Kotlin and Python: A Comparative Performance Analysis

Thanh Cong Nguyen

Abstract

Performance of a programming language is crucial to developers as it directly impacts the efficiency, speed, and scalability of their software applications. Developers rely on performance evaluations to make informed decisions when selecting a language, enabling them to maximize resource utilization, minimize bottlenecks, deliver high-performing applications and ensure optimal user experience and responsiveness. Especially, in context of Ecommerce applications, where response times have direct revenue impact: Amazon discovered that even a slight 100-millisecond delay in response time leads to a 1% decrease in sales [1], while other studies have indicated that a mere 1-second slowdown can cause a significant 16% drop in customer satisfaction [2,3]. This technical report presents an empirical study that aims to compare the performance of Kotlin and Python.

1. Introduction

Kotlin and Python, two popular programming languages, have gained significant attention in recent years. In the realm of software development, performance is a critical factor that directly impacts the success of applications. As developers seek to create efficient and high-performing software, the choice of programming language plays a pivotal role. Kotlin, a statically typed language with full interoperability with Java, has emerged as a versatile choice for a wide range of applications. Its concise syntax, robust type inference, and modern features have attracted developers, particularly for development of Android applications and micro-services. Python, on the other hand, is a dynamically typed language known for its simplicity, readability, and extensive library ecosystem. Python's popularity has soared across domains such as data science due to its ease of use and community support. The outcomes of this research endeavor will serve as a valuable resource for software developers, architects, and decision-makers who are considering adopting Kotlin and Python for their projects.

The findings will facilitate informed decision-making by providing empirical evidence of the performance characteristics of these languages.

2. Methodology

The performance measurement was carried out using the following methodology:

- Objective: Measure execution times for computing common functions in Kotlin and Python.
- Functions: Three common functions, which include generating the last digit Fibonacci number sequence, performing heapsort, and estimating π using Monte Carlo simulation, were used for performance measurements.
- Setup: The operations were performed by utilizing Kotlin running on JVM version 1.8 and Python 3.6 on a MacBook Pro 15-inch 2019 model. The MacBook Pro is powered by a 2.6 GHz 6-Core Intel Core i7 processor, accompanied by 16 GB of 2400 MHz DDR4 RAM. The operating system in use is macOS Big Sur Version 11.6.
- Data collection: Execute the performance tests on both Kotlin and Python implementations of the same functions and collect performance data for each test run.
- Statistical Analysis: Apply statistical method of averaging to analyze the collected performance data.

3. Results

3.1. Last-digit Fibonacci sequence generation

- Kotlin version implementation:

Figure 1 Last-digit Fibonacci sequence - Kotlin

```kotlin
fun generateLastDigitFibonacciNumberSequence(n: Int): IntArray {
    val arr = IntArray(n)
    for (i in 0 until n) {
        arr[i] = if(i <= 1) i else (arr[i - 1] + arr[i - 2]) % 10
    }
    return arr
}
```

- Python version implementation:

Figure 2 Last-digit Fibonacci sequence - Python

```python
def generate_last_digit_fibonacci_number_sequence(n):
    arr = [0, 1]
    for i in range(2, n):
        arr.append((arr[i - 1] + arr[i - 2]) % 10)
    return arr
```

- The obtained results reveal an astounding difference, demonstrating that the Kotlin version performs a staggering 49 times faster than its Python counterpart, providing substantial evidence for the significant performance gap between the two programming languages:

Table 1 Result - last-digit Fibonacci sequence

	Array size	Test size	Average time
Python	100000000	100	19582 ms
Kotlin	100000000	100	399 ms

3.2. Heapsort

- Kotlin version implementation:

Figure 3 Heapsort - Kotlin

```kotlin
fun heapSort(arr: IntArray) {
    val priorityQueue = PriorityQueue<Int>()
    for (number in arr) {
        priorityQueue.offer(number)
    }
    for (i in arr.indices) {
        arr[i] = priorityQueue.poll()
    }
}
```

- Python version implementation:

Figure 4 Heapsort - Python

```python
def heap_sort(arr):
    priority_queue = PriorityQueue()
    for num in arr:
        priority_queue.put(num)
    for i in range(len(arr)):
        arr[i] = priority_queue.get()
```

- The results of the performance tests conducted on the heapsort function show that the Kotlin version performs 31 times faster than the Python version, thus reinforcing the significant performance advantage that Kotlin has over Python:

Table 2 Performance result - Heapsort

	Array size	Test size	Average time
Python	1000000	100	4310 ms
Kotlin	1000000	100	139 ms

