Discrete Structures!

CMPSC 102

Allegheny College

Key Questions and Learning Objectives

- How do I use **iteration** and **conditional logic** in a Python program to perform computational tasks like processing a file's contents and mathematical tasks like using Newton's method to approximate the square root of a number?

- To remember and understand some discrete mathematics and Python programming concepts, setting the stage for exploring discrete structures.

Python Programming Retrospective

- Python code is designed to be **intuitive**

- Key components of Python programming include:
    - Function calls
    - Assignment statements
    - Iteration constructs
    - Conditional logic
    - Variable creation
    - Variable computations
    - Variable output

Python Programming Retrospective

- Investigate the **syntax** and **semantics** of these components

```
x = 5
print("x + 3")
```

- Understand how to **connect** these components together in a program

```
def square(n):
        return n * n

num = int(input("Enter a number: "))
print("The square is:", square(num))
```

- **Implement** Python functions to **understand** mathematical functions

A program is a sequence of statements

Iteration-based program in Python

Using a for loop to iterate over a range of numbers for num in range(1, 6):

```
for num in range(1, 6):
        square = num ** 2
        print(f"The square of {num} is: {square}")
```

A program is a sequence of statements

Programming parallels cooking …

- A Python program is a sequence of statements about mixing things with the rest of the ingredients … like a *recipe*
- There is a list of ingredients
- There is a sequence of events about when to use each ingredient
- Timing (run time) is important
- (Chef, waiter, guests) == (programmers, instructions, users)

Simple and compound statements – main()

Determine whether an integer is prime

(Primes can only be divided by one or themselves.)

```
def main()-> None:
        """ driver function of the program """
        # Example usage
        user_input = int(input("\t Enter a number: "))
        result = is_prime(user_input)

        if result:
                print(f"\t {user_input}: Prime number")
        else:
                print(f"\t {user_input}: Not a prime number")
# end of main()

# place at bottom of the file
main() # call the main() function
```

Simple and compound statements – is_prime()

```
def is_prime(number: int) -> bool:
        """ Determine primality: return 0 and 1"""
        # Handle special cases for 0 and 1
        if number < 2:
                return False

        # Iterate from 2 to the square root of the number
        for i in range(2, int(number**0.5) + 1):
                # If the number is divisible
                # by any value in the range, it's not prime
                if number % i == 0:
                        return False

        # If no divisors are found, the number is prime
        return True
# end of is_prime()
```

Simple and compound statements

Run the program: python3 isPrime.py or, python isPrime.py

Output from program:

Enter a number: 101

101: Prime number

- Programs contain both **simple** and **compound** statements (i.e., steps having multiple processes on one line)
- Which of these statements were simple?
- Which of these statements were compound?

Industry Standard Python – Part 1

Be like the Python professionals

- Always use Python 3 for all of your programs!
- Python2 is no longer supported…
- Add **docstrings** to your Python programs (i.e., informed comments to help others follow reasoning behind the code, including)
  - Modules
  - Classes
  - Functions
- Add comments to enhance understanding of important lines of code

Industry Standard Python – Part 2

Be like the Python professionals

- Add comments for important blocks of your program
- Use descriptive variable and function names
- The book does not always adhere to industry standards!
- All course projects will enforce these standards in GitHub Actions

Program to calculate area of a square - Function squareArea()

```
def squareArea(s: float ) -> float:
        """ determine area of square"""
        return s*s # area of square is s*s
# end of squareArea()
print(squareArea("5"))
```

What inputs *s* are acceptable?

- integers?
- floats?
- booleans?
- strings?
- imaginary numbers? ($1 + 3j$)

File: squareArea.py - Function main()

```
def main() -> None:
        sideLength = 5
        # Testing value
        print(f"Length {sideLength}")
        print(f" Area: {squareArea(sideLength)}")

        # These inputs work
        testValues_list = [2, 0, -3, 2 + 5j]

        # why will these inputs not work?
        # testValues_list = [True, "radius"]

        print(f"\n Iterating over the list.")
        for val in testValues_list: # iteration
        print(f" Length {val}, Area: {squareArea(val)}")
        # end main()

main() # call the driver function
```

File: squareArea.py - Initiation code

Run the program: python3 isPrime.py or, python isPrime.py

Output from program:

Length 5

Area: 25

Iterating over the list.

