day 2 - 13/08

Name: Samiksha Kulkarni

Year and Branch: final year (Information Technology)

Topic:

- List Comprehension
- Dict Comprehension
- File Handling
- Error Handling

✅ List Comprehension

What Is List Comprehension in Python?

List comprehension is a **concise way to create lists** in Python by applying an expression to each element (item) in an iterable (like a list, tuple, or range), optionally filtering elements based on a condition. It lets you write more readable and compact code compared to traditional loops.

Basic Syntax

`[expression for item in iterable if condition]`

- **expression:** What you want to put in the new list.
- **item:** The variable representing each element in the iterable.
- **iterable:** A sequence (list, range, etc.) to iterate over.
- **condition (optional):** If included, filters elements.

1. Basic Transformation

Double each number:

```
numbers = [1, 2, 3, 4]
doubled = [num * 2 for num in numbers]
print(doubled)  # Output: [2, 4, 6, 8]
```

2. Filtering

Select only even numbers:

```
select_even = [num for num in range(1, 20) if num % 2 ==0]
print(select_even)
#output: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

3. Using If-Else

Label numbers as "Even" or "Odd":

```
numbers = [1, 2, 3, 4, 5]
even_odd = ["even" if num%2==0 else "odd" for num in numbers]
print(even_odd)
#output: ['odd', 'even', 'odd', 'even', 'odd']
```

4. Nested List Comprehension

```
# Create a multiplication table

table = [[i * j for j in range(1, 6)] for i in range(2, 5)]
print(table)

# Output: [[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]]
```

5. Flattening Nested Lists

Flattening nested lists means converting a multi-level (nested) list structure—where some or all elements might be lists themselves—into a single, one-dimensional list containing all the individual elements, with no sub-lists

```
merge = [[1,2,3], [4,5,6], [7,8,9]]
flat = [num for sublist in merge for num in sublist]
print(flat)

# Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Advantages

- **More concise:** Reduces lines of code versus traditional loops.
- **Readability:** Compact, easy-to-understand structure for simple operations.
- **Performance:** Often faster than looping with append.

For Loop vs. List Comprehension

The main difference is that a for loop requires multiple lines to create a new list by iterating over items and manually adding each one. Whereas, list comprehension do the same task in a single line, this makes the code simpler and easier to read.

- **For Loop:** Requires multiple lines to create and populate a list.

```
a = [1, 2, 3, 4, 5]
res = []
for val in a:
    res.append(val * 2)
print(res)

# Output: [2, 4, 6, 8, 10]
```

- **List Comprehension:** Achieves the same result in a single, concise line.

```
a = [1, 2, 3, 4, 5]
res = [val * 2 for val in a]
print(res)

# Output: [2, 4, 6, 8, 10]
```

Aspect	For Loop	List Comprehension
Syntax	Requires multiple lines to create and populate a list.	Achieves the same result in a single, concise line.
Readability	Clear and explicit, good for beginners	More compact, best for simple operations
Performance	Slower due to method calls and interpretive overhead	Often faster due to internal optimizations
Use Case	Complex loops with multiple operations	Creating/modifying lists in a straightforward way
Complexity	Handles complex logic easily	Not ideal for very complex or nested operations

✅ Dict Comprehension

What is Dictionary Comprehension in Python?

Dictionary comprehension provides a concise way to create dictionaries by specifying **key–value pairs** in a single expression, often based on an iterable or existing data.

Syntax

`{key_expression: value_expression for item in iterable if condition}`

- **key_expression** : the value to use as the dictionary key.
- **value_expression** : the value to use as the dictionary value.
- **iterable** : a sequence or collection to loop through.
- **condition** *(optional)* : filter which elements are included.

Examples

1. Basic Transformation

Create a dictionary mapping numbers to their squares:

```
numbers = [1, 2, 3, 4]
squares = {n: n**2 for n in numbers}
print(squares)

# Output: {1: 1, 2: 4, 3: 9, 4: 16}
```

---

2. Filtering in Comprehension

Keep only even numbers:

```
even_squares = {n: n**2 for n in range(1, 10) if n % 2 == 0}
print(even_squares)

# Output: {2: 4, 4: 16, 6: 36, 8: 64}
```

---

3. Using Functions in Comprehension

Convert words to their lengths:

```
words = ["apple", "banana", "cherry"]
word_lengths = {word: len(word) for word in words}
print(word_lengths)

# Output: {'apple': 5, 'banana': 6, 'cherry': 6}
```

---

4. Swap Keys and Values

```
data = {'a': 1, 'b': 2, 'c': 3}
swapped = {v: k for k, v in data.items()}
print(swapped)

# Output: {1: 'a', 2: 'b', 3: 'c'}
```

---

5. Nested Loop in Dict Comprehension

Creating a multiplication dictionary:

```
table = {(x, y): x * y for x in range(1, 4) for y in range(1, 4)}
print(table)

# Output: {(1,1):1, (1,2):2, ... , (3,3):9}
```

Advantages of Dict Comprehension:

- **Concise** → No need for long loops and `dict.update()` calls.
- **Readable** → Ideal for small-to-medium transformations.
- **Fast** → Often performs better than manual looping.

