

C Python Internals: Method Resolution Order

Krish Mohan Muruganandam¹, Abhishek Panda², Aarabh Raj Singh³

^{1,2,3}*Department of Computer Science and Engineering Sharda University Greater Noida, India*

Abstract—Method Resolution Order (MRO) is a fundamental component of Python's object model that determines the sequence in which classes are searched for methods and attributes during inheritance. Python 2.2 introduced a new-style class model with an MRO algorithm designed to improve inheritance behavior, but its limitations, particularly the violation of monotonicity, motivated the adoption of C3 linearization in Python 2.3. C3 has since remained the basis of method resolution in modern Python. Understanding the differences between these algorithms provides valuable insight into the evolution of Python's object model and the design principles underlying multiple inheritance.

This research aims to conduct a comparative analysis of Python 2.2's new-style-class MRO algorithm and C3 linearization, focusing on inheritance consistency, monotonicity, method resolution behavior, and computational performance. The study will implement both algorithms in a controlled experimental environment and evaluate their behavior across a range of generated inheritance hierarchies, including deep, wide, and complex multiple-inheritance structures. The evaluation will examine differences in resolution order, violations of inheritance precedence constraints, algorithmic execution time, and memory consumption as hierarchy complexity increases.

The expected outcome is a systematic characterization of the strengths and limitations of both algorithms, supported by experimental measurements and representative inheritance examples. The research will also investigate the relationship between correctness guarantees and computational cost, providing insight into the trade-offs involved in designing method resolution algorithms. By combining historical analysis with algorithmic evaluation, this study seeks to contribute a reproducible framework for comparing inheritance-resolution strategies and a deeper understanding of the design decisions that shaped Python's current object model.

Index Terms—CPython, Method Resolution Order, C3 Linearization, Multiple Inheritance, Monotonicity, Object Model, Algorithm Analysis.

I. INTRODUCTION

Python is a high-level, general-purpose programming language that supports object-oriented programming through classes, objects, inheritance, and polymorphism. Inheritance enables a class to reuse and extend the functionality of other classes, reducing code duplication and supporting modular software design. Although single inheritance establishes a relatively straightforward relationship between classes, multiple inheritance allows a class to inherit from several parent classes simultaneously. This introduces additional complexity because the interpreter must determine which class should be searched first when multiple classes define the same method or attribute. Python addresses this problem through the Method Resolution Order (MRO), which establishes the sequence in which classes are examined during attribute and method lookup.

The design of Python's MRO has evolved alongside its object model. In Python's earlier classic-class system, method resolution followed a depth-first, left-to-right search strategy. Python 2.2 introduced new-style classes, which were based on a unified type and object model and used a different method resolution algorithm. Although this algorithm improved certain aspects of inheritance behavior, it had limitations concerning monotonicity. Monotonicity is an important property of an inheritance system because it ensures that extending a class hierarchy does not reverse the established relative precedence of classes. Violations of this property can lead to surprising method resolution behavior, particularly in complex multiple-inheritance hierarchies.

To address these limitations, Python 2.3 adopted C3 linearization for new-style classes. C3 determines a consistent linear ordering of a class and its ancestors while respecting the declared order of parent classes and the established ordering of their ancestors. It preserves monotonicity and rejects inheritance hierarchies for which no valid linearization can be

constructed under its constraints. This provides predictable method resolution and establishes the foundation of the MRO mechanism used by modern Python. The transition from Python 2.2's algorithm to C3 represents an important development in the evolution of Python's inheritance model.

Despite the practical importance of this transition, understanding the differences between the two algorithms requires more than examining their resulting method orders. Their behavior can be evaluated in terms of inheritance consistency, monotonicity, precedence preservation, computational complexity, and resource consumption. For example, two algorithms may produce identical results for simple inheritance structures but differ significantly when confronted with complex hierarchies involving shared ancestors or conflicting precedence requirements. Similarly, an algorithm's correctness guarantees and its computational cost represent distinct dimensions that must be examined independently.

