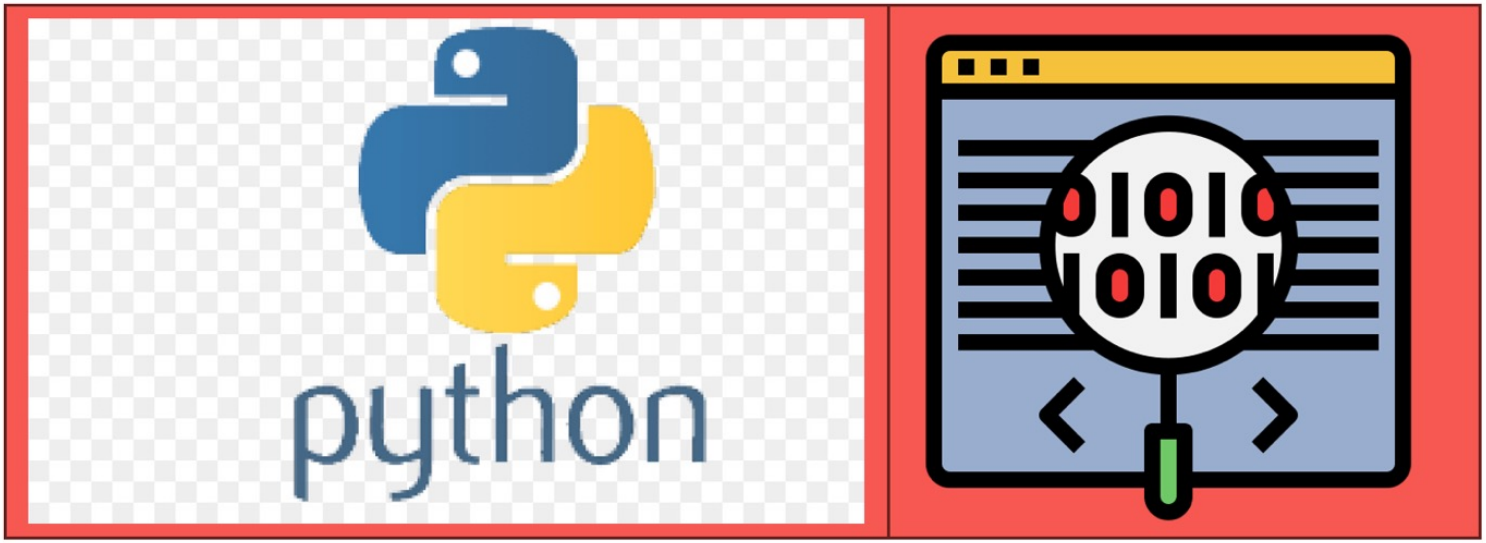
PYTHON MODULE FOR NUMERICAL METHODS



Prepared By:
Dr. Nor Azizah Ali
nzah@utm.my|10-03-2024

Introduction

- Python is a high-level programming language known for its simplicity and readability.
- It is widely used in various fields such as web development, data analysis, artificial intelligence, scientific computing, and more.
- Python's syntax allows programmers to write clear and concise code, making it a popular choice for beginners and experienced developers alike.
- Additionally, Python has a vast ecosystem of libraries and frameworks that extend its capabilities for different purposes.

Python Library

- A Python library is a collection of pre-written code and functionalities that aim to solve specific problems or perform certain tasks.
- These libraries are designed to be reusable and can be imported into Python scripts or projects to extend their capabilities without having to write everything from scratch.
- Python libraries can range from simple utilities to complex frameworks, covering a wide range of domains such as web development, data analysis, machine learning, and more.

Python Library

- **NumPy**, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.
- **SciPy** is a collection of packages addressing several different standard problem domains in scientific computing.
- Together NumPy and SciPy form a reasonably complete and mature computational foundation for many traditional scientific computing applications.

