# Guide for "Applied Numerical Method" Library

## Introduction

The 'Applied Numerical Method' library is a collection of numerical methods for solving mathematical problems. It includes methods such as the Newton-Raphson method for root finding and the Bisection method for interval root finding. This guide provides instructions on how to use the library, examples of its functionality, and support information.

## Installation

To install the library, ensure that you have Python and the 'sympy' library installed on your system. Then, download the 'applied\_numerical\_method.py' file from the provided link and place it in your project directory.

## Using the Library

To use the library, import the desired function from the 'applied\_numerical\_method' module in your Python script. For example, use 'from applied\_numerical\_method import newton\_raphson' to import the Newton-Raphson method function.

## Function Documentation

The library includes the following functions:  
1. Newton-Raphson Method: Used to find the root of a real-valued function.  
2. Bisection Method: Used to find the root of a function within a given interval where the function changes sign.

### 1)Newton-Raphson Method

The Newton-Raphson method is an iterative root-finding algorithm that uses function values and their derivatives to quickly converge to a solution. It requires an initial guess and iteratively improves the estimate of the root.

Function Signature: newton\_raphson(func\_str, initial\_guess, decimal\_places=4, max\_iterations=100)

#### Parameters:

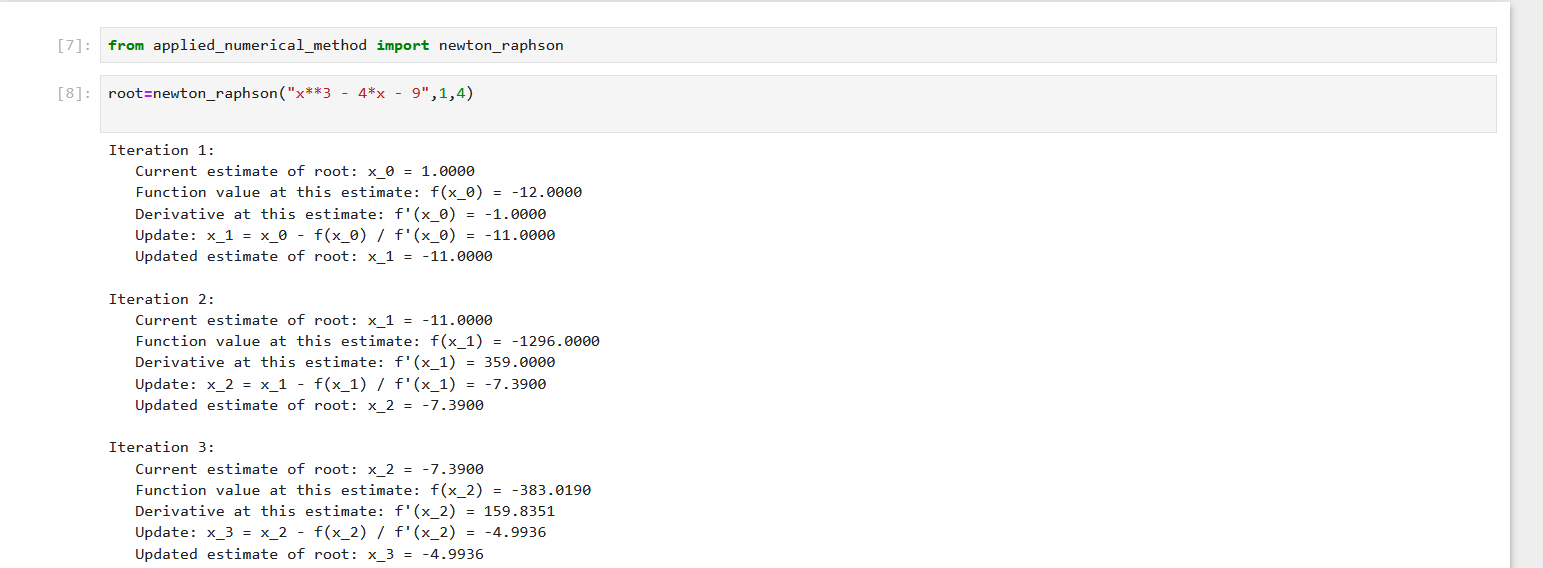
* func\_str: A string representing the mathematical function whose root is to be found.
* initial\_guess: A numeric value that is the starting point for the algorithm.
* decimal\_places: The number of decimal places of precision for the estimated root (default is 4).
* max\_iterations: The maximum number of iterations to perform if the root is not found (default is 100).
* **Warning:**  
  When using logarithmic functions like log(x, 10), it is important to provide interval bounds (a and b) where the logarithmic function is defined and behaves well. Typically, for log(x, 10), the interval should exclude zero and negative numbers. Choosing appropriate intervals is crucial for the accurate and efficient convergence of the method.