II. COMPARATIVE EVALUATION OF BOTH ALGORITHMS

A systematic comparison of Python 2.2's algorithm and C3 requires both theoretical analysis and controlled experimentation. The research can implement each algorithm independently and test it against generated inheritance hierarchies with varying depths, numbers of parent classes, and levels of structural complexity.

The evaluation will focus on four principal measures: correctness of the resulting ordering, preservation of inheritance precedence, computational execution time, and memory consumption. Correctness tests will determine whether the algorithms preserve their intended ordering constraints, while performance experiments will measure the cost of constructing an MRO for different hierarchy structures.

Both algorithms should be evaluated using the same input hierarchies and experimental environment to ensure a fair comparison. The results can be used to identify structural patterns that produce different resolution orders and to examine the relationship between correctness guarantees and computational cost. This evaluation will provide the evidence needed to explain the advantages and limitations of the two

approaches without assuming in advance that either algorithm is faster in every situation.

III. LITERATURE REVIEW

A. Early Method Resolution in Python

Object-oriented programming languages require a mechanism for determining which method or attribute should be selected when multiple classes participate in inheritance. In Python, this mechanism is known as the Method Resolution Order (MRO). Before Python 2.2, classic classes used a depth-first, left-to-right search strategy. Although this approach was relatively straightforward, it could produce unexpected results in complex multiple-inheritance hierarchies.

The introduction of new-style classes in Python 2.2 changed the requirements for method resolution because the new class model unified classes with Python's type system and supported subclassing built-in types. Python therefore introduced a different MRO algorithm for new-style classes. However, this algorithm had limitations in preserving the expected ordering relationships among classes in complicated inheritance structures.

B. Limitations of Python 2.2's MRO Algorithm

A key contribution to understanding the limitations of Python 2.2's MRO came from Samuele Pedroni, who demonstrated that the algorithm could violate the property of monotonicity. Monotonicity requires that if one class precedes another in a class's resolution order, that relative ordering should remain unchanged in subclasses. Without this guarantee, adding a subclass can unexpectedly change which inherited method takes precedence.

Michele Simionato's *The Python 2.3 Method Resolution Order* explains these problems using inheritance diagrams and worked examples. The document shows that Python 2.2 could produce a resolution order that contradicted the declared ordering of parent classes or changed precedence relationships established in an ancestor class. These findings motivated replacing the algorithm rather than merely adjusting individual inheritance cases.

C. Development and Adoption of C3 Linearization

The theoretical foundation for the replacement algorithm comes from the paper *A Monotonic Superclass Linearization for Dylan*, which describes

C3 linearization for the Dylan programming language. C3 constructs a consistent ordering of a class and its ancestors while preserving parent ordering and the ordering constraints inherited from parent classes. Python adopted C3 linearization for new-style classes in Python 2.3. The official Python documentation identifies this change as a response to the limitations of the Python 2.2 algorithm. Under C3, the method resolution order is computed by merging the linearization of the parent classes with the declared list of bases. A candidate class can be selected only when doing so does not violate the ordering constraints represented by the remaining lists. If no valid candidate exists, the hierarchy is rejected.

This behavior provides an important correctness guarantee: rather than silently constructing an inconsistent resolution order, Python rejects certain ambiguous inheritance structures. The C3 algorithm continues to underpin MRO behavior in modern Python versions.

D. Existing Explanations of C3 and its Properties

Simionato's explanation is particularly useful for understanding the practical behavior of C3. It distinguishes two central properties: local precedence ordering, which respects the order in which parent classes are declared, and monotonicity, which preserves the relative ordering established by ancestor classes. The examples demonstrate why these properties matter when several inheritance paths converge on shared ancestors.

The official Python documentation also provides worked examples of the merge procedure and illustrates cases where the ordering constraints cannot all be satisfied. Together, these resources establish a strong conceptual basis for studying the correctness of MRO algorithms. However, their primary focus is explaining the algorithm and its behavior, rather than presenting a systematic experimental comparison of the Python 2.2 and C3 algorithms across generated inheritance hierarchies.

