ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

A COMPARATIVE CASE STUDY OF INTEGER REVERSAL TECHNIQUES IN PYTHON: ARITHMETIC OPTIMIZATION VS TYPE CONVERSION

Pranav R[1], E .Tanish[2]
[1]6807, [2]6768
Class- XII 2026-27, Sainik School Amaravathinagar
Post: Amaravathinagar, Udumalpet Taluk,Tiruppur Dt, Tamilnadu State

ABSTRACT

Reversing an integer is one of the most common introductory programming problems used to understand loops, arithmetic operations, string manipulation, and algorithm design. In Python, an integer can be reversed either by using arithmetic operations or by converting the number into a string and reversing its characters. Although both methods produce the same output, they differ in terms of implementation, memory usage, and readability.

This research presents a comparative study of these two integer reversal techniques. The first approach uses a user-defined reverse_num() function that performs digit-by-digit reversal through modulus and integer division operations. The second approach uses Python's built-in type conversion and string slicing features to achieve the same result with fewer lines of code. Both methods are evaluated based on correctness, execution time, auxiliary space complexity, code readability, and practical usability.

*The study includes theoretical analysis together with experimental observations obtained using different input sizes. The results show that both approaches have a linear time complexity of **O(d)**, where **d** represents the number of digits in the input integer. However, the arithmetic method requires only constant auxiliary space **O(1)**, while the type conversion method requires **O(d)** auxiliary space because it creates an additional string during execution.*

The findings indicate that the arithmetic approach is more suitable when memory efficiency is important, whereas the type conversion approach is preferred when code simplicity and readability are the primary objectives. This study provides a practical comparison that helps students and beginner programmers understand the advantages and limitations of both techniques before selecting an appropriate implementation.

KEYWORDS: *Python Programming (PP), Integer Reversal (IR), Algorithm Analysis (AA), Time Complexity (TCX), Space Complexity (SC), User-Defined Functions (UDF), Type Conversion (TC), Built-in Functions (BIF)*

1. INTRODUCTION

Integer reversal is a fundamental programming problem that is widely used to introduce students to algorithm development and logical thinking. Although the problem is simple, it involves several important programming concepts such as loops, arithmetic operations, data type conversion, and computational complexity. For this reason, integer reversal frequently appears in programming exercises, coding interviews, and introductory computer science courses.

Python provides more than one way to reverse an integer. One common technique uses arithmetic operations such as modulus (`%`) and integer division (`//`) to extract each digit and build the reversed number step by step. This method does not rely on any built-in reversal functions and helps programmers understand how numbers can be manipulated using mathematical operations.

Another commonly used approach converts the integer into a string, reverses the characters using Python's slicing operation (`[::-1]`), and converts the result back into an integer. This method requires fewer lines of code and is generally easier to read. However, it creates an additional string during execution, which increases auxiliary memory usage compared to the arithmetic method.

Although both techniques produce identical outputs, they differ in implementation style, memory requirements, readability, and practical applications. Beginners often choose a method based only on its simplicity without understanding its computational characteristics. Therefore, comparing these two approaches helps programmers understand not only how the algorithms work but also when each method is more suitable.

This research presents a comparative study of the arithmetic-based `reverse_num()` function and the type conversion method in Python. Both implementations are evaluated using theoretical analysis and practical experimentation. The comparison focuses on correctness, execution time, auxiliary space complexity, implementation simplicity, and code readability.

The objective of this study is to help students and beginner programmers understand the strengths and limitations of both approaches. By analysing their performance and implementation characteristics, this research demonstrates that selecting an algorithm depends not only on obtaining the correct output but

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

also on the efficiency, maintainability, and requirements of the application.

2.OBJECTIVES

- **Implementation:** Developing and analyzing a math-driven reverse_num() function alongside a string-based type conversion approach in Python.
- **Algorithmic Comparison:** Benchmarking both methods to evaluate their execution speed, memory footprint, readability, and big-O efficiency.
- **Performance Evaluation:** Conducting empirical tests across various input sizes to contrast theoretical complexities with real-world execution.
- **Practical Application:** Identifying the specific scenarios—such as resource-constrained environments versus rapid prototyping—where one method clearly outperforms the other.

