Data Structure and Algorithm Analysis and Designs

List and Array

ARUN REUNGSINKONKARN, PH.D.

List

List

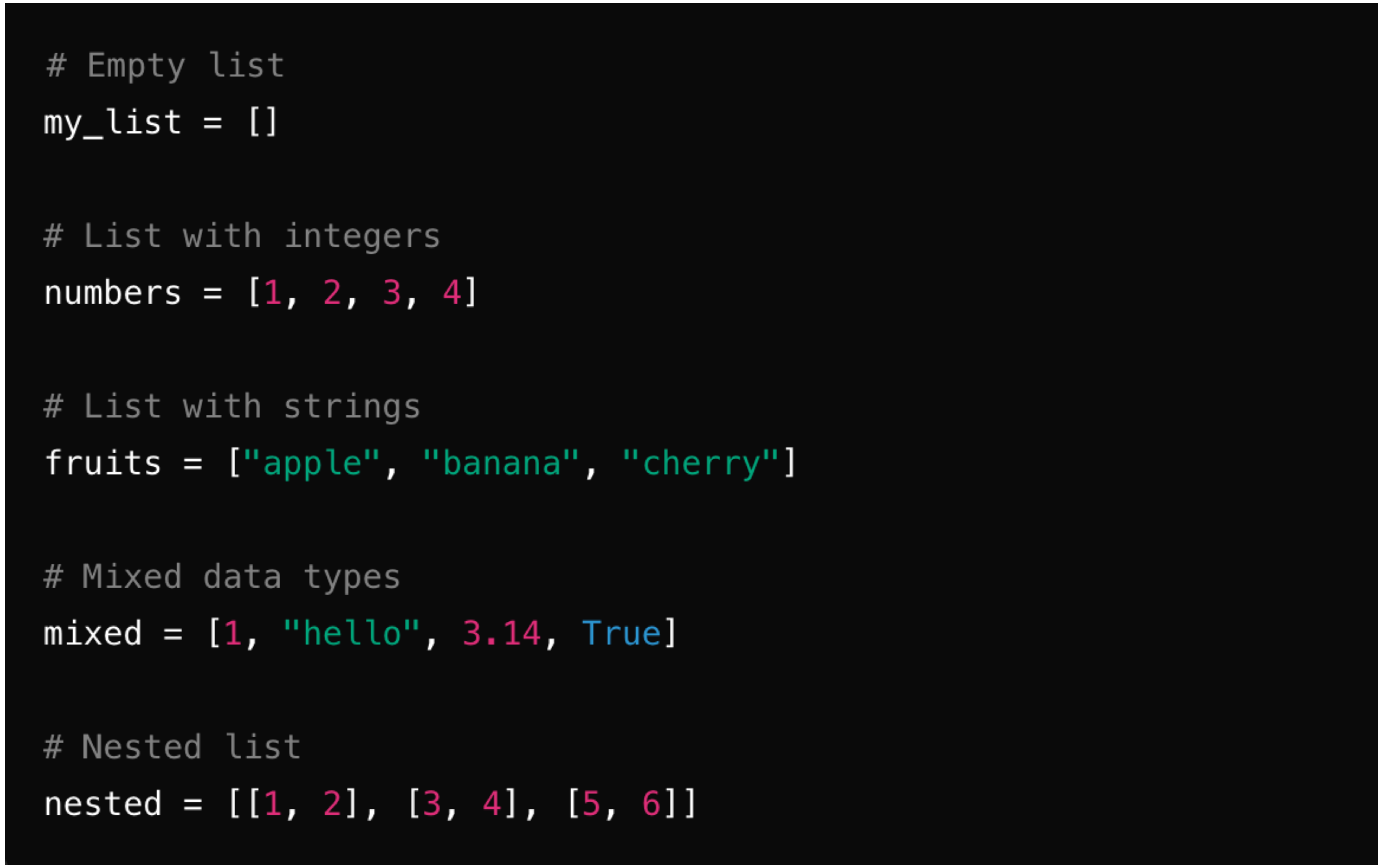
A **list** is a data structure that is used to store an **ordered collection of items**. Lists are dynamic, meaning their size can change, and they can store a mix of different data types, making them one of the most versatile tools.

Characteristics of Lists

1. **Ordered:** Items in a list maintain their order.
2. **Mutable:** Items in a list can be changed after the list is created.
3. **Dynamic:** Lists can grow or shrink as needed.
4. **Allows Duplicates:** Lists can contain the same value multiple times.
5. **Heterogeneous:** Lists can contain elements of different types.

Creating a List

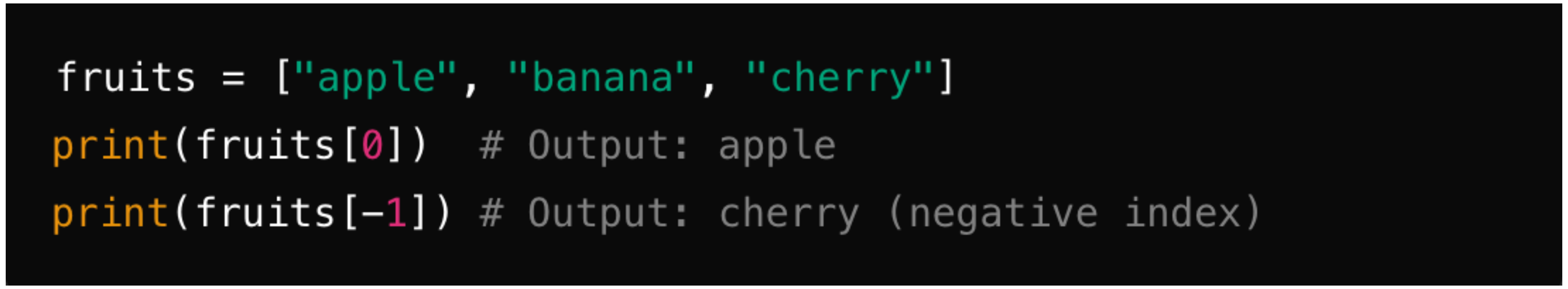
Lists are defined by placing items inside square brackets [ ], separated by commas.



```python
# Empty list
my_list = []

# List with integers
numbers = [1, 2, 3, 4]

# List with strings
fruits = ["apple", "banana", "cherry"]

# Mixed data types
mixed = [1, "hello", 3.14, True]

# Nested list
nested = [[1, 2], [3, 4], [5, 6]]
```

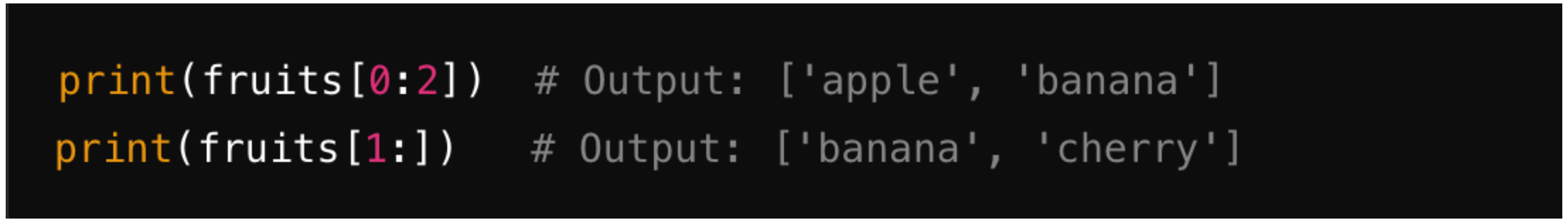
Accessing List Elements

1. Indexing: Access items by their position (starting from 0).



```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[-1]) # Output: cherry (negative index)
```

2. **Slicing**: Extract multiple elements using slicing.

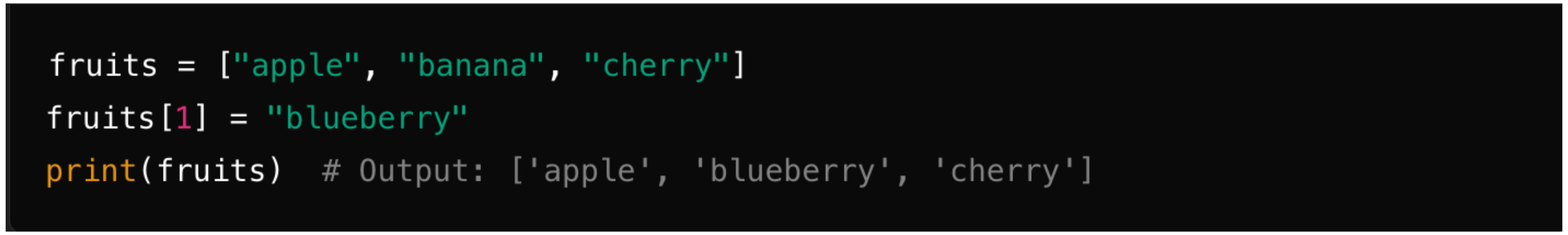


```
print(fruits[0:2])  # Output: ['apple', 'banana']
print(fruits[1:])   # Output: ['banana', 'cherry']
```

Modifying a List

Lists can be modified by changing, adding, or removing elements

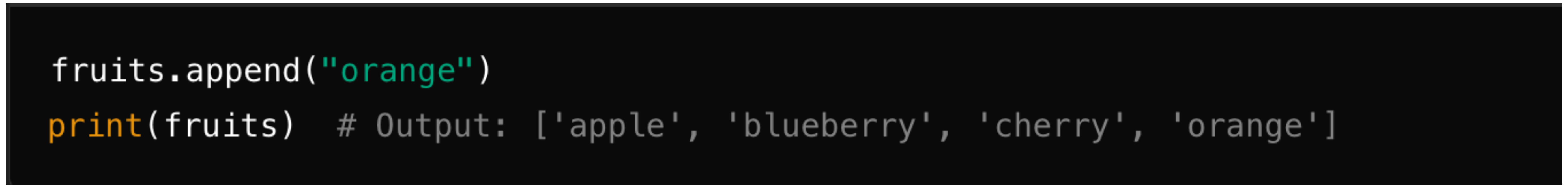
Change an Element:



```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "blueberry"
print(fruits)  # Output: ['apple', 'blueberry', 'cherry']
```

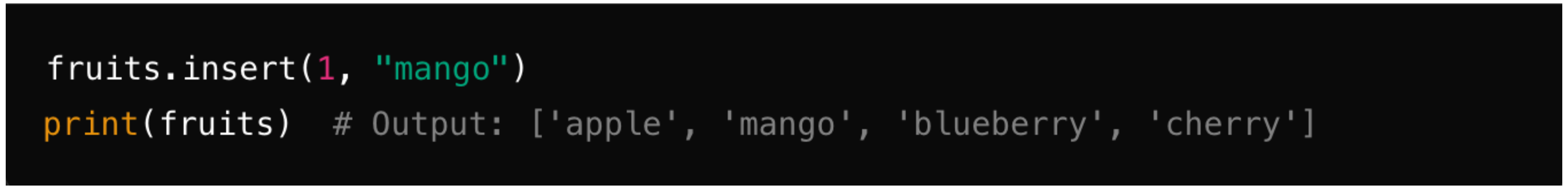
Add Elements:

Append: Add an item to the end of the list.



```
fruits.append("orange")
print(fruits)  # Output: ['apple', 'blueberry', 'cherry', 'orange']
```

Insert: Add an item at a specific position.

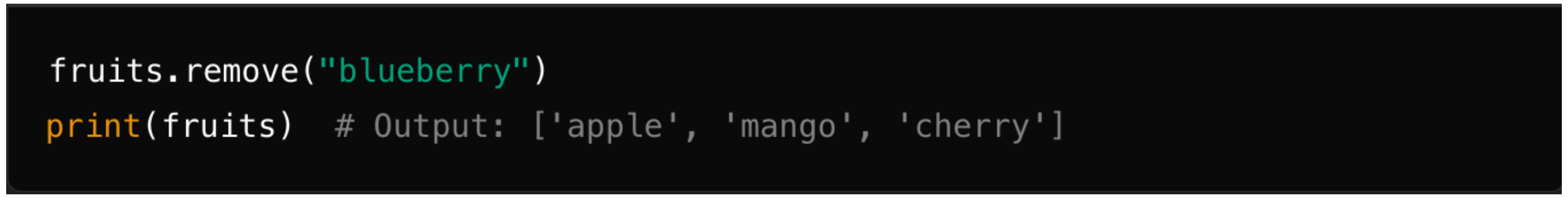


```
fruits.insert(1, "mango")
print(fruits)  # Output: ['apple', 'mango', 'blueberry', 'cherry']
```

Modifying a List (cont.)

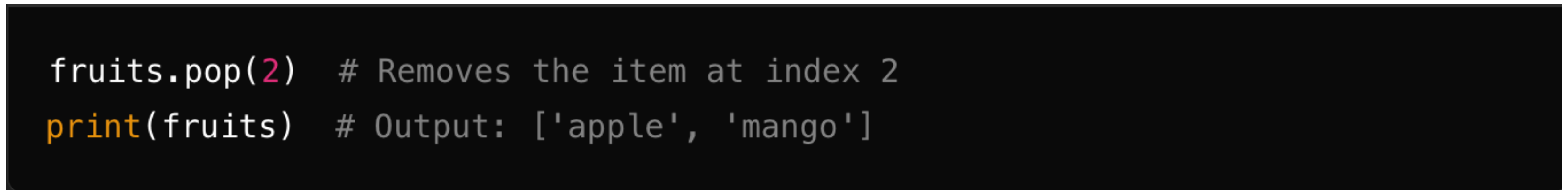
Remove Elements:

- Remove by Value:



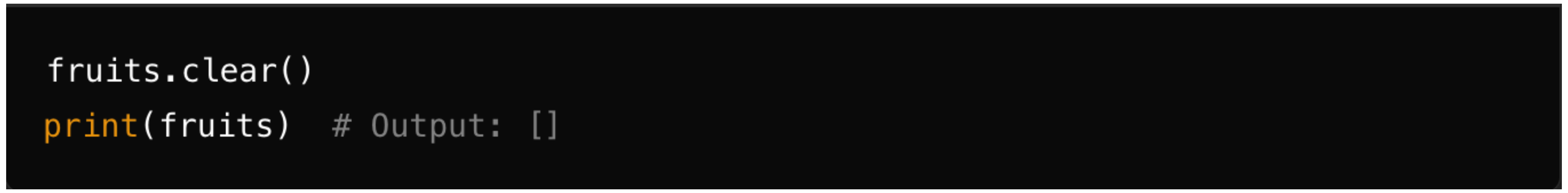
```python
fruits.remove("blueberry")
print(fruits)  # Output: ['apple', 'mango', 'cherry']
```

- Remove by Index:



```python
fruits.pop(2)  # Removes the item at index 2
print(fruits)  # Output: ['apple', 'mango']
```

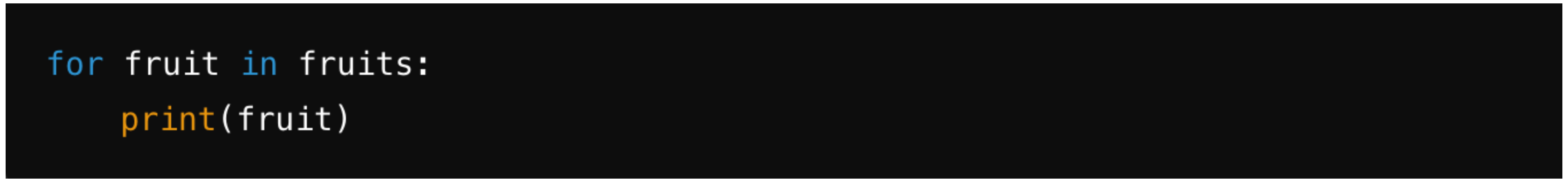
- Clear All:



```python
fruits.clear()
print(fruits)  # Output: []
```

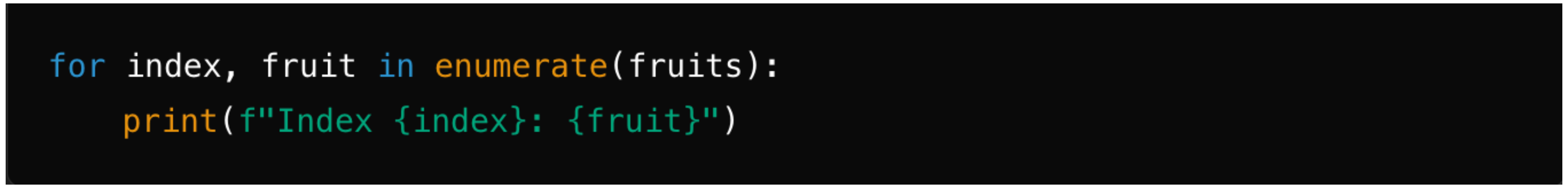
Iterating Through a List

Using a `for` loop:



```
for fruit in fruits:
    print(fruit)
```

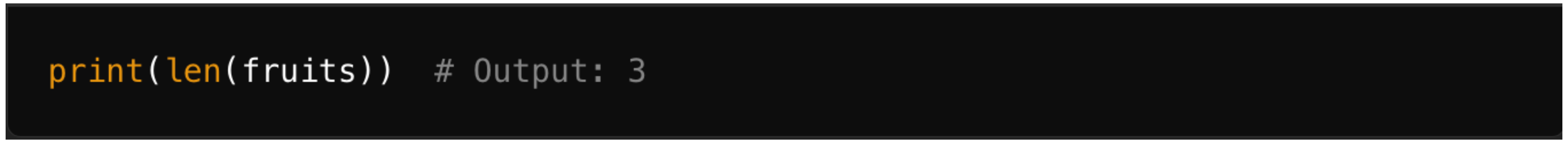
Using enumerate( ) for index and value:



```
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

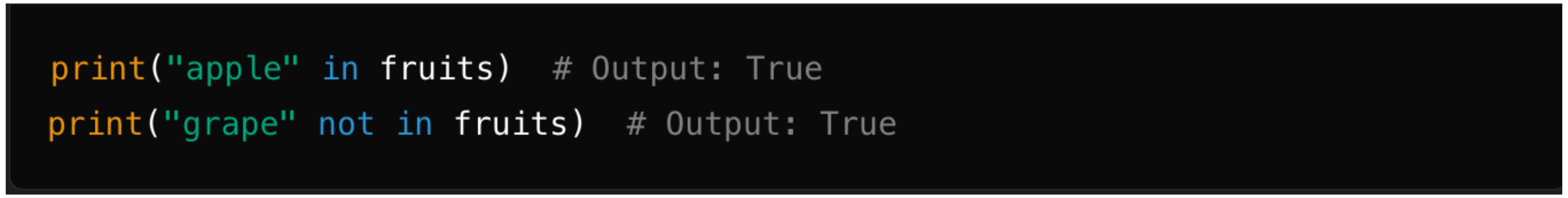
Built-in List Functions and Methods

Length of List:



```
print(len(fruits))  # Output: 3
```

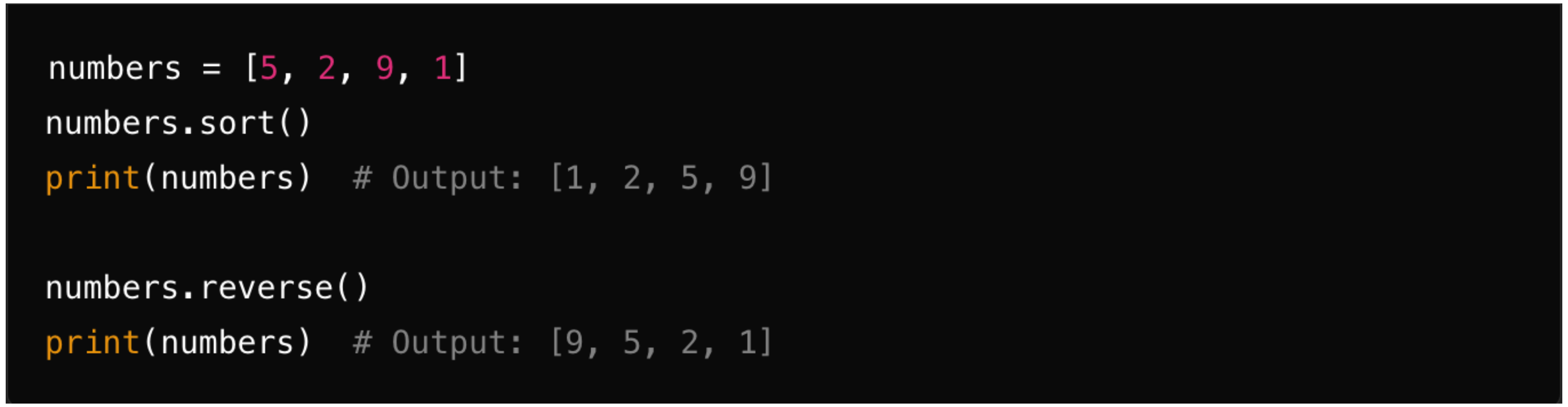
Check Membership:



```
print("apple" in fruits)  # Output: True
print("grape" not in fruits)  # Output: True
```

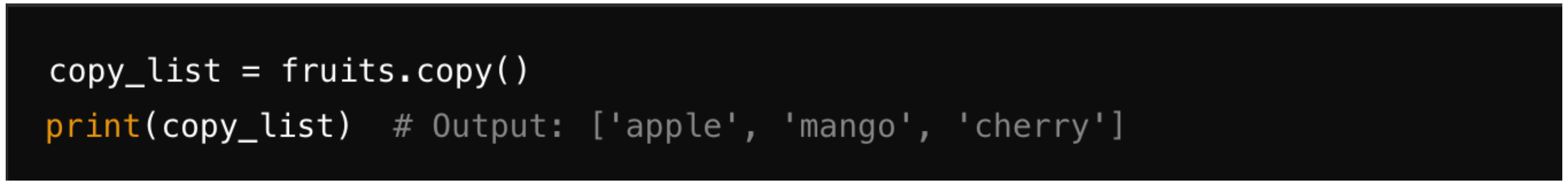
Built-in List Functions and Methods (cont.)

Sorting and Reversing:



```
numbers = [5, 2, 9, 1]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 9]

numbers.reverse()
print(numbers)  # Output: [9, 5, 2, 1]
```

Copying a List:

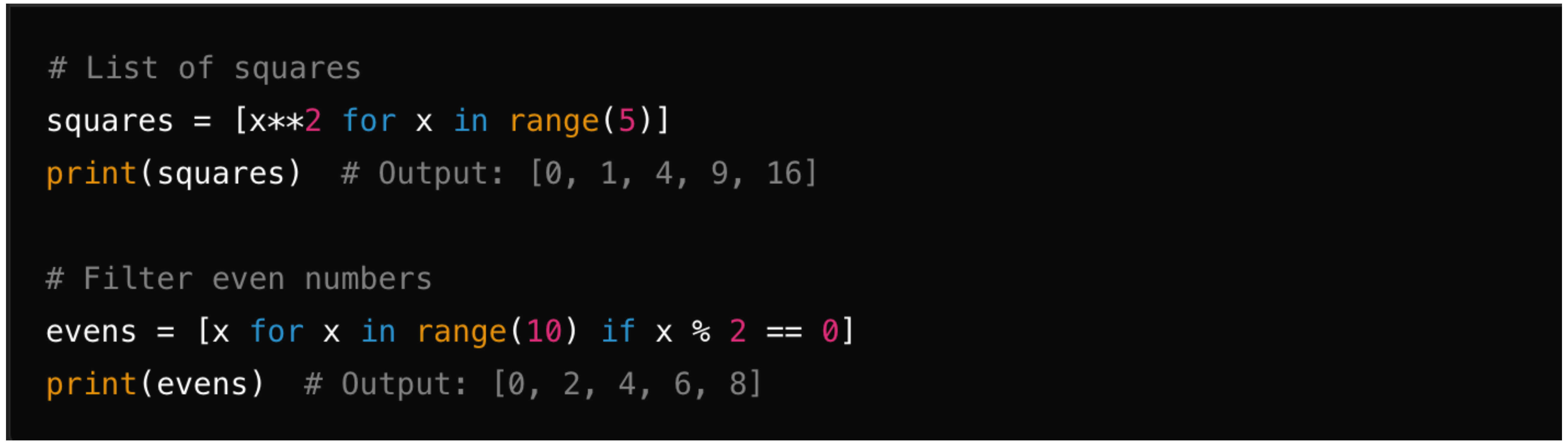


```
copy_list = fruits.copy()
print(copy_list)  # Output: ['apple', 'mango', 'cherry']
```

Examples of Common Use Cases

List Comprehension

Create a new list based on an existing list or sequence:

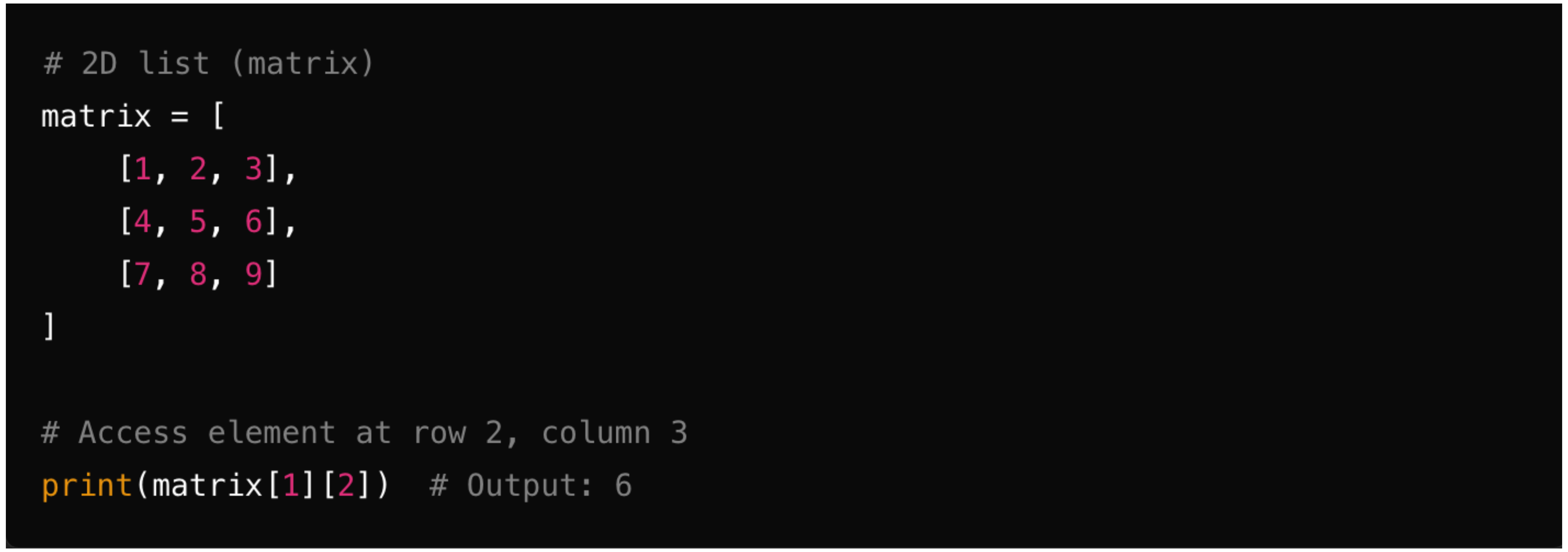


```python
# List of squares
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]

# Filter even numbers
evens = [x for x in range(10) if x % 2 == 0]
print(evens)  # Output: [0, 2, 4, 6, 8]
```

Examples of Common Use Cases (cont.)

Nested Lists

Use lists within lists to represent multi-dimensional data:



```python
# 2D list (matrix)
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access element at row 2, column 3
print(matrix[1][2])  # Output: 6
```

Advantages of Lists

Dynamic: Lists can grow and shrink dynamically as needed.

Versatile: Store any type of data, even combinations of different types.

Built-in Methods: Python provides numerous functions for list operations.

Limitations of Lists

Performance: Lists are slower than arrays (like in NumPy) for numerical operations.

Memory: Lists can consume more memory due to their flexibility.

Exercise 1: Create and Modify a List

Level: Easy

1. Create a list with the following values: [10, 20, 30, 40, 50].
2. Print the list.
3. Change the value at index 2 to 99.
4. Add the number 60 to the end of the list.
5. Print the modified list.

Exercise 2: Filter a List

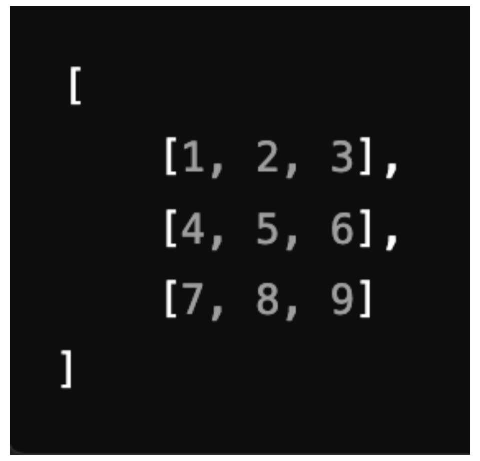
Level: Medium

1. Create a list of integers: [15, 8, 31, 47, 2, 19, 100, 33].
2. Use a loop or list comprehension to create a new list containing only the even numbers from the original list.
3. Print the new list.

Exercise 3: Nested Lists (Matrix Operations)

Level: Hard

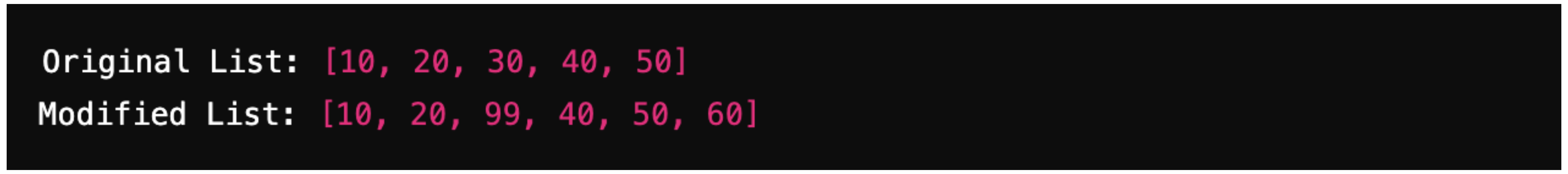
1. Create a 2D list (matrix):



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

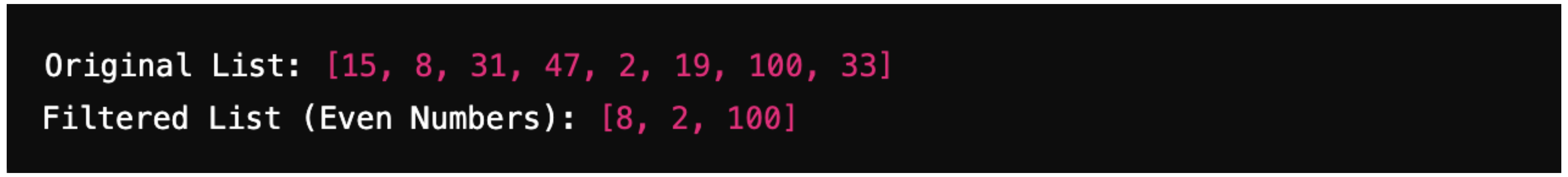
2. Write a program to:
   - Print the elements of the diagonal (e.g., 1, 5, 9).
   - Calculate and print the sum of all elements in the matrix.

Exercise 1: Expected Output



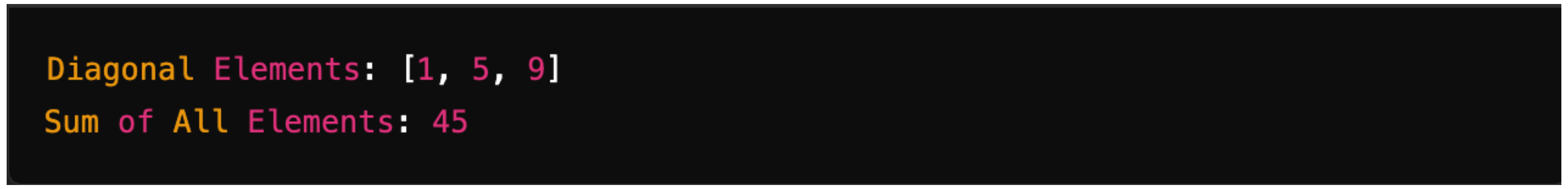
```
Original List: [10, 20, 30, 40, 50]
Modified List: [10, 20, 99, 40, 50, 60]
```

Exercise 2: Expected Output



```
Original List: [15, 8, 31, 47, 2, 19, 100, 33]
Filtered List (Even Numbers): [8, 2, 100]
```

Exercise 3: Expected Output



```
Diagonal Elements: [1, 5, 9]
Sum of All Elements: 45
```

Array

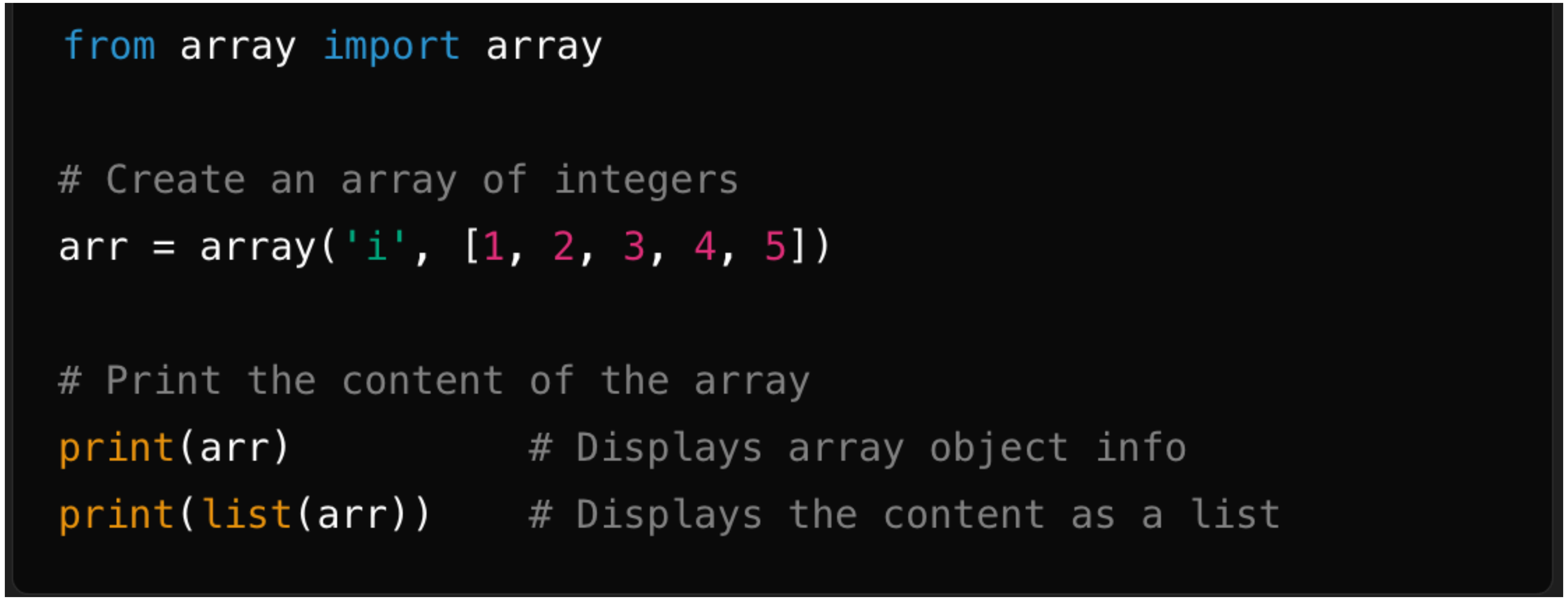
Array

An array is a data structure used to store multiple items of the same data type. While Python's built-in list is often used for similar purposes, the array module provides the array type for optimized array handling when all elements are of the same type. You can also use libraries like NumPy for more advanced array handling.

Array

Creating and Viewing the Content of an Array

1. Using the array Module: The array module is part of Python's standard library.



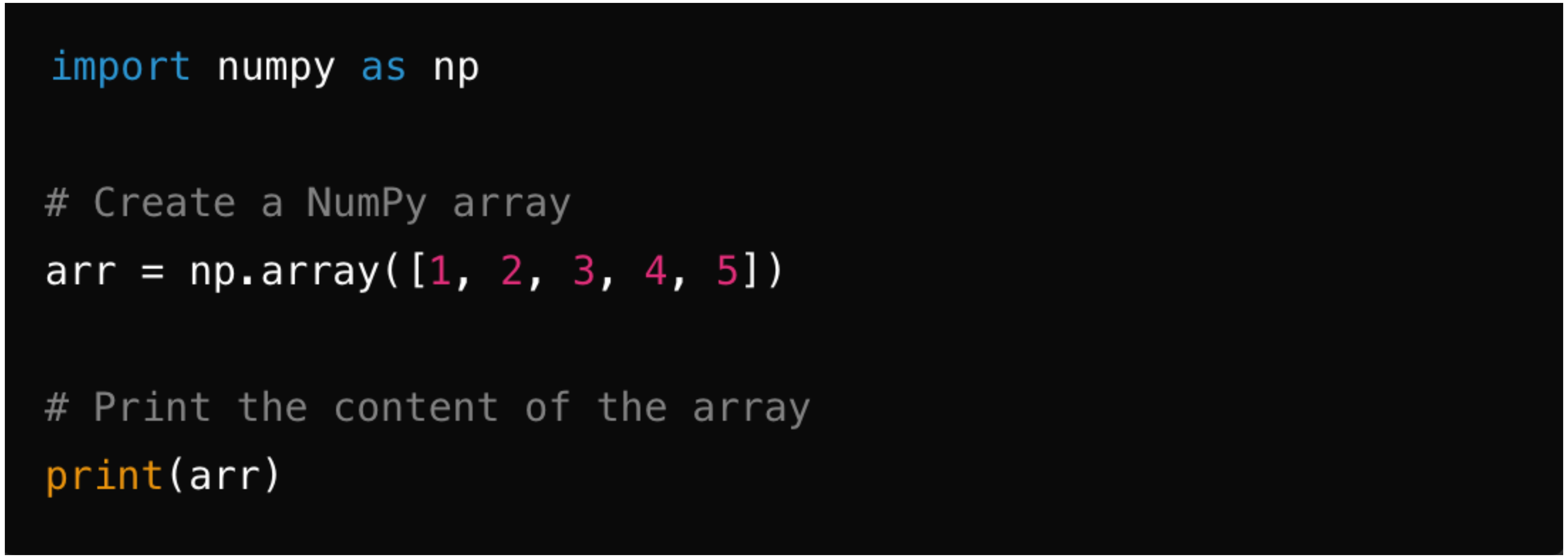
```
from array import array

# Create an array of integers
arr = array('i', [1, 2, 3, 4, 5])

# Print the content of the array
print(arr)          # Displays array object info
print(list(arr))    # Displays the content as a list
```

'i' is the type code for integers.
To view the contents, you can iterate or convert to a list.

Array

Creating and Viewing the Content of an Array

2. Using NumPy Arrays: NumPy is a popular library for numerical computations and array handling.



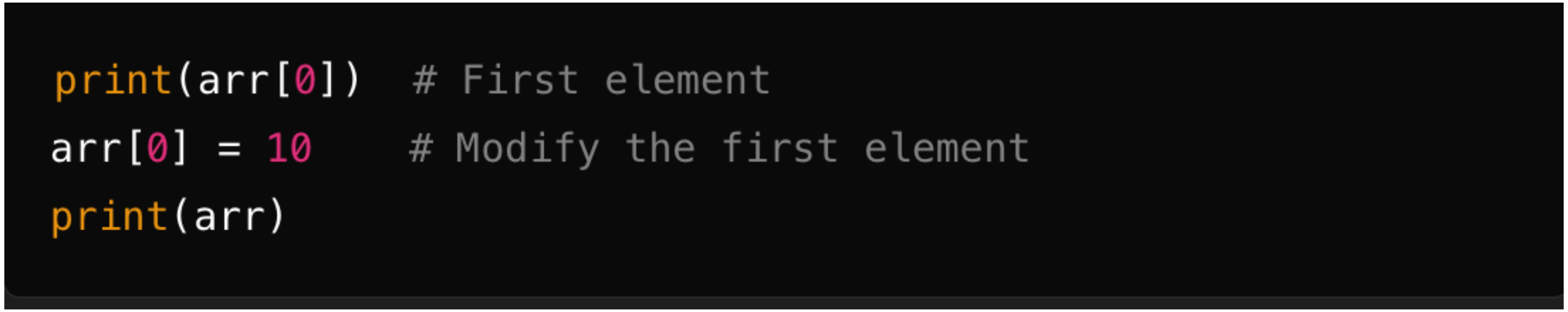
```python
import numpy as np

