Python Functions

Create functions with different numbers of parameters and return types.

1. Function with No Parameters

In [1]:

```
def greet():
    return "Good Morning!"

print(greet())
```

```
Good Morning!
```

2. Function with One Parameter

In [3]:

```
def square(number):
    return number ** 2

result = square(6)
print("Square of 6:", result)
```

```
Square of 6: 36
```

3. Function with Two Parameters

In [5]:

```
def add(a, b):
    return a + b

sum_result = add(3, 4)
print("Sum of 3 and 4:", sum_result)
```

```
Sum of 3 and 4: 7
```

4. Function with Default Parameters

In [7]:

```
def greet_user(name, greeting="Hi"):
    return f"{greeting}, {name}!"

print(greet_user("Alice"))
print(greet_user("Bob", "Hello"))
```

```
Hi, Alice!
Hello, Bob!
```

5. Function with Variable Number of Arguments

In [11]:

```
def concatenate(*args):
    return " ".join(args)

result = concatenate("Hello", "this", "is", "a", "test.")
print("Concatenated String:", result)
```

```
Concatenated String: Hello this is a test.
```

Explore function scope and variable accessibility.

1. Global vs. Local Scope

In [15]:

```python
x = 10

def example_function():
    y = 5
    print("Inside function:")
    print("Local y:", y)
    print("Global x:", x)

example_function()

print("Outside function:")
print("Global x:", x)
```

```
Inside function:
Local y: 5
Global x: 10
Outside function:
Global x: 10
```

2. Modifying Global Variables Inside a Function

In [17]:

```python
count = 0

def increment():
    global count
    count += 1
    print("Count inside function:", count)

increment()
print("Count outside function:", count)
```

```
Count inside function: 1
Count outside function: 1
```

3. Nested Functions and Variable Scope

In [19]:

```python
def outer_function():
    outer_var = "I'm from the outer function!"

    def inner_function():
        inner_var = "I'm from the inner function!"
        print(inner_var)
        print(outer_var)

    inner_function()
    # print(inner_var)

outer_function()
```

```
I'm from the inner function!
I'm from the outer function!
```

4. Function Arguments and Local Scope

In [21]:

```python
def multiply(a, b):
    result = a * b
    return result

product = multiply(3, 4)
print("Product:", product)
```

```
Product: 12
```

5. Nonlocal Variables in Nested Functions

In [23]:

```python
def outer_function():
    outer_var = 10

    def inner_function():
        nonlocal outer_var
        outer_var += 5
        print("Inner outer_var:", outer_var)

    inner_function()
    print("Outer outer_var:", outer_var)

outer_function()
```

```
Inner outer_var: 15
Outer outer_var: 15
```

Implement functions with default argument values.

1. Basic Default Argument

In [25]:

```python
def greet(name="Guest"):
    return f"Hello, {name}!"

print(greet())
print(greet("Alice"))
```

```
Hello, Guest!
Hello, Alice!
```

2. Multiple Default Arguments

In [31]:

```python
def describe_pet(animal_type="dog", pet_name="Fido"):
    return f"I have a {animal_type} named {pet_name}."

print(describe_pet())
print(describe_pet("cat"))
print(describe_pet("rabbit", "Bunny"))
```

```
I have a dog named Fido.
I have a cat named Fido.
I have a rabbit named Bunny.
```

3. Default Values and Keyword Arguments

In [33]:

```python
def book_info(title, author, year=2021):
    return f"{title} by {author}, published in {year}."
```

```
print(book_info("1984", "George Orwell"))
print(book_info("The Great Gatsby", "F. Scott Fitzgerald", 1925))
```

```
1984 by George Orwell, published in 2021.
The Great Gatsby by F. Scott Fitzgerald, published in 1925.
```

4. Combining Default and Non-default Arguments

In [35]:

```
def order_coffee(size, type="Regular"):
    return f"You ordered a {size} cup of {type} coffee."

# Calling the function
print(order_coffee("Medium"))
print(order_coffee("Large", "Decaf"))
```

```
You ordered a Medium cup of Regular coffee.
You ordered a Large cup of Decaf coffee.
```

5. Using Default Values in Recursive Functions

In [37]:

```
def factorial(n, result=1):
    if n == 0:
        return result
    else:
        return factorial(n - 1, result * n)

# Calling the function
print(factorial(5))
```

```
120
```

Write recursive functions.

1. Factorial of a Number

In [39]:

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive case

print(factorial(5))
```

```
120
```

2. Fibonacci Sequence

In [43]:

```
def fibonacci(n):
    if n <= 1:  # Base case
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

print(fibonacci(6))
```

```
8
```

3. Sum of a List

In [45]:

```
def sum_list(lst):
    if not lst:
        return 0
    else:
        return lst[0] + sum_list(lst[1:])

print(sum_list([1, 2, 3, 4, 5]))
```

```
15
```

4. Reverse a String

In [47]:

```
def reverse_string(s):
    if len(s) == 0:
        return s
    else:
        return s[-1] + reverse_string(s[:-1])

print(reverse_string("hello"))
```

```
olleh
```

5. Tower of Hanoi

In [ ]:

```
def tower_of_hanoi(n, source, target, auxiliary):
    if n == 1:  # Base case
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n - 1, source, auxiliary, target)  # Move n-1 disks to auxiliary
    print(f"Move disk {n} from {source} to {target}")  # Move the nth disk to targe
    tower_of_hanoi(n - 1, auxiliary, target, source)  # Move n-1 disks from auxilia

tower_of_hanoi(3, 'A', 'C', 'B')
```

Demonstrate how to use docstrings to document functions.

1. Simple Function with a Docstring

In [51]:

```
def add(a, b):
    return a + b

# Calling the function
result = add(3, 5)
print(result)
```

```
8
```

2. Function with Multiple Parameters

In [53]:

```
def describe_pet(animal_type, pet_name):
    return f"I have a {animal_type} named {pet_name}."

description = describe_pet("dog", "Buddy")
print(description)
```

```
I have a dog named Buddy.
```

3. Function with Default Parameters

In [55]:

```
def greet(name, greeting="Hello"):

    return f"{greeting}, {name}!"

# Calling the function
print(greet("Alice"))
print(greet("Bob", "Hi"))
```

```
Hello, Alice!
Hi, Bob!
```

4. Function with a Return Type

In [59]:

```
def factorial(n):
    if n < 0:
        raise ValueError("Negative numbers do not have a factorial.")
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```

5. Accessing Docstrings

In [61]:

```
print(add.__doc__)
print(factorial.__doc__)
```

```
None
None
```

Lambda Functions

Create simple lambda functions for various operations.

1. Basic Arithmetic Operations

In [63]:

```
add = lambda x, y: x + y
print(add(5, 3))
```

```
8
```

In [67]:

```
subtract = lambda x, y: x - y
print(subtract(10, 4))
```

```
6
```

In [69]:

```
multiply = lambda x, y: x * y
print(multiply(7, 6))
```

```
42
```

In [75]:
```
divide = lambda x, y: x / y if y != 0 else 'Cannot divide by zero'
print(divide(10, 2))
print(divide(10, 0))
```

```
5.0
Cannot divide by zero
```

2. Lambda Functions with ConditionalsMaximum of Two Numbers

In [77]:
```
maximum = lambda a, b: a if a > b else b
print(maximum(4, 7))
```

```
7
```

Check Even or Odd

In [81]:
```
is_even = lambda x: x % 2 == 0
print(is_even(4))
print(is_even(5))
```

```
True
False
```

3. Lambda Functions with map, filter, and reduce Using map to Square a List of Numbers

In [83]:
```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

Using filter to Get Even Numbers

In [85]:
```
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
[2, 4]
```

Using reduce to Sum a List of Numbers

from functools import reduce

sum_of_numbers = reduce(lambda x, y: x + y, numbers) print(sum_of_numbers)

4. Sorting with Lambda FunctionsSort a List of Tuples by Second Element

In [89]:
```
data = [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

```
[(1, 'apple'), (2, 'banana'), (3, 'cherry')]
```

5. Creating a Simple Lambda Function for Concatenation

In [91]:
```
concat = lambda a, b: a + " " + b
print(concat("Hello", "World!"))
```

```
Hello World!
```

Use lambda functions with built-in functions like map, filter, and reduce.

1. Using map()

In [95]:

```
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

2. Using filter()

In [97]:

```
numbers = [1, 2, 3, 4, 5, 6]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)
```

```
[2, 4, 6]
```

3. Using reduce()

In [99]:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)
```

```
15
```

4. Combining map(), filter(), and reduce()

In [101…

```
squared_numbers = list(map(lambda x: x ** 2, numbers))
even_squares = list(filter(lambda x: x % 2 == 0, squared_numbers))
sum_even_squares = reduce(lambda x, y: x + y, even_squares)
print(sum_even_squares)
```

```
20
```

Compare lambda functions with regular functions in terms of syntax and use cases

1. Syntax Comparison

In [105…

```
# regular function
def add(x, y):
    return x + y

print(add(3, 5))
```

```
8
```

In [107…

```
# Lambda Function
add = lambda x, y: x + y

print(add(3, 5))
```

```
8
```

3. Use Cases

In [111…

```
# Regular Function Use Case: Complex Logic
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```

In [113…

```
# Lambda Function Use Case: Simple Operations
points = [(2, 3), (1, 2), (4, 1)]
sorted_points = sorted(points, key=lambda point: point[1])  # Sort by y-coordinate
print(sorted_points)
```

```
[(4, 1), (1, 2), (2, 3)]
```

4. Example of Both in Context

In [115…

```
# Using a Lambda Function with map()
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4]
squared_numbers = list(map(square, numbers))
print(squared_numbers)
```

```
[1, 4, 9, 16]
```

NumPy

Create different types of NumPy arrays (1D, 2D, 3D).

1. Importing NumPy

In [119…

```
import numpy as np
```

2. Creating a 1D Array

In [121…

```
array_1d = np.array([1, 2, 3, 4, 5])
print("1D Array:")
print(array_1d)
```

```
1D Array:
[1 2 3 4 5]
```

3. Creating a 2D Array

In [123…

```
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("\n2D Array:")
print(array_2d)
```

```
2D Array:
[[1 2 3]
 [4 5 6]]
```

4. Creating a 3D Array

In [125…

```
array_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print("\n3D Array:")
print(array_3d)
```

```
3D Array:
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

5. Creating Arrays with Built-in Functions

In [129…

```
# Creating a 1D Array with arange()
array_1d_range = np.arange(10)
print("\n1D Array with arange:")
print(array_1d_range)
```

```
1D Array with arange:
[0 1 2 3 4 5 6 7 8 9]
```

In [131…

```
# Creating a 2D Array with zeros()
array_2d_zeros = np.zeros((3, 4))  # 3 rows and 4 columns
print("\n2D Array of zeros:")
print(array_2d_zeros)
```

```
2D Array of zeros:
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [133…

```
# Creating a 3D Array with ones()
array_3d_ones = np.ones((2, 3, 4))
print("\n3D Array of ones:")
print(array_3d_ones)
```

```
3D Array of ones:
[[[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]

 [[1. 1. 1. 1.]
  [1. 1. 1. 1.]
  [1. 1. 1. 1.]]]
```

Perform basic arithmetic operations on arrays.

1. Importing NumPy

In [137…

```
import numpy as np
```

2. Creating Sample Arrays

In [139…

```
array_a = np.array([1, 2, 3, 4, 5])
array_b = np.array([10, 20, 30, 40, 50])
```

3. Basic Arithmetic Operations

In [141…

