Olaniyi Aiyenitaju

Sarita Majhi

Math with Python: A Context of Algebra

Metropolia University of Applied Sciences

Bachelor of Engineering

Name of the Degree Programme

Bachelor's Thesis

31 June 2024

Abstract

Author:	Olaniyi Aiyenitaju, Sarita Majhi.
Title:	Math with Python: A Context of Algebra
Number of Pages:	38 pages + 3 appendices
Date:	31 June 2024
Degree:	Bachelor of Engineering
Degree Programme:	Information Technology
Professional Major:	IoT and Networks
Supervisors:	, (Project Advisor)

Abstract

Examining the mutually beneficial link between algebra and Python, this study seeks to solve computational difficulties and improve the efficiency of algebraic operations. Navigating Python's symbolic algebraic complexity is the task at hand. These include computation-intensive problems, numerical accuracy problems, and huge expression handling problems.The main conclusions centre on constructive avenues for future research and contributions to the subject. We suggest directions to further improve algebraic symbolic performance, explore quantum algebraic computing, develop specialised Python algebraic libraries, combine machine learning with algebraic tasks, and create interactive educational tools. These discoveries close gaps in the present Python environment and provide creative answers to computational problems. Overall, this work is important since it sets the path for future Python algebraic exploration projects. This study moves algebraic techniques in a programming context forward by addressing the obstacles faced and offering innovative solutions. The possible uses cover a wide range of industries, including cutting-edge areas like quantum computing as well as scientific research and teaching. As background, the study discusses the fundamental value of algebra in mathematics and presents Python as a powerful tool for manipulating algebra. We examine fundamental algebraic operations, equations, functions, and Python algebraic applications. The difficulties and constraints that are faced highlight the necessity of using Python for algebraic work with subtle approaches.

Keywords: Algebra, Python, Python Libraries, SymPy.

Contents

List of Abbreviations

Python: Programming Language

CAS: Computer Algebra System

SymPy: Symbolic Python Library

Numpy: Numerical Python Library

SciPy: Scientific Python Library

IDE: Integrated Development Environment

API: Application Programming Interface

GUI: Graphical User Interface

IRQ: Indexing in the Python Language

ADHD: Attention Deficit Hyperactivity Disorder

1 Introduction

1.1 A Background on the Importance of Algebra in Mathematics

Having origins in ancient civilization, algebra has become a vital and dynamic aspect of mathematical thinking. For mathematical reasoning and language conveyed through symbols, tables, words, and graphs, algebra is an essential subject of study (Edo and Tasik, 2022). Algebra provides a structured method for interpreting mathematical patterns by enabling the methodical investigation of relationships between quantities. Its historical development is a reflection of both the ongoing search for increasingly complex approaches to problem-solving as well as the effort to comprehend basic numerical principles.

The significance of algebra is demonstrated by its capacity to serve as a common language across various mathematical fields (Gronmo, 2018). Fundamental to many STEM (science, technology, engineering, and mathematics) sectors, algebra is a crucial subject in mathematics education that serves as the basis for more advanced mathematical ideas. It is frequently referred to as "the gateway to higher mathematics" (Veith et al., 2023). It goes beyond simple symbol manipulation to serve as a medium for the expression and application of complex mathematical concepts.

Also, algebra acts as a cornerstone for the development of critical thinking abilities and logical reasoning in pupils, assisting them in their academic journey. All vocations and educational levels require proficiency in algebra (Gronmo, 2018). That is to say, a fundamental comprehension of algebra is necessary to succeed in fields other than mathematics. Businesses utilise algebra, for instance, to calculate their annual budget, which includes their annual expenditure. Furthermore, a number of retailers utilise algebra to forecast the level of demand for a specific item before placing their purchases (Edo and Tasik, 2022). Understanding algebraic concepts gives people a toolkit to solve analytical problems in a variety of fields in addition to mathematical difficulties.

1.2 Introduction to Python as a Programming Language

Since its inception in the late 1980s, Python has developed into a highly adaptable and popular programming language. Python is currently widely used by programmers, data scientists, and mathematicians due to its readability and simplicity. It is powerful for complicated computational tasks and easy to use even for beginners as a result of its lean syntax and large library.

Recently, Python has become more and more popular as a high-level general-purpose programming language. Compared to languages like C, programmers may express concepts in fewer lines of code thanks to this language's syntax. The design ethos places a strong emphasis on code readability (Srinath, 2017). The linguistic constructions enable the user to create software that are comprehensible on both small and large sizes (Srinath, 2017). Python is an interpretable, object-oriented, interactive programming language. Dhruv, Patel, and Doshi (2020) claim that it provides high-level data structures like sets, tuples, lists, dynamic typing and binding, associative arrays (sometimes called dictionaries), modules, classes, exceptions, and intelligent control of memory. It is a sophisticated programming language with a very simple and easy syntax that is also utilised for parallel computing systems.

Python stands out from other programming languages because it is a high-level language suitable for a variety of tasks, including web development and scientific computing. Due to its versatility, Python is a great tool for exploring mathematical ideas since it offers an easy-to-use interface for converting abstract mathematical ideas into executable code. In this context, Python plays a function that goes beyond that of a simple programming language and instead serves as a medium for translating algebraic theories into concrete solutions. Python acts as the link between algebraic operations, equations, and functions, allowing us to move from abstract mathematical thinking to real-world application.

2 Overview of Algebraic Concepts

2.1 Basic Algebraic Operations

Algebra has four basic operations, each with special properties and transforming powers of its own.