Length 2, Area: 4

Length 0, Area: 0

Length -3, Area: 9

Length (2+5j), Area: (-21+20j)

Lab 01

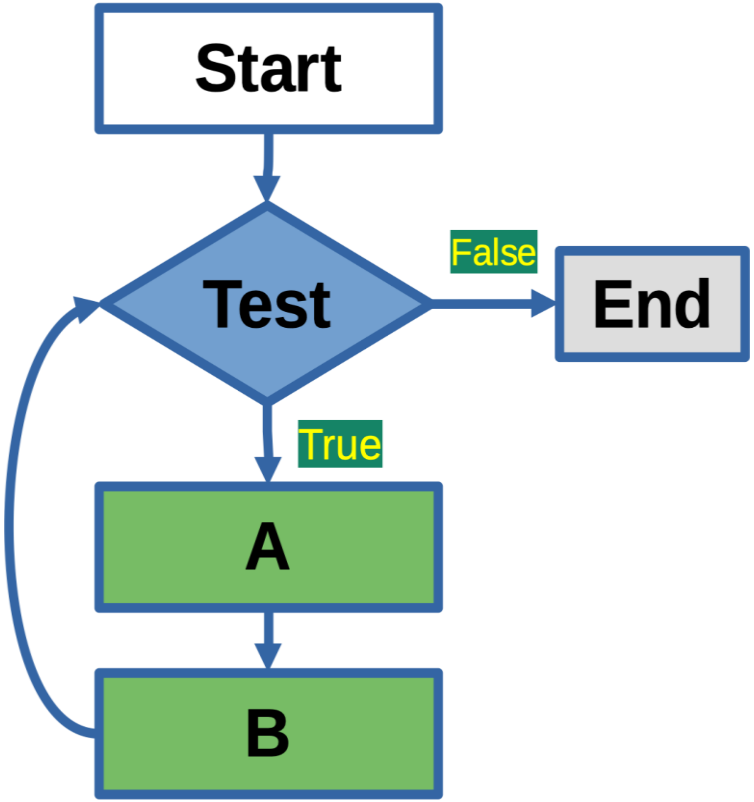
```python
cli = typer.Typer()

@cli.command()
def word(
        word: str = typer.Option(None),
        dir: Path = typer.Option(None),
        file: Path = typer.Option(None),
) -> None:
```

pyproject.toml:

```toml
[tool.poetry.scripts]
search = "search.main:cli"
```

Loops for Iteration - A loop is a way to reuse code blocks



The While Loop

```python
index = 0
while (index < 10): # condition
        print(f"Count :{index}")
        index += 1 # add one to index
```

Output

```
Count :0
Count :1
Count :2
Count :3
Count :4
Count :5
Count :6
Count :7
Count :8
Count :9
```

The for … in range() Loop

```python
for index in range(10):
            print(f"Count :{index}")
```

Output

```
Count :0
Count :1
Count :2
Count :3
Count :4
Count :5
Count :6
Count :7
Count :8
Count :9
```

