`
- Needs a loop condition
- Stores state in variables
- Stops when the condition fails

Recursion

- Function calls itself
- Needs a base case
- Stores state in the call stack
- Stops when the base case is met

Iteration vs Recursion

Iteration

```
def fact_iter(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

Recursion

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n * fact(n-1)
```

Observations:

- Recursion may be simpler and more intuitive
- Efficient from a **programmer's perspective**
- May be less efficient from a **computer's perspective** (call stack overhead)

Rewrite it in a Recursive Way

Iterative Version:

```
def reverse_num(num):
  rev_num = 0

  while(num > 0):
    rev_num = rev_num*10 + num % 10
    num = num // 10

  return rev_num

print(reverse_num(123))
```

Recursive Version:

```
def reverse_num(num, rev_num=0):

    if num == 0:
        return ...

    else:
        return reverse_num(?, ?)

print(reverse_num(123))
```

Rewrite it in a Recursive Way

```
def reverse_num(num, rev_num=0):

    if num == 0:
        return rev_num

    else:
        return reverse_num(num//10, rev_num*10 + num % 10)

print(reverse_num(123))
```

Recursive Trace: reverse_num(12321)

Call Stack Unwinding:

```
return 12321  <- reverse_num(0, 12321)
  return 12321  <- reverse_num(1, 1232)
    return 12321  <- reverse_num(12, 123)
      return 12321  <- reverse_num(123, 12)
        return 12321  <- reverse_num(1232, 1)
          return 12321  <- reverse_num(12321, 0)
```

Values are returned step-by-step as the recursion unwinds

Recursion on Non-Numerics: Palindrome Check

Definition:

- A palindrome is a string that reads the same forwards and backwards

Famous Examples:

- “Able was I, ere I saw Elba”
- “Are we not drawn onward, we few, drawn onward to new era?”

Idea: Compare first and last characters, then recurse on the substring

Recursive Palindrome Check: Key Steps

Preprocessing (Input Transformation):

- Remove punctuation and non-letter characters
- Convert all characters to lowercase

Base Case:

- A string of length 0 or 1 is a palindrome

Recursive Case:

- If the first and last characters match:
  - Recursively check the middle substring
- Otherwise:
  - Not a palindrome

- ‘Able was I, ere I saw Elba’ → ‘ablewasiereisawleba’
- `isPalindrome('ablewasiereisawleba')` is same as
  - `'a' == 'a'` and `isPalindrome('blewasiereisawleb')`

Divide and Conquer

Key Idea:

- Solve a hard problem by breaking it into smaller subproblems

Characteristics:

- Subproblems are easier to solve than the original
- Each subproblem is similar to the original problem
- Solutions of subproblems are combined to solve the original

$$\textit{Break} \rightarrow \textit{Solve} \rightarrow \textit{Combine}$$

Recursion: Summary

What is Recursion?

- A technique where a function calls itself
- Solves problems by reducing them into smaller subproblems

Key Components:

- **Base Case:** stops recursion
- **Recursive Step:** moves toward the base case

How It Works:

- Uses the **call stack** to manage state
- Builds up calls, then resolves them (unwinding)

Recursion vs Iteration:

- Recursion: declarative, problem decomposition
- Iteration: imperative, state updates with loops

Thank You for Your Attention!

“Break the problem into smaller parts, solve them, and combine the results.”

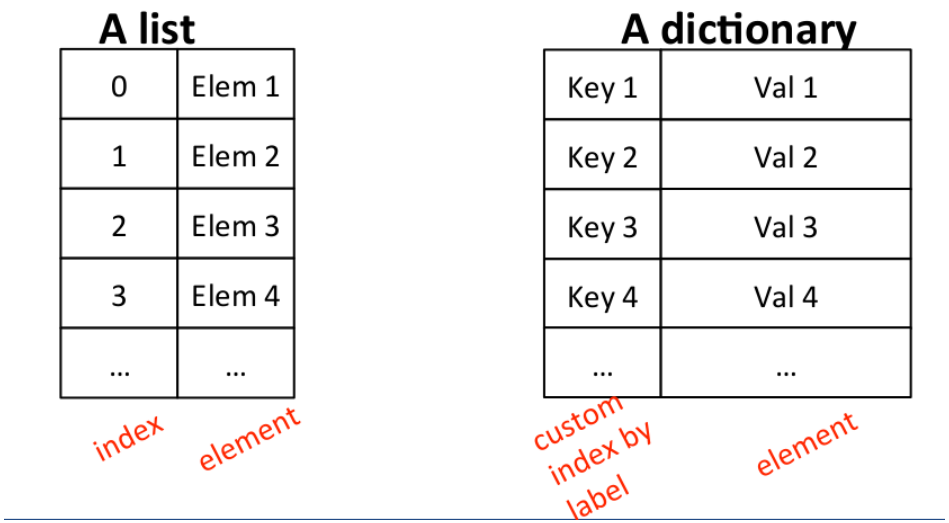
Lecture 5-2: Dictionary

Dictionary

Why use a dictionary?

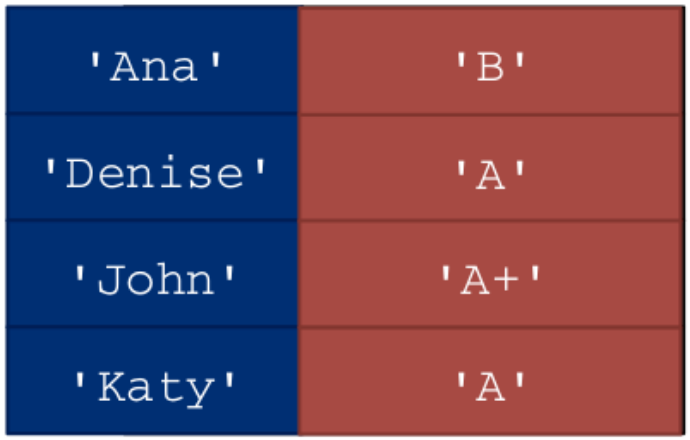
- Allows **direct access** to items of interest using **keys** (not just integers).
- Lets you store all related data in **one unified structure** — no need for separate lists.

```
student = {
   "name":  "Alex",
   "id":  1023,
   "grade":  "A"
}

print(student["name"])
→ Alex
```



A python dictionary

- store pairs of data
  - key
  - value



empty dictionary

```
my_dict = {}
```

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

key1 val1 key2 val2 key3 val3 key4 val4

DICTIONARY LOOKUP

- similar to indexing into a list
- **looks up** the **key**
- **returns** the **value** associated with the key
- if key isn't found, get an error

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

`grades['John']` → evaluates to `'A+'`

`grades['Sylvan']` → gives a `KeyError`

DICTIONARY OPERATIONS

'Ana'	'B'
'Denise'	'A'
'John'	'A+'
'Katy'	'A'
'Sylvan'	'A'

```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- **add** an entry

```
grades['Sylvan'] = 'A'
```

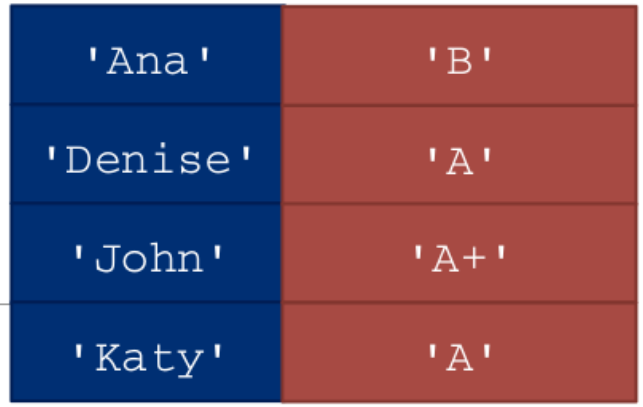
- **test** if key in dictionary

```
'John' in grades      → returns True
'Daniel' in grades    → returns False
```

- **delete** entry

```
del(grades['Ana'])
```

DICTIONARY OPERATIONS




```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
```

- get an **iterable that acts like a tuple of all keys** (no guaranteed order)

  `grades.keys()` → returns `['Denise','Katy','John','Ana']`

- get an **iterable that acts like a tuple of all values**

  `grades.values()` → returns `['A', 'A', 'A+', 'B']` (no guaranteed order)

Dictionary Keys and Values

Values

- Can be **any type** — mutable or immutable
- Can have **duplicates**
- Can even be **lists** or **other dictionaries**

Keys

- Must be **unique**
- Must be **immutable**
- Types: `int`, `float`, `string`, `tuple`, `bool`
- Be careful using `float` as keys

Important: No order is guaranteed for keys or values.

```
d = {4:{1:0}, (1,3):"twelve",
     'const':[3.14,2.7,8.44]}
```

List vs Dictionary

List

- Ordered sequence of elements
- Look up elements by an **integer index**
- Indices have an order
- Index is always an **integer**

Dictionary

- Matches **keys** to **values**
- Look up one item by another item
- No order is guaranteed
- Key can be any **immutable type**

Analyzing Word Frequency with Dictionary

Goal

- Create a frequency dictionary mapping `str : int`
- Count how many times each word appears

Task

- Find the word(s) that occur the most
- Also return how many times they appear

Requirements

- Use a **list** in case multiple words have the same highest frequency
- Return a tuple:

  `(words_list, highest_freq)`

Example Output

```
(['the', 'and'], 5)
```

Dictionary Summary

- A **dictionary** stores data as **key–value pairs**
- Provides **fast access** to values using keys
- **Keys**
  - Must be **unique** and **immutable**
- **Values**
  - Can be of **any type**
  - Can be duplicated
- No guaranteed **order** of elements
- Supports operations:
  - Add, update, delete elements
  - Check membership using `in`
  - Access using keys

Key Idea: *Dictionaries map meaningful keys to values for efficient data retrieval.*

Thank You for Your Attention!

"Programs must be written for people to read,
and only incidentally for machines to execute."

— Harold Abelson

Lecture 6: Object-oriented programming with Python

Agenda

- Python paradigm
- Objects: types, attributes, methods
- OOP essentials; creating/using/destroying objects
- Data abstraction and encapsulation
- Built-in functions and classes
- User-defined functions and classes
- Case study: a simple Queue ADT
- Summary and performance notes (Python vs C vs NumPy)

Python - Paradigm

- Imperative/Procedural: functions, loops, conditionals.
- Functional: first-class functions, HOFs, `lambda`, `map`/`filter`/`reduce`.
- Declarative: list comprehensions, generator expressions.
- Object-Oriented: classes, inheritance, polymorphism.
- Event-Driven: common in GUI/web frameworks (e.g., Flask/Django).

Question: So... which paradigm does Python really follow?

Multi-paradigm

Many modern languages are multi-paradigm — choose the one that fits the problem.

Python Data and Objects

Python supports many different kinds of data:

- `1234, 3.14159, "Hello", [1, 5, 7, 11, 13], {"CA": "California", "MA": "Massachusetts"}`

Key Concept: Object

- Each piece of data is an **object** (data object)
- An object is an **instance of a type (class)**

Examples:

- `1234` → instance of `int`
- `"Hello"` → instance of `str`
- `[1, 5, 7, 11, 13]` → instance of `list`
- `{"CA": "California", "MA": "Massachusetts"}` → instance of `dict`

Types and Objects in Python

Key Ideas:

- Every data type (`int, str, list, dict`) is defined as a **class**
- Every value you create is an **instance (object)** of a class

Example 1: Integer