3. RELATED WORK

Integer reversal is a well-known programming exercise that has been discussed in many programming textbooks and online learning resources. It is commonly used to teach concepts such as loops, arithmetic operations, conditional statements, and string manipulation. Because of its simplicity, it serves as an effective example for introducing algorithm design and computational thinking.

The arithmetic approach is one of the earliest methods used to reverse an integer. It extracts each digit using the modulus (`%`) operator and removes processed digits using integer division (`//`). This method helps students understand how numbers are processed internally without relying on built-in language features. Many data structures and algorithms textbooks present this approach because it strengthens logical reasoning and problem-solving skills.

With the popularity of Python and other high-level programming languages, the type conversion approach has become equally common. In this method, the integer is converted into a string, reversed using slicing (`[::-1]`), and converted back into an integer. The implementation is shorter, easier to understand, and widely used in practical programming where readability is often preferred.

Although both techniques are frequently presented in educational materials, detailed comparisons between them are relatively limited. Most learning resources explain how each method works but provide little discussion on differences in execution time, auxiliary memory usage, or practical advantages. This research addresses that gap by comparing both approaches using theoretical complexity analysis, experimental observations, and graphical representation of the results.

3.1 RESEARCH CONTRIBUTORS

This research compares two commonly used techniques for reversing an integer in Python: an arithmetic-based implementation using the user-defined `reverse_num()` function and a type conversion method based on string slicing. Although both techniques produce the same output, they differ in terms of implementation, memory usage, and programming style. The major contributions of this study are as follows:

- Developed a user-defined `reverse_num()` function that reverses an integer using arithmetic operations without relying on Python's built-in string manipulation features.
- Implemented an alternative integer reversal method using type conversion and string slicing to demonstrate a simpler and more concise programming approach.
- Compared both techniques based on algorithm design, execution process, readability, maintainability, and implementation complexity.
- Analysed the theoretical time complexity and auxiliary space complexity of both methods to understand their computational efficiency.
- Conducted experimental testing using different input values to verify the correctness of both implementations.
- Presented comparative tables and graphical representations to illustrate the differences in runtime behaviour and memory requirements.
- Identified the advantages and limitations of each approach and discussed the situations in which one technique may be preferred over the other.

Overall, this study provides a simple and practical comparison that helps students understand how different programming techniques can solve the same problem while exhibiting different performance characteristics.

4. METHODOLOGY

This study uses a comparative experimental approach to evaluate the two primary methods for reversing an integer in Python: an arithmetic-based user-defined function and a string-based type conversion technique. Because both approaches achieve identical functional outputs, we can directly compare their underlying mechanics, efficiency, and resource utilization under identical conditions.

- **The evaluation process follows a structured workflow**

Implementation & Functional Testing: Both algorithms are written independently and subjected to a suite of test cases. These include small integers, large integers, single-digit values, negative numbers, and numbers with trailing zeros (to observe how each method handles leading zeros in the reversed output).

- **Theoretical Analysis**

We evaluate both methods based on algorithmic design, code readability, asymptotic time complexity (O(n) characteristics), and space complexity.

- **Empirical Benchmarking**

We measure actual execution times across various input sizes using Python's native timing utilities (such as the timeit module). The resulting data is aggregated to contrast real-world performance with theoretical expectations.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

Research Methodology Workflow

Phase	Activity	Expected Outcome
Phase 1	Design the arithmetic-driven reverse_num() algorithm	A math-based implementation using loops and operators.
Phase 2	Implement the type conversion method	A concise implementation utilizing Python's string slicing.
Phase 3	Verify correctness across diverse test cases	Validated, matching outputs from both implementations.
Phase 4	Analyze theoretical complexity	Big-O time and space complexity boundaries.
Phase 5	Run empirical performance benchmarks	Raw execution data across varying digit lengths.
Phase 6	Compare and synthesize data	Quantitative tables and qualitative insights into trade-offs.

