1. Introduction

- Comprehensions are a **stylized, concise way of writing Python code**.
- Everything achievable with comprehensions can also be done with loops.
- They make code **shorter, cleaner, and sometimes easier to read**.
- Widely used in **production code**.

---

2. Why Use Comprehensions?

- Shorter, more **English-like one-liners**.
- Cleaner and more elegant code.
- Often **faster execution** and uses **less memory**.
- Functional programming style.

---

3. Challenges

- Many beginners find comprehensions confusing at first.
- Not always the easiest to understand, but with practice they become natural.

---

4. What Are Comprehensions?

- A **concise way** to create:
    - **Lists**
    - **Sets**

  - **Dictionaries**
  - **Generators**

- Written in **one line of code** (instead of multi-line loops).

---

5. Common Uses in Real Life

- **Filtering items**
  Example: Picking only "hot teas" from a menu.

- **Transforming items**
  Example: Converting prices from INR to USD.

- **Creating new collections**
  Example: Mapping tea names to prices.

- **Flattening nested structures**
  Example: Extracting ingredients from a nested recipe dictionary.

---

6. Benefits / Purpose

- Cleaner code (though not always the easiest).

- Faster execution in many cases.

- Uses less memory.

- Encourages functional programming style.

---

7. Types of Comprehensions

1. **List comprehension**

2. **Set comprehension**

3. **Dictionary comprehension**

4. **Generator comprehension**

(Generators are not a data type but a structure that will be studied later.)

---

8. Learning Plan

- Start with **list comprehensions**.
- Study theory, then practice with code.
- Gain confidence by solving small, practical examples.

✅ **Key Takeaway**:
Comprehensions = **Concise, clean, and powerful way** to build collections in Python. Essential for writing production-ready code.

📘 Python List Comprehensions (Chapter 1)

🔹 What is a List Comprehension?

- A concise way to create lists in Python.

Syntax:

```
[expression for item in iterable if condition]
```

-
- Behind the scenes, it still uses a loop, but in a cleaner, one-liner format.

🔹 Syntax Breakdown

1. **Square brackets** `[ ]` → defines a list.
2. **Expression** → the value you want to store.
3. **For loop** → iterates over an iterable.
4. **If condition (optional)** → filters the values.

Example pattern:

```
[expression for item in iterable if condition]
```

🔹 Example – Menu Filtering

```
menu = [
    "masala chai",
    "iced lemon tea",
    "green tea",
    "iced peach tea",
    "ginger tea"
]

# Extract only "iced" teas
iced_teas = [tea for tea in menu if "iced" in tea]
print(iced_teas)
```

✅ Output:

```
['iced lemon tea', 'iced peach tea']
```

🔹 Variable Naming

- The variable in comprehension (`tea` in the example) comes directly from the loop.
- If you rename it, you must update it consistently in both **expression** and **condition**.

🔹 Adding Conditions

You can apply different conditions, like filtering based on string length:

```
# Teas with name length < 12
short_teas = [tea for tea in menu if len(tea) < 12]
print(short_teas)
```

✅ Output:

```
['iced tea', 'lemon tea']
```

🔹 Key Takeaways

- **Expression** = what goes inside the list.
- **Item** = each element of the iterable.
- **Iterable** = the source collection (list, tuple, string, etc.).
- **Condition** = optional filter.

👉 In short: List comprehensions = **Loop + Condition + Expression** in one neat line.

📝 Python Comprehensions – Set Comprehensions

🔹 Recap

- You already know **list comprehensions**.
- Now, moving to **set comprehensions**.

🔹 Set Comprehension Syntax

```
{ expression for item in iterable if condition }
```

- Same as list comprehension, but with `{ }` instead of `[ ]`.
- Automatically stores **unique values**.

🔹 Example 1: Favorite Teas

```
favorite_chai = [
    "masala chai",
    "green tea",
    "masala chai",
    "lemon tea",
    "green tea",
    "lychee chai"
]

# Unique teas
unique_chai = {chai for chai in favorite_chai}
print(unique_chai)
```

👉 Output:

```
{'masala chai', 'green tea', 'lemon tea', 'lychee chai'}
```

- Notice: Duplicates are removed automatically.