```
x = 1234

print(type(x))
```

Output: `<class 'int'>`

```
print(isinstance(x, int))
```

Output: `True`

Example 2: List

```
l = [1,2,3,4]

print(type(l))
```

Output: `<class 'list'>`

```
print(isinstance(l, list))
```

Output: `True`

Object Lifecycle in Python

An object goes through several stages during its lifetime:

- **Creation**
  - New objects are created from a type (class)
  - Example: `x = 10`, `l = [1,2,3]`
- **Manipulation**
  - Objects can be used, modified, or passed to functions
- **Destruction**
  - Objects are explicitly destroyed using del or just “forget” about them python system will reclaim destroyed or inaccessible objects – called “garbage collection”

Objects as Data Abstraction

Hide internal details and expose only what is needed to interact

- **Internal Representation**
  - Data attributes (state of the object)
- **Interface**
  - Methods (functions/procedures)
  - Define behavior while hiding implementation

C (Procedural)

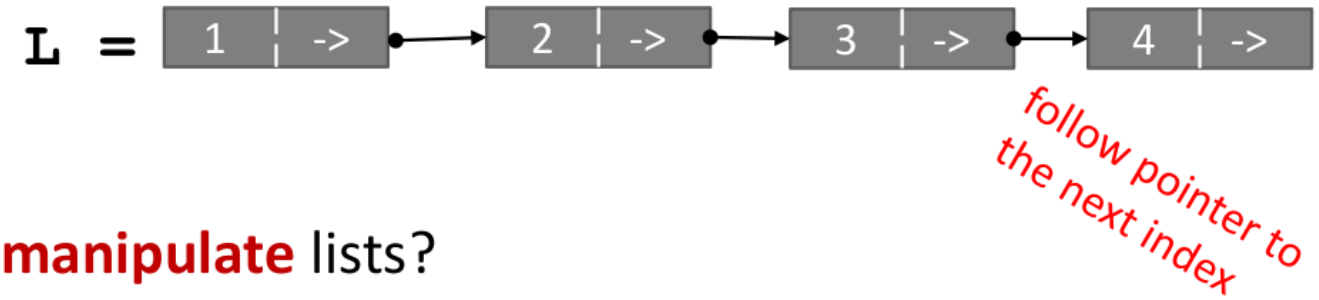
- Data = `value`
- Type is separate from value
- Limited abstraction
- Example:
  - `int x = 10;`

Python (Object-Oriented)

- Data = `object (value + type + behavior)`
- Type is part of the object
- Strong abstraction
- Example:
  - `x = 10`

EXAMPLE: [1,2,3,4] has type list

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`

- internal representation should be private
- correct behavior may be compromised if you manipulate the internal representation directly

Advantages of OOP

- **Bundle data and behavior together**
  - Data and procedures are packaged in objects
  - Interactions happen through well-defined interfaces
- **Divide-and-conquer development**
  - Implement and test the behavior of each class separately
  - Increased modularity reduces complexity
- **Classes make it easy to reuse code**
  - Many Python modules define new classes
  - Each class has a separate environment, so there is no collision between function names
  - Inheritance allows subclasses to redefine or extend a selected subset of a superclass's behavior

Creating and Managing Objects

```
# Create a list and manage elements
lst = [1, 2, 3]
lst.append(4)          # Add an element

# Delete a variable (frees the reference)
del lst

# Create and modify a variable
var = 4
var = float(var)       # Convert int -> float
del var

# Work with files
file = open("file_name", "w")   # Open for writing
file.write("Hello!")            # Write content
file.close()                    # Close the file
```

From Using Objects to Creating Our Own

So far, we have been creating objects of built-in Python types such as:

- `int`, `list`, `str`, `dict`

Now the real question is: How can we create our own class (type) in Python?

And once we create it, how do we use it?

Let us build our own type.

Creating and Using Your Own Types with Classes

Creating a Class

- Define the class name
- Specify attributes and methods
- Describe how objects of this type should behave
- Example: Python developers implemented the `list` class

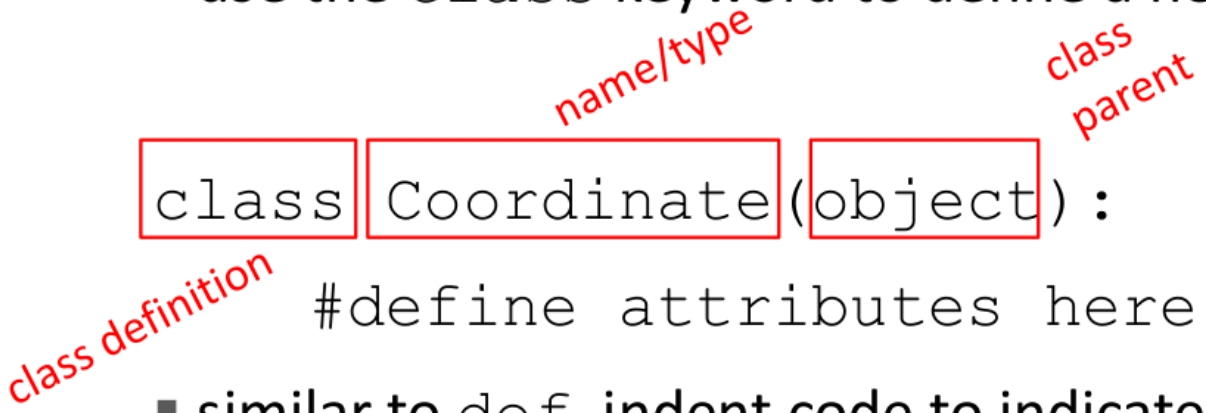
Using a Class

- Create new object instances
- Store data inside objects
- Perform operations using methods
- Example:
  - `L = [1, 2]`
  - `len(L)`

Today: You will learn both how to define a class and how to use its objects.

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type



- similar to `def`, indent code to indicate which statements are part of the **class definition**
- the word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
  - `Coordinate` is a subclass of `object`
  - `object` is a superclass of `Coordinate`

Everything in Python is an Object

Python Type Hierarchy

```
object
  int
  str
  list
  dict
  Student
  Coordinate
```

Defining a Class

```
class Coordinate:
    pass
```

or explicitly:

```
class Coordinate(object):
    pass
```

- Built-in types already exist in Python
- We can also define our own classes
- Every class ultimately derives from `object`

Your own classes become new Python types.

What Are the Attributes?

A class defines the data and behavior (method) of its objects.

Data Attributes

- Describe the state of an object
- Usually represented by other objects stored inside it
- Example: a `Coordinate` object consists of:
  - `x`
  - `y`

Methods (Procedural Attributes)

- A class defines the data and behavior of its objects.
- Example:
  - You could define a distance between lists if you wanted.

Attributes define both what an object knows and what it can do.

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- first have to define **how to create an instance** of object
- use a **special method called `__init__`** to initialize some data attributes

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

special method to create an instance __ is double underscore

what data initializes a Coordinate object

parameter to refer to an instance of the class

two data attributes for every Coordinate object

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

create a new object of type `Coordinate` and pass in 3 and 4 to the `__init__`

use the dot to access an attribute of instance `c`

- data attributes of an instance are called **instance variables**
- don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
  - convention is to use **`self`** as the name of the first argument of all methods
- the "**.**" **operator** is used to access any attribute
  - a data attribute of an object
  - a method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):
    # code here
```

method def

Using the class:

- conventional way

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(c.distance(zero))
```

object to call method on

name of method

parameters not including `self` (`self` is implied to be `c`)

- equivalent to

```
c = Coordinate(3,4)
zero = Coordinate(0,0)
print(Coordinate.distance(c, zero))
```

name of class

name of method

parameters, including an object to call the method on, representing `self`

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **uninformative** print representation by default
- define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- you choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def distance(self, other):
        x_diff_sq = (self.x-other.x)**2
        y_diff_sq = (self.y-other.y)**2
        return (x_diff_sq + y_diff_sq)**0.5
    def __str__(self):
        return "<"+str(self.x)+","+str(self.y)+">"
```

name of special method

must return a string

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
>>> print(c)
<3,4>
>>> print(type(c))
<class __main__.Coordinate>
```

return of the __str__ method

the type of object c is a class Coordinate

- this makes sense since

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

a Coordinate is a class

a Coordinate class is a type of object

- use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
True
```

SPECIAL OPERATORS

- +, -, ==, <, >, len(), print, and many others

https://docs.python.org/3/reference/datamodel.html#basic-customization

- like `print`, can override these to work with your class
- define them with double underscores before/after

```
__add__(self, other)    →    self + other
__sub__(self, other)    →    self - other
__eq__(self, other)     →    self == other
__lt__(self, other)     →    self < other
__len__(self)           →    len(self)
__str__(self)           →    print self
```

... and others

Final Takeaways

- **Python is multi-paradigm** — supports imperative, functional, object-oriented, and more.

Key idea: choose the paradigm that fits the problem.

Final Takeaways

- **Python is multi-paradigm** — supports imperative, functional, object-oriented, and more.
- **Everything in Python is an object** — even numbers, functions, and modules.

Key idea: choose the paradigm that fits the problem.

Final Takeaways

- **Python is multi-paradigm** — supports imperative, functional, object-oriented, and more.
- **Everything in Python is an object** — even numbers, functions, and modules.
- **Every object has** a **type (class)**, **attributes**, and **methods**.

Key idea: choose the paradigm that fits the problem.

Final Takeaways

- **Python is multi-paradigm** — supports imperative, functional, object-oriented, and more.
- **Everything in Python is an object** — even numbers, functions, and modules.
- **Every object has** a **type (class)**, **attributes**, and **methods**.
- **Built-in types (classes)** include: `int`, `float`, `str`, `list`, `dict`, `tuple`, `set`, and file objects like `io.TextIOWrapper`.

Key idea: choose the paradigm that fits the problem.

Thank You for Your Attention!

Good code is not just about making things work — it is about organizing ideas clearly.

Well-structured code is the key to scalable software.

Lecture 7: Advanced concepts in OOP

RECAP: OOP Basics

Everything in Python is an Object

- Every entity in Python is an object (e.g., `int`, `str`, `list`, `dict`)
- Each object has a **type**, **data**, and **behavior**
- Abstract data types through classes

Objects

- Objects are **instances** of classes
- Classes define:
  - **Attributes** (data)
  - **Methods** (behavior)
- Internal implementation is hidden $\rightarrow$ abstraction

The Beauty of OOP

- **Single template (class)** $\rightarrow$ **Multiple standalone objects**
- Objects share structure and behavior
- Each object holds its own data and operates independently

Implementing vs Using a Class

write code from two different perspectives

Implementing the Class

- Define a new object type
- Specify **data attributes**
  - What is the object?
- Define **methods**
  - How to use the object?

Focus: Design and structure

Using the Class

- Create **instances** (objects)
- Call methods on objects
- Perform operations using objects

Focus: Application and usage

Two Perspectives: *Build the tool vs Use the tool*

Class → Object Creation

Class: `human`

Attributes: eyes, ears, brain, legs

Methods: `see()`, `listen()`, `think()`, `walk()`

⇓

Objects

person1
person2
person3

Class: `coordinate`

Attributes: x, y

Methods: `distance()`, `move()`

⇓

Objects

point1 (2,3)
point2 (5,7)
point3 (0,0)

Why Use OOP and Classes of Objects?