The first of these operations, addition, goes beyond simple numeric combination. It serves as a hub, uniting quantities to form combined expressions. Its importance extends beyond the domain of integers to rational numbers, algebraic variables, and complex numbers, where it establishes the groundwork for a variety of algebraic structures.

Though it is frequently thought of as addition's opposite, subtraction has many subtle uses. It breaks down mathematical equations, revealing relationships between quantities and drawing distinctions. When applied to algebraic terms, the abstraction of subtraction comes to life and allows complex statements to be disentangled.

The terrain broadens to encompass the complexities of allocating amounts, mixing variables, and creating proportionate connections when we go to multiplication. Multiplication introduces variables into algebraic expressions, allowing them to grow and develop into polynomials and other more complex mathematical structures.

Finally, division is a subtle operation that helps to untangle and make sense of the relative magnitudes of quantities. It opens the door to rational expressions, ratios, and the investigation of mathematical structures beyond basic equations when applied to algebraic terms.

2.2 Equations and Inequalities

Fundamental algebraic concepts, equations and inequalities, are expressive tools that help explain the relationships and limitations between quantities. Fundamentally, an equation states that two expressions are equal and usually

includes variables in addition to constants. Finding the values for the variables that make an equation true is the first step in solving an equation. For example, the goal is to find the value of x so that the linear equation 2x+5=13 is satisfied. Values that fulfil quadratic equations, like x2 −4x+4=0, must be found since they introduce the square of a variable.

Conversely, inequality represents ranges of values by illustrating relationships in which one side is bigger than, less than, or equal to the other. If 3y−7<5, for example, a linear inequality, it means that 3y−7 is less than 5. Finding the acceptable range of values for y is the first step in solving this inequality. x2 + 2x−8>0 is an example of a quadratic inequality. It introduces quadratic expressions, which are solved to identify intervals in which the expression is higher than zero.

According to Otten's research (2019), understanding linear equations just requires a few simple steps. In order to solve linear equations, one must first understand the concept of equality in mathematical expressions. The premise of an equation, according to Bush and Karp (2013) and Alibali (1999), is that two provided mathematical expressions on each side of the equal sign reflect the same value. One of the most important ideas in solving linear equations is equality. According to Kieran (1997) and Kieran et al. (2016), knowing equality is a fundamental conceptual requirement in equation solving.

Despite its simplicity, the idea of equality in linear equation solution is the subject of several well-documented myths. Students frequently have misconceptions, particularly when it comes to the equal sign. Instead of interpreting the equal sign as a relational symbol meaning "is the same as," students frequently interpret it from an operational standpoint meaning that it tells them "to execute a certain thing" or "determine the outcome" (Knuth et al., 2006). Adding the numbers on the left, for instance, and typing 12 into the blank while solving the equation 8 + 4 = __ + 5 is a typical error (Falkner, Levi, & Carpenter, 1999). According to Alibali et al. (2007), this operational interpretation typically appears early and lasts through the primary and middle school years.

It is necessary to understand that inequality describes connections in which one side is greater than, less than, or equal to the other when expanding the discussion to include inequalities, a similar but separate mathematical notion. Understanding equality in linear equations is a fundamental step towards solving inequalities, where the emphasis is on conveying ranges of values instead of absolute equivalency. Comprehending these ideas in depth is essential for both problem-solving in mathematics and building a strong foundation in algebraic reasoning.

A critical step in solving quadratic equations is determining the x-intercepts, which often requires factorization, the square root method, completing the square, and the quadratic formula (Harripersaud, 2021). Regarding their effectiveness in teaching, learning, and application, each of these approaches has unique benefits and drawbacks. The insights provided by Harripersaud emphasise how important it is to comprehend and use these strategies in order to achieve thorough competency in solving quadratic problems.

Studies, as summarised by Harripersaud (2021), reveal that factorization is a common predisposition for teachers and students, with a preference for readily factorizable coefficients. This tendency is explained by students' alleged difficulties with fractional and radical mathematics (Bosse & Nandakumar, 2005). Given that many quadratic equations resist easy factorization, Harripersaud's emphasis cautions against placing too much importance on factorization. This prompts questions regarding the possible disregard of alternative approaches that promote conceptualization and may be more effective (Bosse & Nandakumar, 2005).

Harripersaud (2021) also emphasises the pedagogical benefit of other approaches, including factorization with algebra tiles, which uses the area model of rectangles and squares to create a connection between quadratics and fundamental notions of multiplication and division (Howden, 2001). This method offers a deeper knowledge of quadratic equations by improving mental comprehension in addition to providing a visual depiction.

Furthermore, while solving the square, Harripersaud highlights the importance of geometric models as tools for comprehending, implementing, and using the quadratic formula (Norton, 2015). This method is consistent with Barnes's (1991) recommendation to use graphing calculators to plot quadratics, investigate cases where there are zero, one, or two roots, and relate these results to discriminant values. These practical methods and illustrations help students grasp quadratic equations more deeply and develop a comprehensive understanding that extends beyond memorising formulas.

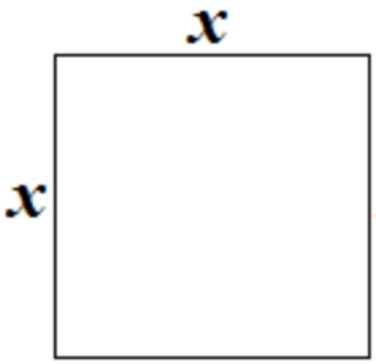

Figure 1. A Perfect Square. Source: Harripersaud, (2021)

A visual study of the idea of a perfect square is shown in Figure 1. The equation can be understood geometrically by multiplying the side length by itself to calculate the area of the square.

