Python Data Structures, File Handling, and Handling

1. Python Data Structures

Lists

Definition: A list is an ordered, mutable (changeable) collection of items that can store elements of different data types. Lists are defined using square brackets [] and elements are separated by commas. Lists allow duplicate elements and maintain insertion order.

Syntax:

python

```
list_name = [element1, element2, element3, ...]
```

Example:

python

```
# Creating a list
fruits = ["apple", "banana", "cherry", "mango"]

# Accessing elements
print(fruits[0])  # apple
print(fruits[-1])  # mango (last element)

# Modifying elements
fruits[1] = "orange"
print(fruits)  # ['apple', 'orange', 'cherry', 'mango']

# List methods
fruits.append("grapes")  # Add at end
fruits.insert(1, "kiwi")  # Insert at index 1
fruits.remove("cherry")  # Remove specific element
```

```
popped = fruits.pop()  # Remove and return last element
print(fruits)
print(f"Length: {len(fruits)}")
```

Tuples

Definition: A tuple is an ordered, immutable (unchangeable) collection of items that can store elements of different data types. Tuples are defined using parentheses () and elements are separated by commas. Once created, tuple elements cannot be modified, added, or removed.

Syntax:

python

```
tuple_name = (element1, element2, element3, ...)
```

Example:

python

```
# Creating a tuple
coordinates = (10, 20, 30)

# Accessing elements
print(coordinates[0])  # 10
print(coordinates[-1])  # 30

# Tuples are immutable
# coordinates[0] = 15  # Error: cannot modify

# Tuple operations
print(len(coordinates))  # 3
print(coordinates.count(10))  # 1
print(coordinates.index(20))  # 1
```

```
# Tuple unpacking
x, y, z = coordinates
print(f"x={x}, y={y}, z={z}")

# Single element tuple (comma required)
single = (5,)
```

Sets

Definition: A set is an unordered collection of unique elements. Sets are defined using curly braces {} or the set() function. Sets automatically remove duplicate values and do not maintain any specific order. Sets are mutable but can only contain immutable (hashable) elements.

Syntax:

```
python
set_name = {element1, element2, element3, ...}
# or
set_name = set([element1, element2, element3, ...])
```

Example:

```
python
# Creating a set
numbers = {1, 2, 3, 4, 5}
print(numbers)  # {1, 2, 3, 4, 5}

# Duplicates are automatically removed
numbers2 = {1, 2, 2, 3, 3, 4}
print(numbers2)  # {1, 2, 3, 4}
```

 # Set operations

numbers.add(6) *# Add element*

numbers.remove(3) *# Remove element*

numbers.discard(10) *# Remove if exists (no error)*

 # Set mathematical operations

set1 = {1, 2, 3, 4}

set2 = {3, 4, 5, 6}

print(set1.union(set2)) *# {1, 2, 3, 4, 5, 6}*

print(set1.intersection(set2)) *# {3, 4}*

print(set1.difference(set2)) *# {1, 2}*

Dictionaries

Definition: A dictionary is an unordered collection of key-value pairs. Dictionaries are defined using curly braces {} with key-value pairs separated by colons. Keys must be unique and immutable (strings, numbers, tuples), while values can be of any data type and can be duplicated.

Syntax:

python

dict_name = {key1: value1, key2: value2, key3: value3, ...}

Example:

python

 # Creating a dictionary

student = {

"name": "Alice",

"age": 20,

```
    "grade": "A",
    "subjects": ["Math", "Science", "English"]
}

# Accessing values
print(student["name"])  # Alice
print(student.get("age"))  # 20

# Modifying values
student["age"] = 21
student["city"] = "New York"  # Add new key-value

# Dictionary methods
print(student.keys())  # dict_keys(['name', 'age', 'grade', 'subjects', 'city'])
print(student.values())  # All values
print(student.items())  # Key-value pairs

# Iterating through dictionary
for key, value in student.items():
    print(f"{key}: {value}")

# Remove items
student.pop("city")  # Remove specific key
```

---

2. List Comprehension

Definition: List comprehension is a concise and elegant way to create new lists based on existing iterables. It allows you to write loops and conditional logic in a single line of code, making the code more readable and Pythonic. The syntax combines a for loop and optional if conditions within square brackets.

