# ****W10D3 -- Performance Optimization: Seed Improvements****

JTC Program: Tech Pathways  
Cohort: S25  
Lesson Plan: Performance Optimization - Seed Improvements  
Type: Lesson Plan  
Week / Day: W10D3  
Version Date: 05/27/2025

## ****Focus Concepts****

* Understanding what performance optimization means and why it matters
* Identifying common performance bottlenecks in beginner Python code
* Learning "seed improvements" - small changes that make big differences
* Comparing list operations, string operations, and data structure choices
* Measuring and comparing code performance using timing techniques
* Building habits for writing efficient code from the start

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Explain the importance of performance optimization in programming
* Identify inefficient code patterns commonly found in beginner programs
* Apply seed improvements to make code run 2-10x faster
* Choose appropriate data structures (lists vs sets vs dictionaries) for different tasks
* Use list comprehensions and efficient string operations
* Measure code performance using basic timing techniques
* Write more efficient code as a default practice

## ****Out-of-Scope Objectives****

* Advanced optimization techniques (profiling, memory optimization)
* Big O notation and algorithmic complexity analysis
* Database query optimization
* Multi-threading and parallel processing
* Hardware-specific optimizations
* Production deployment performance tuning

## ****Required Competencies****

* Solid understanding of Python basics (variables, functions, loops)
* Experience with lists, dictionaries, and basic data structures
* Familiarity with string operations and manipulation
* Understanding of basic function definitions and calling
* Comfort with writing and running Python scripts

## ****Technical Requirements****

* Python 3.x installed
* Code editor (VS Code, PyCharm, or similar)
* Jupyter Notebook (optional but recommended)
* Access to time module (built into Python)
* Sample datasets for performance testing (will be provided)

## ****Prerequisites****

* Completion of Python fundamentals course
* Understanding of basic data structures (lists, dictionaries, sets)
* Experience writing functions and loops
* Familiarity with string operations
* Basic understanding of file operations

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 25 minutes

Resources:

* [Python Performance Tips (Real Python)](https://realpython.com/python-performance/) - Focus on "Quick Wins" section - 15 minutes
* [Writing Faster Python Code](https://wiki.python.org/moin/PythonSpeed/PerformanceTips) - Read first 3 sections - 10 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. What is the main benefit of using list comprehensions over regular for loops?
   * A) They are easier to read
   * \*B) They are typically faster and more memory efficient
   * C) They can handle more complex logic
   * D) They work with all data types
2. Which data structure provides the fastest lookup time for checking if an item exists?
   * A) List
   * \*B) Set
   * C) Dictionary keys only
   * D) Tuple
3. When joining many strings together, which method is most efficient?
   * A) Using the + operator in a loop
   * B) Using f-strings in a loop
   * \*C) Using the join() method
   * D) Using string formatting
4. What does "premature optimization" mean in programming?
   * A) Optimizing code before it's written
   * \*B) Spending time optimizing code before identifying actual performance problems
   * C) Using advanced optimization techniques too early
   * D) Optimizing only the fastest parts of code
5. Why is it important to measure performance improvements?
   * A) To impress other programmers
   * \*B) To verify that changes actually improve performance
   * C) To find bugs in the code
   * D) To make code more readable

## ****Key Terms****

* **Performance Optimization**: The process of making code run faster or use less memory
* **Seed Improvements**: Small, easy-to-implement changes that provide significant performance gains
* **Bottleneck**: The slowest part of a program that limits overall performance
* **List Comprehension**: A concise way to create lists that's typically faster than loops
* **Time Complexity**: How the runtime of an algorithm changes with input size
* **Lookup Time**: How long it takes to find an item in a data structure
* **String Concatenation**: Combining multiple strings into one
* **Pre-allocation**: Setting the size of a data structure before filling it
* **Caching**: Storing results to avoid recalculating them
* **Profiling**: Measuring which parts of code are slowest
* **Big O Notation**: Mathematical way to describe algorithm efficiency (mentioned but not covered in detail)
* **Memory Efficiency**: Using less RAM while running a program
* **Generator**: A memory-efficient way to create sequences on-demand
* **In-place Operations**: Modifying data structures without creating copies
* **Algorithmic Efficiency**: Choosing the right approach to solve a problem efficiently

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to our performance optimization lesson! Have you ever written code that felt slow or noticed your program taking a long time to run? Today we'll learn simple techniques to make your Python code run significantly faster - sometimes 5-10 times faster with just small changes. These aren't advanced computer science concepts, but practical 'seed improvements' that any programmer can learn and apply immediately."