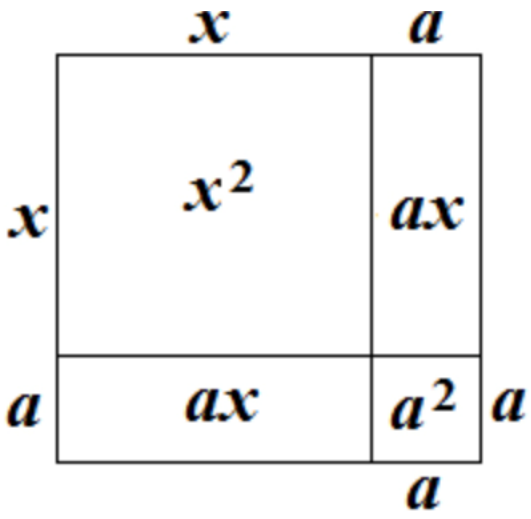

Figure 2. Geometric and Algebraic Perspectives. Source: Harripersaud, (2021)

The algebraic and geometric viewpoints are shown in Figure 2. In terms of geometry, the total size of the distinct areas represents the area of the larger square. We can represent the same result algebraically in terms of algebraic expressions. This condition, known as a perfect square, shows how the constant term and the product of the sides and the coefficient of x are related to each other.

If a rectangle has unequal sides, we call them 'a' and 'b,' where 'a' stands for length and 'b' for breadth. In this case, the area of the rectangle is obtained by multiplying the length by the width. It's interesting to observe that the constant term in the algebraic expression represents the product of 'a' and 'b,' whereas the coefficient of x represents their sum. The foundation for factoring quadratic expressions with a coefficient of x2 equal to 1 is established by these facts.

By extending this concept to the product of two binomial expressions in which the coefficient of x^2 is not 1, we examine expressions such as (x + a)(x + b). In this instance, acbd is produced by multiplying the x^2 coefficient by the constant term. This indicates that two specific factors whose sum yields ad + bc can be found in order to factor quadratic expressions that do not have an x^2 coefficient of 1.

Finally, factorization, completing the square, and applying the quadratic formula are the three algebraic strategies for resolving quadratic problems. 'A' is the equation's coefficient of x^2, 'B' is the coefficient of x, 'c' is the constant that constitutes the term, and 'x' is the variable that's independent in the quadratic equation ax2+bx+c=0. Every approach provides unique benefits and perspectives on handling quadratic problems, which add to a thorough comprehension of these basic mathematical expressions

2.3 Functions and their Representations

Functions provide a strong foundation for modelling a variety of interactions between variables by acting as the pivot that creates a systematic relationship between input and output values (Matik, Poljak, and Rukavina, 2022). The purpose of this part is to dissect functions, highlighting their importance and

exploring the different ways in which they can be expressed, in order to demonstrate their indispensable role in mathematical analysis.

A function is essentially a mathematical rule that maps every element in a domain (a set of elements) to exactly one element in a codomain (a different set of elements) (Matik, Poljak and Rukavina, 2022). Fundamental to the quantification and encapsulation of a wide range of real-world occurrences, from basic linear relationships to the complexities of nonlinear behaviours, is this one-to-one correspondence. Functions are essential for distilling the essence of mathematical systems and offering a methodical way to think about their dynamics.

Functions are represented in a variety of ways, each providing a unique perspective on the properties and behaviour of the function. These forms include algebraic expressions, graphical representation, tabular representation, verbal representation, and functional notation, according to Libretexts Mathematics, (2023).

Algebraic Representation: Algebraic expressions are an effective way to express functions because they capture the principle that determines how input and output are related. An example of a linear function is f(x)=2x+3, where the output f(x) is defined by the input (x) using an algebraic rule.

Graphical Representation: By matching input values to corresponding output values, graphs help to visually communicate the essence of functions. This depiction, which uses a straight line graph of a linear function as an example, clearly shows trends, patterns, and important characteristics of a function.

Tabular Representation: Tables provide an organised framework for discrete functions by concisely presenting input-output pairs. When a comprehensive, tabular display of values is desired, this approach is especially helpful.

Verbal Representation: Using words to express the link between input and output is necessary when describing functions verbally. The behaviour and goal of a

function are effectively and understandably conveyed by this narrative representation.

Functional Notation: When functions are expressed using notation like y=f(x), it clearly indicates how dependent the output (y) is on the input (x). This notation highlights the two variables' inherent relationship.

Not only is an understanding of functions and their many representations essential for using algebra, but it also establishes the foundation for more extensive applications in mathematics and other scientific fields. As we move forward, investigating Python's function handling skills will enhance our capacity to examine, simulate, and resolve complex algebraic issues.

3 Python as a Tool for Algebraic Manipulation

3.1 Introduction to Python Syntax and Data Types

The readability and clarity of Python's syntax make it an ideal language for expressing mathematical operations and algorithms. In order to promote code readability and minimise the need for superfluous punctuation, the syntax uses indentation and whitespace to separate code blocks.

For the purpose of manipulating algebra, one must understand Python's data types. The built-in data types in Python are diverse and have different functions. The fundamental numeric types, which comprise integers (int) and floating-point numbers (float), provide a solid foundation for algebraic operations. Python's support for complex numbers also makes it possible to manipulate mathematical entities outside of the real number system.

