Python Programming (DI02032011)

Complete Lecture Deck – All 45 Lectures

Milav Dabgar

July 23, 2026

Table of Contents I

Welcome to Python Programming

Unit 1: Problem Solving using Flowchart and Algorithm
Lecture 1: Introduction, Steps for problem-solving

Agenda

Introduction to Problem Solving
The Role of Computers
Systematic Approach
Steps in Problem Solving
Summary

What is Problem Solving?

Problem solving is the process of formulating a problem, finding a solution, and expressing the solution.
In computer science, it means transforming a real-world problem into a sequence of instructions a computer can execute.
It is language-independent at the initial stages.

The Role of Computers

Computers are fast, accurate, but fundamentally dumb.
They require explicit, step-by-step instructions.
The programmer's job is to bridge the gap between human intent and machine execution.
Garbage In, Garbage Out (GIGO) principle.

Why a Systematic Approach?

Reduces complexity of large problems.
Makes debugging easier and more predictable.
Allows for team collaboration and modularity.
Ensures edge cases are not missed.

Step 1: Problem Definition

Clearly understand what the problem is.
Identify the exact requirements.
Determine what the output should be.
Example: 'Find the largest of three numbers' requires clearly defining what 'largest' means (e.g., handling equal numbers).

Step 2: Input/Output Specification

Identify the inputs: What data is available?
Identify the outputs: What is the expected result?
Determine the data types (e.g., integers, strings, floating-point numbers).
Example: Input -¿ 3 integers, Output -¿ 1 integer (the largest).

Step 3: Algorithm Development

Create a logical sequence of steps to solve the problem.
Use natural language, pseudocode, or flowcharts.
Focus on the logic, not syntax.
Iterative process: refine the steps until they are unambiguous.

Example: Algorithm for Largest of Three

1. Read three numbers: A, B, C
2. If A ¿= B and A ¿= C, then Largest is A
3. Else if B ¿= A and B ¿= C, then Largest is B
4. Else Largest is C
5. Print Largest

Step 4: Implementation (Coding)

Translate the algorithm into a specific programming language (e.g., Python).
Follow the syntax rules of the language.
Write readable code (use comments, meaningful variable names).
This step is mechanical if the algorithm is solid.

Step 5: Testing and Debugging

Run the program with various test cases.
Include normal cases, edge cases (e.g., negative numbers), and invalid inputs.
Debugging: finding and fixing logical or syntax errors.
Iterative refinement.

Summary

Problem solving is a structured process.
Key steps: Define, Specify I/O, Develop Algorithm, Code, Test.
Planning (Steps 1-3) is more critical than Coding (Step 4).

Algorithm and its Characteristics

Unit 1: Problem Solving using Flowchart and Algorithm
Lecture 2: Algorithm and its characteristics

Agenda

Definition of an Algorithm
Real-world Analogy
Why Algorithms Matter
Characteristics of a Good Algorithm
Example Analysis
Summary

What is an Algorithm?

An algorithm is a finite set of unambiguous instructions that, given some set of initial conditions, can be performed in a prescribed sequence to achieve a certain goal and that has a recognizable set of end conditions. It is the logical blueprint of the solution.

Real-world Analogy

A recipe for baking a cake.
Inputs: Ingredients.
Steps: Mixing, baking at a specific temperature for a specific time.
Output: A baked cake.
If a step says 'bake until done', it is ambiguous (not a good algorithm).

Why Algorithms Matter

They provide a standard way to solve a problem.
They can be evaluated for efficiency (time and space).
They allow solutions to be implemented in any programming language.

Characteristic 1: Unambiguous

Each step must be clear and have only one interpretation.
No room for subjective execution.
Poor example: 'Add some salt.'
Good example: 'Add 1 teaspoon of salt.'

Characteristic 2: Finiteness

The algorithm must terminate after a finite number of steps.
It cannot loop indefinitely.
Infinite loops are a common bug caused by violating this characteristic.

Characteristic 3: Feasibility (Effectiveness)

Every instruction must be basic enough to be carried out.
It must be doable with available resources.
Example: 'Calculate the exact value of Pi' is not feasible as it has infinite digits.

Characteristic 4: Independent

The algorithm should be independent of any computer code.
It should rely on basic mathematical and logical operations.
Anyone reading it should be able to implement it in Python, C++, or Java.

Characteristic 5: Input and Output

Input: An algorithm takes zero or more well-defined inputs.
Output: An algorithm must produce at least one well-defined output.
An algorithm without output is doing no useful work.

Analyzing an Algorithm

Problem: Check if a number is even.

1. Start
2. Read integer N
3. Divide N by 2 and store remainder in R
4. If R == 0, Print 'Even'
5. Else Print 'Odd'
6. Stop

Check: Unambiguous? Yes. Finite? Yes. I/O defined? Yes.

Summary

An algorithm is a finite sequence of precise instructions.
Key characteristics: Unambiguous, Finiteness, Feasibility, Independent, Defined I/O.

Importance of Flowcharts and Algorithms

Unit 1: Problem Solving using Flowchart and Algorithm
Lecture 3: Importance of flowchart and algorithm

Agenda

Recap: What is an Algorithm?
Why are Algorithms Important?
Introduction to Flowcharts
Why use Flowcharts?
Flowcharts vs Algorithms
Summary

Why Algorithms are Important: Clarity

Forces the programmer to think logically and structurally.
Breaks down complex problems into manageable steps.
Helps identify missing steps or edge cases early.

Why Algorithms are Important: Efficiency

Allows for analysis of time and space complexity before coding.
Different algorithms can solve the same problem, but with wildly different performance.
Example: Searching a sorted list vs an unsorted list.

Why Algorithms are Important: Maintenance

Acts as documentation for the logic.
If the code needs to be rewritten in another language, the algorithm remains the same.
Helps new developers understand the core logic quickly.

Introduction to Flowcharts

A flowchart is a graphical or symbolic representation of a process or algorithm.
It uses different shapes to represent different types of instructions.
Arrows connect the shapes to show the flow of control.

Why use Flowcharts: Visual Clarity

A picture is worth a thousand words.
Visual representations are often easier to grasp than blocks of text.
Quickly conveys the overall structure of a program.

Why use Flowcharts: Flow of Control

Clearly illustrates branching (if/else) and looping.
Makes it easy to trace execution paths.
Helps ensure all branches eventually reach a valid end state.

Why use Flowcharts: Communication

Excellent tool for communicating logic to non-technical stakeholders (e.g., managers, clients).
Universally understood symbols bridge the gap between technical and business domains.

Flowcharts vs. Algorithms

Algorithms are text-based (step-by-step). Flowcharts are visual.
Algorithms are better for highly detailed, complex logic.
Flowcharts are better for high-level overviews and visualizing control flow.
They are complementary tools.

Example Scenario

Imagine designing a self-driving car system.
Flowchart: Visualizing the high-level decision tree (Stop at red light -¿ Yes/No).
Algorithm: The exact mathematical steps to calculate braking distance.

Summary

Algorithms ensure clarity, efficiency, and maintainability.
Flowcharts provide visual clarity, ease of communication, and clear flow of control.
Both are essential steps before writing any code.

Symbolic Representation of a Flowchart

Unit 1: Problem Solving using Flowchart and Algorithm
Lecture 4: Symbolic representation of a flowchart

Agenda

Standardized Symbols (ANSI)
Terminal, Input/Output, Process
Decision, Flow Lines
Connectors
Rules for Drawing Flowcharts
Examples

Why Standard Symbols?

To ensure universal understanding.
The American National Standards Institute (ANSI) established a standard set of flowchart symbols.
Using standard symbols makes flowcharts language and platform independent.

Terminal Symbol (Oval)

Shape: Oval or rounded rectangle.
Function: Indicates the Start or Stop of the program.
Every flowchart must have exactly one Start symbol, and at least one Stop/End symbol.

Input/Output Symbol (Parallelogram)

Shape: Parallelogram.
Function: Represents any input or output operation.
Examples: 'Read X', 'Print Y', 'Input user age'.

Process Symbol (Rectangle)

Shape: Rectangle.
Function: Represents a process, calculation, or data manipulation.
Examples: 'Sum = A + B', 'Count = Count + 1'.