Syntax:

python

 # Basic syntax

new_list = [expression for item in iterable]

 # With condition

new_list = [expression for item in iterable if condition]

 # With if-else

new_list = [expression_if_true if condition else expression_if_false for item in iterable]

Example:

python

 # Basic list comprehension - squares of numbers

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

squares = [x ** 2 for x in numbers]

print(squares) *# [1, 4, 9, 16, 25]*

 # With condition - even numbers only

evens = [x for x in range(1, 11) if x % 2 == 0]

print(evens) *# [2, 4, 6, 8, 10]*

 # With if-else - categorize numbers

labels = ["Even" if x % 2 == 0 else "Odd" for x in range(1, 6)]

```
print(labels)  # ['Odd', 'Even', 'Odd', 'Even', 'Odd']


# String manipulation
names = ["alice", "bob", "charlie"]
upper_names = [name.upper() for name in names]
print(upper_names)  # ['ALICE', 'BOB', 'CHARLIE']


# Nested list comprehension - flatten matrix
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
flat = [num for row in matrix for num in row]
print(flat)  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

---

3. File Handling in Python

Definition: File handling is the process of creating, reading, updating, and deleting files using Python. Python provides built-in functions to work with files stored on disk. The open() function is used to open files, and it's important to close files after operations using close() or by using the with statement.

Opening and Closing Files

Syntax:

```
python
file_object = open(filename, mode)
# Perform operations
file_object.close()


# OR using 'with' statement (recommended)
with open(filename, mode) as file_object:
    # Perform operations
```

 # File automatically closes after this block

File Modes:

- 'r' - Read mode (default)
- 'w' - Write mode (overwrites existing file)
- 'a' - Append mode (adds to end of file)
- 'x' - Exclusive creation (fails if file exists)
- 'r+' - Read and write
- 'b' - Binary mode (e.g., 'rb', 'wb')

Writing to a File

Definition: Writing to a file involves opening a file in write mode ('w') or append mode ('a') and using the write() method to add content.

Example:

python

 # Writing to a file (overwrites existing content)

with open("sample.txt", "w") as file:

file.write("Hello, World!\n")

file.write("This is a sample file.\n")

file.write("Python file handling is easy.")

print("File written successfully")

Reading from a File

Definition: Reading from a file involves opening a file in read mode ('r') and using methods like read(), readline(), or readlines() to retrieve content.

Example:

python

 # Reading entire file

with open("sample.txt", "r") as file:

```
    content = file.read()
    print(content)

# Reading line by line
with open("sample.txt", "r") as file:
    for line in file:
        print(line.strip())  # strip() removes newline characters

# Reading all lines into a list
with open("sample.txt", "r") as file:
    lines = file.readlines()
    print(lines)
```

Appending to a File

Definition: Appending to a file involves opening a file in append mode ('a') which adds new content to the end without deleting existing content.

Example:

python

```
# Appending to a file
with open("sample.txt", "a") as file:
    file.write("\nThis line is appended.")
    file.write("\nAnother appended line.")

print("Content appended successfully")
```

File Operations Example

Example:

python

```
# Complete file handling example
filename = "students.txt"

# Writing student data
with open(filename, "w") as file:
    file.write("Name,Age,Grade\n")
    file.write("Alice,20,A\n")
    file.write("Bob,22,B\n")
    file.write("Charlie,21,A\n")

# Reading and displaying
print("File Contents:")
with open(filename, "r") as file:
    print(file.read())

# Appending new student
with open(filename, "a") as file:
    file.write("David,23,B\n")

# Reading line by line
print("\nReading line by line:")
with open(filename, "r") as file:
    for line in file:
        print(line.strip())
```

Checking if File Exists

Example:

```
python

import os


filename = "sample.txt"


if os.path.exists(filename):
    print(f"{filename} exists")
    with open(filename, "r") as file:
        print(file.read())
else:
    print(f"{filename} does not exist")
```

---

4. Exception Handling

Definition: Exception handling is a mechanism to handle runtime errors gracefully, preventing the program from crashing. Exceptions are errors that occur during program execution. Python uses try-except blocks to catch and handle exceptions, allowing the program to continue running or exit gracefully.

try-except Block

Definition: The try block contains code that might raise an exception. The except block catches and handles the exception if one occurs.

Syntax:

```
python

try:
    # Code that might raise an exception
    risky_code()
except ExceptionType:
    # Code to handle the exception
```

```
    handle_error()
```

Example:

python

```
# Basic try-except
try:
    number = int(input("Enter a number: "))
    result = 100 / number
    print(f"Result: {result}")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except ValueError:
    print("Error: Please enter a valid number!")
```

try-except-else Block

Definition: The else block executes only if no exception occurs in the try block. It runs after the try block completes successfully.

Syntax:

python

```
try:
    # Code that might raise an exception
    risky_code()
except ExceptionType:
    # Handle exception
    handle_error()
else:
    # Executes if no exception occurs
    success_code()
```

Example:

python

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Division by zero!")
except ValueError:
    print("Error: Invalid input!")
else:
    print(f"Division successful: {result}")
    print("No errors occurred")
```

try-except-finally Block

Definition: The finally block always executes, regardless of whether an exception occurred or not. It's typically used for cleanup operations like closing files or releasing resources.

Syntax:

python

```
try:
    # Code that might raise an exception
    risky_code()
except ExceptionType:
    # Handle exception
    handle_error()
finally:
    # Always executes
```

```
    cleanup_code()
```

Example:

python

```
try:
    file = open("data.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found!")
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    print("Cleanup: Closing file operations")
    # This always runs
```

try-except-else-finally (Complete)

Definition: A complete exception handling block that combines all components: try for risky code, except for handling errors, else for success operations, and finally for cleanup.

Syntax:

python

```
try:
    # Code that might raise an exception
    risky_code()
except ExceptionType1:
    # Handle specific exception
    handle_error1()
except ExceptionType2:
```

```
    # Handle another exception
    handle_error2()
else:
    # Executes if no exception
    success_code()
finally:
    # Always executes
    cleanup_code()
```

Example:

python

```
def divide_numbers():
    try:
        num1 = int(input("Enter numerator: "))
        num2 = int(input("Enter denominator: "))
        result = num1 / num2
    except ValueError:
        print("Error: Please enter valid integers!")
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    else:
        print(f"Result: {result}")
        print("Division completed successfully")
    finally:
        print("Operation finished")
        print("Thank you for using the calculator")
```

divide_numbers()

Common Exception Types

Example:

python

```
# Multiple exception handling
try:
    # Different types of errors
    numbers = [1, 2, 3]
    print(numbers[5])  # IndexError

except IndexError:
    print("Error: Index out of range!")
except KeyError:
    print("Error: Key not found!")
except TypeError:
    print("Error: Invalid type!")
except Exception as e:
    # Catch any other exception
    print(f"Unexpected error: {e}")
```

Raising Exceptions

Definition: The raise keyword is used to manually throw an exception when a specific condition is met.

Example:

python

```
def check_age(age):
    try:
```

```
        if age < 0:
            raise ValueError("Age cannot be negative!")
        elif age < 18:
            raise Exception("You must be 18 or older!")
        else:
            print("Access granted")
    except ValueError as e:
        print(f"ValueError: {e}")
    except Exception as e:
        print(f"Error: {e}")

check_age(15)  # Error: You must be 18 or older!
check_age(-5)  # ValueError: Age cannot be negative!
check_age(25)  # Access granted
```

File Handling with Exception Handling

Example:

python

```
filename = "data.txt"

try:
    with open(filename, "r") as file:
        content = file.read()
        print("File content:")
        print(content)
except FileNotFoundError:
    print(f"Error: {filename} not found!")
```

```
    print("Creating a new file...")
    with open(filename, "w") as file:
        file.write("Default content")
except PermissionError:
    print("Error: Permission denied!")
except Exception as e:
    print(f"Unexpected error: {e}")
else:
    print("File read successfully")
finally:
    print("File operation completed")
```