A string is anything that is typed and is surrounded by "single" or "double" quote marks. It can be a letter, number, symbol, or space (Church et al., 2021). In Python, strings (str) are essential for representing textual data. When working with algebraic expressions, this data type becomes important since it permits the addition of textual annotations and symbols. When a researcher records a subject's answer to the open-ended question, "How has your depression been affecting your life?" (Church et al., 2021), this is an example of how strings might be employed. Since strings cannot have their contents changed once they are generated, the only thing you can do is write over them (redefine the variable).

Figure 3. Example of a String. Source: Church et al., (2021).

As represented in the image above, the textual response / information given by the subject is represented in quotation marks as a string, to show that the response provided falls into text category.

Church et al. (2021) state that lists in Python are ordered sequences enclosed in square brackets ([]), allowing for the storage of any kind of object, even a combination of several types, inside the same list. An example that serves as an illustration is the compilation of a list in which the IQ scores of a group of individuals are stored. Each participant's score is arranged according to their identification number (Church et al., 2021). Lists' ordered structure makes indexing easier and enables users to obtain data from particular locations in the list. Python indexing starts at 0, therefore the index of the first item in a list is 0, the index of the second is 1, and so on. For example, indexing can be used to retrieve the first object in the list if a researcher wants to access the IQ score of the first participant. They can do this by using square brackets after the variable name, such as IQ_scores[0] (Church et al., 2021).

Figure 4. Example of a List. Source: Church et al., (2021).

In Figure 4 above, a list of numbers is created, and encapsulated with square brackets in order to store information in the list. If there were two different lists present, the two sets of information would be encapsulated in two different square brackets.

Python goes beyond lists and presents tuples, which are ordered lists of items separated by parenthesis (). Tuples can be overwritten, but they cannot be changed once generated, in contrast to lists (Church et al., 2021). Because of their immutability, tuples use less memory, which is useful in situations when memory efficiency is critical. One good use of tuples is to record each participant's

reaction time as well as their group membership, regardless of whether they are in the control group (Church et al., 2021).

Figure 5. Example of a Tuple. Source: Church et al., (2021).

Unlike lists in which square brackets are used, tuples are represented with parentheses in Python as illustrated in figure 5 above.

According to Vanderplas (2017), integers are entire numbers that are devoid of decimal marks and can be either positive or negative. They are necessary in circumstances where it is important to count discrete numbers. For instance, integers could be used in neuroscience research to convey data about the number of activated neurons at a given time in a clear and succinct manner (Church et al., 2021).

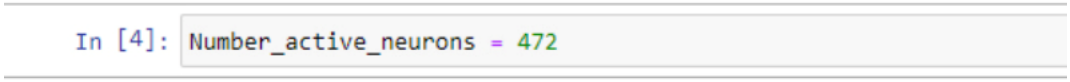

Figure 6. Example of an Integer. Source: Church et al., (2021).

The information above shows integer “472”, a positive whole number without decimal point. The information is clear and concise and can be easily identified by a user as an integer based on the manner in which it is represented.

Floats are real numbers with decimal points that can be used in instances where accuracy in measurements is required. They offer a higher level of precision. Church (2021) highlights how useful they are for documenting continuous variables, like a participant's millisecond reaction time throughout a task. Church noted that the difference between 32- and 64-bit floats is in the accuracy of values

after the decimal point; float64 doubles the accuracy of float32 but uses more bits.

```
In [4]: Reaction_time = 1.25674
```

Figure 7. Example of a Float. Source: Church et al., (2021).

Float is the opposite of integer in that they include decimal point as Figure 7 above reveals.

Logical expressions, or Booleans, are binary choices that are assessed as True or False (1 or 0). Church (2021) gives an example in the context of survey replies, where respondents may mark as positive or negative their experiences with depression or anxiety. In computer logic, booleans are essential for expressing and assessing circumstances.

```
In [1]: experience_anxiety = True
        experience_depression = False
```

Figure 8. Example of a Boolean. Source: Church et al., (2021).

Python's built-in type () function is essential for confirming a variable's data type at any time while coding. It guarantees that the anticipated kind of data is being used and aids in averting mistakes or irregularities in data processing.

```
In [2]: experience_anxiety = True
        experience_depression = False

        type(experience_anxiety)

Out[2]: bool
```

Figure 9. Example of a Type. Source: Church et al., (2021).

Vanderplas (2017) and Church (2021) both emphasise sets, which are unordered groups of singular items encased in curly brackets {}. They offer an adaptable framework for storing various data kinds without creating duplicates. Church (2021) provides an example of this using a collection designed to hold symptoms of conditions such as ADHD. When uniqueness is a crucial requirement and the order of the elements is irrelevant, sets are especially helpful.

Figure 10. Example of a Set. Source: Church et al., (2021).

Figure 10 shows how Type is used in Python. In the figure above, each symptom as represented by the user must be unique and not integrated into another hence, the use of a set.

Additionally, Python presents the idea of variables, enabling users to give values names. This improves readability of the code and complies with algebraic norms, where variables stand for unknowns or variable amounts.

3.2 Performing Operations Using Python

Python supports seven kinds of operators namely Arithmetic, Comparison, Assignment, Logical, Bitwise (see appendix 1), Membership (see appendix 2), and Identity (See appendix 3). In Python, a number involved in an operation is called operand and a command is called operator. For example, in 2 + 3 = 5, 3 and 2 are operands while + is an operator. This section contains a description of four of these operators as implemented in Python.