Executing Python Code

Google Colab: https://colab.research.google.com/?utm_source=scs-index

- With google account

Jupyter Notebook via Conda or pip:
https://www.youtube.com/watch?v=AuTkAWEa06E

- To install Python and Jupyter notebook

Others Python IDE (Integrated Development Environment):

1. Pydev
2. Pycharm
3. Visual Studio Code
4. Vim
5. Spyder

System of Linear Equations

Example: Use *numpy.linalg.solve* to solve the following equations.

$$2x_1 + x_2 + x_3 = 7$$
$$3x_1 + 2x_2 - x_3 = 4$$
$$x_1 - 4x_2 + 2x_2 = -1$$

Code

```
import numpy as np
A = np.array ([[2, 1, 1], [3, 2, -1], [1, -4, 2]])
b = np.array ([7, 4, -1])
x = np.linalg.solve (A,b)
print("x1 = ",round(x[0],2),";","x2 = ",
      round(x[1],2),";","x3 =",round(x[2],2))
```

Result

```
x1 = 1.0 ; x2 = 2.0 ; x3 = 3.0
```

System of Linear Equations

Exercise: Use *numpy.linalg.solve* to solve the following equations.

$$
\begin{aligned}
4x_1 + 3x_2 - 5x_3 &= 2 \\
-2x_1 - 4x_2 + 5x_3 &= 5 \\
8x_1 + 8x_2 &= -3
\end{aligned}
$$

Code

Result

NON-LINEAR EQUATIONS:
Bisection Method

Example:

Suppose we have $f(x) = x^3 - 3x^2 + 8x - 5$, the root is obtained using bisection method that lies in the interval $[0,1]$ and approximates to two decimal places of accuracy.

Code

```
# Defining Function
def f(x):
    return x**3-3*x**2+8*x-5

# Implementing Bisection Method
def bisection(x0,x1,e):
    step = 1
    print('\n\n*** BISECTION METHOD IMPLEMENTATION ***')
    condition = True
    while condition:
        x2 = (x0 + x1)/2
        print('Iteration-%d, x2 = %0.6f and f(x2) = %0.6f' % (step,
              x2, f(x2)))

        if f(x0) * f(x2) < 0:
           x1 = x2
        else:
           x0 = x2
        step = step + 1
        condition = abs(f(x2)) > e

    print('\nRequired Root is : %0.3f' % x2)
```

Code (cont'd)

```
# Input Section
x0 = input('First Guess: ')
x1 = input('Second Guess: ')
e = input('Tolerable Error: ')

# Converting input to float
x0 = float(x0)
x1 = float(x1)
e = float(e)

# Checking Correctness of initial guess values and bisecting
if f(x0) * f(x1) > 0.0:
   print('Given guess values do not bracket the root.')
   print('Try Again with different guess values.')
else:
   bisection(x0,x1,e)
```

Result

```
First Guess: 0
Second Guess: 1
Tolerable Error: 0.005


*** BISECTION METHOD IMPLEMENTATION ***
Iteration-1, x2 = 0.500000 and f(x2) = -1.625000
Iteration-2, x2 = 0.750000 and f(x2) = -0.265625
Iteration-3, x2 = 0.875000 and f(x2) = 0.373047
Iteration-4, x2 = 0.812500 and f(x2) = 0.055908
Iteration-5, x2 = 0.781250 and f(x2) = -0.104218
Iteration-6, x2 = 0.796875 and f(x2) = -0.024006
Iteration-7, x2 = 0.804688 and f(x2) = 0.015987
Iteration-8, x2 = 0.800781 and f(x2) = -0.004000

Required Root is : 0.801
```

NON-LINEAR EQUATIONS:
Secant Method

Example:

Solve this function using the secant method.

$$f(x) = \sin(x) + 3x - e^x$$

If the initial guess are $x_0 = 1$ and $x_1 = 0$. Use $\varepsilon = 0.0005$

Code

```
import numpy as np

# Defining Function
def f(x):
return np.sin(x) + 3*x - np.exp(x)

# Implementing Secant Method

def secant(x0,x1,e,N):
    print('\n\n*** SECANT METHOD IMPLEMENTATION ***')
    step = 1
    condition = True
    while condition:
       if f(x0) == f(x1):
          print('Divide by zero error!')
          break
       x2 = x0 - (x1-x0)*f(x0)/(f(x1)- f(x0))
       print('Iteration-%d, x2 = %0.6f and f(x2) = %0.6f'%(step, x2,
             f(x2)))
       x0 = x1
       x1 = x2
       step = step + 1
       condition = abs(f(x2)) > e
    print('\n Required root is: %0.5f' % x2)
```

Code (cont'd)

```
# Input Section
x0 = input('Enter First Guess: ')
x1 = input('Enter Second Guess: ')
e = input('Tolerable Error: ')

# Converting x0 and e to float
x0 = float(x0)
x1 = float(x1)
e = float(e)


# Starting Secant Method
secant(x0,x1,e,N)
```

