Back

21_Handling_Exceptions.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates basic exception handling using try-except.
# It helps handle errors gracefully.

try:
    num = int(input("Enter a number: "))
    print(f"You entered: {num}")
except ValueError:
    print("That wasn't a valid number!")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

20_File_Writing.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates how to write data to a file.
# It opens a file in write mode and writes text to it.

with open('output.txt', 'w') as file:
    file.write('Hello, File!')
    file.write('\nThis is a new line.')
print('Data written to output.txt')
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

19_Break_Continue.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates the use of break and continue.
# 'break' exits the loop and 'continue' skips to the next iteration.

for i in range(5):
    if i == 3:
        break  # Exit the loop when i equals 3
    print(i)

# Example with 'continue'
for i in range(5):
    if i == 2:
        continue  # Skip when i equals 2
    print(i)
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

18_Nested_Loops.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates nested loops.
# Nested loops are loops inside another loop.

for i in range(3):
    for j in range(2):
        print(f"i: {i}, j: {j}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

17_Dictionary_Loop.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates how to loop through a dictionary.
# We can loop through both keys and values in a dictionary.

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

# Loop through keys
for key in person:
    print(f"Key: {key}, Value: {person[key]}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

16_Set_Operations.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates set operations.
# Sets are unordered collections of unique items.

my_set = {1, 2, 3, 4, 5}

# Add an element to the set
my_set.add(6)
print(f"Set after adding 6: {my_set}")

# Remove an element from the set
my_set.remove(3)
print(f"Set after removing 3: {my_set}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

16_Set_Operations.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates set operations.
# Sets are unordered collections of unique items.

my_set = {1, 2, 3, 4, 5}

# Add an element to the set
my_set.add(6)
print(f"Set after adding 6: {my_set}")

# Remove an element from the set
my_set.remove(3)
print(f"Set after removing 3: {my_set}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

15_Tuple_Operations.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates tuple operations.
# Tuples are immutable sequences, unlike lists.

my_tuple = (1, 2, 3, 4, 5)

# Access elements in a tuple
print(f"First element: {my_tuple[0]}")

# Tuple slicing
print(f"Last three elements: {my_tuple[-3:]}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

14_List_Comprehension.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates list comprehension, which is a compact way to create lists.
# List comprehension is faster and more readable.

# Create a list of squares using list comprehension
squares = [x**2 for x in range(5)]
print(f"Squares: {squares}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

12_List_Indexing.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates list indexing and accessing elements.
# Lists can store multiple values, and we access elements using their index.

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

# Access first item
print(f"First fruit: {fruits[0]}")

# Access last item
print(f"Last fruit: {fruits[-1]}")

# Access second item
print(f"Second fruit: {fruits[1]}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

11_String_Manipulation.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates basic string operations.
# We will manipulate strings using various methods.

text = "Hello, World!"

# Convert string to uppercase
upper_text = text.upper()
print(f"Uppercase: {upper_text}")

# Convert string to lowercase
lower_text = text.lower()
print(f"Lowercase: {lower_text}")

# Find a substring
substring_pos = text.find("World")
print(f"Position of 'World': {substring_pos}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

13_Sum_of_List.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program calculates the sum of a list of numbers.
# We will use the sum() function to calculate the total.

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

# Calculate the sum of the numbers in the list
total = sum(numbers)
print(f"Sum of the list: {total}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

11_String_Manipulation.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates basic string operations.
# We will manipulate strings using various methods.

text = "Hello, World!"

# Convert string to uppercase
upper_text = text.upper()
print(f"Uppercase: {upper_text}")

# Convert string to lowercase
lower_text = text.lower()
print(f"Lowercase: {lower_text}")

# Find a substring
substring_pos = text.find("World")
print(f"Position of 'World': {substring_pos}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

10_Reading_from_File.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program reads content from a file and prints it to the console.
# Make sure the file "sample.txt" exists in the same directory as this script.

# Open the file in read mode ('r')
with open("sample.txt", "r") as file:
    # Read the content of the file and store it in the variable 'content'
    content = file.read()

    # Print the content of the file
    print(content)  # Output: the content inside sample.txt
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

09_Dictionaries_in_Python.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# Dictionaries allow us to store data in key-value pairs.
# In this example, we create a dictionary with information about a person.