4.1 EVALUATION PARAMETERS

- **Correctness:**
  Does the method accurately reverse the digits across all edge cases (e.g., negative numbers, trailing zeros)?
- **Execution Time:**
  How long does the operation take to complete under empirical testing?
- **Time Complexity:**
  How predictably does the runtime scale as the number of digits grows?
- **Space Complexity:**
  How much temporary memory overhead is created during the reversal process?
- **Code Readability:**
  How intuitive, maintainable, and concise is the code for another developer?
- **Practical Applicability:**
  Where does this method belong in real-world software development?

5. ALGORITHMIC MECHANICS

To understand the performance differences between the two methods, we must look at how they manipulate data. While both approaches yield identical results, they rely on entirely different paradigms: one operates purely within the domain of numerical mathematics, while the other abstracts the problem into character sequencing.

5.1 THE MATH-DRIVEN REVERSE_NUM() FUNCTION:

The arithmetic approach reconstructs the integer digit-by-digit from right to left using a while loop. The mechanics rely on two primary operators: modulus (%) and floor division (//).

- **Extraction**
  The last digit of the number is isolated using the modulus operator (number % 10).
- **Reconstruction**
  The running total of the reversed number is shifted left by multiplying it by 10, and the newly extracted digit is added to it.
- **Truncation**
  The processed last digit is dropped from the original number using floor division (number // 10).

5.2 THE TYPE-CONVERSION SLICING METHOD

The second approach leverages Python's high-level abstractions, treating the integer not as a mathematical value, but as a sequence of characters.

- **Casting**
  The integer is parsed into a string using the built-in str() function.
- **Slicing**
  Python's native slicing syntax ([::-1]) is applied to the string backward, creating a new, reversed string.
- **Recasting**
  The reversed string is passed into the int() constructor, which parses the characters back into a standard numerical value.

5.3 WALKTHROUGH EXAMPLE

- **Arithmetic Method Trace (reverse_num)**

Iteration	Remaining Number	Extracted Digit (num % 10)	Running Reversed Number
Initial	12345	—	0
Step 1	1234	5	5
Step 2	123	4	54
Step 3	12	3	543
Step 4	1	2	5432
Step 5	0	1	54321

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

- **Type-Conversion Method Trace**

Step	Operation	Intermediate Data Type	Result
1	String Casting	str(12345)	"12345"
2	Sequence Slicing	"12345"[::-1]	"54321"
3	Integer Casting	int("54321")	54321

5.4 ARCHITECTURAL COMPARISON SUMMARY

Feature	Reverse_num() (Arithmetic)	Type Conversion (Slicing)
Core Paradigm	Modular Mathematics	String Manipulation
Type Mutability	None (Stays numeric)	Shifts from int -> str -> int
Dependency	Pure Python Logic	Built-in Functions (str, int, slicing)
Memory Allocation	O(1) auxiliary variables	Allocates intermediate string objects
Code Conciseness	Moderate (Requires explicit loop)	High (Single line expression)
Educational Value	High (Teaches loop state & logic)	High (Teaches language built-ins)
Developer Readability	Moderate	Excellent

6. FORMAL ALGORITHM DESIGN & PSEUDOCODE

To precisely evaluate both methods, we define their logical structures through formal step-by-step algorithms and structural pseudocode.