Arithmetic Operators

Operator	Description	Example
+	Addition: Returns the product of two values.	10 + 20 will give 30
-	Subtraction: Returns the result of the subtraction of the right and left hand operand.	5 – 2 will give 3
*	Multiplication: Returns the result of the multiplication of two or more operands.	5 * 10 will give 50
/	Division: Returns the result of the division of the left and right hand operand. Usually left divided by right.	18 / 2 will give 9
%	Modulus: Carries out division of operands and reports remainder.	18 / 2 will give 0
* *	Exponent: Performs exponential operations.	x ** y = 10^4
/ /	Floor division: In this division, the quotient obtained is the number of digits that remain after the decimal point.	x / / y will give 4 and x.0 / / y.0 will give 4.0

Comparison Operators

Operator	Description	Example
= =	A condition is true when the value of two operands are the same (equal)..	(x == y) is not true
!=	A condition is true if the value of two operands are not the same (equal).	(x != y) is true
<>	A condition is true if the value of two operands are not the same (equal)..	(x <> y) is true
>	A condition is true when the value of the left operand is greater than the right operand.	(x > y) is not true
<	The condition is satisfied if the left operand's value is lower than the right operand's.	(x < y) is true
>=	Conditions become true if the value of the left operand is larger than or equal to the value of the right operand.	(x >=) is not true
<=	The condition is satisfied if the value of the left operand is less than or	(x <= y) is true

	equal to the value of the right operand.	

Assignment Operators

Operation	**Description**	**Example**
=	Values are assigned to the left operand from the right operand.	When z = x + y, the value of x + y is assigned to z.
+=	This operator is referred to as "Add AND." The left operand's value is assigned to the result of adding the right operand to the left operand.	Z = Z + x is the same as Z + = x.
-=	This operator is known as "Subtract AND." It assigns the result to the value of the left operand by subtracting the right operand from the left operand.	Z = z – x is the same as Z -= x.
*=	The term "Multiply AND" operator refers to this. The left operand is assigned the value of the product of multiplying the right operand by the left operand.	Z * = x is the same as Z = z * x.

/ =	This operator is known as "Divide AND." It assigns the value of the left operand to the remainder after dividing the left operand by the right operand.	Z = Z / x is the same as Z / =.
%=	This operator is referred to as "modulus AND." The modulus is calculated using two operands, and the result is assigned to the value of the left operand.	Z = Z % x is the same as Z = Z % x.
**=	The operator for this is termed "Exponent AND." After computing the exponent, the value is assigned to the operand on the left.	Z = z ** x is the same as z ** = x.
/ / =	This operator is referred to as "Floor division AND." It divides the floor and gives the left operand a value.	Z = z / / x is the same as z / / = x.

Logical Operators

Operator	**Description**	**Example**

and	Logical AND operator is the term for this. Both true operands satisfy the condition.	It is true that (x and y).
or	The term for this operator is logical OR. If any of the operands are non-zero, the condition is true.	It is true that (x or y).
not	Logical NOT operator is the term for this. It flips the operand's logical state. This operator sets a condition to false if it is true otherwise.	not(x and y) is false.

4 Applications of Algebra in Python

4.1 Solving Equations and Systems of Equations

One can use Python to solve equations with a single variable. For symbolic algebra, the SymPy library is especially helpful: from SymPy import symbols, Eq, solution

# Define the variable x = symbols('x')
# State that the equation is equal to Eq(2*x + 5, 13).
Solution = solve(equation, x) print(f"Solutions for x: {solution}") # Solve the equation.

SymPy also provides a handy method for systems of equations. Here's an illustration using a pair of equations:

# Set variables x and y to equal symbols('x y').
# Explain the equation system.
equation1 = Eq(3*y, 12) + 2*x
equation2 = Eq(2*y, - 4*x)
Solution = solve((equation1, equation2), (x, y)) print(f"Solutions for x and y: {solution}") # Solve the system of equations

Tools such as fsolve are available in the SciPy library for numerical solutions. When symbolic solutions are not practical, this is advantageous:

from SciPy.maximise import fsolve

Define a function def equation_to_solve(x) that represents the equation:

return 4*x - 5 #x**2 To discover a numerical solution, use fsolve.

Equation to Solve = fsolve(numeric_solution, 0)

print(f)"Numerical solution for x: {numerical_solution}")

In many cases, challenges in science and engineering include algebraic equations. These equations can be solved by Python, which makes it useful in physics, chemistry, and many engineering fields. For example, optimising engineering designs or figuring out unknown values in physical tests.

Statistics and data analysis both use algebraic equations. Equations from statistical modelling, curve fitting, and regression analysis can be solved with Python and libraries like NumPy and SciPy.

By creating equations to express the goal function and constraints, Python may tackle optimisation problems. Tools for resolving these issues are provided by libraries like SciPy optimise.

SymPy's symbolic computation capabilities are improved by its physics module, specifically the SymPy Physics Vector package. According to Meurer et al. (2016), reference frame-aware vector and dyadic objects can be used to carry out three-dimensional operations such addition, subtraction, scalar multiplication, inner and outer products, and cross products. With the help of these objects, vectors and dyadics can be expressed in concise notation in numerous reference frames with arbitrary relative orientations.

In physics, locations, velocities, accelerations, orientations, angular velocities, angular accelerations, forces, and torques are all specified in large part by vectors and dyadics. As reference frame-aware 3x3 tensors, these objects provide a strong foundation for vector algebra in one, two, or three dimensions, which can be used to construct engineering and physics applications.