✅ File Handling

File handling in Python means **reading from and writing to files** stored on your computer's disk.

It's useful for:

- Saving data permanently (unlike variables which vanish when the program ends)
- Storing logs, configurations, reports, etc.
- working with large datasets

Types of Files

- **Text files** (`.txt` , `.csv` , `.log` , `.json`) → Store data as human-readable text.
- **Binary files** (`.jpg` , `.png` , `.exe`) → Store data in binary format (machine-readable).

File Operations

To work with files in Python, we typically:

1. **Open the file**
2. **Read or write**
3. **Close the file**

Opening Files

The `open()` function is used to open files:

```
file = open('filename.txt', 'mode')
```

Common modes:

- `'r'` - Read (default)
- `'w'` - Write (overwrites existing content)
- `'a'` - Append
- `'r+'` - Read and write
- `'b'` - Binary mode (e.g., `'rb'` , `'wb'`)

The `with` Statement (Recommended)

Always use `with` for automatic file closing:

```
with open('file.txt', 'r') as file:
    content = file.read()
# File is automatically closed here
```

Reading Files

Read entire file:

```
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)
```

Read line by line:

```
with open('data.txt', 'r') as file:
    for line in file:
        print(line.strip())  # strip() removes newline characters
```

Read all lines into a list:

```
with open('data.txt', 'r') as file:
    lines = file.readlines()
```

Read one line at a time:

```
with open('data.txt', 'r') as file:
    first_line = file.readline()
    second_line = file.readline()
```

Writing Files

Write to a file (overwrites existing content):

```
with open('output.txt', 'w') as file:
    file.write('Hello, World!')
    file.write('\nSecond line')
```

Write multiple lines:

```
lines = ['Line 1\n', 'Line 2\n', 'Line 3\n']
with open('output.txt', 'w') as file:
    file.writelines(lines)
```

Append to a file:

```
with open('log.txt', 'a') as file:
    file.write('\nNew log entry')
```

Working with CSV Files

```
import csv

# Reading CSV
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)

# Writing CSV
data = [['Name', 'Age'], ['Alice', 25], ['Bob', 30]]
with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

Working with JSON Files

```
import json

# Reading JSON
with open('data.json', 'r') as file:
    data = json.load(file)

# Writing JSON
data = {'name': 'Alice', 'age': 25}
with open('output.json', 'w') as file:
    json.dump(data, file, indent=2)
```

Error Handling

Always handle potential file errors:

```
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
except Exception as e:
    print(f"An error occurred: {e}")
```

File Operations

Check if file exists:

```
import os
if os.path.exists('file.txt'):
    print("File exists")
```

Get file information:

```
import os
file_stats = os.stat('file.txt')
print(f"Size: {file_stats.st_size} bytes")
print(f"Modified: {file_stats.st_mtime}")
```

Delete a file:

```
import os
os.remove('file.txt')
```

Best Practices

1. Always use `with` statements for automatic file closing
2. Handle exceptions appropriately
3. Specify encoding when working with text files: `open('file.txt', 'r', encoding='utf-8')`
4. Use appropriate file modes
5. Close files explicitly if not using `with`
6. Be careful with file paths (use `os.path.join()` for cross-platform compatibility)

File handling is essential for data persistence, configuration management, logging, and processing external data in Python applications.

✅Error Handling

Error handling in Python allows you to gracefully manage and respond to runtime errors, preventing your program from crashing unexpectedly.

Basic Try-Except Structure

```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Common Exception Types

Built-in exceptions you'll encounter:

- `ValueError` - Invalid value for operation
- `TypeError` - Wrong data type
- `IndexError` - List index out of range
- `KeyError` - Dictionary key doesn't exist
- `FileNotFoundError` - File doesn't exist
- `AttributeError` - Object doesn't have attribute
- `ZeroDivisionError` - Division by zero

Multiple Exception Handling

Handle different exceptions separately:

```
try:
    user_input = input("Enter a number: ")
    number = int(user_input)
    result = 10 / number
    print(f"Result: {result}")
except ValueError:
    print("Please enter a valid number!")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Handle multiple exceptions the same way:

```
try:
    # Some risky code
    pass
except (ValueError, TypeError, KeyError):
    print("One of several errors occurred!")
```

Catching All Exceptions