# Create a NumPy array
arr = np.array([1, 2, 3, 4, 5])

# Print the content of the array
print(arr)
```

NumPy arrays can handle more complex data structures and are highly optimized for operations on large data sets.

Array

Creating and Viewing the Content of an Array

3. Accessing Elements in Arrays: You can access and modify the content of arrays using indexing.

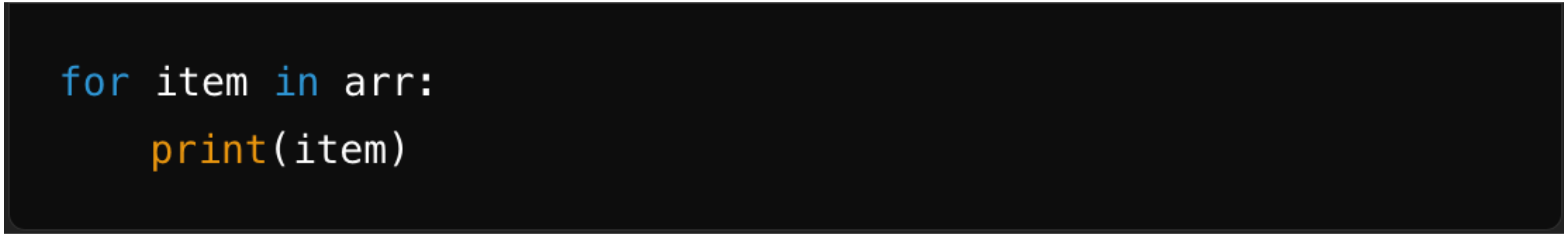


```
print(arr[0])  # First element
arr[0] = 10    # Modify the first element
print(arr)
```

Array

Creating and Viewing the Content of an Array

4. Iterating Through an Array: You can loop through the array to view its elements.



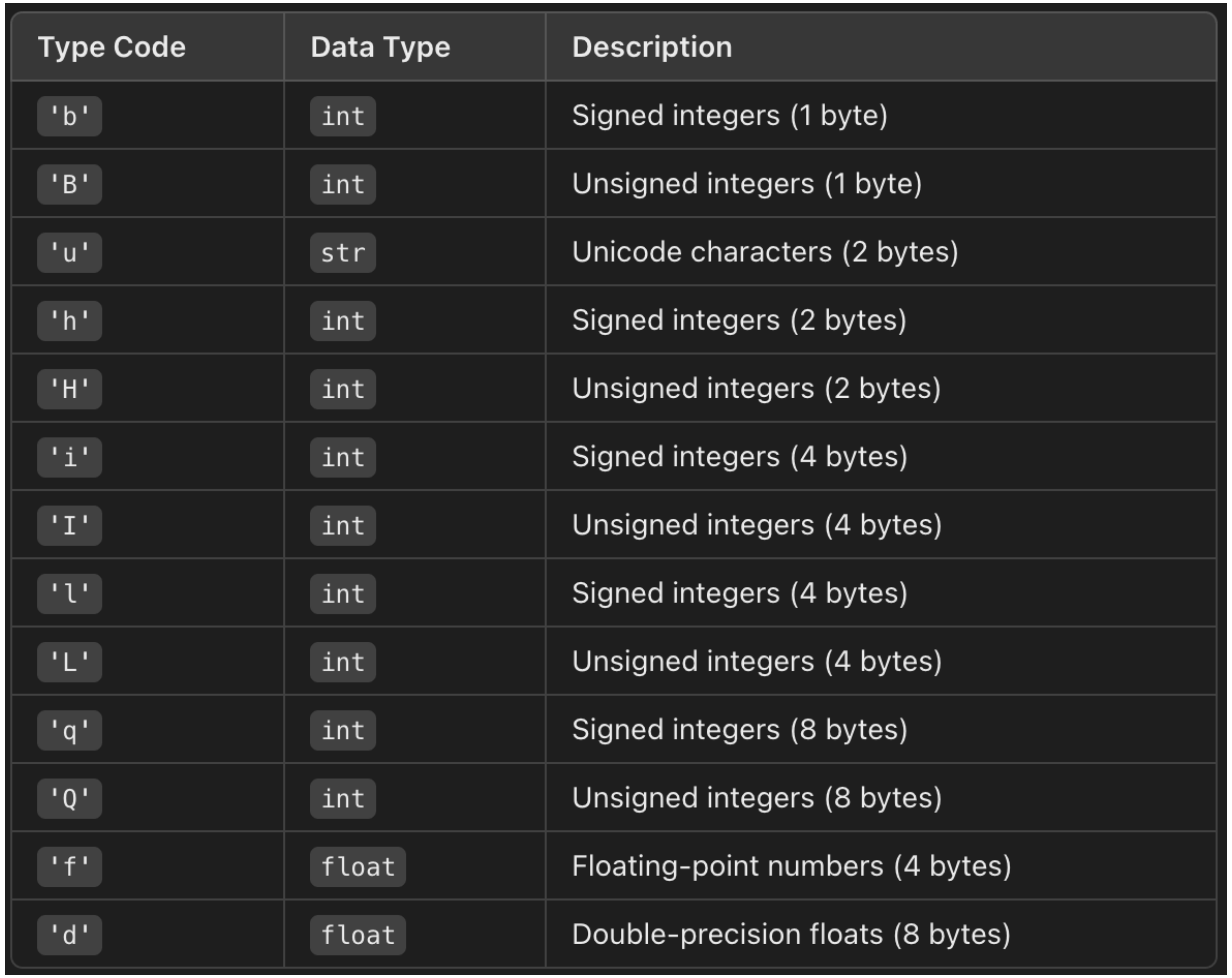
```python
for item in arr:
    print(item)
```

Array’ s Datatypes

In Python, arrays are used to store elements of the **same data type**. The data type depends on the module or library you use. Next pages are the key data types and the respective contexts where they can be stored in arrays.

1. Built-in array Module

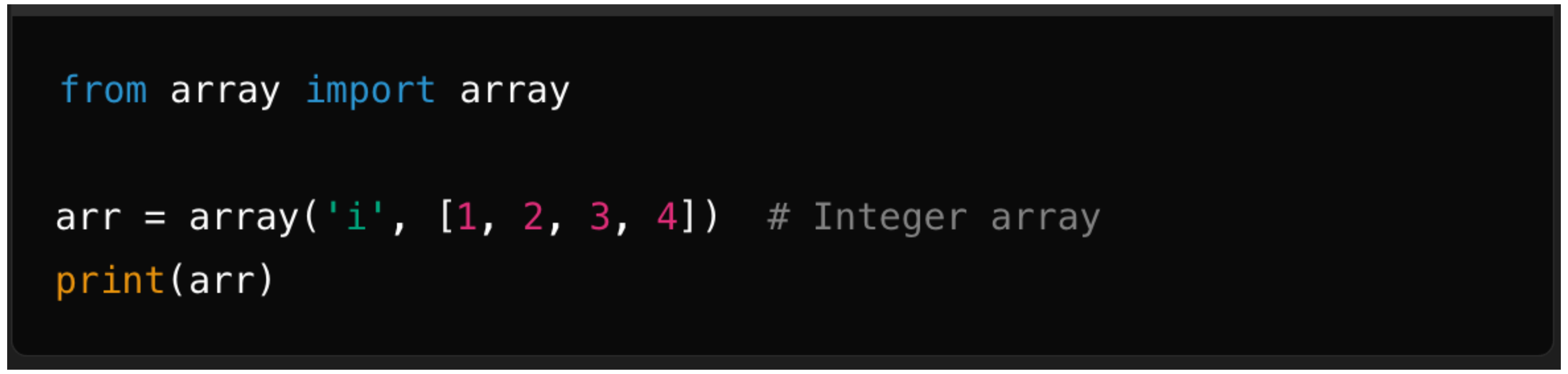
The array module supports only specific data types, which are determined by **type codes**:



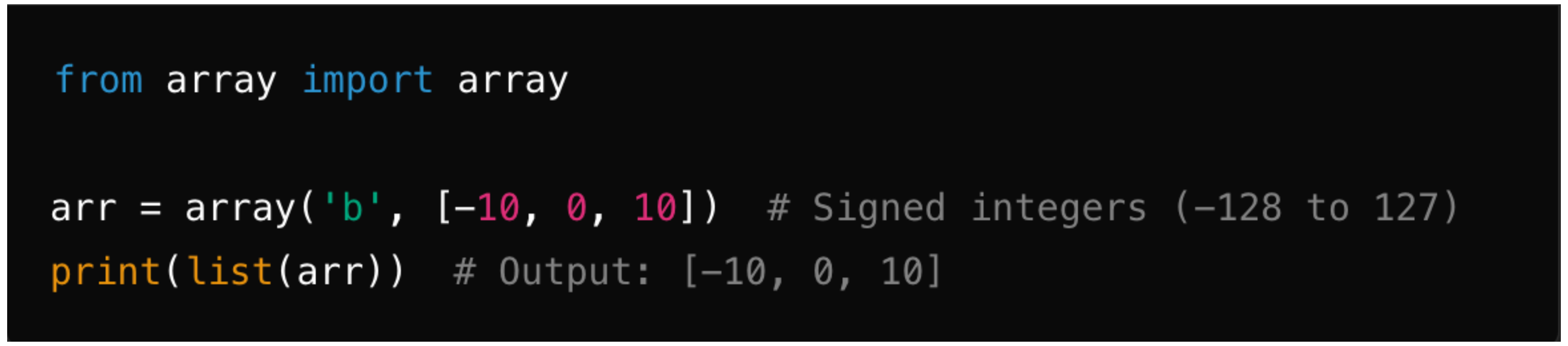
Type Code	Data Type	Description
`'b'`	`int`	Signed integers (1 byte)
`'B'`	`int`	Unsigned integers (1 byte)
`'u'`	`str`	Unicode characters (2 bytes)
`'h'`	`int`	Signed integers (2 bytes)
`'H'`	`int`	Unsigned integers (2 bytes)
`'i'`	`int`	Signed integers (4 bytes)
`'I'`	`int`	Unsigned integers (4 bytes)
`'l'`	`int`	Signed integers (4 bytes)
`'L'`	`int`	Unsigned integers (4 bytes)
`'q'`	`int`	Signed integers (8 bytes)
`'Q'`	`int`	Unsigned integers (8 bytes)
`'f'`	`float`	Floating-point numbers (4 bytes)
`'d'`	`float`	Double-precision floats (8 bytes)

1. Built-in array Module (cont.)

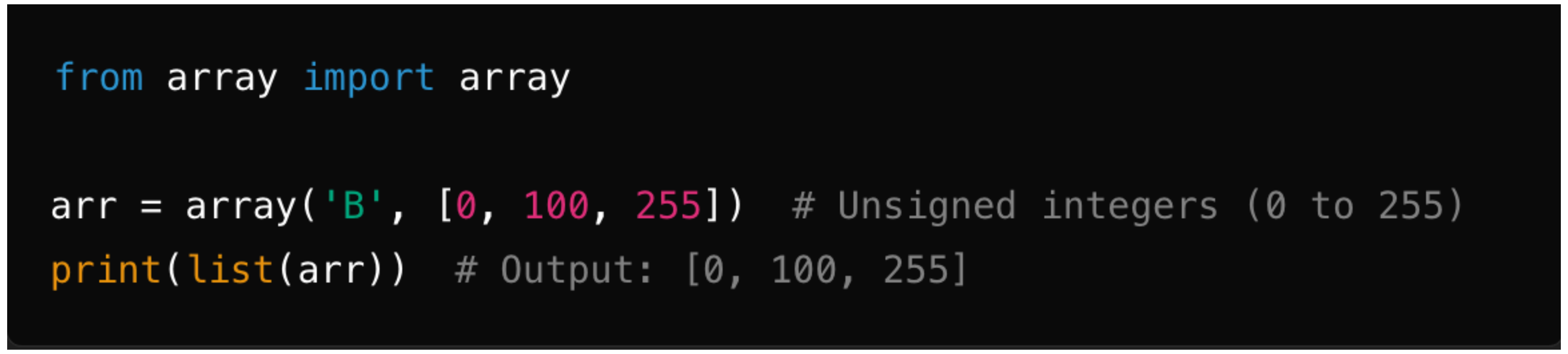
Code Example:



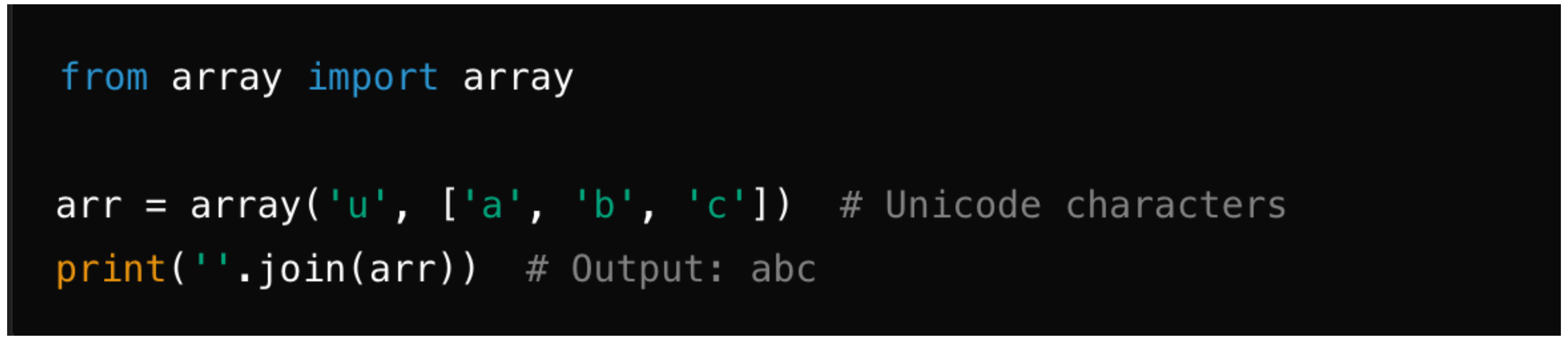
```python
from array import array

arr = array('i', [1, 2, 3, 4])  # Integer array
print(arr)
```



```python
from array import array

arr = array('b', [-10, 0, 10])  # Signed integers (-128 to 127)
print(list(arr))  # Output: [-10, 0, 10]
```



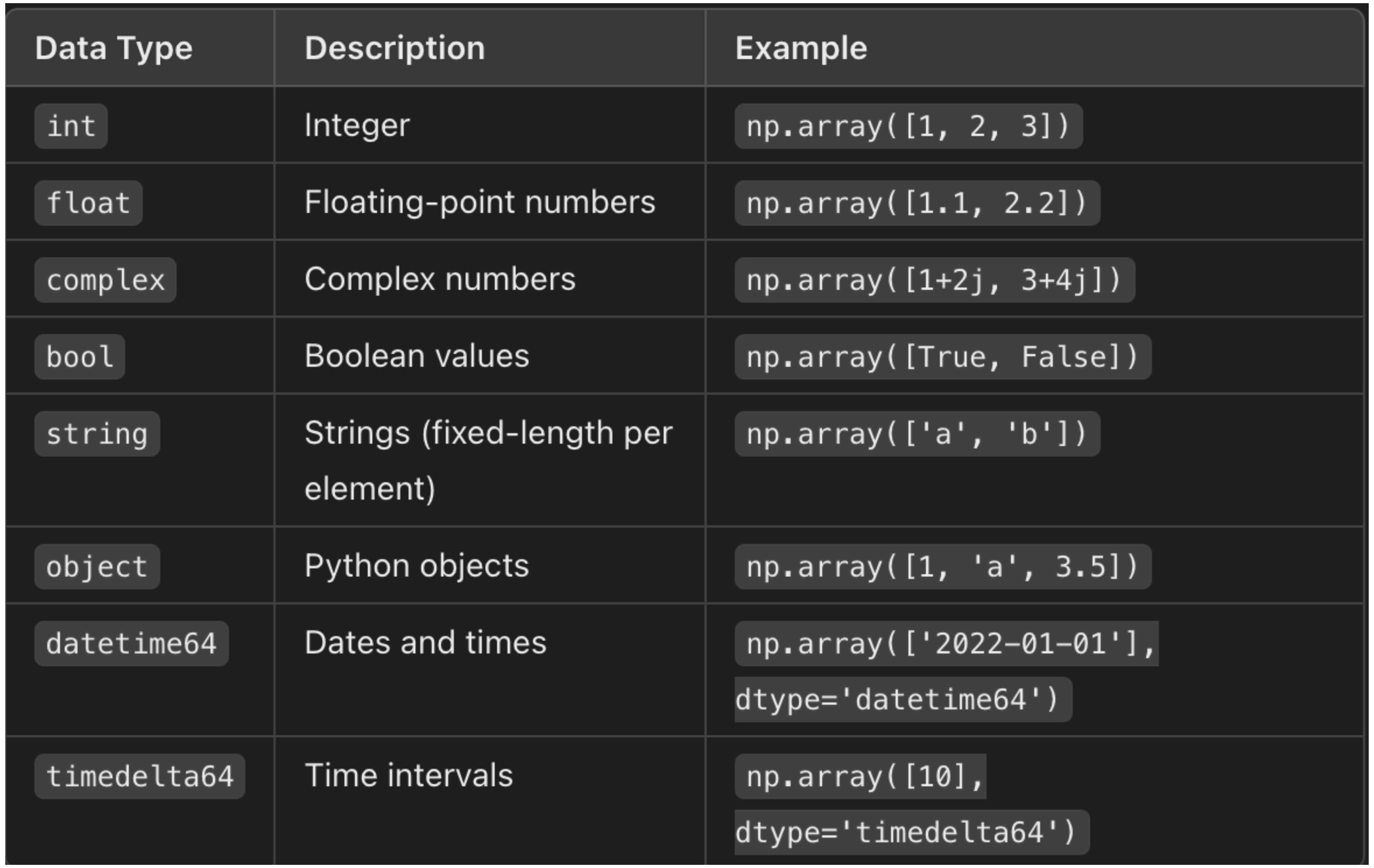
```python
from array import array

arr = array('B', [0, 100, 255])  # Unsigned integers (0 to 255)
print(list(arr))  # Output: [0, 100, 255]
```