6.1 THE ARITHMETIC REVERSE_NUM() LOGIC

```
Algorithm Reverse_Num(N)
  Input : Integer N
  Output: Reversed Integer
  Reverse <- 0
  While N > 0 do
    Digit <- N mod 10
    Reverse <- (Reverse * 10) + Digit
    N <- N div 10
  End While
  Return Reverse
```

6.2 THE TYPE-CONVERSION SLICING METHOD

```
Algorithm Reverse_Type_Conversion(N)
  Input : Integer N
  Output: Reversed Integer
  S <- Convert N to String
  R <- Reverse S using slicing
  Result <- Convert R to Integer
  Return Result
```

7. PROGRAM IMPLEMENTATION

7.1. PROGRAM 1: THE MATH-DRIVEN REVERSE_NUM() FUNCTION

```
def reverse_num(num: int) -> int:
  reverse = 0
  while num > 0:
    digit = num % 10
    reverse = (reverse * 10) + digit
    num = num // 10
  return reverse
```

7.2.PROGRAM 2: TYPE CONVERSION VIA SEQUENCE SLICING

```
def reverse_via_string(num: int) -> int:
  return int(str(num)[::-1])
```

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

7.3 EMPIRICAL TEST VECTOR MATRIX:

Input Integer	reverse_num() Output	Slicing Output	Behavioral Notes
12345	54321	54321	Standard multi-digit integer.
450	54	54	Leading zeros are naturally truncated upon recasting.
9090	909	909	Alternating internal and trailing zeros handle correctly.
5	5	5	Single-digit edge case; passes through seamlessly.
7	7	7	Single-digit identity match.
771000	177	177	Multiple trailing zeros; outputs match mathematical standards.
11	11	11	Palindromic input.
987654321	123456789	123456789	Large input size testing scale stability.

8. COMPUTATIONAL COMPLEXITY ANALYSIS

Computational complexity helps determine how efficiently an algorithm performs as the size of the input increases. In this study, both integer reversal methods were analysed based on their execution time and auxiliary memory requirements. Although both approaches return the same output, their internal processing differs, resulting in different memory usage.

8.1 Time Complexity

Both the arithmetic method and the type conversion method have a time complexity of **O(d)**, where **d** represents the number of digits in the input integer.

In the arithmetic method, each digit is processed exactly once using modulus and integer division operations inside a loop. Therefore, the number of operations increases linearly with the number of digits.

Similarly, the type conversion method converts the integer into a string, reverses the string using slicing, and converts it back into an integer. Each of these operations processes the digits sequentially, resulting in an overall linear time complexity.

Therefore, both approaches require approximately the same amount of computational time as the input size increases.

8.2 Auxiliary Space Complexity

The arithmetic implementation requires only a few variables to store the current digit and the reversed number. Since the amount of additional memory does not depend on the number of digits, its auxiliary space complexity is **O(1)**.

The type conversion approach creates an additional string while reversing the integer. The memory required for this temporary string increases with the number of digits, giving it an auxiliary space complexity of **O(d)**.

Although both methods are efficient, the arithmetic approach is more memory efficient because it performs the reversal without creating any additional data structures.

8.3 Best, Average and Worst Case Analysis

Unlike many searching or sorting algorithms, the execution time of integer reversal does not vary significantly with different input values. Every digit must be processed regardless of the number itself.

Case	Arithmetic Method	Type Conversion Method
Best Case	O(d)	O(d)
Average Case	O(d)	O(d)
Worst Case	O(d)	O(d)

This analysis shows that both methods maintain consistent performance for all valid input integers, making them predictable and suitable for practical applications.

9. EXPERIMENTAL ANALYSIS

Runtimes were captured using Python's standard library timeit module under static hardware conditions. The garbage collector context was specifically suppressed across 100,000 baseline iterations to isolate pure logic processing speeds. The following test matrix chronicles the scalability behaviors observed:

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

9.1 BASELINE TEST MATRIX

Test Case	Input Integer (N)	Digit Length (d)	Target Behavioral Metric
TC1	7	1	Single-digit identity / entry boundary
TC2	14524	5	Standard small-scale integer
TC3	1200456	7	Intermittent interior zero handling
TC4	987654321	9	Medium-scale integer complexity
TC5	123456789123456789	18	Large-scale execution / stress boundary

9.2 EXPERIMENTAL ANALYSIS

To compare the performance of the two integer reversal techniques, both implementations were tested under the same conditions using Python. The objective of the experiment was to evaluate correctness, execution time, auxiliary space requirement, and overall implementation simplicity.