#### Returns:

The estimated root of the function as a float.

Example:

>>> from applied\_numerical\_method import newton\_raphson  
>>> root = newton\_raphson('x\*\*2 - 4', 1.0)  
>>> root  
2.0000



### 2)Bisection Method

The Bisection method is a simple root-finding algorithm that repeatedly bisects an interval and then selects a subinterval in which a root must lie for further processing. It is a very robust method but can be slower than other methods like the Newton-Raphson method.

Function Signature: bisection\_method(func\_str, a=-5, b=5, precision=0.0001, max\_iterations=100)

#### Parameters:

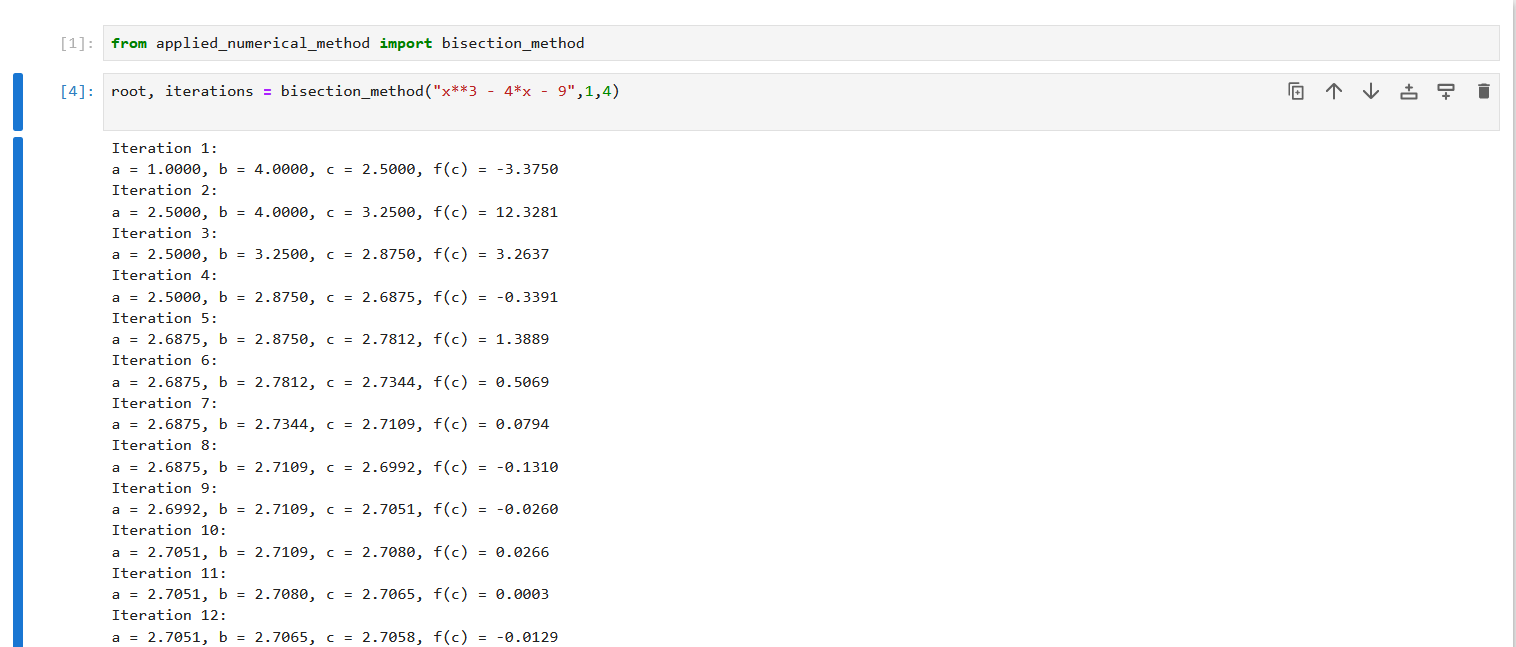
* func\_str: A string representing the mathematical function whose root is to be found within the interval [a, b].
* a: The lower bound of the interval where the root is to be searched for (default is -5).
* b: The upper bound of the interval where the root is to be searched for (default is 5).
* precision: The precision with which to determine the root (default is 0.0001).
* max\_iterations: The maximum number of iterations to perform if the root is not found (default is 100).