Result

```
Enter First Guess: 1
Enter Second Guess: 0
Tolerable Error: 0.0005


*** SECANT METHOD IMPLEMENTATION ***
Iteration-1, x2 = 0.470990 and f(x2) = 0.265159
Iteration-2, x2 = 0.372277 and f(x2) = 0.029534
Iteration-3, x2 = 0.359904 and f(x2) = -0.001295
Iteration-4, x2 = 0.360424 and f(x2) = 0.000006

 Required root is: 0.36042
```

NON-LINEAR EQUATIONS:
Newton Method

Example:

Use the Newton's Method to estimate the root of $f(x) = x^3 - \sin x$ employing an initial guess $x_0 = 1$. Do calculations in 5 decimal points and obtain a solution accurate to 3 decimal places.

$$f(x) = x^3 - \sin x$$

For calculation in 5 decimals points, $\varepsilon = 0.00005$.

Code

```
import numpy as np

# Defining Function
def f(x):
return x**3 - np.sin(x)

# Defining derivative of function
def g(x):
return 3*x**2 - np.cos(x)

# Implementing Newton Raphson Method

def newtonRaphson(x0,e,N):
    print('\n\n*** NEWTON RAPHSON METHOD IMPLEMENTATION ***')
    step = 1
    flag = 1
    condition = True
    while condition:
       if g(x0) == 0.0:
          print('Divide by zero error!')
          break
     x1 = x0 - f(x0)/g(x0)
     print('Iteration-%d, x1 = %0.6f and f(x1) = %0.6f' % (step, x1, f(x1)))
     x0 = x1
     step = step + 1
     condition = abs(f(x1)) > e
     print('\nRequired root is: %0.3f' % x1)
```

Code (cont'd)

```
# Input Section
x0 = input('Enter Guess: ')
e = input('Tolerable Error: ')

# Converting x0 and e to float
x0 = float(x0)
e = float(e)


# Starting Newton Raphson Method
newtonRaphson(x0,e,N)
```

Result

```
Enter Guess: 1
Tolerable Error: 0.0005


*** NEWTON RAPHSON METHOD IMPLEMENTATION ***
Iteration-1, x1 = 0.935549 and f(x1) = 0.013917
Iteration-2, x1 = 0.928702 and f(x1) = 0.000150

Required root is: 0.929
```

EIGEN VALUE AND EIGEN VECTOR:
The Power Method

Example:

Let,

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 1 \\ 4 & -4 & 5 \end{bmatrix}$$

Use the Power method to approximate the most dominant eigen value of the matrix. Let $v^{(0)}=(0,0,1)^T$ and iterate until 8 iteration.

Code

```
import numpy as np

def normalize (x):
fac = abs(x).max()
x_n = x/x.max()
return fac, x_n

x = np.array ([0,0,1])
A = np.array ([[1,2,-1], [1,0,1],[4,-4,5]])

for i in range(8):
x = np.dot(A,x)
lambda_1, x = normalize(x)

print ("Eigen-value: ", round(lambda_1,2))
print ("Eigen-vector: ", x)
```

Result

```
Eigen-value: 3.0
Eigen-vector: [-0.24998095 0.24998095 1. ]
```

EIGEN VALUE AND EIGEN VECTOR:
The Shifted Power Method

Example:

Let,

$$A = \begin{bmatrix} 1 & 2 & -1 \\ 1 & 0 & 1 \\ 4 & -4 & 5 \end{bmatrix}$$

Use the shifted power method to approximate the smallest eigen value of the matrix. Use $v^{(0)} = [0, 1, 0]^T$. Iterate until 8 iteration.

Code

```
from numpy.linalg import inv

x = np.array ([0,1,0])
A = np.array ([[1,2,-1], [1,0,1],[4, -4,5]])

A_inv = inv(A)

for i in range (8):
x = np.dot(A_inv, x)
lambda_3, x = normalize(x)

print ("Eigen-value: ", round(lambda_3,2))
print ("Eigen-vector: ", x)
```

Result

```
Eigen-value: 1.0
Eigen-vector: [-0.5 0.50098039 1. ]
```

INTERPOLATION AND APPROXIMATION:
Newton's Polynomial

Example:

k	0	1	2	3	4	5
x_k	1.0	1.2	1.4	1.6	1.8	2.0
y_k	0.0000	0.1823	0.3365	0.4700	0.5878	0.6931

Given the data above, find:

y(1.1) = ??

y(1.5) = ??