```
# Addition
addition = array_a + array_b
```

```
print("Addition:")
print(addition)
```

```
Addition:
[11 22 33 44 55]
```

In [145…

```
# Subtraction
subtraction = array_b - array_a
print("\nSubtraction:")
print(subtraction)
```

```
Subtraction:
[ 9 18 27 36 45]
```

In [147…

```
# Multiplication
multiplication = array_a * array_b
print("\nMultiplication:")
print(multiplication)
```

```
Multiplication:
[ 10  40  90 160 250]
```

In [149…

```
# Division
division = array_b / array_a
print("\nDivision:")
print(division)
```

```
Division:
[10. 10. 10. 10. 10.]
```

4. Operations on 2D Arrays

In [151…

```
# create
array_2d_a = np.array([[1, 2], [3, 4]])
array_2d_b = np.array([[10, 20], [30, 40]])
```

In [153…

```
# Addition
addition_2d = array_2d_a + array_2d_b
print("\n2D Addition:")
print(addition_2d)
```

```
2D Addition:
[[11 22]
 [33 44]]
```

In [155…

```
# Subtraction
subtraction_2d = array_2d_b - array_2d_a
print("\n2D Subtraction:")
print(subtraction_2d)
```

```
2D Subtraction:
[[ 9 18]
 [27 36]]
```

In [157…

```
# Multiplication
multiplication_2d = array_2d_a * array_2d_b
print("\n2D Multiplication:")
print(multiplication_2d)
```

```
2D Multiplication:
[[ 10  40]
 [ 90 160]]
```

In [159…

```
# Division
division_2d = array_2d_b / array_2d_a
print("\n2D Division:")
print(division_2d)
```

```
2D Division:
[[10. 10.]
 [10. 10.]]
```

5. Scalar Operations

In [161…

```
# Scalar addition
scalar_add = array_a + 5
print("\nScalar Addition:")
print(scalar_add)  # Output: [ 6  7  8  9 10]

# Scalar multiplication
scalar_multiply = array_2d_a * 2
print("\nScalar Multiplication:")
print(scalar_multiply)
```

```
Scalar Addition:
[ 6  7  8  9 10]

Scalar Multiplication:
[[2 4]
 [6 8]]
```

Use indexing and slicing to access elements.

1. Importing NumPy

In [165…

```
import numpy as np
```

2. Creating a Sample Array

In [169…

```
array_1d = np.array([10, 20, 30, 40, 50])
print("1D Array:")
print(array_1d)
```

```
1D Array:
[10 20 30 40 50]
```

3. Indexing in 1D Arrays

In [173…

```
first_element = array_1d[0]
print("\nFirst Element:", first_element)

last_element = array_1d[-1]
print("Last Element:", last_element)
```

```
First Element: 10
Last Element: 50
```

4. Slicing in 1D Arrays

In [175…

```
# Slicing from index 1 to 3 (exclusive of 3)
slice_1d = array_1d[1:4]
print("\nSliced Array (from index 1 to 3):", slice_1d)

# Slicing with step
slice_step = array_1d[::2]
print("Sliced Array with Step 2:", slice_step)
```

```
Sliced Array (from index 1 to 3): [20 30 40]
Sliced Array with Step 2: [10 30 50]
```

5. Creating a 2D Array

In [177…

```
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("\n2D Array:")
print(array_2d)
```

```
2D Array:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

6. Indexing in 2D Arrays

In [179…

```
# Accessing the element in the second row and second column
element_2d = array_2d[1, 1]
print("\nElement at (1, 1):", element_2d)

# Accessing the element in the third row and first column
element_3rd_row = array_2d[2, 0]
print("Element at (2, 0):", element_3rd_row)
```

```
Element at (1, 1): 5
Element at (2, 0): 7
```

7. Slicing in 2D Arrays

In [185…

```
# Slicing Rows
slice_rows = array_2d[0:2]
print("\nSliced Rows (first two):")
print(slice_rows)
```

```
Sliced Rows (first two):
[[1 2 3]
 [4 5 6]]
```

In [183…

```
# Slicing the first column
slice_column = array_2d[:, 0]
print("\nSliced Column (first column):", slice_column)
```

```
Sliced Column (first column): [1 4 7]
```

In [187…

```
# Slicing a Submatrix
submatrix = array_2d[0:2, 0:2]
print("\nSliced Submatrix (first two rows and columns):")
print(submatrix)
```

```
Sliced Submatrix (first two rows and columns):
[[1 2]
 [4 5]]
```

Explore array manipulation functions (reshape, transpose, concatenate).

1. Importing NumPy

In [189…

```
import numpy as np
```

2. Creating a Sample Array

In [193…

```
array_1d = np.array([1, 2, 3, 4, 5, 6])
print("Original 1D Array:")
print(array_1d)
```

```
Original 1D Array:
[1 2 3 4 5 6]
```

3. Reshape

In [197…

```
#Reshape to 2D
array_2d = array_1d.reshape((2, 3))
print("\nReshaped to 2D Array (2x3):")
print(array_2d)
```

```
Reshaped to 2D Array (2x3):
[[1 2 3]
 [4 5 6]]
```

In [199…

```
# Reshape to 3D
array_3d = array_1d.reshape((1, 2, 3))
print("\nReshaped to 3D Array (1x2x3):")
print(array_3d)
```

```
Reshaped to 3D Array (1x2x3):
[[[1 2 3]
  [4 5 6]]]
```

4. Transpose

In [201…

```
transposed_array = array_2d.T
print("\nTransposed 2D Array:")
print(transposed_array)
```

```
Transposed 2D Array:
[[1 4]
 [2 5]
 [3 6]]
```

5. Concatenate

In [203…

```
# Concatenate 1D Arrays
array_a = np.array([1, 2, 3])
array_b = np.array([4, 5, 6])

# Concatenating along the first axis (default)
concatenated_1d = np.concatenate((array_a, array_b))
print("\nConcatenated 1D Array:")
print(concatenated_1d)
```

```
Concatenated 1D Array:
[1 2 3 4 5 6]
```

In [205…

```python
# Creating two 2D arrays
array_2d_a = np.array([[1, 2, 3], [4, 5, 6]])
array_2d_b = np.array([[7, 8, 9], [10, 11, 12]])

# Concatenating along rows (axis 0)
concatenated_2d_rows = np.concatenate((array_2d_a, array_2d_b), axis=0)
print("\nConcatenated 2D Array (along rows):")
print(concatenated_2d_rows)

# Concatenating along columns (axis 1)
concatenated_2d_cols = np.concatenate((array_2d_a, array_2d_b), axis=1)
print("\nConcatenated 2D Array (along columns):")
print(concatenated_2d_cols)
```

```
Concatenated 2D Array (along rows):
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]

Concatenated 2D Array (along columns):
[[ 1  2  3  7  8  9]
 [ 4  5  6 10 11 12]]
```

Create and use NumPy random number generators

1. Importing NumPy

In [207…

```python
import numpy as np
```

2. Creating Random Number Generators

In [209…

```python
rng = np.random.default_rng()
```

3. Generating Random Numbers

In [213…

```python
# Generating 5 random floats
random_floats = rng.random(5)
print("Random Floats:")
print(random_floats)
```

```
Random Floats:
[0.18912871 0.36874626 0.37166363 0.99454203 0.63739684]
```

In [215…

```python
# Generating 5 random integers between 0 and 10 (exclusive)
random_integers = rng.integers(low=0, high=10, size=5)
print("\nRandom Integers:")
print(random_integers)
```

```
Random Integers:
[3 5 7 2 6]
```

4. Generating Random Samples from a Normal Distribution

In [217…

```
normal_samples = rng.normal(loc=0.0, scale=1.0, size=5)
print("\nRandom Samples from Normal Distribution:")
print(normal_samples)
```

```
Random Samples from Normal Distribution:
[-1.33402433 -0.24288822 -2.12286651 -1.06145841 -2.01539458]
```

5. Generating Random Samples from a Uniform Distribution

In [219…

```
uniform_samples = rng.uniform(low=1.0, high=10.0, size=5)
print("\nRandom Samples from Uniform Distribution:")
print(uniform_samples)
```

```
Random Samples from Uniform Distribution:
[7.70950261 8.15242449 3.37743971 9.18275703 6.81372799]
```

6. Setting the Seed for Reproducibility

In [221…

```
rng = np.random.default_rng(seed=42)

random_floats_seeded = rng.random(5)
print("\nRandom Floats with Seed:")
print(random_floats_seeded)
```

```
Random Floats with Seed:
[0.77395605 0.43887844 0.85859792 0.69736803 0.09417735]
```

Pandas

Create Pandas Series and DataFrames.

1. Importing Pandas

In [223…

```
import pandas as pd
```

2. Creating a Pandas Series

In [227…

```
# Creating a Series from a list
data = [10, 20, 30, 40, 50]
series = pd.Series(data)
print("Pandas Series:")
print(series)
```

```
Pandas Series:
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

In [229…

```
# Creating a Series with a custom index
index = ['a', 'b', 'c', 'd', 'e']
series_with_index = pd.Series(data, index=index)
print("\nPandas Series with Custom Index:")
print(series_with_index)
```

```
Pandas Series with Custom Index:
a    10
b    20
c    30
d    40
e    50
dtype: int64
```

3. Creating a Pandas DataFrame

In [231…

```
# Creating a DataFrame from a dictionary
data_dict = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data_dict)
print("\nPandas DataFrame from Dictionary:")
print(df)
```

```
Pandas DataFrame from Dictionary:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

In [233…

```
# Creating a DataFrame from a list of lists
data_list = [
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Los Angeles'],
    ['Charlie', 35, 'Chicago']
]
df_from_list = pd.DataFrame(data_list, columns=['Name', 'Age', 'City'])
print("\nPandas DataFrame from List of Lists:")
print(df_from_list)
```

```
Pandas DataFrame from List of Lists:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

4. Accessing Data in Series and DataFrames

In [235…

```
# Accessing an element by index
print("\nElement at index 'b':", series_with_index['b'])
```

```
Element at index 'b': 20
```

In [237…

```
# Accessing a row by index
print("\nRow at index 1:")
print(df.loc[1])
```

```
Row at index 1:
Name            Bob
Age              30
City    Los Angeles
Name: 1, dtype: object
```

5. Additional DataFrame Creation Methods

In [239…

```
# Creating a DataFrame with random data
random_data = pd.DataFrame(np.random.randn(3, 4), columns=['A', 'B', 'C', 'D'])
print("\nPandas DataFrame with Random Data:")
print(random_data)
```

```
Pandas DataFrame with Random Data:
          A         B         C         D
0 -1.325493  0.164963  0.461721 -0.072134
1  0.994245 -0.454357 -0.400205  2.515375
2  1.137822  0.342445 -1.047122  1.340567
```

Load data from various file formats (CSV, Excel, etc.)

1. Importing Pandas

In [241…

```
import pandas as pd
```

2. Loading Data from a CSV File

In [ ]:

```
df_csv = pd.read_csv('data.csv')
print("Data Loaded from CSV:")
print(df_csv)
```

3. Loading Data from an Excel File

In [ ]:

```
df_excel = pd.read_excel('data.xlsx')
print("\nData Loaded from Excel:")
print(df_excel)
```

4. Loading Data from a JSON File

In [ ]:

```
df_json = pd.read_json('data.json')
print("\nData Loaded from JSON:")
print(df_json)
```

5. Loading Data from a SQL Database

In [ ]:

```
import sqlite3

# Create a connection to the SQLite database
conn = sqlite3.connect('example.db')

# Load data from a SQL query
df_sql = pd.read_sql_query('SELECT * FROM your_table_name', conn)
print("\nData Loaded from SQL Database:")
print(df_sql)

# Don't forget to close the connection
conn.close()
```

6. Loading Data from a Text File

In [ ]:

```
df_txt = pd.read_csv('data.txt', delimiter='\t')
print("\nData Loaded from Text File:")
print(df_txt)
```

Perform data cleaning and manipulation tasks.

1. Importing Pandas

In [251…

```
import pandas as pd
```

2. Creating a Sample DataFrame

In [253…

```
data = {
    'Name': ['Alice', 'Bob', None, 'Charlie', 'David', 'Edward'],
    'Age': [25, 30, 35, None, 45, 50],
    'City': ['New York', 'Los Angeles', 'Chicago', None, 'Houston', 'Phoenix'],
    'Salary': [70000, 80000, None, 120000, 90000, 60000]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
      Name   Age         City    Salary
0    Alice  25.0     New York   70000.0
1      Bob  30.0  Los Angeles   80000.0
2     None  35.0      Chicago       NaN
3  Charlie   NaN         None  120000.0
4    David  45.0      Houston   90000.0
5   Edward  50.0      Phoenix   60000.0
```

3. Handling Missing Values

In [257…

```
# Check for missing values
print("\nMissing Values:")
print(df.isnull().sum())
```

```
Missing Values:
Name      1
Age       1
City      1
Salary    1
dtype: int64
```

In [259…

```
# Dropping rows with any missing values
df_dropped = df.dropna()
print("\nDataFrame after Dropping Rows with Missing Values:")
print(df_dropped)
```

```
DataFrame after Dropping Rows with Missing Values:
     Name   Age         City   Salary
0   Alice  25.0     New York  70000.0
1     Bob  30.0  Los Angeles  80000.0
4   David  45.0      Houston  90000.0
5  Edward  50.0      Phoenix  60000.0
```

In [261…

```
# Filling missing values with a specified value
df_filled = df.fillna({'Age': df['Age'].mean(), 'City': 'Unknown', 'Salary': df['Sa
print("\nDataFrame after Filling Missing Values:")
print(df_filled)
```

```
DataFrame after Filling Missing Values:
      Name   Age         City    Salary
0    Alice  25.0     New York   70000.0
1      Bob  30.0  Los Angeles   80000.0
2     None  35.0      Chicago   80000.0
3  Charlie  37.0      Unknown  120000.0
4    David  45.0      Houston   90000.0
5   Edward  50.0      Phoenix   60000.0
```

4. Data Type Conversion

In [263…

```
df['Age'] = df['Age'].fillna(df['Age'].mean()).astype(int)
print("\nDataFrame after Converting Age to Integer:")
print(df)
```

```
DataFrame after Converting Age to Integer:
      Name  Age         City    Salary
0    Alice   25     New York   70000.0
1      Bob   30  Los Angeles   80000.0
2     None   35      Chicago       NaN
3  Charlie   37         None  120000.0
4    David   45      Houston   90000.0
5   Edward   50      Phoenix   60000.0
```

5. Renaming Columnsdf.rename(columns={'City': 'Location', 'Salary': 'Annual Salary'}, inplace=True) print("\nDataFrame after Renaming Columns:") print(df)6. Filtering Rows

In [265…

```
# Filtering rows where Age is greater than 30
filtered_df = df[df['Age'] > 30]
print("\nFiltered DataFrame (Age > 30):")
print(filtered_df)
```

```
Filtered DataFrame (Age > 30):
      Name  Age     City    Salary
2     None   35  Chicago       NaN
3  Charlie   37     None  120000.0
4    David   45  Houston   90000.0
5   Edward   50  Phoenix   60000.0
```

7. Adding New Columns

In [267…

```
# Adding a new column for experience
df['Experience'] = df['Age'] - 22  # Assuming 22 is the starting age for work
print("\nDataFrame after Adding Experience Column:")
print(df)
```

```
DataFrame after Adding Experience Column:
      Name  Age         City    Salary  Experience
0    Alice   25     New York   70000.0           3
1      Bob   30  Los Angeles   80000.0           8
2     None   35      Chicago       NaN          13
3  Charlie   37         None  120000.0          15
4    David   45      Houston   90000.0          23
5   Edward   50      Phoenix   60000.0          28
```

8. Grouping Data

In [ ]:

```
# Grouping by City and calculating the average salary
grouped_df = df.groupby('Location')['Annual Salary'].mean().reset_index()
```

```
print("\nAverage Salary by Location:")
print(grouped_df)
```

Explore data analysis and visualization using Pandas.

1. Importing Required Libraries

In [271…

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

2. Creating a Sample DataFrame

In [273…

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward', 'Fiona'],
    'Age': [25, 30, 35, 40, 45, 50],
    'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Los Angeles', 'Chic
    'Salary': [70000, 80000, 120000, 110000, 90000, 95000]
}

df = pd.DataFrame(data)
print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
      Name  Age         City  Salary
0    Alice   25     New York   70000
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago  120000
3    David   40     New York  110000
4   Edward   45  Los Angeles   90000
5    Fiona   50      Chicago   95000
```

3. Descriptive Statistics

In [275…

```
# Descriptive statistics
print("\nDescriptive Statistics:")
print(df.describe())
```

```
Descriptive Statistics:
             Age         Salary
count   6.000000       6.000000
mean   37.500000   94166.666667
std     9.354143   18551.729479
min    25.000000   70000.000000
25%    31.250000   82500.000000
50%    37.500000   92500.000000
75%    43.750000  106250.000000
max    50.000000  120000.000000
```

4. Grouping Data

In [277…

```
# Grouping by City and calculating average salary
average_salary_by_city = df.groupby('City')['Salary'].mean().reset_index()
print("\nAverage Salary by City:")
print(average_salary_by_city)
```

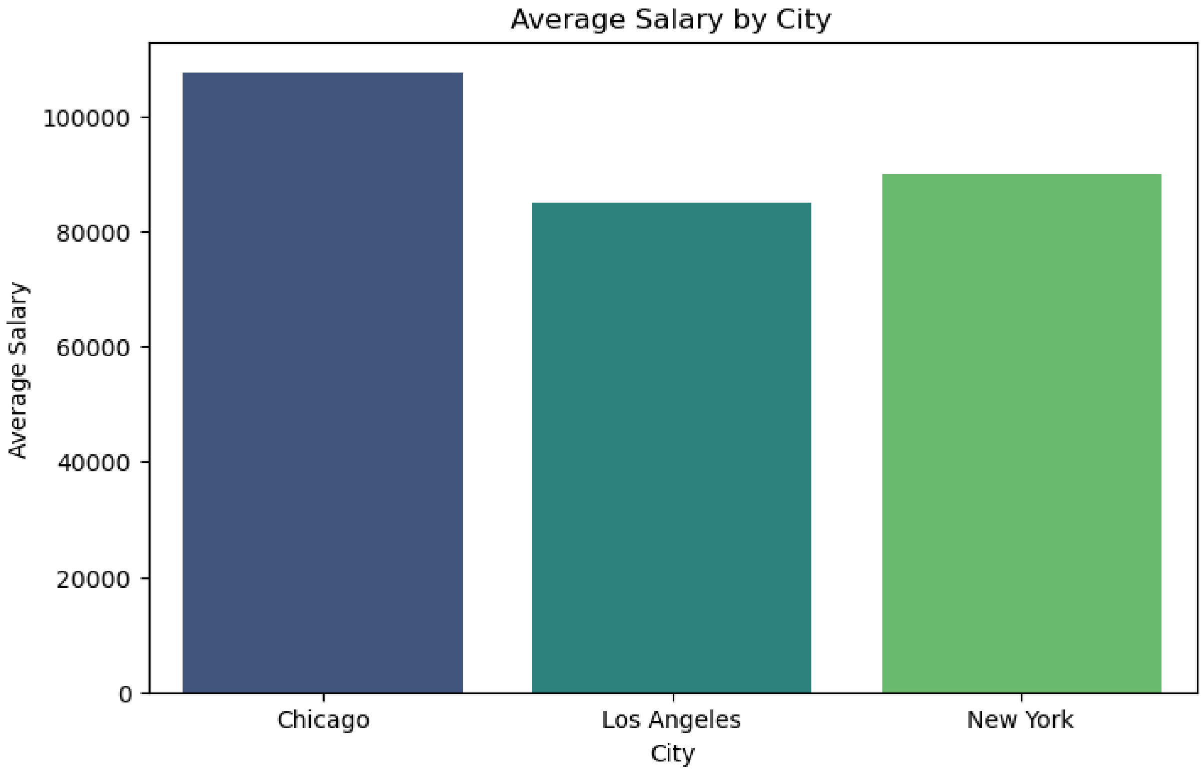
```
Average Salary by City:
          City    Salary
0      Chicago  107500.0
1  Los Angeles   85000.0
2     New York   90000.0
```

5. Visualizing Data

In [279…

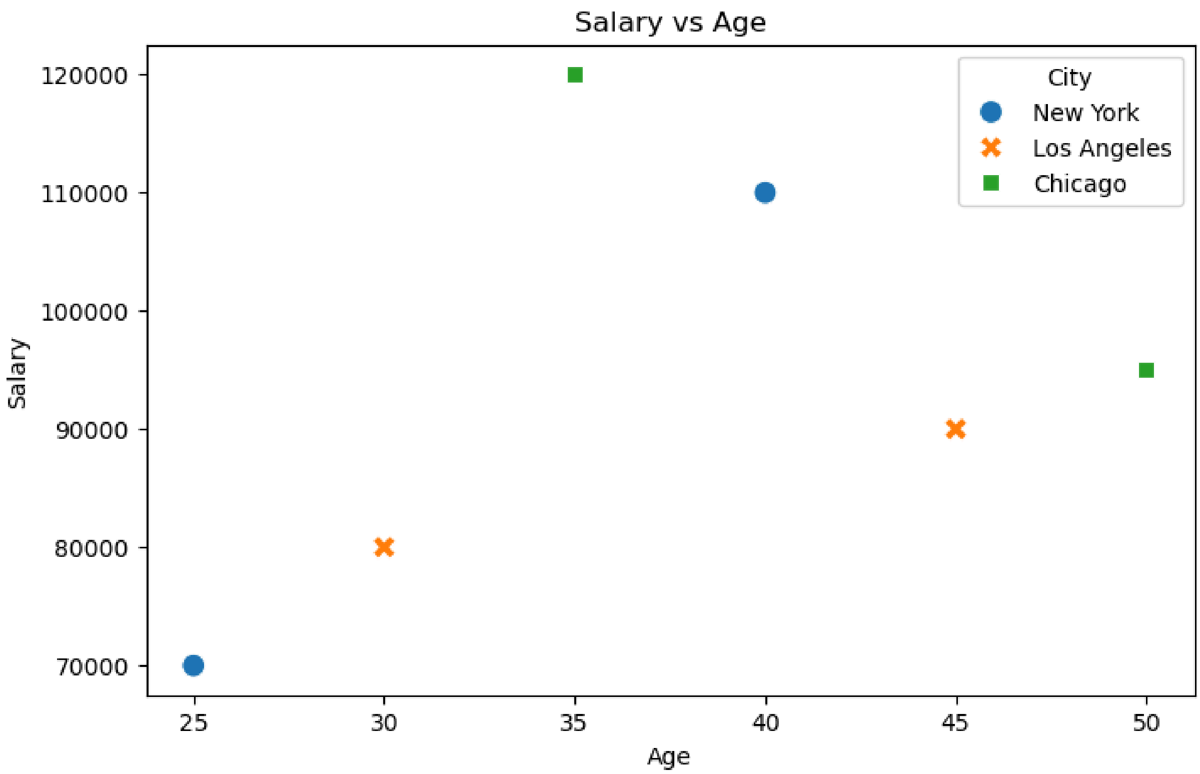
```
# Bar plot for average salary by city
plt.figure(figsize=(8, 5))
sns.barplot(x='City', y='Salary', data=average_salary_by_city, palette='viridis')
plt.title('Average Salary by City')
plt.xlabel('City')
plt.ylabel('Average Salary')
plt.show()
```

```
C:\Users\rajsh\AppData\Local\Temp\ipykernel_26176\3133909832.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='City', y='Salary', data=average_salary_by_city, palette='viridis')
```



