


1. WHAT IS FILE FUNCTION IN PYTHON? WHAT ARE KEYWORDS TO CREATE AND WRITE FILE.

 In Python, file handling involves a variety of built-in functions and keywords that allow you to create, read, write and manipulate files.

File Functions in Python:

1. **open()**: Opens a file. It requires the file name and mode (e.g., read, write).
2. **read()**: Reads the content of the file.
3. **write()**: Writes content to the file.
4. **close()**: Closes the file.
5. **readline()**: Reads one line from the file.
6. **readlines()**: Reads all the lines and returns them as a list.

Keywords (Modes) to Create and Write a File:

1. **'w' (Write mode)** : Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
2. **'a' (Append mode)**: Opens a file for appending. Creates a new file if it does not exist.
3. **'x' (Exclusive creation)**: Creates a new file and raises an error if it already exists.
4. **'r+' (Read and write mode)**: Opens the file for both reading and writing.

Example: Creating and Writing to a File:

```
# Creating and writing to a file  
with open("example.txt", "w") as file:  
file.write("Hello, world!")
```

Explanation:

- `open("example.txt", "w")`: Opens example.txt in write mode.
- `with` statement: Ensures the file is properly closed after its suite finishes.
- `file.write("Hello, world!")`: Writes the string "Hello, world!" to the file.

2. EXPLAIN EXCEPTION HANDLING? WHAT IS AN ERROR IN PYTHON?

Error :

- ✚ An error in Python is an issue that arises when the program cannot execute a piece of code.
- ✚ Errors can be broadly categorized into two types:

Types Of Errors:

1. Syntax Errors:

- These occur when the Python interpreter encounters incorrect or malformed code. These errors are detected at compile time, before the code is executed.
- Example:

`print("Hello World`


- This will raise a `SyntaxError` because the string is not closed properly.

2. Exceptions:


- These are errors detected during execution. They disrupt the normal flow of the program.
- Example:

`x = 1 / 0`

- This will raise a `ZeroDivisionError` because you can not divide by zero.

 Errors help in identifying what went wrong in a program, so they can be fixed to ensure smooth execution.

Exception Handling:

 Exception handling in Python is a way to manage and respond to errors that occur during program execution. It allows you to continue running your program, even when unexpected issues arise. You use `try`, `except`, `else`, and `finally` blocks to handle exceptions gracefully.

Basic Syntax:

try:

Code that might cause an exception

x = 1 / 0

except ZeroDivisionError:

Code to handle the exception

print("Cannot divide by zero!")

else:

Code to execute if no exceptions occur

print("No errors occurred!")

finally:

Code to execute no matter what

print("This will always run.")

Common Exception Types:

- **ZeroDivisionError:** Raised when dividing by zero.
- **TypeError:** Raised when an operation or function is applied to an object of inappropriate type.
- **ValueError:** Raised when a function receives an argument of the correct type but inappropriate value.
- **KeyError:** Raised when trying to access a dictionary key that doesn't exist.

3. HOW MANY EXCEPT STATEMENTS CAN A TRY-EXCEPT BLOCK HAVE? NAME SOME BUILT-IN EXCEPTION CLASSES.

- ✚ A try-except block in Python can have multiple except statements to handle different types of exceptions.
- ✚ There's no strict limit to the number of except clauses you can have within a single try-except block.

Example:

try:

Code that might raise an exception

x = int("not a number")

except ValueError:

print("Caught a ValueError")

except TypeError:

print("Caught a TypeError")

except Exception as e:

print(f"Caught an unexpected exception: {e}")

Some Built-in Exception Classes:

1. **Exception:** The base class for all exceptions.
2. **AttributeError:** Raised when an attribute reference or assignment fails.
3. **IndexError:** Raised when a sequence subscript is out of range.

4. **KeyError:** Raised when a dictionary key is not found.
5. **ValueError:** Raised when a function receives an argument of the right type but inappropriate value.
6. **TypeError:** Raised when an operation or function is applied to an object of inappropriate type.
7. **ZeroDivisionError:** Raised when attempting to divide by zero.
8. **FileNotFoundError:** Raised when a file or directory is requested but doesn't exist.