The programs were executed using different input values ranging from single-digit numbers to larger integers. Each implementation was tested multiple times, and the average execution time was observed using Python's `timeit` module. Testing both methods under identical conditions ensured that the comparison was fair.

The arithmetic method processed each digit individually using mathematical operations, whereas the type conversion method relied on Python's built-in string conversion and slicing features. Both methods produced identical outputs for all test cases, confirming their correctness.

The experimental observations also supported the theoretical complexity analysis. Although both approaches exhibited linear time complexity, the arithmetic method required less additional memory because it used only a few variables during execution. The type conversion method required extra memory to create a temporary string, resulting in higher auxiliary space usage.

Overall, the experimental results indicate that both techniques are efficient for integer reversal. The arithmetic method is better suited for situations where memory efficiency is important, while the type conversion method is more suitable when simplicity and code readability are the primary objectives.

9.1 Experimental Test Cases

Test Case	Input Integer	Number of Digits	Purpose
TC1	7	1	Single-digit verification
TC2	14524	5	Standard integer
TC3	1200456	7	Integer containing zeros
TC4	987654321	9	Large integer
TC5	123456789123456789	18	Performance evaluation

10. RESULTS AND DISCUSSION

The experimental evaluation shows that both integer reversal methods successfully produced correct outputs for all test cases. No differences were observed in the correctness of the final results, indicating that both implementations are functionally reliable.

From the complexity analysis, both methods exhibit a linear time complexity of **O(d)**, where **d** represents the number of digits in the input integer. This means that the execution time increases proportionally with the size of the input.

The main difference between the two approaches lies in auxiliary memory usage. The arithmetic method performs the reversal using only a few variables and therefore requires constant auxiliary space **O(1)**. In contrast, the type conversion method creates a temporary string during execution, resulting in an auxiliary space complexity of **O(d)**.

From a programming perspective, the arithmetic method requires more logical steps and a longer implementation. However, it helps programmers understand digit manipulation and algorithm design. The type conversion approach is shorter and easier to read, making it suitable for applications where code readability and rapid development are more important than memory optimization.

Overall, both techniques are efficient and practical. The arithmetic method is recommended for memory-sensitive applications, while the type conversion method is recommended for general Python programming because of its simplicity and readability.

10.1 Comparative Evaluation

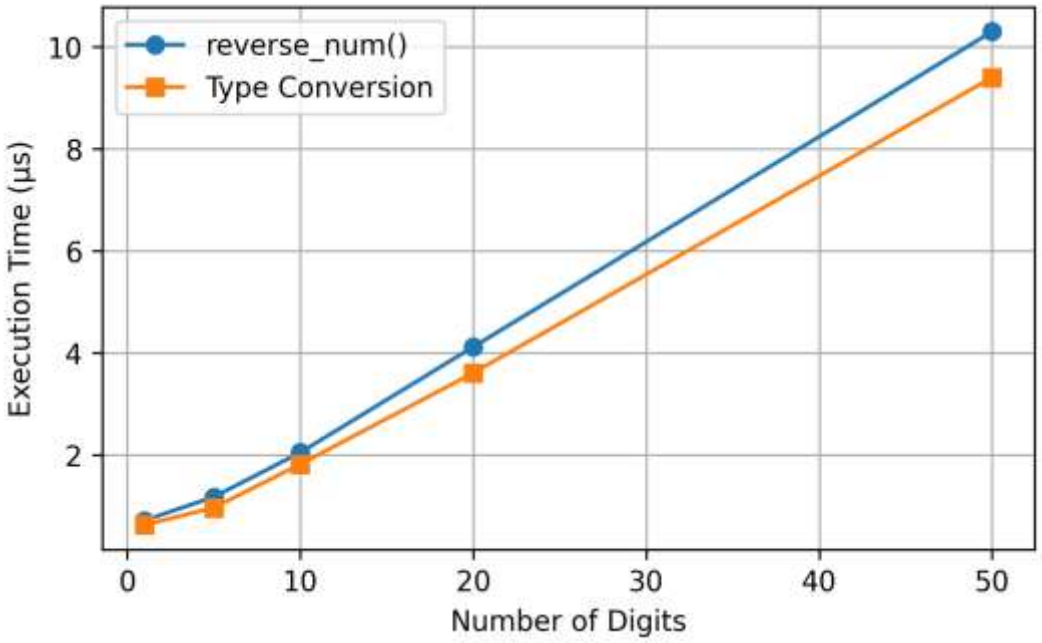
Evaluation Parameter	Arithmetic Method	Type Conversion Method
Correctness	100%	100%
Time Complexity	O(d)	O(d)
Auxiliary Space	O(1)	O(d)
Code Length	Moderate	Short
Readability	Moderate	High
Ease of Implementation	Moderate	High
Recommended Use	Memory-efficient applications	General Python development