In [281…

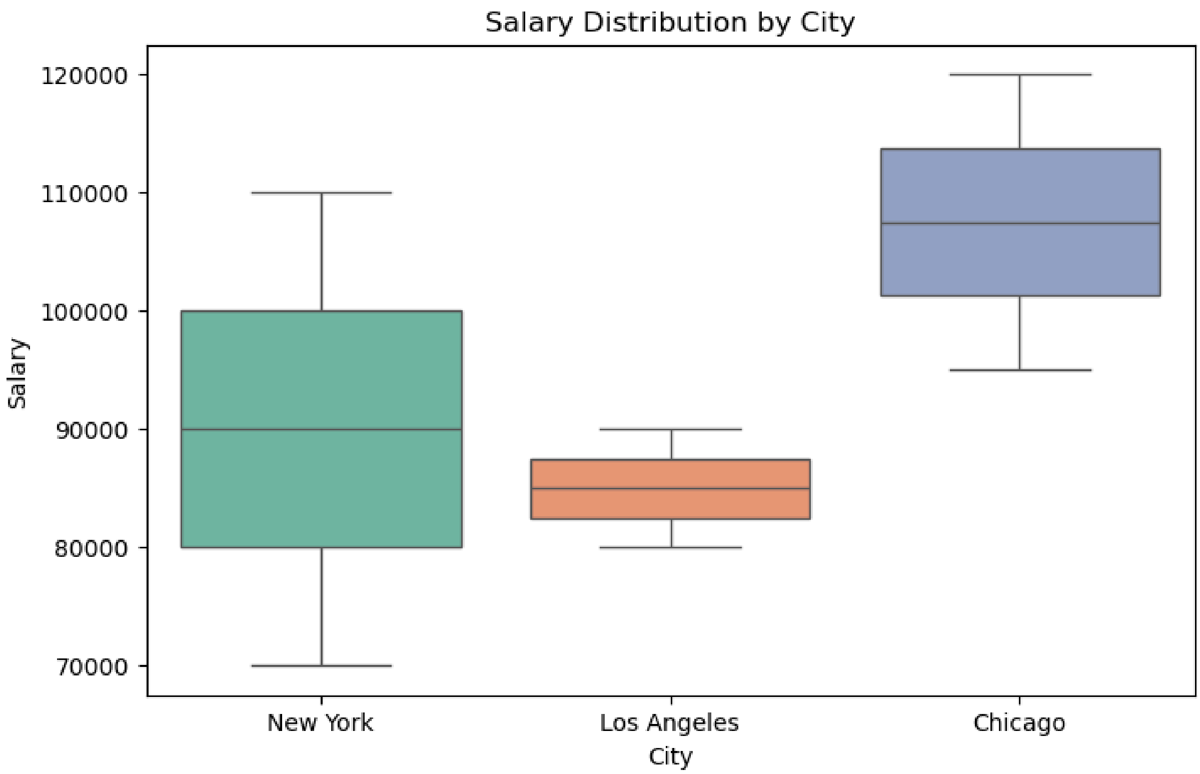
```
# Scatter plot for Age vs Salary
plt.figure(figsize=(8, 5))
sns.scatterplot(x='Age', y='Salary', data=df, hue='City', style='City', s=100)
plt.title('Salary vs Age')
plt.xlabel('Age')
plt.ylabel('Salary')
plt.legend(title='City')
plt.show()
```



In [283…

```
# Box plot to show salary distribution by city
plt.figure(figsize=(8, 5))
sns.boxplot(x='City', y='Salary', data=df, palette='Set2')
plt.title('Salary Distribution by City')
plt.xlabel('City')
plt.ylabel('Salary')
plt.show()
```

```
C:\Users\rajsh\AppData\Local\Temp\ipykernel_26176\1967012850.py:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.1
4.0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.boxplot(x='City', y='Salary', data=df, palette='Set2')
```


Salary Distribution by City

6. Correlation Analysis

In [ ]:

```
# Correlation matrix
correlation_matrix = df.corr()
print("\nCorrelation Matrix:")
print(correlation_matrix)

# Heatmap for correlation
plt.figure(figsize=(8, 5))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', square=True)
plt.title('Correlation Matrix Heatmap')
plt.show()
```

Create pivot tables and group data for analysis.

1. Importing Required Libraries

In [291…

```
import pandas as pd
```

2. Creating a Sample DataFrame

In [293…

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Edward', 'Fiona', 'George'],
    'City': ['New York', 'Los Angeles', 'Chicago', 'New York', 'Los Angeles', 'Chic
    'Sales': [250, 300, 200, 400, 500, 300, 150],
    'Quarter': ['Q1', 'Q1', 'Q2', 'Q2', 'Q2', 'Q3', 'Q3']
}

df = pd.DataFrame(data)
```

```
print("Original DataFrame:")
print(df)
```

```
Original DataFrame:
      Name         City  Sales Quarter
0    Alice     New York    250      Q1
1      Bob  Los Angeles    300      Q1
2  Charlie      Chicago    200      Q2
3    David     New York    400      Q2
4   Edward  Los Angeles    500      Q2
5    Fiona      Chicago    300      Q3
6   George     New York    150      Q3
```

3. Creating a Pivot Table

In [295…

```
pivot_table = pd.pivot_table(df, values='Sales', index='City', columns='Quarter', a
print("\nPivot Table (Total Sales by City and Quarter):")
print(pivot_table)
```

```
Pivot Table (Total Sales by City and Quarter):
Quarter       Q1   Q2   Q3
City
Chicago        0  200  300
Los Angeles  300  500    0
New York     250  400  150
```

4. Grouping Data

In [297…

```
# Grouping by City and calculating the total sales
grouped_data = df.groupby('City')['Sales'].sum().reset_index()
print("\nGrouped Data (Total Sales by City):")
print(grouped_data)
```

```
Grouped Data (Total Sales by City):
          City  Sales
0      Chicago    500
1  Los Angeles    800
2     New York    800
```

5. Grouping by Multiple Columns

In [299…

```
# Grouping by City and Quarter and calculating total sales
grouped_by_city_quarter = df.groupby(['City', 'Quarter'])['Sales'].sum().reset_inde
print("\nGrouped Data (Total Sales by City and Quarter):")
print(grouped_by_city_quarter)
```

```
Grouped Data (Total Sales by City and Quarter):
          City Quarter  Sales
0      Chicago      Q2    200
1      Chicago      Q3    300
2  Los Angeles      Q1    300
3  Los Angeles      Q2    500
4     New York      Q1    250
5     New York      Q2    400
6     New York      Q3    150
```

6. Aggregating with Multiple Functions

In [301…

```
grouped_multiple = df.groupby('City').agg({'Sales': ['sum', 'mean', 'max']}).reset_
print("\nGrouped Data with Multiple Aggregation Functions:")
print(grouped_multiple)
```

```
Grouped Data with Multiple Aggregation Functions:
          City Sales                 
                 sum        mean  max
0      Chicago   500  250.000000  300
1  Los Angeles   800  400.000000  500
2     New York   800  266.666667  400
```

If Statements

Demonstrate conditional logic using if, else, and elif statements

1: Age Classification

In [303…

```
def classify_age(age):
    if age < 0:
        return "Invalid age"
    elif age < 13:
        return "Child"
    elif age < 20:
        return "Teenager"
    elif age < 65:
        return "Adult"
    else:
        return "Senior"

# Test the function with different ages
ages = [5, 13, 17, 30, 70, -1]

for age in ages:
    classification = classify_age(age)
    print(f"Age: {age} - Classification: {classification}")
```

```
Age: 5 - Classification: Child
Age: 13 - Classification: Teenager
Age: 17 - Classification: Teenager
Age: 30 - Classification: Adult
Age: 70 - Classification: Senior
Age: -1 - Classification: Invalid age
```

2. Simple Calculator

In [305…

```
def simple_calculator(a, b, operation):
    if operation == 'add':
        return a + b
    elif operation == 'subtract':
        return a - b
    elif operation == 'multiply':
        return a * b
    elif operation == 'divide':
        if b != 0:
            return a / b
        else:
            return "Error: Division by zero"
    else:
        return "Invalid operation"

print(simple_calculator(10, 5, 'add'))
print(simple_calculator(10, 5, 'subtract'))
print(simple_calculator(10, 5, 'multiply'))
print(simple_calculator(10, 0, 'divide'))
print(simple_calculator(10, 5, 'unknown'))
```

```
15
5
50
Error: Division by zero
Invalid operation
```

3: Temperature Converter

In [307…

```
def convert_temperature(value, scale):
    if scale == 'C':
        return (value * 9/5) + 32  # Convert to Fahrenheit
    elif scale == 'F':
        return (value - 32) * 5/9  # Convert to Celsius
    else:
        return "Invalid scale"

# Test the temperature converter
print(convert_temperature(100, 'C'))  # Output: 212.0
print(convert_temperature(32, 'F'))   # Output: 0.0
print(convert_temperature(0, 'K'))    # Output: Invalid scale
```

```
212.0
0.0
Invalid scale
```

4: Grading System

In [ ]:

```
def determine_grade(score):
    if score < 0 or score > 100:
        return "Invalid score"
    elif score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"

# Test the grading function
scores = [95, 82, 67, 58, 105, -10]

for score in scores:
    grade = determine_grade(score)
    print(f"Score: {score} - Grade: {grade}")
```

Create complex conditional expressions.

1: Checking Eligibility for a Discount

In [311…

```
def check_discount(membership, purchase_amount):
    if (membership == "premium" and purchase_amount > 100) or (membership == "regul
        return "Eligible for discount"
    else:
        return "Not eligible for discount"

# Test the function
print(check_discount("premium", 150))
print(check_discount("regular", 250))
print(check_discount("regular", 150))
print(check_discount("basic", 150))
```

```
Eligible for discount
Eligible for discount
Not eligible for discount
Not eligible for discount
```

2: Evaluating a Student's Status

In [313…

```
def evaluate_student(grade):
    if grade >= 90:
        return "Excellent"
    elif 75 <= grade < 90:
        return "Good"
    elif 50 <= grade < 75:
        return "Average"
    elif grade < 50:
        return "Needs Improvement"
    else:
        return "Invalid grade"

# Test the function
grades = [95, 85, 60, 45, -10]

for grade in grades:
    status = evaluate_student(grade)
    print(f"Grade: {grade} - Status: {status}")
```

