Q1.
Using the Exception class while creating a custom exception in Python is important because it provides a standardized structure and behavior for your custom exception. By inheriting from the Exception class, your custom exception gains access to all the features and functionalities provided by the base class, such as the ability to handle exceptions in a unified way using try and except blocks. Additionally, it helps in maintaining consistency and clarity in your codebase, making it easier for other developers to understand and work with your custom exceptions.

Q2. Here's a Python program to print the Python Exception Hierarchy:

```
def print_exception_hierarchy(exception_cls, depth=0):
    print(' ' * depth + exception_cls.__name__)
    for subclass in exception_cls.__subclasses__():
        print_exception_hierarchy(subclass, depth + 2)

print_exception_hierarchy(BaseException)
```

Q3. The ArithmeticError class in Python defines errors that occur during arithmetic operations. Two errors defined in the ArithmeticError class are:

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

```
try:
    result = 10 ** 1000
except OverflowError as e:
    print("OverflowError:", e)
```

ZeroDivisionError: Raised when division or modulo operation is performed with zero as the divisor.
Example:

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print("ZeroDivisionError:", e)
```

Q4. The LookupError class is used to handle errors related to looking up keys or indices in sequences like lists or dictionaries. Two examples of errors under LookupError are KeyError and IndexError:

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

```
try:
    dictionary = {"key": "value"}
    print(dictionary["non_existent_key"])
except KeyError as e:
    print("KeyError:", e)
```

IndexError: Raised when a sequence subscript is out of range.
Example:try:

```
    my_list = [1, 2, 3]
    print(my_list[3])
except IndexError as e:
    print("IndexError:", e)
```

Q5. ImportError is raised when an import statement fails to find the module, or when the imported module does not contain the requested attribute. ModuleNotFoundError is a subclass of ImportError and is raised specifically when a module could not be found.

Q6. Some best practices for exception handling in Python include:

Use Specific Exceptions: Catch specific exceptions rather than broad ones to handle errors more precisely.
Keep Exception Blocks Minimal: Keep the try block as minimal as possible to narrow down the scope of where exceptions might occur.
Handle Exceptions Gracefully: Provide meaningful error messages and handle exceptions gracefully to prevent crashes and make debugging easier.
Avoid Bare Excepts: Avoid using bare except clauses as they catch all exceptions, making it harder to diagnose specific issues.
Use finally for Cleanup: Utilize finally blocks for cleanup code that needs to run whether an exception occurs or not.

Log Exceptions: Use logging to record exceptions, including stack traces and context, for easier debugging and monitoring.
Consider Context Managers: Use context managers (with statements) for resource management to ensure resources are properly cleaned up, even in the event of an exception.
Follow PEP 8: Adhere to the Python style guide, PEP 8, for consistent and readable exception handling code.