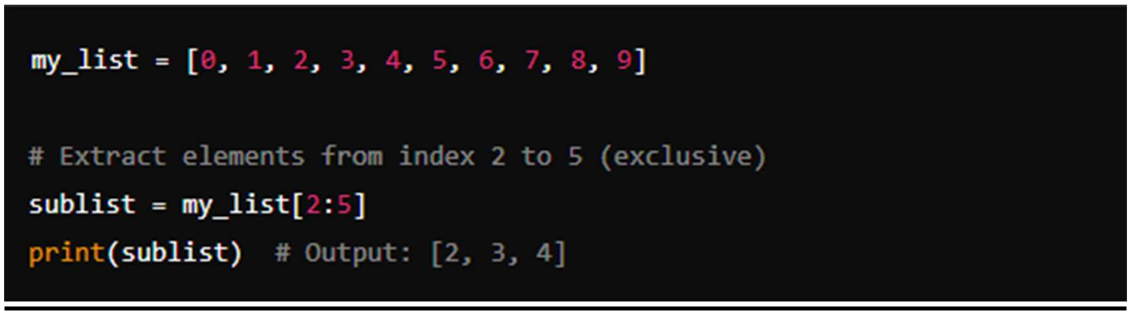
Advanced List Slicing in Python

List slicing in Python provides a powerful mechanism for extracting elements from a list based on their positions. Advanced list slicing takes this concept further, enabling you to extract sublists with more complex patterns or criteria.

Basic List Slicing

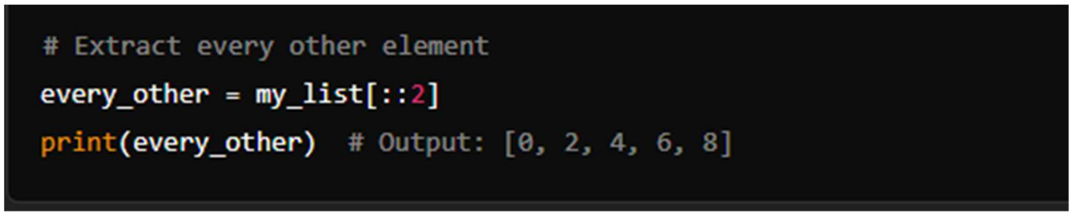
Before going into advanced slicing, let's review basic list slicing:



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

# Extract elements from index 2 to 5 (exclusive)
sublist = my_list[2:5]
print(sublist)  # Output: [2, 3, 4]
```

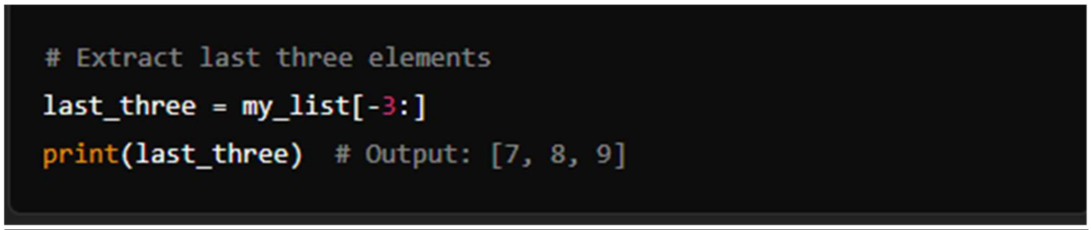
Advanced List Slicing Techniques

1. **Step Size: You can specify a step size to skip elements while slicing.**



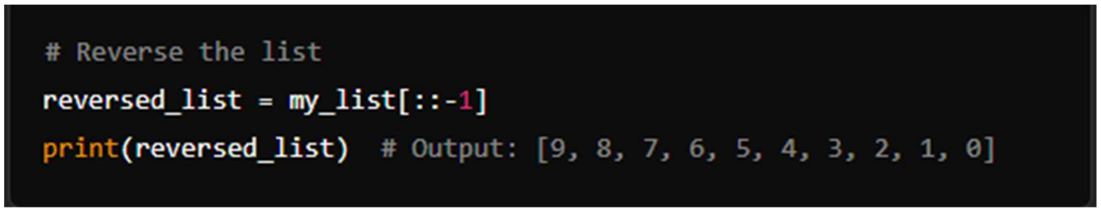
```python
# Extract every other element
every_other = my_list[::2]
print(every_other)  # Output: [0, 2, 4, 6, 8]
```

2. **Negative Indices: Negative indices count from the end of the list.**



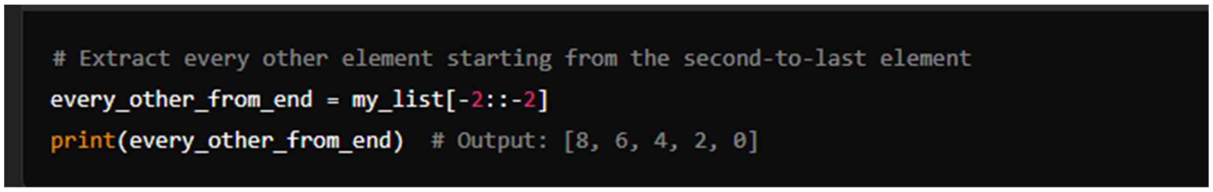
```python
# Extract last three elements
last_three = my_list[-3:]
print(last_three)  # Output: [7, 8, 9]
```

3. **Reverse a List: Use a negative step size to reverse the list.**



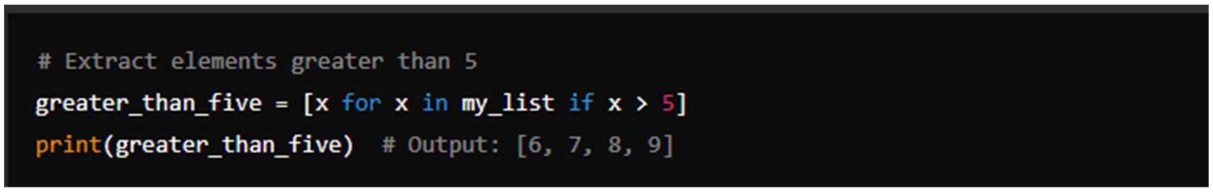
```python
# Reverse the list
reversed_list = my_list[::-1]
print(reversed_list)  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