```
Grade: 95 - Status: Excellent
Grade: 85 - Status: Good
Grade: 60 - Status: Average
Grade: 45 - Status: Needs Improvement
Grade: -10 - Status: Needs Improvement
```

3: Complex Age Group Classification

In [315…

```
def classify_age(age):
    if age < 0:
        return "Invalid age"
    elif age <= 12:
        return "Child"
    elif 13 <= age <= 19:
        return "Teenager"
    elif 20 <= age <= 64:
        return "Adult"
    elif age >= 65:
        return "Senior"
    else:
        return "Invalid age"

# Test the function
ages = [10, 15, 30, 65, 70, -5]

for age in ages:
    classification = classify_age(age)
    print(f"Age: {age} - Classification: {classification}")
```

```
Age: 10 - Classification: Child
Age: 15 - Classification: Teenager
Age: 30 - Classification: Adult
Age: 65 - Classification: Senior
Age: 70 - Classification: Senior
Age: -5 - Classification: Invalid age
```

4: Login Access Control

In [317…

```
def check_login(username, password):
    if username == "admin" and password == "admin123":
        return "Access granted: Admin"
    elif username == "user" and password == "user123":
        return "Access granted: User"
    elif username != "admin" and username != "user":
        return "Access denied: Invalid username"
    else:
        return "Access denied: Incorrect password"

# Test the function
print(check_login("admin", "admin123"))
print(check_login("user", "user123"))
print(check_login("guest", "guest123"))
print(check_login("user", "wrongpass"))
```

```
Access granted: Admin
Access granted: User
Access denied: Invalid username
Access denied: Incorrect password
```

Implement nested if statements.

1: Grade Classification

In [319…

```
def classify_student(score):
    if score >= 0 and score <= 100:  # Check for valid score range
        if score >= 90:
            return "Grade: A"
        elif score >= 80:
            return "Grade: B"
        elif score >= 70:
            return "Grade: C"
        elif score >= 60:
            return "Grade: D"
        else:
            return "Grade: F - Needs Improvement"
    else:
        return "Invalid score"

scores = [95, 82, 67, 45, -5, 110]

for score in scores:
    result = classify_student(score)
    print(f"Score: {score} - Result: {result}")
```

```
Score: 95 - Result: Grade: A
Score: 82 - Result: Grade: B
Score: 67 - Result: Grade: D
Score: 45 - Result: Grade: F - Needs Improvement
Score: -5 - Result: Invalid score
Score: 110 - Result: Invalid score
```

2: Vehicle Classification

In [321…

```
def classify_vehicle(vehicle_type, fuel_type):
    if vehicle_type == "Car":
        if fuel_type == "Petrol":
            return "Petrol Car"
        elif fuel_type == "Diesel":
            return "Diesel Car"
        else:
            return "Electric Car"
    elif vehicle_type == "Truck":
        if fuel_type == "Diesel":
            return "Diesel Truck"
        else:
            return "Electric Truck"
    elif vehicle_type == "Motorcycle":
        if fuel_type == "Petrol":
            return "Petrol Motorcycle"
        else:
            return "Electric Motorcycle"
    else:
        return "Unknown vehicle type"

# Test the function
vehicles = [
    ("Car", "Petrol"),
    ("Truck", "Diesel"),
    ("Motorcycle", "Electric"),
    ("Car", "Diesel"),
    ("Bicycle", "None")
]

for vehicle in vehicles:
    vehicle_classification = classify_vehicle(*vehicle)
    print(f"Vehicle Type: {vehicle[0]}, Fuel Type: {vehicle[1]} - Classification: {
```

```
Vehicle Type: Car, Fuel Type: Petrol - Classification: Petrol Car
Vehicle Type: Truck, Fuel Type: Diesel - Classification: Diesel Truck
Vehicle Type: Motorcycle, Fuel Type: Electric - Classification: Electric Motorcycle
Vehicle Type: Car, Fuel Type: Diesel - Classification: Diesel Car
Vehicle Type: Bicycle, Fuel Type: None - Classification: Unknown vehicle type
```

3: Login System

In [323…

```
def login(username, password):
    if username == "admin":
        if password == "admin123":
            return "Welcome Admin!"
        else:
            return "Incorrect password for Admin."
    elif username == "user":
        if password == "user123":
            return "Welcome User!"
        else:
            return "Incorrect password for User."
    else:
        return "Username not recognized."

# Test the function
login_attempts = [
    ("admin", "admin123"),
    ("admin", "wrongpass"),
    ("user", "user123"),
    ("guest", "guestpass")
]
for attempt in login_attempts:
    result = login(*attempt)
    print(f"Login attempt: {attempt} - Result: {result}")
```

```
Login attempt: ('admin', 'admin123') - Result: Welcome Admin!
Login attempt: ('admin', 'wrongpass') - Result: Incorrect password for Admin.
Login attempt: ('user', 'user123') - Result: Welcome User!
Login attempt: ('guest', 'guestpass') - Result: Username not recognized.
```

Loops

Use for loops to iterate over sequences.

1: Iterating Over a List

In [325…

```
numbers = [1, 2, 3, 4, 5]

print("Iterating over a list:")
for number in numbers:
    print(number)
```

```
Iterating over a list:
1
2
3
4
5
```

2: Iterating Over a String

In [327…

```
message = "Hello, World!"

print("\nIterating over a string:")
for char in message:
    print(char)
```

```
Iterating over a string:
H
e
l
l
o
,
 
W
o
r
l
d
!
```

3: Iterating Over a Tuple

In [331…

```
fruits = ("apple", "banana", "cherry")

print("\nIterating over a tuple:")
for fruit in fruits:
    print(fruit)
```

```
Iterating over a tuple:
apple
banana
cherry
```

4: Iterating Over a Dictionary

In [333…

```
student_grades = {
    "Alice": 85,
    "Bob": 90,
    "Charlie": 78
}

print("\nIterating over a dictionary:")
for student, grade in student_grades.items():
    print(f"{student}: {grade}")
```

```
Iterating over a dictionary:
Alice: 85
Bob: 90
Charlie: 78
```

5: Using range() with a for Loopprint("\nUsing range() to iterate:") for i in range(5): # This will iterate from 0 to 4 print(i) 6: Nested For Loops

In [337…

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print("\nIterating over a matrix:")
for row in matrix:
    for element in row:
        print(element, end=' ')
    print()  # Print a newline after each row
```

```
Iterating over a matrix:
1 2 3
4 5 6
7 8 9
```

Employ while loops for indefinite iteration.

1: Basic Counter

In [343…

```
count = 1

print("Counting from 1 to 5:")
while count <= 5:
    print(count)
    count += 1
```

```
Counting from 1 to 5:
1
2
3
4
5
```

2: User Input Validation

In [347…

```
user_input = ""

while user_input.lower() != "yes":
    user_input = input("Do you want to continue? (yes/no): ")

print("Thank you for confirming!")
```

```
Thank you for confirming!
```

3: Summing Numbers

In [351…

```
total = 0
number = 0

print("Enter numbers to sum them up (enter a negative number to stop):")

while number >= 0:
    number = int(input("Enter a number: "))
    if number >= 0:
        total += number

print(f"The total sum is: {total}")
```

```
Enter numbers to sum them up (enter a negative number to stop):
The total sum is: 0
```

4: Infinite Loop with Break

In [353…

```
while True:
    command = input("Enter 'exit' to quit: ")
    if command.lower() == "exit":
        print("Exiting the loop.")
        break  # Exit the loop
    else:
        print("You entered:", command)
```

```
Exiting the loop.
```

5: Guessing Game

In [355…

```
import random

secret_number = random.randint(1, 10)
guess = 0

print("Guess the secret number between 1 and 10:")

while guess != secret_number:
    guess = int(input("Enter your guess: "))
    if guess < secret_number:
        print("Too low! Try again.")
    elif guess > secret_number:
        print("Too high! Try again.")
    else:
        print("Congratulations! You guessed it right.")
```

```
Guess the secret number between 1 and 10:
Congratulations! You guessed it right.
```

6: Countdown Timer

In [357…

```
import time

countdown = 5

print("Countdown Timer:")
while countdown > 0:
    print(countdown)
    time.sleep(1)  # Pause for 1 second
    countdown -= 1

print("Time's up!")
```

```
Countdown Timer:
5
4
3
2
1
Time's up!
```

Implement nested loops.

1: Multiplication Table

In [359…

```
print("Multiplication Table:")
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i * j:2}", end=' ')
    print()
```

```
Multiplication Table:
 1  2  3  4  5
 2  4  6  8 10
 3  6  9 12 15
 4  8 12 16 20
 5 10 15 20 25
```

In [361…

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print("Matrix Elements:")
for row in matrix:
    for element in row:
        print(element, end=' ')
    print()
```

```
Matrix Elements:
1 2 3
4 5 6
7 8 9
```

3: Generating Combinations

In [363…

```
fruits = ["apple", "banana", "cherry"]
colors = ["red", "yellow", "green"]

print("Fruit and Color Combinations:")
for fruit in fruits:
    for color in colors:
        print(f"{fruit} - {color}")
```

```
Fruit and Color Combinations:
apple - red
apple - yellow
apple - green
banana - red
banana - yellow
banana - green
cherry - red
cherry - yellow
cherry - green
```

4: Nested While Loops

In [365…

```
rows = 5
current_row = 1

print("Star Pattern:")
while current_row <= rows:
    current_star = 1
    while current_star <= current_row:
        print("*", end=' ')
        current_star += 1
    print()  # Newline after each row
    current_row += 1
```

```
Star Pattern:
*
* *
* * *
* * * *
* * * * *
```

5: Finding Common Elements

In [369…

```
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]

common_elements = []
print("Common Elements:")
for a in list1:
    for b in list2:
        if a == b:
            common_elements.append(a)

print(common_elements)
```

```
Common Elements:
[4, 5]
```

Utilize break and continue statements.

1: Using break to Exit a Loop

In [371…

```
print("Finding first even number:")
for number in range(1, 11):
    if number % 2 == 0:
        print(f"The first even number is: {number}")
        break
```

```
Finding first even number:
The first even number is: 2
```

2: Using continue to Skip Iterations

In [373…

```python
print("Skipping odd numbers:")
for number in range(1, 11):
    if number % 2 != 0:
        continue  # Skip the rest of the loop for odd numbers
    print(number)
```

```
Skipping odd numbers:
2
4
6
8
10
```

3: Combining break and continue

In [375…

```python
print("Finding numbers greater than 5, skipping 3:")
for number in range(10):
    if number == 3:
        continue  # Skip number 3
    if number > 5:
        print(f"Found a number greater than 5: {number}")
        break
```

```
Finding numbers greater than 5, skipping 3:
Found a number greater than 5: 6
```

4: Using break in a While Loop

In [377…

```python
count = 0
print("Counting until 5:")
while True:
    count += 1
    if count > 5:
        break  # Exit the loop when count exceeds 5
    print(count)
```

```
Counting until 5:
1
2
3
4
5
```

5: User Input with continue

In [379…

```python
print("Enter numbers (enter a negative number to stop):")
while True:
    number = int(input("Enter a number: "))
    if number < 0:
        print("Exiting loop.")
        break  # Exit the loop on negative input
    if number % 2 != 0:
        print("Skipping odd number.")
        continue  # Skip processing for odd numbers
    print(f"Processing even number: {number}")
```

```
Enter numbers (enter a negative number to stop):
```

```
Exiting loop.
```

Lists, Tuples, Sets, Dictionaries

Create and manipulate lists, tuples, sets, and dictionaries.

Lists1. create a list

In [381…

```
colors = ["red", "green", "blue", "yellow"]

print(colors)
```

```
['red', 'green', 'blue', 'yellow']
```

2. Appending to a List

In [383…

```
my_list = []

my_list.append("apple")
my_list.append("banana")
my_list.append("cherry")

print("List after appending:", my_list)
```

```
List after appending: ['apple', 'banana', 'cherry']
```

3. Popping from a Listfruits = ["apple", "banana", "cherry", "date"] print("Original list:", fruits) last_fruit = fruits.pop() print("Popped item:", last_fruit) print("List after popping:", fruits)4. List of Strings

In [387…

```
fruits = ["apple", "banana", "cherry", "date"]

print("Fruits:", fruits)
```

```
Fruits: ['apple', 'banana', 'cherry', 'date']
```

5. Nested List

In [389…

```
nested_list = [[1, 2, 3], ["a", "b", "c"], [True, False]]

print("Nested List:", nested_list)
```

```
Nested List: [[1, 2, 3], ['a', 'b', 'c'], [True, False]]
```

Tuples1. Basic Tuple

In [393…

```
numbers = (1, 2, 3, 4, 5)

print("Basic Tuple:", numbers)
```

```
Basic Tuple: (1, 2, 3, 4, 5)
```

2. Tuple of Strings

In [395…

```
names = ("Alice", "Bob", "Charlie")

print("Tuple of Names:", names)
```

```
Tuple of Names: ('Alice', 'Bob', 'Charlie')
```

3. Mixed Data Types - Tuples

In [399...