- **Mimic real life**
  - Model real-world entities (e.g., human, car, student)
- **Group similar objects**
  - Objects of the same type share structure and behavior
  - Example: all students have name, ID, and actions

OOP helps us think about programs the way we think about the real world

Real World $\rightarrow$ **Program Model**

`Class: Student`

$\Downarrow$

student1 student2 student3

Groups of Objects Have Attributes (Recap)

Data Attributes (State)

- How do we represent the object?
- **What it is**

Examples:

- `coordinate` → x, y
- `animal` → age, name

Describe the state of the object

Procedural Attributes (Behavior / Methods)

- How do we interact with the object?
- **What it does**

Examples:

- `coordinate` → `distance()`
- `animal` → `make_sound()`

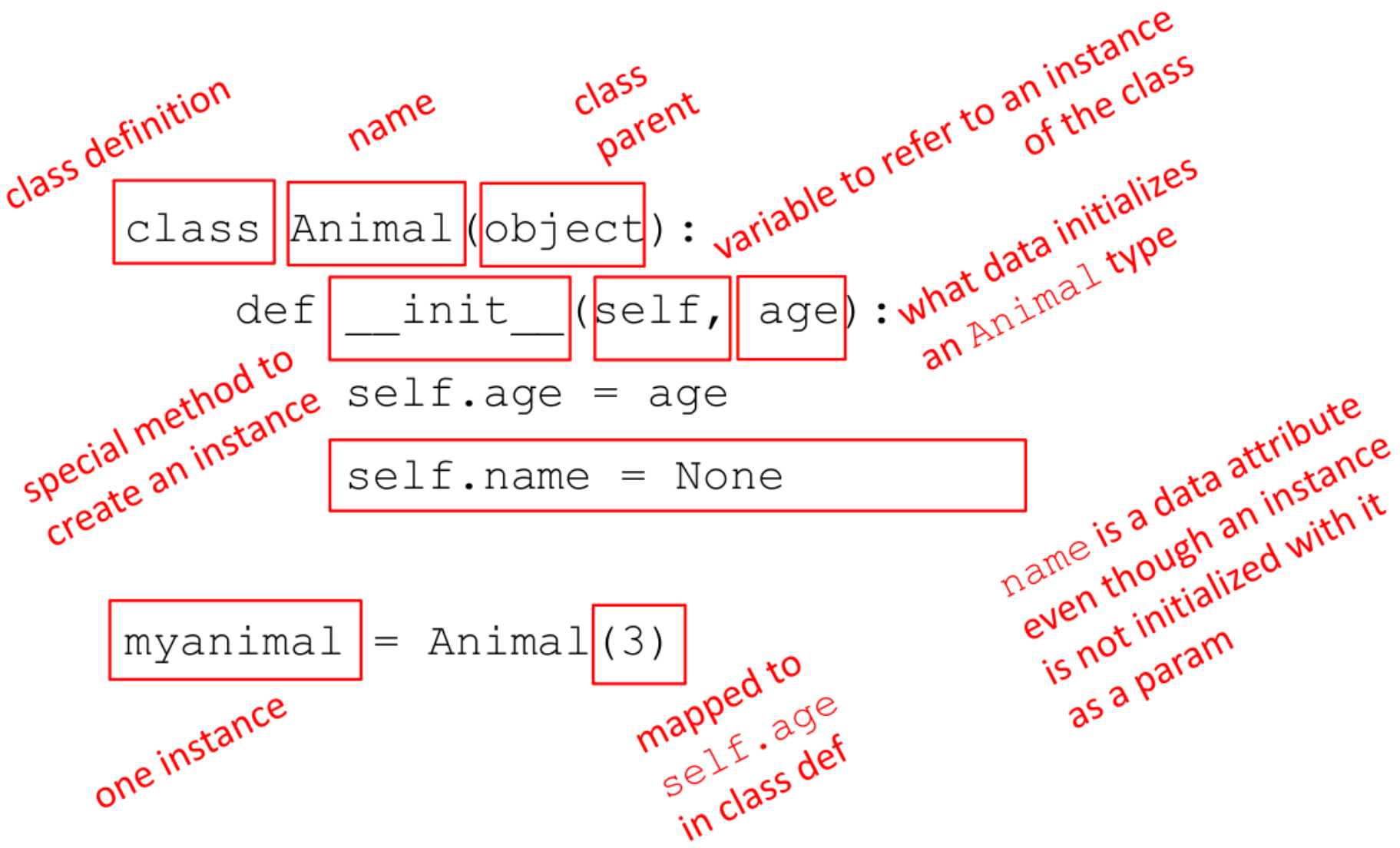
Define behavior and operations

Objects = Data (state) + Behavior (methods)

Some advanced concepts in OOP

- Getter/Setter
- Pythonic Getter/Setter: `@property`
- Information hiding
- Class variables
- Inheritance
- Multiple Inheritance and MRO

HOW TO DEFINE A CLASS (RECAP)



GETTER AND SETTER METHODS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

```
a = Animal()
print(a.get_age())
a.set_age(10)
```

controlled access to data attributes from outside the class.

- Getters read data
- Setters modify data safely

Pythonic Getter/Setter: `@property`

What is `@property`?

- A decorator that allows a method to be accessed like an attribute
- Used to implement getter and setter methods
- Provides controlled access to object data

Example: `@property`

```
class Animal(object):
    def __init__(self, age):
        self._age = age
        self._name = None

    # Getter for age
    @property
    def age(self):
        return self._age

    # Setter for age
    @age.setter
    def age(self, newage):
        self._age = newage

a = Animal(10)
print(a.age)
a.age = 10
```

Access like a variable, control like a method.

AN INSTANCE and DOT NOTATION (RECAP)

- instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods) though it is better to use getters and setters to access data attributes

```
a.age
```

- access data attribute
- allowed, but not recommended

```
a.get_age()
```

- access method
- best to use getters and setters

INFORMATION HIDING

- author of class definition may change data attribute variable names

replaced `age` data attribute by `years`

```
class Animal(object):
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- if you are accessing data attributes outside the class and class definition changes, may get errors
- outside of class, use getters and setters instead
- use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

PYTHON NOT GREAT AT INFORMATION HIDING

- allows you to access data from outside class definition
  - `print(a.age)`
- allows you to write to data from outside class definition
  - `a.age = 'infinite'`
- allows you to create data attributes for an instance from outside class definition
  - `a.size = "tiny"`
- it's not good style to do any of these!

DEFAULT ARGUMENTS

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

HIERARCHIES

Animal

People

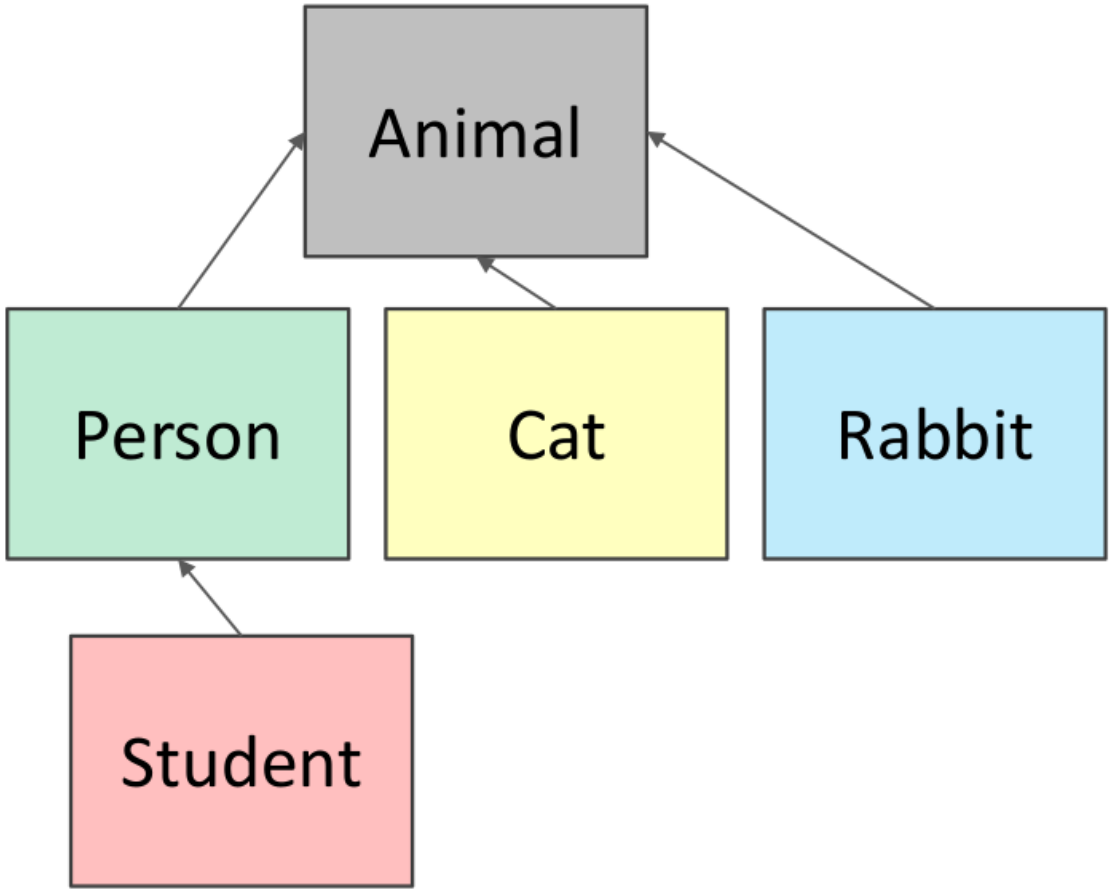
Student

Cat

Rabbit

HIERARCHIES

- **parent class** (superclass)
- **child class** (subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior



INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc

INHERITANCE: SUBCLASS

inherits all attributes of `Animal`:
`__init__()`
`age, name`
`get_age(), get_name()`
`set_age(), set_name()`
`__str__()`

```
class Cat(Animal):
    def speak(self):
        print("meow")
    def __str__(self):
        return "cat:"+str(self.name)+":"+str(self.age)
```

add new functionality via speak method

overrides `__str__`

- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with `Cat`'s new method
- `__init__` is not missing, uses the `Animal` version

WHICH METHOD TO USE?

- subclass can have methods with same name as superclass
- for an instance of a class, look for a method name in current class definition
- if not found, look for method name up the hierarchy (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []
    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)
    def speak(self):
        print("hello")
    def age_diff(self, other):
        diff = self.age - other.age
        print(abs(diff), "year difference")
    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal constructor

call Animal's method

add a new data attribute

new methods

override Animal's __str__ method

```
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major
    def change_major(self, major):
        self.major = major
    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")
    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` class

inherits `Person` and `Animal` attributes

adds new data

- I looked up how to use the `random` class in the python docs
- `random()` method gives back float in [0, 1)

CLASS VARIABLES AND THE Rabbit SUBCLASS

- **class variables** and their values are shared between all instances of a class

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
```

parent class

class variable

instance variable

access class variable

incrementing class variable changes it for all instances that may reference it

- `tag` used to give **unique id** to each new rabbit instance

Rabbit GETTER METHODS