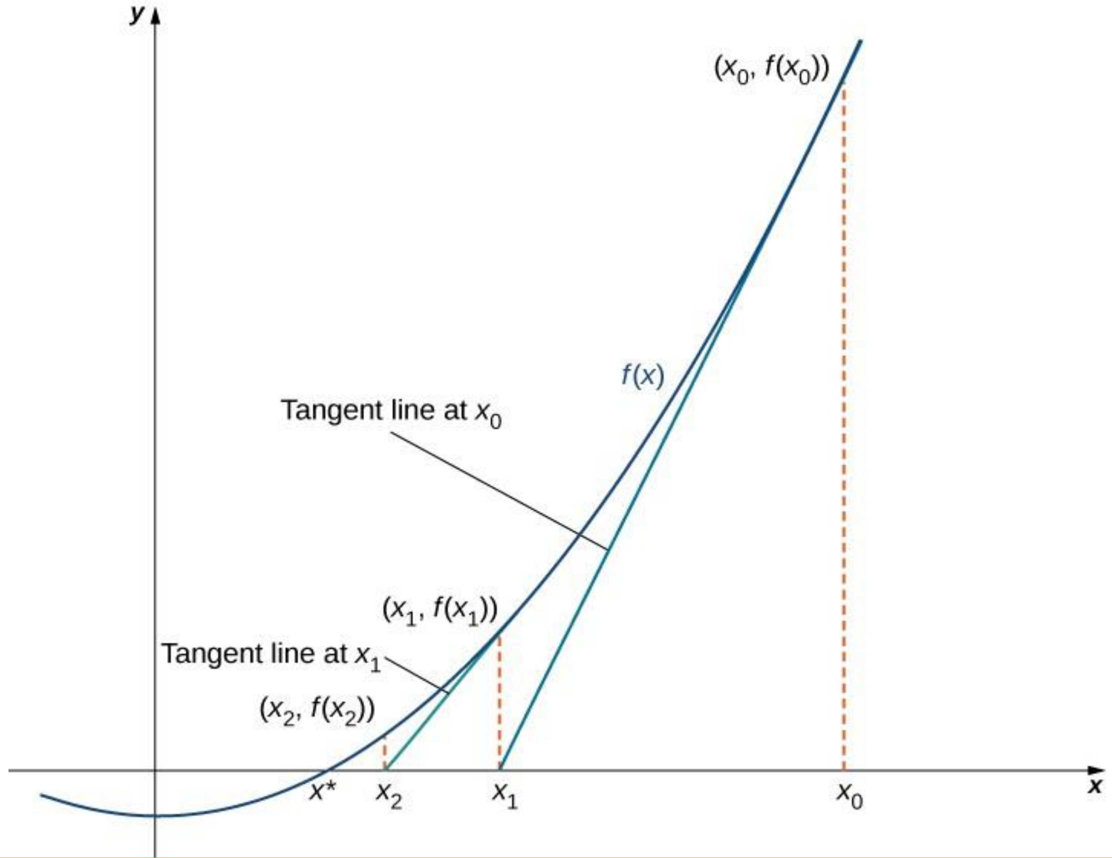
Square Roots - Mathematical loops to find quadruple roots

How to compute : $\sqrt{x}$?

Method

The function initializes the guess for the square root and iteratively refines it using an approximation formula until the approximation is within the specified tolerance.

Newton's Method - Mathematical loops to find square roots



The While Loop Application - Finding a Square Root

```
n = 4
guess = 1.0
while abs(n - guess*guess) > 0.0001:
        guess = guess - (guess*guess - n)/(2*guess)
        print(f"n = {n} : guess = {guess}")
```

- Iteratively **guesses** the square root until **within tolerance**
- The while loop uses ‘abs‘ for computing an absolute value
- This loop computes the root as 2.000000000929222947
- The math.sqrt(n) function confirms this approximation!
- Any questions about this way to approximate a square root?

The While Loop Application - Finding a Square Root

```python
n = 4
guess = 1.0
while abs(n - guess*guess) > 0.0001:
        guess = guess - (guess*guess - n)/(2*guess)
        print(f"n = {n} : guess = {guess}")
```

Output

```
n = 4 : guess = 2.5
n = 4 : guess = 2.05
n = 4 : guess = 2.000609756097561
n = 4 : guess = 2.0000000929222947
```

The For Loop Application - Finding a Square Root

```python
n = 4
guess = 1.0
for i in range(5):
        abs(n - guess*guess) > 0.0001
        guess = guess - (guess*guess - n)/(2*guess)
        print(f"n = {n} : guess = {guess}")
```

Output

```
n = 4 : guess = 2.5
n = 4 : guess = 2.05
n = 4 : guess = 2.000609756097561
n = 4 : guess = 2.0000000929222947
```