```python
from array import array

arr = array('u', ['a', 'b', 'c'])  # Unicode characters
print(''.join(arr))  # Output: abc
```

2. NumPy Arrays

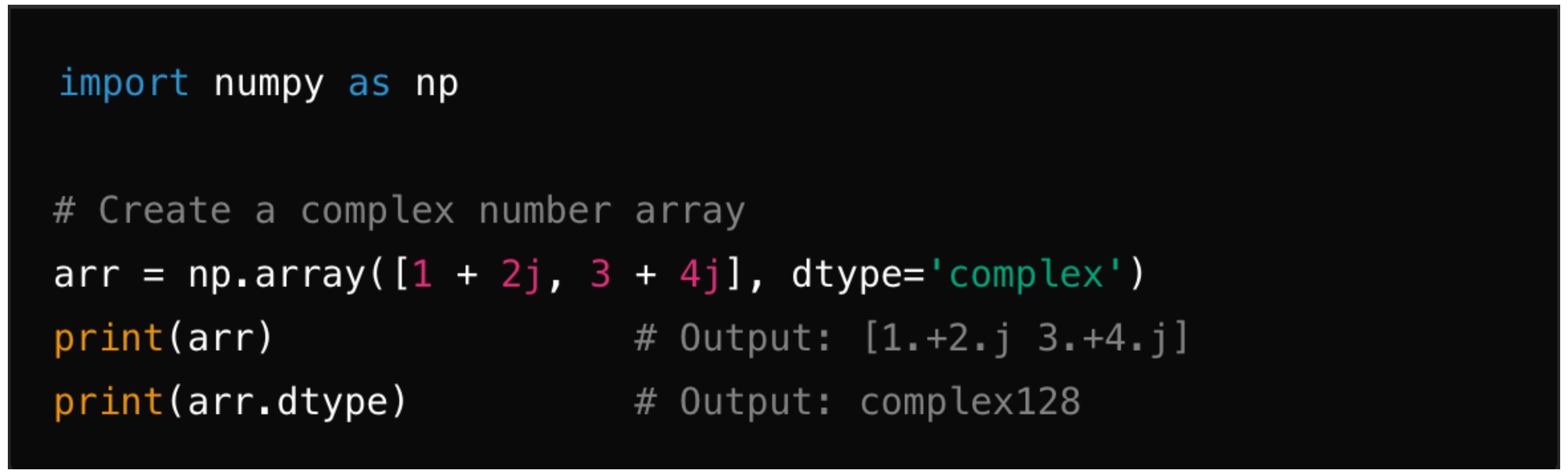
NumPy arrays (numpy.array) are far more flexible and can store a wide range of data types.



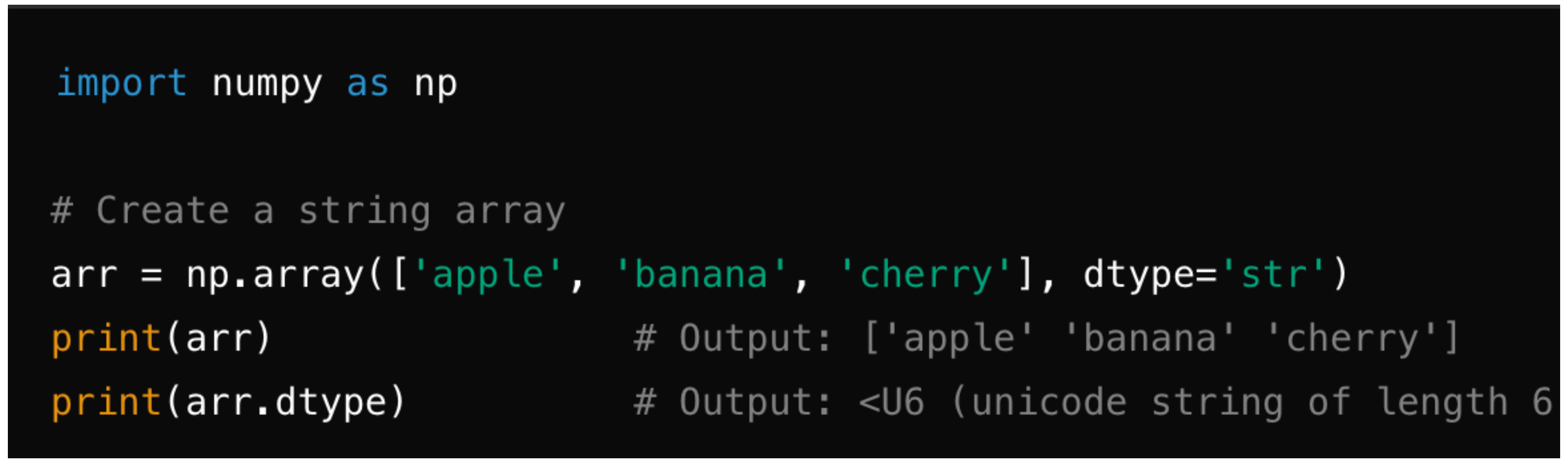
Data Type	Description	Example
`int`	Integer	`np.array([1, 2, 3])`
`float`	Floating-point numbers	`np.array([1.1, 2.2])`
`complex`	Complex numbers	`np.array([1+2j, 3+4j])`
`bool`	Boolean values	`np.array([True, False])`
`string`	Strings (fixed-length per element)	`np.array(['a', 'b'])`
`object`	Python objects	`np.array([1, 'a', 3.5])`
`datetime64`	Dates and times	`np.array(['2022-01-01'], dtype='datetime64')`
`timedelta64`	Time intervals	`np.array([10], dtype='timedelta64')`

2. NumPy Arrays

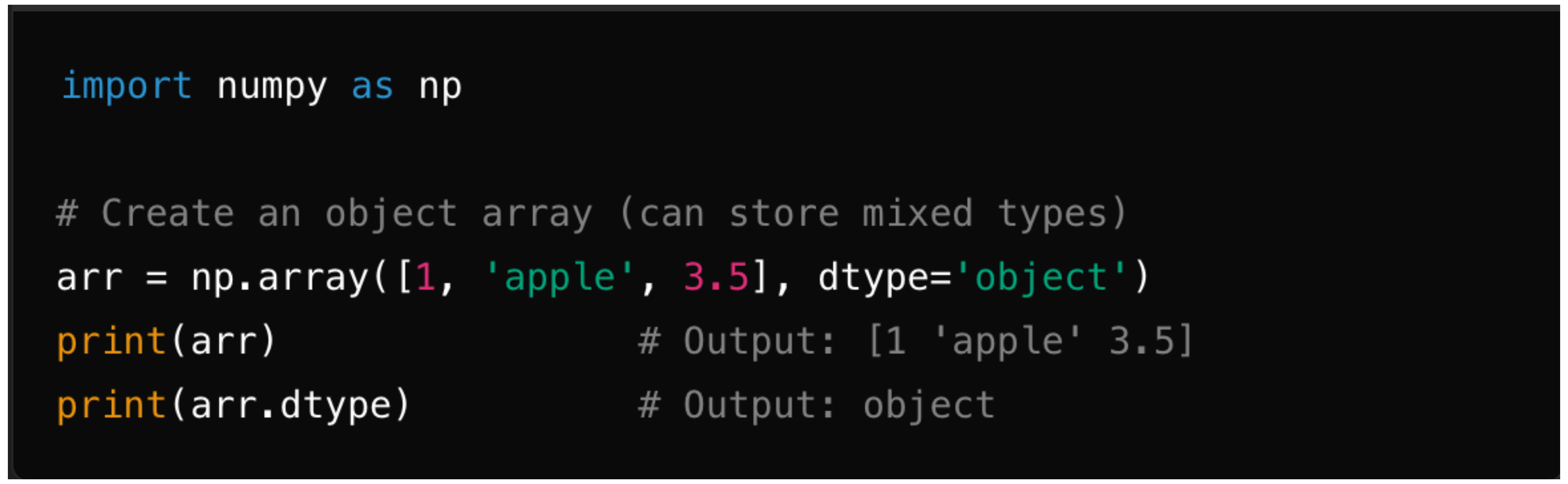
Code Examples



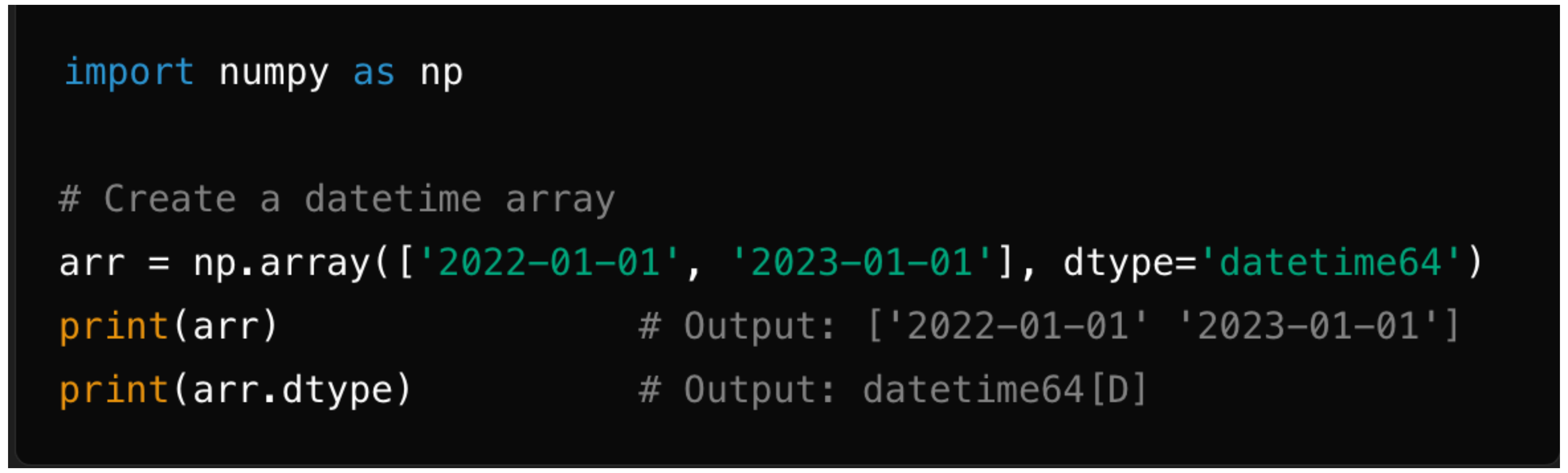
```python
import numpy as np

# Create a complex number array
arr = np.array([1 + 2j, 3 + 4j], dtype='complex')
print(arr)                # Output: [1.+2.j 3.+4.j]
print(arr.dtype)          # Output: complex128
```



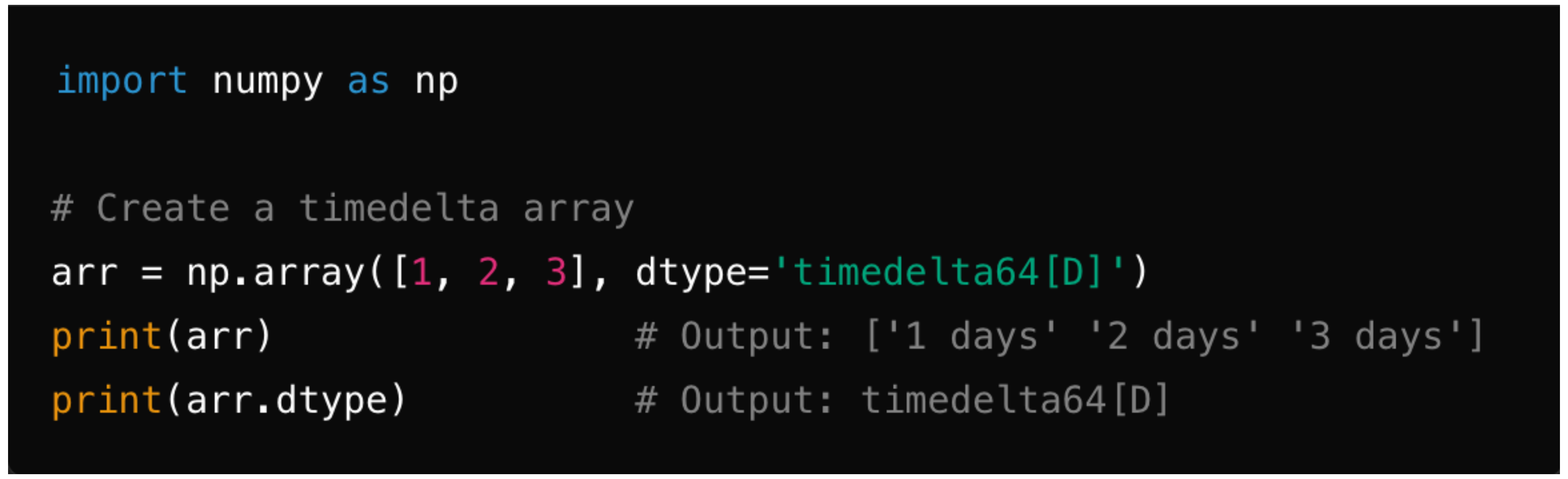
```python
import numpy as np

# Create a string array
arr = np.array(['apple', 'banana', 'cherry'], dtype='str')
print(arr)                # Output: ['apple' 'banana' 'cherry']
print(arr.dtype)          # Output: <U6 (unicode string of length 6
```



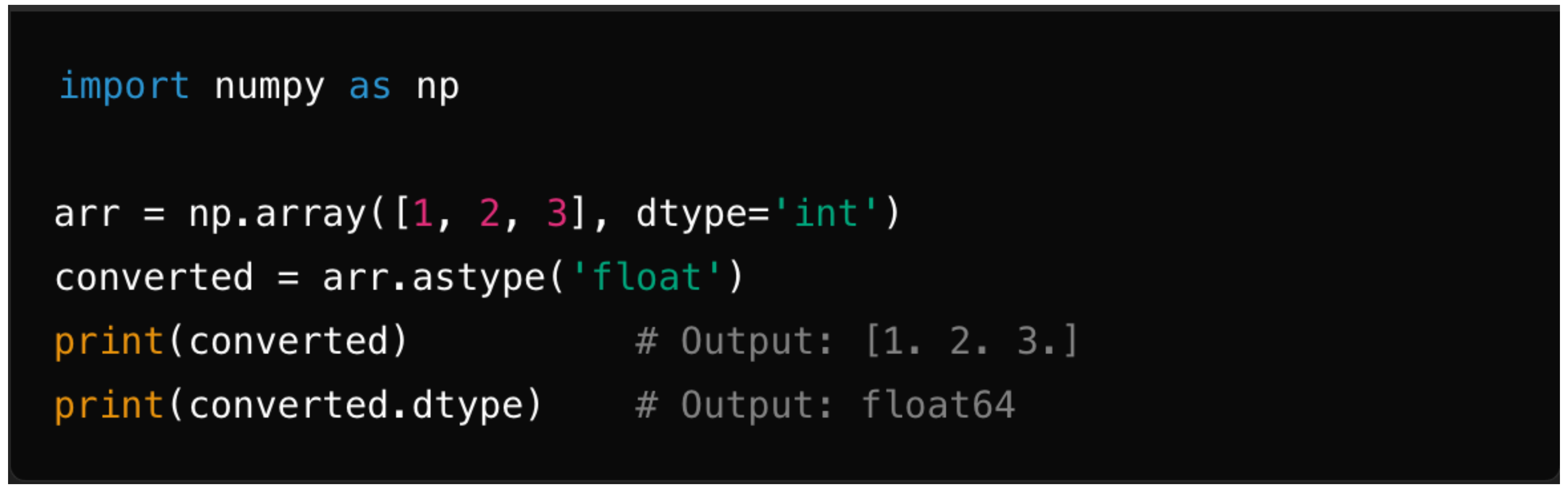
```python
import numpy as np

# Create an object array (can store mixed types)
arr = np.array([1, 'apple', 3.5], dtype='object')
print(arr)                # Output: [1 'apple' 3.5]
print(arr.dtype)          # Output: object
```



```python
import numpy as np

# Create a datetime array
arr = np.array(['2022-01-01', '2023-01-01'], dtype='datetime64')
print(arr)                # Output: ['2022-01-01' '2023-01-01']
print(arr.dtype)          # Output: datetime64[D]
```



```python
import numpy as np

# Create a timedelta array
arr = np.array([1, 2, 3], dtype='timedelta64[D]')
print(arr)                # Output: ['1 days' '2 days' '3 days']
print(arr.dtype)          # Output: timedelta64[D]
```



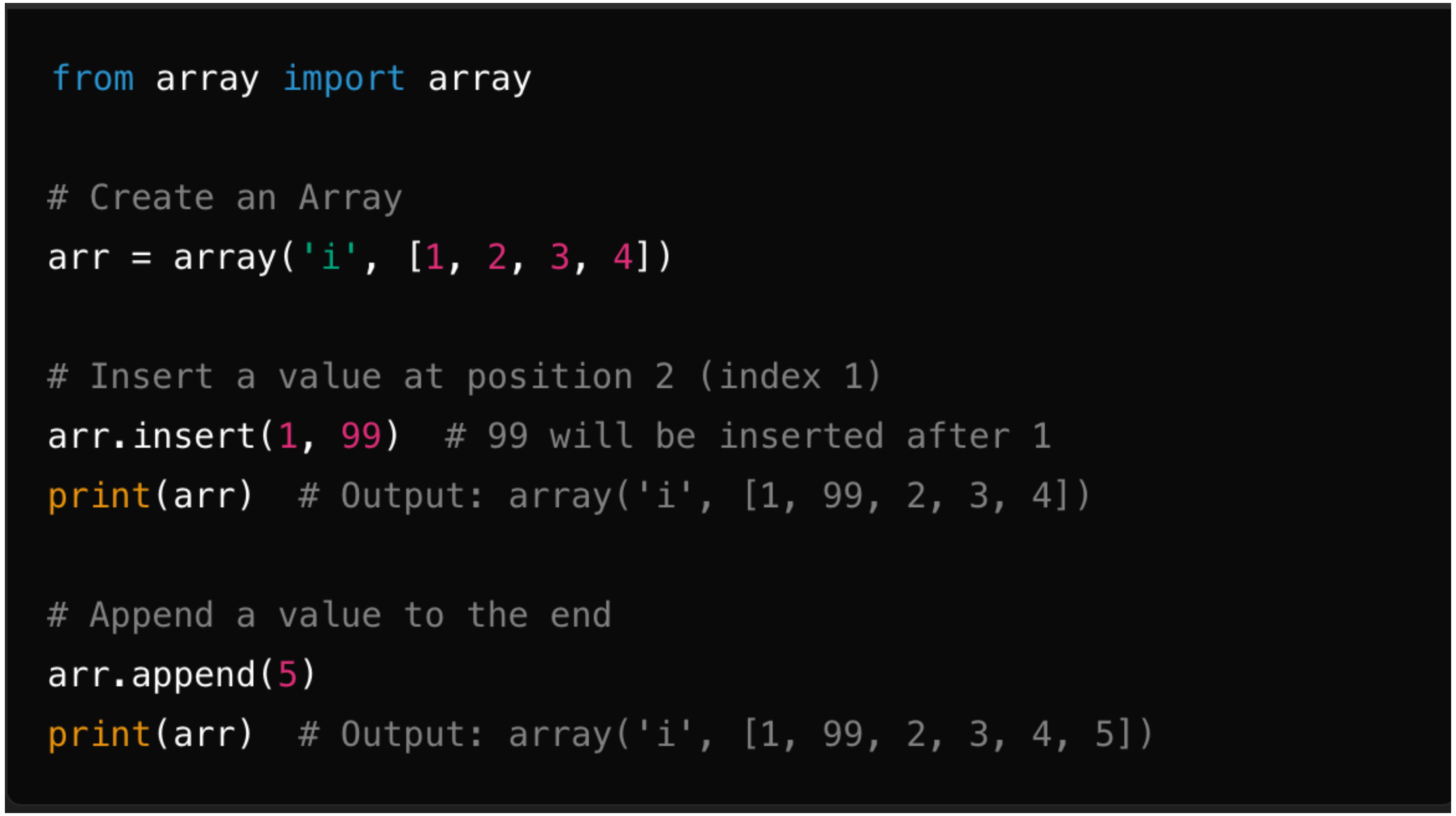
```python
import numpy as np

arr = np.array([1, 2, 3], dtype='int')
converted = arr.astype('float')
print(converted)          # Output: [1. 2. 3.]
print(converted.dtype)    # Output: float64
```

Array Manipulation (Insert, Update and Delete)

Examples of how to perform **Insert**, **Update**, and **Delete** operations on an array using Python's `array` module:

1. Insert Values into an Array

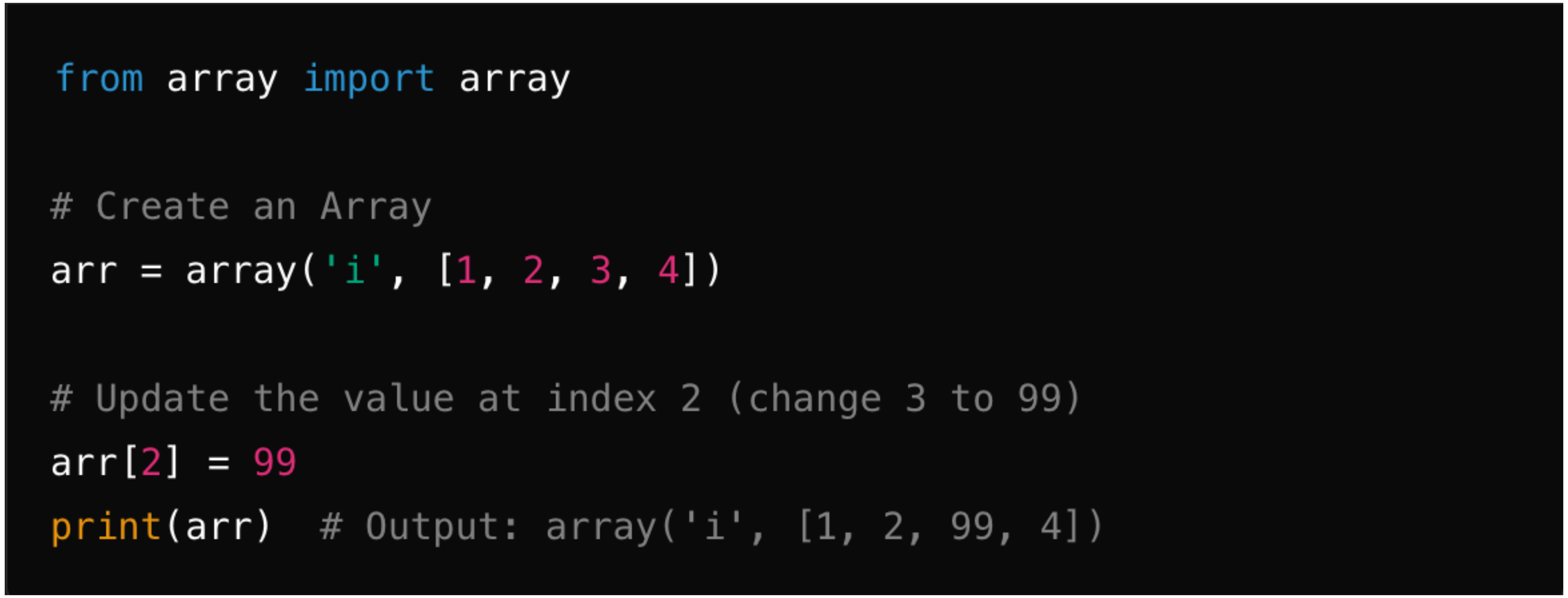
You can insert values at a specific position using the insert() method or append values at the end using the append() method.



