02 Arrays in Python

Arrays in Python

Introduction to Arrays

An **array** is a fundamental data structure used to store and organize collections of items. In Python, arrays are implemented in different ways depending on the use case:

1. **Lists:** Python's built-in list type is the most commonly used array-like structure.
2. **Arrays Module:** The `array` module provides more memory-efficient arrays for storing homogeneous (same-type) data.
3. **NumPy Arrays:** For numerical computations, NumPy arrays are highly optimized and widely used.
4. **Other Implementations:** Libraries like Pandas (Series/DataFrame) and SciPy also provide specialized array structures.

---

1. Lists: Python's Built-in Array-Like Structure

What is a List?

A **list** is a dynamic array that can hold elements of different types. It is one of Python's most versatile and commonly used data structures.

Key Features of Lists

- **Dynamic Size:** Lists can grow or shrink as needed.
- **Heterogeneous Data:** Lists can store elements of different types (e.g., integers, strings, objects).
- **Indexing:** Elements are accessed using zero-based indexing.
- **Mutable:** Lists can be modified after creation.

Creating a List

```python
# Empty list
empty_list = []

# List with initial values
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]

# List comprehension - a powerful way to create lists
squares = [x**2 for x in range(1, 11)]  # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Creating a list of repeated elements
zeros = [0] * 5  # [0, 0, 0, 0, 0]

# Converting other iterables to lists
chars_list = list("Python")  # ['P', 'y', 't', 'h', 'o', 'n']
```

Accessing Elements

Elements in a list are accessed using square brackets `[]` and zero-based indexing:

```python
numbers = [10, 20, 30, 40, 50]
print(numbers[0])  # Output: 10
print(numbers[-1])  # Output: 50 (last element)

# Slicing
print(numbers[1:4])  # Output: [20, 30, 40]
print(numbers[:3])   # Output: [10, 20, 30]
print(numbers[2:])   # Output: [30, 40, 50]
```

```python
print(numbers[::2])  # Output: [10, 30, 50] (step of 2)
print(numbers[::-1]) # Output: [50, 40, 30, 20, 10] (reverse)
```

Modifying a List

Lists are mutable, meaning you can change their contents:

```python
numbers = [1, 2, 3]
numbers[0] = 10  # Update first element
numbers.append(4)  # Add an element to the end
numbers.remove(2)  # Remove element with value 2

# Insert at specific position
numbers.insert(1, 15)  # Insert 15 at index 1

# Delete elements
del numbers[0]  # Delete item at index 0
popped_value = numbers.pop()  # Remove and return the last item
popped_at_index = numbers.pop(1)  # Remove and return item at index 1
```

Common Operations on Lists

Operation	Example	Description
Append	`numbers.append(6)`	Adds an element to the end
Extend	`numbers.extend([7, 8])`	Adds multiple elements to the end
Insert	`numbers.insert(2, 99)`	Inserts an element at a specific index
Remove	`numbers.remove(3)`	Removes the first occurrence of an element
Pop	`numbers.pop()`	Removes and returns the last element
Index	`numbers.index(4)`	Finds the index of an element
Count	`numbers.count(2)`	Counts occurrences of an element
Sort	`numbers.sort()`	Sorts the list in-place
Reverse	`numbers.reverse()`	Reverses the list in-place
Clear	`numbers.clear()`	Removes all elements
Copy	`numbers.copy()`	Creates a shallow copy

Advanced List Operations

```python
# List comprehension with conditional statements
evens = [x for x in range(20) if x % 2 == 0]  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# Nested list comprehension
matrix = [[i*j for j in range(1, 4)] for i in range(1, 4)]
# [[1, 2, 3], [2, 4, 6], [3, 6, 9]]

# Sorting with custom key function
names = ["Alice", "Bob", "Charlie", "Dave"]
names.sort(key=len)  # Sort by length: ["Bob", "Dave", "Alice", "Charlie"]

# Filtering with list comprehension
filtered = [x for x in range(100) if x % 10 == 0]  # [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]

# Flattening a 2D list
nested = [[1, 2], [3, 4], [5, 6]]
flattened = [item for sublist in nested for item in sublist]  # [1, 2, 3, 4, 5, 6]

# Using zip to combine lists
```

```python
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
combined = list(zip(names, ages))  # [('Alice', 25), ('Bob', 30), ('Charlie', 35)]
```

Real-World Use Cases for Lists

1. **To-Do Applications:** Store and manage task items

```python
todo_list = ["Buy groceries", "Pay bills", "Call doctor", "Send email"]
completed = []

# Mark a task as complete
task = todo_list.pop(0)
completed.append(task)
```