The SymPy Physics Vector module, for example, allows vectors to be created and altered across several reference frames. The orthogonal unit vectors of three reference frames (A, B, and C) oriented in various orientations relative to each other can be used to produce a vector, as demonstrated by the following snippet of Python code:

import from sympy.physics.vector ReferenceFrame; import pi, sqrt from sympy

'A' as ReferenceFrame equals A.

ReferenceFrame('B') = B. ReferenceFrame('C') = C.b.orient(A, 'body', (pi, pi/3, pi/4), 'zxz') C.orient(B, 'axis', (pi/2, B.x))

(A * x + B * z + C * y) = v

This code illustrates how to rotate the C frame around the B frame's X unit vector and use Z-X-Z to orient the B frame in order to express a vector in the A frame.

4.2 Python Libraries Useful in Algebra

Python is a particularly strong and flexible programming language that provides a wide range of specialised libraries to solve challenging mathematical problems. Many Python libraries that are helpful in addressing algebra problems are examined in this section.

NumPy

The foundation for manipulating arrays and performing mathematical operations in Python is NumPy, a basic toolkit for numerical computations. Mahalaxmi et al. (2023) highlight how important it is to support N-dimensional arrays and how essential it is to the SciPy library's operation. When it comes to performing numerical operations and transformations, NumPy helps to overcome the shortcomings of Python's built-in data structures. Mahalaxmi et al. (2023) list the following as some of NumPy's primary characteristics:

N-Dimensional Arrays: Based on similar primitives, NumPy presents the ndarray, a specialised version of an array. During calculations and operations, vectorized arrays can be precisely manipulated because to its strong structure.

Efficiency through Vectorization: A NumPy array's size and shape are fixed at m * n when it is expanded by default. Since a fresh array of the same size is made for every addition, this method improves computational performance while guaranteeing efficiency and correctness.

High-Level Mathematical Operations: NumPy accelerates and increases the efficiency of numerical computations by offering rapid, precompiled functions for

mathematical operations. The library takes an object-oriented approach, providing a flexible and expandable foundation to manage challenging mathematical tasks.

Foundation for SciPy and Pandas: NumPy's ndarray structural framework serves as the cornerstone for other well-known libraries like SciPy and Pandas, emphasising its critical role in the larger Python scientific community.

Due to its versatility, NumPy may be used for a wide range of mathematical tasks, making it an invaluable tool for scientists, engineers, and explorers. Its easy interaction with several Python IDEs makes it even more accessible, enabling accurate array manipulation for speedy calculations.

SciPy

One particularly useful tool for manipulating algebraic expressions, solving equations symbolically, and performing complex mathematical operations is SciPy, a symbolic mathematics library for Python. According to Mahalaxmi et al. (2023), it plays a crucial role in giving Python access to a symbolic layer, which enables complex mathematical calculations. Mahalaxmi et al. (2023) list the following as some of SciPy's primary characteristics:

Symbolic Computation: With SymPy, symbolic calculation is made easier and mathematical equations containing variables instead of numbers may be worked with. This is especially helpful for assignments that call for precise representations.

Equation Solving: The library is very good at solving equations symbolically and offers accurate answers to a wide range of mathematical issues.

Calculus Operations: SymPy is useful for jobs involving mathematical analysis since it can do a broad range of calculus operations, such as derivatives, integrals, and limit calculations.

Linear Algebra: SymPy's functionality is expanded to include linear algebra, enabling users to work with matrices and vectors symbolically.

Python integration: SymPy easily combines with Python, offering a symbolic layer to support libraries for numerical computation such as NumPy and SciPy.

Python's usefulness for algebraic tasks is improved by SymPy's symbolic approach, which bridges the gap between symbolic and numerical computation. For researchers and mathematicians working on a variety of mathematical issues, its integration with other scientific libraries makes it an invaluable tool.

5 Challenges and Limitations of Using Python for Algebra

5.1 Potential Pitfalls and Errors in Algebraic Computation

When using Python for algebraic computations, practitioners face difficulties that call for a sophisticated comprehension to guarantee precise and dependable outcomes. Floating-point arithmetic and numerical precision present a significant hurdle. Like many other programming languages, Python uses floating-point arithmetic, which can lead to issues with numerical precision. Because rounding mistakes are inevitable in complex algebraic calculations, users should proceed with caution and, if accuracy is critical, look into other methods or symbolic computation libraries like SymPy.

Although symbolic algebraic manipulation is a powerful tool, as expressions become more complicated, it can become computationally costly. This presents a performance barrier, as symbolic libraries such as SymPy may become slower as algebraic complexity rises. Using the right algorithms, optimisations, and, when practical, hybrid techniques that combine symbolic and numerical solutions in accordance with the task's particular needs are all necessary to mitigate this difficulty.

Another problem in using Python for algebraic computations is handling mistakes robustly. Although dynamic languages are flexible, they can be challenging to handle errors in, particularly when working with a variety of algebraic expressions and equations. In order to tackle this issue, practitioners are recommended to incorporate strong error-handling methods, properly verify inputs, and carry out extensive testing of algebraic functions.

Since NumPy and SymPy are frequently needed for Python's algebraic capabilities, external dependencies provide still another level of complication. Algebraic code behaviour and compatibility may be affected by updates or modifications to these dependencies. In order to preserve the integrity of their algebraic computations, practitioners should update dependencies on a regular basis, stay up to speed on library changes, and make sure that versions work together.

