Python Loops vs NumPy Vectorization

Performance Comparison - Basic Operations

```
import numpy as np
import time

# Create large array
n = 1000000
data = np.random.rand(n)

# Method 1: Python loop (SLOW)
start = time.time()
result_loop = []
for x in data:
    result_loop.append(x * 2)
result_loop = np.array(result_loop)
loop_time = time.time() - start

# Method 2: Vectorized (FAST)
start = time.time()
result_vectorized = data * 2
vectorized_time = time.time() - start

print(f"Loop time: {loop_time:.4f} seconds")
print(f"Vectorized time: {vectorized_time:.4f} seconds")
print(f"Speedup: {loop_time / vectorized_time:.1f}x")
```

Square Root Example

```
import numpy as np
import time

n = 500000
data = np.random.rand(n) * 100

# Loop version
start = time.time()
result_loop = []
for x in data:
    result_loop.append(x ** 0.5)
loop_time = time.time() - start

# Vectorized version
start = time.time()
result_vec = np.sqrt(data)
vec_time = time.time() - start

print(f"Loop: {loop_time:.4f}s")
print(f"Vectorized: {vec_time:.4f}s")
print(f"Speedup: {loop_time/vec_time:.1f}x")
print(f"\nResults match: {np.allclose(result_loop, result_vec)}")
```

Conditional Operations

```
# Count elements greater than threshold
data = np.random.rand(1000000)
threshold = 0.5

# Loop version
```

```
start = time.time()
count_loop = 0
for x in data:
    if x > threshold:
        count_loop += 1
loop_time = time.time() - start

# Vectorized version
start = time.time()
count_vec = np.sum(data > threshold)
vec_time = time.time() - start

print(f"Loop count: {count_loop} ({loop_time:.4f}s)")
print(f"Vectorized count: {count_vec} ({vec_time:.4f}s)")
print(f"Speedup: {loop_time/vec_time:.1f}x")
```

Matrix Operations

```
# Matrix multiplication comparison
A = np.random.rand(500, 500)
B = np.random.rand(500, 500)

# Manual loop (very slow - don't run for large matrices!)
# This is just to show concept
def slow_matmul(A, B):
    m, n = A.shape
    n, p = B.shape
    C = np.zeros((m, p))
    for i in range(m):
        for j in range(p):
            for k in range(n):
                C[i, j] += A[i, k] * B[k, j]
    return C

# For demonstration, use small matrices
A_small = np.random.rand(100, 100)
B_small = np.random.rand(100, 100)

start = time.time()
C_loop = slow_matmul(A_small, B_small)
loop_time = time.time() - start

start = time.time()
C_vec = A_small @ B_small
vec_time = time.time() - start

print(f"Loop: {loop_time:.4f}s")
print(f"Vectorized: {vec_time:.4f}s")
print(f"Speedup: {loop_time/vec_time:.1f}x")
```

Best Practices for Vectorization

```
# AVOID: Loop with growing list
# BAD - very slow
result = []
for i in range(100000):
    result.append(i ** 2)

# BETTER - list comprehension
result = [i ** 2 for i in range(100000)]

# BEST - NumPy vectorization
result = np.arange(100000) ** 2

# AVOID: Modifying arrays in loops
data = np.random.rand(100000)
# BAD
for i in range(len(data)):
```

```
    data[i] = data[i] * 2 + 1

# GOOD - vectorized
data = np.random.rand(100000)
data = data * 2 + 1

# Use vectorized functions
# BAD
import math
result = [math.sin(x) for x in data]

# GOOD
result = np.sin(data)

print("Always prefer NumPy vectorized operations!")
print("They are 10-100x faster than Python loops")
```