**Admin Tasks:**

* Take attendance
* Ensure everyone can run Python and import the time module
* Quick check: have students run a simple timing example

**Prompting Questions:**

* "Have you ever noticed your Python code running slowly? What was it doing?"
* "What do you think makes some code faster than other code?"

**Poll Questions:**

* "On a scale of 1-5, how often do you think about making your code faster while writing it?"
* "What's more important to you right now: code that works or code that's fast?"

### ****6:45 PM -- 7:15 PM: Session 1 -- Understanding Performance and Seed Improvements****

**Objective:** Understand what performance optimization means and why small changes can make big differences.

**Instructor Script:** "Let's start by understanding what we mean by performance optimization and see some dramatic examples of how small changes can make huge improvements."

#### ****What is Performance Optimization:****

Performance optimization is making your code run faster or use less memory. For beginners, we focus on "seed improvements" - small, easy changes that give big results.

import time

# Example: The power of small changes

# Let's see a dramatic difference with a simple example

# Slow way: Using + to build a string

def slow\_string\_building(words):

result = ""

for word in words:

result += word + " "

return result

# Fast way: Using join()

def fast\_string\_building(words):

return " ".join(words)

# Test with many words

test\_words = ["hello"] \* 10000

# Time the slow way

start\_time = time.time()

slow\_result = slow\_string\_building(test\_words)

slow\_time = time.time() - start\_time

# Time the fast way

start\_time = time.time()

fast\_result = fast\_string\_building(test\_words)

fast\_time = time.time() - start\_time

print(f"Slow method: {slow\_time:.4f} seconds")

print(f"Fast method: {fast\_time:.4f} seconds")

print(f"Speed improvement: {slow\_time/fast\_time:.1f}x faster!")

#### ****Why Performance Matters for Beginners:****

1. **User Experience**: Faster programs feel more responsive
2. **Scalability**: Good habits help when working with larger datasets
3. **Resource Efficiency**: Uses less computer power and battery
4. **Professional Development**: Writing efficient code is a valued skill
5. **Problem Solving**: Forces you to think about how your code actually works

#### ****The Philosophy of Seed Improvements:****

* **Start Small**: Focus on easy wins before complex optimizations
* **Measure First**: Always time your code before and after changes
* **Learn Patterns**: Recognize common inefficient patterns
* **Build Habits**: Make efficient choices your default approach

# Simple timing function we'll use throughout the lesson

def time\_function(func, \*args):

start\_time = time.time()

result = func(\*args)

end\_time = time.time()

return result, end\_time - start\_time

# Example usage

def example\_function(n):

return sum(range(n))

result, duration = time\_function(example\_function, 100000)

print(f"Function took {duration:.4f} seconds")

### ****7:15 PM -- 7:45 PM: Session 2 -- List and Data Structure Optimizations****

**Objective:** Learn how choosing the right data structure and operations can dramatically improve performance.

**Instructor Script:** "One of the biggest performance improvements comes from choosing the right data structure for the job. Let's see how lists, sets, and dictionaries each excel in different situations."

#### ****List Comprehensions vs Regular Loops:****

# Create test data

numbers = list(range(10000))

# Slow way: Regular loop with append

def slow\_square\_numbers(numbers):

result = []

for num in numbers:

result.append(num \*\* 2)

return result

# Fast way: List comprehension

def fast\_square\_numbers(numbers):

return [num \*\* 2 for num in numbers]

# Compare performance

slow\_result, slow\_time = time\_function(slow\_square\_numbers, numbers)

fast\_result, fast\_time = time\_function(fast\_square\_numbers, numbers)

print(f"Regular loop: {slow\_time:.4f} seconds")

print(f"List comprehension: {fast\_time:.4f} seconds")

print(f"Improvement: {slow\_time/fast\_time:.1f}x faster")