# Define a dictionary with keys and values
person = {
    "name": "Alice",  # Key: "name", Value: "Alice"
    "age": 30,        # Key: "age", Value: 30
    "city": "New York" # Key: "city", Value: "New York"
}

# Access values from the dictionary using the key
print(person["name"])  # Output: Alice
print(person.get("age"))  # Output: 30
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

08_List_Operations.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# Lists allow us to store multiple items in a single variable.
# In this example, we demonstrate how to create a list, add an item, remove an item,
# and iterate through the list.

# Create a list of fruits
fruits = ["apple", "banana", "cherry"]

# Add an item to the list
fruits.append("orange")  # Adds 'orange' to the end of the list

# Remove an item from the list
fruits.remove("banana")  # Removes 'banana' from the list

# Iterate through the list and print each fruit
for fruit in fruits:
    print(fruit)  # This will print each fruit in the list
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

07_Functions_in_Python.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# Functions allow us to group a set of instructions into a reusable block of code.
# In this example, we define a function 'greet' that takes a 'name' as input.

# Define the function
def greet(name):
    # The function returns a greeting message using the name provided
    return f"Hello, {name}!"  # 'f' before the string indicates a formatted string

# Call the function and pass "Alice" as the argument
print(greet("Alice"))  # Output: "Hello, Alice!"
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

06_While_Loop.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# A 'while' loop allows us to repeat a block of code as long as a condition is True.
# Here, we use a variable 'count' that starts at 0 and increments until it reaches 5.

# Initialize the count variable
count = 0

# The loop will continue as long as count is less than 5
while count < 5:
    print(f"Count: {count}")  # Print the current value of 'count'
    count += 1  # Increment the count by 1 after each iteration
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

04_If_Else_Statements.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This program demonstrates the use of an 'if-else' statement.
# An 'if' statement allows us to run a block of code only if a certain condition is met.

# Define a variable for age
age = 18  # Age can be any number, here we are testing the value 18

# If the age is greater than or equal to 18, print "You are an adult."
if age >= 18:
    print("You are an adult.")  # This code runs if the condition is True
else:
    print("You are a minor.")  # This code runs if the condition is False
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

15_Dropping_Duplicates.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a sample DataFrame with duplicate rows
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'Bob'],
    'Age': [24, 27, 22, 27],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Los Angeles']
}

df = pd.DataFrame(data)

# Drop duplicate rows from the DataFrame
df_no_duplicates = df.drop_duplicates()

# Display the updated DataFrame without duplicates
print(df_no_duplicates)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

02_Variables_and_Data_Types.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# In this example, we demonstrate the use of variables and different data types.
# Variables are used to store data that can be used later in the program.

# String variable
name = "Alice"  # Store a name in the variable 'name'

# Integer variable
age = 30  # Store an integer (whole number) in the variable 'age'

# Float variable
height = 5.5  # Store a floating-point number (decimal) in 'height'

# Boolean variable
is_active = True  # Store a boolean value in 'is_active'

# Print the values of the variables to the console
print(f"Name: {name}, Age: {age}, Height: {height}, Active: {is_active}")
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

05_For_Loop.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# A 'for' loop allows us to repeat a block of code a specific number of times.
# Here, we will use the 'range' function to repeat the code 5 times.

# 'range(5)' generates a sequence of numbers from 0 to 4 (5 is excluded)
for i in range(5):
    print(f"Iteration {i}")  # Print the current iteration number
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

03_Simple_Calculator.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This simple calculator performs basic arithmetic operations: addition, subtraction,
# multiplication, and division. We will work with two numbers, num1 and num2.

# Define two numbers for the calculation
num1 = 10
num2 = 5

# Addition: Add num1 and num2
sum_result = num1 + num2
print(f"Sum: {sum_result}")  # Print the result of addition

# Subtraction: Subtract num2 from num1
diff_result = num1 - num2
print(f"Difference: {diff_result}")  # Print the result of subtraction

# Multiplication: Multiply num1 by num2
prod_result = num1 * num2
print(f"Product: {prod_result}")  # Print the result of multiplication

# Division: Divide num1 by num2
div_result = num1 / num2
print(f"Division: {div_result}")  # Print the result of division
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