#### Returns:

A tuple containing the estimated root and the number of iterations taken to find it.

Example:

>>> from applied\_numerical\_method import bisection\_method  
>>> root, iterations = bisection\_method('x\*\*3 - 4\*x - 9')  
>>> root  
2.7065



# 3)Secant Method

The Secant Method is an iterative technique used for finding the roots of a real-valued function. It's a modification of the Newton-Raphson method that does not require the computation of derivatives. This method uses two initial guesses to start the iteration process and is generally faster than the Bisection method under suitable conditions.

## Function Signature:

python  
def secant\_method(func\_str, x0=0.0, x1=1.0, decimal\_places=4, max\_iterations=100)

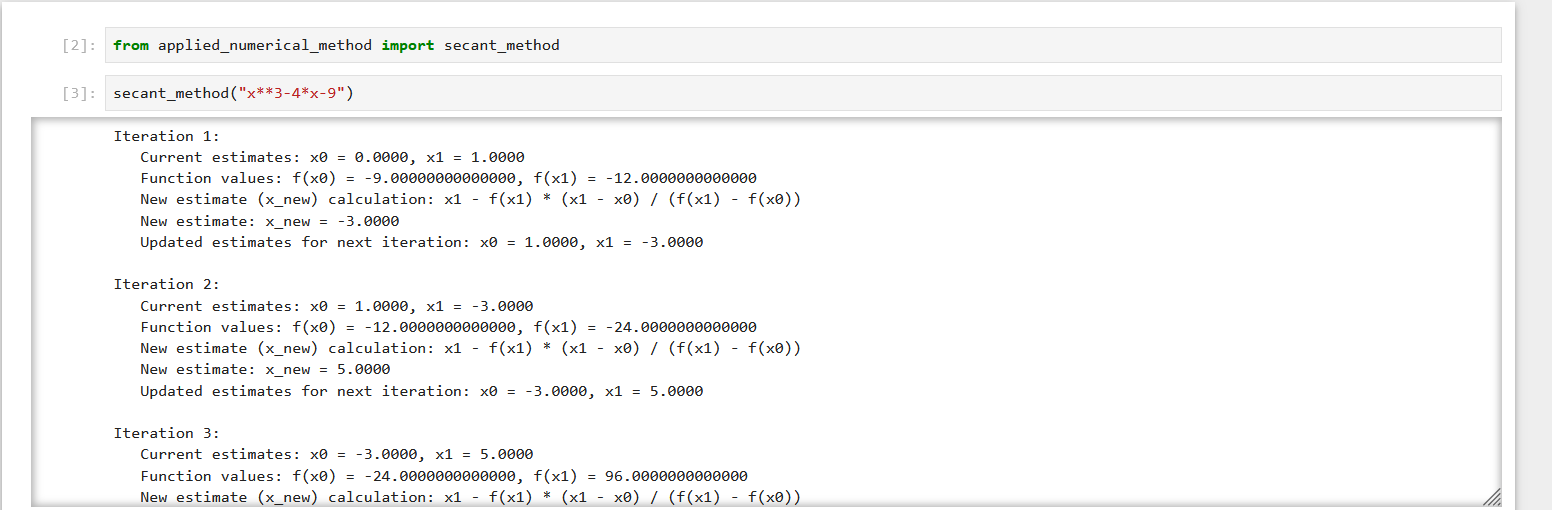
## Parameters:

func\_str: String representation of the function for which the root is to be found.  
x0, x1: Initial guesses for the root.  
decimal\_places: Precision of the solution.  
max\_iterations: Maximum number of iterations to perform.  
The method iteratively refines the guesses based on the function values at these points, moving closer to the root in each step. It stops when the difference between successive approximations is less than the specified tolerance or when the maximum number of iterations is reached.