4. **Skipping Elements with Step Size: You can combine step size with negative indices.**



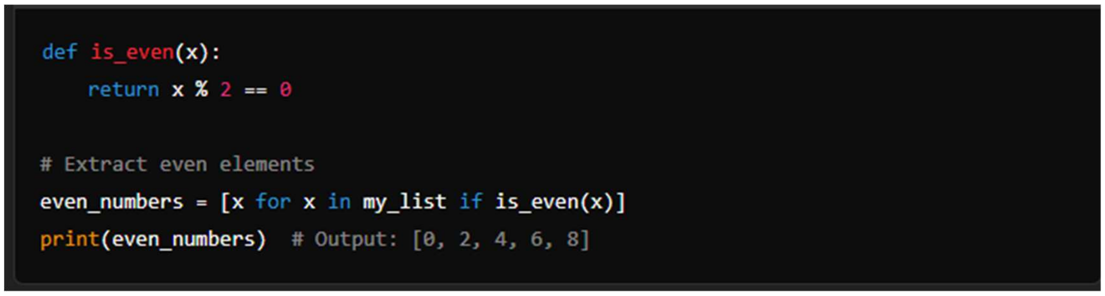
```python
# Extract every other element starting from the second-to-last element
every_other_from_end = my_list[-2::-2]
print(every_other_from_end)  # Output: [8, 6, 4, 2, 0]
```

5. **Conditional Slicing: Use a conditional expression inside square brackets to filter elements based on a condition.**



```python
# Extract elements greater than 5
greater_than_five = [x for x in my_list if x > 5]
print(greater_than_five)  # Output: [6, 7, 8, 9]
```

6. **Slicing with Functions: You can use functions to filter elements dynamically.**



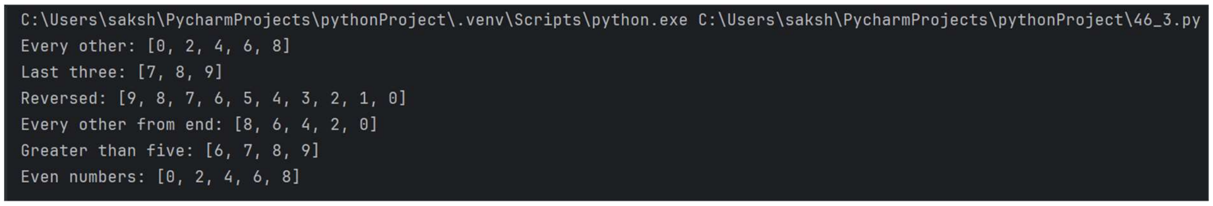
```python
def is_even(x):
    return x % 2 == 0

# Extract even elements
even_numbers = [x for x in my_list if is_even(x)]
print(even_numbers)  # Output: [0, 2, 4, 6, 8]
```