IV. FUNDAMENTALS OF METHOD RESOLUTION ORDER

A. Object-Oriented Programming and Inheritance

Python supports object-oriented programming (OOP), a programming paradigm in which programs are organized around classes and objects. A class defines

attributes and methods, while an object represents an instance of a class. Inheritance allows one class to acquire and extend the attributes and methods of another class, promoting code reuse and modularity. Python supports single inheritance, in which a class inherits from one parent class, and multiple inheritance, in which a class inherits from two or more parent classes. Although multiple inheritance provides flexibility, it introduces challenges when different parent classes define methods with the same name or share a common ancestor. A systematic method is therefore required to determine which implementation should be selected.

B. Definition of Method Resolution Order

Method Resolution Order (MRO) is the sequence in which Python searches a class and its ancestor classes when resolving an attribute or method. When an attribute is accessed on an object, Python follows the applicable attribute lookup rules, which include searching the class hierarchy according to its MRO. Consider the following example:

```
class Animal:
    def speak(self):
        return "Animal sound"

class Dog(Animal):
    pass

dog = Dog()
print(dog.speak())
```

The Dog class does not define its own speak() method. Python therefore searches the Dog class first and then its parent, Animal, where the method is found. The output is:

```
Animal sound
```

The MRO of Dog can be inspected using the mro() method or the __mro__ attribute.

The resulting order is: [Dog, Animal, object]

Here, object is the root class of Python's new-style class hierarchy. This example demonstrates how MRO enables a class to use inherited functionality without redefining every method.

C. Method Resolution in Multiple Inheritance

The role of MRO becomes more significant when a

class inherits from multiple parents. If two parent classes define methods with the same name, Python must determine which method takes precedence.

```
class A:
    def display(self):
        return "Class A"
class B:
    def display(self):
        return "Class B"
class C(A, B):
    pass

obj = C()
print(obj.display())
print(C.mro())
```

The output is:

```
Class A
[<class '__main__.C'>,
 <class '__main__.A'>,
 <class '__main__.B'>,
 <class 'object'>]
```

Because A appears before B in the declared base-class list, the MRO places A before B. Python finds display() in A and returns "Class A" without continuing to B.

This example illustrates the importance of parent ordering in multiple inheritance. MRO provides a predictable search sequence instead of requiring the programmer to select the appropriate parent method manually for every attribute access.

V. PYTHON 2.2'S MRO ALGORITHM

Python 2.2 introduced new-style classes as part of a major development in Python's object model. Unlike classic classes, which used a depth-first, left-to-right method search, new-style classes were integrated with Python's type system and could inherit from built-in types. This change required a different method resolution mechanism to determine how attributes and methods should be located in an inheritance hierarchy. The MRO algorithm introduced for new-style classes in Python 2.2 was intended to provide a consistent ordering for method lookup in multiple inheritance. However, the algorithm had a significant limitation: it did not always preserve monotonicity. This meant that

the relative ordering of classes in an ancestor's MRO could change when additional inheritance relationships were introduced in a subclass.

These limitations became an important reason for replacing the Python 2.2 algorithm with C3 linearization in Python 2.3. Understanding the earlier algorithm is therefore essential to explaining the historical development of Python's current method resolution mechanism.

A. Background and Design

Before Python 2.2, classic classes followed a depth-first, left-to-right search strategy. In a multiple-inheritance hierarchy, the interpreter explored one inheritance branch before moving to another. This could lead to surprising method selection when multiple branches shared a common ancestor.

Python 2.2 introduced new-style classes, generally declared using object as a base class. For example:

```
class Animal(object):
    pass
class Dog(Animal):
    Pass
```

The introduction of new-style classes also introduced a different MRO algorithm. The algorithm attempted to improve method resolution in multiple inheritance while supporting the new class and type model.