```python
from array import array

# Create an Array
arr = array('i', [1, 2, 3, 4])

# Insert a value at position 2 (index 1)
arr.insert(1, 99)  # 99 will be inserted after 1
print(arr)  # Output: array('i', [1, 99, 2, 3, 4])

# Append a value to the end
arr.append(5)
print(arr)  # Output: array('i', [1, 99, 2, 3, 4, 5])
```

2. Update Values in an Array

You can update values in an array by referencing their index and assigning a new value.

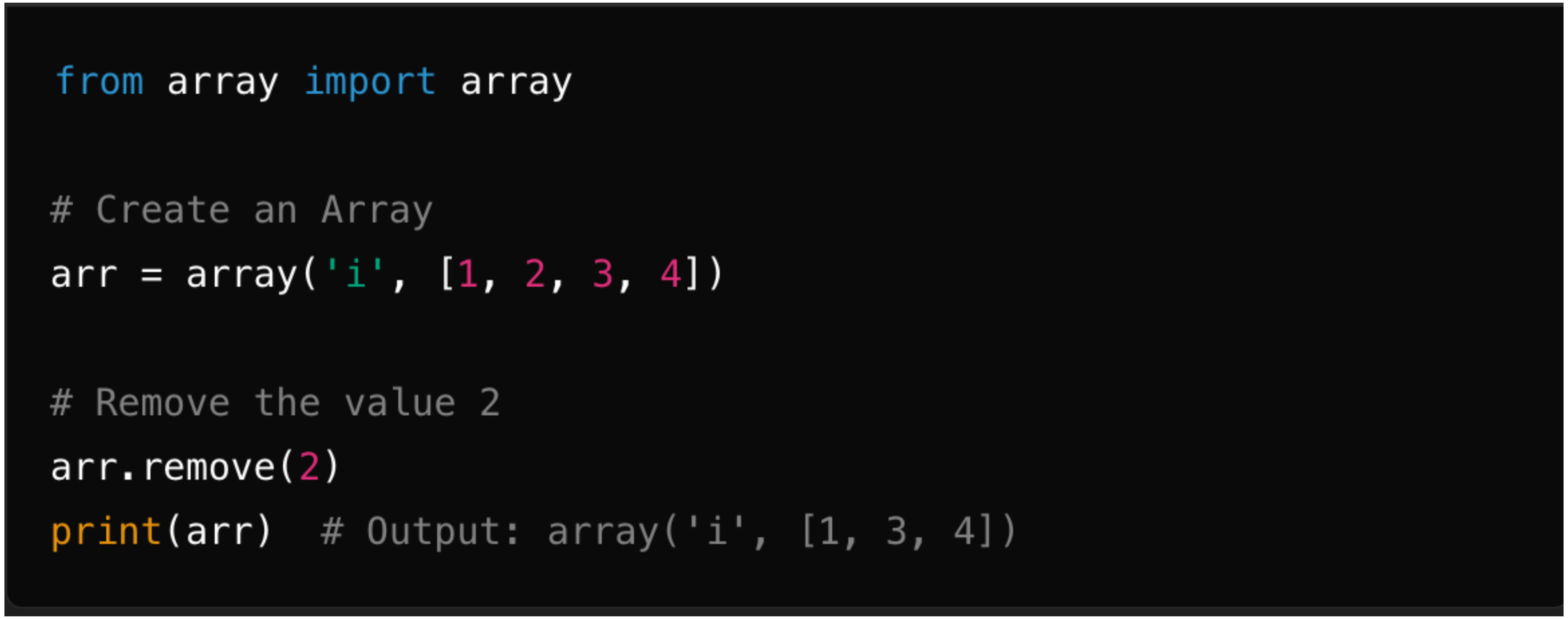


```
from array import array

# Create an Array
arr = array('i', [1, 2, 3, 4])

# Update the value at index 2 (change 3 to 99)
arr[2] = 99
print(arr)  # Output: array('i', [1, 2, 99, 4])
```

3. Delete Values from an Array

You can delete values from an array using the remove() method or the del statement.

3.1 Remove a Specific Value

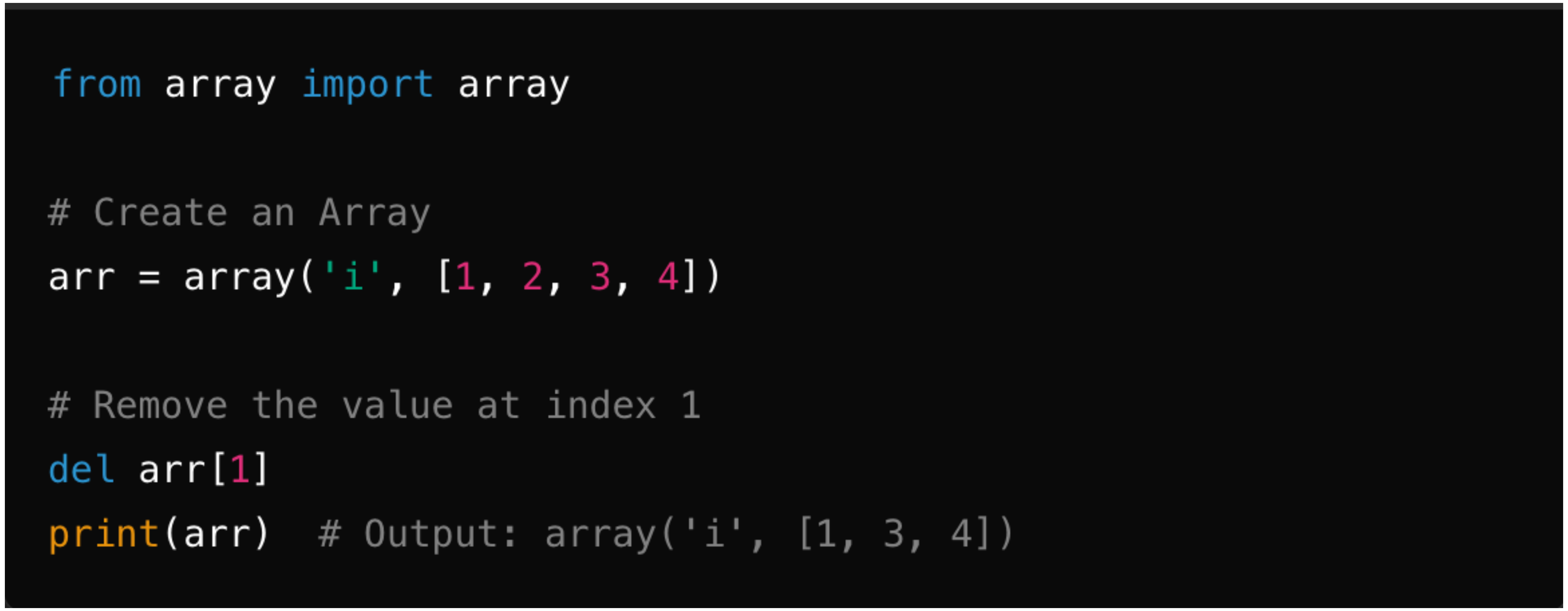


```python
from array import array

# Create an Array
arr = array('i', [1, 2, 3, 4])

# Remove the value 2
arr.remove(2)
print(arr)  # Output: array('i', [1, 3, 4])
```

3. Delete Values from an Array (cont.)

You can delete values from an array using the remove() method or the del statement.

3.2 Remove a Value by Index

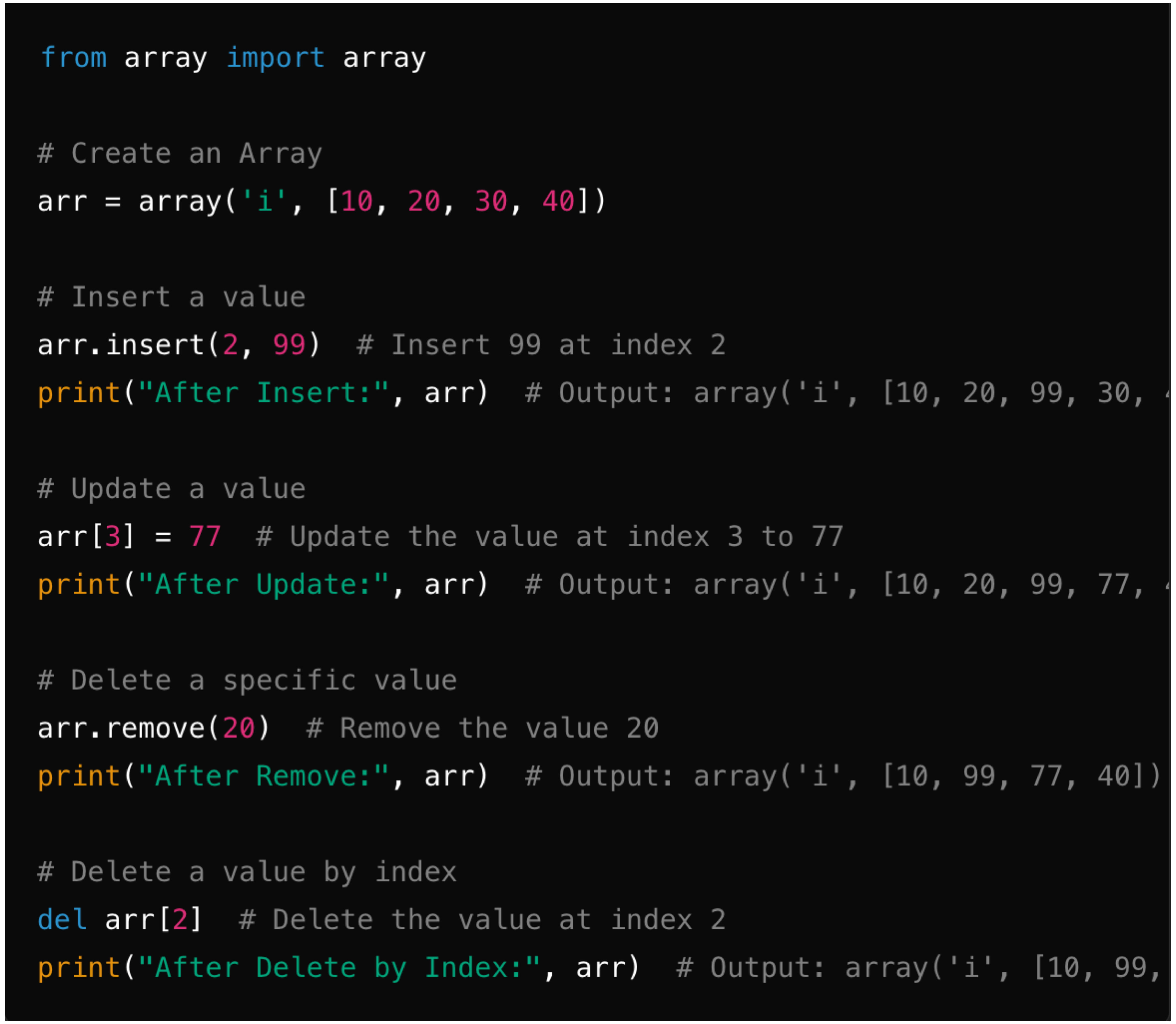


```
from array import array

# Create an Array
arr = array('i', [1, 2, 3, 4])

# Remove the value at index 1
del arr[1]
print(arr)  # Output: array('i', [1, 3, 4])
```

3. Delete Values from an Array (cont.)

You can delete values from an array using the remove() method or the del statement.

Combine Example



```
from array import array

# Create an Array
arr = array('i', [10, 20, 30, 40])

# Insert a value
arr.insert(2, 99)  # Insert 99 at index 2
print("After Insert:", arr)  # Output: array('i', [10, 20, 99, 30,

# Update a value
arr[3] = 77  # Update the value at index 3 to 77
print("After Update:", arr)  # Output: array('i', [10, 20, 99, 77,

# Delete a specific value
arr.remove(20)  # Remove the value 20
print("After Remove:", arr)  # Output: array('i', [10, 99, 77, 40])

# Delete a value by index
del arr[2]  # Delete the value at index 2
print("After Delete by Index:", arr)  # Output: array('i', [10, 99,
```

Exercise 1: Insert and Access Elements

Level: Easy

Write a program to:

1. Create an array of integers with values [10, 20, 30, 40, 50].
2. Insert the value 25 at index 2.
3. Print the entire array.
4. Print the element at index 3.

Exercise 2: Update and Delete Elements

Level: Medium

Write a program to:

1. Create an array of signed integers with values [-10, -20, -30, -40, -50].
2. Update the value at index 1 to -15.
3. Delete the value -40 from the array.
4. Print the final array

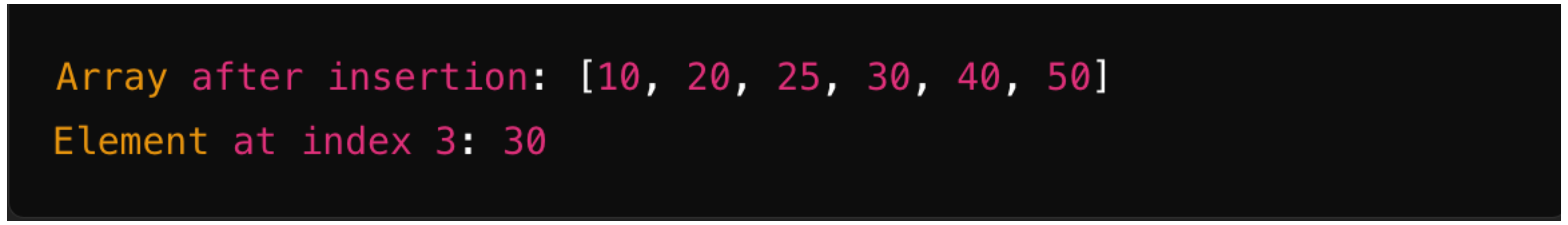
Exercise 3: Array Manipulation with User Input

Level: Hard

Write a program to:

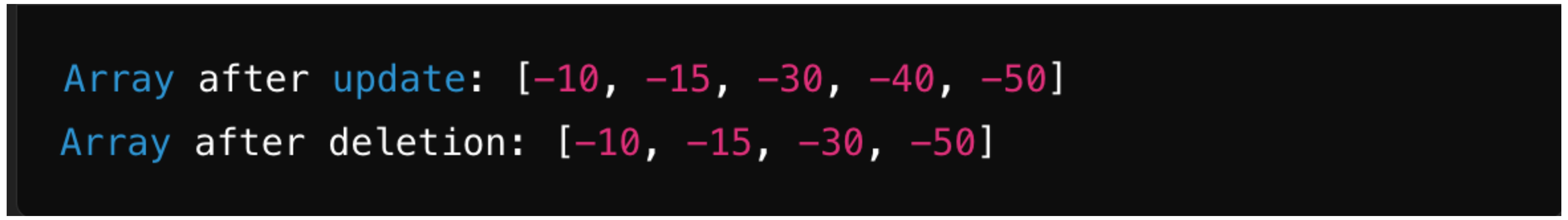
1. Ask the user to input n numbers to create an integer array.
2. Insert the number 99 at the second position in the array.
3. Remove all occurrences of the smallest number in the array.
4. Print the final array.

Exercise 1: Expected Output



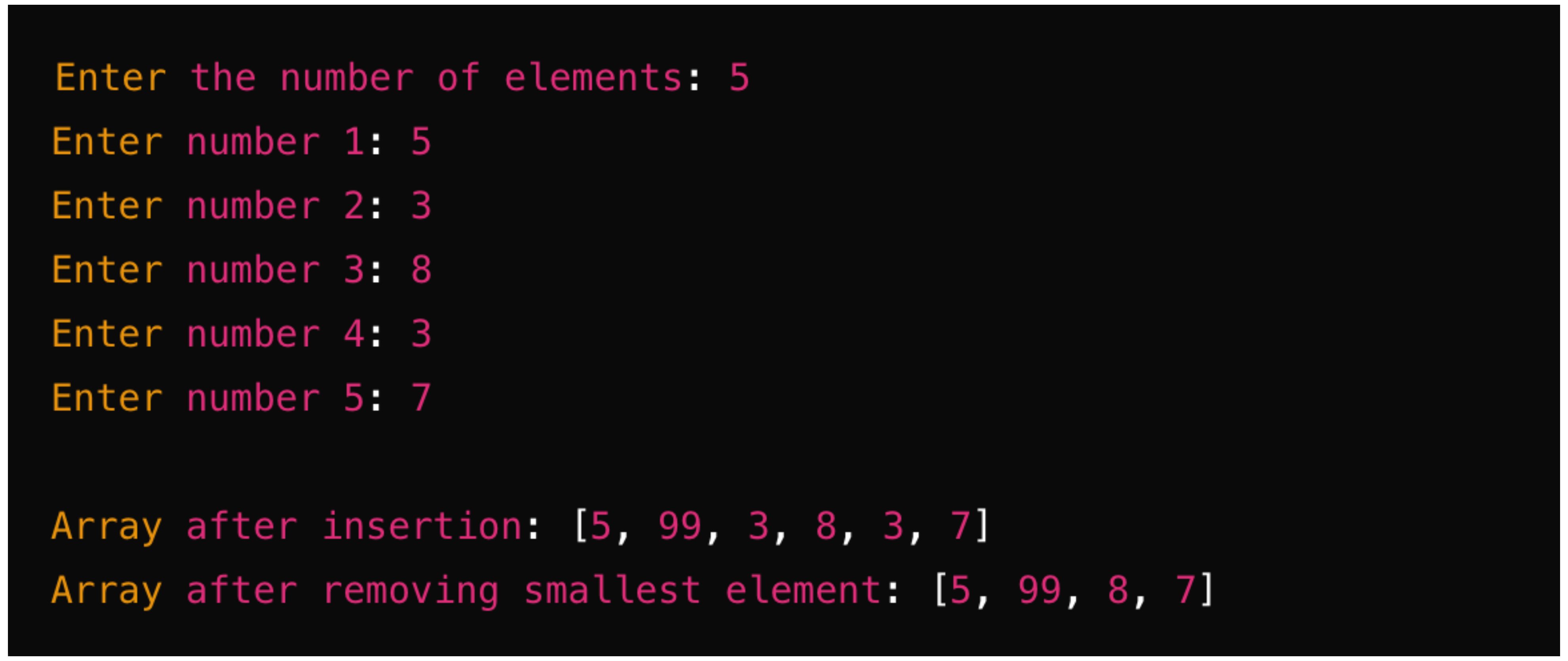
```
Array after insertion: [10, 20, 25, 30, 40, 50]
Element at index 3: 30
```

Exercise 2: Expected Output