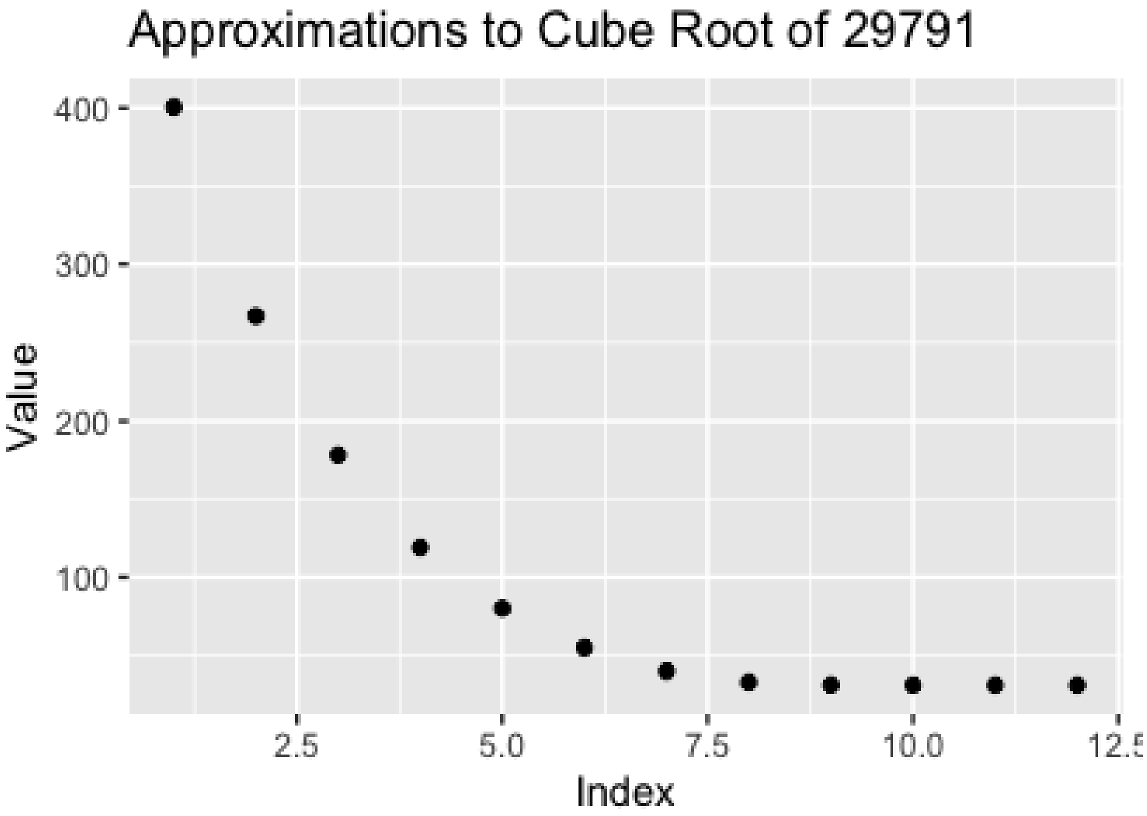
Cube Roots - Mathematical loops to find cube roots

How to compute : $\sqrt[3]{x}$?