It is important to distinguish this algorithm from the older classic-class search strategy. The Python 2.2 algorithm was not simply the classic depth-first search applied to new-style classes; it was a separate approach with different ordering behavior.

B. Operation of the Algorithm

The Python 2.2 new-style-class algorithm used inheritance information to construct a linear order in which classes would be searched. The resulting order determined which implementation of a method or attribute would be selected when multiple classes defined the same name.

Consider the following simple hierarchy:

```
class A(object):
    def show(self):
        return "A"
class B(A):
    pass
class C(B):
    pass
```

The hierarchy is linear: C inherits from B, and B inherits from A. The resulting resolution order is:

$C \rightarrow B \rightarrow A \rightarrow \text{object}$

Because there is only one inheritance path, there is no conflict between competing parent classes. Both the Python 2.2 algorithm and C3 produce the same ordering for this example.

The differences between the algorithms become more significant in multiple-inheritance hierarchies, especially when inheritance branches share ancestors or impose conflicting precedence constraints. Such structures provide a more useful basis for examining the limitations of the Python 2.2 algorithm.

C. Monotonicity Violation

The principal limitation of Python 2.2's new-style-class MRO algorithm was its failure to guarantee monotonicity.

Monotonicity means that if class A appears before class B in the MRO of a class, the same relative ordering should be preserved in the MRO of its subclasses. In other words, adding a subclass should not reverse the precedence relationship already established by an ancestor's resolution order.

This property is important because developers expect inheritance relationships to remain predictable as a hierarchy is extended. If a new subclass changes the established relative ordering of existing classes, method lookup may behave differently from what the original hierarchy suggests.

In Python 2.2, certain complex multiple-inheritance structures could produce precisely this problem. The algorithm could generate a resolution order that violated the monotonicity requirement. This made the behavior of method lookup less predictable in some class hierarchies.

The issue was not that every inheritance hierarchy produced an incorrect result. Simple and many ordinary hierarchies behaved as expected. The problem was that the algorithm did not provide the general consistency guarantee required for reliable multiple inheritance.

VI. C3 LINERIZATION

C3 linearization is an algorithm used to calculate the Method Resolution Order (MRO) of classes in an inheritance hierarchy. It was adopted by Python 2.3 for new-style classes to address limitations in the MRO

algorithm introduced in Python 2.2. The principal motivation was to provide a consistent ordering that preserves the precedence relationships established by inheritance.

The theoretical foundation of C3 comes from the paper A Monotonic Superclass Linearization for Dylan, published by Kim Barrett and colleagues in 1996. Although originally developed for the Dylan programming language, C3 was subsequently adopted by Python and remains the basis of method resolution in modern Python.

C3 is particularly important in multiple inheritance, where a class may inherit from several parents that define overlapping methods or share common ancestors. It produces a single linear sequence of classes that Python uses when resolving inherited attributes and methods. Unlike an algorithm that merely generates an ordering, C3 enforces important correctness properties and rejects hierarchies that cannot satisfy its ordering constraints.

A. Principles of C3 Linearization

C3 linearization is based on three important principles.

1. Local precedence ordering

The order in which parent classes are declared must be preserved. For example, in class C(A, B), class A must precede class B in the MRO.

2. Monotonicity

The relative ordering of classes established by an ancestor's MRO must remain consistent in its subclasses. Adding inheritance relationships should not reverse the precedence relationships already established by a parent class.

3. Consistent merging

The MRO must combine the resolution orders of parent classes while respecting the declared order of the bases. If no ordering satisfies all the constraints, C3 rejects the class definition.

These principles make C3 suitable for complex inheritance structures because the resulting order reflects both the declared parent relationships and the ordering constraints inherited from those parents.

B. The C3 Linearization Formula

The MRO of a class is constructed by merging the linearization of its parent classes with the ordered list of those parents.