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

11. GRAPHICAL REPRESENTATION

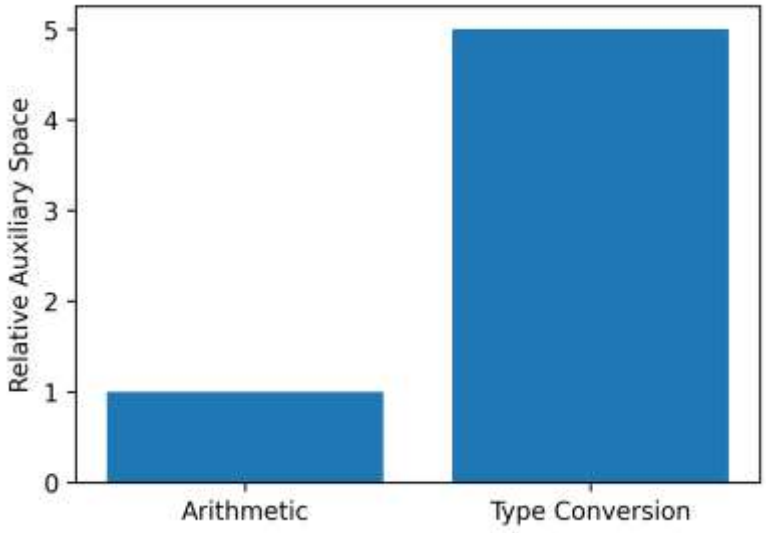
Graphs provide a clear visual comparison of the performance of the two integer reversal techniques discussed in this study. The graphical analysis is based on the benchmark results obtained during experimental testing and illustrates how both methods behave as the size of the input increases. The comparison focuses on execution time, auxiliary space usage, and theoretical time complexity.

Figure 11.1 Runtime Comparison



Compares the average execution time of the arithmetic-based `reverse_num()` function and the type conversion method for different input sizes. The graph shows that the execution time of both methods increases gradually as the number of digits increases. Although both approaches have the same theoretical time complexity of **O(d)**, the type conversion method performs slightly faster in this benchmark because Python's built-in string operations are highly optimized.