**Key Points:**

* List comprehensions are not just shorter - they're actually faster
* Python optimizes list comprehensions internally
* Use them when you're creating a new list from an existing one

#### ****Sets vs Lists for Membership Testing:****

# Create test data

items\_list = list(range(10000))

items\_set = set(range(10000))

search\_item = 9999 # Worst case - it's at the end

# Slow way: Searching in a list

def find\_in\_list(item, items\_list):

return item in items\_list

# Fast way: Searching in a set

def find\_in\_set(item, items\_set):

return item in items\_set

# Compare performance

\_, list\_time = time\_function(find\_in\_list, search\_item, items\_list)

\_, set\_time = time\_function(find\_in\_set, search\_item, items\_set)

print(f"List search: {list\_time:.6f} seconds")

print(f"Set search: {set\_time:.6f} seconds")

print(f"Set is {list\_time/set\_time:.0f}x faster!")

**Key Points:**

* Sets use hash tables for O(1) average lookup time
* Lists require O(n) linear search time
* Use sets when you need to check "is this item in my collection?"

#### ****Dictionary Counting vs Manual Counting:****

# Sample data

words = ["apple", "banana", "apple", "cherry", "banana", "apple"] \* 1000

# Slow way: Count each word manually

def slow\_count\_words(words):

unique\_words = list(set(words))

counts = {}

for word in unique\_words:

counts[word] = words.count(word) # This searches the entire list!

return counts

# Fast way: Count as you go

def fast\_count\_words(words):

counts = {}

for word in words:

counts[word] = counts.get(word, 0) + 1

return counts

# Compare performance

slow\_counts, slow\_time = time\_function(slow\_count\_words, words)

fast\_counts, fast\_time = time\_function(fast\_count\_words, words)

print(f"Manual counting: {slow\_time:.4f} seconds")

print(f"Dictionary counting: {fast\_time:.4f} seconds")

print(f"Improvement: {slow\_time/fast\_time:.1f}x faster")

**Key Points:**

* Avoid using .count() in loops - it searches the entire list each time
* Build dictionaries incrementally for counting
* The .get() method provides a clean way to handle missing keys

### ****7:45 PM -- 8:00 PM: Break****

### ****8:00 PM -- 8:30 PM: Session 3 -- String and Text Processing Optimizations****

**Objective:** Learn efficient techniques for working with strings and text data.

**Instructor Script:** "String operations are common in many programs, and there are several patterns that can dramatically slow down your code. Let's learn the most efficient ways to work with text."

#### ****String Concatenation Methods:****

# Test data

words = ["Python", "is", "a", "great", "programming", "language"] \* 500

# Slow way: Using += in a loop

def slow\_join\_strings(words):

result = ""

for word in words:

result += word + " "

return result.strip()

# Fast way: Using join()

def fast\_join\_strings(words):

return " ".join(words)

# Compare performance

slow\_result, slow\_time = time\_function(slow\_join\_strings, words)

fast\_result, fast\_time = time\_function(fast\_join\_strings, words)

print(f"Using += operator: {slow\_time:.4f} seconds")

print(f"Using join() method: {fast\_time:.4f} seconds")

print(f"Improvement: {slow\_time/fast\_time:.1f}x faster")

**Why join() is faster:**

* String concatenation with += creates a new string object each time
* join() allocates memory once and fills it efficiently
* The difference becomes dramatic with many strings

#### ****String Formatting Performance:****

name = "Alice"

age = 25

city = "New York"

# Test different formatting methods

def old\_formatting():

return "Hello, %s! You are %d years old and live in %s." % (name, age, city)

def format\_method():

return "Hello, {}! You are {} years old and live in {}.".format(name, age, city)

def f\_string\_formatting():

return f"Hello, {name}! You are {age} years old and live in {city}."

# Compare performance (run many iterations to see difference)

iterations = 100000

print("String formatting performance comparison:")

for method\_name, method\_func in [

("% formatting", old\_formatting),

(".format() method", format\_method),

("f-string formatting", f\_string\_formatting)

]:

\_, duration = time\_function(lambda: [method\_func() for \_ in range(iterations)])

print(f"{method\_name}: {duration:.4f} seconds")

**Key Points:**

* f-strings are generally the fastest and most readable
* Avoid old % formatting in new code
* For simple cases, the difference is small, but it adds up

### ****8:30 PM -- 8:55 PM: Session 4 -- Building a Performance-Optimized Program****

**Objective:** Apply multiple optimization techniques to build a complete, efficient program.

**Instructor Script:** "Now let's put everything together by building a text analysis program that demonstrates multiple optimization techniques working together."