Decision Symbol (Diamond)

Shape: Diamond.
Function: Indicates a decision point where the flow can branch.
Contains a condition (e.g., 'Is X ¿ Y?').
Must have multiple outward flow lines labeled (e.g., 'Yes' and 'No').

Flow Lines and Connectors

Flow Lines (Arrows): Show the exact sequence of execution.
Connectors (Small Circle): Used to connect different parts of a flowchart across pages or complex diagrams to avoid messy lines.
Labeled with a letter or number to match ends.

Rules for Drawing Flowcharts

Generally, flow should be from top to bottom, or left to right.
Flow lines should not cross each other.
Only one flow line should enter a process symbol, and only one should exit.
Only one flow line should enter a decision symbol, but two or three may exit.

Example 1: Add Two Numbers

1. Oval: Start
2. Parallelogram: Read A, B
3. Rectangle: Sum $=$ A $+$ B
4. Parallelogram: Print Sum
5. Oval: Stop

Example 2: Check Voting Eligibility

1. Start (Oval)
2. Read Age (Parallelogram)
3. Is Age ¿= 18? (Diamond)

- Yes path -¿ Print 'Eligible' (Parallelogram)
- No path -¿ Print 'Not Eligible' (Parallelogram)

4. Both paths converge to Stop (Oval)

Summary

Standard symbols: Oval (Start/Stop), Parallelogram (I/O), Rectangle (Process), Diamond (Decision).
Arrows show control flow.
Adhering to rules ensures readability.

Limitations, Control Flow, and Pseudocode

Unit 1: Problem Solving using Flowchart and Algorithm
Lecture 5: Limitations of flowchart, Flow of control, Pseudocode

Agenda

Limitations of Flowcharts
Concept of Control Flow
Types of Control Flow
What is Pseudocode?
Advantages of Pseudocode
Writing Pseudocode Examples
Summary of Unit 1

Limitations of Flowcharts

Complex Logic: Become incredibly messy and unreadable for large, complex programs.
Modifications: Difficult to alter. Often require redrawing the entire flowchart.
Reproduction: Hard to type out; requires specific software tools to draw cleanly.
Lack of standard detailing: No standard way to express detailed data structures.

Flow of Control

Flow of control refers to the order in which individual statements, instructions, or function calls are executed.
It determines how the program progresses from start to finish.
Without control flow, a program executes strictly top-to-bottom.

Types of Control Flow

1. Sequential: Default mode. Line 1, then Line 2, etc.
2. Selection (Branching): Making decisions (If/Else). Paths diverge.
3. Iteration (Looping): Repeating a block of code multiple times (For/While).

What is Pseudocode?

Pseudocode ('fake code') is an informal, high-level description of an algorithm.
Uses structural conventions of a normal programming language, but is intended for human reading rather than machine reading.
No strict syntax rules.

Why use Pseudocode?

Solves flowchart limitations: Easy to write in any text editor, easy to modify.
Scales better for complex logic.
Focuses on logic without getting bogged down by syntax errors.
Easily translated into actual programming code (like Python).

Basic Pseudocode Keywords

Input/Output: READ, GET, PRINT, DISPLAY
Processing: COMPUTE, CALCULATE, SET, INITIALIZE
Selection: IF, THEN, ELSE, ENDIF
Iteration: WHILE, DO, FOR, ENDWHILE

Example: Factorial of a Number

```
READ n
SET fact = 1
SET i = 1
WHILE i <= n DO
    fact = fact * i
    i = i + 1
ENDWHILE
PRINT fact
```

Example: Largest of Three (Pseudocode)

```
READ a, b, c
IF a > b AND a > c THEN
    PRINT a
ELSE IF b > a AND b > c THEN
    PRINT b
ELSE
    PRINT c
ENDIF
```

Comparing the Tools

Plain English: Vague, ambiguous.
Flowchart: Great visual, hard to maintain/scale.
Pseudocode: Best balance for complex logic, easy to edit.
Python Code: Final step, requires strict syntax.

Unit 1 Summary

Problem solving is a systematic process.
Algorithms are step-by-step logical solutions.
Flowcharts visualize control flow.
Pseudocode bridges the gap between logic and implementation.
You are now ready to start coding in Python!

Unit 2: Introduction to Python

Lecture 6: Introduction and Features
Overview of Python's history, philosophy, and core features.

Agenda

1. What is Python?
2. Brief History and Origin
3. The Zen of Python
4. Core Python Features
5. Comparison with other languages

What is Python?

Python is a high-level, general-purpose programming language.
It emphasizes code readability with the use of significant indentation.
Supports multiple programming paradigms: procedural, object-oriented, and functional.
Often described as a 'batteries included' language due to its comprehensive standard library.

History of Python

Created by Guido van Rossum in the late 1980s.
First released (version 0.9.0) in 1991.
Designed as a successor to the ABC programming language.
Name inspired by the British comedy group Monty Python, not the snake.

The Zen of Python

A collection of 19 guiding principles for writing computer programs.
Accessible by typing ‘import this‘ in the Python interpreter.
Key principles include:
- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.

Key Feature: Easy to Learn and Use

Simple syntax similar to the English language.
Minimal use of special characters; relies on indentation instead of braces ‘{}‘.
Allows developers to write programs with fewer lines than some other languages.

Key Feature: Interpreted Language

Python code is executed line-by-line by the interpreter.
Makes debugging easier as execution stops on the first error.
No separate compilation step is required, facilitating rapid application development.

Key Feature: Dynamically Typed

Variables do not need an explicit declaration to reserve memory space. The type of the variable is determined at runtime based on the assigned value.

```
x = 10       # x is an integer
x = 'Hello' # x is now a string
```

Key Feature: Platform Independent

Python code can run on various operating systems without modification.
Write Once, Run Anywhere (WORA).
Interpreters are available for Windows, macOS, Linux, and more.

Key Feature: Extensive Standard Library

Provides rich modules and functions for various tasks.
Includes libraries for regular expressions, string manipulation, internet protocols, databases, etc.
Reduces the need to write code from scratch.

A Simple Comparison: Python vs Java

Python:

```
print('Hello, World!')
```

Java:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Summary

Python is a high-level, interpreted, dynamically typed language.
Its design philosophy emphasizes code readability.
Key features include cross-platform support, a massive standard library, and multi-paradigm support.

Applications of Python Programming

Lecture 7
Exploring the diverse real-world use cases of Python.

Agenda

1. Web Development
2. Data Science and Analytics
3. Machine Learning and AI
4. Automation and Scripting
5. Desktop GUI Applications
6. Game Development

1. Web Development

Python is widely used for server-side web development.
Offers robust frameworks that handle routing, templating, and database interaction.
Popular Frameworks:
- **Django:** A high-level, full-stack framework that encourages rapid development.
- **Flask:** A lightweight, micro-framework ideal for small to medium apps and APIs.

2. Data Science and Analytics

Python is the leading language in data science.
Used for data cleaning, visualization, and complex statistical analysis.
Key Libraries:
- **Pandas:** Powerful data structures for data manipulation.
- **NumPy:** Support for large, multi-dimensional arrays and matrices.
- **Matplotlib / Seaborn:** Data visualization and plotting.

Data Science Example

A simple data analysis snippet using Pandas:

```
import pandas as pd

# Load data from a CSV file
df = pd.read_csv('sales.csv')

# Calculate average sales per region
avg_sales = df.groupby('region')['amount'].mean()
print(avg_sales)
```

3. Machine Learning and AI

Python dominates the Artificial Intelligence and Machine Learning landscape.
Provides pre-built algorithms and tools for training models.
Key Libraries/Frameworks:
- **Scikit-learn:** For classic ML algorithms (classification, regression).
- **TensorFlow / Keras:** Deep learning developed by Google.
- **PyTorch:** Deep learning favored in research, developed by Meta.

4. Automation and Scripting

Python is excellent for automating repetitive tasks (scripting).
Can interact with the OS, filesystems, and networks easily.
Common Tasks:
- Renaming files in a directory.
- Scraping data from websites using ‘BeautifulSoup‘ or ‘Selenium‘.
- Automating emails or Excel reports.