```
Array after update: [-10, -15, -30, -40, -50]
Array after deletion: [-10, -15, -30, -50]
```

Exercise 3: Expected Output



```
Enter the number of elements: 5
Enter number 1: 5
Enter number 2: 3
Enter number 3: 8
Enter number 4: 3
Enter number 5: 7

Array after insertion: [5, 99, 3, 8, 3, 7]
Array after removing smallest element: [5, 99, 8, 7]
```

Two-Dimensional Array (2D Array)

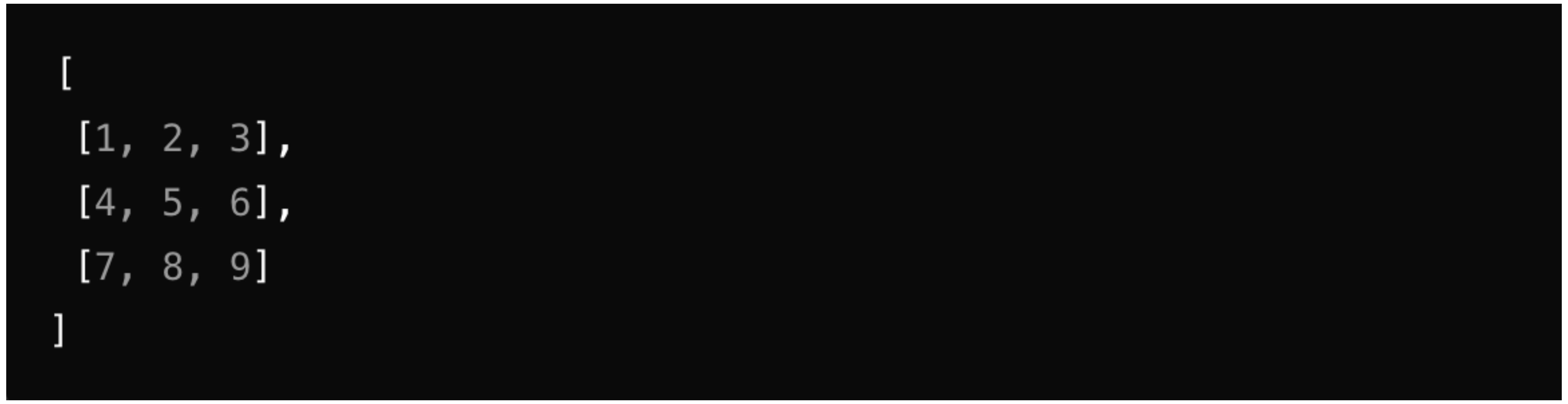
A **Two-Dimensional array (2D Array)** is a data structure that stores elements in a grid format, with rows and columns. It can be thought of as an array of arrays, where each sub-array represents a row of data.

Structure of a 2D Array

A 2D array has:

- **Rows**: The horizontal groupings of elements.
- **Columns**: The vertical groupings of elements.

For example, consider a 2D array:



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

Here:

- It has **3 rows** and **3 columns**.
- The element at the second row and third column is 6.

Uses of a 2D Array

2D arrays are commonly used in:

1. **Matrices**: Representing mathematical matrices for algebraic computations.
2. **Tabular Data**: Storing data in rows and columns like a spreadsheet.
3. **Grids**: Representing grids in games or maps.
4. **Images**: Storing pixel intensity values in image processing.

Characteristics of a 2D Array

Dimensions: A 2D array is indexed by two indices, representing rows and columns. Access element at row i and column j as array[i][j].

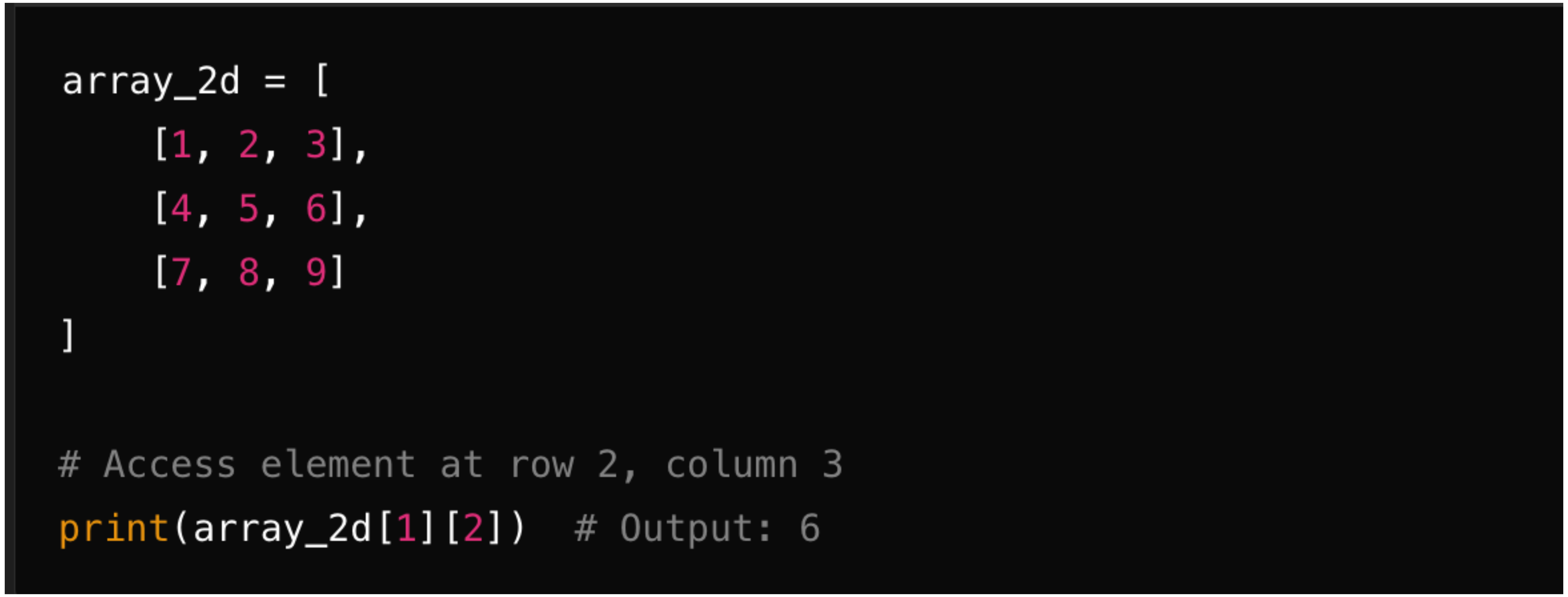
Fixed Size: Traditional arrays (like in C/C++) require pre-defined sizes for rows and columns. In Python, however, arrays are dynamic.

Homogeneous Elements: In libraries like NumPy, all elements in the array must have the same data type.

Representation in Python

Using Lists

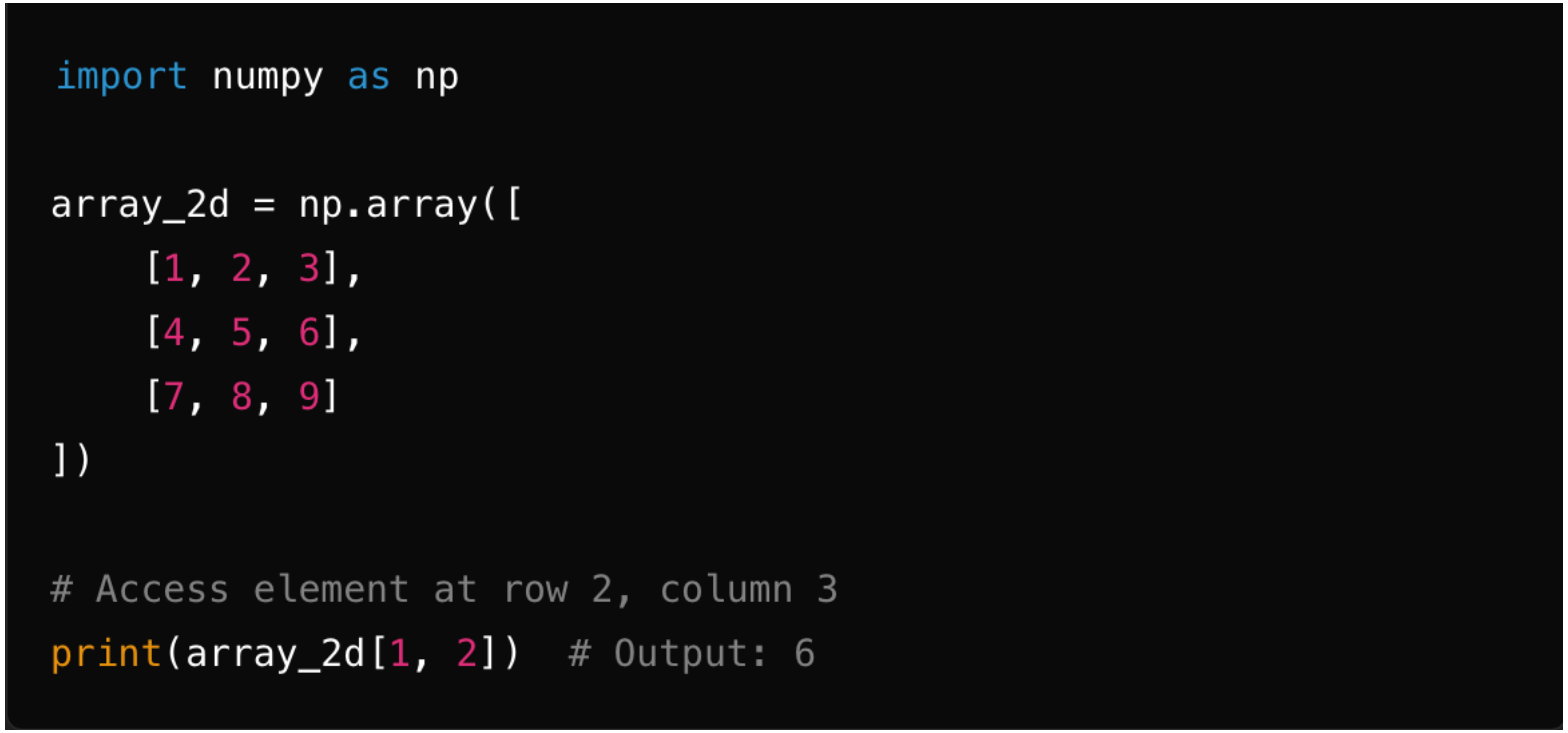
In Python, a 2D array can be represented as a list of lists:



```python
array_2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access element at row 2, column 3
print(array_2d[1][2])  # Output: 6
```

Using NumPy

NumPy provides an efficient way to handle 2D arrays:



```
import numpy as np

array_2d = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Access element at row 2, column 3
print(array_2d[1, 2])  # Output: 6
```

Advantages of a 2D Array

1. **Organized Data:** Makes it easy to work with grid or table-like data.
2. **Efficient Access:** Direct indexing provides quick access to any element.
3. **Scalability:** 2D arrays can represent larger data efficiently, especially using libraries like NumPy.

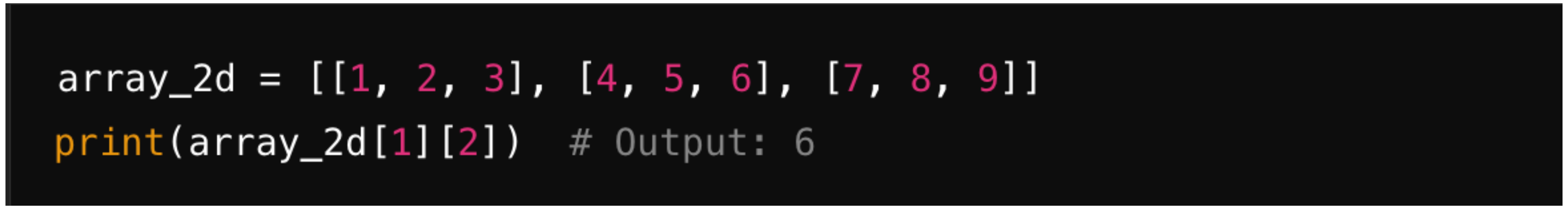
Limitations of 2D Arrays

1. **Memory Usage:** For large arrays, memory consumption can be significant.
2. **Homogeneity:** In libraries like NumPy, all elements must be of the same type.
3. **Limited by Dimensions:** Cannot directly represent data with more than two dimensions (use multi-dimensional arrays for this).

Key Operations in 2D Arrays

1. Accessing Elements

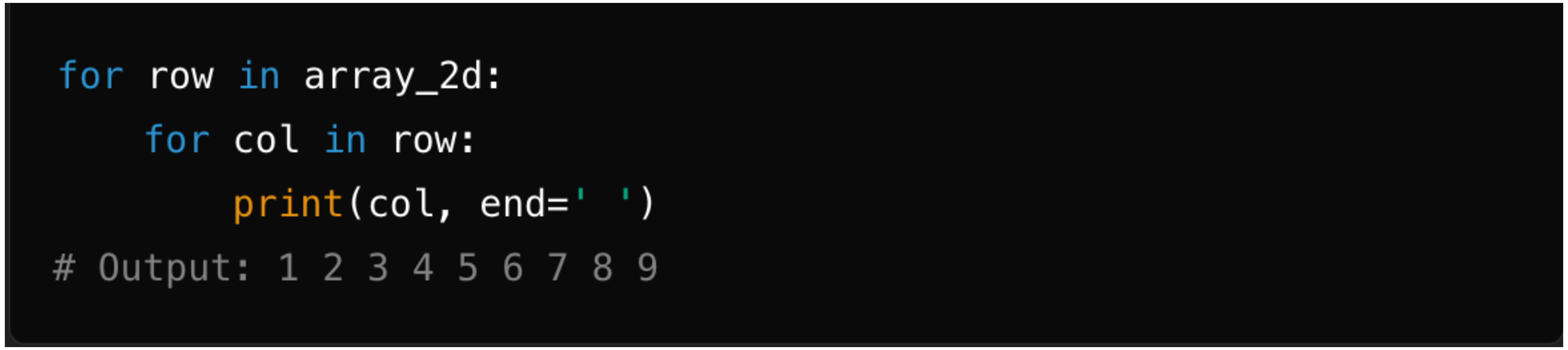
Access an element using two indices: row index and column index.



```
array_2d = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(array_2d[1][2])  # Output: 6
```

Key Operations in 2D Arrays (cont.)

2. Traversing a 2D Array

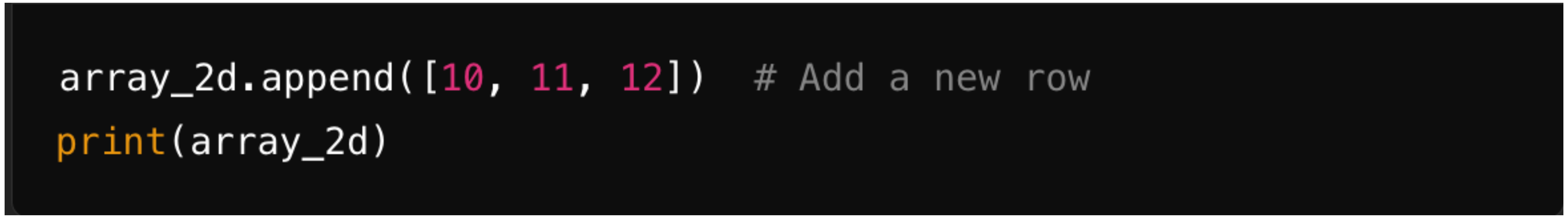


```
for row in array_2d:
    for col in row:
        print(col, end=' ')
# Output: 1 2 3 4 5 6 7 8 9
```

Key Operations in 2D Arrays (cont.)

3. Adding Rows or Columns

Python lists allow dynamic resizing:

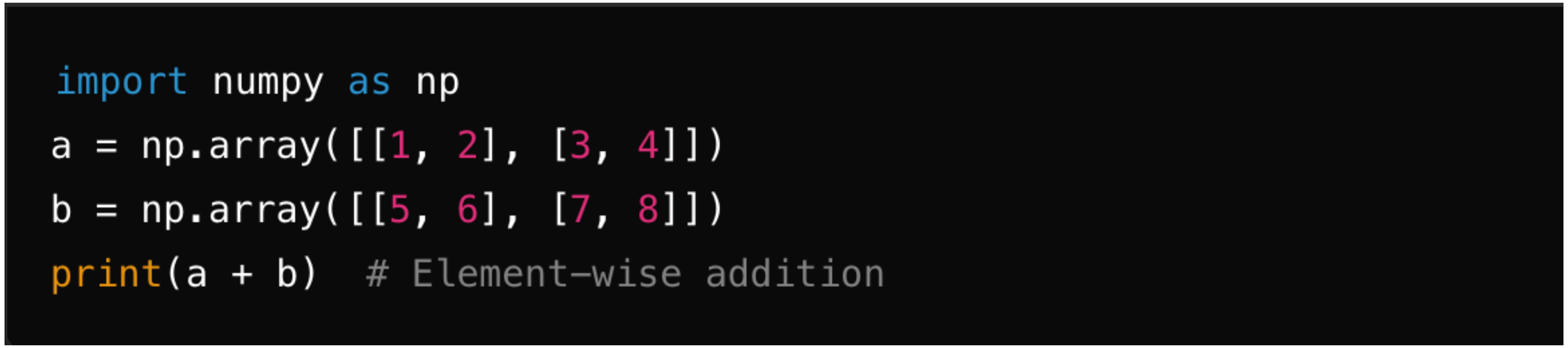