CODE:

```python
# Sample list
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# 1. Extract every other element
every_other = my_list[::2]
print("Every other:", every_other)  # Output: [0, 2, 4, 6, 8]

# 2. Extract last three elements
last_three = my_list[-3:]
print("Last three:", last_three)  # Output: [7, 8, 9]

# 3. Reverse the list
reversed_list = my_list[::-1]
print("Reversed:", reversed_list)  # Output: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# 4. Extract every other element starting from the second-to-last element
every_other_from_end = my_list[-2::-2]
print("Every other from end:", every_other_from_end)  # Output: [8, 6, 4, 2, 0]
```

```
# 5. Conditional slicing: Extract elements
greater than 5
greater_than_five = [x for x in my_list if x >
5]
print("Greater than five:", greater_than_five)
# Output: [6, 7, 8, 9]

# 6. Slicing with functions: Extract even
elements
def is_even(x):
    return x % 2 == 0

even_numbers = [x for x in my_list if
is_even(x)]
print("Even numbers:", even_numbers)  # Output:
[0, 2, 4, 6, 8]
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\46_3.py
Every other: [0, 2, 4, 6, 8]
Last three: [7, 8, 9]
Reversed: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
Every other from end: [8, 6, 4, 2, 0]
Greater than five: [6, 7, 8, 9]
Even numbers: [0, 2, 4, 6, 8]
```

BISECT MODULE in Python:

The `bisect` module in Python provides support for maintaining a list in sorted order without having to explicitly sort the list after each insertion. It achieves this through efficient binary search algorithms for inserting and locating elements in a sorted sequence.

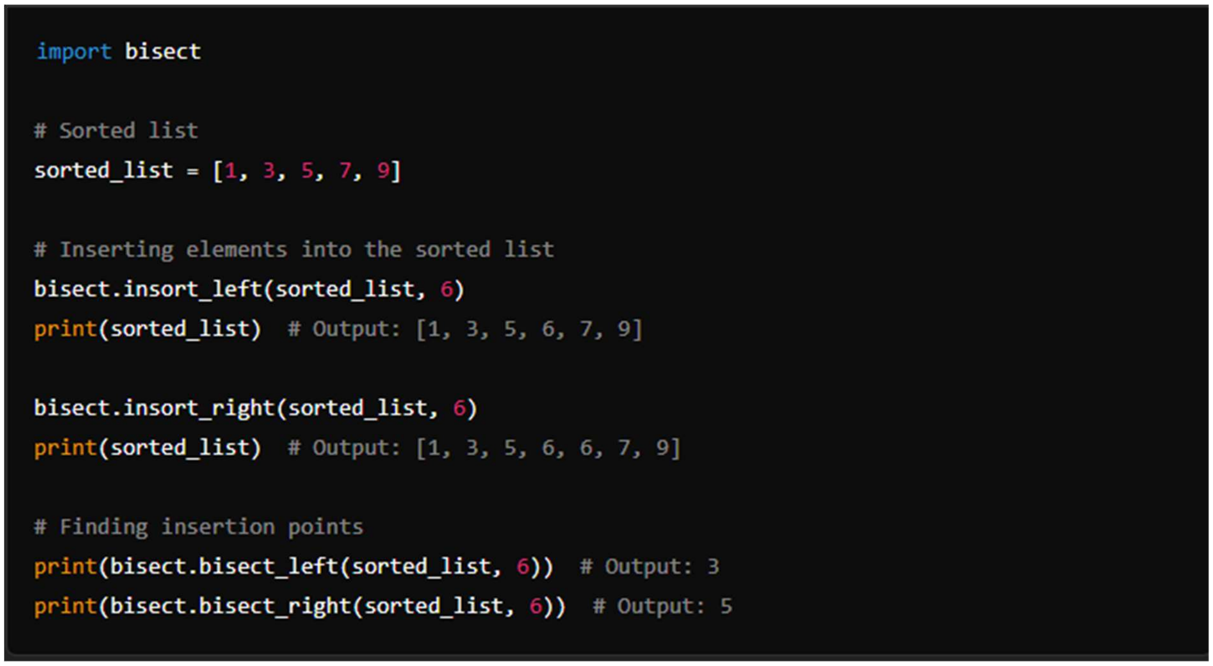
Basic Functionality

The `bisect` module primarily includes two main functions: `bisect_left()` and `bisect_right()`. These functions find the insertion points for a specified element in a sorted list to maintain its sorted order.