Automation Example

Renaming all ‘.txt‘ files in a folder to ‘.md‘:

```
import os

for filename in os.listdir('.'):
    if filename.endswith('.txt'):
        new_name = filename.replace('.txt', '.md')
        os.rename(filename, new_name)
```

5. Desktop GUI Applications

Python can be used to create cross-platform desktop applications.
While not as common as web apps, the tools are robust.
Frameworks:
- **Tkinter:** The standard GUI toolkit included with Python.
- **PyQt / PySide:** Wrappers for the powerful Qt framework.
- **Kivy:** Great for multi-touch applications and mobile.

6. Game Development

Python is used in game development, primarily for 2D games or as a scripting language in 3D engines.
Key Library: Pygame
- Provides modules for graphics, sound, and input handling.
- Excellent for learning game programming concepts.
Many AAA game engines (like Unreal Engine) use Python for tool scripting and automation in the pipeline.

Other Notable Domains

Network Programming:
- Tools like ‘Twisted‘ and ‘asyncio‘ handle asynchronous network operations.
Embedded Systems and IoT:
- ‘MicroPython‘ and ‘CircuitPython‘ run on microcontrollers.
Cybersecurity:
- Writing penetration testing scripts and analyzing malware.

Summary

Python is truly general-purpose.
It excels in Web Development, Data Science, and Machine Learning.
It is also a fantastic tool for automation, desktop apps, and even game development.
The vast ecosystem of third-party libraries enables these applications.

Python Installation and Setup

Lecture 8
Setting up your environment to write and run Python code.

Agenda

1. Downloading Python
2. Installing on Windows, macOS, and Linux
3. Understanding the PATH variable
4. Verifying the Installation
5. Introduction to Python IDEs
6. Running your first script

Downloading Python

The official source for Python is **python.org**.
Always download the latest stable release (e.g., Python 3.12+).
Python 2 vs Python 3:
- Python 2 reached its End of Life in 2020.
- We exclusively use Python 3.
Choose the installer matching your operating system architecture (usually 64-bit).

Installation on Windows

1. Run the downloaded ‘.exe‘ installer.
2. **CRITICAL STEP:** Check the box that says **”Add Python to PATH”** at the bottom of the first screen.
3. Click ”Install Now”.
4. (Optional) Disable path length limit at the end of the installation.

What is the PATH Variable?

An environment variable that tells the operating system where to look for executable files.
When you type ‘python‘ in the command prompt, the OS searches the directories listed in PATH.
If Python’s installation directory is not in PATH, you would have to type the full path (e.g., ‘C:\{}Python39\{}python.exe‘) every time.

Installation on macOS and Linux

macOS:
- Use the macOS installer from python.org.
- Alternatively, use Homebrew: ‘brew install python‘
Linux:
- Most Linux distros come with Python pre-installed.
- To update or install: ‘sudo apt update && sudo apt install python3‘ (Debian/Ubuntu).

Verifying the Installation

Open a Command Prompt (Windows) or Terminal (macOS/Linux).
Type:

```
python --version
# OR
python3 --version
```

Expected Output:
‘Python 3.12.2‘
If you see this, Python is successfully installed and added to PATH.

Integrated Development Environments (IDEs)

While you can write Python in Notepad, an IDE provides essential tools:
- Syntax highlighting.
- Autocomplete / IntelliSense.
- Debugging tools.
IDLE: Python's built-in, basic IDE. Good for quick tests.
VS Code: A powerful, free code editor by Microsoft. Highly recommended.
PyCharm: A dedicated, feature-rich Python IDE by JetBrains.

Setting up VS Code for Python

1. Download and install VS Code.
2. Open VS Code and go to the Extensions view (‘Ctrl+Shift+X‘).
3. Search for "Python" and install the official extension by Microsoft.
4. Create a new file, save it as ‘hello.py‘.
5. VS Code will automatically detect your Python interpreter.

Running Your First Script

In your ‘hello.py‘ file, write:

```
print('Hello, World!')
```

To run it:
- In VS Code: Click the "Play" button in the top right.
- From Terminal: Navigate to the folder and run:

```
python hello.py
```

The Python Interactive Shell (REPL)

REPL stands for **Read-Eval-Print Loop**.
Type ‘python‘ in your terminal without a filename.
You get a ‘¿¿¿‘ prompt.
You can type Python code and get immediate results.

```
>>> 2 + 3
5
>>> print('Hi')
Hi
```

Type ‘exit()‘ or press ‘Ctrl+Z‘/‘Ctrl+D‘ to leave.

Summary

We installed Python 3 from the official website.
We ensured Python was added to our system's PATH variable.
We verified the installation using the terminal.
We set up VS Code and ran our first 'hello.py' script.

Basic Structure of a Python Program

Lecture 9
Understanding how Python code is organized and formatted.

Agenda

1. Anatomy of a Python Script
2. The Role of Indentation
3. Writing Comments
4. Statements and Multi-line Statements
5. Importing Modules
6. The ‘__main__‘ block

Anatomy of a Python Script

A typical Python program consists of:
1. **Imports:** Bringing in external modules/libraries.
2. **Global Variables:** Variables accessible throughout the file.
3. **Function/Class Definitions:** Reusable blocks of code.
4. **Main Execution Block:** The entry point of the script.

Python executes files from top to bottom.

Indentation: Python's Defining Feature

Most languages (C, Java, JavaScript) use curly braces ‘{}‘ to define blocks of code.
Python uses **whitespace (indentation)**.
- Defines scope for loops, functions, and conditional statements.
- Standard convention is **4 spaces** per indentation level.
- Do not mix spaces and tabs! (Causes ‘TabError‘).

Indentation Example

Correct:

```
if 5 > 2:
    print('Five is greater than two.')
    print('Still inside the if block.')
print('Outside the if block.')
```

Incorrect (IndentationError):

```
if 5 > 2:
print('This will cause an error')
```

Comments in Python

Comments are ignored by the interpreter.
Used to explain code, make it readable, and prevent execution during testing.
Single-line comments: Start with a hash ‘#‘.

```
# This is a comment
x = 5  # Inline comment
```

Multi-line Comments and Docstrings

Python does not have a specific syntax for multi-line comments.
You can use ‘#‘ on multiple lines, or use multi-line strings (“”’‘ or “”””‘) that are not assigned to a variable.
Docstrings (Documentation Strings):

```
def my_function():
    """
    This function does nothing yet.
    Used to document functions/classes.
    """
    pass
```

Python Statements

A statement is an instruction that the Python interpreter can execute.
- Example: ‘x = 5‘ is an assignment statement.
Usually, one statement per line. No semicolon ‘;‘ is needed at the end.
You *can* use semicolons to put multiple statements on one line (but it is unpythonic).

```
x = 5; y = 10; print(x + y)  # Discouraged
```

Multi-line Statements

To split a long statement over multiple lines, use the backslash ‘\{}‘ as a line continuation character.

```
total = 1 + 2 + 3 + \
        4 + 5 + 6
```

Alternatively, line continuation is implied inside parentheses ‘()‘, brackets ‘[]‘, and braces ‘{}‘.

```
total = (1 + 2 + 3 +
         4 + 5 + 6) # Preferred method
```

Importing Modules

Python is extended through modules.
Use the ‘import‘ keyword to bring in external code.
Usually placed at the very top of the file.

```
import math
print(math.sqrt(16))

from datetime import date
print(date.today())
```

The __main__ Block

How do we prevent code from running when a file is imported elsewhere?
Use the ‘if __name__ == "__main__":‘ check.

```
def calculate():
    return 10 * 5

if __name__ == "__main__":
    # This only runs if this script is executed directly
    # It will NOT run if this file is imported
    result = calculate()
    print(result)
```

Summary

Python relies on **indentation** to define blocks, not braces.
Use ‘#‘ for comments and triple quotes ‘”””“ for docstrings.
Statements don’t need semicolons.
Use parentheses for long multi-line statements.
The ‘if __name__ == ’__main__’‘ pattern is standard for scripts.

Tokens in Python

Lecture 10: Keywords, Variables, Data Types, & Operators
Understanding the fundamental building blocks of Python data handling.