03_Simple_Calculator.py

Run

Edit and run code freely. Tutorial code resets on return.

```python
# This simple calculator performs basic arithmetic operations: addition, subtraction,
# multiplication, and division. We will work with two numbers, num1 and num2.

# Define two numbers for the calculation
num1 = 10
num2 = 5

# Addition: Add num1 and num2
sum_result = num1 + num2
print(f"Sum: {sum_result}")  # Print the result of addition

# Subtraction: Subtract num2 from num1
diff_result = num1 - num2
print(f"Difference: {diff_result}")  # Print the result of subtraction

# Multiplication: Multiply num1 by num2
prod_result = num1 * num2
print(f"Product: {prod_result}")  # Print the result of multiplication

# Division: Divide num1 by num2
div_result = num1 / num2
print(f"Division: {div_result}")  # Print the result of division
```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

13_Concatenating_Dataframes.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create two DataFrames to concatenate
data1 = {
    'Name': ['Alice', 'Bob'],
    'Age': [24, 27]
}
data2 = {
    'Name': ['Charlie', 'David'],
    'Age': [22, 32]
}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Concatenate the DataFrames along rows
concatenated_df = pd.concat([df1, df2], axis=0)

# Display the concatenated DataFrame
print(concatenated_df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

12_Joining_Tables.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create two DataFrames
left = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
})
right = pd.DataFrame({
    'ID': [1, 2, 4],
    'Age': [24, 27, 30]
})

# Perform a SQL-like join on the 'ID' column
joined_df = left.set_index('ID').join(right.set_index('ID'), how='inner')

# Display the joined DataFrame
print(joined_df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

10_Sorting_Values.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32]
}

df = pd.DataFrame(data)

# Sort the DataFrame by 'Age' in ascending order
sorted_df = df.sort_values(by='Age')

# Display the sorted DataFrame
print(sorted_df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

11_Dataframe_Merging.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create two DataFrames to be merged
data1 = {
    'ID': [1, 2, 3],
    'Name': ['Alice', 'Bob', 'Charlie']
}
data2 = {
    'ID': [2, 3, 4],
    'Age': [27, 22, 30]
}

df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Merge the DataFrames using a common column 'ID'
merged_df = pd.merge(df1, df2, on='ID', how='inner')

# Display the merged DataFrame
print(merged_df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

08_Missing_Values.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd
import numpy as np

# Create a DataFrame with missing values
data = {
    'Name': ['Alice', 'Bob', 'Charlie', None],
    'Age': [24, np.nan, 22, 32],
    'City': ['New York', 'Los Angeles', None, 'Houston']
}

df = pd.DataFrame(data)

# Handle missing values by filling them with default values
df_filled = df.fillna({'Name': 'Unknown', 'Age': df['Age'].mean(), 'City': 'Unknown'})

# Display the updated DataFrame
print(df_filled)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

07_Renaming_Columns.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32]
}

df = pd.DataFrame(data)

# Rename the 'Age' column to 'Years'
df = df.rename(columns={'Age': 'Years'})

# Display the updated DataFrame
print(df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

06_Deleting_Columns.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

df = pd.DataFrame(data)

# Delete the 'City' column from the DataFrame
df = df.drop(columns=['City'])

# Display the updated DataFrame
print(df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

05_Adding_Columns.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32]
}

df = pd.DataFrame(data)

# Add a new column 'City' to the DataFrame
df['City'] = ['New York', 'Los Angeles', 'Chicago', 'Houston']

# Display the updated DataFrame
print(df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

04_Filtering_Rows.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

df = pd.DataFrame(data)

# Filter rows where Age is greater than 25
filtered_df = df[df['Age'] > 25]

# Display the filtered DataFrame
print(filtered_df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

03_Dataframe_Inspection.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

df = pd.DataFrame(data)

# Inspect the structure of the DataFrame
print(df.info())  # Display basic information about DataFrame
print(df.describe())  # Display summary statistics for numerical columns

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

01_Dataframe_Creation.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame from a dictionary of data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

# Create DataFrame from dictionary
df = pd.DataFrame(data)

# Display DataFrame
print(df)

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

02_Csv_Reading.py

Run

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Read data from a CSV file using Pandas
# Make sure 'sample.csv' exists in the current directory
df = pd.read_csv('sample.csv')

# Display the first 5 rows of the DataFrame
print(df.head())

```

= . , ( ) : [ ] ' " _ - { } < > ; # ` \ / |

Back

04_Filtering_Rows.py

Edit and run code freely. Tutorial code resets on return.

```python

# Import Pandas library
import pandas as pd

# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

df = pd.DataFrame(data)

# Filter rows where Age is greater than 25
filtered_df = df[df['Age'] > 25]

# Display the filtered DataFrame
print(filtered_df)

```