**PW SKILLS**

**EXCEPTION HANDLING ASSIGNMENT 2**

Q1. We use the Exception class as the base class for custom exceptions for several reasons:

- Inheritance: By inheriting from the Exception class, our custom exception inherits all the behaviors and attributes of the base Exception class. This includes methods like `\_\_str\_\_` for generating error messages and `args` for storing exception arguments.

- Uniformity: Using the Exception class ensures that our custom exceptions adhere to the same interface and conventions as built-in exceptions. This makes it easier for other developers to understand and handle our custom exceptions.

- Compatibility: Many built-in exception handling mechanisms in Python are designed to work with instances of the Exception class. By using this class as the base for custom exceptions, we ensure compatibility with existing exception handling infrastructure.

Q2. Python Exception Hierarchy:

```python

BaseException

+-- SystemExit

+-- KeyboardInterrupt

+-- GeneratorExit

+-- Exception

+-- StopIteration

+-- StopAsyncIteration

+-- ArithmeticError

| +-- ZeroDivisionError

| +-- FloatingPointError

| +-- OverflowError

| +-- ...

+-- AssertionError

+-- AttributeError

+-- ...

```

Q3. Errors defined in the ArithmeticError class:

- ArithmeticError is the base class for errors that occur during arithmetic operations. Two common errors are:

- ZeroDivisionError: Raised when attempting to divide by zero.

```python

result = 10 / 0 # ZeroDivisionError: division by zero

```

- OverflowError: Raised when the result of an arithmetic operation is too large to be represented.

```python

result = 10 \*\* 1000 # OverflowError: (34, 'Result too large')

```

Q4. LookupError class is used as the base class for errors related to key or index lookup failures. Examples include KeyError and IndexError:

- KeyError: Raised when a dictionary key is not found.

```python

my\_dict = {'a': 1, 'b': 2}

value = my\_dict['c'] # KeyError: 'c'

```

- IndexError: Raised when trying to access an index that is out of range.

```python

my\_list = [1, 2, 3]

value = my\_list[3] # IndexError: list index out of range

```

Q5. ImportError occurs when the import statement fails to find and load the requested module. ModuleNotFoundError is a subclass of ImportError that specifically indicates that the requested module could not be found.

- Example:

```python

import non\_existent\_module # ImportError: No module named 'non\_existent\_module'

```

Q6. Best practices for exception handling in Python:

- Be specific: Catch only the exceptions you expect and handle them appropriately.

- Provide informative error messages: Include details in error messages to aid debugging.

- Use try-except blocks sparingly: Only use try-except blocks where necessary to handle exceptional cases.

- Keep the try block small: Place only the code that might raise an exception inside the try block.

- Use finally for cleanup: Use the finally block to perform cleanup actions, such as closing files or releasing resources.

- Log exceptions: Use logging to record exceptions and their context for debugging and monitoring purposes.

- Consider context managers: Use context managers (`with` statement) to automatically handle resource cleanup.

- Test exception handling: Write unit tests to ensure that exception handling behaves as expected in different scenarios.