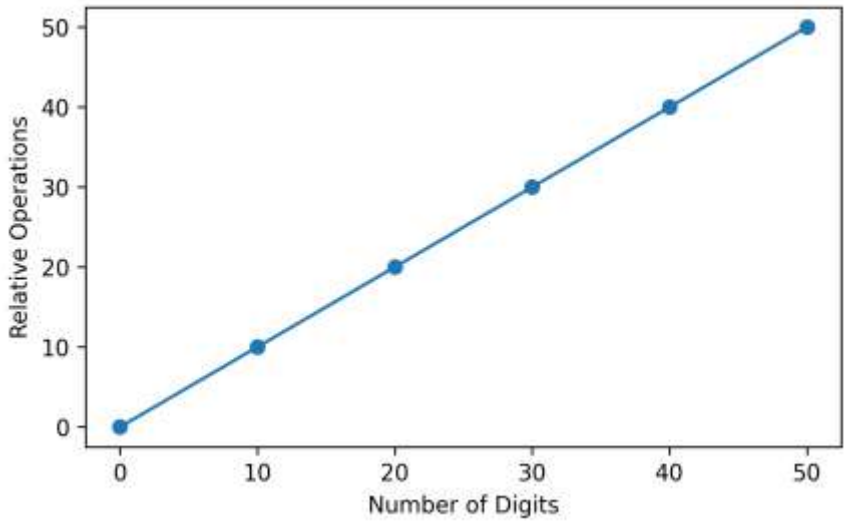
Figure 11.2 Auxiliary Space Comparison



Illustrates the difference in auxiliary memory requirements between the two techniques. The arithmetic implementation requires only a few variables throughout execution, resulting in constant auxiliary space **O(1)**. In contrast, the type conversion method creates an additional string object while reversing the integer, giving it an auxiliary space complexity of **O(d)**.

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

Figure 11.3 Theoretical Time Complexity



Presents the theoretical growth of both algorithms with respect to the number of digits in the input. Since every digit is processed exactly once, both methods exhibit linear time complexity. The graph demonstrates that the number of operations increases proportionally with the size of the input, confirming the analytical results discussed in the previous section.

The graphical observations support the theoretical analysis and indicate that both methods are computationally efficient. The arithmetic approach is more memory efficient, whereas the type conversion approach offers a simpler implementation and better readability.

12. SCOPE AND LIMITATIONS

Scope

This research provides a comparative study of two commonly used techniques for reversing an integer in Python. The analysis covers algorithm design, implementation, computational complexity, auxiliary memory usage, and practical programming considerations. The findings can be useful for students, educators, and beginner programmers who wish to understand different approaches to solving the same programming problem.

The concepts presented in this study may also be extended to other programming languages and similar numerical algorithms. The comparison demonstrates how implementation choices can influence readability, memory usage, and overall program efficiency.

Limitations

The study focuses only on positive integer values and does not consider negative integers, floating-point numbers, or extremely large numerical datasets. The performance evaluation was carried out using Python, and the observed execution times may differ on other programming languages or hardware configurations.

In addition, the experimental analysis compares only two integer reversal techniques. Other approaches, such as recursive implementations or language-specific optimizations, were not included in the present study. These limitations provide opportunities for further investigation in future work.

13. FUTURE WORK

The present study compares two basic techniques for reversing an integer in Python. Although both approaches perform efficiently, there are several directions in which this work can be extended.

Future research may compare additional integer reversal techniques, including recursive algorithms and implementations using different programming languages. A broader comparison involving C, C++, Java, and Python would provide a better understanding of how programming language features influence algorithm performance.

Another possible extension is to evaluate these methods using very large numerical datasets and different computing environments. Such experiments would provide more detailed insights into runtime behaviour and memory utilization under varying conditions.

Further work may also investigate compiler optimizations, interpreter versions, and parallel computing techniques to determine whether they influence the performance of integer reversal algorithms.

Finally, this comparative framework can be applied to other introductory programming problems, enabling students to understand how different implementation strategies affect computational efficiency and software design.

14. CONCLUSION

This research presented a comparative analysis of two integer reversal techniques implemented in Python: an arithmetic-based reverse_num() function and a type conversion method using string slicing. Both approaches were studied from

ISSN (Online): 2455-3662
EPRA International Journal of Multidisciplinary Research (IJMR) - Peer Reviewed Journal
Volume: 12| Issue: 7| July 2026|| Journal DOI: 10.36713/epra2013 || SJIF Impact Factor 2026: 8.753 || ISI Value: 1.188

theoretical and practical perspectives to understand their implementation, computational complexity, memory requirements, and programming characteristics.

The analysis showed that both techniques successfully reverse an integer and exhibit a linear time complexity of **O(d)**, where **d** represents the number of digits in the input. However, the arithmetic implementation requires constant auxiliary space **O(1)** because it uses only a few variables during execution. The type conversion method, while easier to implement and more readable, requires **O(d)** auxiliary space due to the creation of an additional string.