4. WHEN WILL THE ELSE PART OF TRY-EXCEPT-ELSE BE EXECUTED?

- ✚ The else part of a try-except else block is executed only if no exceptions are raised in the try block. In other words, the code within the else block runs when the try block completes successfully without encountering any errors.

Example:

try:

Code that might raise an exception

result = 10 / 2

except ZeroDivisionError:

Code to handle division by zero

print("Caught a ZeroDivisionError")

else:

Code that runs if no exceptions are raised

print("No exceptions were raised. Result is:", result)

- ✚ In this example, because the try block doesn't raise an exception, the else block is executed, and it prints the result.
- ✚ The else block is a good place for code that should run only if the try block was successful, keeping your error handling and regular logic separate and clean.

5. CAN ONE BLOCK OF EXCEPT STATEMENTS HANDLE MULTIPLE EXCEPTION?

- ✚ Yes, one except block can handle multiple exceptions. You can specify multiple exceptions within a single except statement by enclosing them in parentheses.

Example:

try:

Code that might raise an exception

x = int("not a number")

except (ValueError, TypeError) as e:

Code to handle both ValueError and TypeError

print(f"Caught an exception: {e}")

- ✚ In this example, if the try block raises either a ValueError or a TypeError, the except block will handle it.
- ✚ This approach keeps your code concise and avoids redundancy when you want to handle multiple exceptions in the same way.

6. WHEN IS THE FINALLY BLOCK EXECUTED?

- ✚ The finally block is executed after the try and except blocks, regardless of whether an exception was raised or not. It is used to execute code that should run no matter what, such as cleanup tasks or releasing resources.

Example:

try:

Code that might raise an exception

result = 10 / 0

except ZeroDivisionError:

Code to handle the exception


```
print("Caught a ZeroDivisionError")
```

finally:

```
# Code that will always run
```

```
print("This will always be executed, whether an exception  
occurred or not.")
```

- ✚ In this example, even though a ZeroDivisionError is raised and caught, the finally block will still execute. It ensures that certain code runs no matter the outcome of the try and except blocks.

7. WHAT HAPPENS WHEN „1“== 1 IS EXECUTED?

- ✚ When `1 == 1` is executed in Python, the result is True because both sides of the equality operator are the integer 1, so they are equal.
- ✚ However, when `"1" == 1` (where "1" is a string and 1 is an integer), the result is False. This is because the equality operator `==` checks for both the value and the type. Here, the types are different (string vs. integer), so Python determines they are not equal.

8. HOW DO YOU HANDLE EXCEPTIONS WITH TRY/EXCEPT/FINALLY IN PYTHON? EXPLAIN WITH CODING SNIPPETS.

✚ Handling exceptions in Python using try, except, and finally allows you to manage errors gracefully and ensure certain code runs regardless of what happens.

Structure:

- **try:** Block where you put the code that might raise an exception.
- **except:** Block where you handle the exception(s).
- **finally:** Block that always executes, no matter what happens in the try and except blocks.

Example:

try:

Code that might raise an exception

result = 10 / 0

except ZeroDivisionError:

Code to handle the ZeroDivisionError

print("Caught a ZeroDivisionError: Cannot divide by zero.")

except Exception as e:

Code to handle any other exception

print(f"Caught an unexpected exception: {e}")

else:

Code that runs if no exceptions are raised

print("No exceptions were raised. Result is:", result)

finally:

Code that always runs

print("This will always be executed, whether an exception occurred or not.")

Explanation:

1. try Block:

result = 10 / 0

- The code tries to divide by zero, which raises a Zero DivisionError.

2. except Block:

except ZeroDivisionError:

print("Caught a ZeroDivisionError: Cannot divide by zero.")

- This block catches the ZeroDivisionError and handles it by printing a message.

3. General except Block:

except Exception as e:

print(f"Caught an unexpected exception: {e}")

- This catches any other exceptions that aren't specifically handled and prints the exception message.

