1. len(): Purpose: The len() function is used to find out the length (number of items) of an object such as a string, list, tuple, dictionary, etc. How it works: It counts the total number of elements inside an object. When to use: Whenever you need to find the length of a sequence (like a string, list, etc.), len() is your go-to function.

In [1]:
```
name = "Python"
print(len(name))
```

```
6
```

1. type(): Purpose: The type() function tells you the type (data type) of an object. How it works: Python assigns types to objects, such as int, float, list, dict, etc. type() lets you check what type an object is. When to use: Use type() when debugging or working with dynamic data where you need to confirm the type of an object.

In [2]:
```
num = 5
print(type(num))
```

```
<class 'int'>
```

1. abs(): Purpose: The abs() function returns the absolute value of a number, meaning it removes any negative sign. How it works: If the number is negative, it simply makes it positive; if it's already positive, it remains unchanged. When to use: Whenever you need the positive magnitude of a number, such as in distance calculations.

In [3]:
```
negative_num = -10
print(abs(negative_num))
```

```
10
```

1. min() and max(): Purpose: These functions are used to find the smallest (min()) or largest (max()) element in an iterable (list, tuple, set, etc.). How they work: They traverse through the iterable and compare all the values, returning the minimum or maximum value. When to use: Use these when you need to find the extreme values in a dataset.

In [4]:
```
numbers = [3, 1, 7, 9, 2]
print(min(numbers))
print(max(numbers))
```

```
1
9
```

1. sum(): Purpose: The sum() function adds up all the elements in an iterable (like a list). How it works: It iterates over the elements and returns the total. When to use: Use sum() when you need to calculate the total of numeric values, like when calculating a bill or the total marks in a subject.

In [5]:
```
numbers = [1, 2, 3, 4]
print(sum(numbers))
```

```
10
```

1. round(): Purpose: The round() function rounds a floating-point number to a given precision (number of decimal places). How it works: By default, it rounds to the nearest whole number, but you can specify the number of decimal places. When to use: Use round() when you want to simplify or reduce precision for things like prices or percentages.

In [6]:
```
num = 3.14159
print(round(num, 2))
```

```
3.14
```

1. sorted(): Purpose: The sorted() function returns a sorted version of an iterable. How it works: It creates a new list containing the elements in sorted order, leaving the original iterable unchanged. When to use: Use sorted() when you need the elements of an iterable arranged in a particular order (ascending by default).

In [7]:
```
unsorted_list = [3, 1, 2, 4]
print(sorted(unsorted_list))
```

```
[1, 2, 3, 4]
```

1. enumerate(): Purpose: The enumerate() function adds a counter (index) to an iterable. How it works: It produces a tuple containing the index and the corresponding value from the iterable. When to use: Use enumerate() when you need both the index and the value from an iterable (like in a for-loop).

In [8]:
```
languages = ['Python', 'Java', 'C++']
for index, lang in enumerate(languages):
    print(index, lang)
```

```
0 Python
1 Java
2 C++
```

In [ ]:
```
9. zip():
Purpose: The zip() function combines two or more iterables into pairs.
How it works: It takes elements from the iterables one by one and forms pairs.
When to use: Use zip() when you need to pair elements from multiple lists, like pai
```

In [9]:
```
names = ['sidhu', 'Balu']
scores = [85, 90]
for name, score in zip(names, scores):
    print(name, score)
```

```
sidhu 85
Balu 90
```

1. any() and all(): Purpose: any() returns True if at least one element is True. all() returns True only if all elements are True. How they work: They evaluate boolean values of elements. When to use: Use any() or all() when working with conditions or logical checks.

In [10]:
```
conditions = [True, False, True]
print(any(conditions))
```

```
print(all(conditions))
```

```
True
False
```

1. input(): Purpose: The input() function takes input from the user as a string. How it works: It pauses program execution and waits for the user to enter data. When to use: Use input() when interacting with users in your programs.

In [11]:
```
user_input = input("Enter your name: ")
print("Hello, " + user_input)
```

```
Enter your name: deepika
Hello, deepika
```

1. range(): Purpose: The range() function generates a sequence of numbers, often used in loops. How it works: It produces numbers starting from 0 (by default) and increments by 1 (also by default) up to a specified number. When to use: Use range() to create a sequence of numbers for iteration.

