

Comprehensions in Python:

- List Comprehensions
 - Dictionary Comprehensions
 - Set Comprehensions
-

1. Comprehensions in Python

Definition:

Comprehensions provide a concise way to create **lists, dictionaries, or sets** in a single line using loops and conditions.

They are more readable and faster than writing traditional for loops.

2. List Comprehension

Definition:

A list comprehension is a compact way of generating a list using an expression followed by a for loop, optionally with conditions.

Syntax:

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

Example 1: Generate squares of numbers

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

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

Example 2: Filter even numbers

```
evens = [x for x in range(10) if x % 2 == 0]
```

```
print(evens) # Output: [0, 2, 4, 6, 8]
```

3. Dictionary Comprehension

Definition:

Dictionary comprehensions create dictionaries in one line, using key-value pairs inside {}.

Syntax:

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

Example 1: Create a dictionary of squares

```
squares_dict = {x: x**2 for x in range(1, 6)}
```

```
print(squares_dict)
```

```
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Example 2: Filter dictionary by condition

```
num_dict = {x: x**2 for x in range(10) if x % 2 == 0}
```

```
print(num_dict)
```

```
# Output: {0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

4. Set Comprehension

Definition:

Set comprehensions are similar to list comprehensions, but they generate a **set** (unique elements, unordered).

Syntax:

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

Example 1: Generate set of squares

```
squares_set = {x**2 for x in range(1, 6)}
```

```
print(squares_set)
```

```
# Output: {1, 4, 9, 16, 25}
```

Example 2: Remove duplicates

```
nums = [1, 2, 2, 3, 4, 4, 5]
```

```
unique_nums = {x for x in nums}
```

```
print(unique_nums)
```

```
# Output: {1, 2, 3, 4, 5}
```

1. Creating a string and variable assignments

A **string** in Python is a sequence of characters enclosed in single (') or double (") quotes. Strings can also be assigned to variables.

Example:

```
# Using single or double quotes
string1 = 'Hello'
string2 = "World"

# Assigning to a variable
greeting = string1 + " " + string2
print(greeting) # Output: Hello World
```

2. String indexing & slicing

Each character in a string has an **index** starting from 0. Negative indexing starts from the end with -1 as the last character. **Slicing** allows you to extract parts of the string.

Example:

```
text = "Python"

# Indexing
print(text[0]) # Output: P
print(text[-1]) # Output: n

# Slicing [start:stop:step]
print(text[0:4]) # Output: Pyth (from index 0 to 3)
print(text[::-1]) # Output: nohtyP (reverse the string)
```

3. String concatenation & repetition

Concatenation means joining strings using +.

Repetition means repeating strings using *.

Example:

```
str1 = "Hello"
```

```
str2 = "World"
```

```
# Concatenation
```

```
result = str1 + " " + str2
```

```
print(result) # Output: Hello World
```

```
# Repetition
```

```
print(str1 * 3) # Output: HelloHelloHello
```

4. Basic built-in string methods

Python provides many built-in string methods to manipulate text.

Example:

```
text = " python programming "
```

```
# Remove spaces
```

```
print(text.strip())    # Output: python programming
```

```
# Convert to uppercase
```

```
print(text.upper())    # Output: PYTHON PROGRAMMING
```

```
# Convert to lowercase
```

```
print(text.lower())    # Output: python programming
```

Replace text

```
print(text.replace("python", "Java")) # Output: Java programming
```

Split into list

```
print(text.split()) # Output: ['python', 'programming']
```

Check if string starts or ends with something

```
print(text.startswith(" python")) # Output: True
```

```
print(text.endswith("ing ")) # Output: True
```