- **`bisect_left(a, x, lo=0, hi=len(a))`: This function finds the index where the element `x` should be inserted in the sorted list `a` to maintain its sorted order. If the element is already present, it returns the leftmost insertion point. The `lo` and `hi` parameters allow specifying a subrange of the list to search within.**
- **`bisect_right(a, x, lo=0, hi=len(a))`: Similar to `bisect_left()`, this function finds the rightmost insertion point for the element `x` in the sorted list `a`.**

Example Usage

Let's explore how you can use the `bisect` module in practice:



```python
import bisect

# Sorted list
sorted_list = [1, 3, 5, 7, 9]

# Inserting elements into the sorted list
bisect.insort_left(sorted_list, 6)
print(sorted_list)  # Output: [1, 3, 5, 6, 7, 9]

bisect.insort_right(sorted_list, 6)
print(sorted_list)  # Output: [1, 3, 5, 6, 6, 7, 9]

# Finding insertion points
print(bisect.bisect_left(sorted_list, 6))  # Output: 3
print(bisect.bisect_right(sorted_list, 6))  # Output: 5
```

In this example:

- **We start with a sorted list `[1, 3, 5, 7, 9]`.**
- **We use `insort_left()` and `insort_right()` to insert the element 6 into the list while maintaining its sorted order.**
- **We use `bisect_left()` and `bisect_right()` to find the leftmost and rightmost insertion points for the element 6 in the list.**

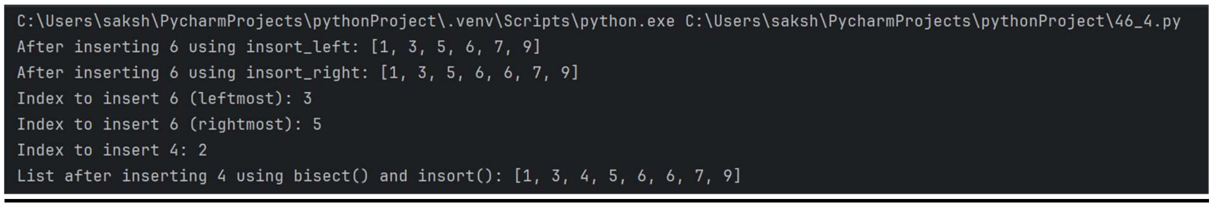
Additional Functions

The `bisect` module also provides other functions like `bisect()` and `insort()` which are aliases for `bisect_right()` and `insort_right()` respectively. These functions are more intuitive for many users as they return the same insertion point as `bisect_right()` and insert elements in the same way as `insort_right()`.

CODE:

```
import bisect

# Sorted list
sorted_list = [1, 3, 5, 7, 9]

# Inserting elements into the sorted list
bisect.insort_left(sorted_list, 6)
print("After inserting 6 using insort_left:",
sorted_list)  # Output: [1, 3, 5, 6, 7, 9]

bisect.insort_right(sorted_list, 6)
print("After inserting 6 using insort_right:",
sorted_list)  # Output: [1, 3, 5, 6, 6, 7, 9]

# Finding insertion points
print("Index to insert 6 (leftmost):",
bisect.bisect_left(sorted_list, 6))  # Output: 3
print("Index to insert 6 (rightmost):",
bisect.bisect_right(sorted_list, 6))  # Output:
5

# Additional functions (bisect() and insort())
index_to_insert = bisect.bisect(sorted_list, 4)
bisect.insort(sorted_list, 4)
print("Index to insert 4:", index_to_insert)  #
Output: 2
print("List after inserting 4 using bisect() and
insort():", sorted_list)  # Output: [1, 3, 4, 5,
6, 6, 7, 9]
```

Output:



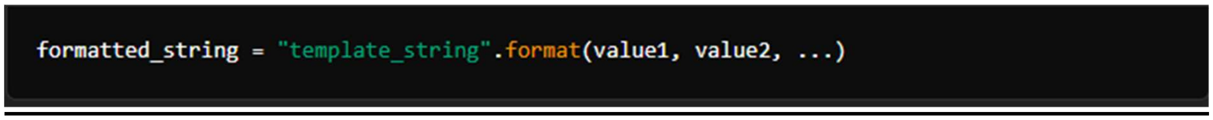
```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\46_4.py
After inserting 6 using insort_left: [1, 3, 5, 6, 7, 9]
After inserting 6 using insort_right: [1, 3, 5, 6, 6, 7, 9]
Index to insert 6 (leftmost): 3
Index to insert 6 (rightmost): 5
Index to insert 4: 2
List after inserting 4 using bisect() and insort(): [1, 3, 4, 5, 6, 6, 7, 9]
```

FORMAT Function in Python:

The `format()` function in Python is a powerful method for formatting strings. It allows you to create dynamic strings with placeholders that can be replaced by values or variables. This function provides flexibility and control over how data is presented in your Python programs.