**Warning:**  
When using logarithmic functions like log(x, 10), it is important to provide interval bounds (a and b) where the logarithmic function is defined and behaves well. Typically, for log(x, 10), the interval should exclude zero and negative numbers. Choosing appropriate intervals is crucial for the accurate and efficient convergence of the method.

## Example Usage:

python  
root = secant\_method('x\*\*2 - 5', 2.0, 3.0, decimal\_places=5)  
This call finds the square root of 5 using the Secant Method with initial guesses 2.0 and 3.0, and a precision of 5 decimal places.



# 4)False Position Method Function Overview

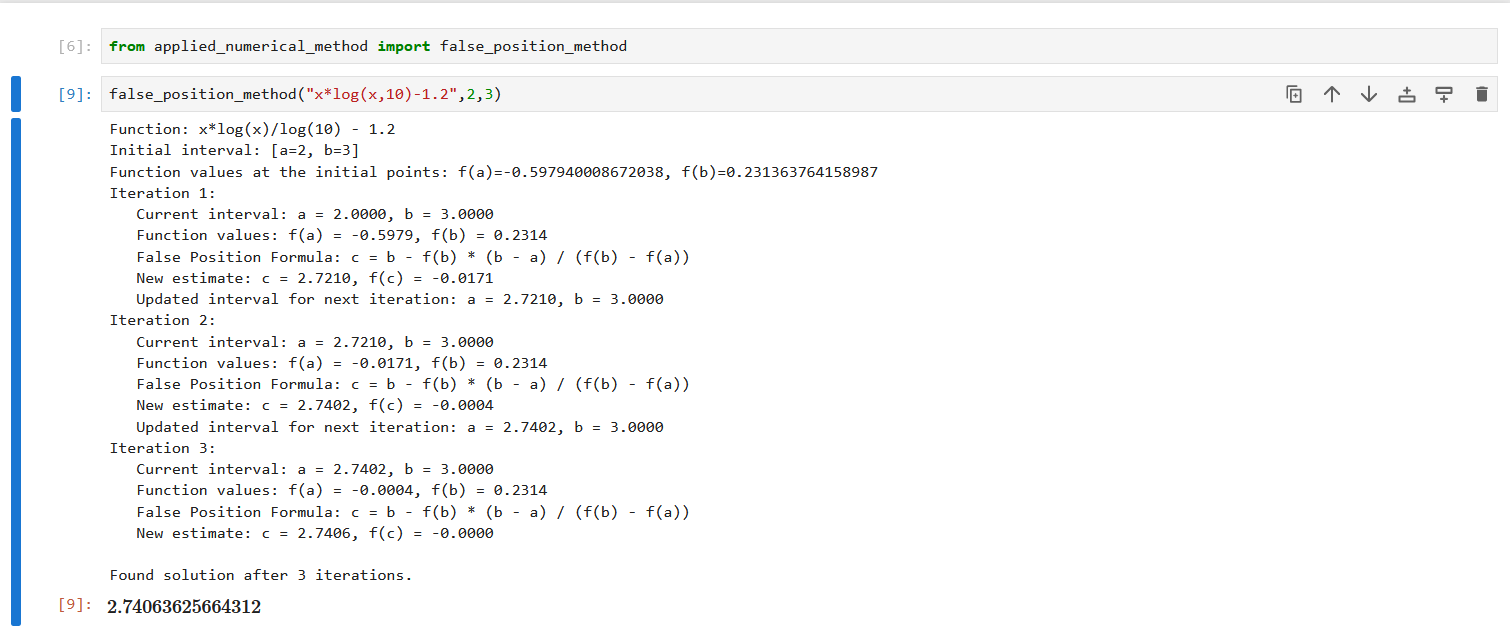
**Overview:**  
The False Position Method, also known as the Regula Falsi Method, is a root-finding algorithm that iteratively converges to a root of a given function. This method is particularly useful for functions that are continuous in the given interval. It works by narrowing down the interval where the root lies based on the Intermediate Value Theorem.

**Usage:**  
The function 'false\_position\_method' takes a string representation of a function, interval bounds (a and b), the number of decimal places for precision, and the maximum number of iterations as arguments. It calculates the root of the function within the specified interval and precision.