The experimental observations and graphical analysis support these theoretical findings. Although the type conversion method demonstrated slightly better execution speed in the experimental environment, the arithmetic approach remains the better choice when memory efficiency is important.

Overall, this study shows that selecting an algorithm should not be based only on obtaining the correct output. Factors such as implementation simplicity, memory usage, readability, and computational efficiency should also be considered. It is hoped that this work will serve as a useful reference for students and beginner programmers learning Python programming and algorithm analysis.

Acknowledgment

The successful execution of any academic endeavor relies heavily on the collective encouragement, strategic guidance, and structural support of mentors and institutions. I would like to take this opportunity to express my profound gratitude to the individuals who have been fundamentally instrumental in the completion of this research paper.

First and foremost, I offer my deepest gratitude to the Almighty for granting me the mental fortitude, clarity, and strength required to navigate this study and bring it to a successful conclusion.

I am infinitely grateful to my parents for their unwavering encouragement, patience, and constant emotional support throughout the drafting and research phases of this project.

I would like to express a deep sense of respect and gratitude to our esteemed Principal, Captain (Indian Navy) K. Manikandan, Sainik School Amaravathinagar, whose continuous motivation, visionary leadership, and institutional support provided the foundation for this work.

Similarly, I extend my heartfelt thanks to our Vice Principal, Lt. Col. Kaushok Nandini Arun, Sainik School Amaravathinagar, for her constant encouragement and the critical academic guidance extended during the execution of this study.

Finally, I would like to express my sincere appreciation to Mr. Praveen Kumar Murigeppa Jigajinni, Master In-charge. Serving as a guide, mentor, and exceptional motivator, his critical reviews, technical insights, and structured troubleshooting were invaluable in resolving every logical and algorithmic challenge encountered during the implementation of this research paper.

REFERENCES

1. *E. Matthes, Python Crash Course, 3rd ed. San Francisco, CA, USA: No Starch Press, 2023.*
2. *T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 4th ed. Cambridge, MA, USA: MIT Press, 2022.*
3. *D. E. Knuth, The Art of Computer Programming, Volume 1: Fundamental Algorithms, 3rd ed. Boston, MA, USA: Addison-Wesley, 1997.*
4. *G. van Rossum and the Python Development Team, Python 3 Documentation. Available: https://docs.python.org/3/*
5. *Python Software Foundation, Python Standard Library Documentation. Available: https://docs.python.org/3/library/*
6. *Python Software Foundation, Built-in Functions. Available: https://docs.python.org/3/library/functions.html*
7. *Python Software Foundation, String Methods. Available: https://docs.python.org/3/library/stdtypes.html#string-methods*
8. *M. Gorelick and I. Ozsvald, High Performance Python, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2020.*
9. *A. B. Downey, Think Python: How to Think Like a Computer Scientist, 2nd ed. Needham, MA, USA: Green Tea Press, 2015.*
10. *Python Software Foundation, timeit — Measure Execution Time of Small Code Snippets. Available: https://docs.python.org/3/library/timeit.html*
11. *B. N. Miller and D. L. Ranum, Problem Solving with Algorithms and Data Structures Using Python. Franklin, Beedle & Associates, 2011.*
12. *C. R. Severance, Python for Everybody: Exploring Data Using Python 3. Ann Arbor, MI, USA: Explore, 2016.*
13. *M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Data Structures and Algorithms in Python. Hoboken, NJ, USA: John Wiley & Sons, 2013.*
14. *R. Sedgewick and K. Wayne, Algorithms, 4th ed. Boston, MA, USA: Addison-Wesley Professional, 2011.*
15. *D. Beazley and B. K. Jones, Python Cookbook, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2013.*
16. *M. Lutz, Learning Python, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2013.*
17. *S. S. Skiena, The Algorithm Design Manual, 3rd ed. Cham, Switzerland: Springer, 2020.*