Agenda

1. Keywords and Identifiers
2. Variables and Assignment
3. Primitive Data Types
4. Arithmetic Operators
5. Relational (Comparison) Operators
6. Logical and Assignment Operators

Python Keywords

Keywords are reserved words that have special meaning to the interpreter.
They cannot be used as variable names, function names, or any other identifier.
Examples: ‘if‘, ‘else‘, ‘while‘, ‘for‘, ‘def‘, ‘class‘, ‘import‘, ‘True‘, ‘False‘, ‘None‘.
Most are lowercase, except ‘True‘, ‘False‘, and ‘None‘.
To see the list:

```
import keyword
print(keyword.kwlist)
```

Identifiers and Naming Rules

Identifiers are the names given to variables, classes, methods, etc.
Rules:
1. Can contain letters (a-z, A-Z), digits (0-9), and underscores ‘_‘.
2. Cannot start with a digit (‘1variable‘ is invalid).
3. Cannot be a keyword.
4. Case-sensitive (‘Var‘ and ‘var‘ are different).

Best Practice (PEP 8):
- Variables and functions: ‘snake_case‘
- Classes: ‘CamelCase‘

Variables and Assignment

Variables act as containers for storing data values.
Python has no command for declaring a variable; it is created the moment you assign a value.

```
age = 20          # Integer assignment
name = 'Alice'    # String assignment

# Multiple Assignment
x, y, z = 1, 2, 3
a = b = c = 10
```

Variables are actually references (pointers) to objects in memory.

Core Data Types

Data types define the kind of data a variable holds.
Numeric Types:
- ‘int‘: Integers (e.g., ‘10‘, ‘-5‘)
- ‘float‘: Floating-point numbers (e.g., ‘3.14‘, ‘-0.001‘)
- ‘complex‘: Complex numbers (e.g., ‘3 + 4j‘)
Sequence Types:
- ‘str‘: Strings, sequence of characters (e.g., ‘’Hello’‘)
Boolean:
- ‘bool‘: Represents ‘True‘ or ‘False‘

Checking Data Types

Use the built-in ‘type()‘ function to find out the type of a variable.

```
x = 10
print(type(x))  # <class 'int'>

y = 3.14
print(type(y))  # <class 'float'>

z = 'Hello'
print(type(z))  # <class 'str'>

is_active = True
print(type(is_active))  # <class 'bool'>
```

Arithmetic Operators

Used with numeric values to perform common mathematical operations.

- ‘+‘ Addition
- ‘-‘ Subtraction
- ‘*‘ Multiplication
- ‘/‘ Division (always returns float)
- ‘//‘ Floor Division (discards remainder)
- ‘%‘ Modulus (returns remainder)
- ‘**‘ Exponentiation (power)

```
print(10 / 3)    # 3.3333333333333335
print(10 // 3)   # 3
```

Relational (Comparison) Operators

Used to compare two values. They return a Boolean (‘True‘ or ‘False‘).
- ‘==‘ Equal to
- ‘!=‘ Not equal to
- ‘¿‘ Greater than
- ‘¡‘ Less than
- ‘¿=‘ Greater than or equal to
- ‘¡=‘ Less than or equal to

```
x = 5
y = 10
print(x == y)   # False
print(x < y)    # True
```

Logical Operators

Used to combine conditional statements.
- ‘and‘: Returns True if both statements are true.
- ‘or‘: Returns True if one of the statements is true.
- ‘not‘: Reverse the result, returns False if the result is true.

```
x = 5
print(x > 3 and x < 10)   # True
print(not(x > 3))         # False
```

Short-circuit evaluation is used: if the first operand in ‘and‘ is False, the second is not evaluated.

Assignment Operators

Used to assign values to variables.
Include compound assignment operators for shorthand operations.
- ‘=‘ Assign
- ‘+=‘ Add and assign (‘x += 3‘ is ‘x = x + 3‘)
- ‘-=‘ Subtract and assign
- ‘*=‘ Multiply and assign
- ‘/=‘ Divide and assign

```
count = 0
count += 1  # count is now 1
```

Note: Python does not have ‘++‘ or ‘–‘ increment/decrement operators.

Summary

Keywords are reserved; identifiers must follow naming rules.
Variables are dynamically created on assignment.
Basic types include ‘int‘, ‘float‘, ‘str‘, and ‘bool‘.
We use Arithmetic, Relational, Logical, and Assignment operators to manipulate data.

Type Conversion in Python

Lecture 11
Understanding how data is converted from one type to another.

Agenda

1. What is Type Conversion?
2. Implicit Type Conversion
3. Explicit Type Conversion (Type Casting)
4. Built-in Casting Functions (‘int()‘, ‘float()‘, ‘str()‘)
5. Edge Cases and Conversion Errors
6. Practical Examples

What is Type Conversion?

The process of converting the value of one data type (integer, string, float, etc.) to another data type.
Also known as **Type Casting**.
Why do we need it?
- Receiving input from a user (which is always a string) and performing math on it.
- Combining different data types in output strings.
- Preventing data loss during mathematical operations.

Two Types of Conversion

Python handles type conversion in two ways:
1. Implicit Type Conversion (Coercion):
- Python automatically converts one data type to another.
- No user involvement is needed.
2. Explicit Type Conversion (Type Casting):
- The programmer manually converts the data type using built-in functions.
- Necessary when Python cannot automatically resolve type mismatches.

Implicit Type Conversion

When operating on mixed numeric types, Python automatically converts the smaller type to the larger/wider type to prevent data loss.

```
num_int = 123
num_float = 1.23

# Adding int and float
num_new = num_int + num_float

print("Type of num_new:", type(num_new))
# Output: <class 'float'>
print("Value:", num_new)
# Output: 124.23
```

Limitations of Implicit Conversion

Python cannot automatically convert entirely different data types, like strings and integers.

```
num_int = 123
num_str = "456"

# This will cause a TypeError
result = num_int + num_str
```

Error: ‘TypeError: unsupported operand type(s) for +: 'int' and 'str'‘
This is where Explicit Conversion is required.

Explicit Type Conversion (Casting)

We use built-in functions to force the conversion.
Common Functions:
- ‘int(a)‘: Converts ‘a‘ to an integer.
- ‘float(a)‘: Converts ‘a‘ to a float.
- ‘str(a)‘: Converts ‘a‘ to a string.
- ‘bool(a)‘: Converts ‘a‘ to a boolean.

Example: Casting String to Integer

Solving the TypeError from earlier:

```
num_int = 123
num_str = "456"

# Convert string to integer explicitly
num_str_converted = int(num_str)

result = num_int + num_str_converted
print("Sum:", result)
# Output: Sum: 579
```

Example: Casting Numbers to String

Useful for string concatenation.

```
age = 20

# TypeError: can only concatenate str (not "int") to str
# message = "I am " + age + " years old"

# Correct way using explicit conversion
message = "I am " + str(age) + " years old"
print(message)
```

(Note: Modern Python usually handles this with f-strings, which implicitly convert to string: ‘f”I am {age}”‘)

Loss of Data in Conversion

Explicit conversion can sometimes result in data loss.
Converting a float to an int **truncates** the decimal part.

```
pi = 3.14159
integer_pi = int(pi)

print(integer_pi)
# Output: 3
```

It does not round; it completely drops the decimal values.

Conversion Errors (ValueError)

Explicit conversion will fail if the data cannot be logically converted.

```
text = "Hello"
num = int(text)
```

Error: ‘ValueError: invalid literal for int() with base 10: ’Hello’‘
You can only cast strings to ints if the string contains only digits.
Similarly, ‘int(”3.14”)‘ will fail. You must cast to float first.

Summary

Implicit Conversion: Python handles it automatically (e.g., int + float = float).
Explicit Conversion: Programmer uses functions like ‘int()‘, ‘float()‘, ‘str()‘.
Explicit conversion is required when working with incompatible types (like string + int).
Be aware of potential data loss (float to int) and ValueErrors.

Introduction to Flow of Control

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 12: Introduction to Flow of Control
Focus: Control flow, sequential vs branching vs looping

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Introduction to Flow of Control

Introduction to Flow of Control is a core construct in Python for controlling execution flow.
It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Introduction to Flow of Control mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Introduction to Flow of Control

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Introduction to Flow of Control

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Introduction to Flow of Control

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Introduction to Flow of Control.
Key concepts covered:
- Control flow, sequential vs branching vs looping
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization

Next: Review the provided exercises and practice edge cases.