**Warning:**  
When using logarithmic functions like log(x, 10), it is important to provide interval bounds (a and b) where the logarithmic function is defined and behaves well. Typically, for log(x, 10), the interval should exclude zero and negative numbers. Choosing appropriate intervals is crucial for the accurate and efficient convergence of the method.

**Function Prototype:**  
false\_position\_method(func\_str, a=0, b=1, decimal\_places=4, max\_iterations=100)

**Parameters:**  
func\_str: String representation of the function to find the root of.  
a: The lower bound of the interval (default value is 0.0).  
b: The upper bound of the interval (default value is 1.0).  
decimal\_places: The number of decimal places for the root's precision (default value is 4).  
max\_iterations: The maximum number of iterations to perform (default value is 100).



Newton's Forward Interpolation Method

# Introduction

Newton's Forward Interpolation is a numerical method used for estimating the values of a function at a given point using linear interpolation. This method is particularly useful when dealing with equally spaced data points. The method uses a polynomial approximation to interpolate the values of the function.

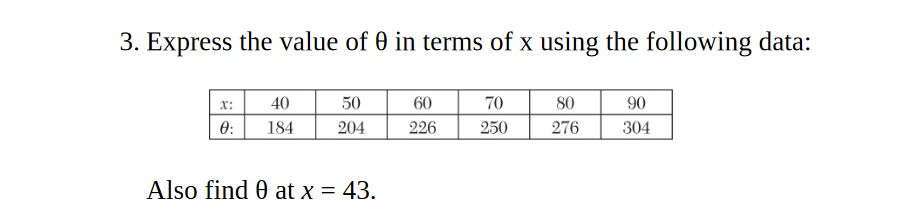
# Implementation in Python

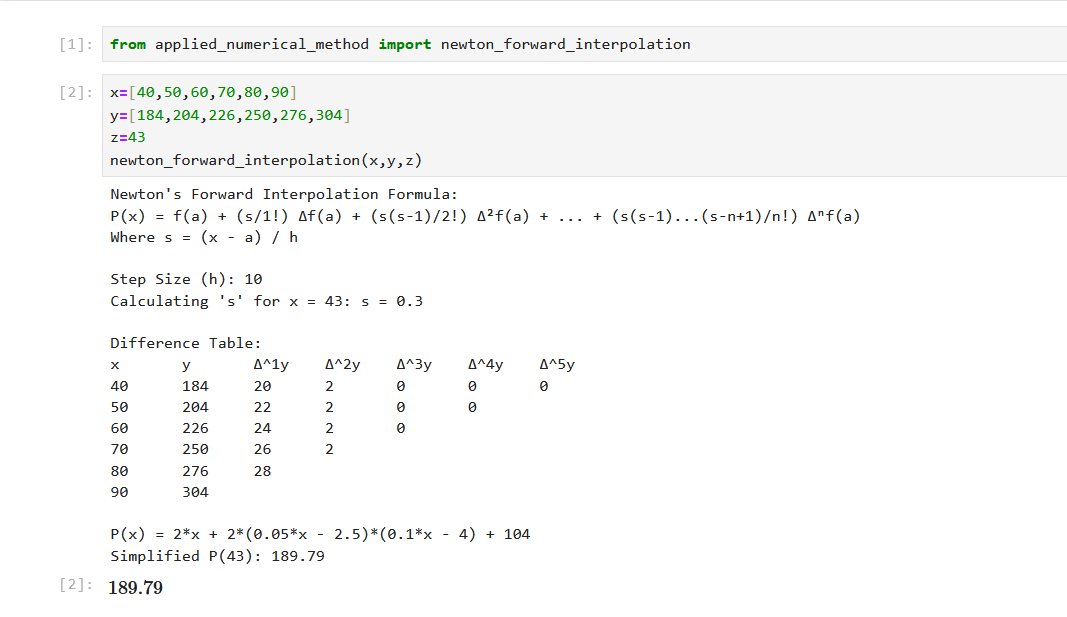
The implementation of Newton's Forward Interpolation in Python involves several steps: creating a difference table, formatting the table for display, and applying the formula to estimate the function value at a given point.

# Example Usage

To use this implementation, input the x and y data points along with the x value at which the function needs to be estimated. The function returns the estimated y value.