Basic Syntax

The basic syntax of the `format()` function is as follows:

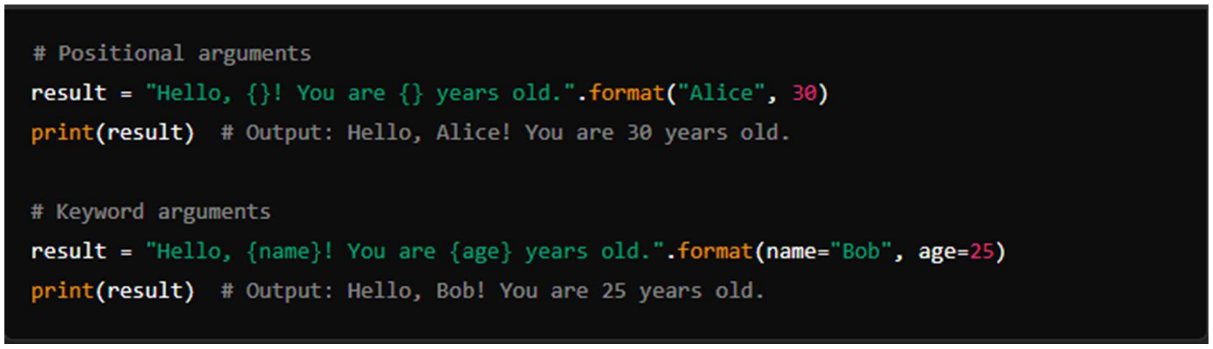


```
formatted_string = "template_string".format(value1, value2, ...)
```

Here, `template_string` is a string containing placeholders (curly braces `{}`), and `value1`, `value2`, etc., are the values or variables used to replace the placeholders.

Positional and Keyword Arguments

You can pass arguments to the `format()` function as positional or keyword arguments. Positional arguments are replaced in the order they appear, while keyword arguments are replaced based on the keys they correspond to in the template string.

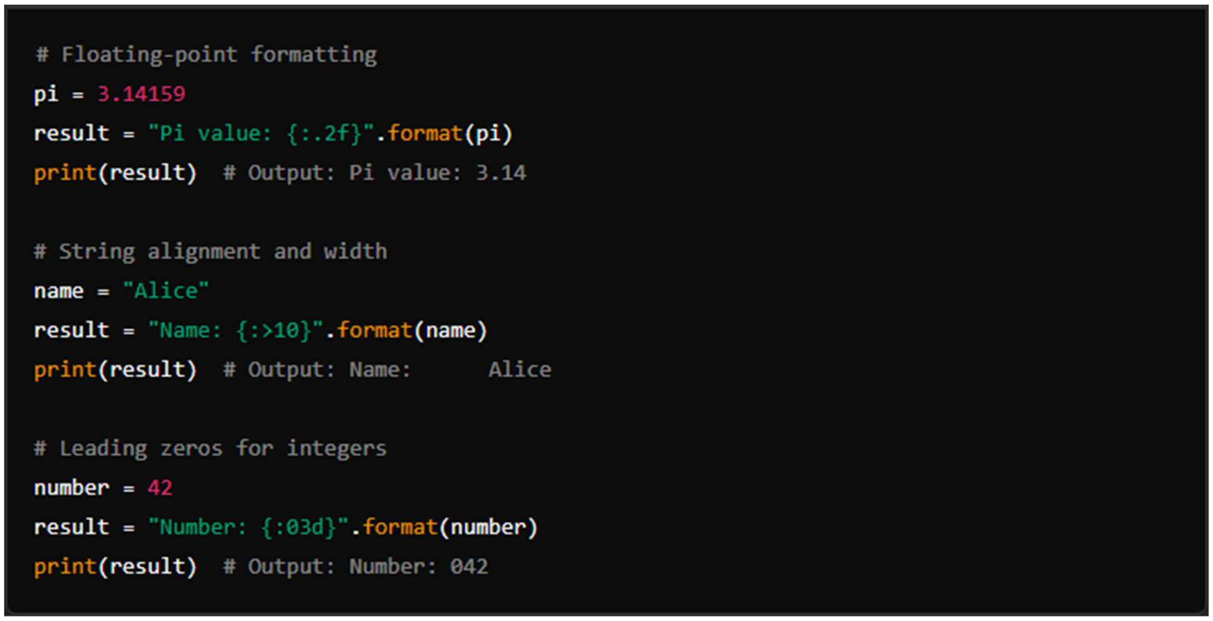


```python
# Positional arguments
result = "Hello, {}! You are {} years old.".format("Alice", 30)
print(result)  # Output: Hello, Alice! You are 30 years old.

# Keyword arguments
result = "Hello, {name}! You are {age} years old.".format(name="Bob", age=25)
print(result)  # Output: Hello, Bob! You are 25 years old.
```

Formatting Options

The `format()` function supports various formatting options to control how values are displayed, including:

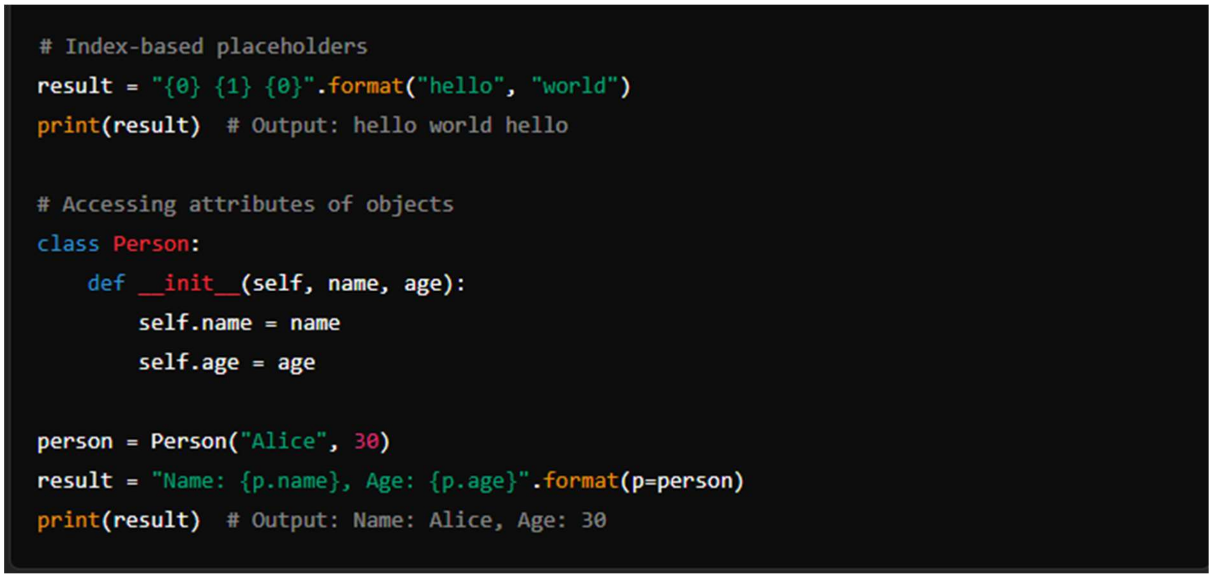
- **Specifying the number of decimal places for floating-point numbers.**
- **Controlling the alignment and width of strings.**
- **Adding leading zeros to integers.**



```python
# Floating-point formatting
pi = 3.14159
result = "Pi value: {:.2f}".format(pi)
print(result)  # Output: Pi value: 3.14

# String alignment and width
name = "Alice"
result = "Name: {:>10}".format(name)
print(result)  # Output: Name:      Alice

# Leading zeros for integers
number = 42
result = "Number: {:03d}".format(number)
print(result)  # Output: Number: 042
```

Advanced Usage

The `format()` function also allows for more advanced formatting, including using index-based placeholders, accessing attributes of objects, and calling methods on objects.

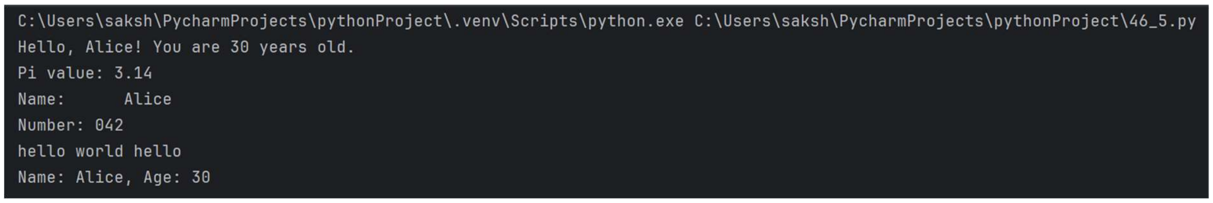


```python
# Index-based placeholders
result = "{0} {1} {0}".format("hello", "world")
print(result)  # Output: hello world hello

# Accessing attributes of objects
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Alice", 30)
result = "Name: {p.name}, Age: {p.age}".format(p=person)
print(result)  # Output: Name: Alice, Age: 30
```

CODE:

```python
# Basic usage
name = "Alice"
age = 30
result = "Hello, {}! You are {} years old.".format(name, age)
print(result)  # Output: Hello, Alice! You are 30 years old.

# Floating-point formatting
pi = 3.14159
result = "Pi value: {:.2f}".format(pi)
print(result)  # Output: Pi value: 3.14

# String alignment and width
name = "Alice"
result = "Name: {:>10}".format(name)
print(result)  # Output: Name:      Alice

# Leading zeros for integers
number = 42
result = "Number: {:03d}".format(number)
print(result)  # Output: Number: 042

# Index-based placeholders
result = "{0} {1} {0}".format("hello", "world")
print(result)  # Output: hello world hello

# Accessing attributes of objects
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Alice", 30)
result = "Name: {p.name}, Age:
{p.age}".format(p=person)
print(result)  # Output: Name: Alice, Age: 30
```