```
mixed_tuple = (1, "hello", 3.14, False)

print("Mixed Data Types Tuple:", mixed_tuple)
```

```
Mixed Data Types Tuple: (1, 'hello', 3.14, False)
```

4. Nested Tuple

In [401...

```
nested_tuple = ((1, 2, 3), ("a", "b", "c"), (True, False))

print("Nested Tuple:", nested_tuple)
```

```
Nested Tuple: ((1, 2, 3), ('a', 'b', 'c'), (True, False))
```

5. Tuple Unpacking

In [403...

```
coordinates = (10, 20)

x, y = coordinates

print("X Coordinate:", x)
print("Y Coordinate:", y)
```

```
X Coordinate: 10
Y Coordinate: 20
```

Sets1. Basic Set

In [407...

```
numbers = {1, 2, 3, 4, 5}

print("Basic Set:", numbers)
```

```
Basic Set: {1, 2, 3, 4, 5}
```

2. Set of Strings

In [411...

```
fruits = {"apple", "banana", "cherry", "banana"}  # Duplicates will be ignored

print("Set of Fruits:", fruits)
```

```
Set of Fruits: {'apple', 'banana', 'cherry'}
```

3. Mixed Data Types

In [415...

```
mixed_set = {1, "hello", 3.14, (1, 2)}

print("Mixed Data Types Set:", mixed_set)
```

```
Mixed Data Types Set: {3.14, 1, (1, 2), 'hello'}
```

4: Set Comprehension

In [419...

```
squares_of_evens = {x**2 for x in range(11) if x % 2 == 0}

print("Squares of Even Numbers:", squares_of_evens)
```

```
Squares of Even Numbers: {0, 64, 4, 36, 100, 16}
```

5: Checking Membership and Subsets

In [421...

```
set_x = {1, 2, 3}
set_y = {1, 2, 3, 4, 5}

# Checking membership
is_two_in_x = 2 in set_x
is_four_in_x = 4 in set_x
```

```
# Checking if set_x is a subset of set_y
is_subset = set_x.issubset(set_y)

# Displaying the results
print("Is 2 in set_x?", is_two_in_x)
print("Is 4 in set_x?", is_four_in_x)
print("Is set_x a subset of set_y?", is_subset)
```

```
Is 2 in set_x? True
Is 4 in set_x? False
Is set_x a subset of set_y? True
```

Dictionary1. Basic Dictionary

In [423…

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

print("Basic Dictionary:", person)
```

```
Basic Dictionary: {'name': 'Alice', 'age': 30, 'city': 'New York'}
```

2. Accessing Values

In [427…

```
fruits = {
    "apple": 2,
    "banana": 5,
    "cherry": 10
}

print("Number of bananas:", fruits["banana"])
```

```
Number of bananas: 5
```

3. Adding and Updating Values

In [431…

```
car = {
    "brand": "Toyota",
    "model": "Camry",
    "year": 2020
}

car["color"] = "blue"

car["year"] = 2021

print("Updated Car Dictionary:", car)
```

```
Updated Car Dictionary: {'brand': 'Toyota', 'model': 'Camry', 'year': 2021, 'color':
'blue'}
```

4. Dictionary Methods

In [435…

```
employee = {
    "name": "Bob",
    "age": 25,
    "department": "HR"
}
```

```
keys = employee.keys()
values = employee.values()
items = employee.items()

print("Keys:", keys)
print("Values:", values)
print("Items:", items)
```

```
Keys: dict_keys(['name', 'age', 'department'])
Values: dict_values(['Bob', 25, 'HR'])
Items: dict_items([('name', 'Bob'), ('age', 25), ('department', 'HR')])
```

5. Dictionary Comprehension

In [437…

```
squares = {x: x**2 for x in range(1, 6)}

print("Dictionary of Squares:", squares)
```

```
Dictionary of Squares: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Understand the differences between these data structures

1. Lists Definition: An ordered collection of items that can contain duplicates. Mutability: Mutable (you can change, add, or remove items). Syntax: Defined using square brackets []. Access: Items can be accessed by their index. Performance: Generally slower than tuples due to their mutability. Eg: fruits = ["apple", "banana", "cherry"] fruits.append("orange") # Adding an item2. Tuples Definition: An ordered collection of items that can also contain duplicates. Mutability: Immutable (once created, cannot be modified). Syntax: Defined using parentheses (). Access: Items can be accessed by their index. Performance: Generally faster than lists due to their immutability. Eg: coordinates = (10.0, 20.0) # coordinates[0] = 15.0 # This will raise an error since tuples are immutable3. Sets Definition: An unordered collection of unique items (no duplicates allowed). Mutability: Mutable (you can add or remove items). Syntax: Defined using curly braces {} or the set() constructor. Access: Items cannot be accessed by index; they are unordered. Performance: Fast membership tests and operations like unions and intersections. Eg: unique_fruits = {"apple", "banana", "cherry", "apple"} # "apple" will be stored only once unique_fruits.add("orange") # Adding a unique item4. Dictionaries Definition: An unordered collection of key-value pairs, where keys must be unique. Mutability: Mutable (you can change, add, or remove key-value pairs). Syntax: Defined using curly braces {} with colons separating keys and values. Access: Values can be accessed using their keys. Performance: Fast lookups by key. Eg: student_grades = {"Alice": 85, "Bob": 90} student_grades["Charlie"] = 78 # Adding a new key-value pair

Perform operations like indexing, slicing, adding, removing elements.

1. Lists - Indexing, Slicing, Adding elements, Removing Elements

In [449…

```
# Indexing - Lists
fruits = ["apple", "banana", "cherry"]
first_fruit = fruits[0]  # Accessing the first element
print("First fruit:", first_fruit)
```

```
First fruit: apple
```

In [451…

```
# Slicing - Lists
first_two_fruits = fruits[:2]
print("First two fruits:", first_two_fruits)
```

```
First two fruits: ['apple', 'banana']
```

In [453…

```
# Adding Elements - Lists
fruits.append("orange")  # Adding to the end
fruits.insert(1, "kiwi")  # Inserting at index 1
print("Updated fruits list:", fruits)
```

```
Updated fruits list: ['apple', 'kiwi', 'banana', 'cherry', 'orange']
```

In [455…

```
# Removing Elements - Lists
fruits.remove("banana")  # Remove by value
popped_fruit = fruits.pop()  # Remove the last element and return it
print("Removed fruit:", popped_fruit)
print("Fruits after removal:", fruits)
```

```
Removed fruit: orange
Fruits after removal: ['apple', 'kiwi', 'cherry']
```

2. Tuples - Indexing, Slicing, Adding elements, Removing Elements

In [457…

```
# Indexing
coordinates = (10.0, 20.0)
x = coordinates[0]  # Accessing the first element
print("X coordinate:", x)
```

```
X coordinate: 10.0
```

In [459…

```
# slicing
first_coordinate = coordinates[:1]
print("First coordinate:", first_coordinate)
```

```
First coordinate: (10.0,)
```

In [461…

```
# Adding elements
new_coordinates = coordinates + (30.0,)
print("New coordinates tuple:", new_coordinates)
```

```
New coordinates tuple: (10.0, 20.0, 30.0)
```

In [463…

```
# Removing Elements
coordinates = (10.0, 20.0, 30.0)
updated_coordinates = tuple(x for x in coordinates if x != 20.0)
print("Updated coordinates tuple:", updated_coordinates)
```

```
Updated coordinates tuple: (10.0, 30.0)
```

3. Sets - Indexing, Slicing, Adding elements, Removing Elements

In [465…

```
# Adding Elements
unique_fruits = {"apple", "banana"}
unique_fruits.add("cherry")
print("Unique fruits set after addition:", unique_fruits)
```

```
Unique fruits set after addition: {'apple', 'banana', 'cherry'}
```

In [467…

```
# Removing Elements
unique_fruits.remove("banana")
# unique_fruits.remove("orange")
unique_fruits.discard("orange")
print("Unique fruits set after removal:", unique_fruits)
```

```
Unique fruits set after removal: {'apple', 'cherry'}
```

4. Dictionaries - Indexing, Slicing, Adding elements, Removing Elements

In [469…

```
# Adding Elements
student_grades = {"Alice": 85, "Bob": 90}
student_grades["Charlie"] = 78  # Adding a new key-value pair
print("Student grades after addition:", student_grades)
```

```
Student grades after addition: {'Alice': 85, 'Bob': 90, 'Charlie': 78}
```

In [471…

```
# Removing Elements
del student_grades["Bob"]  # Remove by key
print("Student grades after removal:", student_grades)
```

```
Student grades after removal: {'Alice': 85, 'Charlie': 78}
```

In [475…

```
# Accessing Values
alice_grade = student_grades.get("Alice")  # Safe access
print("Alice's grade:", alice_grade)

# Accessing a non-existing key safely
unknown_grade = student_grades.get("Unknown", "Not Found")  # Default value if key
print("Unknown grade:", unknown_grade)
```

```
Alice's grade: 85
Unknown grade: Not Found
```

Explore built-in methods for each data structure.

1. Lists - append(), extend(), insert(), remove(), pop(), sort(), reverse()

In [477…

```
my_list = [1, 2, 3]

# Append an item
my_list.append(4)  # [1, 2, 3, 4]

# Extend the list with another list
my_list.extend([5, 6])  # [1, 2, 3, 4, 5, 6]

# Insert an item at a specific index
my_list.insert(0, 0)  # [0, 1, 2, 3, 4, 5, 6]

# Remove an item
my_list.remove(3)  # [0, 1, 2, 4, 5, 6]

# Pop an item (removes and returns the last item)
last_item = my_list.pop()  # last_item is 6, my_list is now [0, 1, 2, 4, 5]

# Sort the list
my_list.sort()  # [0, 1, 2, 4, 5]

# Reverse the list
my_list.reverse()  # [5, 4, 2, 1, 0]
```

2. Tuples - count(), index()

In [489…

```
my_tuple = (1, 2, 2, 3)