```
class Rabbit(Animal):
    tag = 1
    def __init__(self, age, parent1=None, parent2=None):
        Animal.__init__(self, age)
        self.parent1 = parent1
        self.parent2 = parent2
        self.rid = Rabbit.tag
        Rabbit.tag += 1
    def get_rid(self):
        return str(self.rid).zfill(3)
    def get_parent1(self):
        return self.parent1
    def get_parent2(self):
        return self.parent2
```

method on a string to pad the beginning with zeros for example, 001 not 1

- getter methods specific for a `Rabbit` class
- there are also getters `get_name` and `get_age` inherited from `Animal`

WORKING WITH YOUR OWN TYPES

```
def __add__(self, other):
        # returning object of same type as this class
        return Rabbit(0, self, other)
```

recall Rabbit's `__init__(self, age, parent1=None, parent2=None)`

- define **+ operator** between two `Rabbit` instances
  - define what something like this does: `r4 = r1 + r2` where `r1` and `r2` are `Rabbit` instances
  - `r4` is a new `Rabbit` instance with age 0
  - `r4` has `self` as one parent and `other` as the other parent
  - in `__init__`, **`parent1` and `parent2` are of type `Rabbit`**

SPECIAL METHOD TO COMPARE TWO Rabbits

- decide that two rabbits are equal if they have the **same two parents**

booleans

```
def __eq__(self, other):
    parents_same = self.parent1.rid == other.parent1.rid \
                   and self.parent2.rid == other.parent2.rid
    parents_opposite = self.parent2.rid == other.parent1.rid \
                       and self.parent1.rid == other.parent2.rid
    return parents_same or parents_opposite
```

- compare ids of parents since **ids are unique** (due to class var)
- note you can't compare objects directly
  - for ex. with `self.parent1 == other.parent1`
  - this calls the `__eq__` method over and over until call it on `None` and gives an `AttributeError` when it tries to do `None.parent1`

Multiple Inheritance and MRO

Python supports multiple inheritance.

```
class A:
    def hello(self):
        print("A")

class B:
    def hello(self):
        print("B")

class C(A, B):
    pass

c = C()
c.hello()
```

Question: Which method is called?

Python resolves this using **MRO** (Method Resolution Order).

```
print(C.__mro__)
```

Python searches classes in a specific order.

```
(<class '__main__.C'>,
 <class '__main__.A'>,
 <class '__main__.B'>,
 <class 'object'>)
```

Multiple Inheritance and MRO

Python supports multiple inheritance.

```
class A:
    def hello(self):
        print("A")

class B:
    def hello(self):
        print("B")

class C(A, B):
    pass

c = C()
c.hello()
```

Question: Which method is called?

Python resolves this using **MRO** (Method Resolution Order).

```
print(C.__mro__)
```

Python searches classes in a specific order.

```
(<class '__main__.C'>,
 <class '__main__.A'>,
 <class '__main__.B'>,
 <class 'object'>)
```

$C \rightarrow A \rightarrow B \rightarrow object$

OBJECT ORIENTED PROGRAMMING

- Create your own collections of data
- Organize information
- Division of work
- Access information in a consistent manner;
- Adds layers of complexity;
- Like functions, classes are a mechanism for decomposition and abstraction in programming

Thank You for Your Attention!

You are not just writing programs—you are designing structured systems of interacting objects.

Lecture 8-1: Python Modules

Python Modules

- A **module** is a file containing Python code.
- It can define functions, classes, and variables, and may also contain executable code.
- It helps organize Python code into reusable units, making large projects easier to manage and maintain.

Types of Python Modules

1. **Built-in Modules:** Included in Python's standard library. Always available with the standard distribution.
   - *Examples:* `math`, `random`, `os`, `sys`.
2. **Third-party Modules:** Created by external developers and the community. Installed using package managers like `pip`.
   - *Examples:* `requests` (HTTP), `pandas` (data manipulation), `matplotlib` (visualization).
3. **Custom Modules:** Created by developers by organizing application-specific code into separate `.py` files for internal reusability.

Package Managers

A system tool responsible for installing, updating, configuring, and removing software packages on a computer's operating system or runtime environment.

System Package Manager Example (Ubuntu / Apt):

```
# Install SSH
sudo apt install ssh
# Remove OpenSSH Server
sudo apt remove openssh-server
# List installed packages containing 'ssh'
dpkg -l | grep ssh
```

Using Third-Party Modules via pip

Python uses **`pip`** as its dedicated package manager to download and manage external libraries and frameworks.

Common pip Commands:

```
# Install a package
pip install pygame

# List all installed python packages
pip list

# Uninstall a package
pip uninstall pygame
```

Importing Third-Party Modules

Once installed, third-party modules can be imported using different syntax structures depending on your project requirements.

Method 1: Importing the entire module

```
import requests

response = requests.get("https://www.google.com")
print(response.status_code)  # Output: 200
```

Method 2: Importing specific attributes/functions

```
from requests import get

response = get("https://www.google.com")
print(response.status_code)  # Output: 200
```

Handling ModuleNotFoundError

If you attempt to import a library that has not been installed yet, Python will raise a `ModuleNotFoundError`.

```
import pygame
% ModuleNotFoundError: No module named 'pygame'
```

The Fix: Verify status using `pip list` and install the missing module:

```
# Check if installed
pip list | grep pygame

# Install the module
pip install pygame

# Verify installation (Output example: 2.6.1)
pip list | grep pygame
```

Pygame Implementation Example

```
import pygame

pygame.init()
width = 400
height = 100
screen = pygame.display.set_mode((width, height))
font = pygame.font.SysFont(None, 48)
text = font.render("Welcome to pygame", True, (255, 255, 255))

run = True
while run:
    screen.fill((0, 0, 0))
    % This line is very long, but breaklines=true will safely wrap it visually:
    screen.blit(text, ((width - text.get_width()) // 2, (height - text.get_height()) // 2))
    pygame.display.flip()

    for event in pygame.event.get():
        % This long line will also wrap naturally at the 'or' statements:
        if event.type == pygame.QUIT or event.type == pygame.MOUSEBUTTONUP or event.type ==
             pygame.KEYUP:
            run = False
```

Creating Your Own Module

You can easily split your project across multiple files by writing your own modules.

mymodule.py

```
# Module definitions
var = 5
lst = [1, 2, 3]

def func(var):
    print(var)
```

main.py

```
# Executable script
import mymodule
from mymodule import lst

# using definitions
# in mymodule
print(mymodule.var)
print(lst)
mymodule.func(6)
```

Common Python Modules: Examples

- OS interacting module
- Data science modules

OS Interaction from Python

The `os` Module provides a powerful way to interact directly with your underlying operating system. It is designed to be **cross-platform**, meaning the same code can run on Windows, macOS, and Linux.

Platform-Specific Limitations:

- Certain behaviors (like file permissions or system paths) handle security and storage differently between Windows and Unix-based systems.
- Always test your code on the specific target operating system.

Checking Process IDs:

```
import os
print(os.getpid())   # Returns the current process ID
```

Directory Management with `os`

You can easily check your path, look inside folders, create, and destroy directories.

```
import os

# 1. Where am I now? (Get Current Working Directory)
print(os.getcwd())

# 2. List items inside a specific directory
print(os.listdir('/home/p4'))  # Equivalent to running 'ls'

# 3. Create a new directory
os.mkdir("/home/p4/dir_directory")

# 4. Remove/Delete an empty directory
os.rmdir("/home/p4/dir_directory")
```

Executing Shell Commands via `os.system()`

The `os.system()` function allows Python to pass raw string commands straight to the underlying operating system terminal/shell.

```
import os

# Execute terminal commands directly
os.system("pwd")                # Prints working directory to
↪  console
os.system("ls /home/p4")        # Lists files via system shell
```

When running `os.system()`, it returns an exit code integer:

- **0** usually means the command executed successfully without any errors.
- Any non-zero value indicates a system error code or warning.

Modules for Data Science

Python's dominance in Data Science is driven by an interconnected ecosystem of third-party libraries. These tools handle everything from raw math to advanced prediction models.

The Core Data Science Stack

- **NumPy:** Array manipulation and vector operations.
- **Pandas:** Structured data manipulation and analysis.
- **Matplotlib & Seaborn:** Data visualization and static plots.
- **Scikit-Learn:** Foundations of Machine Learning.

Data Processing: NumPy and Pandas

Before analyzing data, it must be stored and structured efficiently.

1. NumPy Arrays (Fast Math operations):

```
import numpy as np
matrix = np.array([[1, 2], [3, 4]])
print(matrix * 2)  # Multiplies every element efficiently
```

2. Pandas DataFrames (Excel-like tables):

```
import pandas as pd
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print(df.head())  # Prints a formatted table view
```

Visualization and Machine Learning

Once structured, data is visualized for insights and processed for pattern prediction.

3. Matplotlib (Plotting graphs):

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [10, 20, 30])
plt.show()  # Renders a line chart window
```

4. Scikit-Learn (Predictive Modeling):

```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
# model.fit(X, y) used to train predictive AI equations
```

Lecture 8-2: Python Environment manager

Python Environment Management

Python uses two main utilities to isolate project environments and prevent conflicts

1. **Command Line Tools:** Focused purely on lightweight shell operations.
   - *Example:* `virtualenv`
2. **Command Line & GUI Combined Tools:** Feature graphical dashboards alongside a standard shell.
   - *Examples:* `conda` (command line) and **Anaconda Navigator** (Graphical User Interface).

Env Management with Python CLI Tool

```
# 1. Installing virtualenv
sudo apt-get install virtualenv  # Using system apt
pip install virtualenv           # Using pip manager

# 2. Creating a Workspace
cd /path/to/your/project
virtualenv myenv                 # Creates an isolated sandbox folder

#3. Activating \& Deactivating
source myenv/bin/activate        # Turn on environment (Linux/macOS)
deactivate                       # Turn off / exit environment workspace
```

Environment Management with Conda

`conda` provides comprehensive environment control tools using straightforward terminal syntax structures.

```
# 1. Inspect all active and inactive environments
conda env list

# 2. Spin up a new sandbox (e.g., Gaming with Python 3.6)
conda create --name Gaming python=3.6

# 3. Enter the environment workspace (Activate)
conda activate Gaming

# 4. Exit back to the base system environment layer (Deactivate)
conda deactivate

# 5. Completely purge/delete a targeted environment
conda env remove --name Gaming
```

Package management: Pip vs. Conda

You can use either `pip` or `conda` to handle workspace operations, but they operate under fundamentally different scopes.

- **Conda:** A cross-language environment and package manager designed to track complex system dependencies across multiple computing applications.
- **Pip:** A language-specific management ecosystem built exclusively to distribute Python packages and libraries.

Install, list, search, uninstall packages

Pip Command	Conda Command
`pip install pygame`	`conda install pygame`
`pip list`	`conda list`
`pip list | grep pygame`	`conda list | grep pygame`
`pip uninstall pygame`	`conda uninstall pygame`

Project Reproducibility: requirements.txt

Sharing code is not enough; you must also share the list of libraries your code relies on to prevent version conflicts on other machines.

The Solution: A `requirements.txt` file, which acts as a manifest listing all external dependencies and their exact versions.