Selection: If statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 13: Selection: If statement
Focus: Boolean expressions, conditional execution, indentation

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Selection: If statement

Selection: If statement is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Selection: If statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Selection: If statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Selection: If statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Selection: If statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Selection: If statement.
Key concepts covered:
- Boolean expressions, conditional execution, indentation
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Elif statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 14: Elif statement
Focus: Multi-way branching, if-elif-else ladders, mutually exclusive conditions

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Elif statement

Elif statement is a core construct in Python for controlling execution flow.
It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Elif statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Elif statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Elif statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Elif statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Elif statement.
Key concepts covered:
- Multi-way branching, if-elif-else ladders, mutually exclusive conditions
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Nested if statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 15: Nested if statement
Focus: If within if, complex logic trees, short-circuit evaluation

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Nested if statement

Nested if statement is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Nested if statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Nested if statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Nested if statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Nested if statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Nested if statement.
Key concepts covered:
- If within if, complex logic trees, short-circuit evaluation
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Repetition: For loop

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 16: Repetition: For loop
Focus: Iterables, iterators, range(), basic for loop syntax

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Repetition: For loop

Repetition: For loop is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Repetition: For loop mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Repetition: For loop

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Repetition: For loop

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Repetition: For loop

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Repetition: For loop.
Key concepts covered:
- Iterables, iterators, range(), basic for loop syntax
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization

Next: Review the provided exercises and practice edge cases.

For loop examples

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 17: For loop examples
Focus: List comprehensions, iterating dictionaries, zip(), enumerate()

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to For loop examples

For loop examples is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # For loop examples mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: For loop examples

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: For loop examples

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: For loop examples

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of For loop examples.
Key concepts covered:
- List comprehensions, iterating dictionaries, zip(), enumerate()
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization

Next: Review the provided exercises and practice edge cases.

While loop

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 18: While loop
Focus: Condition-controlled loops, infinite loops, state management

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to While loop

While loop is a core construct in Python for controlling execution flow.
It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # While loop mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: While loop

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: While loop

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: While loop

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of While loop.
Key concepts covered:
- Condition-controlled loops, infinite loops, state management
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

While loop examples

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 19: While loop examples
Focus: Input validation, simulation, dynamic termination conditions

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to While loop examples

While loop examples is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # While loop examples mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: While loop examples

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: While loop examples

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: While loop examples

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of While loop examples.
Key concepts covered:
- Input validation, simulation, dynamic termination conditions
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Nested loop

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 20: Nested loop
Focus: Loop within loop, independent vs dependent iterators, time complexity O(N^2)

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Nested loop

Nested loop is a core construct in Python for controlling execution flow.
It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Nested loop mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Nested loop

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Nested loop

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Nested loop

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Nested loop.
Key concepts covered:
- Loop within loop, independent vs dependent iterators, time complexity O(N^2)
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Nested loop examples

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 21: Nested loop examples
Focus: Matrix operations, pattern printing, combinatorial search

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Nested loop examples

Nested loop examples is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Nested loop examples mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Nested loop examples

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Nested loop examples

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Nested loop examples

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Nested loop examples.
Key concepts covered:
- Matrix operations, pattern printing, combinatorial search
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Break Statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 22: Break Statement
Focus: Early termination of loops, search algorithms, break vs return

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Break Statement

Break Statement is a core construct in Python for controlling execution flow.
It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Break Statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Break Statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Break Statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Break Statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Break Statement.
Key concepts covered:
- Early termination of loops, search algorithms, break vs return
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization
Next: Review the provided exercises and practice edge cases.

Continue Statement

Course: Python Programming (DI02032011)
Unit 3: Flow of Control
Lecture 23: Continue Statement
Focus: Skipping iterations, loop filtering, continue in while vs for

Agenda

1. Introduction to Concept
2. Syntax & Mechanics
3. Memory & Under-the-hood
4. Basic Examples
5. Advanced Examples
6. Common Pitfalls & Edge Cases
7. Performance Considerations
8. Best Practices
9. Real-World Application
10. Summary

Introduction to Continue Statement

Continue Statement is a core construct in Python for controlling execution flow. It fundamentally alters the instruction pointer in the CPython bytecode evaluator.

```
import dis
def demo():
    # Continue Statement mechanics
    pass
dis.dis(demo)
```

Bytecode instructions like ‘POP_JUMP_IF_FALSE‘ or ‘JUMP_ABSOLUTE‘ drive this behavior.

Syntax & Mechanics: Continue Statement

Python enforces block structure via indentation.
A colon ‘:‘ initiates a new block.

```
# Abstract structural example
statement_1
control_flow_keyword condition:
    statement_2  # Indented (usually 4 spaces)
    statement_3
statement_4
```

Mixing tabs and spaces will result in ‘TabError‘.

Memory & Under-the-hood

Variables defined inside control flow blocks (like if/for/while) DO NOT have block scope.
Python only has function, class, and module scopes.

```
if True:
    leak_var = 100
print(leak_var)  # Valid! Outputs 100
```

However, if the block never executes, the variable remains undefined (resulting in ‘NameError‘).

Basic Example: Continue Statement

Let's look at a fundamental use case.

```
def basic_demo(data):
    # Processing data
    res = []
    for item in data:
        res.append(item * 2)
    return res
```

This simple pattern is ubiquitous but can often be optimized.

Advanced Example: Continue Statement

Leveraging deeper Python features.

```
def advanced_demo(data):
    # More Pythonic approach
    return [x * 2 for x in data if x is not None]
```

Using comprehensions or built-in functions often executes closer to C speed, bypassing the CPython evaluation loop overhead for each iteration.

Common Pitfalls & Edge Cases

1. Modifying a collection while iterating over it.
2. Misunderstanding Truthy/Falsy (‘if [0]:‘ vs ‘if 0:‘).
3. Infinite loops due to floating point inaccuracies (‘while x != 1.0:‘).

```
lst = [1, 2, 3]
for x in lst:
    if x == 2:
        lst.remove(x)  # DANGEROUS!
```

Always iterate over a copy or use list comprehensions.

Performance Considerations

Function calls inside tight loops are expensive.

```
# Slow
for i in range(10000):
    math.sqrt(i)

# Fast (local variable binding)
sqrt = math.sqrt
for i in range(10000):
    sqrt(i)
```

Attribute lookups (‘math.sqrt‘) require dictionary lookups on every iteration.

Best Practices

- Keep blocks small and modular. Extract deep nesting into functions.
- Use ‘enumerate()‘ instead of ‘range(len())‘.
- Avoid deep nesting (Arrow Anti-Pattern).

```
# Avoid this:
if a:
    if b:
        if c:
            do_something()
# Do this:
if a and b and c:
    do_something()
```

Real-World Application

Parsing a log file state-machine style.

```
def parse_logs(file_path):
    error_count = 0
    with open(file_path) as f:
        for line in f:
            if 'ERROR' in line:
                error_count += 1
                if error_count > 100:
                    print('Threshold exceeded!')
                    break
```

File objects are iterators; this reads one line at a time, preventing OOM on huge files.

Summary

We explored the mechanics of Continue Statement.
Key concepts covered:
- Skipping iterations, loop filtering, continue in while vs for
- Indentation and block structures
- Memory scoping behavior
- Iteration and performance optimization

Next: Review the provided exercises and practice edge cases.

Lecture 24: Intro to Functions

Introduction to user-defined functions.
Understanding DRY (Don't Repeat Yourself).

Agenda

What is a Function?
Defining Functions in Python
Function Syntax
Calling a Function
Return Values
Docstrings
Function Best Practices

What is a Function?

A function is a block of organized, reusable code that is used to perform a single, related action.
Functions provide better modularity for your application and a high degree of code reusing.
Python gives you many built-in functions like print(), etc., but you can also create your own functions (user-defined functions).

Why Use Functions?

DRY Principle: Avoid code duplication.
Abstraction: Hide complex implementation details.
Modularity: Break down large problems into smaller, manageable chunks.
Maintainability: Easier to fix bugs when code is organized.