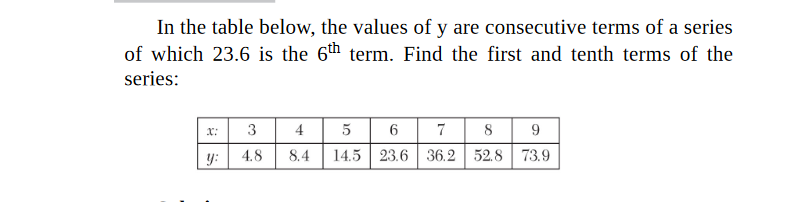
x\_data = [0, 1, 2, 3]  
y\_data = [1, 3, 7, 13]  
x\_value\_to\_estimate = 2.5  
  
estimated\_value = newton\_forward\_interpolation(x\_data, y\_data, x\_value\_to\_estimate)  
print(f"Estimated value at x={x\_value\_to\_estimate} is: {estimated\_value}")

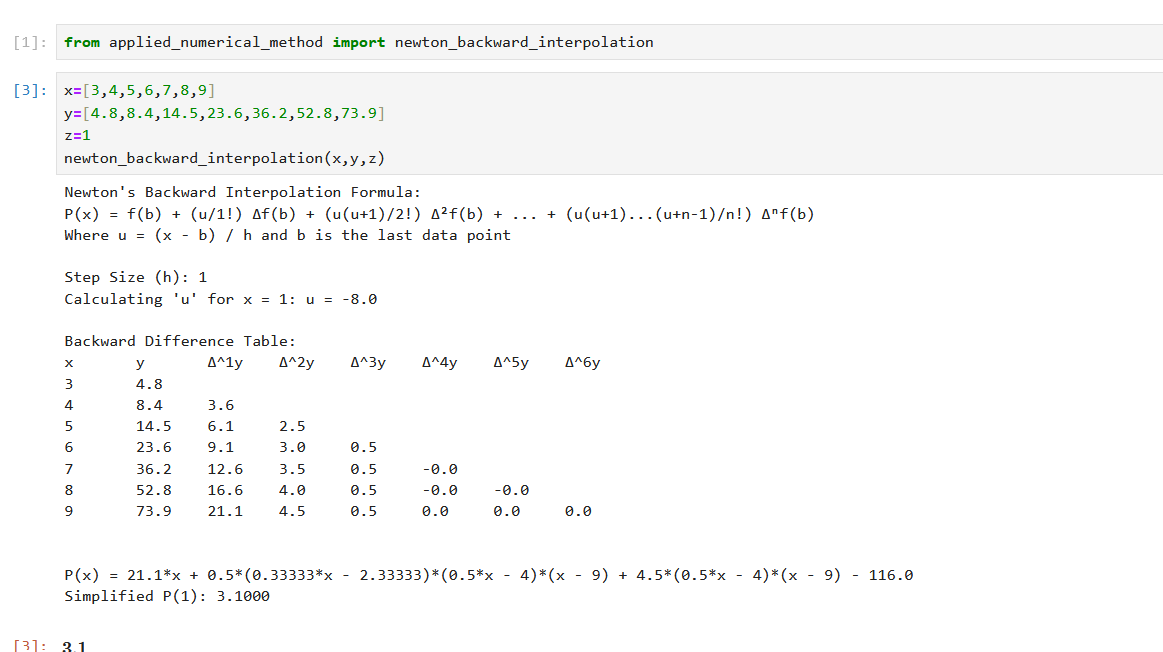




# Newton's Backward Interpolation

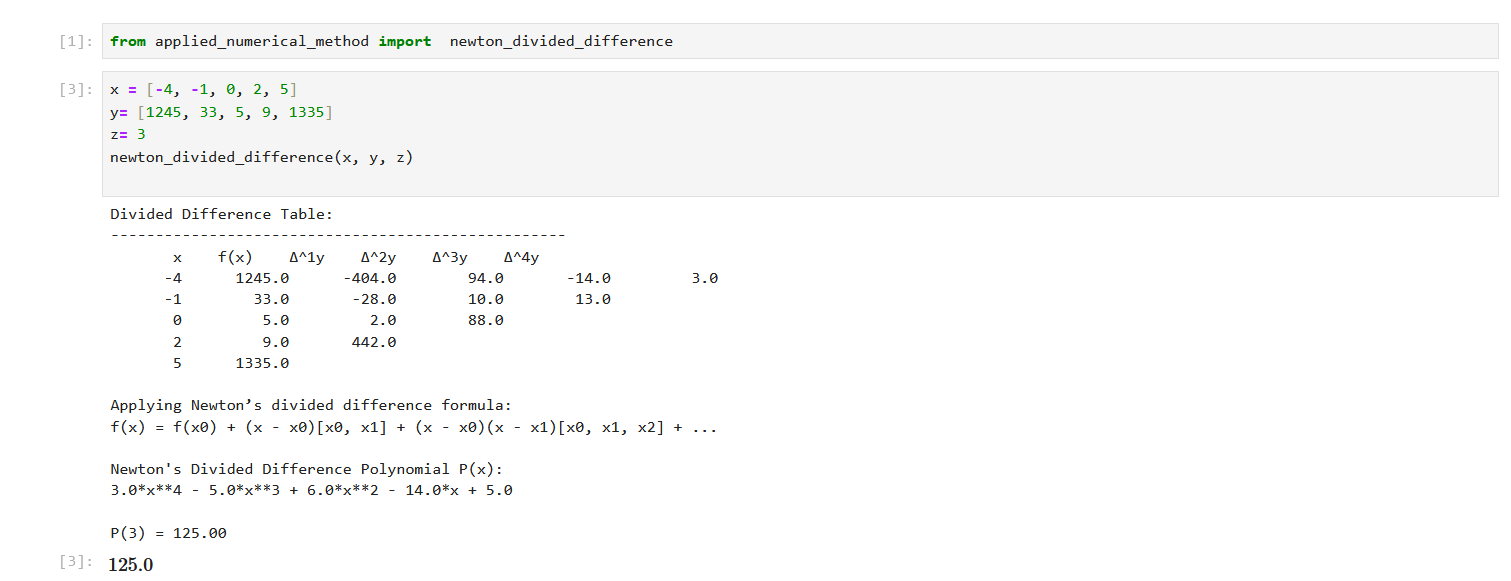
Newton's Backward Interpolation is used for interpolating values of a function at points near the end of a set of known values.  
  
To use this method:  
1. Prepare your data points (x\_data and y\_data).  
