FREE EDITION · NOTES + 3 SAMPLE MCQS

CUET · COMPUTER SCIENCE · CLASS XII · CODE 308

Exception Handling in Python

CUET unit: Exception Handling in Python

By **UniDrill** · NCERT-grounded study material

WWW.UNIDRILL.IN

📌 Snapshot

- Exception handling is a structured approach to runtime errors in Python programs.
- Syntax errors are detected before execution; exceptions are raised during execution of syntactically correct code.
- Python has a built-in exception hierarchy (12 common exceptions); raise exceptions programmatically using `raise` and `assert`.
- The core construct is `try...except...else...finally`, which CUET tests both in code-tracing and definition-based questions.
- Knowing which clause executes under which condition (error vs. no error vs. always) is the single most-tested skill here.

📖 Detailed Notes

2.1 Core concepts

- **Introduction to errors:** A Python program may fail to execute at all, or may execute but produce unexpected output. This happens due to syntax errors, runtime errors, or logical errors. In Python, exceptions are errors triggered automatically; they can also be forcefully triggered and handled through program code. (NCERT §1.1, p. 1)
- **Syntax errors (parsing errors):** Detected when the programmer has not followed the rules of the language while writing the program. The interpreter does not execute the program until syntax errors are rectified. In shell mode, Python displays the error name and a brief description. In script mode, a dialog box with the error name and description is shown. (NCERT §1.2, pp. 1–2)
- **Exceptions:** Even if a statement is syntactically correct, an error may arise during execution — for example, dividing by zero or opening a non-existent file. Such errors disrupt normal execution and are called exceptions. An exception is a Python object that represents an error; when raised, it must be handled by the programmer to prevent abnormal termination. SyntaxError shown in Figures 1.1 and 1.3 is itself an exception, but all other exceptions are generated only when the program is syntactically correct. (NCERT §1.3, p. 3)

- **Built-in exceptions:** Python's standard library provides an extensive collection of built-in exceptions for commonly occurring errors. On occurrence, the interpreter displays the raised exception name and reason; 12 common built-in exceptions are listed in Table 1.1: SyntaxError, ValueError, IOError, KeyboardInterrupt, ImportError, EOFError, ZeroDivisionError, IndexError, NameError, IndentationError, TypeError, and OverFlowError. Programmers can also create user-defined (custom) exceptions. (NCERT §1.4, pp. 3–4)
- **Raising exceptions — `raise` statement:** Each time an error is detected, the Python interpreter raises (throws) an exception. Programmers can also forcefully raise exceptions using `raise` and `assert`. Raising an exception interrupts normal program flow and jumps to the exception handler code. Once raised, no further statement in the current block is executed. Syntax: `raise exception-name[(optional argument)]`. The optional argument is typically a string message displayed when the exception is raised. When `raise IndexError` is used explicitly, a stack Traceback is displayed showing the sequence of function calls. (NCERT §1.5, §1.5.1, pp. 4–5)
- `assert` **statement:** Used to test an expression in program code. If the expression evaluates to False, an `AssertionError` exception is raised. Used at the beginning of a function or after a function call to check for valid input. Syntax: `assert Expression[, arguments]`. On passing a negative value in Program 1-1, `AssertionError: OOPS... Negative Number` is raised and subsequent statements are not executed. (NCERT §1.5.2, pp. 6–7)
- **Need for exception handling:** Exception handling is used in Python and most programming languages (C++, Java, Ruby). It captures runtime errors and prevents the program from crashing. Python categorises exceptions into distinct types; exception handlers separate error-detection code from main logic; the compiler/interpreter tracks the exact error position; both user-defined and built-in exceptions can be handled. (NCERT §1.6.1, p. 7)
- **Process of handling exceptions:** When an error occurs, the Python interpreter creates an exception object containing error type, file name, and position. This object is handed to the runtime system — the act of creating and passing it is called throwing an exception. The runtime system searches for an exception handler in the current method; if not found, it searches the call stack in reverse order (forwarding the exception). Executing a suitable handler is called catching the exception. If no handler is found in the entire call stack, the program terminates. (NCERT §1.6.2, pp. 7–9)
- **Catching exceptions — `try...except` block:** Suspicious code is placed inside a `try` block; every `try` block is followed by one or more `except` blocks containing handler code. When an exception occurs in the `try` block, execution stops and control transfers to the matching `except` block. After the `except` block executes, the statement following the `try...except` construct runs normally. Multiple `except` blocks can handle different exceptions for a single `try` block. An `except` clause

without a named exception acts as a catch-all and should be placed last. (NCERT §1.6.3, pp. 9–12)