Code

```
import numpy as np
def divided_diff(x,y):
#function to calculate the divided differences table
n = len(y)
coef = np.zeros ([n,n])
# the first column is y
coef[:,0] = y

for j in range (1,n):
  for i in range(n-j):
    coef[i][j] = \
   (coef[i+1][j-1] - coef[i][j-1]) / (x[i+j]-x[i])
  return coef

def newton_poly(coef, x_data, x):
#evaluate the newton polynomial at x
  n = len (x_data) - 1
  p = coef[n]
  for k in range(1, n+1):
    p = coef[n-k] + (x-x_data[n-k])*p
  return p
```

Code (cont'd)

```
#Add the input data (x,y)
x = [1.0, 1.2, 1.4, 1.6, 1.8, 2.0]
y = [0.5, 0.4545, 0.4167, 0.3846, 0.3571, 0.3333]

#get the divided difference coef
a_s = divided_diff(x, y)[0,:]

#evaluate on new data points
x_new = 1.1
y_new = newton_poly(a_s, x, x_new)

print("x = ", x_new)
print("y_hat = ", round(y_new, 4))
```

Result

```
x = 1.1
y_hat = 0.4761
```

INTERPOLATION AND APPROXIMATION:
Lagrange Polynomial Interpolation

Example:

Given,

k	0	1	2	3	4
x_k	1.0	1.6	2.5	3.0	3.2
y_k	0.5000	0.3846	0.2857	0.2500	0.2381

Determine the approximation value of y(1.3).

Code

```
from scipy.interpolate import lagrange

#Add the input data (x,y)
x = [1.0, 1.6, 2.5, 3.0, 3.2]
y = [0.5, 0.3846, 0.2857, 0.25, 0.2381]

#evaluate on new data points
f = lagrange(x,y)
x_new2 = 1.3
y_new2 = f(x_new2)
print("x = ", x_new2)
print("y_hat = ", round(y_new2, 4))
```

Result

```
x = 1.3
y_hat = 0.4353
```

INTERPOLATION AND APPROXIMATION:
Least Square Approximation (Linear Function)

Example:

Given,

k	0	1	2	3	4
x_k	1	3	4	5	8
f_k	5	9	11	13	19

Determine the approximation value of f(4.5).

Code

```
from scipy import optimize

#Add the input data (x,y)
x = [1.0, 3.0, 4.0, 5.0, 8.0]
y = [5.0, 9.0, 11.0, 13.0, 19.0]

def func(x, a, b):
  y = a*x+b
  return y

alpha = optimize.curve_fit(func, xdata = x, ydata = y)[0]
a0 = round (alpha[1],2)
a1 = round (alpha[0],2)
print ("a0 = ",a0 )
print ("a1 = ",a1)
x_new3 = 4.5
estimate_f = a0 + a1*(x_new3)
print("x = ",x_new3)
print("f_estimate = ",estimate_f)
```

Result

```
a0 = 3.0
a1 = 2.0
x = 4.5
f_estimate = 12.0
```

INTERPOLATION AND APPROXIMATION:
Least Square Approximation (Non-linear Function)

Example:

Find a polynomial with degree two which suits the following data.

k	0	1	2	3
x_k	1	2	4	6
f_k	10	5	2	1

Code

```
#We can also use polynomial and least squares to fit a nonlinear
function.
#Use numpy.polyfit to obtain the coefficients of different
#order polynomials with the least squares.
import numpy as np

#Add the input data (x,y)
x_d = np.array([1.0, 2.0, 4.0, 6.0])
y_d = np.array([10.0, 5.0, 2.0, 1.0])

for i in range(1,7):
#get the polynomial coefficients; i = degree of polynomial
#General code: y_est = np.polyfit(x_d, y_d, i)
#Example; i = 2 (polynomial with degree 2)
    y_est = np.polyfit(x_d, y_d, 2)
a0 = round(y_est[2],3)
a1 = round(y_est[1],3)
a2 = round(y_est[0],3)
print ("a0 = ",a0)
print ("a1 = ",a1)
print ("a2 = ",a2)
```

Result

```
a0 = 14.312
a1 = -5.266
a2 = 0.513
```

NUMERICAL DIFFERENTIATION:
First Derivative
(Forward Difference Formula)

Example:

Given the following data:

x	1.00	1.05	1.10	1.15	1.20
$f(x)$	1.00000	1.02470	1.04881	1.07238	1.09545

Use forward difference formulas to estimate the following:

$f'(1.0)$
$f'(1.05)$
$f'(1.10)$
$f'(1.15)$.