#### ****Hands-On Exercise: Text Analyzer****

Students will work on optimizing a text analysis program that:

1. Reads and processes text data
2. Counts word frequencies
3. Finds the most common words
4. Analyzes text patterns

class TextAnalyzer:

def \_\_init\_\_(self):

self.word\_cache = {} # Cache results to avoid recalculation

def clean\_text\_slow(self, text):

"""Slow version - shows what NOT to do"""

result = ""

for char in text.lower():

if char.isalpha() or char.isspace():

result += char

else:

result += " "

return result

def clean\_text\_fast(self, text):

"""Fast version using list comprehension and join"""

chars = [char if char.isalpha() or char.isspace() else ' '

for char in text.lower()]

return ''.join(chars)

def count\_words\_slow(self, text):

"""Slow word counting"""

words = text.split()

word\_counts = {}

unique\_words = list(set(words))

for word in unique\_words:

word\_counts[word] = words.count(word) # Slow!

return word\_counts

def count\_words\_fast(self, text):

"""Fast word counting"""

words = text.split()

word\_counts = {}

for word in words:

word\_counts[word] = word\_counts.get(word, 0) + 1

return word\_counts

def find\_common\_words\_fast(self, word\_counts, min\_count=5):

"""Use list comprehension for filtering"""

return [word for word, count in word\_counts.items() if count >= min\_count]

def analyze\_text(self, text, use\_optimizations=True):

"""Complete analysis with timing"""

if use\_optimizations:

clean\_time\_func = self.clean\_text\_fast

count\_time\_func = self.count\_words\_fast

else:

clean\_time\_func = self.clean\_text\_slow

count\_time\_func = self.count\_words\_slow

# Clean text

clean\_text, clean\_time = time\_function(clean\_time\_func, text)

# Count words

word\_counts, count\_time = time\_function(count\_time\_func, clean\_text)

# Find common words

common\_words, common\_time = time\_function(

self.find\_common\_words\_fast, word\_counts, 3

)

return {

'word\_count': len(clean\_text.split()),

'unique\_words': len(word\_counts),

'common\_words': len(common\_words),

'times': {

'cleaning': clean\_time,

'counting': count\_time,

'finding\_common': common\_time,

'total': clean\_time + count\_time + common\_time

}

}

# Demo the analyzer

sample\_text = """

Python is a great programming language. Python is easy to learn and use.

Many people choose Python because it's readable and powerful.

Learning Python opens many opportunities in technology careers.

""" \* 100 # Make it bigger to see timing differences

analyzer = TextAnalyzer()

print("=== Performance Comparison ===")

slow\_results = analyzer.analyze\_text(sample\_text, use\_optimizations=False)

fast\_results = analyzer.analyze\_text(sample\_text, use\_optimizations=True)

print(f"Slow methods total: {slow\_results['times']['total']:.4f} seconds")

print(f"Fast methods total: {fast\_results['times']['total']:.4f} seconds")

print(f"Speed improvement: {slow\_results['times']['total']/fast\_results['times']['total']:.1f}x faster")

#### ****Interactive Optimization Challenge:****

Give students a piece of inefficient code and have them optimize it:

# Challenge: Optimize this function

def find\_duplicates\_slow(numbers):

"""Find numbers that appear more than once"""

duplicates = []

for i, num in enumerate(numbers):

count = 0

for j, other\_num in enumerate(numbers):

if num == other\_num:

count += 1

if count > 1 and num not in duplicates:

duplicates.append(num)

return duplicates

# Students should optimize to something like:

def find\_duplicates\_fast(numbers):

"""Optimized version using sets and dictionary counting"""

seen = set()

duplicates = set()

for num in numbers:

if num in seen:

duplicates.add(num)

else:

seen.add(num)

return list(duplicates)

### ****8:55 PM -- 9:05 PM: Session 5 -- Performance Measurement and Best Practices****

**Objective:** Learn how to measure performance improvements and develop good optimization habits.

**Instructor Script:** "Optimization without measurement is just guessing. Let's learn how to properly measure performance and establish good practices for writing efficient code from the start."