Output:



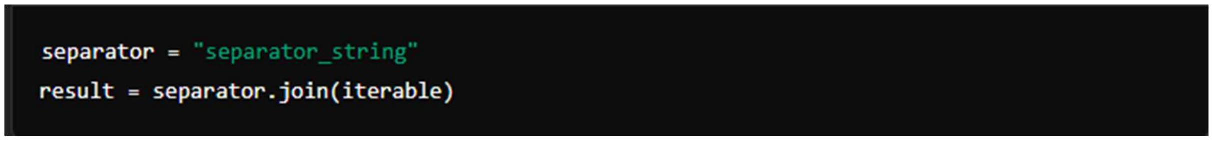
```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\46_5.py
Hello, Alice! You are 30 years old.
Pi value: 3.14
Name:      Alice
Number: 042
hello world hello
Name: Alice, Age: 30
```

JOIN Function in Python

The `join()` function in Python is a powerful method for joining the elements of an iterable (such as a list, tuple, or string) into a single string. It is particularly useful when you need to concatenate multiple strings together efficiently.

Basic Syntax

The basic syntax of the `join()` function is as follows:

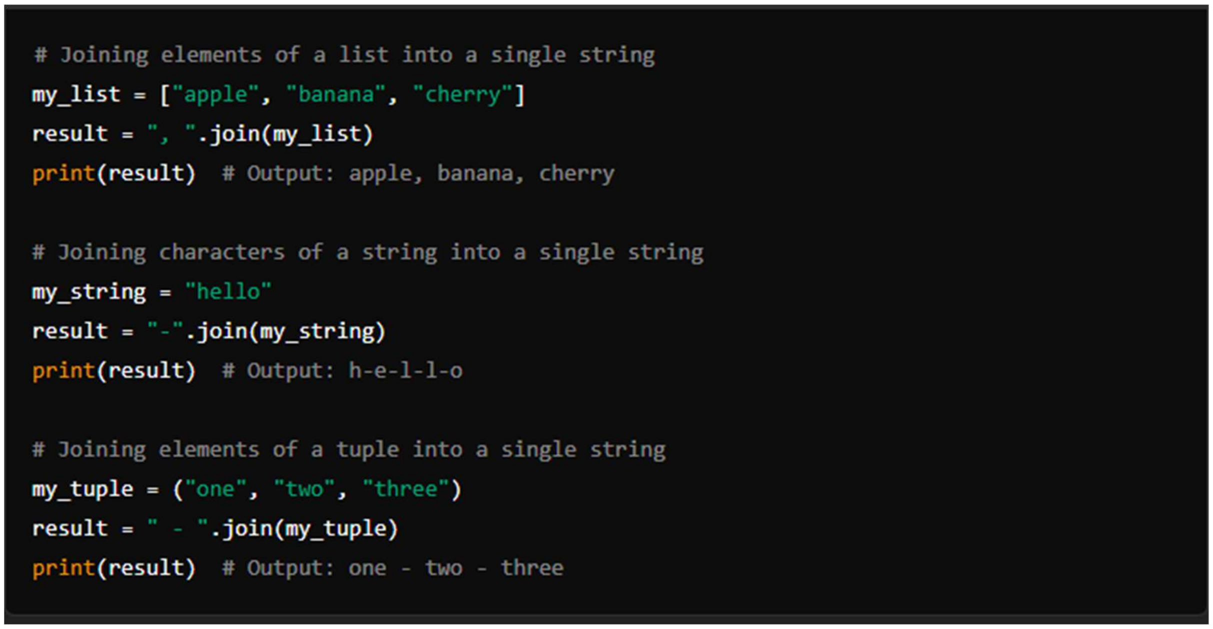


```
separator = "separator_string"
result = separator.join(iterable)
```

Here, `separator_string` is the string that will be used to join the elements, and `iterable` is the iterable containing the elements to be joined.

