**Module 9: Data Structures**

In Python, **list comprehensions** provide a concise and elegant way to create new lists by applying expressions to each element in an iterable (like a list, tuple, string, or range).

1. **Basics of List Comprehension**

**Syntax:**

[expression for item in iterable if condition]

* **expression**: The value to store in the new list (can include transformations).
* **item**: The variable that represents each element in the iterable.
* **iterable**: Any iterable (list, string, range, etc.).
* **condition** (optional): A filter to include elements that satisfy the condition.

**Example:**

# Create a list of squares from 1 to 5

squares = [x\*\*2 for x in range(1, 6)]

print(squares) # [1, 4, 9, 16, 25]

1. **List Comprehension with Conditions**

You can add an if condition to filter elements.

Example:

# Even numbers between 1 and 10

evens = [x for x in range(1, 11) if x % 2 == 0]

print(evens) # [2, 4, 6, 8, 10]

Example with else:

# Mark numbers as even or odd

labels = ["Even" if x % 2 == 0 else "Odd" for x in range(1, 6)]

print(labels) # ['Odd', 'Even', 'Odd', 'Even', 'Odd']

1. **Nested List Comprehension**

You can nest list comprehensions (useful for 2D lists/matrices).

Example:

# Flatten a 2D matrix

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

flattened = [num for row in matrix for num in row]

print(flattened) # [1, 2, 3, 4, 5, 6, 7, 8, 9]

1. **Using Functions in List Comprehension**

def square(x):

return x \* x

result = [square(x) for x in range(1, 6)]

print(result) # [1, 4, 9, 16, 25]

1. Practical Examples

Extract vowels from a string:

text = "python"

vowels = [ch for ch in text if ch in "aeiou"]

print(vowels) # ['o']

Convert Celsius to Fahrenheit:

celsius = [0, 10, 20, 30]

fahrenheit = [(c \* 9/5) + 32 for c in celsius]

print(fahrenheit) # [32.0, 50.0, 68.0, 86.0]

1. **Data Structures with Comprehensions**

Python also supports dictionary comprehensions, set comprehensions, and generator expressions.

Dictionary Comprehension:

squares\_dict = {x: x\*\*2 for x in range(1, 6)}

print(squares\_dict) # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

Set Comprehension:

unique\_squares = {x\*\*2 for x in [1, 2, 2, 3]}

print(unique\_squares) # {1, 4, 9}

**Advantages of List Comprehension**

More readable and concise than using loops.

Faster than for loops for creating lists (since it’s optimized in C).

1. **Default Parameters**

In Python, default parameters are used in function definitions to provide default values for function arguments. When working with data structures like lists, dictionaries, sets, etc., it's important to use default parameters carefully—especially with mutable objects (like lists and dictionaries).

1. Using Default Parameters with Immutable Types (Safe)

Immutable types like int, str, or tuple are safe to use as default parameters.

def greet(name="Guest"):

print(f"Hello, {name}!")

greet() # Output: Hello, Guest!

greet("Amit") # Output: Hello, Amit!

2. Avoid Mutable Defaults (Common Pitfall)

If you use a mutable object like a list or dictionary as a default parameter, Python uses the same object across function calls. This can lead to unexpected behavior.

def add\_item(item, item\_list=[]): # DANGEROUS

item\_list.append(item)

return item\_list

print(add\_item("Apple")) # ['Apple']

print(add\_item("Banana")) # ['Apple', 'Banana'] — Not expected

1. **Variable Arguments**

Variable Arguments allow functions to accept an arbitrary number of arguments, and they can be very useful when working with data structures like lists, dictionaries, sets, etc.

There are two types of variable arguments:

**1. \*args (Non-keyword variable arguments)**

* Collects **extra positional arguments** as a **tuple**.
* Commonly used when you don’t know how many arguments will be passed.

**Example using \*args with a list:**

def add\_to\_list(\*items):

result = list(items)

return result

print(add\_to\_list(10, 20, 30))

# Output: [10, 20, 30]

2. **\*\*kwargs (Keyword variable arguments)**

* **Collects extra keyword arguments** as a **dictionary.**
* Useful when you want to pass named arguments and don’t know their names ahead of time.

Example using \*\*kwargs with a dictionary:

def create\_student\_record(\*\*info):

return info

print(create\_student\_record(name='Amit', age=30, grade='A'))

# Output: {'name': 'Amit', 'age': 30, 'grade': 'A'}

**Combined Example: Using Both**

def data\_structure\_example(\*args, \*\*kwargs):

print("Tuple from \*args:", args)

print("Dict from \*\*kwargs:", kwargs)

data\_structure\_example(1, 2, 3, name='Amit', role='Trainer')

# Output:

# Tuple from \*args: (1, 2, 3)

# Dict from \*\*kwargs: {'name': 'Amit', 'role': 'Trainer'}

**When Used in Data Structures:**

* You can use \*args to build lists or tuples dynamically.
* You can use \*\*kwargs to populate dictionaries.
* Useful in classes, data processing functions, or configurations.

1. **Specialized Sorts**

In Python Data Structures, specialized sorts refer to sorting techniques or algorithms tailored for specific scenarios or data structures. These are not just general-purpose sorting methods like .sort() or sorted(), but are optimized or specialized for performance, data shape, or constraints.

**1. Key-based Sorting (key= parameter)**

Used with sorted() or .sort() for custom sorting logic.

data = ['apple', 'banana', 'cherry', 'date']

# Sort by length of each word

sorted\_data = sorted(data, key=len)

print(sorted\_data)

**2. Stable Sorts**

Python’s built-in sorted() and .sort() use **Timsort** (a hybrid of merge sort and insertion sort), which is **stable**.

students = [('John', 85), ('Jane', 90), ('Dave', 85), ('Ram', 90), ('shyam', 85)]

# Stable sort preserves order of equal elements

print(sorted(students, key=lambda x: x[1]))

**3. Sorting Tuples by Multiple Keys**

data = [('Alice', 25), ('Bob', 30), ('Alice', 22), ('pooja', 30), ('manjarik', 36)]

# Sort by name, then by age

print(sorted(data, key=lambda x: (x[0], x[1])))

**4. Heap Sort using heapq (min-heap)**

Efficient for priority queue behavior and partial sorting.

import heapq

nums = [5, 1, 9, 3, 7]

heapq.heapify(nums)

print(heapq.heappop(nums)) # Smallest item

**5. Custom Comparator with functools.cmp\_to\_key()**

Used when sorting needs comparison between pairs.

from functools import cmp\_to\_key

def custom\_cmp(a, b):

return len(a) - len(b)

words = ['banana', 'fig', 'apple']

sorted\_words = sorted(words, key=cmp\_to\_key(custom\_cmp))

print(sorted\_words)

**6. Counting Sort (when elements are integers in limited range)**

Good for small-range integers.

def counting\_sort(arr):

count = [0] \* (max(arr) + 1)

for num in arr:

count[num] += 1

sorted\_arr = []

for i, c in enumerate(count):

sorted\_arr.extend([i] \* c)

return sorted\_arr

print(counting\_sort([4, 2, 2, 8, 3, 3, 1]))

**7. Sorting Dictionaries**

By keys or values.

d = {'a': 3, 'b': 1, 'c': 2}

print(sorted(d.items(), key=lambda x: x[1])) # Sort by value

**8. Sort with operator module**

More efficient and readable than lambda in some cases.

from operator import itemgetter

records = [('Alice', 25), ('Bob', 22), ('Dave', 30)]

print(sorted(records, key=itemgetter(1)))