#### ****Proper Performance Measurement:****

import time

import statistics

def measure\_performance(func, \*args, iterations=5):

"""Run a function multiple times and get reliable timing"""

times = []

for \_ in range(iterations):

start\_time = time.time()

result = func(\*args)

end\_time = time.time()

times.append(end\_time - start\_time)

return {

'result': result,

'mean\_time': statistics.mean(times),

'min\_time': min(times),

'max\_time': max(times),

'std\_dev': statistics.stdev(times) if len(times) > 1 else 0

}

# Example usage

def example\_slow\_function(n):

return sum([i\*\*2 for i in range(n)])

def example\_fast\_function(n):

return sum(i\*\*2 for i in range(n)) # Generator expression

# Compare with reliable measurement

slow\_stats = measure\_performance(example\_slow\_function, 10000)

fast\_stats = measure\_performance(example\_fast\_function, 10000)

print(f"Slow function: {slow\_stats['mean\_time']:.4f} ± {slow\_stats['std\_dev']:.4f} seconds")

print(f"Fast function: {fast\_stats['mean\_time']:.4f} ± {fast\_stats['std\_dev']:.4f} seconds")

#### ****Optimization Best Practices:****

1. **Measure First**: Always profile before optimizing
2. **Start with Algorithms**: Choose the right approach before micro-optimizing
3. **Focus on Bottlenecks**: Optimize the slowest parts first
4. **Readability Matters**: Don't sacrifice code clarity for tiny gains
5. **Test Thoroughly**: Ensure optimizations don't break functionality

#### ****Common Optimization Patterns Checklist:****

* ✅ Use list comprehensions instead of append loops
* ✅ Use sets for membership testing
* ✅ Use dictionaries for counting and lookups
* ✅ Use join() for string concatenation
* ✅ Use f-strings for formatting
* ✅ Pre-allocate lists when size is known
* ✅ Cache expensive calculations
* ✅ Use built-in functions (sum, max, min) over manual loops

### ****9:05 PM -- 9:25 PM: Hands-On Practice Session****

**Activity:** Students work on optimizing provided code examples and measuring improvements.

**Provided Exercises:**

1. Optimize a data processing function
2. Improve a text search algorithm
3. Speed up a mathematical calculation
4. Enhance a file processing script

### ****9:25 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today we've learned that performance optimization doesn't require advanced computer science knowledge. Small, thoughtful changes - our 'seed improvements' - can make dramatic differences in how fast your code runs. The key is developing good habits: choose the right data structures, use efficient built-in methods, and always measure your improvements."

**Review Key Points:**

* Small changes can yield big performance improvements
* The right data structure choice is crucial
* Built-in Python methods are usually optimized
* Always measure performance before and after changes
* Write efficient code as your default practice

**Prompting Question:** "What's one optimization technique you'll start using in your code right away?"

## ****After-Class Quiz (5 questions)****

1. What is the most efficient way to check if an item exists in a large collection of unique items?
   * A) Use a list and the in operator
   * \*B) Use a set and the in operator
   * C) Use a dictionary and check the keys
   * D) Loop through the collection manually
2. When you need to join many strings together, which method is most efficient?
   * A) Using the + operator in a loop
   * B) Using f-string formatting in a loop
   * \*C) Using the join() method
   * D) Using string multiplication
3. What makes list comprehensions faster than regular for loops with append?
   * A) They use less memory
   * B) They're easier to read
   * \*C) Python optimizes them internally
   * D) They can be parallelized
4. What's the main advantage of using dict.get(key, default) over checking if a key exists first?
   * A) It's more readable
   * \*B) It avoids looking up the key twice
   * C) It handles errors better
   * D) It uses less memory
5. Why is it important to measure performance before and after optimization?
   * A) To make sure the code still works
   * B) To document the changes
   * \*C) To verify that the optimization actually improved performance
   * D) To compare with other programmers' solutions