1. Generate the Manifest

Run this inside your active environment to export your current package list:

```
# Saves packages & versions
pip freeze > requirements.txt
```

2. Recreate the Environment

A teammate can install every required package in one go:

```
# Installs listed packages
pip install -r requirements.txt
```

Example output in file:

```
pygame==2.6.1
requests==2.31.0
numpy==1.26.4
```

Project Reproducibility: environment.yml

For Conda, sharing a project goes beyond Python packages: captures Python versions, system dependencies, and external libraries(complete blueprint of your Conda environment)

1. Export the Environment

Run this from your active conda environment to export its layout:

```
# Saves channels & packages
conda env export > environment.yml
```

2. Recreate the Environment

A teammate can build the exact same environment in one command:

```
# Installs listed dependencies
conda env create -f environment.yml
```

Example output in file:

```
name: Gaming
channels: - defaults
dependencies: - python=3.6 - pygame=2.6.1
```

Always include an `environment.yml` file in your project's root folder when sharing Conda projects.

Thank You for Your Attention!

You are now equipped to build isolated, conflict-free environments
and manage packages confidently for any Python project.

Lecture 9: File handling in Python

Overview of File Handling & Iterables

- **File handling**
  - Creating and managing file objects
    - File handling methods
    - Understanding File Paths in Python
    - File modes
- **Iterable objects in Python**
- **Managing Files with the os Module**

So Far: What Is an Object in Python?

- In Python, everything is treated as an **object**

What can we do with objects?

1. **Create** new objects
   - Example: `x = 5`, `L = [1, 2, 3]`
2. **Use and manipulate** objects
   - Perform operations and call functions/methods
   - Example: `len(L)`, `x + 2`
3. **Objects are removed automatically**
   - Python frees memory when objects are no longer needed
   - This process is called **garbage collection**

File Objects in Python

- A file in Python is also represented as an **object**
- This follows the same object lifecycle we have already seen
- Lifecycle of a File Object

1. **Create / Open**
   - Use `open()` to create a file object
   - This establishes a connection between Python and the file on disk: `f = open("data.txt", "w")`
2. **Use / Process**
   - Read from or write to the file: `f.write("Hello")`
3. **Close / Clean Up**
   - Use `close()` to release the file connection: `f.close()`
   - Later, Python automatically removes the file object from memory

Python Provides a Safe Structure for File Handling

- Python offers a cleaner way to manage file objects using `with`
- It automatically handles opening and closing the file

Open file $\rightarrow$ **Process file** $\rightarrow$ **Automatic cleanup**

Recommended File Handling Structure

```
with open("data.txt", "w") as f:
    f.write("Hello")
```

`with` makes resource management safer and simpler.

Common File Handling Methods in Python

- Python provides several methods to read data from a file
- The choice depends on how much content you want to process

Main Reading Methods

- `read()` – reads the entire file as one string
- `readline()` – reads the next line
- `readlines()` – reads all lines into a list
- `for line in file` – recommended for large files

Other Useful Methods

- `write()` – writes text to a file
- `writelines()` – writes multiple lines

Python offers flexible tools for reading and writing files.

Understanding File Paths in Python

- A **file path** tells Python where to find a file
- Choosing the correct path is essential when opening files

Relative Path

- Starts from the current working directory
- Shorter and commonly used in projects

```
with open("data/file.txt")
```

Absolute Path

- Full path from the root folder
- Works regardless of current location

```
/home/user/project/
data/file.txt
```

Important

- Python looks for relative paths from the current working directory
- Wrong paths are a common source of errors

Cross-platform issue

- Windows uses `\`, Linux/macOS use `/`
- Paths may break when code is moved between systems

Solution:

- Use `os.path` or `pathlib` for portable paths

Solution

- Use `os.path` or `pathlib` for portable paths

Example using `os.path`:

```
import os
path = os.path.join("data", "file.txt")
with open(path, "r") as f:
    print(f.read())
```

Example using `pathlib` (recommended):

```
from pathlib import Path
path = Path("data") / "file.txt"
with open(path, "r") as f:
    print(f.read())
```

Iterable Objects in Python

- An **iterable** is any object that can return its elements one by one
- Iterable objects can be used in a `for` loop
- Most Python iterables work with built-in functions like `map()`, `filter()`, and `sorted()`.

Common Iterable Objects

- **List**: `[1, 2, 3]`
- **Tuple**: `(1, 2, 3)`
- **String**: `"hello"`
- **Dictionary**: iterates over keys by default
- **File object**: iterates line by line

Iterables make repetition simple and natural in Python.

File Modes in Python

- When opening a file, we must choose how we want to use it
- The file mode tells Python what kind of access is needed

Common File Modes

- `"r"` – read an existing file
- `"w"` – write (creates or overwrites file)
- `"a"` – append to the end of a file
- `"rb"` – read a file in binary mode

Example

```
with open("data.txt", "a") as f:
    f.write("New line")
```

Choosing the correct file mode is essential.

Using Iterable Objects

Looping with `for`

```
my_list = [1, 2, 3]

for item in my_list:
    print(item)
```

Using built-in functions

```
my_list = [1, 2, 3]

squares = map(
    lambda x: x**2,
    my_list
)
```

Reading a file

```
with open("file.txt") as f:
    for line in f:
        print(line)
```

- `map()`
- `filter()`
- `sorted()`

Python uses iterables everywhere.

Managing Files with the `os` Module

- The `os` module allows Python to interact with the operating system
- It can be used to manage files and directories

Common File Operations

- Rename files
- Remove files
- Create directories
- Run system commands

Examples:

```
import os
os.rename("example.txt", "example2.txt")
os.system("ls -l")
os.system("mv example2.txt example.txt")
```

The `os` module connects Python with your system.

Summary

- A file in Python is an **object**
  - It has its own data, methods, and lifecycle
- A file object is also an **iterable**
  - Python can process files line by line
- File handling follows a clear structure
  - Open $\rightarrow$ Process $\rightarrow$ Close
  - `with open(...)` makes this safer
- Python also provides tools for file management
  - Example: `os` module

A file is not special — it follows the same object principles as everything else in Python.

Thank You for Your Attention!

Computer science may seem vast and complex,

but understanding patterns and connections makes it simpler.

A file is just another object —

once you see the similarities,

complex ideas become easier to understand.

Lecture 9: Testing, Debugging, Exceptions, Assertions

Lecture Roadmap

1. Why programs fail
2. Defensive programming
3. Testing and validation
4. Debugging strategies
5. Exceptions and exception handling
6. Assertions as defensive programming

Why Do Programs Fail?

- Programs may fail because of syntax errors, runtime errors, or logical errors
- Reliable programs are designed to prevent, detect, and handle errors
- Testing, debugging, exceptions, and assertions help us improve program quality

Good programming is not only writing code, but making code reliable.

Types of Errors in Python

- **Syntax errors**: prevent the program from running

```
if x > 5
    print(x)
```

- **Runtime errors**: occur while the program is running

```
x = 10 / 0
```

- **Logical errors**: program runs, but produces incorrect results

```
def square(x):
    return x * 2    # wrong logic
```

Syntax stops execution, runtime crashes during execution, logical errors silently produce wrong results.

Three Related Activities in Software Reliability

From Prevention to Correction

Defensive Programming

- Write specifications
- Modularize code
- Check assumptions

Testing

- Compare results to specifications
- Ask: “How can I break it?”
- Validate behavior

Debugging

- Study what led to an error
- Ask: “Why is it wrong?”
- Fix the cause

Defensive Programming

- Design code with testing and debugging in mind
- Break programs into smaller modules
- Document assumptions and constraints
- Check expected inputs and outputs

Defensive programming tries to prevent bugs before they happen.

When Are We Ready to Test?

- The program must run first
  - Remove syntax errors
  - Remove obvious semantic errors
- Prepare expected results
  - Choose test inputs
  - Define expected outputs

Testing requires knowing what the correct answer should be.

Classes of Tests

Unit Testing

- Test each function separately
- Validate small parts

Regression Testing

- Add tests after fixing bugs
- Prevent old bugs from returning

Integration Testing

- Test the whole system
- Check component interaction

Testing Approaches

Black-box Testing

- Based on specification
- Does not inspect code
- Tests input-output behavior

Glass-box Testing

- Based on code structure
- Tests branches and paths
- Considers loops and conditions

Black-box asks what the code should do; glass-box asks how the code works.

Black-box Testing Example

Specification-based testing

```
def sqrt(x, eps):
    """Assumes x >= 0 and eps > 0
    Returns res such that
    x-eps <= res*res <= x+eps"""
```

- Boundary values
- Typical values
- Small and large values
- Special cases such as perfect squares

Glass-box Testing Example

```
def abs_value(x):
    if x < -1:
        return -x
    else:
        return x
```

- Testing `2` and `-2` covers both branches
- But `abs_value(-1)` still gives wrong result
- Boundary cases still matter

Testing: What Comes Next?

- Testing helps detect errors in a program
- However, it does not explain why those errors occur

What should we do after detecting an error?

This leads us to the next step: Debugging

Debugging

- Debugging means finding why a program behaves incorrectly
- Tools can help, but systematic thinking is essential
- Useful tools:
  - print statements
  - debugger
  - Python Tutor
  - careful reasoning

Debugging as a Scientific Process

- Study the available evidence
- Form a hypothesis
- Run a repeatable experiment
- Use the simplest input that exposes the bug

Do not only ask "What is wrong?" Ask "How did this result happen?"

Good Debugging Practice

Avoid

- Writing the whole program first
- Testing only at the end
- Changing many things at once

Prefer

- Write one function
- Test it
- Debug it
- Add regression tests

Are Programs Ever Perfect?

After testing...

After debugging...

Are programs really perfect?

Users behave unpredictably.
(Users may provide unexpected input.)

Systems fail.

What happens next?

From Debugging to Error Handling

- Debugging helps us find and fix errors during development
- However, not all errors can be predicted or eliminated
- Some errors occur only during program execution

How can a program handle unexpected errors at runtime?

This leads to the concept of exception handling

Handling Runtime Errors in Python

- Runtime errors occur during program execution
- They may interrupt or terminate the program
- Python provides structured handling using `try/except`
- This allows programs to respond to errors gracefully

Robust programs do not avoid errors—they handle them effectively.

