Exception Handling in Python

Introduction

Exceptions are events detected during program execution that interrupt the normal flow of instructions. They are different from syntax errors which are detected before execution. Handling exceptions properly makes programs more robust and user-friendly.

1. Default Exceptions and Errors

Python provides many built-in exception types. Here are common ones with short explanations and example code.

ZeroDivisionError: Raised when division or modulo by zero occurs.

```
print(10 / 0)  # ZeroDivisionError
```

TypeError: Raised when an operation or function is applied to an object of inappropriate type.

```
print('2' + 2)  # TypeError
```

ValueError: Raised when a function receives an argument of the right type but inappropriate value.

```
int('abc')  # ValueError
```

IndexError: Raised when sequence index is out of range.

```
lst = [1,2]
print(lst[5])  # IndexError
```

KeyError: Raised when a mapping (dict) key is not found.

```
d = {}
print(d['missing'])  # KeyError
```

FileNotFoundError: Raised when trying to open a file that does not exist.

```
open('no_such_file.txt')  # FileNotFoundError
```

AttributeError: Raised when attribute reference or assignment fails.

```
x = 5
x.append(3)  # AttributeError
```

ImportError / ModuleNotFoundError: Raised when import fails.

```
import not_a_real_module  # ModuleNotFoundError
```

2. Catching Exceptions (try / except)

Use try/except to catch exceptions and respond gracefully instead of letting the program crash. Key notes:
- Catch specific exception classes where possible.
- Use `except Exception as e:` to capture the exception object.
- Avoid bare except: it hides unexpected errors.

Syntax:

```
try:
    # risky operation
except SomeException as e:
    # handle the exception
```

Examples:

```
# Example 1: handle ValueError
try:
    n = int(input("Enter a number: "))
except ValueError as e:
    print("That's not a valid integer:", e)

# Example 2: multiple except blocks
try:
    data = {"a": 1}
    print(data['b'])
    result = 10 / 0
except KeyError:
    print("Missing key")
except ZeroDivisionError:
    print("Division by zero occurred")
```

Catching multiple exception types in one clause:

```
try:
    # code that may raise ValueError or TypeError
    x = int('a')
```

```
except (ValueError, TypeError) as e:
    print("Value or Type error:", e)
```

If you need to see the full traceback for debugging, use the traceback module:

```
import traceback

try:
    1 / 0
except Exception:
    traceback.print_exc()  # prints full traceback to stderr
```

3. Raising Exceptions (raise)

You can raise exceptions intentionally with `raise`. This is useful to validate inputs or signal error conditions.

Syntax:

```
raise ExceptionType('message')
```

Examples:

```
def set_age(age):
    if age < 0:
        raise ValueError("age cannot be negative")
    return age

try:
    set_age(-3)
except ValueError as e:
    print("Caught:", e)
```

You can also re-raise the current exception inside an except block using `raise` without arguments:

```
try:
    1 / 0
except ZeroDivisionError:
    print("Got division error, re-raising")
    raise  # re-raises the same ZeroDivisionError
```

4. try / except / else / finally

Python's try statement can include optional `else` and `finally` clauses:
- `else` runs only if the try block did not raise an exception.
- `finally` always runs (useful for cleanup).

Structure:

```
try:
    # code that might raise
except SomeError:
    # handle exceptions
else:
    # runs if no exception occurred
finally:
    # always runs (cleanup)
```

Example combining all:

```
try:
    f = open("data.txt", "r")
    contents = f.read()
except FileNotFoundError:
    print("File not found; creating file and writing default content.")
    with open("data.txt", "w") as wf:
        wf.write("default")
else:
    print("File read successfully: length =", len(contents))
finally:
    try:
        f.close()
    except Exception:
        pass
    print("Cleanup done")
```

5. assert, raise, finally (differences and usage)

`assert` is a debugging aid that checks a condition and raises `AssertionError` if false. It can be disabled when Python runs with optimization (`python -O`), so don't use `assert` for data validation in production code.

Syntax:

```
assert condition, 'optional message'
```

Examples:

```
# Good for quick internal sanity checks:
def average(nums):
    assert len(nums) > 0, "nums must not be empty"
    return sum(nums) / len(nums)

# But don't rely on assert for user input validation because it can be disabled.
```

`raise` should be used to signal actual errors you want to enforce in production; `finally` is for cleanup that must run.

6. User-defined Exceptions

Create your own exception classes by inheriting from `Exception`. Attach extra attributes and override `__str__` if needed.

Basic syntax:

```
class MyError(Exception):
    pass
```

Practical example with custom attributes and __str__():

```
class NegativeValueError(Exception):
    def __init__(self, value, message="Negative values are not allowed"):
        self.value = value
        self.message = message
        super().__init__(message)
    def __str__(self):
        return f"{self.message}: {self.value}"

def process(x):
    if x < 0:
        raise NegativeValueError(x)
    return x * 2

try:
    process(-5)
except NegativeValueError as e:
    print("Custom exception caught:", e)
```

Best practices for custom exceptions:
- Inherit from `Exception` (not `BaseException`).
- Give clear names ending with `Error`.
- Keep them simple and document when they are raised.

7. Exception chaining (`raise ... from ...`)

Python supports explicit exception chaining. Use `raise NewExc(...) from original_exc` to indicate that the new exception was caused by the original one. This preserves the original traceback and clarifies the cause in logs.

Example:

```
def read_int_from_file(path):
    try:
        with open(path, 'r') as f:
            return int(f.read().strip())
    except FileNotFoundError as e:
        # raise a higher-level error while preserving cause
        raise RuntimeError("Configuration file missing") from e
```

```
try:
    read_int_from_file("no_file.txt")
except RuntimeError as e:
    import traceback
    traceback.print_exc()
```

8. Logging exceptions and traceback

For production code, log exceptions instead of printing. Use the `logging` module and `traceback` if you need full details.

```
import logging, traceback

logging.basicConfig(level=logging.ERROR)

try:
    1/0
except Exception as e:
    logging.error("An error occurred: %s", e)
    logging.error(traceback.format_exc())
```

9. Real-world example: robust file processing with retries

Example that attempts to read a file multiple times and falls back to default content if it fails:

```
import time

def read_with_retries(path, retries=3, delay=1):
    for attempt in range(1, retries + 1):
        try:
            with open(path, 'r') as f:
                return f.read()
        except FileNotFoundError as e:
            if attempt == retries:
                raise RuntimeError("Failed after retries") from e
            time.sleep(delay)  # wait before retrying

try:
    contents = read_with_retries("maybe.txt", retries=2, delay=0.5)
except RuntimeError as e:
    print("Could not read file:", e)
```