Selecting between symbolic and quantitative approaches presents a crucial trade-off. Numerical computations prioritise efficiency at the possible expense of symbolic precision, whereas symbolic computations deliver precision but may compromise speed. In order to successfully navigate this obstacle, practitioners must carefully evaluate the unique requirements of the algebraic problem, allowing them to strike a balance between efficiency and precision.

To put it simply, knowing these difficulties gives practitioners the ability to choose wisely when using Python for algebraic operations. People can use Python to explore the algebraic environment by comprehending the subtleties of numerical precision, symbolic manipulation, error handling, dependencies, and performance trade-offs.

5.2 Limitations of Symbolic Algebra in Python

Symbolic algebra provides a powerful foundation for working with mathematical expressions symbolically, made possible by Python packages such as SymPy. Nevertheless, there are some issues with this method that practitioners doing symbolic algebraic calculations should take into account.

A notable limitation is the computational burden that comes with symbolic algebra. The efficiency of symbolic computation libraries, particularly SymPy, may deteriorate as algebraic workloads become more complex. This means that solving complicated symbolic issues requires careful resource management.

One significant drawback appears when moving from symbolic to numerical computations. Symbolic algebra does well when it comes to exact representations, but it struggles with numerical analyses. When obtaining numerical results via symbolic programming, users should be aware of the inherent constraints since precision problems and rounding errors may arise.

Another problem that libraries for symbolic algebra face is handling huge expressions. The amount of memory needed to store and manipulate large symbolic expressions can cause inefficiencies, which may affect how well algebraic computations execute overall. It is important for practitioners to understand these constraints, particularly when working with large mathematical expressions.

Moreover, for some numerical tasks, symbolic algebra might not always be the best option. For certain kinds of calculations, numerically-focused libraries such as NumPy, whose algorithms are tailored for numerical computations, can perform faster and more efficiently with respect to resources than symbolic techniques. In such cases, it becomes crucial to balance the trade-offs between symbolic precision and numerical efficiency.

Although Python's symbolic algebra improves the symbolic manipulation of mathematical statements, practitioners should be aware of these constraints. It is possible to use Python's symbolic algebraic capabilities in a sophisticated and efficient way by being aware of the trade-offs between efficiency and processing intensity, numerical precision concerns, huge expressions, and other issues.

6 Conclusion

The essential significance of algebra in mathematics as well as the introduction of Python as a powerful programming language for algebraic tasks were covered in this investigation of algebra with Python. The fundamentals of algebra were studied, including how to navigate equations, functions, and basic operations. The symbiotic relationship between mathematics and programming was revealed with the integration of Python for algebraic manipulation, revealing the syntax, data types, and prospective applications.

The examination also covered algebraic applications in Python, explaining how to solve equations, graph functions, and identify patterns—all of which were conducted with ease thanks to Python tools. The difficulties and restrictions that came with using Python for algebra revealed possible problems with computational efficiency, symbolic manipulation, and numerical accuracy.

The challenges of dealing with huge expressions, trade-offs in numerical precision, and processing intensity were encountered when analysing the constraints of symbolic algebra in Python. Nevertheless, despite these difficulties, symbolic algebra's adaptability in providing accurate symbolic representations was still apparent.

These contributions are useful for understanding the complex interactions between symbolic and numerical methods when exploring the dynamic terrain of

algebra with Python. Understanding the complexities of computation, subtleties of precision, and successful techniques gives practitioners the skills they need to use Python efficiently for algebraic tasks. As such, this investigation not only highlights the mutually beneficial relationship between mathematics and Python but also opens up new avenues for future research by encouraging more study of algebraic domains with the use of this potent programming language.

Even with the wealth of material in this report, there were still certain restrictions on the investigation. The difficulties in managing big expressions, computing intensity, and numerical precision in symbolic algebra highlighted the need for more research to find solutions to these limitations. Recognising these constraints reminds us that in order to properly progress the field in the future, these obstacles must be wisely navigated.

As this examination into algebra with Python comes to an end, there are a number of exciting directions that could lead to more research and development in this dynamic nexus of programming and mathematics.

First and foremost, there is a need to improve Python's symbolic algebraic computation efficiency. Subsequent studies may concentrate on improving symbolic algebra's computational efficiency, possibly by investigating new algorithms, optimising existing ones, or utilising parallel processing capabilities.

Combining machine learning methods with algebraic computations is an intriguing avenue for further research. There is potential for novel applications when examining how machine learning models can help with algebraic problem solving, symbolic manipulation optimisation, or pattern identification within algebraic structures.

Future work may also focus on creating specialised algebraic libraries made specifically for Python. By filling in holes in the existing Python ecosystem and offering optimised solutions for a range of mathematical problems, these specialised libraries might handle particular algebraic challenges.

Technological developments in algebraic tasks human-computer interface offer a fascinating prospect. Algebraic problem-solving could be made more approachable for a wider audience by incorporating natural language processing and user-friendly interfaces to make the process easier.

Examining Python's function in quantum algebraic computations becomes fascinating as quantum computing gains popularity. Analysing Python's utility for quantum algebraic tasks and its ability to interface with quantum computing languages may delineate a new area of algebraic research.

Furthermore, the creation of interactive teaching resources that use Python to teach and learn algebra has promise. Developing compelling platforms that combine algebraic ideas with Python programming could help students gain a deeper understanding of both algebra and programming.

In summary, there is a great deal of promise for algebraic investigation with Python in the future. This field can advance into fascinating new domains by addressing computational difficulties, investigating multidisciplinary intersections, and supporting innovative teaching practices. The future course of algebraic investigation using Python will probably be shaped by the combined efforts of mathematicians, computer scientists, and educators.