Defining Functions

Use the ‘def‘ keyword followed by the function name and parentheses ‘()‘.
Any input parameters should be placed within these parentheses.
The code block within every function starts with a colon ‘:‘ and is indented.

Calling a Function

To execute the function, simply type its name followed by parentheses. Function calls can be placed anywhere in the code after the function is defined.

Return Statement

The ‘return‘ statement exits a function, optionally passing back an expression to the caller.
A return statement with no arguments is the same as returning ‘None‘.

Returning Multiple Values

Python allows a function to return multiple values as a tuple.

Docstrings

Docstrings (documentation strings) describe what your function does. Enclosed in triple quotes “”””“ immediately after the function header.

Function Stubs (pass)

Sometimes you want to define a function signature but implement it later. Use the ‘pass‘ keyword to avoid syntax errors.

Summary

Functions modularize code and prevent repetition.
Defined with ‘def‘, executed with ‘()‘.
Can return values using ‘return‘.

Next Lecture

In the next lecture, we will dive deeper into Arguments and Parameters, exploring positional, keyword, and default arguments.

Lecture 25: Arguments and Parameters

Deep dive into function arguments.
Positional, keyword, default, and variable-length arguments.

Agenda

Parameters vs Arguments
Positional Arguments
Keyword Arguments
Default Parameters
*args (Variable Positional)
**kwargs (Variable Keyword)
Argument Ordering

Parameters vs Arguments

Parameter: A variable defined in the function declaration.
Argument: The actual value passed to the function when it is called.

Positional Arguments

Arguments passed to a function in the exact order they are defined.
The number of arguments must match the number of parameters.

Keyword Arguments

Arguments passed by explicitly stating the parameter name.
Order does not matter.

Default Parameters

Parameters can have default values. If no argument is provided, the default is used.
Default parameters must follow non-default parameters.

Mutable Default Parameters Pitfall

NEVER use mutable objects (like lists or dictionaries) as default arguments.
They are evaluated only once at definition time, leading to shared state.

Variable-Length Positional Arguments (*args)

Allows a function to accept any number of positional arguments.
Stored as a tuple inside the function.

Variable-Length Keyword Arguments (**kwargs)

Allows a function to accept any number of keyword arguments.
Stored as a dictionary inside the function.

Argument Ordering Rule

When combining argument types, they MUST follow this order:

1. Positional arguments
2. *args
3. Default arguments / Keyword arguments
4. **kwargs

Summary

Python offers immense flexibility in how functions accept data. Mastering *args and **kwargs is essential for writing generic, reusable functions.

Next Lecture

Next, we will explore Variable Scope: the difference between global and local variables.

Lecture 26: Variable Scope

Understanding Local, Global, and Enclosing scopes.
The LEGB rule.

Agenda

What is Scope?
Local Variables
Global Variables
The ‘global‘ Keyword
Enclosing Scope & ‘nonlocal‘
The LEGB Rule
Common Pitfalls

What is Scope?

Scope refers to the region of code where a variable is accessible.
Variables created inside a function are not accessible outside of it.
Python resolves variable names using the LEGB rule (Local, Enclosing, Global, Built-in).

Local Variables

Variables defined inside a function have local scope.
They are destroyed when the function returns.

Global Variables

Variables defined at the top level of a script/module.
Accessible from anywhere within the file.

Shadowing Global Variables

If you assign to a variable inside a function, Python assumes it's a local variable.
This can shadow a global variable with the same name.

The ‘global‘ Keyword

To modify a global variable inside a function, use the ‘global‘ keyword.

Enclosing Scope (Nested Functions)

Functions can be defined inside other functions.
The inner function can access variables in the outer function's scope (Enclosing scope).

The ‘nonlocal‘ Keyword

Used to modify a variable in an enclosing (non-global) scope.

The LEGB Rule Summary

When you reference a variable, Python searches in this order:

1. Local (inside current function)
2. Enclosing (inside enclosing functions)
3. Global (top level of module)
4. Built-in (Python's built-in namespace)

Summary

Scope protects variables from accidental modification.
Understand LEGB to avoid NameErrors and shadowing bugs.
Use ‘global‘ and ‘nonlocal‘ sparingly.

Next Lecture

Next, we look at the Python Standard Library and Built-in Functions.

Lecture 27: Built-in Functions

Exploring Python's standard built-in functions.
Functions that are always available without importing.

Agenda

What is the Standard Library?
Type Conversions
Sequence Operations
Object Introspection
Map and Filter
Zip and Enumerate
Common Pitfalls

The Built-in Namespace

Python comes with many functions ready to use.
These reside in the 'Built-in' scope of the LEGB rule.
No ‘import‘ statement is required.
Examples: ‘print()‘, ‘len()‘, ‘type()‘, ‘id()‘.

Type Conversion Functions

Functions to cast between different data types.
‘int(x)‘, ‘float(x)‘, ‘str(x)‘, ‘bool(x)‘.

Sequence & Collection Functions

Functions that operate on iterables.
‘len(s)‘: Returns the number of items.
‘sorted(iterable)‘: Returns a new sorted list.
‘reversed(seq)‘: Returns a reverse iterator.

Introspection Functions

Functions to inspect objects at runtime.
‘type(obj)‘: Returns the type of the object.
‘id(obj)‘: Returns the memory address.
‘isinstance(obj, class)‘: Checks inheritance.
‘dir(obj)‘: Lists attributes and methods.

The ‘enumerate()‘ Function

Adds a counter to an iterable and returns it as an enumerate object.
Highly useful in ‘for‘ loops.

The ‘zip()‘ Function

Takes iterables, aggregates them in a tuple, and returns it.

Higher-Order Functions: map()

Applies a function to all the items in an input iterable.

Higher-Order Functions: filter()

Constructs an iterator from elements for which a function returns True.

Summary

Built-in functions provide powerful, optimized operations.
Using them makes your code more Pythonic and efficient.
Avoid reinventing the wheel!

Next Lecture

We will focus deeply on input/output functions: ‘input()‘ and ‘print()‘.

Lecture 28: I/O Operations

Standard Input and Output in Python.
Mastering ‘input()‘ and ‘print()‘.

Agenda

Standard I/O Streams
The ‘print()‘ function in detail
Formatting Output
The ‘input()‘ function
Type Casting Input
Handling Multiple Inputs
Summary

Standard Streams

Python programs typically interact with standard streams.
stdin (Standard Input): Usually the keyboard.
stdout (Standard Output): Usually the screen/console.
stderr (Standard Error): Used for error messages.

The print() Function

Used to send data to standard output.
Signature: ‘print(*objects, sep=’ ’, end=’\{}n’, file=sys.stdout, flush=False)‘.
Can print multiple objects separated by commas.

Customizing print()

The ‘sep‘ parameter changes the separator between objects.
The ‘end‘ parameter changes what is printed at the very end.

Output Formatting (f-strings)

Introduced in Python 3.6, f-strings offer the most readable way to format output.
Prefix string with 'f' and use ‘{}‘ for expressions.

The input() Function

Reads a line from input, converts it to a string (stripping a trailing newline), and returns it.
Optionally takes a prompt string.

Type Casting with input()

Because ‘input()‘ ALWAYS returns a string, you must cast it for math operations.

Handling Multiple Inputs

You can read multiple values on a single line using ‘.split()‘.

Flushing the Output Buffer

Sometimes output is buffered and not displayed immediately.
Use ‘flush=True‘ in ‘print()‘ to force immediate output, crucial for loading bars.

Summary

‘print()‘ is versatile with ‘sep‘ and ‘end‘.
Always cast the result of ‘input()‘ if numerical data is expected.
f-strings are the modern standard for formatting.

Next Lecture

Next, we delve into mathematical built-in functions like ‘abs()‘, ‘max()‘, and ‘min()‘.

Lecture 29: Math Functions (Part 1)

Exploring built-in math capabilities.
abs(), divmod(), max(), and min().

Agenda

Mathematical built-ins
The abs() function
The divmod() function
The max() function
The min() function
Using max/min with key parameter
Summary

Built-in Math

Python provides several mathematical functions in the built-in namespace.
No need to ‘import math‘ for these fundamental operations.
Highly optimized and written in C.

