Python Advanced Assignment-6

Q1. Describe three applications for exception processing.

Ans.

1.Handling User Input Errors: Exception processing can handle errors that arise when users provide invalid input, such as entering a non-integer value when an integer is expected. It allows the program to catch these errors and prompt the user for valid input instead of crashing.

```
try:
    number = int(input("Enter an integer: "))
except ValueError:
    print("That's not a valid integer. Please try again.")
```

2.File Operations: Exception handling is used to manage errors when working with files, such as attempting to read a file that doesn’t exist. It allows the program to provide a meaningful error message or take alternative actions, like creating the file.

```
try:
    file = open("data.txt", "r")
except FileNotFoundError:
    print("File not found. Creating a new file.")
    file = open("data.txt", "w")
```

3. Network Connections: In applications that involve network communications, exceptions can manage connection errors, timeouts, or data transmission failures. This enables the program to attempt reconnection, log errors, or gracefully terminate the process.

```
import requests
try:
    response = requests.get("http://example.com")
except requests.ConnectionError:
    print("Failed to connect. Please check your network.")
```

Q2. What happens if you don't do something extra to treat an exception?

Ans.

If you don’t handle an exception explicitly using a `try-except` block, the program will terminate immediately when the exception occurs, and Python will display a traceback error message. This abrupt termination may not provide the user with a clear understanding of what went wrong or allow for any form of graceful recovery or cleanup.

Q3. What are your options for recovering from an exception in your script?

Ans.

1. Use a `try-except` block: The most common way to recover from exceptions is by wrapping the code that might raise an exception in a `try` block, and then handling the specific exception in the `except` block. This allows the program to continue executing or provide alternative behavior.

```
try:
    result = 10 / 0
except ZeroDivisionError:
    result = None
    print("Cannot divide by zero, setting result to None.")
```

2. Use `try-except-else`: The `else` block executes if no exception occurs in the `try` block, allowing for error-free code execution and providing an opportunity to handle cases when everything runs smoothly.

```
try:
    number = int(input("Enter a number: "))
except ValueError:
    print("Invalid input.")
else:
    print(f"Valid input: {number}")
```

3. Use `try-except-finally`: The `finally` block executes regardless of whether an exception occurs, allowing for clean-up operations such as closing files or network connections, ensuring that resources are properly released.

```
try:
    file = open("data.txt", "r")
except FileNotFoundError:
    print("File not found.")
finally:
    print("Execution complete.")
```

Q4. Describe two methods for triggering exceptions in your script.

Ans.

1. Using the `raise` statement: You can manually trigger an exception by using the `raise` statement with a specified exception type. This is useful for custom error handling or when a specific condition arises in the code that should be treated as an error.

```
age = -1
if age < 0:
    raise ValueError("Age cannot be negative.")
```

2. Using assertions with `assert`: Assertions are used to test assumptions made in the code. If the condition is `False`, an `AssertionError` is raised. This is often used for debugging or validating internal states.

```
x = 5
assert x > 0, "x must be positive"
```

Q5. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

Ans.

1. Using the `finally` block: In a `try-except-finally` construct, the `finally` block is executed no matter what, whether an exception occurred or not. It's typically used for resource cleanup, such as closing files or releasing locks.

```
try:
    file = open("data.txt", "r")
except FileNotFoundError:
    print("File not found.")
finally:
    print("This will run regardless of whether an exception occurred.")
```

2. Using the `with` statement: The `with` statement automatically manages the resource and ensures that cleanup code runs when exiting the context, even if an exception occurs. This is commonly used with file operations or threading locks.

```
with open("data.txt", "w") as file:
    file.write("Hello, world!")
# The file is automatically closed after this block, even if an error occurs
```