2. **Web Development:** Managing HTTP request/response data

```python
user_data = [
    {"id": 1, "name": "Alice", "email": "alice@example.com"},
    {"id": 2, "name": "Bob", "email": "bob@example.com"},
    {"id": 3, "name": "Charlie", "email": "charlie@example.com"}
]

# Find a user by ID
user_id = 2
found_user = None
for user in user_data:
    if user["id"] == user_id:
        found_user = user
        break
```

3. **Data Processing:** Collecting and processing results

```python
# Process a list of website URLs to check availability
urls = ["https://example.com", "https://test.org", "https://python.org"]
status_codes = []

# (In a real application, you would use requests library)
import requests

for url in urls:
    try:
        response = requests.get(url)
        status_codes.append((url, response.status_code))
    except Exception as e:
        status_codes.append((url, str(e)))
```

---

2. Arrays from the `array` Module

The `array` module provides a space-efficient way to store large collections of numeric data. Unlike lists, arrays in the `array` module are **homogeneous**, meaning all elements must be of the same type.

Why Use the `array` Module?

- **Memory Efficiency:** Arrays consume less memory than lists when storing large amounts of numeric data.
- **Type-Specific Storage:** Ensures all elements are of the same type, which can improve performance in certain scenarios.
- **I/O Efficiency:** Can be more efficiently written to and read from files.

Type Codes for `array` Module

Type Code	C Type	Python Type	Size (bytes)
'b'	signed char	int	1
'B'	unsigned char	int	1
'h'	signed short	int	2
'H'	unsigned short	int	2
'i'	signed int	int	4
'I'	unsigned int	int	4
'l'	signed long	int	4 or 8
'L'	unsigned long	int	4 or 8
'q'	signed long long	int	8
'Q'	unsigned long long	int	8
'f'	float	float	4
'd'	double	float	8

Creating an Array

```python
import array

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

# Create an array of floats
float_array = array.array('f', [1.0, 2.5, 3.7])

# Create an array from a list
values = [10, 20, 30, 40, 50]
int_array2 = array.array('i', values)

# Create an empty array and append values
empty_array = array.array('i')
for i in range(5):
    empty_array.append(i * 10)
```

Accessing and Modifying Elements

Accessing and modifying elements in an `array` works similarly to lists:

```python
int_array = array.array('i', [10, 20, 30])
print(int_array[0])  # Output: 10

int_array[1] = 25  # Modify second element

# Slicing
sub_array = int_array[1:3]  # Creates a new array with elements at indices 1 and 2
```

Common Operations on Arrays

```python
import array

int_array = array.array('i', [10, 20, 30, 40, 50])

# Add elements
int_array.append(60)
```

```
int_array.extend([70, 80, 90])
int_array.insert(2, 25)  # Insert 25 at index 2

# Remove elements
int_array.remove(30)  # Remove first occurrence of 30
popped = int_array.pop()  # Remove and return last element
popped_index = int_array.pop(2)  # Remove and return element at index 2

# Search
index = int_array.index(40)  # Find index of first occurrence of 40

# Convert to list and back
as_list = int_array.tolist()
back_to_array = array.array('i', as_list)

# File I/O
with open('array.bin', 'wb') as f:
    int_array.tofile(f)  # Write array to binary file

# Read from file
new_array = array.array('i')
with open('array.bin', 'rb') as f:
    new_array.fromfile(f, len(int_array))
```

Memory Usage Comparison

```
import array
import sys

# Create a list and an array with 1 million integers
int_list = list(range(1000000))
int_array = array.array('i', range(1000000))

# Compare memory usage
list_size = sys.getsizeof(int_list) + sum(sys.getsizeof(i) for i in int_list[:5])
array_size = sys.getsizeof(int_array)

print(f"Estimated list size: {list_size:,} bytes")
print(f"Array size: {array_size:,} bytes")
# Result: The array is significantly smaller
```

Real-World Use Cases for `array` Module

1. **Binary File I/O**: Efficiently storing and reading numeric data

```
import array

# Reading raw data from a binary sensor
sensor_readings = array.array('f')  # Array of floats

# Simulate reading 1000 temperature values from a binary file
with open('sensor_data.bin', 'rb') as f:
    # Read up to 1000 float values (4 bytes each)
    sensor_readings.fromfile(f, 1000)

# Process the readings
avg_temp = sum(sensor_readings) / len(sensor_readings)
```

3. NumPy Arrays

For scientific computing and numerical analysis, **NumPy arrays** are the preferred choice. They are highly optimized for performance and provide powerful tools for mathematical operations.

Why Use NumPy Arrays?

- **Performance:** NumPy arrays are faster and more memory-efficient than Python lists.
- **Vectorized Operations:** Supports element-wise operations without explicit loops.
- **Multidimensional Support:** Can represent matrices and higher-dimensional data.
- **Broadcasting:** Automatically works with arrays of different shapes.
- **Advanced Indexing:** Supports boolean and fancy indexing.