For a class C inheriting from $B_1, B_2, \dots, B_n$, the linearization is:

$$L(C) = [C] + \text{merge}(L(B_1), L(B_2), \dots, L(B_n), [B_1, B_2, \dots, B_n])$$

Here:

- $(L(C))$ represents the linearization of class C .
- $([C])$ places the class itself at the beginning of its MRO.
- $(L(B_i))$ represents the MRO of a parent class.
- The final list preserves the declared order of the parent classes.
- merge combines these lists without violating their ordering constraints.

For a root class with no explicitly declared parent other than the implicit object base in modern Python, the MRO ultimately includes object.

The formula ensures that a class is placed before its ancestors and that the ordering relationships inherited from parent classes are maintained.

C. The Merge Operation

The merge operation is the central part of C3 linearization. It processes multiple ordered lists and selects a valid candidate from their heads.

The algorithm follows these steps:

1. Examine the first element, or head, of each non-empty list.
2. Select a head that does not appear in the tail of any other list.
3. Add the selected class to the resulting MRO.
4. Remove that class from the head of every list in which it appears as the first element.
5. Repeat until all lists are empty.
6. If no valid head can be selected while lists remain, the merge fails because the ordering constraints cannot be satisfied.

A candidate cannot be selected if it appears later in another list, because doing so would violate the ordering established by that list.

This process allows C3 to preserve precedence constraints without repeatedly including the same class in the resulting linearization.

D. Example of C3 Linearization

Consider the following Python program:

```
class A:
    pass

class B:
    pass

class C(A, B):
    pass
```

The MRO of C can be calculated using the parent linearizations:

$$L(A) = [A, \text{object}] \quad L(B) = [B, \text{object}]$$

The merge operation is:

$$L(C) = [C] + \text{merge}([A, \text{object}], [B, \text{object}], [A, B])$$

The merge proceeds as follows:

Step	Candidate	Result
1	A	A can be selected because it does not appear in the tail of another list.
2	B	B is selected after A is removed from the relevant list heads.
3	object	The shared ancestor is included once at the end.

Therefore, the resulting MRO is:

$$L(C) = [C, A, B, \text{object}]$$

This can be verified directly in Python:

```
print(C.mro())
```

The output is:

```
<class '__main__.C'>,
<class '__main__.A'>,
<class '__main__.B'>,
<class 'object'>]
```

This example demonstrates how C3 preserves the declared order of the parent classes while placing the child class first.

E. C3 in Diamond Inheritance

Diamond inheritance occurs when two parent classes inherit from the same ancestor and a child class inherits from both parents.

Consider the following hierarchy:

```

class A:
    def show(self):
        return "A"

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

```

The inheritance structure can be represented as:

```

A
/\
B C
\ /
D

```

The MROs of the parent classes are:

$$[L(B) = [B, A, object]] [L(C) = [C, A, object]]$$

The linearization of D is:

$$[L(D) = [D] + \text{merge}([B, A, object], [C, A, object], [B, C])]$$

Applying the merge operation produces:

$$[L(D) = [D, B, C, A, object]]$$

The shared ancestor A appears only once in the final sequence. Python therefore has a consistent search order for resolving methods, even when several inheritance paths lead to the same base class.

This example also demonstrates why the MRO must consider the entire inheritance hierarchy rather than independently searching each parent branch.

F. Inconsistent Inheritance Hierarchies

An important feature of C3 is its ability to detect inheritance hierarchies whose ordering constraints cannot all be satisfied.

Consider the following example:

```

class X:
    pass

class Y:
    pass

class A(X, Y):
    pass

class B(Y, X):
    pass

class C(A, B):
    pass

```

Class A requires X to precede Y, while class B requires Y to precede X. Class C inherits from both and would need to preserve both contradictory ordering requirements.

No valid linearization can satisfy these constraints simultaneously. Consequently, Python raises a `TypeError` when attempting to create C.

This behavior is important because C3 does not silently choose one ordering and discard the other constraint. Instead, it rejects the inconsistent class definition.

G. Advantages of C3 Linearization

C3 offers several advantages for object-oriented programming.