🔹 Adding Conditions

```
# Only teas with name length > 8
unique_long_chai = {chai for chai in favorite_chai if len(chai) > 8}
print(unique_long_chai)
```

👉 Example output:

```
{'masala chai', 'lemon tea', 'lychee chai'}
```

🔹 Example 2: Recipes (Nested Dictionary)

```
recipes = {
    "masala chai": ["ginger", "cardamom", "clove"],
    "elaichi chai": ["cardamom", "milk"],
    "spicy chai": ["ginger", "black pepper", "clove"]
}
```

🎯 Task: Find **all unique spices**.

```
unique_spices = {ingredient
                 for ingredients in recipes.values()
                 for ingredient in ingredients}

print(unique_spices)
```

👉 Output:

```
{'ginger', 'cardamom', 'clove', 'milk', 'black pepper'}
```

🔹 Key Points

1. Use `{}` instead of `[]` → set comprehension.

2. Good for getting **unique values** directly.
3. Can be nested (iterate inside lists/dictionaries).
4. Expression part defines the final stored value.

⚡ This lesson builds intuition for how **expression placement matters** in comprehensions, not just the loop/condition part.

📘 Python Comprehensions – Dictionary Comprehensions

🔹 Introduction

- Just like lists (`[ ]`) and sets (`{}`), **dictionaries** also have a comprehension form.
- Difference: In dictionary comprehensions, the **expression** must return a **key–value pair** (`key: value`).

Syntax:

```
{ key_expression : value_expression for key, value in iterable if
condition }
```

-

🔹 Example – Tea Prices in INR

```
tea_prices_inr = {
    "masala chai": 40,
    "green tea": 50,
    "lemon tea": 200
}
```

Task: Convert all prices to **USD** (divide by 80).

🔹 Using Dictionary Comprehension

```
tea_prices_usd = {tea: price/80 for tea, price in
tea_prices_inr.items()}
print(tea_prices_usd)
```

✅ Output:

```
{'masala chai': 0.5, 'green tea': 0.625, 'lemon tea': 2.5}
```

🔹 Key Points

1. **Curly braces `{}`** → used for both sets and dicts.
   - ○ Dict requires `key: value` pairs.

Use `.items()` to loop through both **keys** and **values**.

```
 for tea, price in tea_prices_inr.items():
```

2.
3. **Expression part** decides what is stored (here: `tea: price/80`).
4. Comprehensions **shrink code** and are **cleaner** than loops.

🔹 Benefits

- Convert / transform data easily.

- Makes code **short, readable, and elegant**.
- Same logic: always **start reading from the `for` loop**, then see what the expression returns.

🔹 Example with Formatted Strings

```
tea_prices_usd = {tea: f"${price/80:.2f}" for tea, price in
tea_prices_inr.items()}
print(tea_prices_usd)
```

✅ Output:

```
{'masala chai': '$0.50', 'green tea': '$0.62', 'lemon tea': '$2.50'}
```

✅ Takeaway

- Dictionary comprehensions = **loop + condition + key: value expression**.
- Great for data transformation (like **converting currency, mapping, filtering**).
- Practice is key to becoming comfortable.

⚡ Python Comprehensions – Generator Comprehensions

🔹 What Are They?

- A **generator comprehension** looks like a list/set/dict comprehension.

- **But** instead of building the whole collection in memory, it **yields one item at a time**.
- Used to **save memory** when working with large datasets.

🔹 Syntax

```
(expression for item in iterable if condition)
```

- Same as list comprehension, but uses **parentheses `()`** instead of `[]`.

🔹 Why Use Them?

- **List comprehension**: builds the **entire list** in memory.
- **Generator comprehension**: returns a **generator object** that produces values **lazily (one by one)**.
- Saves memory → especially useful for **large datasets**.

🔹 Example – Daily Sales

```
daily_sales = [5, 10, 12, 7, 3, 8, 9, 15]

# Generator comprehension: sales above 5
sales_gen = (sale for sale in daily_sales if sale > 5)

print(sales_gen)
# <generator object ...>
```

👉 It doesn’t show the values directly because it streams them.
You must **consume** it with functions like `sum()`, `list()`, `next()`, or a loop.

🔹 Consuming a Generator

```
# Sum of sales > 5
total_sales = sum(sale for sale in daily_sales if sale > 5)
print(total_sales)  # 59
```

✅ Memory efficient → processes values one at a time.
❌ A list comprehension would first create `[10, 12, 7, 8, 9, 15]` in memory.

🔹 Key Differences

Feature	List Comprehension	Generator Comprehension
Syntax	`[ ... ]`	`( ... )`
Stores in memory	Entire list	Only 1 item at a time
Performance	Fast for small data	Efficient for big data
Example output	`[10, 12, 7, 8, 9, 15]`	`<generator object>`

✅ Takeaway

- **Use list comprehensions** when the dataset is small and you need random access.
- **Use generator comprehensions** when handling **large data streams** where memory efficiency matters.
- They are **lazy, memory-friendly, and Pythonic**.

That wraps up **all four comprehension types**:

1. List

2. Set

3. Dictionary

4. Generator