```
array_2d.append([10, 11, 12])  # Add a new row
print(array_2d)
```

Key Operations in 2D Arrays (cont.)

4. Performing Mathematical Operations

NumPy enables matrix-like operations:



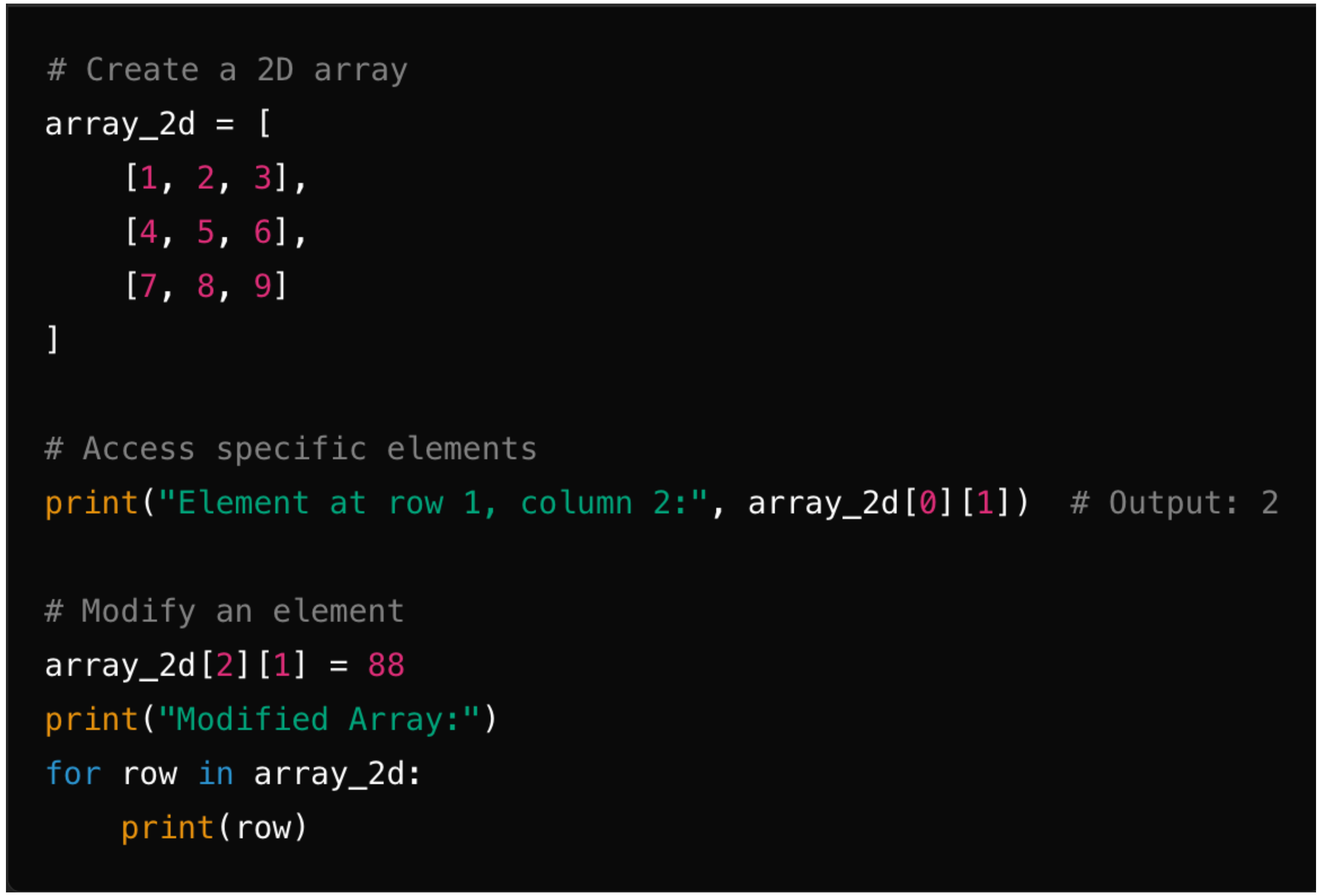
```python
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(a + b)  # Element-wise addition
```

Real-Life Applications of 2D Arrays

1. **Spreadsheets:** Representing rows and columns in tools like Excel.
2. **Image Representation:** Pixels stored as intensity values.
3. **Game Boards:** Representing grids in games like chess or Sudoku.

Examples of how to use a 2D array in Python:

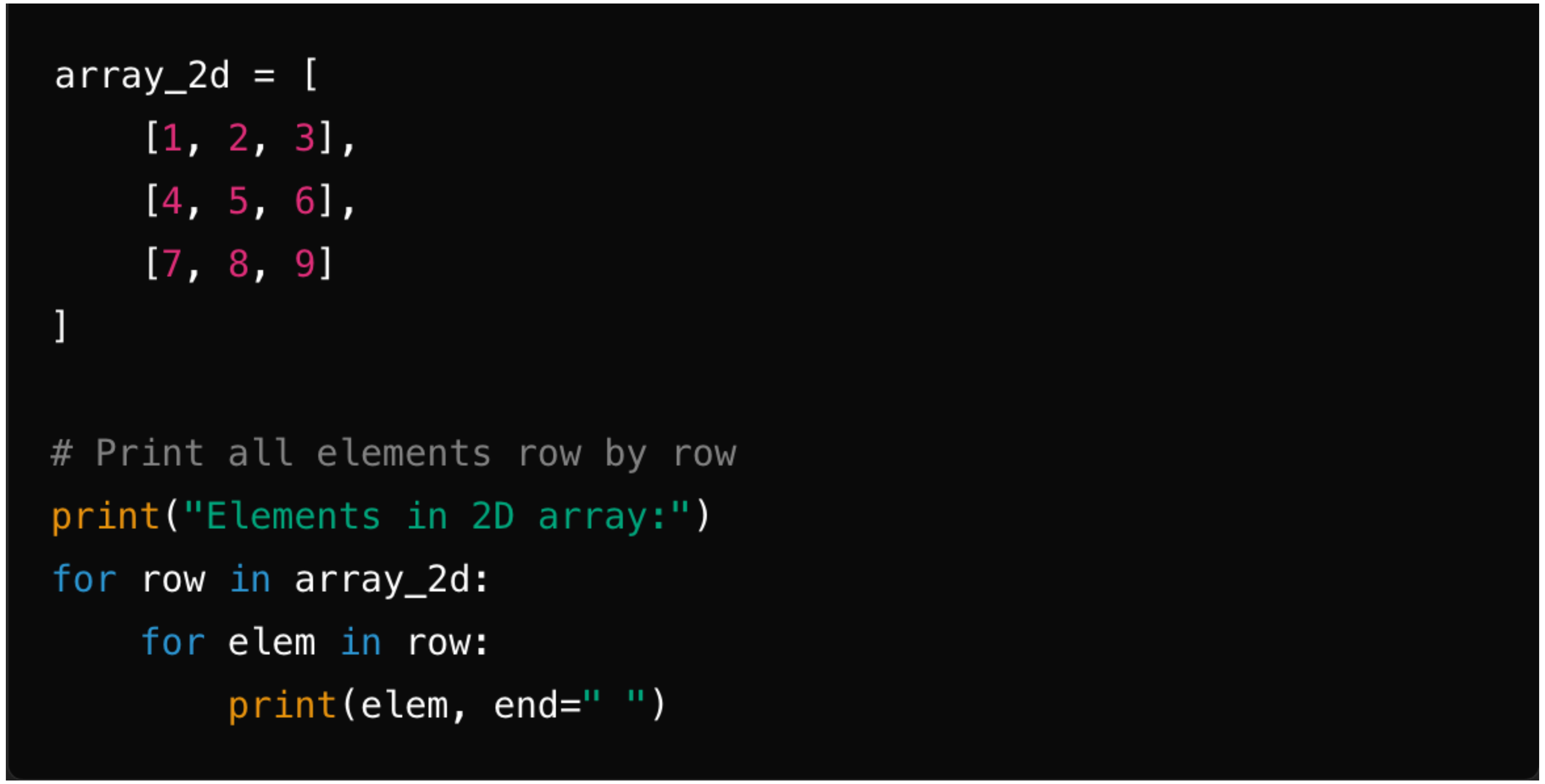
1. Create and Access a 2D Array (List of Lists)



```python
# Create a 2D array
array_2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access specific elements
print("Element at row 1, column 2:", array_2d[0][1])  # Output: 2

# Modify an element
array_2d[2][1] = 88
print("Modified Array:")
for row in array_2d:
    print(row)
```

Examples of how to use a 2D array in Python:

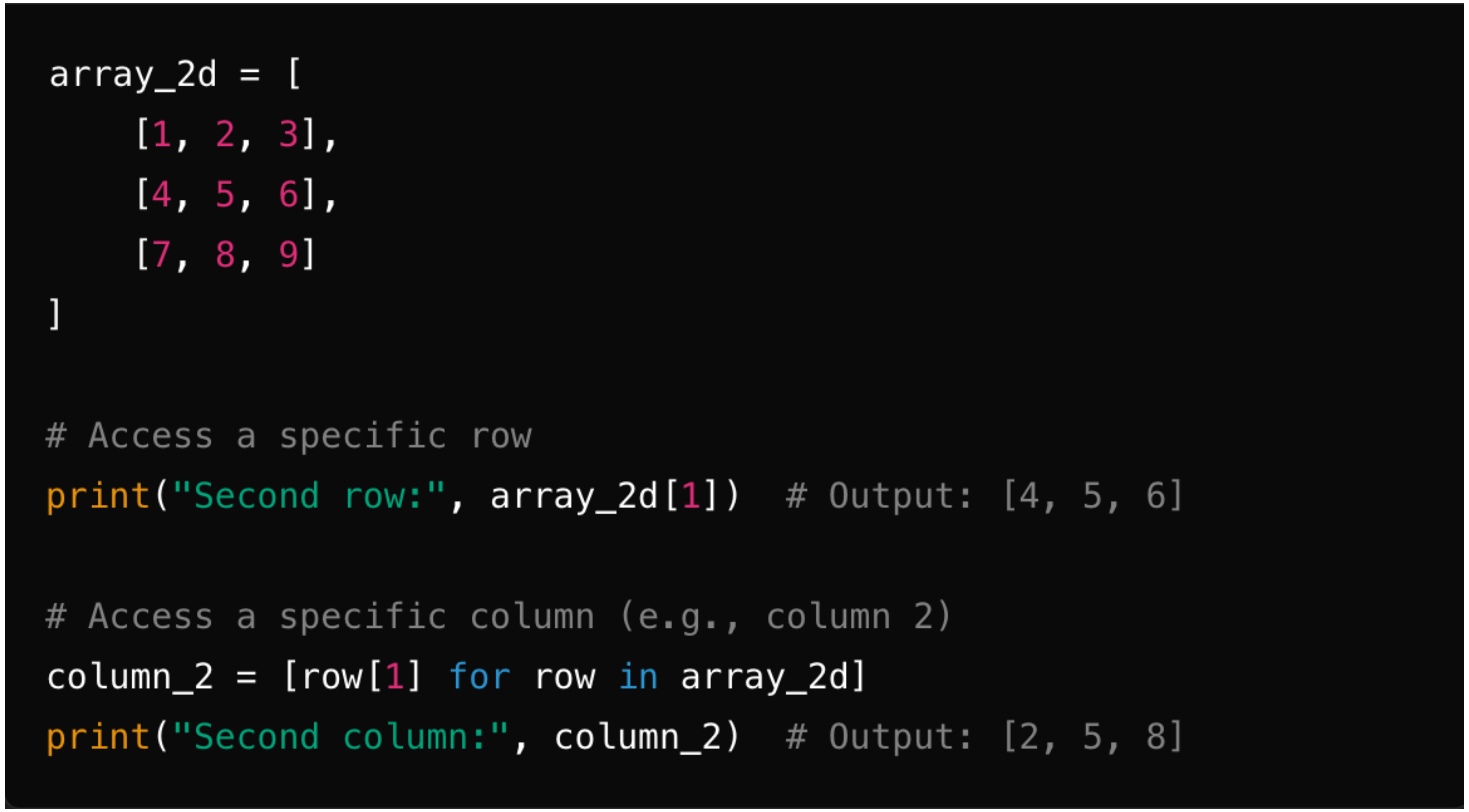
2. Traversing a 2D Array



```python
array_2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Print all elements row by row
print("Elements in 2D array:")
for row in array_2d:
    for elem in row:
        print(elem, end=" ")
```

Examples of how to use a 2D array in Python:

3. Row and Column Operations

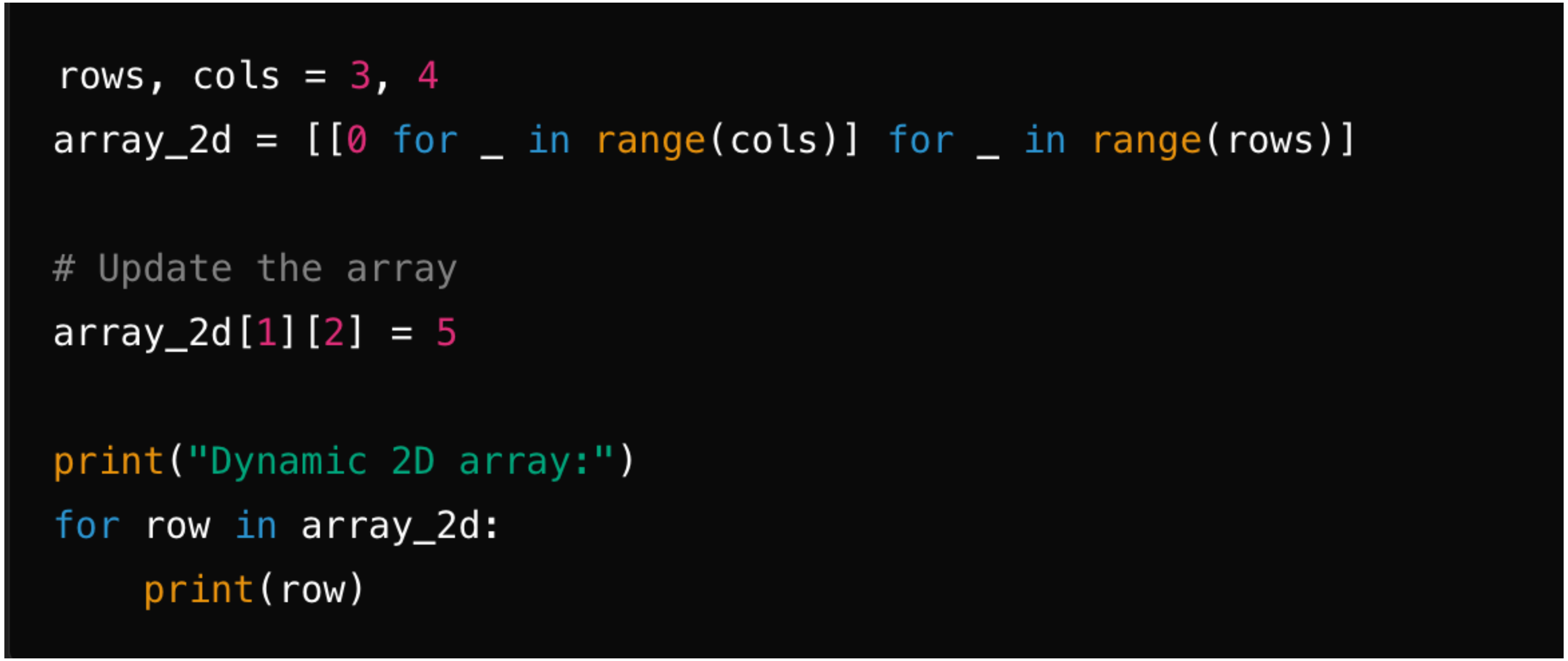


```python
array_2d = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access a specific row
print("Second row:", array_2d[1])  # Output: [4, 5, 6]

# Access a specific column (e.g., column 2)
column_2 = [row[1] for row in array_2d]
print("Second column:", column_2)  # Output: [2, 5, 8]
```

Examples of how to use a 2D array in Python:

4. Create a Dynamic 2D Array

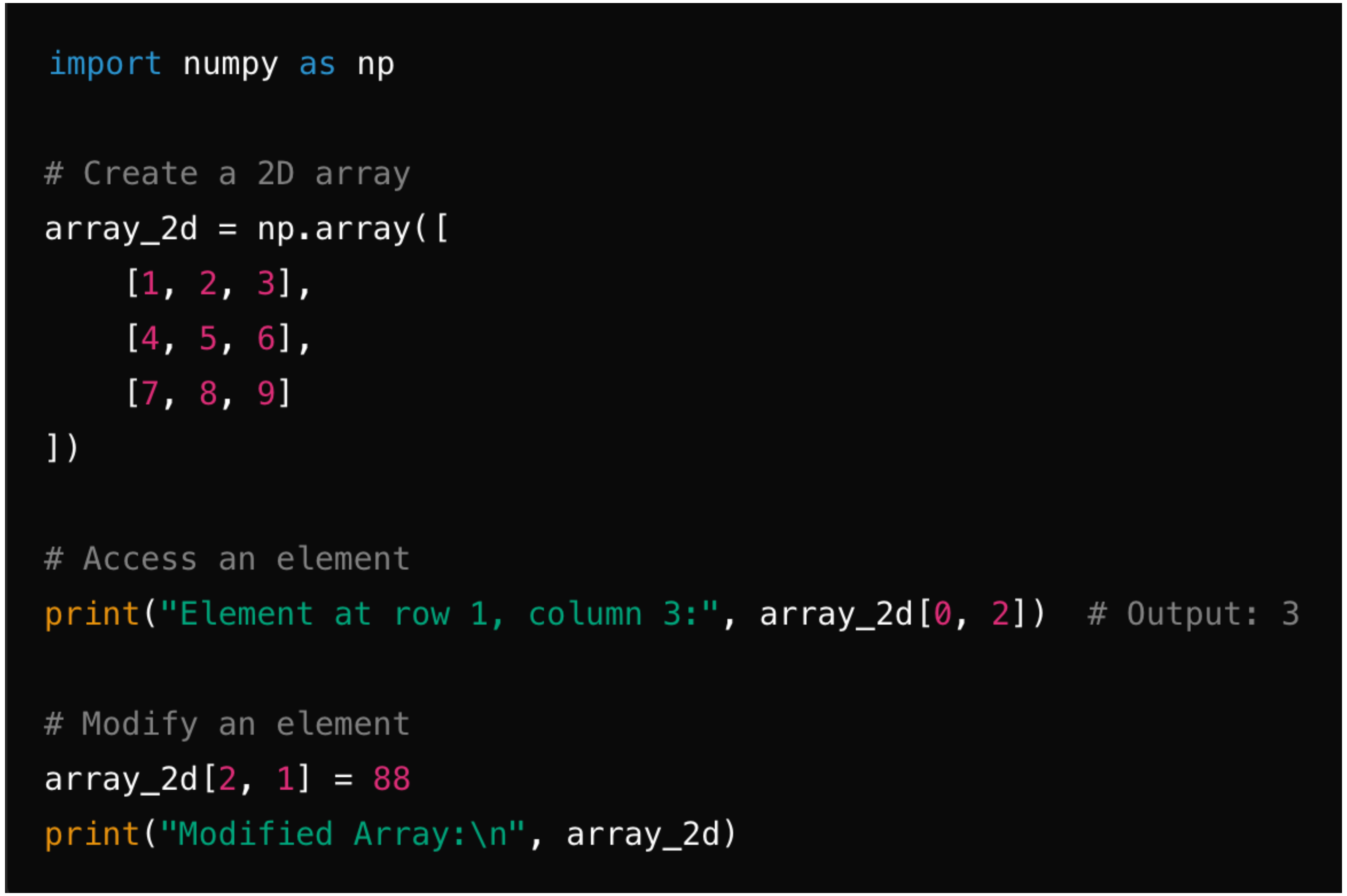
You can dynamically create a 2D array using loops.