```
try:
    # Risky code
    pass
except Exception as e:
    print(f"An error occurred: {e}")
    print(f"Error type: {type(e).__name__}")
```

The Else Clause

Runs only if no exceptions occur:

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Invalid input!")
else:
    print(f"You entered: {number}")
    # This runs only if no exception occurred
```

The Finally Clause

Always executes, regardless of exceptions:

```
try:
    file = open('data.txt', 'r')
    data = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    # This always runs
    if 'file' in locals():
        file.close()
    print("Cleanup completed")
```

Raising Custom Exceptions

Raise built-in exceptions:

```
def validate_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    if age > 150:
        raise ValueError("Age seems unrealistic!")
    return age

try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation error: {e}")
```

Create custom exception classes:

```
class CustomError(Exception):
    """Custom exception for specific use cases"""
    pass

class ValidationError(Exception):
    def __init__(self, message, error_code=None):
        super().__init__(message)
        self.error_code = error_code

# Usage
def process_data(data):
    if not data:
        raise ValidationError("Data cannot be empty", error_code=400)

try:
```

```
    process_data("")
except ValidationError as e:
    print(f"Error: {e}")
    print(f"Error code: {e.error_code}")
```

Practical Examples

Safe dictionary access:

```
user_data = {'name': 'Alice', 'age': 25}

try:
    email = user_data['email']
except KeyError:
    email = 'Not provided'

# Alternative using get() method
email = user_data.get('email', 'Not provided')
```

Safe file operations:

```
def read_config_file(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
        print(f"Config file {filename} not found. Using defaults.")
        return "{}"
    except PermissionError:
        print(f"Permission denied reading {filename}")
        return None
    except Exception as e:
        print(f"Unexpected error reading config: {e}")
        return None
```

Input validation:

```
def get_positive_integer():
    while True:
        try:
            value = int(input("Enter a positive integer: "))
            if value <= 0:
                raise ValueError("Number must be positive")
            return value
        except ValueError as e:
            if "invalid literal" in str(e):
                print("Please enter a valid integer!")
            else:
                print(f"Error: {e}")
```

Exception Chaining

Track the original cause of an exception:

```
def divide_numbers(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        raise ValueError("Invalid division operation") from e
```

```
try:
    result = divide_numbers(10, 0)
except ValueError as e:
    print(f"Error: {e}")
    print(f"Original cause: {e.__cause__}")
```

Best Practices

1. **Be specific** - Catch specific exceptions rather than using bare `except:`
2. **Don't ignore exceptions** - Always handle them appropriately
3. **Use finally for cleanup** - Ensure resources are properly released
4. **Log exceptions** - Keep track of errors for debugging
5. **Fail fast** - Don't let invalid states propagate
6. **Use custom exceptions** - Create meaningful exception types for your domain

Error handling is crucial for building robust applications that can gracefully handle unexpected situations and provide meaningful feedback to users.

✅ Industry Coding Standards

Coding standards and best practices are a set of guidelines and conventions followed in software development to ensure code quality, readability, maintainability, and consistency across a project or team.

1. **Core Principles:**

- **Readability** - Code should be self-documenting and easily understood by other developers
- **Consistency** - Follow uniform naming conventions, formatting, and structure
- **Maintainability** - Write code that can be easily modified and extended
- **Performance** - Optimize for efficiency without sacrificing clarity
- **Security** - Follow secure coding practices to prevent vulnerabilities

2. **Naming Conventions:**

   Consistent use of naming conventions for variables, functions, classes, and files (e.g., camelCase, snake_case, PascalCase) to improve readability and understanding.

3. **Code Formatting:**

   Adherence to consistent indentation, spacing, line length, and brace placement to enhance visual clarity and make code easier to navigate.

4. **Commenting and Documentation:**

   Strategic use of comments to explain complex logic, design decisions, and non-obvious code sections, along with comprehensive documentation (e.g., README files, docstrings) for modules and APIs.

5. **Code Structure and Organization:**

   Breaking down complex problems into smaller, manageable functions or modules, avoiding deep nesting, and organizing files and folders logically.

6. **Error Handling:**

   Implementing robust error handling mechanisms to gracefully manage exceptions and prevent crashes.

7. **Code Reusability:**

   Applying principles like DRY (Don't Repeat Yourself) by creating reusable functions, classes, or modules to avoid duplication and promote efficiency.

8. **Version Control Best Practices:**

   Utilizing version control systems (e.g., Git) effectively with meaningful commit messages, appropriate branching strategies, and collaborative workflows.

9. **Performance Optimization:**

   Considering performance implications and optimizing code where necessary, without compromising readability or maintainability.

10. **Security Considerations:**

    Following secure coding practices to prevent vulnerabilities and protect against common security threats.