Installing NumPy

If you don't already have NumPy installed, you can install it using pip:

```
pip install numpy
```

Creating a NumPy Array

```python
import numpy as np

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

# Create a 2D array (matrix)
matrix = np.array([[1, 2, 3], [4, 5, 6]])

# Create arrays with specific data types
float_arr = np.array([1, 2, 3], dtype=np.float64)
int_arr = np.array([1.1, 2.2, 3.3], dtype=np.int32)

# Create arrays with specific values
zeros = np.zeros((3, 4))  # 3x4 array of zeros
ones = np.ones((2, 3, 4))  # 2x3x4 array of ones
empty = np.empty((2, 3))  # 2x3 array with uninitialized values
full = np.full((2, 2), 7)  # 2x2 array filled with 7

# Create arrays with sequences
range_arr = np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
linear = np.linspace(0, 1, 5)  # 5 evenly spaced values from 0 to 1: [0, 0.25, 0.5, 0.75, 1]

# Create arrays with random values
uniform_random = np.random.random((2, 2))  # Random values from 0 to 1
normal_random = np.random.normal(0, 1, (2, 2))  # Normal distribution (mean=0, std=1)
random_ints = np.random.randint(0, 10, (3, 3))  # Random integers from 0 to 9
```

Array Attributes and Basic Operations

```python
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

# Array attributes
print(arr.shape)      # (2, 3) - dimensions
print(arr.ndim)       # 2 - number of dimensions
print(arr.size)       # 6 - total number of elements
print(arr.dtype)      # int64 - data type of elements
print(arr.itemsize)   # 8 - size in bytes of each element
```

```python
print(arr.nbytes)     # 48 - total size in bytes

# Reshape an array
reshaped = arr.reshape(3, 2)  # [[1, 2], [3, 4], [5, 6]]
flattened = arr.flatten()     # [1, 2, 3, 4, 5, 6]
transposed = arr.T            # [[1, 4], [2, 5], [3, 6]]

# Array operations
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Arithmetic operations
print(a + b)       # [5, 7, 9] - element-wise addition
print(a - b)       # [-3, -3, -3] - element-wise subtraction
print(a * b)       # [4, 10, 18] - element-wise multiplication
print(a / b)       # [0.25, 0.4, 0.5] - element-wise division
print(a ** 2)      # [1, 4, 9] - element-wise exponentiation
print(np.sqrt(a))  # [1., 1.41421356, 1.73205081] - element-wise square root

# Statistical operations
data = np.array([1, 2, 3, 4, 5])
print(np.sum(data))     # 15 - sum of all elements
print(np.mean(data))    # 3.0 - mean of all elements
print(np.max(data))     # 5 - maximum value
print(np.min(data))     # 1 - minimum value
print(np.std(data))     # 1.41421356 - standard deviation
```

Array Indexing and Slicing

```python
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# Basic indexing
print(arr[0, 0])    # 1
print(arr[2, 3])    # 12
print(arr[1])       # [5, 6, 7, 8] - entire second row

# Slicing
print(arr[:, 1])    # [2, 6, 10] - second column
print(arr[1:3, :2]) # [[5, 6], [9, 10]] - subset of rows 1-2, columns 0-1

# Boolean indexing
mask = arr > 5
print(arr[mask])    # [6, 7, 8, 9, 10, 11, 12] - values greater than 5

# Fancy indexing
row_indices = np.array([0, 2])
col_indices = np.array([1, 3])
print(arr[row_indices])                # [[1, 2, 3, 4], [9, 10, 11, 12]] - selected rows
print(arr[:, col_indices])             # [[2, 4], [6, 8], [10, 12]] - selected columns
print(arr[row_indices[:, np.newaxis], col_indices])  # [[2, 4], [10, 12]] - specific elements
```

Vectorized Operations and Broadcasting

```python
import numpy as np
import time

# Vectorized operations vs. loops
size = 1000000
```

```python
a = np.random.random(size)
b = np.random.random(size)

# Using a loop
start = time.time()
result_loop = np.zeros(size)
for i in range(size):
    result_loop[i] = a[i] * b[i]
loop_time = time.time() - start

# Using vectorized operation
start = time.time()
result_vec = a * b
vec_time = time.time() - start

print(f"Loop time: {loop_time:.6f} seconds")
print(f"Vectorized time: {vec_time:.6f} seconds")
print(f"Speedup: {loop_time/vec_time:.1f}x")

# Broadcasting example
a = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3 array
b = np.array([10, 20, 30])            # 1D array with 3 elements

# NumPy broadcasts b to match a's shape
result = a + b      # [[11, 22, 33], [14, 25, 36]]

# Adding a column vector
c = np.array([[100], [200]])  # 2x1 array
result2 = a + c     # [[101, 102, 103], [204, 205, 206]]
```