Code

```
import numpy as np

#step size
h = 0.05
#define grid
x = np.array([1.0, 1.05, 1.10, 1.15, 1.20])
y = np.array([1.0, 1.02470, 1.04881, 1.07238, 1.09545])

#compute vector of forward differences
forward_diff = np.diff(y)/h
#compute corresponding grid
x_diff = x[:-1:]

#Result for:
print("f'(1.0) = ", round(forward_diff[0],4))
print("f'(1.05) = ", round(forward_diff[1],4))
print("f'(1.10) = ", round(forward_diff[2],4))
print("f'(1.15) = ", round(forward_diff[3],4))

#WARNING! When using the command np.diff, the size of the output is one
#less than the size of the input since it needs two arguments to
#produce a difference.
```

Result

```
f'(1.0) = 0.494
f'(1.05) = 0.4822
f'(1.10) = 0.4714
f'(1.15) = 0.4614
```

NUMERICAL DIFFERENTIATION:
Second Derivative
(Central Differences Formula)

Example:

Given the following data:

x	1.00	1.05	1.10	1.15	1.20
$f(x)$	1.00000	1.02470	1.04881	1.07238	1.09545

Estimate the value of $f''(1.10)$

Code

```
import numpy as np

def second_derivative(data_x, data_y):
    n = len(data_x)
    second_derivatives = np.zeros(n)
for i in range(n):
  if i == 0:
  #Forward difference for the first point
    second_derivatives[i]=(2*data_y[i]-5*data_y[i+1]+4*data_y[i+2]-
                           data_y[i+3])/((data_x[i+3]-data_x[i])**2)
  elif i == 1:
  #Forward difference for the second point
     second_derivatives[i]=(data_y[i-1]-2*data_y[i]+data_y[i+1])/
          ((data_x[i] - data_x[i-1]) * (data_x[i+1] - data_x[i]))
  elif i == n-2:
  #Backward difference for the second last point
     second_derivatives[i] = (data_y[i-1]-2*data_y[i] + data_y[i+1])/
         ((data_x[i] - data_x[i-1]) * (data_x[i+1] - data_x[i]))
  elif i == n-1:
  #Backward difference for the last point
     second_derivatives[i]=(data_y[i]-2*data_y[i-1]+ data_y[i-2])/
          ((data_x[i] - data_x[i-3])**2)
  else:
  #Central difference for all other points
     second_derivatives[i]=(data_y[i-1]-2*data_y[i]+data_y[i+1])/
          ((data_x[i]-data_x[i-1])*(data_x[i+1] - data_x[i]))
return second_derivatives
```

Code
(cont'd)

```
# Input data
data_x = np.array([1.0, 1.05, 1.10, 1.15, 1.20])
data_y = np.array([1.0, 1.02470, 1.04881, 1.07238, 1.09545])

# Calculate the second derivatives
second_derivatives = second_derivative(data_x, data_y)

# Print the second derivatives for f"(1.10)
print("f''(1.10)= ", round(second_derivatives[2],5))
```

Result

```
f''(1.10)= -0.216
```

NUMERICAL INTEGRATION:
TRAPEZOID RULE

Example:

Approximate the following integral using the **Trapezoidal rule** with h = 0.5 and h = 0.25.

$$\int_1^4 \frac{x}{\sqrt{x+4}}\, dx$$

Code

```
#Trapezoidal Method
import numpy as np

#Define function to integrate
def f(x):
    return x / np.sqrt(x + 4)

#Implementing trapezoidal method
def trapezoidal(x0,xn,n):
  #calculating step size
  h = (xn - x0) / n
  #Finding sum
  integration = f(x0) + f(xn)
  for i in range(1,n):
    k = x0 + i*h
    integration = integration + 2 * f(k)
  #Finding final integration value
  integration = integration * h/2
  return integration
```

Code (cont'd)

```
# Input section
lower_limit = float(input("Enter lower limit of integration: "))
upper_limit = float(input("Enter upper limit of integration: "))
sub_interval = int(input("Enter number of sub intervals: "))

# Call trapezoidal() method and get result
result = trapezoidal(lower_limit, upper_limit, sub_interval)
print("Integration result by Trapezoidal method is: %0.4f" %(result))
```