The abs() Function

Returns the absolute value of a number.
For integers and floats, it returns the positive value.
For complex numbers, it returns the magnitude.

The divmod() Function

Takes two non-complex numbers and returns a tuple: (quotient, remainder).
Equivalent to ‘(a // b, a % b)‘.

Why use divmod()?

It is more efficient than performing ‘//‘ and ‘%‘ separately.
Commonly used in time conversions (e.g., seconds to minutes and seconds).

The max() Function

Returns the largest item in an iterable or the largest of two or more arguments.

The min() Function

Returns the smallest item in an iterable or the smallest of two or more arguments.

Advanced max() and min()

Both accept a ‘key‘ parameter, a one-argument function used to customize the sort order.

Corner Cases

Passing an empty iterable to max/min raises a ValueError.
Use the ‘default‘ keyword argument to prevent this.

Summary

‘abs()‘ for absolute values.
‘divmod()‘ for efficient quotient/remainder.
‘max()‘ and ‘min()‘ are powerful, especially with the ‘key‘ argument.

Next Lecture

Next, we continue with more math functions: ‘pow()‘ and ‘sum()‘.

Lecture 30: Math Functions (Part 2)

Continuing with built-in math functions.
Deep dive into pow() and sum().

Agenda

The pow() function
Two-argument pow() vs **
Three-argument pow() (Modular Exponentiation)
The sum() function
sum() start parameter
Floating point precision with sum
Summary

The pow() Function

Returns base to the power of exp.
Signature: ‘pow(base, exp, mod=None)‘.

pow() vs ** Operator

For two arguments, ‘pow(a, b)‘ is equivalent to ‘a ** b‘.
However, ‘pow()‘ is a function and can be passed as an argument to higher-order functions like ‘map()‘.

Three-Argument pow()

‘pow(base, exp, mod)‘ computes ‘(base ** exp) % mod‘.
It is FAR more efficient than doing the exponentiation and then modulo, especially for large numbers.
Essential in cryptography (e.g., RSA).

The sum() Function

Sums the items of an iterable from left to right and returns the total.
Signature: ‘sum(iterable, start=0)‘.

The start Parameter in sum()

The ‘start‘ value is added to the total of the items.
Useful if you want to start counting from a specific baseline.

sum() with Non-Numbers

‘sum()‘ is optimized for numbers. Using it to concatenate strings will raise a TypeError.
For strings, use ‘”.join(strings)‘ instead.
You CAN use it to flatten a list of lists, but ‘itertools.chain‘ is better.

Precision Issues with sum()

When summing many floating-point numbers, precision can be lost.
For exact floating-point summation, use ‘math.fsum()‘ instead of built-in ‘sum()‘.

Performance Characteristics

Both ‘pow()‘ and ‘sum()‘ are implemented in C and are highly optimized. Always prefer ‘sum(list)‘ over a manual ‘for‘ loop accumulating a total.

Summary

‘pow()‘ with three arguments is crucial for modular arithmetic.
‘sum()‘ is efficient but watch out for float precision and don’t use it for strings.

Next Lecture

In the final lecture of this unit, we will explore external Modules: math, random, and statistics.

Lecture 31: Essential Python Modules

Importing and utilizing standard library modules.
math, random, and statistics.

Agenda

What is a Module?
Importing Modules
The math Module
The random Module
Generating Random Data
The statistics Module
Summary

What is a Module?

A module is a file containing Python definitions and statements.
Modules allow you to organize code logically.
The Python Standard Library comes with dozens of useful modules.

Importing Modules

Use the ‘import‘ keyword to bring a module into your script.
You can import specific functions using ‘from ... import ...‘.

The math Module

Provides access to mathematical functions defined by the C standard.
Constants: ‘math.pi‘, ‘math.e‘.
Functions: ‘math.sqrt()‘, ‘math.ceil()‘, ‘math.floor()‘, trigonometry.

The random Module

Implements pseudo-random number generators for various distributions. Warning: Not suitable for cryptographic purposes (use ‘secrets‘ module instead).

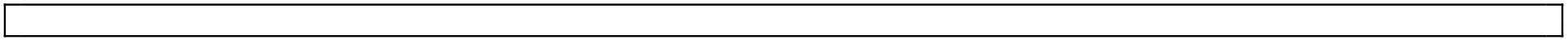
Common random Functions

‘randint(a, b)‘: Random integer N such that a ¡= N ¡= b.
‘choice(seq)‘: Random element from a non-empty sequence.
‘shuffle(seq)‘: Shuffle sequence in place.

The statistics Module

Provides functions for calculating mathematical statistics of numeric data.
Introduced in Python 3.4.
Functions include mean, median, mode, variance, and standard deviation.

Using statistics



Advanced statistics

‘stdev()‘ computes the sample standard deviation.
‘variance()‘ computes the sample variance.

Summary

Modules extend Python's core functionality.
'math' for complex math, 'random' for unpredictable output, 'statistics' for data analysis.

Next Unit

This concludes Unit 4 on Functions. In Unit 5, we will explore Strings and Data Structures.

Introduction to String, String Operations

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 32: Introduction to String, String Operations

Agenda

- String definition
- Immutability
- Concatenation
- Repetition
- Membership

Deep Dive: String definition

Understanding the core mechanics of String definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Immutability

Understanding the core mechanics of Immutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Concatenation

Understanding the core mechanics of Concatenation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Repetition

Understanding the core mechanics of Repetition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Membership

Understanding the core mechanics of Membership.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: String definition

Understanding the core mechanics of String definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Immutability

Understanding the core mechanics of Immutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Concatenation

Understanding the core mechanics of Concatenation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Traversing a String
- Preparation reading recommended.

Traversing a String

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 33: Traversing a String

Agenda

- Indexing
- Slicing
- For loops
- While loops
- Enumerate with strings

Deep Dive: Indexing

Understanding the core mechanics of Indexing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Slicing

Understanding the core mechanics of Slicing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: For loops

Understanding the core mechanics of For loops.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: While loops

Understanding the core mechanics of While loops.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Enumerate with strings

Understanding the core mechanics of Enumerate with strings.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Indexing

Understanding the core mechanics of Indexing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Slicing

Understanding the core mechanics of Slicing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: For loops

Understanding the core mechanics of For loops.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: String Methods and Built-in Functions
- Preparation reading recommended.

String Methods and Built-in Functions

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 34: String Methods and Built-in Functions

Agenda

- Case conversion
- Find and Replace
- Split and Join
- String Formatting

Deep Dive: Case conversion

Understanding the core mechanics of Case conversion.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Find and Replace

Understanding the core mechanics of Find and Replace.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Split and Join

Understanding the core mechanics of Split and Join.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: String Formatting

Understanding the core mechanics of String Formatting.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Case conversion

Understanding the core mechanics of Case conversion.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Find and Replace

Understanding the core mechanics of Find and Replace.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Split and Join

Understanding the core mechanics of Split and Join.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: String Formatting

Understanding the core mechanics of String Formatting.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Introduction to List and its Operations
- Preparation reading recommended.

Introduction to List and its Operations

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 35: Introduction to List and its Operations

Agenda

- List definition
- Mutability
- List Memory Mapping
- Basic Operations

Deep Dive: List definition

Understanding the core mechanics of List definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Mutability

Understanding the core mechanics of Mutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: List Memory Mapping

Understanding the core mechanics of List Memory Mapping.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Basic Operations

Understanding the core mechanics of Basic Operations.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: List definition

Understanding the core mechanics of List definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Mutability

Understanding the core mechanics of Mutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: List Memory Mapping

Understanding the core mechanics of List Memory Mapping.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Basic Operations

Understanding the core mechanics of Basic Operations.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: List Methods and Built-in Functions
- Preparation reading recommended.