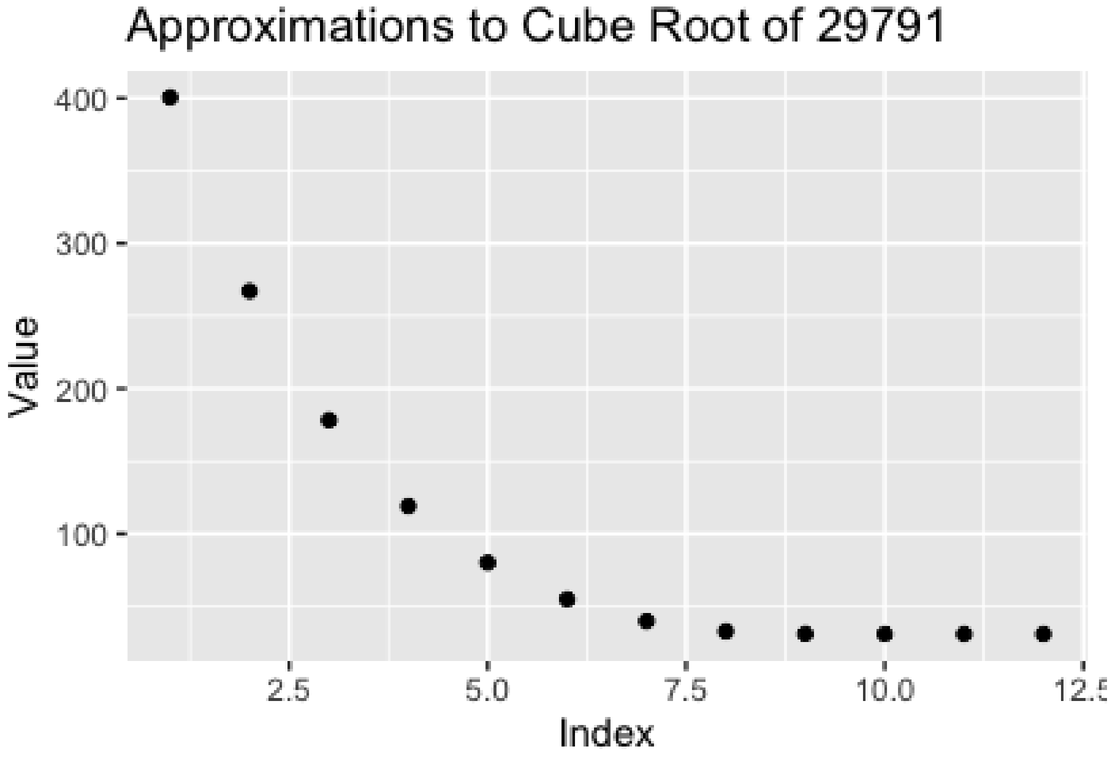
Method

The function initializes the guess for the cube root and iteratively refines it using an approximation formula until the approximation is within the specified tolerance.

Cube Roots - Approximations to find cube roots; result = 31



Cube Roots - Approximations to find cube roots; result = 31



400.5466666666667,
267.0930063983859,
178.2012041240575,
119.1135133172400 8,
80.10891816753315,
54.9533436656015 9,
39.92389386164385 5,
32.84607262847925 6,
31.10181238464575 4,
31.0003329211397 3,
31.0000000035753 2,
31.0

Cube Roots - Code

```
def cube_root_approximation(number, tolerance=1e-6):
        # Initial guess for the cube root
        # guess = number / 2.0 # one way to start
        guess = 5
        # Iterate until the approximation
        # is within the specified tolerance
        while abs(guess**3 - number) > tolerance:
                # Update the guess using the approximation formula
                guess = (2 * guess + number / (guess**2)) / 3.0
                print(f" guess = {guess}")
        return guess

# Example: Calculate the cube root of 29791 i
input_number = 29791
result = cube_root_approximation(input_number)

# Display the result
print(f"The cube root of {input_number}")
print(f" is approximately: {result}")
```

