Chapter: Mastering NumPy Indexing, Slicing, and Memory Views for IQ Data Processing

---

Introduction

In Software Defined Radio (SDR) and IQ data processing, efficiently accessing, manipulating, and understanding your data is crucial. NumPy arrays are the workhorse for handling IQ samples, but to use them effectively, you must grasp how **indexing**, **slicing**, and **memory views** work.

This chapter builds your understanding step-by-step with clear explanations and examples focused on IQ data. By the end, you will confidently navigate array manipulations and optimize your SDR workflows.

---

1. NumPy Arrays: The Foundation

What is a NumPy array?

A NumPy array is a grid of values, all of the same type, indexed by a tuple of non-negative integers.

- It can be 1D (vector), 2D (matrix), or N-dimensional.
- NumPy arrays support fast vectorized operations — no explicit Python loops needed.
- For IQ data, arrays typically represent complex samples, or sometimes separate real I and Q components stored in 2D arrays.

---

2. Understanding Array Indexing

Single-Dimensional Arrays

Indexing in Python starts at **0**. This means the first element of an array is accessed with index 0.

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])  # Output: 10
print(arr[4])  # Output: 50
```

Negative Indices: Counting Backwards

Negative indices count from the end:

- `-1` → last element
- `-2` → second last
- etc.

```
print(arr[-1])  # Output: 50
print(arr[-3])  # Output: 30
```

---

3. Multidimensional Arrays and Their Indexing

For 2D arrays (common for separate I/Q channels), indexing uses a tuple:

```
arr2d = np.array([
  [10, 100],  # Row 0
  [20, 200],  # Row 1
  [30, 300],  # Row 2
  [40, 400],  # Row 3
])
```

Accessing Rows and Columns

- `arr2d[0, 0]` → element in row 0, column 0 → `10`
- `arr2d[1, 1]` → element in row 1, column 1 → `200`

Selecting Entire Rows or Columns

- `arr2d[1, :]` → all columns of row 1 → `[20, 200]`
- `arr2d[:, 0]` → all rows of column 0 → `[10, 20, 30, 40]`

Explanation:

- The colon `:` means "take all" along that axis.
- First index is rows; second is columns.

---

4. Slicing — Extracting Subarrays Efficiently

Basic Slicing Syntax

```
arr[start:stop:step]
```

- `start`: index to start at (inclusive)
- `stop`: index to stop at (exclusive)
- `step`: step size between indices

Example:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])     # Output: [20, 30, 40]
print(arr[::2])     # Output: [10, 30, 50] (every second element)
print(arr[::-1])    # Output: [50, 40, 30, 20, 10] (reverse)
```

Multidimensional Slicing

For 2D arrays:

```
arr2d[1:3, 0]  # rows 1 and 2, column 0
arr2d[:, 1]    # all rows, column 1
arr2d[::2, 1]  # every second row, column 1
```

---

5. Negative Indices with Slicing

Negative indices also work with slicing:

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[-3:])  # Output: [30, 40, 50] (last 3 elements)
print(arr[:-2])  # Output: [10, 20, 30] (up to 2nd last element)
```

---

6. The Magic of Views — No Copying, Just a Window

One of NumPy's most powerful features is **views**: slicing operations do **not create copies** of the data, but rather **create new arrays that view the original data**.

Why does this matter?

- **Efficiency:** No copying means very low memory overhead.
- **Mutability:** Changing the view changes the original array!

Example of a view:

```
arr = np.array([10, 20, 30, 40, 50])
view = arr[1:4]  # slice, creates a view
view[0] = 999    # modifies arr as well
print(arr)       # Output: [10, 999, 30, 40, 50]
```

Visualize:

```
Original:  [10] [20] [30] [40] [50]
Index:      0    1    2    3    4

View (slice 1:4) points to:
           [20] [30] [40]
Modifying view[0] changes arr[1] too.
```

---

7. When Views Can Bite You — Be Careful

Since views are just windows, modifying them can unexpectedly alter your original data. If you want an independent copy, use:

```
copy = arr[1:4].copy()
```

---

8. Applying These Concepts to IQ Data

IQ data shapes and indexing

- **1D Complex array:**
  IQ samples combined as `I + jQ`. Indexing is simple:

```
iq[0]  # first complex sample
iq[10:20]  # samples 10 to 19
```

- **2D Real array:**
  IQ stored as two columns (`I`, `Q`):

```
iq2d[:, 0]  # all I samples
iq2d[:, 1]  # all Q samples
iq2d[10:20, :]  # samples 10 to 19, both I and Q
```

Example

```
import numpy as np

iq2d = np.array([
  [1, 10],
  [2, 20],
  [3, 30],
  [4, 40],
  [5, 50]
])

print("I samples:", iq2d[:, 0])   # [1 2 3 4 5]
print("Q samples:", iq2d[:, 1])   # [10 20 30 40 50]

window = iq2d[1:4, :]
print("Window (samples 1 to 3):\n", window)
```

---

9. Combining Strides and Slices — Advanced Selection

You can combine start, stop, and step in multidimensional slicing:

```
# Every second sample, Q channel only
every_2nd_Q = iq2d[::2, 1]
print(every_2nd_Q)  # outputs: [10 30 50]
```

---

10. Summary and Key Takeaways

Concept	What It Means	Example
Indexing starts at 0	First element is index 0	`arr[0]`
Negative indices	Count from the end starting -1	`arr[-1]` last element
Slicing	Extract subarrays	`arr[2:5]` ,

Concept	What It Means	Example
	without copies	`arr[::-1]`
Views	Slices share data with original	Modifying slice changes source
Copy when needed	Make independent arrays	`arr[2:5].copy()`
Multi-dimensional indexing	Use tuples for rows, cols	`arr2d[:, 1]` all col 1
Step in slicing	Skip elements in slices	`arr[::2]` every other element

---

11. Why You Must Master This

- You'll efficiently extract and manipulate IQ chunks — crucial for FFT, filtering, demodulation.
- You'll avoid memory pitfalls in large SDR captures.
- Your DSP code will be cleaner, faster, and less error-prone.