List Methods and Built-in Functions

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 36: List Methods and Built-in Functions

Agenda

- append vs extend
- insert, pop, remove
- sort, reverse
- List comprehensions intro

Deep Dive: append vs extend

Understanding the core mechanics of append vs extend.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: insert, pop, remove

Understanding the core mechanics of insert, pop, remove.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: sort, reverse

Understanding the core mechanics of sort, reverse.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: List comprehensions intro

Understanding the core mechanics of List comprehensions intro.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: append vs extend

Understanding the core mechanics of append vs extend.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: insert, pop, remove

Understanding the core mechanics of insert, pop, remove.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: sort, reverse

Understanding the core mechanics of sort, reverse.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: List comprehensions intro

Understanding the core mechanics of List comprehensions intro.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Set Built-in Functions: Create a Set, Accessing Python Sets - Preparation reading recommended.

Set Built-in Functions: Create a Set, Accessing Python Sets

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 37: Set Built-in Functions: Create a Set, Accessing Python Sets

Agenda

- Set definition
- Hashability
- Empty set creation
- Accessing elements

Deep Dive: Set definition

Understanding the core mechanics of Set definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Hashability

Understanding the core mechanics of Hashability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Empty set creation

Understanding the core mechanics of Empty set creation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Accessing elements

Understanding the core mechanics of Accessing elements.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Set definition

Understanding the core mechanics of Set definition.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Hashability

Understanding the core mechanics of Hashability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Empty set creation

Understanding the core mechanics of Empty set creation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Accessing elements

Understanding the core mechanics of Accessing elements.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Delete from set, Update set
- Preparation reading recommended.

Delete from set, Update set

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 38: Delete from set, Update set

Agenda

- add vs update
- remove vs discard
- pop and clear
- Memory implications

Deep Dive: add vs update

Understanding the core mechanics of add vs update.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: remove vs discard

Understanding the core mechanics of remove vs discard.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: pop and clear

Understanding the core mechanics of pop and clear.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Memory implications

Understanding the core mechanics of Memory implications.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: add vs update

Understanding the core mechanics of add vs update.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: remove vs discard

Understanding the core mechanics of remove vs discard.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: pop and clear

Understanding the core mechanics of pop and clear.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Memory implications

Understanding the core mechanics of Memory implications.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Python Set Operations
- Preparation reading recommended.

Python Set Operations

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 39: Python Set Operations

Agenda

- Union (—)
- Intersection (&)
- Difference (-)
- Symmetric Difference (^)

Deep Dive: Union (—)

Understanding the core mechanics of Union (—).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Intersection (&)

Understanding the core mechanics of Intersection (&).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Difference (-)

Understanding the core mechanics of Difference (-).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Symmetric Difference (^)

Understanding the core mechanics of Symmetric Difference (^).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Union (—)

Understanding the core mechanics of Union (—).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Intersection (&)

Understanding the core mechanics of Intersection (&).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Difference (-)

Understanding the core mechanics of Difference (-).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Symmetric Difference (^)

Understanding the core mechanics of Symmetric Difference (^).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Tuple Built-in Functions: Creating Tuples, Accessing Tuple - Preparation reading recommended.

Tuple Built-in Functions: Creating Tuples, Accessing Tuple

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 40: Tuple Built-in Functions: Creating Tuples, Accessing Tuple

Agenda

- Tuple immutability
- Single element tuple
- Tuple packing/unpacking
- Memory efficiency

Deep Dive: Tuple immutability

Understanding the core mechanics of Tuple immutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Single element tuple

Understanding the core mechanics of Single element tuple.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Tuple packing/unpacking

Understanding the core mechanics of Tuple packing/unpacking.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Memory efficiency

Understanding the core mechanics of Memory efficiency.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Tuple immutability

Understanding the core mechanics of Tuple immutability.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Single element tuple

Understanding the core mechanics of Single element tuple.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Tuple packing/unpacking

Understanding the core mechanics of Tuple packing/unpacking.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Memory efficiency

Understanding the core mechanics of Memory efficiency.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Iterate over tuple and Slicing tuple
- Preparation reading recommended.

Iterate over tuple and Slicing tuple

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 41: Iterate over tuple and Slicing tuple

Agenda

- Iteration performance
- Advanced slicing
- Negative indexing
- Immutability consequences

Deep Dive: Iteration performance

Understanding the core mechanics of Iteration performance.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Advanced slicing

Understanding the core mechanics of Advanced slicing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Negative indexing

Understanding the core mechanics of Negative indexing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Immutability consequences

Understanding the core mechanics of Immutability consequences.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Iteration performance

Understanding the core mechanics of Iteration performance.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Advanced slicing

Understanding the core mechanics of Advanced slicing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Negative indexing

Understanding the core mechanics of Negative indexing.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Immutability consequences

Understanding the core mechanics of Immutability consequences.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Python Tuple Operations, Functions and Methods
- Preparation reading recommended.

Python Tuple Operations, Functions and Methods

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 42: Python Tuple Operations, Functions and Methods

Agenda

- count and index
- Concatenation
- Functions as tuple returners
- Tuple comprehensions (generators)

Deep Dive: count and index

Understanding the core mechanics of count and index.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Concatenation

Understanding the core mechanics of Concatenation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Functions as tuple returners

Understanding the core mechanics of Functions as tuple returners.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Tuple comprehensions (generators)

Understanding the core mechanics of Tuple comprehensions (generators).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: count and index

Understanding the core mechanics of count and index.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Concatenation

Understanding the core mechanics of Concatenation.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Functions as tuple returners

Understanding the core mechanics of Functions as tuple returners.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Tuple comprehensions (generators)

Understanding the core mechanics of Tuple comprehensions (generators).
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Dictionary Built-in Functions: Creating Dictionary, Accessing Items

- Preparation reading recommended.

Dictionary Built-in Functions: Creating Dictionary, Accessing Items

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 43: Dictionary Built-in Functions: Creating Dictionary, Accessing Items

Agenda

- Key-Value mapping
- Hashable keys
- O(1) access time
- Creation methods

Deep Dive: Key-Value mapping

Understanding the core mechanics of Key-Value mapping.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Hashable keys

Understanding the core mechanics of Hashable keys.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: O(1) access time

Understanding the core mechanics of O(1) access time.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Creation methods

Understanding the core mechanics of Creation methods.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Key-Value mapping

Understanding the core mechanics of Key-Value mapping.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Hashable keys

Understanding the core mechanics of Hashable keys.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: O(1) access time

Understanding the core mechanics of O(1) access time.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Creation methods

Understanding the core mechanics of Creation methods.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Add, Update, Remove in Dictionary
- Preparation reading recommended.

Add, Update, Remove in Dictionary

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 44: Add, Update, Remove in Dictionary

Agenda

- Adding items
- update method
- pop and popitem
- del keyword

Deep Dive: Adding items

Understanding the core mechanics of Adding items.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: update method

Understanding the core mechanics of update method.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: pop and popitem

Understanding the core mechanics of pop and popitem.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: del keyword

Understanding the core mechanics of del keyword.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Adding items

Understanding the core mechanics of Adding items.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: update method

Understanding the core mechanics of update method.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: pop and popitem

Understanding the core mechanics of pop and popitem.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: del keyword

Understanding the core mechanics of del keyword.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: Built-In Dictionary Methods and functions
- Preparation reading recommended.

Built-In Dictionary Methods and functions

Unit 5: Dictionary, List, Set, String and Tuple
Lecture 45: Built-In Dictionary Methods and functions

Agenda

- keys(), values(), items()
- get() vs []
- setdefault()
- Dictionary comprehensions

Deep Dive: keys(), values(), items()

Understanding the core mechanics of keys(), values(), items().
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: get() vs []

Understanding the core mechanics of get() vs [].
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: setdefault()

Understanding the core mechanics of setdefault().
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Dictionary comprehensions

Understanding the core mechanics of Dictionary comprehensions.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: keys(), values(), items()

Understanding the core mechanics of keys(), values(), items().
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: get() vs []

Understanding the core mechanics of get() vs [].
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: setdefault()

Understanding the core mechanics of setdefault().
Memory complexity and time complexity considerations.
Common edge cases in Python.

Deep Dive: Dictionary comprehensions

Understanding the core mechanics of Dictionary comprehensions.
Memory complexity and time complexity considerations.
Common edge cases in Python.

Summary

- Review of key concepts.
- Best practices and Pythonic idioms.
- Common pitfalls avoided.

Next Lecture Preview

Next up: End of Unit
- Preparation reading recommended.