In [12]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

1. map(): Purpose: The map() function applies a function to all items in an iterable. How it works: It takes a function and an iterable, then applies the function to each item. When to use: Use map() when you want to apply a transformation to every item in an iterable.

In [13]:
```
def square(x):
    return x * x

numbers = [1, 2, 3]
result = map(square, numbers)
print(list(result))
```

```
[1, 4, 9]
```

1. filter(): Purpose: The filter() function filters elements based on a function. How it works: It applies a filtering function to each item, only keeping the ones where the function returns True. When to use: Use filter() to remove unwanted values from an iterable.

In [14]:
```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4]
result = filter(is_even, numbers)
print(list(result))
```

```
[2, 4]
```

lambda function is a small anonymous function defined using the lambda keyword. It's often used when a simple function is required for a short period of time, such as in cases of

filtering, sorting, or mapping. Syntax of Lambda Functions lambda arguments: expression lambda: The keyword used to define a lambda function. arguments: Comma-separated inputs (similar to regular function arguments). expression: A single expression that is evaluated and returned. Lambda functions can only contain one expression, no statements or multiple lines.

In [15]:
```
add = lambda x, y: x + y
print(add(5, 3))
```

```
8
```

In [16]:
```
#Regular Function
def add(x, y):
    return x + y

print(add(5, 3))
```

```
8
```

Using lambda with filter() The filter() function is used to filter elements based on a condition. You can pass a lambda function to filter() to specify the condition.

In [17]:
```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
[2, 4, 6]
```

Using lambda with map() The map() function is used to apply a function to every element of an iterable (like a list).

In [18]:
```
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

conditional Lambda Functions You can use an inline if-else statement inside a lambda function.

In [20]:
```
max_value = lambda x, y: x if x > y else y
print(max_value(10, 20))
```

```
20
```

LIST COMPREHENSION: In programming, comprehension refers to a readable way to generate or transform data structures (like lists, sets, or dictionaries) by applying an expression to each element in an existing iterable (such as a list or range), often with an optional condition.

In Python, comprehension allows you to create new data structures from existing ones in a single line, while reducing the need for code, like for loops.

Syntax: expression for item in iterable if condition

Where:

expression: The value or operation applied to each element (like item * 2, item.upper(), etc.). item: The variable that takes the value of each element from the iterable. iterable: A sequence (like a list, string, or range). condition (optional): A filtering condition to select certain elements.

In [1]:
```
#Basic List Comprehension: Creating a List of Even Numbers
#Without list comprehension:
even_numbers = []
for i in range(10):
    even_numbers.append(i)

print(even_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [3]:
```
#With list comprehension:
even_numbers = [i for i in range(10)]
print(even_numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [4]:
```
# List Comprehension with Condition: Extracting Even Numbers
#Without list comprehension:
even_numbers = []
for i in range(10):
    if i % 2 == 0:
        even_numbers.append(i)

print(even_numbers)
```

```
[0, 2, 4, 6, 8]
```

In [5]:
```
#With list comprehension:
even_numbers = [i for i in range(10) if i % 2 == 0]
print(even_numbers)
```

```
[0, 2, 4, 6, 8]
```

Applying a Function to Each Element You can apply functions to each element while generating a list. For example, let's convert each string in a list to uppercase.

Without list comprehension:

In [6]:
```
names = ["cse", "cse-h1", "cse-h2"]
uppercase_names = []
for name in names:
    uppercase_names.append(name.upper())

print(uppercase_names)
```

```
['CSE', 'CSE-H1', 'CSE-H2']
```

In [7]:
```
#With list comprehension:
names = ["samyak", "surabhi", "#include"]
uppercase_names = [name.upper() for name in names]
print(uppercase_names)
```

```
['SAMYAK', 'SURABHI', '#INCLUDE']
```

List Comprehension with Multiple Conditions You can add multiple conditions to filter the list further. For example, creating a list of numbers divisible by both 2 and 3.

In [8]:
```
#Without list comprehension:
numbers = []
for i in range(20):
    if i % 2 == 0 and i % 3 == 0:
        numbers.append(i)

print(numbers)
```

```
[0, 6, 12, 18]
```

In [9]:
```
#With list comprehension:
numbers = [i for i in range(20) if i % 2 == 0 and i % 3 == 0]
print(numbers)
```

```
[0, 6, 12, 18]
```

In [ ]:
```

```