Exception Handling Structure

- **try**: write code that may cause an error
- **except**: handle the error if it occurs
- **else**: execute if no error occurs
- **finally**: always execute, often for cleanup

$$\textbf{try} \rightarrow \textbf{except} \rightarrow \textbf{else} \rightarrow \textbf{finally}$$

Handling Specific Exceptions

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    print(a / b)
except ValueError:
    print("Invalid number")
except ZeroDivisionError:
    print("Cannot divide by zero")
except:
    print("Unexpected error")
```

Handle specific exceptions whenever possible.

Example: Realistic example

```
try:
    f = open("grades.txt", "r")
    line = f.readline()
    grade = int(line)
    average = 100 / grade
except FileNotFoundError:
    print("Error: file not found")
except ValueError:
    print("Error: invalid data")
except ZeroDivisionError:
    print("Error: grade cannot be zero")
else:
    print("Computation successful")
    print("Average:", average)
finally:
    print("Closing file")
    f.close()
```

Possible Outputs

Case 1: File contains `25`

```
Computation successful
Average: 4.0
Closing file
```

Case 2: File contains `0`

```
Error: grade cannot be zero
Closing file
```

Case 3: File contains invalid text

```
Error: invalid data
Closing file
```

Beyond Handling Exceptions

So far, we handled exceptions raised by Python.

But what if our program detects an error?

What if a function cannot produce a valid result?

Can we create our own error conditions?

Raising Exceptions with `raise`

- `raise` triggers an exception manually
- It is used when the program detects an invalid situation
- It clearly signals that normal execution cannot continue

```
age = -5

if age < 0:
    raise ValueError("Age cannot be negative")
```

Built-in Exception Classes in Python

- Python provides many built-in exception classes
- Each represents a specific type of runtime error

Examples:

- `ValueError` – invalid value
- `TypeError` – wrong type
- `ZeroDivisionError` – division by zero
- `IndexError` – invalid index

These exceptions help us identify and handle different kinds of errors.

A New Question

Python provides many built-in exceptions.

But are they enough for every situation?

What if our program has its own rules?

Can we define our own exceptions?

Custom Exception Classes

- Custom exceptions represent application-specific errors
- They make error handling more meaningful and precis
- They usually inherit from `Exception`

```
class NegativeAgeError(Exception):
    pass

age = -5

if age < 0:
    raise NegativeAgeError("Invalid age")
```

Built-in vs Custom Exceptions

Built-in Exceptions

- Predefined in Python
- Handle common runtime errors
- Examples: `ValueError`, `TypeError`

Custom Exceptions

- Defined by the programmer
- Represent application-specific errors
- Inherit from `Exception`

```
try:
    x = int("abc")
except ValueError:
    print("Invalid value")
```

```
class NegativeAgeError(
    Exception):
    pass

if age < 0:
    raise NegativeAgeError(
        "Age cannot be
            negative")
```

From Handling Errors to Preventing Bugs

- Exception handling allows programs to respond to runtime errors
- However, some errors indicate problems in our program logic

How can we check that our assumptions are always true?

This leads to the concept of assertions

Assertions vs Exceptions

- **Exceptions** handle unexpected situations at runtime
- **Assertions** check assumptions during development

Exceptions $\rightarrow$ handle errors
Assertions $\rightarrow$ detect bugs

Assertions

- Assertions check whether assumptions are true
- If the condition is false, Python raises `AssertionError`
- Assertions support defensive programming
- They help detect bugs early

Assertion Example

```
def avg(grades):
    assert len(grades) != 0, "no grades data"
    return sum(grades) / len(grades)

print(avg([80, 90, 100]))    # valid input
print(avg([]))               # invalid input
```

Output:

```
90.0
Traceback (most recent call last):
  ...
AssertionError: no grades data
```

- If the condition fails, an `AssertionError` is raised
- Helps detect incorrect usage of the function early

Where to Use Assertions

- Check argument types
- Check constraints on values
- Check data structure invariants
- Check return values
- Detect violations of function assumptions

Use assertions as a supplement to testing, not as a replacement.

Use assertions for internal correctness, not user input validation.

Summary

- Defensive programming helps prevent bugs
- Testing checks whether code matches its specification
- Debugging explains why code behaves incorrectly
- Exceptions handle unexpected runtime conditions
- Raising exceptions signals invalid situations explicitly
- Assertions check assumptions during execution

Reliable programs are designed, tested, and debugged systematically.

Thank You for Your Attention!

Errors $\rightarrow$ Prevention $\rightarrow$ Testing $\rightarrow$ Debugging $\rightarrow$ Exceptions $\rightarrow$ Assertions

Building reliable programs is a systematic process.

Lecture 11-1: Pythonic idioms

Pythonic Idioms

What are Pythonic Idioms?

- Common and preferred ways of writing Python code
- Emphasize **readability**, **simplicity**, and **clarity**
- Reflect Python's design philosophy

Why use them?

- Write cleaner and shorter code
- Reduce complexity and errors
- Improve maintainability

Today's Focus

- Truthy / Falsy conditions
- Membership operator (`in`)
- Chained comparisons
- `while-else` control flow
- List comprehensions
- Lambda functions
- `zip()` and unpacking

Pythonic Idiom: Truthy / Falsy

Concept

- Every value in Python can be converted to a boolean using `bool()`
- Values are implicitly interpreted as **True** or **False**
- Explicit comparisons are often unnecessary

Non-Pythonic

```
if len(my_list) > 0:
    print("Not empty")
```

Pythonic

```
if my_list:
    print("Not empty")
```

Common falsy values:

- `None, False, 0, "", [], {}`

Pythonic Idiom: while–else

Concept

- `else` runs if the loop completes normally
- Not executed if loop exits via `break`

```
x = 5

while x > 0:
    print(x)
    x -= 1
else:
    print("Loop finished")
```

Key Idea

- `else` = no interruption (no break)

Pythonic Idiom: List Comprehension

Concept

- Compact way to create lists
- Combines loop + condition + expression

Non-Pythonic

```
squares = []
for x in range(5):
    squares.append(x**2)
```

Pythonic

```
squares = [x**2 for x in range(5)]
```

Pythonic Idiom: zip() Function

Concept

- Combines multiple iterables into a single iterator
- Pairs elements based on their position

Example

```
names = ["Alice", "Bob", "Charlie"]
scores = [85, 90, 78]

for name, score in zip(names, scores):
    print(name, score)
```

Output

```
Alice 85
Bob 90
Charlie 78
```

Pythonic Idiom: Swapping Variables

Concept

- Swap values without using a temporary variable
- Uses **tuple unpacking**

Non-Pythonic

```
temp = a
a = b
b = temp
```

Pythonic

```
a, b = b, a
```

Key Idea

- Right side creates a tuple
- Left side unpacks values into variables

Pythonic Idiom: Unpacking (Extracting Values)

Concept

- Extract values from a collection into variables
- Works with tuples, lists, and other iterables

Example

```
point = (3, 5)

x, y = point
print(x, y)
```

Multiple Assignment

```
a, b, c = [1, 2, 3]
```

Key Idea

- Sequence $\rightarrow$ variables (position-based)

Example: Unpacking with Function Return Values

- Functions can return multiple values
- Python automatically packs them into a tuple
- Values can be unpacked into variables

```
def get_user():
    return "Alice", 25

name, age = get_user()
print(name)   # Alice
print(age)    # 25
```

Key Idea

- Return: values $\rightarrow$ tuple
- Assignment: tuple $\rightarrow$ variables

Pythonic Idiom: Membership Operator

Concept

- Check if a value exists in a collection
- Uses `in` / `not in`
- Works with **any iterable**

Examples

```
# List

if 3 in [1,2,3]:

# String
if "a" in "apple":
```

Non-Pythonic

```
numbers = [1,2, 3]
found = False
for n in numbers:
    if n == 3:
        found = True
```

Pythonic

```
if 3 in numbers:
    print("Found")
```

Key Idea

- Iterable membership check
- Cleaner and more readable

Pythonic Idiom: Chained Comparison

Concept

- Combine multiple comparisons in one expression
- Improves readability and conciseness

Non-Pythonic

```
if x > 0 and x < 10:
    print("In range")
```

Pythonic

```
if 0 < x < 10:
    print("In range")
```

Key Idea

- Equivalent to: `0 < x and x < 10`

Pythonic Idiom: Lambda Functions

Concept

- Small anonymous (unnamed) functions
- Used for short, simple operations

Example

```
add = lambda a, b: a + b
print(add(2, 3))
```

Common Usage

- With `map()`, `filter()`, `sorted()`

```
numbers = [1, 2, 3]
result = list(map(lambda x: x*2, numbers))
```

Summary: Thinking Pythonically

Key Principles

- Write code that is **clear, concise, and readable**
- Prefer **built-in language features**
- Avoid unnecessary complexity and repetition

Pythonic Patterns Covered

- Truthy / Falsy conditions
- Membership operator (`in`)
- Chained comparisons
- `while-else` control flow
- List comprehensions
- Lambda functions
- `zip()` and unpacking

Thank You for Your Attention!

Clarity over complexity.

Readability over optimization.

Lecture 11-2: Managing Database with Python

From Python Foundations to Real-World Applications

- Transitioning from core Python concepts to practical use cases
- Applying Python to solve real problems in software development and data-driven tasks

Focus Areas:

1. Web Development
2. Data Analysis

A Database = Organized Data Storage

- A **database** is a structured collection of data.
- It helps us:
  - **store** information
  - **manage** data efficiently
  - **retrieve** data quickly
- Databases are managed by:
  - Users
  - Developers
  - Database Administrators (**DBAs**)

A Real-World Example

Imagine a system like **Canvas** storing student grades:

- Student ID
- Student Name
- Course
- Assignment / Exam
- Score

Question: How can we store and manage this data efficiently for many students?

Database Management with DBMS

- Databases are managed using a **Database Management System (DBMS)**.
- Popular DBMS examples:
  - MySQL
  - PostgreSQL
  - SQLite
- Users interact with the DBMS using:
  - **SQL** (*Structured Query Language*)
  - Used to create, read, update, and delete data

SQL: Communicating with a Database

- **SQL** = **Structured Query Language**
- Used to send commands to a **DBMS**

Example SQL Command:

```
CREATE DATABASE database_name;
```

Users $\rightarrow$ SQL Commands $\rightarrow$ DBMS $\rightarrow$ Database

Basic Database Operations

CRUD represents the four fundamental operations used to manage data in a database.

- **Create** – Add new data
- **Read** – Retrieve data
- **Update** – Modify existing data
- **Delete** – Remove data

Using Python for Database Management

- Python can manage databases using built-in and external modules: SQLite,MySQL Connector, psycopg2 (PostgreSQL)
- **SQLite**:
  - Lightweight and easy to use
  - Requires **no server**
  - Ideal for learning and small projects
- **MySQL / PostgreSQL**:
  - Better for large-scale systems
  - Suitable for web and enterprise applications

SQL Is Largely Universal

- SQL syntax is mostly **universal** across database systems.
- Once you understand **SQL logic**, you can work with:
  - MySQL
  - PostgreSQL
  - SQLite
  - Other DBMS platforms
- Only small implementation details may differ.

Databases vs. CSV / Excel Files

- **Databases are better for:**
  - Efficient storage of large or complex data
  - Maintaining data consistency and reducing duplication
  - Supporting multiple users at the same time
  - Performing powerful queries and managing relationships
- **CSV / Excel files are useful for:**
  - Small and simple datasets
  - Quick manual editing and sharing

Databases and Python

- A **database** stores structured data.
- A **DBMS** manages and organizes that data.
- Users interact with databases using **SQL** and CRUD operations.
- **Python** connects to DBMS tools (such as **SQLite**) to automate database tasks.
- Databases are more **scalable**, **efficient**, and **reliable** than simple files.
- Next, let's learn how Python interacts with a database

Python and SQLite Architecture

Python Program $\rightarrow$ `sqlite3` $\rightarrow$ SQLite Engine $\rightarrow$ Database File

- **Python Program**: writes code and sends commands
- **sqlite3 module**: connects Python to SQLite
- **SQLite Engine**: processes SQL queries
- **Database File**: stores data permanently

A Simple Way to Learn Databases

- **No setup required**
  - No separate server installation needed
- **Uses a local file**
  - The database is stored directly on your computer
- **Easy to learn SQL basics**
  - Practice creating tables, inserting, and querying data
- **Ideal for beginners and small projects**

Basic Steps to Work with SQLite

1. **Connect to a Database**
   - Use `sqlite3.connect()` to open or create a database file
2. **Create a Cursor**
   - Use `cursor()` to create a cursor object
   - The cursor executes SQL commands
3. **Execute SQL Queries**
   - Use `execute()` to create tables or modify data
4. **Commit Changes**
   - Use `commit()` to save changes permanently
5. **Close the Connection**
   - Use `close()` to release resources

Live Coding with SQLite in Python

Today, we will learn how to:

1. Create a database file
2. Connect Python to the database
3. Create a table
4. Insert data into the table
5. Query and display results

From theory to practice: building your first database program

Working with Database Objects in Python

- In Python, a database connection is treated as an **object**.
- `sqlite3.connect()` returns a:
  - **Connection object –** manages the link to the database
- `conn.cursor()` returns a:
  - **Cursor object –** executes SQL commands and fetches results
- These objects provide:
  - Methods (e.g., `execute()`, `commit()`, `close()`)
  - A simple way to manage database operations

Thank You for Your Attention!

Your code can now talk to a database. The next question is: how can we use it in real systems, such as a web application backend?

Lecture 12: Managing API with Python

API and Flask in Python

API (Application Programming Interface)

- A set of protocols, tools, and definitions
- Enables communication between software components
- Allows different systems to interact with each other

Building APIs with Python
Flask

- Lightweight and flexible web framework
- Provides tools and libraries for rapid development
- Minimalistic design and easy to learn

Why Flask for APIs?

- Simple and easy to set up
- Highly flexible and extensible
- Commonly used for RESTful API development

Flask for APIs: Routing and HTTP Methods

Routing and Endpoint Definition

- Define **routes (URL patterns)** mapped to functions
- Functions (called **view functions / endpoints**) handle HTTP requests
- Can perform tasks such as:
  - Processing data
  - Interacting with databases
  - Generating responses

HTTP Methods Support

- Supports standard HTTP methods:
  - **GET**, **POST**, **PUT**, **DELETE**
- Specify allowed methods per route
- Enables structured interaction with the API

Flask Example: Hello World API