4. else Block:

else:

print("No exceptions were raised. Result is:", result)

- This block runs only if no exceptions are raised in the try block.

5. finally Block:

finally:

print("This will always be executed, whether an exception occurred or not.")

- The finally block runs regardless of whether an exception occurred, ensuring cleanup or final actions.

9. WHAT ARE OOPS CONCEPTS? IS MULTIPLE INHERITANCE SUPPORTED IN JAVA.

OOP Concepts:

- ✚ Object-Oriented Programming (OOP) is a paradigm centered around objects rather than functions or logic. The primary concepts of OOP include:

1. Encapsulation:

- Bundling the data (attributes) and methods (functions) that operate on the data into a single unit, or class. It also restricts direct access to some of the object's components, which can prevent the accidental modification of data.

2. Abstraction:

- Hiding the complex implementation details and showing only the necessary features of an object. It helps in reducing programming complexity and effort.

3. Inheritance:

- A mechanism where one class (child class) inherits attributes and methods from another class (parent class). It promotes code reusability.

4. Polymorphism:

- The ability of different objects to respond uniquely to the same method call. It can be achieved through

method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

Multiple Inheritance in Java:

- **Multiple Inheritance:**

- This is the concept where a class can inherit attributes and methods from more than one parent class.

- **Java Support:**

- Java **does not support multiple inheritance** through classes to avoid the complexity and ambiguity caused by the "Diamond Problem." However, Java allows multiple inheritance through interfaces. A class can implement multiple interfaces, thus achieving multiple inheritance.

10. HOW TO DEFINE A CLASS IN PYTHON? WHAT IS SELF? GIVE AN EXAMPLE OF A PYTHON CLASS.

Defining a Class in Python

- ✚ A class in Python is a blueprint for creating objects. It defines a set of attributes and methods that the objects created from the class will have.

The self Parameter

- **self:** It is a reference to the current instance of the class. It is used to access variables and methods associated with the current object.

Example of a Python Class

- ✚ Here's a simple example to demonstrate how to define a class in Python and use self:

class Person:

def __init__(self, name, age):

self.name = name # Assign the name to the instance

self.age = age # Assign the age to the instance

def greet(self):

print(f"Hello, my name is {self.name} and I am {self.age} years old.")

Create an instance of the Person class

person1 = Person("Alice", 30)

```
# Call the greet method  
person1.greet()
```

Explanation:

1. Class Definition:

```
class Person:
```

- Defines a class named Person.

2. Constructor (__init__ method):

```
def __init__(self, name, age):
```

```
    self.name = name
```

```
    self.age = age
```

- Initializes the object's attributes (name and age) when an instance of the class is created.

3. Instance Method:

```
def greet(self):
```

```
    print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

- A method that uses self to access the instance's attributes.

4. Creating an Instance:

```
person1 = Person("Alice", 30)
```


- Creates an object (person1) of the Person class.

5. Calling a Method:

person1.greet()

- Calls the greet method on the person1 object.

11. EXPLAIN INHERITANCE IN PYTHON WITH AN EXAMPLE? WHAT IS INIT? OR WHAT IS A CONSTRUCTOR IN PYTHON?

Inheritance in Python:

- ✚ Inheritance is a core concept in OOP that allows a class (child class) to inherit attributes and methods from another class (parent class). This promotes code reusability and hierarchical classification.

Example of Inheritance:

Parent class

class Animal:

def __init__(self, name):

self.name = name

```
def speak(self):
```

```
    pass
```

```
# Child class inheriting from Animal
```

```
class Dog(Animal):
```

```
    def speak(self):
```

```
        return f'{self.name} says Woof!'
```

```
# Creating an instance of Dog
```

```
dog = Dog("Buddy")
```

```
print(dog.speak()) # Output: Buddy says Woof!
```

`__init__` Method:

- The `__init__` method is a special method in Python, known as a constructor. It is automatically called when an object is created.
- It is used to initialize the object's attributes.

Example of `__init__`:

```
class Person:
```


```
    def __init__(self, name, age):
```

```
        self.name = name
```

```
        self.age = age
```

```
def greet(self):  
    return f"Hello, my name is {self.name} and I am  
{self.age} years old."  
  