- `try...except...else` **clause:** An optional `else` block can follow the `except` blocks. The `else` block executes only if no exception is raised in the `try` block; if an exception is raised, the `else` block is skipped. (NCERT §1.6.4, pp. 12–13)
- `finally` **clause:** The `try` statement can have an optional `finally` clause whose statements are always executed regardless of whether an exception occurred in the `try` block. It is a common practice to use `finally` with file operations to ensure the file object is closed. `finally` must be placed at the end of the `try` clause, after all `except` blocks and the `else` block. If an exception is not handled by any `except` clause, it is re-raised after the `finally` block executes. (NCERT §1.7, §1.7.1, pp. 13–15)

2.2 Definitions to memorise

Term	Definition	Page
Syntax Error	An error detected when the rules of the programming language are not followed while writing the program; also called a parsing error.	1
Exception	A Python object that represents an error occurring during program execution even when the statement is syntactically correct.	3
Built-in Exception	Commonly occurring exceptions defined in Python's standard library/interpreter that provide standardised handling for typical errors.	3
Exception Handler	Code designed to execute when a specific exception is raised; separates error-correction logic from main program logic.	7
Throwing an Exception	The process of creating an exception object and handing it over to the runtime system when an error occurs.	8
Catching an Exception	The process of executing a suitable exception handler found in the call stack.	8–9
Call Stack	The entire list of methods searched in reverse hierarchical order when an exception handler is not found in the current method.	8
raise Statement	A Python statement used to forcefully throw an exception; syntax: `raise exception-name[(optional argument)]`.	5
assert Statement	A Python statement that tests an expression; raises `AssertionError` if the expression evaluates to False.	6
finally Clause	An optional clause in a try statement whose code always executes regardless of whether an exception was raised or not.	13
User-defined Exception	A custom exception created by a programmer to suit specific requirements, as opposed to a built-in exception.	4

Term	Definition	Page
`try` block	Block of code in which suspicious statements that may raise exceptions are placed	9
`except` block	Block following `try` that handles a specific exception	10
`else` block	Optional clause after all `except` blocks that runs only when no exception was raised	12
ZeroDivisionError	Built-in exception raised when the denominator in a division is zero	4
ValueError	Built-in exception raised when a function receives an argument of correct type but inappropriate value	4
TypeError	Built-in exception raised when an operator is applied to a value of incorrect data type	4
IndexError	Built-in exception raised when a sequence subscript is out of range	4
NameError	Built-in exception raised when a local or global name is not defined	4
IOError	Built-in exception raised when an I/O operation fails (e.g., file not found)	4
ImportError	Built-in exception raised when an import statement cannot find the module	4
OverflowError	Built-in exception raised when arithmetic result exceeds numeric type limits	4
KeyboardInterrupt	Built-in exception raised when the user interrupts execution (Delete/Esc)	4
EOFError	Built-in exception raised when input() hits end-of-file unexpectedly	4
AssertionError	Built-in exception raised when an assert expression evaluates to False	6
Stack Traceback	Sequence of function calls printed when an exception propagates uncaught	5

2.3 Diagrams / processes to remember

- **Figure 1.1 (p. 2):** Screenshot showing a SyntaxError in Python shell mode — "Missing parentheses in call to 'print'" — illustrating how the interpreter reports a syntax error with a brief explanation and suggestion.
- **Figure 1.4 (p. 4):** Screenshot demonstrating three built-in exceptions raised in a single shell session: `ZeroDivisionError` (print(50/0)), `NameError` (print(var+40)), and `TypeError` (10+'5') — key for identifying which exception corresponds to which error condition.