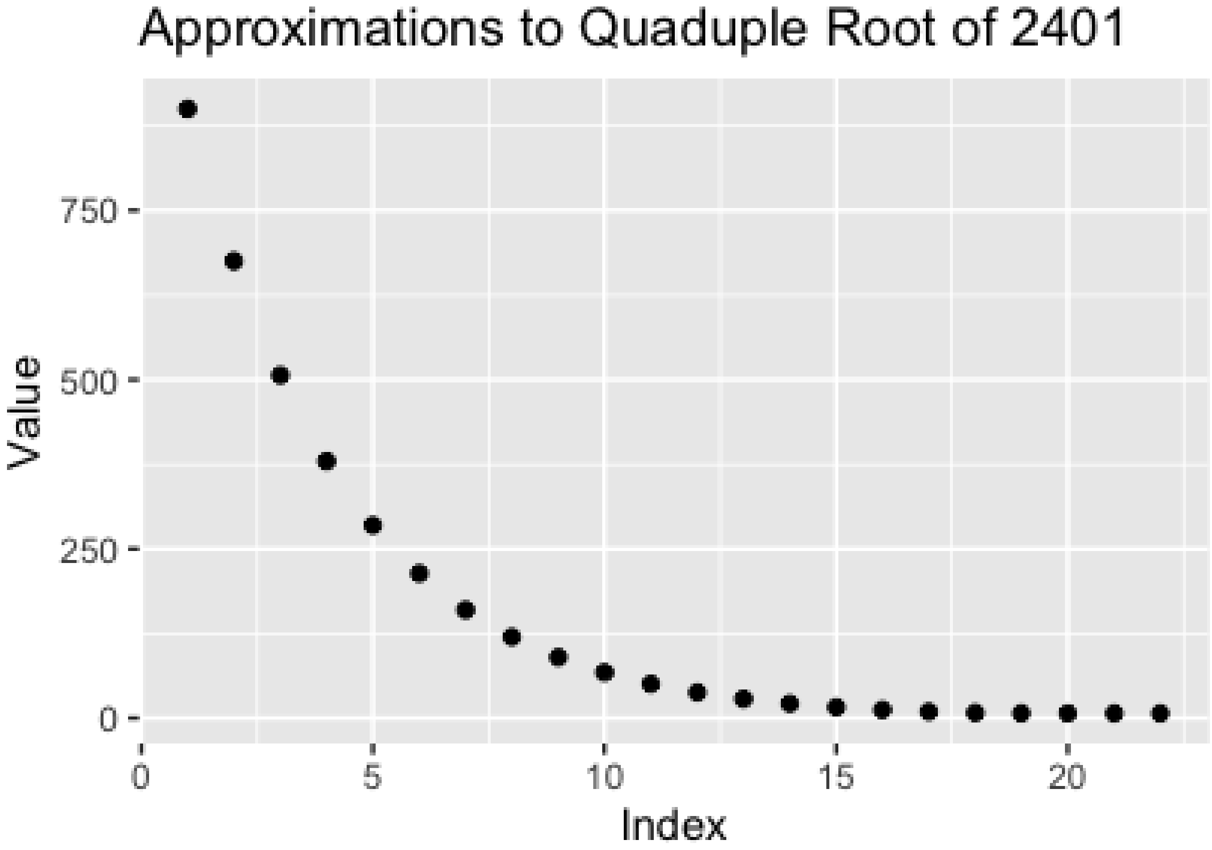
Quadruple Roots - Mathematical loops to find cube roots

How to compute : $\sqrt[4]{x}$?