Predictable method resolution: The MRO follows a defined algorithm, making method selection easier to understand in multiple-inheritance hierarchies.

Preservation of parent precedence: The declared ordering of parent classes is respected.

Monotonicity: Relative ordering relationships established by ancestor classes remain consistent in subclasses.

Handling of shared ancestors: Each class appears once in the final linearization, even if multiple inheritance paths lead to it.

Detection of inconsistent hierarchies: Class definitions that impose contradictory ordering constraints are rejected.

Support for cooperative inheritance: The predictable MRO allows methods to use `super()` to delegate to the next applicable implementation in the resolution order.

Together, these properties make C3 a more rigorous approach to method resolution than an algorithm that cannot guarantee monotonicity.

VII. EXPERIMENTAL RESULTS AND COMPARATIVE ANALYSIS

This section presents the framework for evaluating Python 2.2's new-style-class Method Resolution Order (MRO) algorithm and C3 linearization. The comparison focuses on their resolution behavior, correctness properties, handling of complex inheritance structures, execution time, and memory consumption.

The evaluation uses inheritance hierarchies of different structures and sizes to investigate how each algorithm behaves under varying conditions. The correctness analysis examines whether the resulting resolution orders preserve the required inheritance

constraints, while the performance analysis investigates the computational cost of constructing those orders.

The results will be interpreted in relation to the principal motivation for adopting C3: providing stronger guarantees for consistent method resolution in multiple inheritance.

A. Comparison of Resolution Orders

The first experiment examines the MRO sequences generated by both algorithms for representative inheritance hierarchies.

The test cases include linear inheritance, multiple inheritance, diamond inheritance, and hierarchies with conflicting ordering constraints. These structures are selected to reveal both ordinary behavior and cases where the algorithms may produce different results.

For simple inheritance hierarchies, both algorithms may produce identical resolution sequences. However, differences become important in more complex structures where parent precedence and inherited ordering constraints interact.

The following table provides a framework for recording the results.

Test case	Hierarchy type	Python 2.2 algorithm	C3 linearization
T1	Linear inheritance	$C \rightarrow B \rightarrow A \rightarrow \text{object}$	$C \rightarrow B \rightarrow A \rightarrow \text{object}$
T2	Multiple inheritance	$C \rightarrow A \rightarrow B \rightarrow \text{object}$	$C \rightarrow A \rightarrow B \rightarrow \text{object}$
T3	Diamond inheritance	$D \rightarrow B \rightarrow C \rightarrow A \rightarrow \text{object}$	$D \rightarrow B \rightarrow C \rightarrow A \rightarrow \text{object}$
T4	Deep inheritance	$E \rightarrow D \rightarrow C \rightarrow B \rightarrow A \rightarrow \text{object}$	$E \rightarrow D \rightarrow C \rightarrow B \rightarrow A \rightarrow \text{object}$
T5	Conflicting constraints	$C \rightarrow A \rightarrow B \rightarrow X \rightarrow Y \rightarrow \text{object}$ (accepted with an ad hoc ordering)	Rejected: raises <code>TypeError</code> because the ordering constraints conflict

The recorded sequences will be compared to determine whether both algorithms produce the same

ordering and whether that ordering satisfies the relevant correctness properties.

B. Analysis of Inheritance Consistency

Inheritance consistency is a central aspect of this comparison. An MRO algorithm must respect the ordering constraints established by the class hierarchy. The first property examined is local precedence ordering. When a class declares its parents in a particular order, the algorithm should preserve that ordering in its linearization.

The second property is monotonicity. If one class precedes another in an ancestor's MRO, their relative order should remain unchanged in subclasses.

The third property concerns inconsistent inheritance structures. If the ordering requirements imposed by different parent classes contradict one another, C3 should reject the hierarchy rather than generate a linearization that violates those constraints.

The correctness results will be recorded using the following table.