3.3. Estimate π - Monte Carlo Simulation

- Kotlin version implementation:

Figure 5 Estimate π - Kotlin

```kotlin
fun estimatePiUsingMonteCarloSimulation(n: Int): Double {
    var numberOfPointsInside = 0
    for (i in 0 until n) {
        val x = Math.random()
        val y = Math.random()
        val distance = sqrt( x: x * x + y * y)
        if (distance <= 1) {
            numberOfPointsInside++
        }
    }
    return 4.0 * numberOfPointsInside / n
}
```

- Python version implementation:

Figure 6 Estimate π - Python

```python
def estimate_pi_using_monte_carlo_simulation(n):
    number_of_points_inside = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        distance = x * x + y * y
        if distance <= 1:
            number_of_points_inside += 1
    return 4.0 * number_of_points_inside / n
```

- Once again, the results of the performance tests show that the Kotlin version outperformed the Python version by a factor of 7.69 times in the estimation of π using Monte Carlo Simulation:

Table 3 Performance results - π estimation

	N	Test size	Average time
Python	100000000	100	27897 ms
Kotlin	100000000	100	3983 ms

4. Discussion

In this section, I present a discussion on the performance tests conducted between Kotlin and Python, as well as considerations regarding their respective ecosystems and libraries.

Performance: Kotlin, being statically typed and compiled to bytecode, exhibits faster execution times compared to Python, which is dynamically typed and interpreted. Kotlin's compiled nature allows it to take advantage of optimizations during the compilation process, resulting in improved performance. Python's interpreted nature, on the other hand, can introduce some overhead, leading to slower execution times, especially in computationally intensive tasks.

Ecosystem and Libraries: The ecosystem and availability of libraries greatly influence the efficiency and productivity of developers. Python boasts an extensive ecosystem with a wide range of libraries and frameworks. This rich ecosystem contributes to Python's versatility and makes it a popular choice for diverse tasks such as data analysis, and machine learning. Kotlin, being a relatively newer language, has a growing ecosystem, but it may not possess the same breadth and depth of libraries as Python. However, Kotlin can leverage the vast collection of existing Java libraries, providing access to numerous well-established and performant solutions.

Thus, the choice between Kotlin and Python should be made based on the specific requirements of the research project, considering factors such as performance, development speed, available libraries, and the trade-offs between them. Researchers and developers should carefully evaluate the unique needs of their projects to select the most suitable language for optimal performance and productivity. For tasks where computing time is not as important as developing time such as numerical computations, data analysis and scientific simulations, Python is a good choice thanks to its extensive library ecosystem, simplicity and versatility. On the other hand, Kotlin's performance advantages become more pronounced in scenarios where computational efficiency is crucial, such as Ecommerce applications, algorithmic trading, or large-scale data processing.

5. Conclusion

In this report, I conducted a performance comparison between Kotlin and Python, two popular programming languages. My findings indicate that Kotlin generally outperforms Python in terms of execution speed. Kotlin's static typing and compilation to bytecode provide inherent advantages, resulting in faster execution times compared to Python's dynamic typing and interpretation. However, it is

important to note that Python excels in other areas, such as its extensive ecosystem and vast collection of libraries. The availability of these libraries enhances development productivity. Ultimately, the choice between Kotlin and Python for a specific project should consider the trade-offs between performance, ecosystem, and development productivity. If computational efficiency is paramount, Kotlin is a favorable choice. On the other hand, Python's extensive library ecosystem and ease of use make it a versatile language suitable for a wide range of applications.

As future research directions, it would be valuable to delve deeper into memory management for both languages. Memory management is a critical aspect of programming languages and can impact overall performance. Kotlin and Python employ different memory management approaches. Kotlin, similar to Java, utilizes automatic memory management through garbage collection. Python, on the other hand, combines garbage collection with a reference counting mechanism. While Python's reference counting can introduce additional memory overhead, both languages generally handle memory management efficiently, and the performance impact is typically not significant for most applications.

Overall, this technical report provides valuable insights into the performance characteristics of Kotlin and Python, empowering researchers and developers to make informed decisions based on their specific project requirements, balancing performance considerations with ecosystem support and development productivity.

References

[1] Greg Linden: “Make Data Useful,” slides from presentation at Stanford University Data Mining class (CS345), December 2006.

[2] Tammy Everts: “The Real Cost of Slow Time vs Downtime,” slideshare.net, November 5, 2014.

[3] Jake Brutlag: “Speed Matters,” ai.googleblog.com, June 23, 2009.