# Count occurrences of an item
count_of_twos = my_tuple.count(2)

# Find the index of the first occurrence of an item
index_of_three = my_tuple.index(3)
```

3. Sets - add(), remove(), discard(), union(), intersection(), difference(), clear()

In [491…

```
my_set = {1, 2, 3}

# Add an item
my_set.add(4)  # {1, 2, 3, 4}

# Remove an item
my_set.remove(2)  # {1, 3, 4}

# Discard an item (no error if the item is not present)
my_set.discard(10)  # {1, 3, 4}

# Union with another set
other_set = {3, 4, 5, 6}
union_set = my_set.union(other_set)  # {1, 3, 4, 5, 6}

# Intersection with another set
intersection_set = my_set.intersection(other_set)  # {3, 4}

# Difference between sets
difference_set = my_set.difference(other_set)  # {1}

# Clear all items from the set
my_set.clear()  # set()
```

4. Dictionaries - get(), keys(), values(), items(), pop(), update(), clear()

In [499…

```
my_dict = {'a': 1, 'b': 2}

# Get a value by key
value_a = my_dict.get('a')  # 1

# Get all keys
keys = my_dict.keys()  # dict_keys(['a', 'b'])

# Get all values
values = my_dict.values()  # dict_values([1, 2])

# Get all key-value pairs
items = my_dict.items()  # dict_items([('a', 1), ('b', 2)])

# Remove an item and get its value
removed_value = my_dict.pop('b')  # removed_value is 2, my_dict is now {'a': 1}

# Update the dictionary with another dictionary
my_dict.update({'b': 3, 'c': 4})  # {'a': 1, 'b': 3, 'c': 4}
```

```
# Clear all items from the dictionary
my_dict.clear()  # {}
```

Operators

Use arithmetic, comparison, logical, and assignment operators.

Arithmetic

In [501…

```
a = 10
b = 3

addition = a + b          # Addition
subtraction = a - b       # Subtraction
multiplication = a * b    # Multiplication
division = a / b          # Division (float)
floor_division = a // b   # Floor Division
modulus = a % b           # Modulus
exponentiation = a ** b   # Exponentiation

print("Addition:", addition)
print("Subtraction:", subtraction)
print("Multiplication:", multiplication)
print("Division:", division)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
```

Comparison Operators

In [503…

```
x = 5
y = 10

is_equal = (x == y)               # Equal to
is_not_equal = (x != y)           # Not equal to
is_greater = (x > y)              # Greater than
is_less = (x < y)                 # Less than
is_greater_equal = (x >= y)       # Greater than or equal to
is_less_equal = (x <= y)          # Less than or equal to

print("Is Equal:", is_equal)
print("Is Not Equal:", is_not_equal)
print("Is Greater:", is_greater)
print("Is Less:", is_less)
print("Is Greater or Equal:", is_greater_equal)
print("Is Less or Equal:", is_less_equal)
```

```
Is Equal: False
Is Not Equal: True
Is Greater: False
Is Less: True
Is Greater or Equal: False
Is Less or Equal: True
```

Logical Operators

In [505…

```
a = True
b = False

logical_and = a and b       # Logical AND
logical_or = a or b         # Logical OR
logical_not = not a         # Logical NOT

print("Logical AND:", logical_and)
print("Logical OR:", logical_or)
print("Logical NOT:", logical_not)
```

```
Logical AND: False
Logical OR: True
Logical NOT: False
```

Assignment operators.

In [507…

```
x = 5

x += 3   # Add and assign (x becomes 8)
x -= 2   # Subtract and assign (x becomes 6)
x *= 4   # Multiply and assign (x becomes 24)
x /= 6   # Divide and assign (x becomes 4.0)
x //= 2  # Floor divide and assign (x becomes 2.0)
x %= 3   # Modulus and assign (x becomes 2.0)
x **= 3  # Exponentiate and assign (x becomes 8.0)

print("Final value of x:", x)
```

```
Final value of x: 8.0
```

Understand operator precedence.

Operator Precedence Overview Parentheses (): Overrides other precedence rules. Exponentiation *: *Raises numbers to the power of others. Unary Plus and Minus +x, -x: Applies to a single operand. Multiplication* , Division /, Floor Division //, Modulus %: Arithmetic operations. Addition +, Subtraction -: Basic arithmetic. Bitwise Shifts <<, >>: Bit manipulation. Bitwise AND &: Logical conjunction for bits. Bitwise XOR ^: Exclusive OR for bits. Bitwise OR |: Logical disjunction for bits. Comparison Operators ==, !=, >, <, >=, <=: Used to compare values. Logical NOT not: Negates a boolean value. Logical AND and: Conjunction for boolean values. Logical OR or: Disjunction for boolean values.

Apply operators in expressions and calculations.

In [510…

```python
# 1. Arithmetic Operators


# Defining variables
a = 15
b = 4

# Addition
addition = a + b  # 19
print("Addition:", addition)

# Subtraction
subtraction = a - b  # 11
print("Subtraction:", subtraction)

# Multiplication
multiplication = a * b  # 60
print("Multiplication:", multiplication)

# Division
division = a / b  # 3.75
print("Division:", division)

# Floor Division
floor_division = a // b  # 3
print("Floor Division:", floor_division)

# Modulus
modulus = a % b  # 3
print("Modulus:", modulus)

# Exponentiation
exponentiation = a ** b  # 50625
print("Exponentiation:", exponentiation)
```

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

In [512…

```python
# 2. Comparison Operators
x = 10
y = 20

# Equal to
is_equal = (x == y)  # False
print("Is Equal:", is_equal)
```

```python
# Not equal to
is_not_equal = (x != y)  # True
print("Is Not Equal:", is_not_equal)

# Greater than
is_greater = (x > y)  # False
print("Is Greater:", is_greater)

# Less than
is_less = (x < y)  # True
print("Is Less:", is_less)

# Greater than or equal to
is_greater_equal = (x >= y)  # False
print("Is Greater or Equal:", is_greater_equal)

# Less than or equal to
is_less_equal = (x <= y)  # True
print("Is Less or Equal:", is_less_equal)
```

```
Is Equal: False
Is Not Equal: True
Is Greater: False
Is Less: True
Is Greater or Equal: False
Is Less or Equal: True
```

In [516…

```python
# 3. Logical Operators

a = True
b = False

# Logical AND
and_result = a and b  # False
print("Logical AND:", and_result)

# Logical OR
or_result = a or b  # True
print("Logical OR:", or_result)

# Logical NOT
not_result = not a  # False
print("Logical NOT:", not_result)
```

```
Logical AND: False
Logical OR: True
Logical NOT: False
```

In [518…

```python
# 4. Bitwise Operators

x = 10  # Binary: 1010
y = 4   # Binary: 0100

# Bitwise AND
bitwise_and = x & y  # 0 (Binary: 0000)
print("Bitwise AND:", bitwise_and)
```

```
# Bitwise OR
bitwise_or = x | y  # 14 (Binary: 1110)
print("Bitwise OR:", bitwise_or)

# Bitwise XOR
bitwise_xor = x ^ y  # 14 (Binary: 1110)
print("Bitwise XOR:", bitwise_xor)

# Bitwise NOT
bitwise_not = ~x  # -11 (Binary: ...11110101)
print("Bitwise NOT:", bitwise_not)

# Left Shift
left_shift = x << 2  # 40 (Binary: 101000)
print("Left Shift:", left_shift)

# Right Shift
right_shift = x >> 2  # 2 (Binary: 0010)
print("Right Shift:", right_shift)
```

```
Bitwise AND: 0
Bitwise OR: 14
Bitwise XOR: 14
Bitwise NOT: -11
Left Shift: 40
Right Shift: 2
```

In [520…

```
# 5. Combining Operators in Expressions

a = 10
b = 5
c = 2

# Combined expression
result = (a + b) * c - (b ** 2) / (a - b)
# = (15) * 2 - (25) / (5)
# = 30 - 5
# = 25
print("Combined Expression Result:", result)
```

```
Combined Expression Result: 25.0
```

Reading CSV files

Read CSV files into Pandas DataFrames.

1. Install Pandas (if you haven't already)

In [522…

```
pip install pandas
```

```
Requirement already satisfied: pandas in c:\users\rajsh\anaconda3\lib\site-packages
(2.2.2)
Requirement already satisfied: numpy>=1.26.0 in c:\users\rajsh\anaconda3\lib\site-pa
ckages (from pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in c:\users\rajsh\anaconda3\li
b\site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in c:\users\rajsh\anaconda3\lib\site-pac
kages (from pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in c:\users\rajsh\anaconda3\lib\site-p
ackages (from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\rajsh\anaconda3\lib\site-package
s (from python-dateutil>=2.8.2->pandas) (1.16.0)
Note: you may need to restart the kernel to use updated packages.
```

2. Sample CSV File

In [ ]:

```
name,age,city
Alice,30,New York
Bob,25,Los Angeles
Charlie,35,Chicago
```

3. Read the CSV File into a DataFrame

In [ ]:

```
import pandas as pd

# Read the CSV file
df = pd.read_csv('data.csv')

# Display the DataFrame
print(df)
```

4. Common Parameters for read_csv()

In [ ]:

```
df = pd.read_csv(
    'data.csv',
    sep=',',                 # Separator
    header=0,              # Use the first row as the header
    index_col=None,         # No specific index column
    usecols=['name', 'age'],  # Read only the 'name' and 'age' columns
    dtype={'age': int}      # Ensure 'age' is read as an integer
)

print(df)
```

In [ ]:

```
5. Handling Missing Values
```

In [ ]:

```
# Example CSV with missing values
# name,age,city
# Alice,30,New York
# Bob,,Los Angeles
# Charlie,35,Chicago

df = pd.read_csv('data_with_missing.csv')

# Display the DataFrame
print(df)
```

Explore different CSV reading options and parameters.

# Basic Syntax import pandas as pd df = pd.read_csv('filename.csv', **kwargs)# sep: Specify the delimiter to use. Default is ,. df = pd.read_csv('data.tsv', sep='\t') # header: Specify the row(s) to use as the column names. df = pd.read_csv('data.csv', header=None) # quoting: Control how quotes are handled in the CSV. Use csv.QUOTE_MINIMAL, csv.QUOTE_ALL, etc. df = pd.read_csv('data.csv', quoting=csv.QUOTE_ALL) # Quote all fields #names: Provide a list of column names to use. df = pd.read_csv('data.csv', names=['A', 'B', 'C'], header=0) #index_col: Specify a column to use as the index for the DataFrame. df = pd.read_csv('data.csv', index_col='name') # usecols: Specify which columns to read. This can be a list of column names or indices. df = pd.read_csv('data.csv', usecols=['name', 'age']) # Read only 'name' and 'age' # dtype: Specify the data type for each column. df = pd.read_csv('data.csv', dtype={'age': int}) # Ensure 'age' is an integer # na_values: Specify additional strings to recognize as NA/NaN. df = pd.read_csv('data.csv', na_values=['N/A', 'NULL']) # Treat 'N/A' and 'NULL' as NaN # parse_dates: Specify columns to parse as dates. df = pd.read_csv('data.csv', parse_dates=['date_column']) # Parse 'date_column' as datetime # skiprows: Specify the number of rows to skip at the start of the file. df = pd.read_csv('data.csv', skiprows=2) # Skip the first two rows # nrows: Specify the number of rows to read. df = pd.read_csv('data.csv', nrows=5) # Read only the first 5 rows #comment: Specify a character to indicate comments. Lines starting with this character will be ignored. df = pd.read_csv('data.csv', comment='#') # Ignore lines starting with '#' # encoding: Specify the character encoding for the CSV file (e.g., 'utf-8', 'latin1'). df = pd.read_csv('data.csv', encoding='latin1') # Use Latin-1 encoding # quoting: Control how quotes are handled in the CSV. Use csv.QUOTE_MINIMAL, csv.QUOTE_ALL, etc. import csv df = pd.read_csv('data.csv', quoting=csv.QUOTE_ALL) # Quote all fields #Example of Combining Parameters import pandas as pd df = pd.read_csv('data.csv', sep=',', # Separator header=0, # Use the first row as header index_col='name', # Use 'name' column as index usecols=['name', 'age'], # Read only 'name' and 'age' dtype={'age': int}, # Ensure 'age' is an integer na_values=['N/A', 'NULL'], # Treat 'N/A' and 'NULL' as NaN parse_dates=['date'], # Parse 'date' column as datetime skiprows=1 # Skip the first row) print(df)

Handle missing values and data cleaning

1. Detecting Missing Values

In [529…