```
from flask import Flask

# Create Flask application instance
app = Flask(__name__)

print(__name__)

# Define route (URL -> function)
@app.route('/', methods=['GET'])
def hello_world():
    return "Hello world"

# Run development server
app.run(debug=True, port=5015)
```

Key Points

- `@app.route()` maps URL (http://localhost:5015/) to function (endpoint)
- Handles HTTP **GET** request
- Returns a simple response ("Hello world")
- Runs local server on port **5015** with debug mode

Flask Request Flow: Hello World API

Request Flow

Client
(Browser / Postman)
↓
`GET /`
↓
`@app.route('/')`
↓
`hello_world()`
↓
`"Hello world"`
↓
Response to Client

What Happens?

- Client sends a **GET request**
- Flask receives the request
- Matches URL with **route**
- Calls the corresponding **function**
- Function generates a **response**
- Response is sent back to the client

Key Idea

- URL → Route → Function → Response

Connecting Client and Database via API

From Client to Database

Objective

- Connect a client application to a database through an API
- Enable structured data interaction over HTTP

Example Overview

- Build a simple API using **Flask**
- Use **SQLite** as the database
- Manage user information through API endpoints

Supported Operations (CRUD)

- **Create** – Add new user data
- **Read** – Retrieve existing data
- **Update** – Modify user information
- **Delete** – Remove user records

Flask + SQLite API Example

Goal

- Build an API to manage employee data
- Connect **Flask** with **SQLite**

What This API Does

- Retrieve all employees
- Retrieve a specific employee
- Add a new employee

Architecture

- Client $\rightarrow$ Flask API $\rightarrow$ SQLite Database

Setup: Flask and SQLite

```
import sqlite3
from flask import Flask, jsonify, request

app = Flask(__name__)
```

Explanation

- `sqlite3`: connect to database
- `Flask`: create web application
- `request`: handle incoming data
- `jsonify`: return JSON responses

GET /employees: Retrieve Specific User

```
@app.route('/employees', methods=["GET"])
def get_data():
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute("SELECT * FROM employees")
    rows = cursor.fetchall()

    cursor.close()
    conn.close()

    user_list = [
      {"id": u[0], "name": u[1],
       "dep": u[2], "sal": u[3]}
      for u in rows
    ]
```

GET /employees/<id>: Retrieve Specific User

```
@app.route('/employees/<int:id>', methods=["GET"])
def get_data_id(id):
    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute(
        "SELECT * FROM employees WHERE id = ?",
        (id,)
    )
    row = cursor.fetchone()

    cursor.close()
    conn.close()

    if row is None:
        return "User not found!"
```

POST /employees/<new>: Create Specific User

```
@app.route('/employees/new', methods=["POST"])
def put_data():
    new_user = request.json

    conn = sqlite3.connect('employee.db')
    cursor = conn.cursor()

    cursor.execute(
        'INSERT INTO employees(name, department, salar
         VALUES (?, ?, ?)',
        (new_user["name"],
         new_user["department"],
         new_user["salary"])
    )

    conn.commit()
```

Run the Flask Application

```
if __name__ == '__main__':
    app.run(debug=True, port=5015)
```

Key Points

- Runs local development server
- Debug mode enables auto-reload
- API available at:
  - `http://localhost:5015/employees`

Testing the API

How to Test the API

- Use tools like **cURL** or a web browser
- **cURL**: command-line tool for sending HTTP requests
- Useful for testing API endpoints and responses

Examples (cURL)
Create a New User (POST)

```
curl -X POST http://127.0.0.1:5015/employees/new \
-H "Content-Type: application/json" \
-d '{"name":"Alice","department":"sales","salary":300}
```

Get All Users (GET)

```
curl -X GET http://127.0.0.1:5015/employees
```

Testing the API (Continued)

Update a User (PUT)

```
curl -X PUT http://127.0.0.1:5015/employees/1 \
-H "Content-Type: application/json" \
-d '{"name":"Alice Smith","salary":310}'
```

Delete a User (DELETE)

```
curl -X DELETE http://127.0.0.1:5015/employees/1
```

Key Idea

- Different HTTP methods $\rightarrow$ different operations (CRUD)
- API can be tested without a graphical interface

Client–Server Interaction (Flask + SQLite)

Client	Flask API	Database
Browser / cURL	Routes + Logic	SQLite
GET /employees	Process Request	SELECT query
POST /employees	Validate Data	INSERT query
Receives Response	Format JSON	Return rows

Flow

- Client sends HTTP request
- Flask processes request and executes SQL
- Database returns data
- Flask returns response to client

Thank You for Your Attention!

From simple functions to structured classes,

your programs learned to organize and communicate within.

Now, through APIs, your code can reach beyond—

interacting with the outside world.

Lecture 13-1: Using Python in Data Science

What is Data Analysis?

Data analysis is a **systematic process** of examining raw data to discover **useful patterns, trends, and insights**.

Its goal is to transform **raw or unstructured data** into **meaningful information** that supports:

- informed decision-making,
- problem solving,
- identifying trends and anomalies,
- better understanding of real-world phenomena.

In short:

$$\text{Raw Data} \rightarrow \text{Analysis} \rightarrow \text{Insights}$$

Main Steps in Data Analysis

Data analysis follows a series of steps to transform raw data into useful insights:

1. **Defining the Objective** Identify the problem or question to solve.
2. **Data Collection** Gather relevant data from different sources.
3. **Data Cleaning and Preparation** Handle missing values, errors, and prepare data for analysis.
4. **Data Exploration** Inspect the structure and basic characteristics of the data.
5. **Exploratory Data Analysis (EDA)** Use descriptive statistics and visualization to identify patterns and trends.
6. **Advanced analysis may also include:**
   - machine learning
   - predictive modeling

Pandas

It is one of the most widely used and powerful Python libraries for:

- data analysis, data manipulation, data cleaning, working with structured datasets.

Main Data Structures in:

- **Series** – a one-dimensional labeled array
- **DataFrame** – a two-dimensional labeled table

These structures make working with data much easier and can be created from:

- in-memory Python data structures (lists, dictionaries),
- external files such as CSV, excel files,
- databases and spreadsheets.

Key Features of Pandas

Pandas provides powerful tools for working with structured data efficiently.

- **Importing Data:** Load data from CSV files, Excel files, databases, and more.
- **Cleaning Data:** Handle missing values, duplicates, and inconsistent formats.
- **Exploring Data:** Inspect structure, summary statistics, and patterns.
- **Preprocessing Data:** Transform, filter, and prepare data for analysis.
- **Efficient Operations on Tabular Data:** Perform calculations, grouping, sorting, and filtering easily.

Benefits of Pandas

Pandas makes working with data easier, faster, and more efficient.

- **Simplifies Complex Data Operations:** Perform tasks such as filtering, sorting, grouping, and merging with simple syntax.
- **Optimized for Large Datasets:** Efficiently handles large volumes of structured data.
- **High-Performance Built-in Functions:** Provides fast tools for:
  - data manipulation,
  - aggregation,
  - filtering,
  - statistical analysis.

Pandas Dataframe data structure

A **DataFrame** can be created from Python's built-in data structures, such as dictionaries and lists.

```
import pandas as pd

data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [21, 20, 22]
}

df = pd.DataFrame(data)

print(df)
print(type(data))
print(type(df))
print(isinstance(df, pd.DataFrame))
```

Accessing Data Elements: DataFrame vs Dictionary

Python Dictionary

```
print(data)

print(data['Name'])
print(data['Age'])
```

Output:

```
{
 'Name': ['Alice', 'Bob', 'Charlie'],
 'Age': [21, 20, 22]
}

['Alice', 'Bob', 'Charlie']
[21, 20, 22]
```

Pandas DataFrame

```
print(df)

print(df['Name'])
print(df['Age'])
```

Output:

```
      Name  Age
0    Alice   21
1      Bob   20
2  Charlie   22

0      Alice
1        Bob
2    Charlie

0    21
1    20
2    22
```

Data Manipulation: Dictionary vs DataFrame

Python Dictionary

```
sum = 0
for age in data['Age']:
    sum += age

mean_age = sum / len(data['Age'])
print(mean_age)

i = 0
for name in data['Name']:
    data['Name'][i] = name.upper()
    i += 1

print(data)
```

Output:

```
21.0

{
 'Name': ['ALICE', 'BOB', 'CHARLIE'],
 'Age': [21, 20, 22]
```

Pandas DataFrame

```
print(df['Age'].mean())

df['Name'] = df['Name'].str.upper()

print(df)
```

Output:

```
21.0

      Name  Age
0    ALICE   21
1      BOB   20
2  CHARLIE   22
```

Data Filtering: Dictionary vs DataFrame

Python Dictionary

```
for age in data['Age']:
    if age >= 21:
        print(age)
```

Output:

```
21
22
```

Pandas DataFrame

```
print(df[df['Age'] >= 21])
```

Output:

```
      Name  Age
0    ALICE   21
2  CHARLIE   22
```

Key Takeaway

For tabular data, a **DataFrame** is much more efficient and convenient than manually working with dictionaries and loops.

Pandas provides powerful built-in tools for:

- filtering and selecting data,
- computing statistics,
- transforming columns,
- handling large datasets efficiently.

Computing Basic Statistics with Pandas

Example: Create a DataFrame from numerical data

