Python Exception Handling

Python Exception Handling allows a program to gracefully deal with unexpected events (like invalid input or missing files) without crashing. Instead of terminating abruptly, Python lets you detect the problem, respond to it, and continue execution when possible.

Let's see an example to understand it better:

Basic Example: Handling Simple Exception

Here's a basic example demonstrating how to catch an exception and handle it gracefully:

```
n = 10

try:
    res = n / 0
except ZeroDivisionError:
    print("Can't be divided by zero!")
```

Output

```
Can't be divided by zero!
```

Explanation: Dividing a number by 0 raises a ZeroDivisionError. The try block contains code that may fail and except block catches the error, printing a safe message instead of stopping the program.

Difference Between Exception and Error

Errors and exceptions are both issues in a program, but they differ in severity and handling. Let's see how:

- **Error:** Serious problems in the program logic that cannot be handled. Examples include syntax errors or memory errors.
- **Exception:** Less severe problems that occur at runtime and can be managed using exception handling (e.g., invalid input, missing files).

Example: This example shows the difference between a syntax error and a runtime exception.

```
# Syntax Error (Error)
```

```
print("Hello world"  # Missing closing parenthesis


# ZeroDivisionError (Exception)

n = 10

res = n / 0
```

Explanation: A syntax error stops the code from running at all, while an exception like ZeroDivisionError occurs during execution and can be caught with exception handling.

Syntax and Usage

Python provides four main keywords for handling exceptions: try, except, else and finally each plays a unique role. Let's see syntax:

```
try:
# Code
except SomeException:
# Code
else:
# Code
finally:
# Code
```

- **try:** Runs the risky code that might cause an error.
- **except:** Catches and handles the error if one occurs.
- **else:** Executes only if no exception occurs in try.
- **finally:** Runs regardless of what happens useful for cleanup tasks like closing files.

Example: This code attempts division and handles errors gracefully using try-except-else-finally.

```
try:
    n = 0
    res = 100 / n


except ZeroDivisionError:
    print("You can't divide by zero!")
```

```
except ValueError:
    print("Enter a valid number!")

else:
    print("Result is", res)

finally:
    print("Execution complete.")
```

Output

```
You can't divide by zero!
Execution complete.
```

Explanation: try block attempts division, except blocks catch specific errors, else block would run if no errors occurred, while the finally block always runs, signaling the end of execution.

Please refer Python Built-in Exceptions for some common exceptions.

Python Catching Exceptions

When working with exceptions in Python, we can handle errors more efficiently by specifying the types of exceptions we expect. This can make code both safer and easier to debug.

1. Catching Specific Exceptions

Catching specific exceptions makes code to respond to different exception types differently. It precisely makes your code safer and easier to debug. It avoids masking bugs by only reacting to the exact problems you expect.

Example: This code handles ValueError and ZeroDivisionError with different messages.

```
try:
    x = int("str")  # This will cause ValueError
    inv = 1 / x   # Inverse calculation
```

```
except ValueError:
    print("Not Valid!")

except ZeroDivisionError:
    print("Zero has no inverse!")
```

Output

Not Valid!

Explanation: A ValueError occurs because "str" cannot be converted to an integer. If conversion had succeeded but x were 0, a ZeroDivisionError would have been caught instead.

2. Catching Multiple Exceptions

We can catch multiple exceptions in a single block if we need to handle them in the same way or we can separate them if different types of exceptions require different handling.

Example: This code attempts to convert list elements and handles ValueError, TypeError and IndexError.

```
a = ["10", "twenty", 30]  # Mixed list of integers and strings
try:
    total = int(a[0]) + int(a[1])  # 'twenty' cannot be converted to int

except (ValueError, TypeError) as e:
    print("Error", e)

except IndexError:
    print("Index out of range.")
```

Output

Error invalid literal for int() with base 10: 'twenty'

Explanation: The ValueError is raised when trying to convert "twenty" to an integer. A TypeError could occur if incompatible types were used, while IndexError would trigger if the list index was out of range.

3. Catch-All Handlers and Their Risks

Sometimes we may use a catch-all handler to catch any exception, but it can hide useful debugging info.

Example: This code tries dividing a string by a number, which causes a TypeError.

```
try:
    res = "100" / 20 # Risky operation: dividing string by number


except ArithmeticError:
    print("Arithmetic problem.")


except:
    print("Something went wrong!")
```

Output

Something went wrong!

Explanation: A TypeError occurs because you can't divide a string by a number. The bare except catches it, but this can make debugging harder since the actual error type is hidden. Use bare except only as a last-resort safety net.

Raise an Exception

We raise an exception in Python using the raise keyword followed by an instance of the exception class that we want to trigger. We can choose from built-in exceptions or define our own custom exceptions by inheriting from Python's built-in Exception class.

Basic Syntax:

raise ExceptionType("Error message")

Example: This code raises a ValueError if an invalid age is given.

```
def set(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    print(f"Age set to {age}")


try:
    set(-5)
except ValueError as e:
    print(e)
```

Output

Age cannot be negative.

Explanation: The function checks if age is invalid. If it is, it raises a ValueError. This prevents invalid states from entering the program.

Custom Exceptions

You can also create custom exceptions by defining a new class that inherits from Python's built-in Exception class. This is useful for application-specific errors. Let's see an example to understand how.

Example: This code defines a custom AgeError and uses it for validation.

```
class AgeError(Exception):
    pass


def set(age):
    if age < 0:
        raise AgeError("Age cannot be negative.")
    print(f"Age set to {age}")
```

```
try:
    set(-5)
except AgeError as e:
    print(e)
```

Output

Age cannot be negative.

Explanation: Here, AgeError is a custom exception type. This makes error messages more meaningful in larger applications.

Advantages

Below are some benefits of using exception handling:

- **Improved reliability:** Programs don’t crash on unexpected input.
- **Separation of concerns:** Error-handling code stays separate from business logic.
- **Cleaner code:** Fewer conditional checks scattered in code.
- **Helpful debugging:** Tracebacks show exactly where the problem occurred.

Disadvantages

Exception handling have some cons as well which are listed below:

- **Performance overhead:** Handling exceptions is slower than simple condition checks.
- **Added complexity:** Multiple exception types may complicate code.
- **Security risks:** Poorly handled exceptions might leak sensitive details.