2. Call the newton\_backward\_interpolation function with your data points and the x value for which you want the interpolated y value.  
3. The function will return the interpolated value at the specified x.  
  
The function internally creates a backward difference table and applies Newton's backward interpolation formula to compute the result





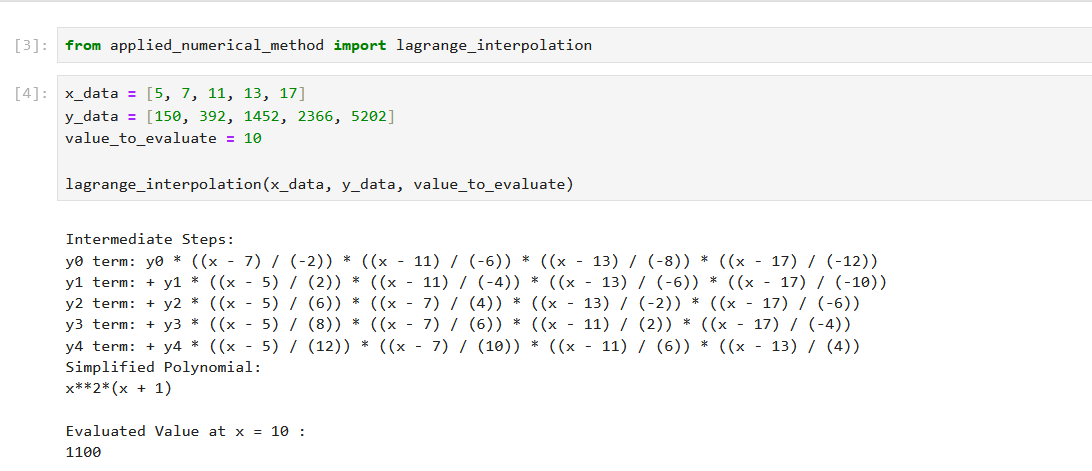
Newton’s Divided Difference Method

When you need only p(x) then give random value of z becaue function always calculate p(x)



Lagrange Interpolation Formula

When you need only p(x) then give random value of z becaue function always calculate p(x)



cubic spline(INTERPLOATION)