Example Usage

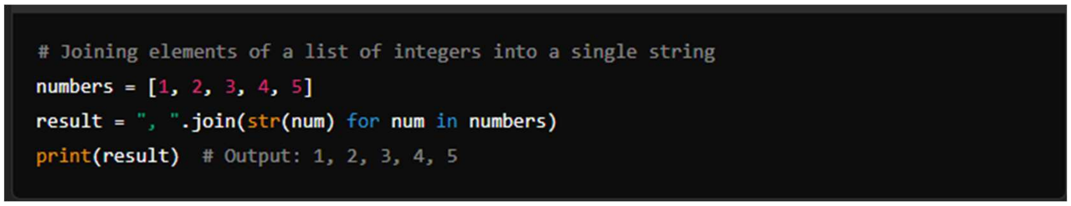
Let's explore how you can use the `join()` function in practice:



```
# Joining elements of a list into a single string
my_list = ["apple", "banana", "cherry"]
result = ", ".join(my_list)
print(result)  # Output: apple, banana, cherry

# Joining characters of a string into a single string
my_string = "hello"
result = "-".join(my_string)
print(result)  # Output: h-e-l-l-o

# Joining elements of a tuple into a single string
my_tuple = ("one", "two", "three")
result = " - ".join(my_tuple)
print(result)  # Output: one - two - three
```

Handling Different Types of Iterables

The `join()` function can work with various types of iterables, including lists, tuples, and strings. However, it expects all elements of the iterable to be strings. If the iterable contains non-string elements, you'll need to convert them to strings first.

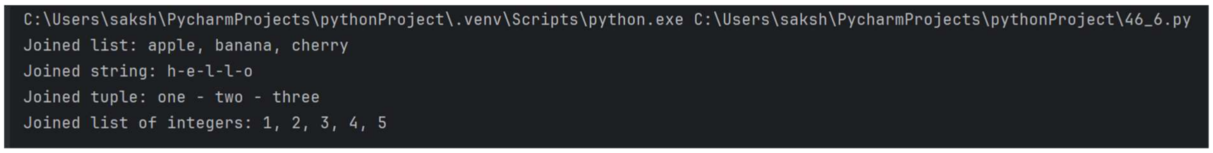


```
# Joining elements of a list of integers into a single string
numbers = [1, 2, 3, 4, 5]
result = ", ".join(str(num) for num in numbers)
print(result)  # Output: 1, 2, 3, 4, 5
```

CODE:

```python
# Joining elements of a list into a single string
my_list = ["apple", "banana", "cherry"]
result = ", ".join(my_list)
print("Joined list:", result)  # Output: apple, banana, cherry

# Joining characters of a string into a single string
my_string = "hello"
result = "-".join(my_string)
print("Joined string:", result)  # Output: h-e-l-l-o

# Joining elements of a tuple into a single string
my_tuple = ("one", "two", "three")
result = " - ".join(my_tuple)
print("Joined tuple:", result)  # Output: one - two - three

# Joining elements of a list of integers into a single string
numbers = [1, 2, 3, 4, 5]
result = ", ".join(str(num) for num in numbers)
print("Joined list of integers:", result)  # Output: 1, 2, 3, 4, 5
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\46_6.py
Joined list: apple, banana, cherry
Joined string: h-e-l-l-o
Joined tuple: one - two - three
Joined list of integers: 1, 2, 3, 4, 5
```

PD
PENTEST DIARIES
HTTPS://WWW.SAKSHAMDIXIT.COM

‘is” vs ‘==’ & what is the difference in python:

In Python, `is` and == are two different operators used for comparison, and they have different behaviors:

`is` Operator:

The `is` operator checks whether two variables refer to the same object in memory, i.e., it checks if the two variables are pointing to the same memory location.

== Operator:

The == operator checks whether the values of two variables are equal or not, i.e., it compares the values stored in the variables regardless of whether they refer to the same object in memory.

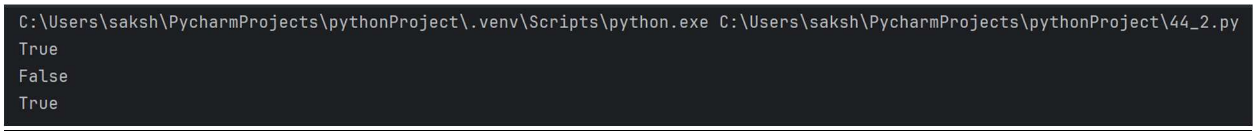
Here's a code example to illustrate the difference:

CODE:

```
# Define two lists with the same values
list1 = [1, 2, 3]
list2 = [1, 2, 3]

# Using '==' to compare the values
print(list1 == list2)   # Output: True

# Using 'is' to compare the memory locations
print(list1 is list2)   # Output: False

# Define another variable pointing to list1
list3 = list1
```

```python
# Using 'is' to compare the memory locations
print(list1 is list3)  # Output: True
```

Output:



```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\44_2.py
True
False
True
```

Key Differences:

- **`==` compares the values of two variables.**
- **`is` compares the memory locations of two variables.**

Best Practices:

- **Use `==` when comparing the values of variables.**
- **Use `is` when checking for identity, i.e., when you want to check if two variables refer to the same object in memory.**

Understanding the distinction between `is` and `==` is important for writing correct and efficient Python code, especially when dealing with mutable objects like lists, dictionaries, and custom objects.