References

Alibali, M. W. (1999). How children change their minds: Strategy change can be gradual or abrupt. Developmental Psychology, 35, 127–145. doi:https://doi.org/10.1037/0012-1649.35.1.127.

Alibali, M. W., Knuth, E. J., Hattikudur, S., McNeil, N. M., & Stephens, A. C. (2007). A longitudinal look at middle-school students' understanding of the equal sign and equivalent equations. Mathematical Thinking and Learning, 9, 221–247. https://doi.org/10.1080/10986060701360902

Barnes, M. (1991). Investigating change: An introduction to calculus for Australian schools. Carlton, VIC: Curriculum Corporation.

Bosse, M. J., & Nandakumar, N. R. (2005). Section A factorability of quadratics: Motivation for more techniques. Teaching Mathematics and Its Applications, 24(4), 143-153.

Bush, S. B., & Karp, K. S. (2013). Prerequisite algebra skills and associated misconceptions of middle grade students: A review. The Journal of Mathematical Behavior, 32, 613–632. https://doi.org/10.1016/j.jmathb.2013.07.002

Church, K. et al. (2021) 'Introduction to python's syntax', The Quantitative Methods for Psychology, 17(1). doi:10.20982/tqmp.17.1.s001.

Dhruv, A., Patel, R., Doshi, N. (2022). Python: The Most Advanced Programming Language for Computer Science Applications. DOI: 10.5220/0010307900003051

Edo, S., Tasik, W. (2022). Investigation of Students' Algebraic Conceptual Understanding and the Ability to Solve PISA-Like Mathematics Problems in a Modeling Task. Mathematics Teaching Research Journal, 14(2).

Falkner, K. P., Levi, L., & Carpenter, T. P. (1999). Children's understanding of equality: A foundation for algebra. Teaching Children Mathematics, 6(4), 232–236 Retrieved from https://eric.ed.gov/?id=EJ600209.

Harripersaud, A. (2021), The Quadratic Equation Concept, American Journal of Mathematics and Statistics, Vol. 11 No. 3, 2021, pp. 67-71. doi: 10.5923/j.ajms.20211103.03.

Howden, H. (2001). Algebra tiles for the overhead projector. Veron Hills: Learning Resources.

Kehler-Poljak, G., Jukić Matić, L. and Rukavina, S. (2022) 'The influence of curriculum on the concept of function: An empirical study of pre-service teachers', European Journal of Science and Mathematics Education, 10(3), pp. 380–395. doi:10.30935/scimath/12042.

Kieran, C. (1997). Mathematical concepts at the secondary school level: The learning of algebra and functions. In T. Nunes & P. Bryant (Eds.), Learning and teaching mathematics: An international perspective (pp. 133–158). East Sussex: Psychology Press.

Kieran, C., Pang, J., Schifter, D., & Ng, S. F. (2016). Early algebra: Research into its nature, its learning, its teaching. New York: Springer (open access eBook). https://doi.org/10.1007/978-3-319-32258-2.

Libretexts (2020) 1.1: Four ways to represent a function, Mathematics LibreTexts. Available at: https://math.libretexts.org/Bookshelves/Calculus/Map%3A_Calculus__Early_Transcendentals_(Stewart)/01%3A_Functions_and_Models/1.01%3A_Four_Ways_to_Represent_a_Function (Accessed: 21 January 2024).

Grønmo, L.S. (2018). The Role of Algebra in School Mathematics. In: Kaiser, G., Forgasz, H., Graven, M., Kuzniak, A., Simmt, E., Xu, B. (eds) Invited Lectures from the 13th International Congress on Mathematical Education. ICME-13 Monographs. Springer, Cham. https://doi.org/10.1007/978-3-319-72170-5_11

Meurer, A. et al. (2016) Sympy: Symbolic computing in python [Preprint]. doi:10.7287/peerj.preprints.2083v3.

Norton, S. (2015). Teaching and learning fundamental mathematics: Quadratic equations. Producer Freddy Komp. Brisbane.

Otten, M., Van den Heuvel-Panhuizen, M. & Veldhuis, M. The balance model for teaching linear equations: a systematic literature review. IJ STEM Ed6, 30 (2019). https://doi.org/10.1186/s40594-019-0183-2

Srinath, K. (2017). Python – The Fastest Growing Programming Language. International Research Journal of Engineering and Technology (IRJET), 4(12).

VanderPlas, J. (2023) Python Data Science Handbook. O'Reilly Media, Inc.

Veith, J.M. et al. (2023) 'Mathematics Education Research on algebra over the last two decades: Quo Vadis?', Frontiers in Education, 8. doi:10.3389/feduc.2023.1211920.

Appendix 1: Bitwise Operators

Python Bitwise Operators:

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

There are following Bitwise operators supported by Python language

[Show Example]

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(a & b) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(a | b) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(a ^ b) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~a) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 will give 15 which is 0000 1111

Appendix 2: Membership Operators

Python Membership Operators:

In addition to the operators discussed previously, Python has membership operators, which test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators explained below:

[Show Example]

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here **in** results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here **not in** results in a 1 if x is not a member of sequence y.

Appendix 3: Identity Operators

Python Identity Operators:

Identity operators compare the memory locations of two objects. There are two Identity operators explained below:

[Show Example]

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here **is** results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here **is not** results in 1 if id(x) is not equal to id(y).