Experiment No.: 01
Experiment Name: Introduction to Python and Error Analysis.

Theory:
Python stands out as a widely used high-level language in areas such as engineering, scientific research, and computational mathematics. Its clear syntax and rich ecosystem of libraries—most notably NumPy, SciPy, and Matplotlib—make it especially useful for numerical problem-solving.

Error analysis is a central aspect of numerical methods, as it helps judge both the correctness and reliability of results. Since computers typically deliver approximations instead of exact solutions, estimating the deviation from the true value is necessary.

Types of Errors:

- Absolute Error (AE):

The difference between the true value and the approximate value.

$$AE = |\text{True Value} - \text{Approximate Value}|$$

- Relative Error (RE):

Measures the error in relation to the size of the true value.

$$RE = \frac{AE}{True\ Value}$$

- Percentage Error (PE):

Represents the relative error as a percentage.

$$PE = RE \times 100$$

These measures help in evaluating how accurate an approximation is and guide improvements in numerical techniques.

Program 1: Programming Code

```
1.  true_value = float(input("Enter the true value: "))
2.  approx_value = float(input("Enter the approximate value: "))
3.
4.  abs_error = abs(true_value- approx_value)
5.  rel_error = abs_error / true_value
6.  perc_error = rel_error * 100
7.
8.  print(f"Absolute Error: {abs_error:.3f}")
9.  print(f"Relative Error: {rel_error:.3f}")
10. print(f"Percentage Error: {perc_error:.3f}%")
```

Output:

```
Enter the true value: 88.6
Enter the approximate value: 87.96
Absolute Error: 0.640
Relative Error: 0.007
Percentage Error: 0.722%
```

Discussion & Conclusion:

During this lab session, we practiced using Python to compare approximate results with their exact values. We focused on three error types—absolute error, relative error, and percentage error—to evaluate accuracy from different perspectives. Through these exercises, we not only gained basic experience with Python programming but also developed an understanding of error analysis. This highlighted the critical role of checking accuracy when applying numerical methods to practical situations.

Experiment No.: 02
Experiment Name: Implementation of Bisection Method for Solving Non-Linear Equation.

Theory:
The Bisection Method is a root-finding technique that relies on the Intermediate Value Theorem, which states that if a continuous function changes sign over an interval [xl, xu], then at least one real root lies within that interval. The method works by halving the interval repeatedly and checking where the sign change occurs, thus narrowing down the location of the root. At each step, the midpoint is calculated using:

$$Xr = \frac{xl + xu}{2}$$

The iterations continue until the approximate error satisfies the required tolerance or the iteration limit is reached. This method is known for its simplicity, robustness, and guaranteed convergence under suitable initial conditions.

Program2: Programming

```
1.  import pandas as pd
2.
3.  def f(x, equation):
4.      return eval(equation)
5.
6.  equation = input("Enter the equation in terms of x (e.g., x**3 + 4*x**2 - 1): ")
7.  a = float(input("Enter the lower bound (a): "))
8.  b = float(input("Enter the upper bound (b): "))
9.  tol = float(input("Enter the tolerance: "))
10.
11. fa = f(a, equation)
12. fb = f(b, equation)
13.
14. if fa * fb > 0:
15.     print("Bisection method fails. f(a) and f(b) should have opposite signs.")
16. else:
17.     data = []
18.     prev_c = None
19.     iteration = 1
20.
21.     while True:
22.         c = (a + b) / 2
23.         fc = f(c, equation)
24.
25.         if prev_c is not None:
26.             approx_error = abs((c - prev_c) / c) * 100
27.         else:
28.             approx_error = None
29.
30.         data.append([iteration, a, b, c, fa, fb, fc, approx_error])
31.
32.         if approx_error is not None and approx_error < tol:
33.             break
34.
```

```
        prev_c = c

        if fa * fc < 0:
            b = c
            fb = fc
        else:
            a = c
            fa = fc

        iteration += 1

    df = pd.DataFrame(data, columns=["Iter", "a", "b", "x_c", "f(a)", "f(b)", "f(x_c)", "Approx. Error (%)"])
    print("\nBisection Method Iteration Table (Stopping when error < 5%):")
    print(df.to_string(index=False))

    print(f"\nRoot found: x_c = {c}, f(x_c) = {fc}")
    print(f"Approximate relative error: {approx_error:.2f}%")
```

Output:

```
Enter the equation in terms of x (e.g., x**3 + 4*x**2 - 1): 0.5*x**3 - x**2
Enter the lower bound (a): 1
Enter the upper bound (b): 3
Enter the tolerance: 5

Bisection Method Iteration Table (Stopping when error < 5%):
 Iter    a   b   x_c      f(a)  f(b)   f(x_c)  Approx. Error (%)
    1 1.00 3.0 2.000 -0.500000   4.5 0.000000               NaN
    2 2.00 3.0 2.500  0.000000   4.5 1.562500         20.000000
    3 2.50 3.0 2.750  1.562500   4.5 2.835938          9.090909
    4 2.75 3.0 2.875  2.835938   4.5 3.616211          4.347826

Root found: x_c = 2.875, f(x_c) = 3.6162109375
Approximate relative error: 4.35%
```