```
import pandas as pd

lst = [10, 2, 3, 4, 5, 7, 9, 1, 1]

df_lst = pd.DataFrame(lst)
print(df_lst)
```

Compute basic statistics:

```
print(df_lst.mean())
print(df_lst.mode())
print(df_lst.median())
print(df_lst.std())
print(df_lst.describe())
```

Pandas makes statistical analysis easy with built-in functions.

Thank You for Your Attention!

Master the core concepts and techniques on small examples first,
Then apply them confidently to larger datasets and real-world problems.

Lecture 13-2: Implementing simple Machine learning projects with Python

What is Scikit-Learn (`sklearn`)?

Overview & Architecture

- **Open-Source Foundation:** A comprehensive and robust library built for machine learning in Python.
- **Unified Ecosystem:** Developed seamlessly on top of:
  - **NumPy** for high-performance numerical computing.
  - **Matplotlib** for data visualization.
- **Integration:** Integrates perfectly with data tools like **Pandas**.

Key Capabilities

- Provides robust, built-in tools for both **implementing** and **evaluating** ML models.

Machine learning - short intro

In machine learning, the main goal is to:

1. **Identify patterns** hidden in raw data.
2. **Utilize patterns** to make accurate predictions.

Machine Learning Methods:

- Supervised learning
- Unsupervised Learning

Both methods serve unique, complementary roles depending on data availability and ultimate goals.

Supervised Learning (Guided Pattern Discovery)

What is Supervised Learning?

- Operates on **labeled data** (input features with corresponding correct output labels).
- Learns patterns by mapping inputs to target outputs with direct guidance.

Two Primary Problem Types

- **Regression:** Predicting continuous quantities.
  - *Example:* House Size $\rightarrow$ House Price.
- **Classification:** Predicting categorical labels.
  - *Example:* Email Features $\rightarrow$ Spam (1) vs. Normal (0).

Core Concept Like learning with a teacher: every practice problem comes with an answer key.

Unsupervised Learning (Exploration & Discovery)

What is Unsupervised Learning?

- Operates on **unlabeled data** (no explicit output labels provided).
- Aims to reveal inherent structures, relationships, or hidden patterns.

Core Concept Like exploring a new city: you observe and group things by similarity without a map.

Key Tasks & Techniques

- **Clustering:** Grouping data points based on feature similarity.
- **Dimensionality Reduction:** Reducing the number of input features while preserving critical information.
- **Anomaly Detection:** Spotting outliers or unusual points (e.g., using *Euclidean distance*).

Built-in Machine Learning Models in `scikit-learn`

Supervised Learning

- **Classification**
  - **Logistic Regression / SVM:** Finds optimal decision boundaries to separate classes.
  - **Random Forest:** Ensemble of decision trees maximizing voting accuracy.
- **Regression**
  - **Linear & Ridge Regression:** Models linear relationships while penalizing overfitting.
  - **Gradient Boosting:** Sequentially corrects errors of weak learners to forecast continuous values.

Unsupervised Learning

- **Clustering**
  - **K-Means:** Partitions data into K distinct clusters based on geometric centroids.
  - **DBSCAN:** Identifies arbitrary shapes using local high-density core points.
- **Dimensionality Reduction**
  - **PCA (Principal Component Analysis):** Projects data onto orthogonal axes maximizing variance.
  - **t-SNE:** Maps high-dimensional structures to 2D/3D for localized visual layout.

How to Select a Machine Learning Model

Key Selection Criteria

- **Data Scale:**
  - Scikit-Learn generally requires > 50 **samples** to start.
  - High-dimensional features ($D > N$) benefit from L1/L2 regularized models.
- **Target Variable Type:**
  - **Category:** Classification (discrete output).
  - **Quantity:** Regression (continuous output).
  - **No Targets:** Clustering or Dimensionality Reduction.
- **Performance vs. Explainability:**
  - High explanation needed $\rightarrow$ *Linear / Decision Trees*.
  - Raw accuracy priority $\rightarrow$ *Ensembles (Random Forest, Gradient Boosting)*.

Strategic Estimator Workflow

Follow the standard `sklearn` algorithm heuristic:

1. **Start Simple:** Begin with a linear estimator (e.g., *Linear SVC* or *Ridge*).
2. **Examine Dataset Size:**
   - If $< 100k$ samples, try kernel approximation or ensembles.
   - If $> 100k$ samples, pivot to out-of-core solvers (e.g., *SGD Classifier*).
3. **Increase Complexity:** Progress to non-linear kernels or boosting architectures if baseline accuracy underperforms.

Pro-Tip: Always establish a dummy baseline model first to benchmark metric improvements.

END-TO-END PIPELINE

The Machine Learning Workflow with scikit-learn

Standardizing the Lifecycle of Supervised Models

The Supervised Machine Learning Workflow

Phase 1: Initialization & Training

1. **Data Preparation (Splitting):**
   - Dividing the main dataset into **Training** (pattern extraction) and **Testing** sets (performance validation).
2. **Model Selection:**
   - Choosing the correct estimator or algorithm mapping your specific data constraints.
3. **Training the Model (Fitting):**
   - Feeding data so the algorithm adjusts its weights to minimize errors or maximize optimization goals.

Phase 2: Evaluation & Deployment

4. **Model Evaluation:**
   - Metrics assessment (Accuracy, Precision, Recall) using robust tools like cross-validation.
5. **Iterative Optimization:**
   - Enhancing performance through active experimentation and hyperparameter tuning.
6. **Persistence:**
   - Saving and loading a finalized pre-trained model for external production deployment.

Workflow Flexibility & Adaptability

Why Developers Choose Scikit-Learn

- **Streamlined Experience:** Combines a user-friendly design interface with highly consistent, well-documented APIs.
- **Accessibility:** Serves as an ideal platform for beginners while keeping the depth needed by veteran practitioners.

A Note on Extensibility

Because `scikit-learn` is a vast ecosystem, this standard timeline is **just one benchmark framework**. Real-world pipelines can vary significantly depending on data shapes and custom criteria.

Note: The steps presented above focus specifically on supervised scenarios containing features and targets.

Examples

- Regression problem: Predicting house price
- Classification problem: Classifying heart desease based on data

Thank You for Your Attention!

While Python is highly versatile across many software engineering fields, one of its most widely utilized and transformative domains is **Data Science** and **Machine Learning**.

Lecture 14: Python Recap & Summary

COMPARATIVE ANALYSIS

Programming Paradigms: C, Java, and Python

Contrasting Imperative, Declarative, Functional, and OOP Designs

1. Imperative Style: How to Solve

Concept & Focus

- Focuses on **how** to accomplish a task.
- Describes exact step-by-step control flow, state mutations, and manual loops.
- Familiar and straightforward, mapping directly to CPU operations.

C vs Python

C (Purely Imperative)

```c
#include <stdio.h>
int main() {
    int nums[] = {1, 2, 3, 4};
    int squared[4];
    for(int i = 0; i < 4; i++) {
        squared[i] = nums[i] * nums[i];
        printf("%d ", squared[i]);
    }
    return 0;
}
```

C vs Python

Python (Imperative Paradigm)

- Python completely supports this traditional procedural instruction flow.

Python (Imperative Loop)

```
numbers = [1, 2, 3, 4]
squared = []

# Manually looping and managing the state
for num in numbers:
    squared.append(num ** 2)

print(squared) # [1, 4, 9, 16]
```

2. Declarative Style: What to Solve

- Focuses on specifying the **desired outcome** rather than step-by-step control.
- Loop and index structures are completely abstracted away in the engine.

Python List Comprehension (Declarative)

```
numbers = [1, 2, 3, 4]

# No loops, counters, or appends declared!
squared = [num ** 2 for num in numbers]

print(squared) # [1, 4, 9, 16]
```

Contrasting Declarative Styles: Python vs. SQL

- Purely Declarative Frameworks:
  In a purely declarative domain-specific language like **SQL**, concepts like explicit loops, array indices, or procedural state manipulation do not exist. You describe the *criteria for the final dataset*, leaving the execution engine to find the optimal path.

Example Task: Filter for even values and square them

```sql
-- Database engine handles execution path
SELECT value * value AS squared
FROM numbers_table
WHERE value % 2 = 0;
```

- **SELECT** maps to Transformation
- **WHERE** maps to Filter condition

Functional Style: Pure Functions

- Built on function evaluation and immutable bindings.
- Treats functions as first-class citizens, passing them as arguments.

Python Map & Lambda (Functional)

```
numbers = [1, 2, 3, 4]

# Higher-order function takes
# anonymous function as argument
squared = list(map(lambda num: num ** 2, numbers))

print(squared) # [1, 4, 9, 16]
```

Pure Functional Implementation: Haskell

```haskell
-- Define a list of integers
numbers :: [Int]
numbers = [1, 2, 3, 4]

--- Map an anonymous lambda function over the list
squared :: [Int]
squared = map (\num -> num ^ 2) numbers

% Output: [1, 4, 9, 16]
```

Java (Strict OOP Model)

- Everything is inside a class; methods and attributes are tightly encapsulated.

```
public class Squarer {
    private int number;
    public Squarer(int number) {
        this.number = number;
    }
    public int getSquare() {
        return this.number * this.number;
    }
    public static void main(String[] args) {
        Squarer s = new Squarer(4);
        System.out.println(s.getSquare());
    }
}
```

Python (Flexible OOP Model)

- Explicit `self` reference, dynamic typing, and optional boilerplate.

```
class Squarer:
    def __init__(self, number):
        self.number = number

    def get_square(self):
        return self.number ** 2

# Instantiating and invoking methods
s = Squarer(4)
print(s.get_square()) # 16
```

Summary: Key Takeaways

Python's Core Philosophy

- **Multi-Paradigm Support:** Seamlessly bridges imperative, object-oriented, declarative, and functional designs.
- **Readability First:** Prioritizes simplicity, clean syntax, and rapid prototyping over low-level micro-optimizations.
- **Versatility:** Acts as a foundational tool across diverse fields, from web development to Data Science and Machine Learning.

Summary: Paradigms & Python's Strength

Real-Life Analogy: Cleaning a Room

- **Imperative:** Give step-by-step physical instructions.
- **Declarative:** Point to a picture and say "Make it look like this."
- **Functional:** Delegate specific, isolated tasks to different helpers.
- **Object-Oriented:** Define a "Room" object and call its `.clean()` method.

Python is a multi-paradigm language, meaning it lets you choose the most flexible tool for the job.

The Multi-Paradigm Trade-off

Offers ultimate adaptability by letting you blend styles (e.g., declarative data processing + OO modeling).

The Risk: Requires discipline; without it, you risk creating inconsistent, "hybrid" codebases where styles clash.

Readability & Pragmatism

Python deliberately refuses to force a single cognitive mold. By prioritizing structural clarity over micro-optimized execution, it minimizes developer overhead.

Foundational tool for **Web Dev, Data Science, and ML**.

Summary: Language as a Catalyst for Creation

- **Choosing the Right Tool**
  - There is no universally "best" programming language. The optimal choice depends entirely on your domain requirements and architectural goals.

- **Beyond Graduation**
  - A programming language is ultimately just a tool to bring your ideas to life. Your value as an engineer lies in *how* you apply these tools to build meaningful solutions.