**AI Day 13 Notes**

**Syed Mansoor ul Hassan Bukhari**

**Python Exception Handling:**

1. **Definition**: Exception handling in Python is used to manage errors that occur during the execution of a program. It allows the program to continue running or gracefully terminate.

try:

# Code that may raise an exception

x = 10 / 0

except ZeroDivisionError:

# Code to handle the exception

print("Cannot divide by zero!")

1. **Try and Except**: The try block lets you test a block of code for errors. The except block lets you handle the error.

try:

x = int("hello")

except ValueError:

print("ValueError: Invalid literal for int()")

1. **Multiple Exceptions**: You can handle multiple exceptions by specifying multiple except blocks.

try:

x = int("hello")

except ValueError:

print("ValueError: Invalid literal for int()")

except TypeError:

print("TypeError: Unsupported operation")

1. **Else Clause**: The else block lets you execute code if no exceptions were raised.

try:

x = int("10")

except ValueError:

print("ValueError: Invalid literal for int()")

else:

print("No exceptions occurred")

1. **Finally Clause**: The finally block lets you execute code, regardless of whether an exception was raised or not.

try:

x = int("10")

except ValueError:

print("ValueError: Invalid literal for int()")

finally:

print("This will always execute")

1. **Raising Exceptions**: You can raise an exception using the raise keyword.

x = -1

if x < 0:

raise ValueError("Negative values are not allowed")

1. **Custom Exceptions**: You can define your own exceptions by creating a new class that inherits from the built-in Exception class.

class MyCustomError(Exception):

pass

try:

raise MyCustomError("This is a custom error")

except MyCustomError as e:

print(e)

1. **Exception Hierarchy**: Python has a built-in hierarchy of exceptions. The BaseException class is the base class for all built-in exceptions.

try:

x = 10 / 0

except BaseException as e:

print(f"An error occurred: {e}")

1. **Handling Multiple Exceptions in One Block**: You can handle multiple exceptions in a single except block by specifying a tuple of exception types.

try:

x = int("hello")

except (ValueError, TypeError) as e:

print(f"An error occurred: {e}")

1. **Assertions**: The assert statement is used to test if a condition is true. If the condition is false, an AssertionError is raised.

x = 10

assert x > 0, "x should be greater than 0"

1. **Logging Exceptions**: The logging module can be used to log exceptions.

import logging

try:

x = 10 / 0

except ZeroDivisionError as e:

logging.error("An error occurred", exc\_info=True)

1. **Context Managers**: The with statement can be used to handle exceptions in a context manager.

class MyContextManager:

def \_\_enter\_\_(self):

print("Entering context")

return self

def \_\_exit\_\_(self, exc\_type, exc\_value, traceback):

print("Exiting context")

if exc\_type:

print(f"An error occurred: {exc\_value}")

return True

with MyContextManager():

x = 10 / 0

1. **Ignoring Exceptions**: You can ignore exceptions by using a bare except block, but this is not recommended.

try:

x = 10 / 0

except:

pass

1. **Re-raising Exceptions**: You can re-raise an exception using the raise keyword without specifying the exception.

try:

x = 10 / 0

except ZeroDivisionError:

print("Handling ZeroDivisionError")

raise

1. **Exception Chaining**: You can chain exceptions using the from keyword.

try:

x = int("hello")

except ValueError as e:

raise RuntimeError("A runtime error occurred") from e

1. **SystemExit Exception**: The SystemExit exception is raised when the sys.exit() function is called.

import sys

try:

sys.exit()

except SystemExit:

print("SystemExit exception caught")

1. **KeyboardInterrupt Exception**: The KeyboardInterrupt exception is raised when the user interrupts the program (usually by pressing Ctrl+C).

try:

while True:

pass

except KeyboardInterrupt:

print("KeyboardInterrupt exception caught")