Discussion and Conclusion:

The Bisection Method offers a stable and consistent means of locating real roots of continuous functions. In this experiment, it was applied to the function $f(x)=0.5x^3 - x^2$ over the interval [1,3]. By confirming that the initial guesses satisfied the sign-change condition, the method iteratively halved the interval and successfully converged to a root within the tolerance of 0.05. Although the process is slower than more advanced numerical techniques, its guaranteed convergence and reliability make it a practical choice for solving root-finding problems under suitable conditions.

Experiment No.: 03
Experiment Name: Implementation of False Position Method for Solving Non-Linear Equations

Theory:
The False Position Method, or *Regula Falsi*, is a numerical procedure designed to determine the root of a continuous function in the interval [a,b] provided that f(a)·f(b)<0, ensuring a root exists within the interval. The technique differs from the Bisection Method by replacing the midpoint with a more refined estimate obtained through linear interpolation between the function's values at the endpoints. The root approximation is given by:

$$Xr = \frac{\mathrm{a.f(b)} - \mathrm{b.f(a)}}{\mathrm{a.f(b)} - \mathrm{b.f(a)}}$$

Since this approach leverages the slope of the function between the endpoints, it can achieve faster convergence than the Bisection Method, particularly for functions that behave nearly linearly over the chosen interval.
Program 3: Programming code.

```
1.  import pandas as pd
2.
3.  def f(x, equation):
4.      return eval(equation)
5.
6.  equation = input("Enter the equation in terms of x (e.g., x**3 + 4*x**2 - 1): ")
7.  a = float(input("Enter the lower bound (a): "))
8.  b = float(input("Enter the upper bound (b): "))
9.  tol = float(input("Enter the tolerance: "))
10.
11. fa = f(a, equation)
12. fb = f(b, equation)
13.
14. if fa * fb > 0:
15.     print("Bisection method fails. f(a) and f(b) should have opposite signs.")
16. else:
17.     data = []
18.     prev_c = None
19.     iteration = 1
20.
21.     while True:
22.         c = (a * fb - b * fa) / (fb - fa)
23.         fc = f(c, equation)
24.
25.         if prev_c is not None:
26.             approx_error = abs((c - prev_c) / c) * 100
27.         else:
28.             approx_error = None
29.
30.         data.append([iteration, a, b, c, fa, fb, fc, approx_error])
31.
32.         if approx_error is not None and approx_error < tol:
33.             break
34.
35.         prev_c = c
36.
37.         if fa * fc < 0:
```

```
            b = c
            fb = fc
        else:
            a = c
            fa = fc

        iteration += 1

    df = pd.DataFrame(data, columns=["Iter", "a", "b", "x_c", "f(a)", "f(b)", "f(x_c)",
"Approx. Error (%)"])
    print("\Falsi Method Iteration Table (Stopping when error < 5%):")
    print(df.to_string(index=False))

    print(f"\nRoot found: x_c = {c}, f(x_c) = {fc}")
    print(f"Approximate relative error: {approx_error:.2f}%")
```

Output:

```
Enter the equation in terms of x (e.g., x**3 + 4*x**2 - 1): 0.5*x**3-x**2
Enter the lower bound (a): 1
Enter the upper bound (b): 3
Enter the tolerance: 5
\Falsi Method Iteration Table (Stopping when error < 5%):
 Iter        a   b      x_c      f(a)  f(b)    f(x_c)  Approx. Error (%)
    1 1.000000 3.0 1.200000 -0.500000   4.5 -0.576000               NaN
    2 1.200000 3.0 1.404255 -0.576000   4.5 -0.587384         14.545455
    3 1.404255 3.0 1.588498 -0.587384   4.5 -0.519177         11.598569
    4 1.588498 3.0 1.734502 -0.519177   4.5 -0.399375          8.417617
    5 1.734502 3.0 1.837660 -0.399375   4.5 -0.274111          5.613534
    6 1.837660 3.0 1.904397 -0.274111   4.5 -0.173363          3.504367

Root found: x_c = 1.9043968582752038, f(x_c) = -0.17336326651373568
Approximate relative error: 3.50%
```

Discussion & Conclusion:

In this experiment, the False Position Method was applied to the function $f(x)=0.5x^3-x^2$ with initial guesses a=1, b=3, which satisfy the sign-change condition. Using linear interpolation, the method refined the estimate of the root at each step and successfully converged. Compared with the Bisection Method, it can provide faster convergence for functions that behave nearly linearly in the interval, though its performance may decrease if one endpoint does not shift during iterations. Despite this limitation, the method remains simple, effective, and guarantees convergence when the initial interval is valid.