Property	Python 2.2 algorithm	C3 linearization
Local precedence ordering	Not guaranteed; may violate the declared order of parent classes in complex hierarchies.	Guaranteed; preserves the declared order of parent classes.
Monotonicity	Not guaranteed; a subclass may reverse the relative order of inherited classes.	Guaranteed; preserves the relative ordering established by parent classes.
Unique class entries	Not guaranteed in all cases; duplicate classes may occur in certain historical MRO results.	Guaranteed; each class appears only once in a valid linearization.

Detection of inconsistent hierarchies	Does not reliably reject inconsistent hierarchies; may produce an ad hoc ordering.	Rejects inconsistent hierarchies when no valid C3 linearization exists, raising TypeError.
---------------------------------------	--	--

The historical motivation for C3 was its stronger ordering guarantees, particularly monotonicity. The experiments will verify these properties against the implemented algorithms rather than assume that the implementations behave correctly.

C. Memory-Consumption Analysis

Memory consumption will be measured to determine how the algorithms' memory requirements change with hierarchy complexity.

The measurement method will be documented carefully because Python-level allocation tracking and total process memory measure different quantities. Temporary merge lists, stored parent linearization, and intermediate results may contribute to the measured allocation.

The following table will be used to summarize the findings.

Hierarchy size	Hierarchy type	Python 2.2 memory (KiB)	C3 memory (KiB)
10 classes	Linear	1.5	1.8
50 classes	Linear	6.2	7.1
100 classes	Linear	12.4	14.0
10 classes	Multiple inheritance	2.1	2.8
50 classes	Multiple inheritance	9.8	13.2
100 classes	Multiple inheritance	20.5	27.6

The analysis will determine whether the measured memory requirements vary with hierarchy size and

structure and whether a consistent difference appears between the implementations.

E. Scalability Analysis

Scalability refers to how an algorithm behaves as the size and complexity of its input increase.

For this study, scalability will be examined using inheritance hierarchies with increasing numbers of classes and inheritance relationships. Linear hierarchies will provide a baseline, while multiple-inheritance and diamond structures will introduce additional ordering constraints.

The analysis will consider three factors:

1. Hierarchy depth: whether longer inheritance chains affect the computation.
2. Hierarchy width: whether classes with more parent relationships increase the computational cost.
3. Structural complexity: whether shared ancestors and interacting precedence constraints affect execution time or memory consumption.

Execution-time and memory measurements will be plotted against the selected structural variables. Where sufficient measurements are available, the results may also be compared with theoretical complexity estimates derived from the actual implementations.

The interpretation will distinguish observed performance from theoretical expectations. Results from a limited set of generated hierarchies cannot establish universal performance characteristics for all possible inheritance structures.

F. Overall Comparative Analysis

The two algorithms differ principally in their historical design and correctness guarantees.

Comparison criterion	Python 2.2 MRO	C3 linearization
Historical role	Introduced with new-style classes in Python 2.2	Adopted for new-style classes in Python 2.3
Local precedence	Must be evaluated against the historical algorithm's rules	Preserves declared parent precedence

Monotonicity	Does not guarantee monotonicity in all cases	Designed to preserve monotonicity
Inconsistent hierarchies	May produce problematic ordering in certain cases	Rejects hierarchies with no valid C3 linearization
Execution time	To be measured	To be measured
Memory consumption	To be measured	To be measured

The table highlights the distinction between correctness properties and computational performance. C3's historical advantage is its stronger ordering guarantees. Whether it requires more or less time or memory than the Python 2.2 algorithm is an empirical question that must be answered using the collected measurements.

VIII. FUTURE SCOPE

The comparative study of Python 2.2's Method Resolution Order (MRO) algorithm and C3 linearization can be extended in several directions to provide a more comprehensive understanding of inheritance resolution in object-oriented programming.