```
import pandas as pd

# Sample DataFrame
data = {
    'name': ['Alice', 'Bob', None, 'Charlie'],
    'age': [30, None, 25, 35],
    'city': ['New York', 'Los Angeles', 'Chicago', None]
}
df = pd.DataFrame(data)

# Check for missing values
print(df.isnull())
```

```
    name    age   city
0  False  False  False
1  False   True  False
2   True  False  False
3  False  False   True
```

2. Summarizing Missing Values

In [531…

```
missing_count = df.isnull().sum()
print("Missing Values Count:\n", missing_count)
```

```
Missing Values Count:
 name    1
age     1
city    1
dtype: int64
```

3. Dropping Missing Values

```
In [533…
# Drop rows with any missing values
df_cleaned = df.dropna()
print(df_cleaned)
```

```
    name   age      city
0  Alice  30.0  New York
```

4. Filling Missing Values

```
In [535…
df_filled = df.fillna({'name': 'Unknown', 'age': 0, 'city': 'Unknown'})
print(df_filled)
```

```
      name   age         city
0    Alice  30.0     New York
1      Bob   0.0  Los Angeles
2  Unknown  25.0      Chicago
3  Charlie  35.0      Unknown
```

5. Removing Duplicates

```
In [537…
# Sample DataFrame with duplicates
data_with_duplicates = {
    'name': ['Alice', 'Bob', 'Alice', 'Charlie'],
    'age': [30, 25, 30, 35]
}
df_duplicates = pd.DataFrame(data_with_duplicates)

# Remove duplicates
df_no_duplicates = df_duplicates.drop_duplicates()
print(df_no_duplicates)
```

```
      name  age
0    Alice   30
1      Bob   25
3  Charlie   35
```

6. Renaming Columns

```
In [539…
df_renamed = df.rename(columns={'name': 'Name', 'age': 'Age'})
print(df_renamed)
```

```
      Name   Age         city
0    Alice  30.0     New York
1      Bob   NaN  Los Angeles
2     None  25.0      Chicago
3  Charlie  35.0         None
```

7. Changing Data Types

```
In [ ]:
df['age'] = df['age'].astype(int)  # Convert age to integer
print(df)
```

Python String Methods

Manipulate strings using various built-in methods.

In [546…

```
#1. Creating Strings
my_string = "Hello, World!"
print(my_string)  # Output: Hello, World!
```

```
Hello, World!
```

2. Accessing Characters

In [549…

```
# Accessing characters
first_char = my_string[0]  # 'H'
last_char = my_string[-1]   # '!'
print(first_char, last_char)
```

```
H !
```

3. String Length

In [551…

```
length = len(my_string)  # 13
print("Length:", length)
```

```
Length: 13
```

4. Changing Case

In [553…

```
# Changing case
print(my_string.upper())  # Output: HELLO, WORLD!
print(my_string.lower())  # Output: hello, world!
print(my_string.title())  # Output: Hello, World!
print(my_string.capitalize())  # Output: Hello, world!
```

```
HELLO, WORLD!
hello, world!
Hello, World!
Hello, world!
```

5. Stripping Whitespace

In [ ]:

```
hitespace_string = "   Hello, World!   "
print(whitespace_string.strip())  # Output: "Hello, World!"
```

6. Replacing Substrings

In [559…

```
replaced_string = my_string.replace("World", "Python")
print(replaced_string)  # Output: Hello, Python!
```

```
Hello, Python!
```

7. Splitting and Joining Strings

In [561…

```
words = my_string.split(", ")  # ['Hello', 'World!']
print(words)

# Joining a list into a string
```

```
joined_string = " - ".join(words)  # Hello - World!
print(joined_string)
```

```
['Hello', 'World!']
Hello - World!
```

8. Finding Substrings

In [563…

```
position = my_string.find("World")  # 7
print("Position of 'World':", position)

# Using index() will raise an error if not found
try:
    index_position = my_string.index("Python")  # Raises ValueError
except ValueError:
    print("Substring not found.")
```

```
Position of 'World': 7
Substring not found.
```

9. Checking String Contents

In [565…

```
alpha_string = "Hello"
digit_string = "12345"
alphanumeric_string = "Hello123"

print(alpha_string.isalpha())  # True
print(digit_string.isdigit())  # True
print(alphanumeric_string.isalnum())  # True
```

```
True
True
True
```

10. Formatting Strings

In [567…

```
name = "Alice"
age = 30

# Using f-strings (Python 3.6+)
formatted_string = f"{name} is {age} years old."
print(formatted_string)

# Using format()
formatted_string_format = "{} is {} years old.".format(name, age)
print(formatted_string_format)

# Using % operator
formatted_string_percent = "%s is %d years old." % (name, age)
print(formatted_string_percent)
```

```
Alice is 30 years old.
Alice is 30 years old.
Alice is 30 years old.
```

11. Checking String Start and End

In [569…

```
print(my_string.startswith("Hello"))  # True
print(my_string.endswith("!"))  # True
```

```
True
True
```

12. Counting Substrings

In [571…

```
count = my_string.count("o")  # 2
print("Count of 'o':", count)
```

```
Count of 'o': 2
```

Perform operations like concatenation, slicing, finding substrings.

1. Concatenation

In [575…

```
# String concatenation
string1 = "Hello"
string2 = "World"

# Using the + operator
concatenated_string = string1 + ", " + string2 + "!"  # "Hello, World!"
print(concatenated_string)

# Using join() method
joined_string = " ".join([string1, string2])  # "Hello World"
print(joined_string)
```

```
Hello, World!
Hello World
```

2. Slicing

In [577…

```
# Sample string
my_string = "Hello, World!"

# Slicing
substring1 = my_string[0:5]   # 'Hello' (from index 0 to 4)
substring2 = my_string[7:]     # 'World!' (from index 7 to end)
substring3 = my_string[:5]     # 'Hello' (from start to index 4)
substring4 = my_string[-6:]    # 'World!' (last 6 characters)
print(substring1, substring2, substring3, substring4)

# Slicing with step
substring_step = my_string[::2]  # 'Hlo ol!' (every second character)
print(substring_step)
```

```
Hello World! Hello World!
Hlo ol!
```

3. Finding Substrings

In [579…

```
# Sample string
search_string = "Hello, World!"

# Using find() method
position = search_string.find("World")  # Returns the starting index (7)
print("Position of 'World':", position)

# Using index() method
try:
    index_position = search_string.index("World")  # Returns the starting index (7)
    print("Index of 'World':", index_position)
except ValueError:
    print("'World' not found.")

# Searching for a non-existent substring
not_found_position = search_string.find("Python")  # Returns -1
print("Position of 'Python':", not_found_position)

# Using count() to count occurrences
count_occurrences = search_string.count("o")  # 2
print("Count of 'o':", count_occurrences)
```

```
Position of 'World': 7
Index of 'World': 7
Position of 'Python': -1
Count of 'o': 2
```

Convert strings to uppercase, lowercase, and title case.

1. Uppercase Conversion

In [581…

```
# Sample string
my_string = "Hello, World!"

# Convert to uppercase
uppercase_string = my_string.upper()
print("Uppercase:", uppercase_string)  # Output: "HELLO, WORLD!"
```

```
Uppercase: HELLO, WORLD!
```

2. Lowercase Conversion

In [583…

```
lowercase_string = my_string.lower()
print("Lowercase:", lowercase_string)  # Output: "hello, world!"
```

```
Lowercase: hello, world!
```

3. Title Case Conversion

In [585…

```
# Convert to title case
titlecase_string = my_string.title()
print("Title Case:", titlecase_string)  # Output: "Hello, World!"
```

```
Title Case: Hello, World!
```

In [587…

```
# Complete Example

# Sample strings
str1 = "python programming"
str2 = "welcome to the jungle"

# Uppercase
print("Uppercase:", str1.upper())  # Output: "PYTHON PROGRAMMING"

# Lowercase
print("Lowercase:", str2.lower())  # Output: "welcome to the jungle"
```

```
# Title Case
print("Title Case:", str1.title())  # Output: "Python Programming"
print("Title Case:", str2.title())  # Output: "Welcome To The Jungle"
```

```
Uppercase: PYTHON PROGRAMMING
Lowercase: welcome to the jungle
Title Case: Python Programming
Title Case: Welcome To The Jungle
```

Remove whitespace and split strings.

1. Removing Whitespace

In [589…

```
# Sample string with leading and trailing whitespace
whitespace_string = "   Hello, World!   "

# Remove leading and trailing whitespace
stripped_string = whitespace_string.strip()
print("Stripped:", stripped_string)  # Output: "Hello, World!"

# Remove leading whitespace
left_stripped = whitespace_string.lstrip()
print("Left Stripped:", left_stripped)  # Output: "Hello, World!   "

# Remove trailing whitespace
right_stripped = whitespace_string.rstrip()
print("Right Stripped:", right_stripped)  # Output: "   Hello, World!"
```

```
Stripped: Hello, World!
Left Stripped: Hello, World!
Right Stripped:    Hello, World!
```

2. Splitting Strings

In [593…

```
# Sample string
sample_string = "Hello, World! Welcome to Python."

# Split by whitespace (default behavior)
words = sample_string.split()
print("Words List:", words)
# Output: ['Hello,', 'World!', 'Welcome', 'to', 'Python.']

# Split by a specific delimiter (e.g., ',')
split_by_comma = sample_string.split(',')
print("Split by Comma:", split_by_comma)
# Output: ['Hello', ' World! Welcome to Python.']

# Split by a specific substring (e.g., 'to')
split_by_to = sample_string.split('to')
print("Split by 'to':", split_by_to)
# Output: ['Hello, World! Welc', 'me ', ' Python.']
```

```
Words List: ['Hello,', 'World!', 'Welcome', 'to', 'Python.']
Split by Comma: ['Hello', ' World! Welcome to Python.']
Split by 'to': ['Hello, World! Welcome ', ' Python.']
```

In [595…

```
# Complete Example

# Sample string with extra spaces
text = "   Python is great for   data analysis.   "

# Remove whitespace
cleaned_text = text.strip()
print("Cleaned Text:", cleaned_text)  # Output: "Python is great for   data analysi

# Split the cleaned text into words
words_list = cleaned_text.split()
print("Words List:", words_list)
# Output: ['Python', 'is', 'great', 'for', 'data', 'analysis.']
```

```
Cleaned Text: Python is great for   data analysis.
Words List: ['Python', 'is', 'great', 'for', 'data', 'analysis.']
```