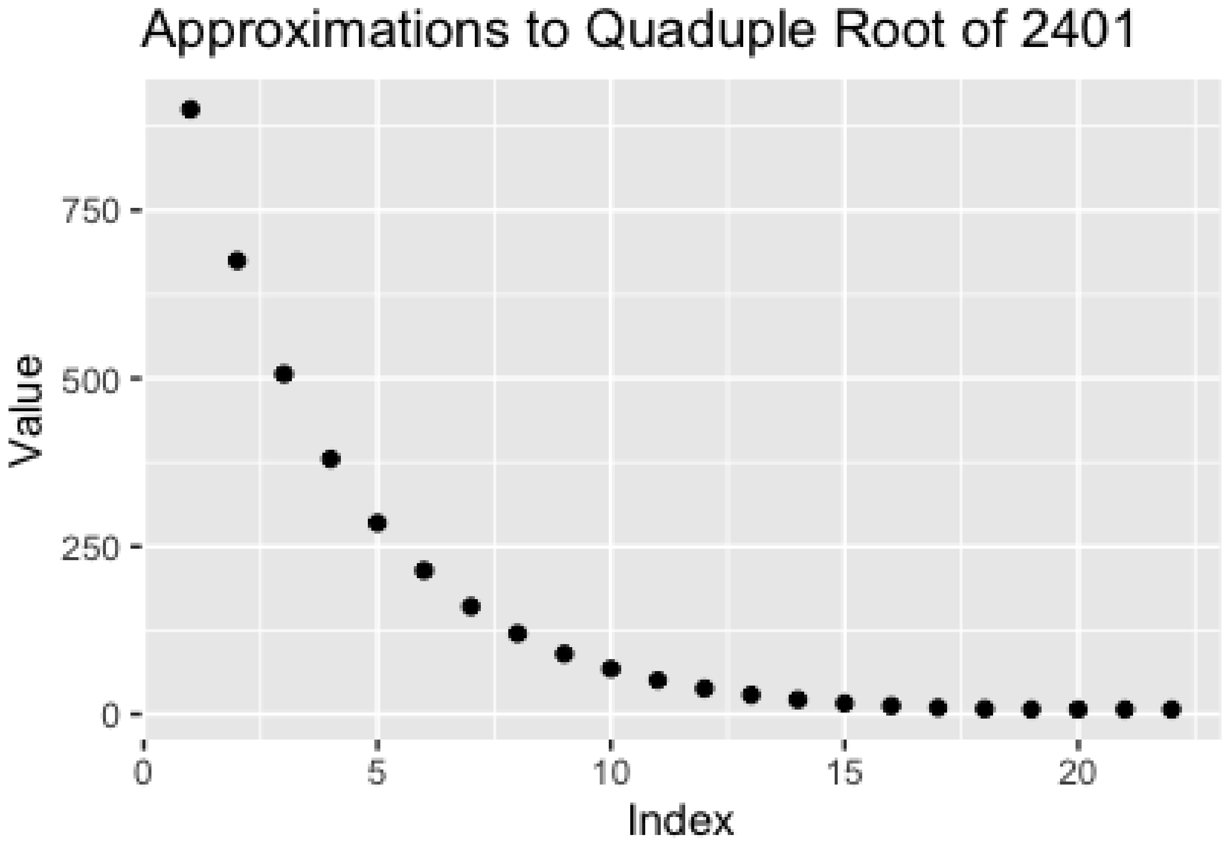
Method

The function initializes the guess for the quadruple root and iteratively refines it using an approximation formula until the approximation is within the specified tolerance.

Quadruple Roots - Approximations to find quadruple roots; result = 7



Quadruple Roots - Approximations to find quadruple roots; result = 7



900.375000346933,
675.2812510825596,
506.46094026121705,
379.8457098164694,
284.8842933147822,
213.66324594739956,
160.24749599845413,
120.18576786629791,
90.13967165836334,
67.60557331037666,
50.70612258852655,
38.03419610727984,
28.5365566959587,
21.428247703754217,
16.132191727838684,
12.242116142093641,
9.508748977118197,
7.82973307939813,
7.122821698405513,
7.00314043464742,
7.000002111777238,
7.000000000000956

Quadruple Roots - Code

```
def quadruple_root_approximation(number, tolerance=1e-6):
        # Initial guess for the fourth root
        guess = number / 2.0
        # Iterate until the approximation
        # is within the specified tolerance
        while abs(guess**4 - number) > tolerance:
                # Update the guess using the approximation formula
                guess = (3 * guess + number / (guess**3)) / 4.0
                print(f" guess = {guess}")
        return guess

# Example: Calculate the fourth root of 2401
input_number = 2401
result = quadruple_root_approximation(input_number)

# Display the result
print(f"The fourth root of {input_number}")
print(f" is approximately: {result}")
```

Quadratic Roots – The Problem Defined

To Solve: $x^2 + 3x \; - 4 = 0 \quad (1)$

Want to have roots

$$x_1 = ?\, and\ x_2 = ?$$

Quadratic Root Calculation

Quadratic Equation:

$$\mathrm{a}x^2 + bx + c = 0 \quad (2)$$

Quadratic Formula

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \quad (3)$$

Special Note

Note the $x_{1,2}$ to imply that there are two solutions (i.e., x_1 and x_2) to find for a second-degree equation as observed from the x^2.

Programmed Solution

```
def calc_quad_eqn_roots(
a: float, b: float, c: float) -> float:
        """Calculate roots of quadratic equation."""
        D = (b * b - 4 * a * c) ** 0.5
        x_one = (-b + D) / (2 * a)
        x_two = (-b - D) / (2 * a)
        return x_one, x_two

print(f"{calc_quad_eqn_roots(1,2,1)}")
```

- Three floating-point inputs: a, b, and c
- Two floating-point outputs: x_{one} and x_{two}
- How does it calculate the roots of a quadratic equation?