- **Figure 1.8 (p. 9):** Flowchart of the exception handling process — the most important diagram. Flow: error encountered → create exception object → exception raised → runtime searches current method → if found, execute handler (catching); if not found, search call stack in reverse (forwarding) → if still not found, program terminates.
- **Figure 1.12 (p. 13):** Output of Program 1-5 showing `try...except...else` behaviour — when no exception occurs, the `else` block prints the quotient.
- **Figure 1.13 (p. 15):** Output of Program 1-7 demonstrating `finally` clause recovery — "OVER AND OUT" prints even when a `ValueError` is raised, then the exception is re-raised.

2.4 Common confusions / NTA trap points

- **SyntaxError is also an exception:** Students often think SyntaxError and exceptions are mutually exclusive. SyntaxError (shown in Figs. 1.1 and 1.3) is also an exception. However, all exceptions other than SyntaxError are generated only when the program is syntactically correct. (p. 3)
- `else` **vs.** `finally` **execution conditions:** The `else` block runs only when NO exception is raised in `try`. The `finally` block runs ALWAYS — whether or not an exception occurred. NTA may swap these in options.
- `finally` **does not suppress unhandled exceptions:** If an exception is not caught by any `except` clause, `finally` executes first and then the exception is re-raised. The `finally` clause does NOT terminate (handle) the exception — unlike `except`.
- `raise` **stops all further execution in the block:** After a `raise` statement, no subsequent statement in the current block executes (the `print("NO EXECUTION")` in Figure 1.6 is never reached). NTA may ask what output a code snippet produces when `raise` appears mid-block.
- **Bare** `except:` **clause order (NCERT § 1.6.3, p. 11).** An `except` clause without a named exception must always be the LAST except block; placing it before specific handlers would shadow them.
- **One** `except` **matches per try (NCERT § 1.6.3, p. 11).** Only the first matching `except` block runs; the others are skipped.
- **Errors before** `try` **are not caught (NCERT § 1.6.3, p. 9).** A statement before the `try` block that raises an error is NOT caught — the `try` must surround the suspicious code.
- `finally` **ALWAYS runs (NCERT § 1.7, p. 13).** Including after a successful `try`, after an `except` handler, and even when an unhandled exception propagates.
- **Unhandled exceptions terminate the program (NCERT § 1.6.2, p. 8).** If no matching handler is found anywhere in the call stack, Python prints the traceback and exits.

- `assert` ≠ `if` **(NCERT §1.5.2, p. 6).** `assert` is for debugging/validation — it raises an AssertionError. `if` is for normal flow.
- `raise` **with no exception name re-raises the current exception (Python convention; NCERT mentions raise with name).** NCERT focuses on `raise ExceptionName`.

🎯 Practice MCQs

Q1. Which of the following statements about syntax errors in Python is correct?

A. Syntax errors are raised only when a file cannot be opened.

B. Syntax errors are also known as parsing errors and prevent program execution until rectified.

C. Syntax errors are raised after the program has been successfully compiled and is running.

D. Syntax errors are detected only in script mode, not in shell mode.

Q2. Consider the following code: ```python numbers = [40, 50, 60, 70] length = 10 if length > len(numbers): raise IndexError print("NO EXECUTION") else: print(length) ``` What will be the output?

A. 10

B. NO EXECUTION

C. IndexError is raised and "NO EXECUTION" is printed.

D. IndexError is raised and "NO EXECUTION" is not printed.

Q3. Which of the following correctly describes the `assert` statement in Python?

A. It is used to catch exceptions in the try block.

B. It tests an expression and raises `AssertionError` if the expression evaluates to False.

C. It tests an expression and raises `ValueError` if the expression evaluates to False.

D. It is used only at the end of a function to verify output values.

PYQ Alignment

Exception handling appears regularly in CUET Computer Science (308) question papers; typically 1–2 direct MCQs test identification of built-in exception names and their triggers (from Table 1.1), and 1–2 code-tracing questions require students to predict output of `try...except...else...finally` constructs — particularly distinguishing which block executes when an exception is raised versus when it is not. See PYQ archive for Computer Science.