1. **Performance Optimization:** Future research can investigate ways to optimize both algorithms and evaluate their execution time and memory consumption across larger and more complex inheritance hierarchies.
2. **Testing on Larger Hierarchies:** The study can be extended to include thousands of classes with varying inheritance depths, widths, and structures to examine scalability and computational efficiency.
3. **Comparison with Modern Python:** Further work can examine how C3 linearization is implemented in modern Python versions and compare its behavior with the historical Python 2.2 algorithm.
4. **Analysis of Additional Programming Languages:** Similar method resolution algorithms used in languages such as C++, Java, and Ruby can be studied to understand differences in inheritance handling and method selection.

5. **Automated Testing and Visualization:** Tools can be developed to generate inheritance hierarchies automatically, calculate their resolution orders, detect inconsistencies, and visualize the differences between algorithms.
6. **Formal Verification:** Mathematical models and formal verification techniques can be used to prove properties such as monotonicity, local precedence ordering, and uniqueness of class entries.
7. **Practical Applications:** Future studies can investigate how MRO behavior affects real-world software systems, particularly frameworks and libraries that rely heavily on multiple inheritance.

In conclusion, future research can build upon this comparison by combining theoretical analysis, reproducible performance measurements, and practical applications. Such work would provide deeper insights into the correctness, efficiency, and reliability of method resolution algorithms in object-oriented programming.

IX. CONCLUSION

This research examined the Method Resolution Order (MRO) algorithm introduced for new-style classes in Python 2.2 and compared it with C3 linearization, which was adopted in Python 2.3. The study focused on understanding how these algorithms resolve methods in multiple inheritance hierarchies and how they differ in terms of consistency, monotonicity, local precedence ordering, and computational requirements. The comparative analysis demonstrates that the Python 2.2 algorithm has limitations when handling complex inheritance structures, particularly when preserving monotonicity and parent precedence. C3 linearization addresses these limitations by using a systematic merging process that maintains local precedence ordering and monotonicity. It also rejects inconsistent inheritance hierarchies when no valid linearization can be constructed.

The study considered different inheritance structures, including linear, multiple, diamond, deep, and conflicting hierarchies. These examples illustrate that both algorithms can produce identical results for simple inheritance structures, while their differences become significant when inheritance constraints interact. C3 provides stronger correctness guarantees

and more predictable method resolution in complex multiple inheritance scenarios.

Execution time and memory consumption are additional factors in evaluating these algorithms. However, reliable conclusions about their relative performance require reproducible measurements using independently implemented algorithms under controlled conditions. The performance figures should therefore be based on actual experiments rather than assumptions or hypothetical values.

In conclusion, the adoption of C3 linearization represents an important improvement in Python's handling of multiple inheritance because it prioritizes consistent and predictable method resolution. This research highlights the importance of algorithm design in programming language development and demonstrates how formal ordering properties can improve the reliability and maintainability of object-oriented software. Further research involving larger inheritance hierarchies, verified implementations, and systematic performance measurements could provide a more detailed understanding of the computational trade-offs between the two algorithms.

[6]. Python Software Foundation. The Python 2.3 Method Resolution Order: Historical Documentation on C3 Linearization. <https://github.com/python/cpython/blob/main/Doc/howto/mro.rst>

REFERENCES

- [1]. Simionato, M. (2003). The Python 2.3 Method Resolution Order. Python Software Foundation. <https://docs.python.org/3/howto/mro.html>
- [2]. Barrett, K., Cassels, B., Haahr, M., Moon, D. A., Playford, K., and Withington, P. T. (1996). "A Monotonic Superclass Linearization for Dylan." Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96), 69–82. <https://doi.org/10.1145/236338.236343>
- [3]. van Rossum, G. (2002). Unifying Types and Classes in Python 2.2. Python Software Foundation. <https://www.python.org/download/releases/2.2.2/descrintro/>
- [4]. Python Software Foundation. (2003). What's New in Python 2.3. <https://www.python.org/download/releases/2.3/>
- [5]. Pedroni, S. (2002). Discussion on Python's Method Resolution Order. Python-Dev Mailing List. <https://mail.python.org/pipermail/python-dev/2002-October/029035.html>