```python
rows, cols = 3, 4
array_2d = [[0 for _ in range(cols)] for _ in range(rows)]

# Update the array
array_2d[1][2] = 5

print("Dynamic 2D array:")
for row in array_2d:
    print(row)
```

Examples of how to use a 2D array in Python:

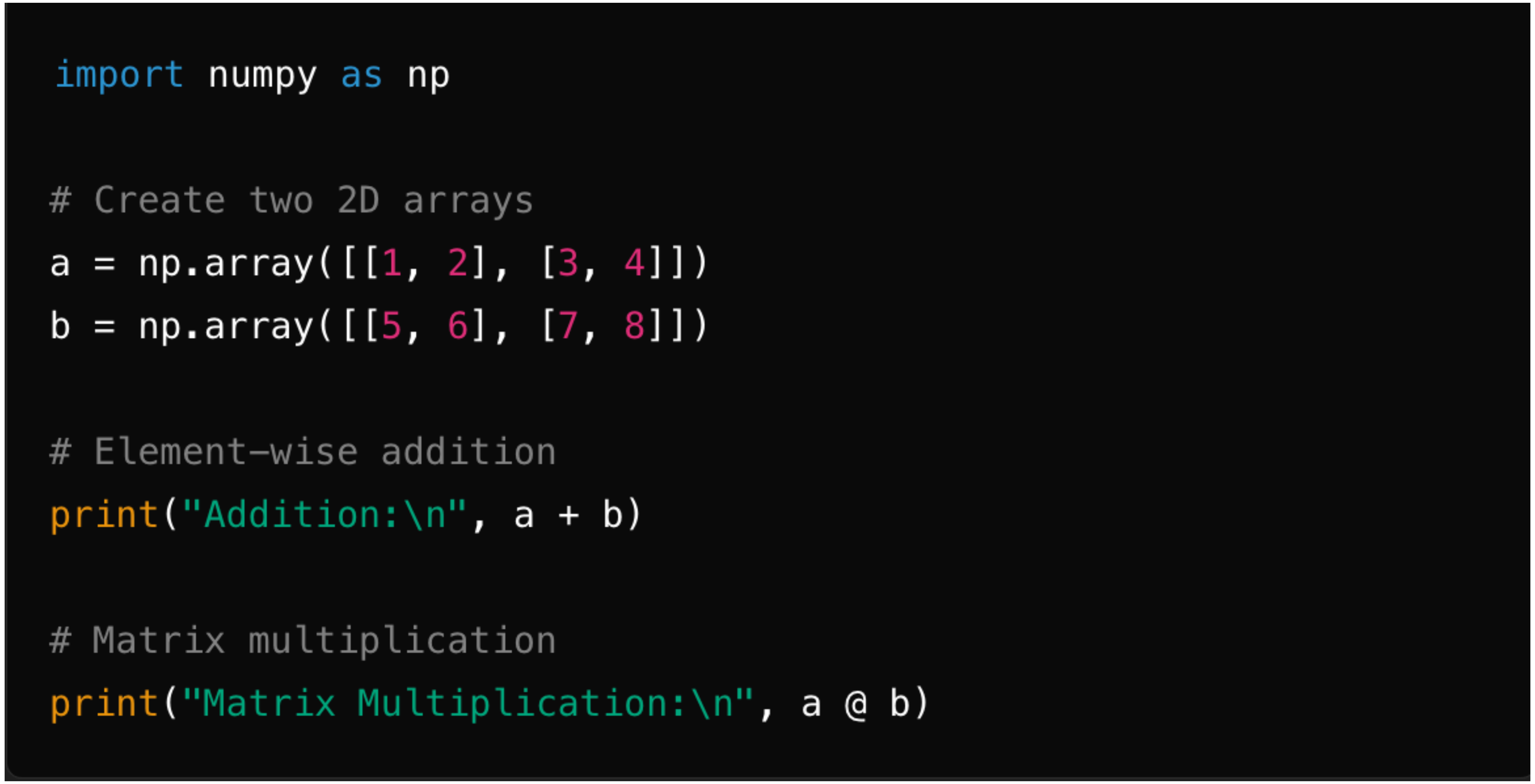
5. Using NumPy for 2D Arrays



```python
import numpy as np

# Create a 2D array
array_2d = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
])

# Access an element
print("Element at row 1, column 3:", array_2d[0, 2])  # Output: 3

# Modify an element
array_2d[2, 1] = 88
print("Modified Array:\n", array_2d)
```

Examples of how to use a 2D array in Python:

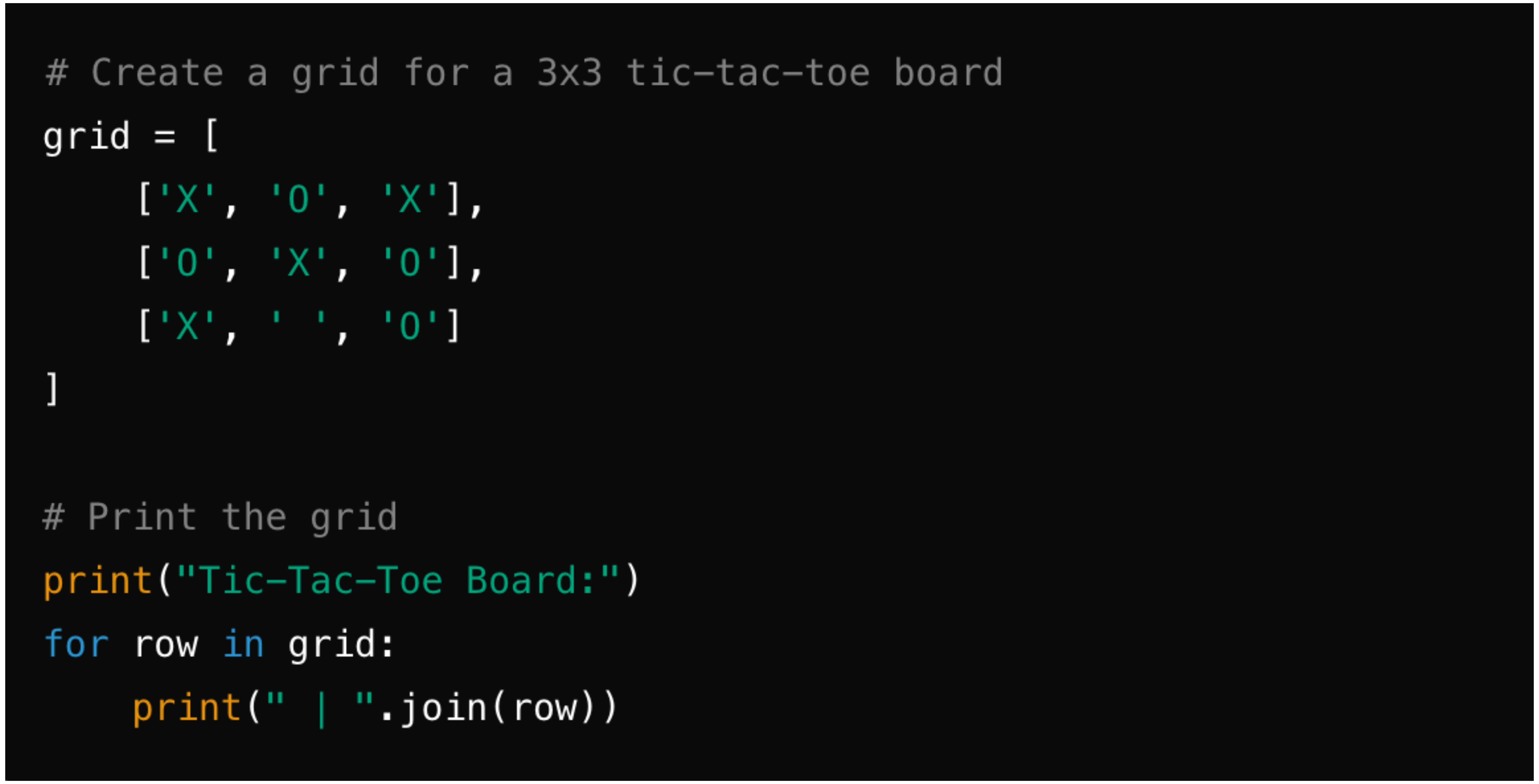
6. Perform Mathematical Operations on a 2D Array



```
import numpy as np

# Create two 2D arrays
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])

# Element-wise addition
print("Addition:\n", a + b)

# Matrix multiplication
print("Matrix Multiplication:\n", a @ b)
```

Examples of how to use a 2D array in Python:

7. Example: Representing a Grid



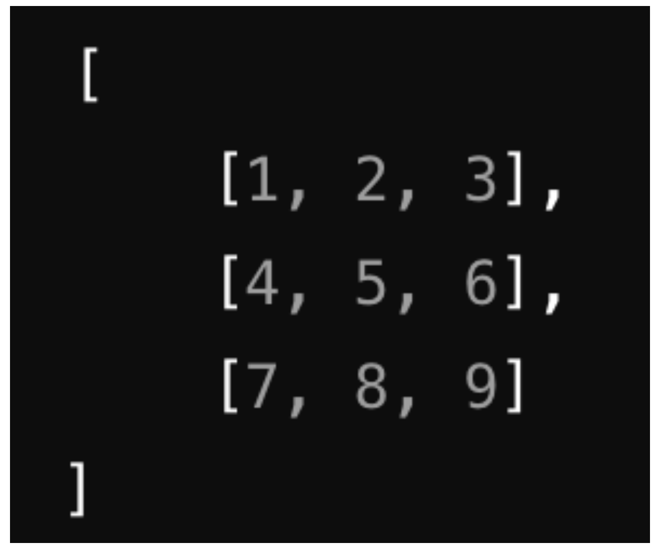
```python
# Create a grid for a 3x3 tic-tac-toe board
grid = [
    ['X', 'O', 'X'],
    ['O', 'X', 'O'],
    ['X', ' ', 'O']
]

# Print the grid
print("Tic-Tac-Toe Board:")
for row in grid:
    print(" | ".join(row))
```

Exercise 1: Create and Access Elements in a 2D Array

Level: Easy

Write a program to:

1. Create an array of integers with values

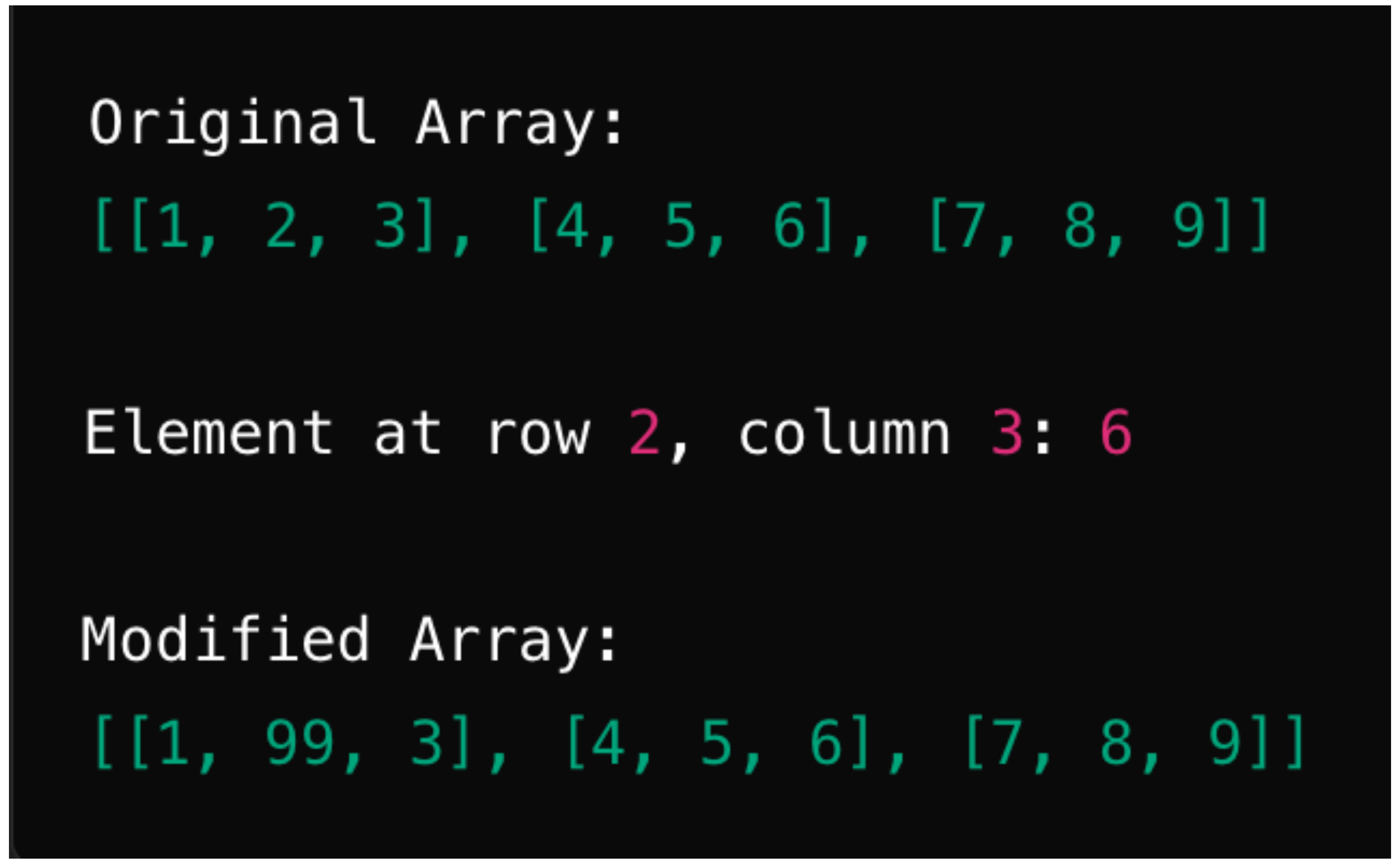


```
[
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

2. Print the entire array.
3. Access and print the element at row 2, column 3.
4. Modify the element at row 1, column 2 to `99`.
5. Print the modified array.

\

Exercise 1: Expected Output



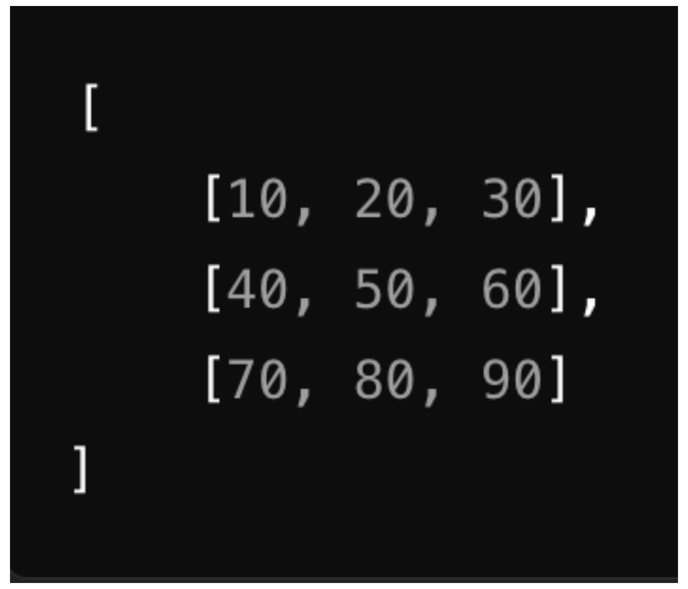
```
Original Array:
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Element at row 2, column 3: 6

Modified Array:
[[1, 99, 3], [4, 5, 6], [7, 8, 9]]
```

Exercise 2: Row and Column Operations

Level: Medium

Write a program to:

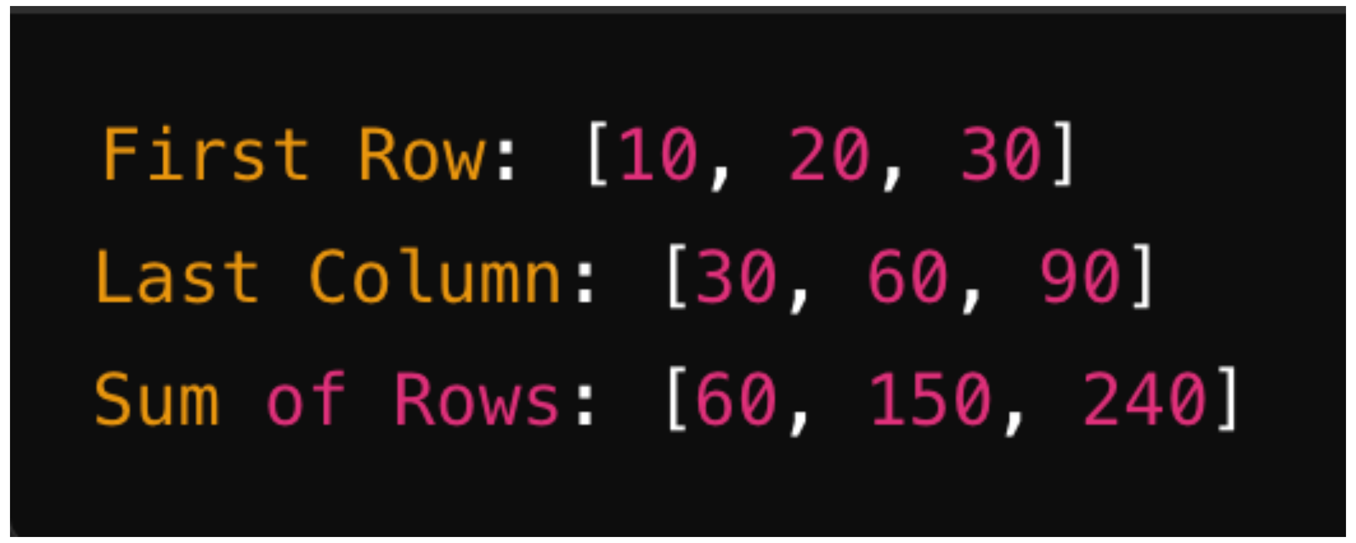
1. Create a 2D array:



```
[
    [10, 20, 30],
    [40, 50, 60],
    [70, 80, 90]
]
```

2. Print the first row and the last column.
3. Calculate the sum of all elements in each row and print the result as a list.

Exercise 2: Expected Output



```
First Row: [10, 20, 30]
Last Column: [30, 60, 90]
Sum of Rows: [60, 150, 240]
```

\

Exercise 3: Dynamic 2D Array with Input

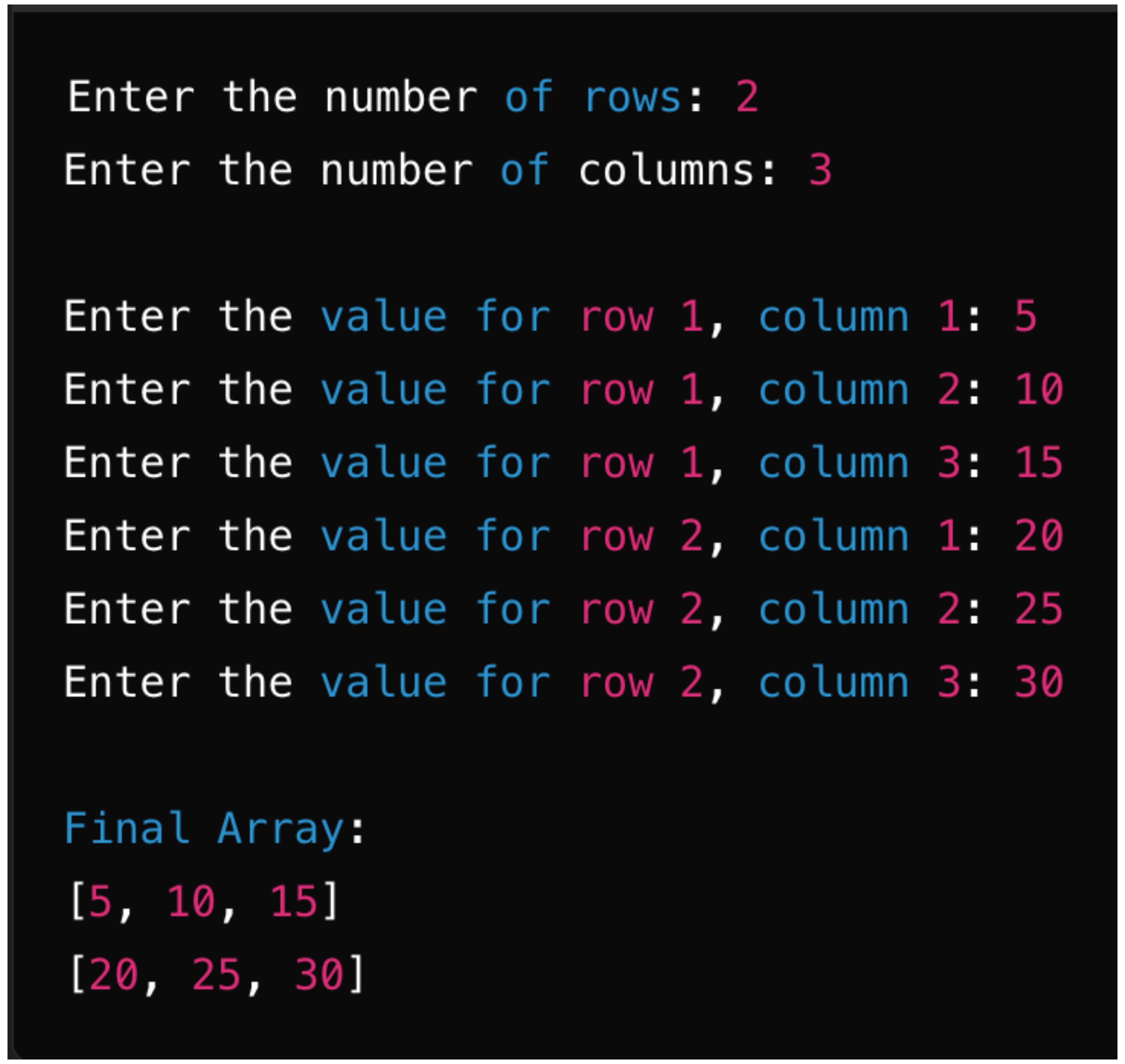
Level: Hard

Write a program to:

1. Ask the user to input the number of rows and columns for a 2D array.
2. Dynamically create a 2D array filled with `0`s.
3. Ask the user to input values for specific positions in the array (e.g., row 1, column 1).
4. Print the final array.

\

Exercise 3: Expected Output



```
Enter the number of rows: 2
Enter the number of columns: 3

Enter the value for row 1, column 1: 5
Enter the value for row 1, column 2: 10
Enter the value for row 1, column 3: 15
Enter the value for row 2, column 1: 20
Enter the value for row 2, column 2: 25
Enter the value for row 2, column 3: 30

Final Array:
[5, 10, 15]
[20, 25, 30]
```

Assignment 1

List and Array (30 Points)