Result

```
Enter lower limit of integration: 1
Enter upper limit of integration: 4
Enter number of sub intervals: 6
Integration result by Trapezoidal method is: 2.8897
```

NUMERICAL INTEGRATION:
SIMPSON'S 1/3 RULE

Example:

Approximate the following integral using the **Simpson's rule** with h = 0.5 and h = 0.25.

$$\int_1^4 \frac{x}{\sqrt{x+4}}\,dx$$

Code

```
#Simpson's 1/3 Method
import numpy as np

#Define function to integrate
def f(x):
   return x / np.sqrt(x + 4)

#Implementing Simpson's 1/3
def simpson13(x0, xn, n):
  #calculating step size
   h = (xn - x0) / n

  #finding sum
   integration = f(x0) + f(xn)

   for i in range(1, n):
      k = x0 + i * h
      if i % 2 == 0:
        integration = integration + 2 * f(k)
      else:
        integration = integration + 4 * f(k)

   #finding final integration value
    integration *= h / 3
    return integration
```

Code (cont'd)

```
# Input section
Lower_limit = float(input("Enter lower limit of
integration: "))
Upper_limit = float(input("Enter upper limit of
integration: "))
sub_interval = int(input("Enter number of sub intervals:
"))

# Call Simpson13() method to get result
result = simpson13(Lower_limit, Upper_limit, sub_interval)
print("Integration result by Simpson's 1/3 method is:
%0.4f" % result)
```

Result

```
Enter lower limit of integration: 1
Enter upper limit of integration: 4
Enter number of sub intervals: 12
Integration result by Simpson's 1/3 method is: 2.8925
```

NUMERICAL INTEGRATION:
SIMPSON'S 3/8 RULE

Example:

Approximate the following integral using the **Simpson's rule** with $h = 0.25$.

$$\int_1^4 \frac{x}{\sqrt{x+4}}\, dx$$

Code

```
#Simpson's 3/8 Rule
import numpy as np
#Define function to integrate
def f(x):
  return x / np.sqrt(x + 4)

#Implementing Simpson's 3/8
def simpson38(x0,xn,n):
  #calculating step size
  h = (xn - x0) / n
  #Finding sum
  integration = f(x0) + f(xn)
  for i in range(1,n):
     k = x0 + i*h
     if i%3 == 0:
       integration = integration + 2 * f(k)
     else:
       integration = integration + 3 * f(k)
   #Finding final integration value
    integration = integration * 3 * h / 8
   return integration
```

Code (cont'd)

```
# Input section
lower_limit = float(input("Enter lower limit of
integration: "))
upper_limit = float(input("Enter upper limit of
integration: "))
sub_interval = int(input("Enter number of sub intervals:
"))

# Call trapezoidal() method and get result
result = simpson38(lower_limit, upper_limit, sub_interval)
print("Integration result by Simpson's 3/8 method is:
%0.4f" % (result) )
```

Result

```
Enter lower limit of integration: 1
Enter upper limit of integration: 4
Enter number of sub intervals: 12
Integration result by Simpson's 3/8 method is: 2.8925
```

NUMERICAL INTEGRATION:
ROMBERG INTEGRATION

Example:

Use Romberg integration to approximate

$$\int_1^4 \frac{x}{\sqrt{x+4}}\,dx$$

Code

```
import numpy as np

#Romberg integration method to integrate a function from a to b.

def romberg_integration(func, a, b, n):

#Initialize the Romberg table with zeros
R = np.zeros((n, n))

# Calculate the first column of the Romberg table using trapezoidal rule
h = b - a
R[0, 0] = 0.5 * h * (func(a) + func(b))

for i in range(1, n):
    h /= 2
    sum_f = 0
    for k in range(1, 2**i, 2):
       sum_f += func(a + k * h)
    R[i,0] = 0.5 * R[i-1,0] + sum_f*h

# Populate the rest of the Romberg table
for j in range(1,n):
    for i in range(j, n):
       R[i,j] = (4**j*R[i, j-1] - R[i-1, j-1])/(4**j-1)
```

Code (cont'd)

```
# Return the integral approximation
return R[n - 1, n - 1]

# Example usage:
def f(x):
    return x / np.sqrt(x + 4) # Function to integrate

a = 1 # Lower limit
b = 4 # Upper limit
n = 5 # Number of iterations

result = romberg_integration(f, a, b, n)
print("Approximate integral using Romberg method is: %0.4f"
      %(result))
```

Result

```
Approximate integral using Romberg method is: 2.8925
```