# Creating an instance of Person  
person = Person("Alice", 30)  
print(person.greet()) # Output: Hello, my name is Alice and  
I am 30 years old.
```

12. WHAT IS INSTANTIATION IN TERMS OF OOP TERMINOLOGY?

 Instantiation in Object-Oriented Programming (OOP) refers to the process of creating an instance of a class. When a class is defined, it serves as a blueprint for creating objects. Instantiation is the act of using this blueprint to create a specific object with its own unique set of attributes and methods.

Example:

```
class Car:  
    def __init__(self, make, model, year):
```

```
self.make = make  
self.model = model  
self.year = year
```

```
def display_info(self):  
    return f'{self.year} {self.make} {self.model}'
```

Instantiating an object of the Car class

```
my_car = Car("Toyota", "Camry", 2020)
```

```
print(my_car.display_info()) # Output: 2020 Toyota Camry
```

In this example:

- **Class Definition:** The Car class defines attributes and methods for car objects.
- **Instantiation:** The my_car object is created as an instance of the Car class.
- **Object Usage:** The my_car object has its own unique attributes and can use the methods defined in the Car class.

13. WHAT IS USED TO CHECK WHETHER AN OBJECT O IS AN INSTANCE OF CLASS A?

✚ To check whether an object o is an instance of a class A, you use the `isinstance()` function. This function returns True if the object is an instance or subclass instance of the specified class, and False otherwise.

Example:

```
class A:
```

```
    pass
```

```
o = A()
```

```
# Check if o is an instance of class A
```

```
if isinstance(o, A):
```

```
    print("o is an instance of class A")
```

```
else:
```

```
    print("o is not an instance of class A")
```

✚ In this example, the `isinstance(o, A)` call will return True because o is indeed an instance of class A.

14. WHAT RELATIONSHIP IS APPROPRIATE FOR COURSE AND FACULTY?

- ✚ A Course and Faculty typically share an **Association** relationship in Object-Oriented Programming (OOP) terms. This means that the Faculty is associated with a Course, but neither necessarily owns the other.

Association Example:

class Faculty:

```
def __init__(self, name):  
    self.name = name
```

class Course:

```
def __init__(self, title):  
    self.title = title  
    self.faculties = []
```

```
def add_faculty(self, faculty):  
    self.faculties.append(faculty)
```

Create instances

```
faculty1 = Faculty("Dr. Smith")
```

```
faculty2 = Faculty("Prof. Johnson")
```

```
course = Course("Advanced Python Programming")
```

```
# Associate faculties with the course
```

```
course.add_faculty(faculty1)
```

```
course.add_faculty(faculty2)
```

```
# Output the association
```

```
print(f"Course: {course.title}")
```

```
for faculty in course.faculties:
```

```
    print(f"Faculty: {faculty.name}")
```

Explanation:

- **Association:** The Course class has a list of Faculty objects, indicating which faculties are associated with the course.
- **Example:** faculty1 and faculty2 are associated with the Advanced Python Programming course.

15. WHAT RELATIONSHIP IS APPROPRIATE FOR STUDENT AND PERSON?

✚ The relationship between Student and Person is typically an **Inheritance** relationship in Object-Oriented Programming (OOP). This means that Student is a subclass of Person, inheriting its attributes and methods.

Example:

class Person:

def __init__(self, name, age):

self.name = name

self.age = age

class Student(Person):

def __init__(self, name, age, student_id):

super().__init__(name, age)

self.student_id = student_id

def study(self):

print(f"{self.name} is studying.")

Create an instance of Student

student = Student("Alice", 20, "S12345")


```
print(f"Name: {student.name}, Age: {student.age}, Student  
ID: {student.student_id}")  
student.study()
```

Explanation:

1. **Person Class:** The Person class defines common attributes like name and age.
2. **Student Class:** The Student class inherits from Person, adding its own attribute `student_id`.
3. **Inheritance:** Student inherits attributes and methods from Person, demonstrating the is-a relationship (a Student is a Person).