What Is an Exception & the try-except Block

AiCE Software Engineering Foundations — Python Programming (Intermediate)

Every program that interacts with the real world will eventually encounter something unexpected — a user who types a letter where a number was expected, a file that's been moved or deleted, a network that drops mid-request. Exception handling is how you plan for those moments and respond to them gracefully, instead of crashing. This lesson opens the topic: what an exception actually is, and the try-except block that catches one.

1. What Is an Exception?

An exception is an error that occurs while a program is running. Unlike syntax errors — which Python catches before the program even starts — exceptions happen during execution, when something unexpected goes wrong.

You've already encountered several exceptions throughout this program:

ValueError — raised when a function receives a value of the right type but the wrong content. int('twenty-two') raises ValueError because 'twenty-two' can't be converted to an integer.

TypeError — raised when an operation is applied to an incompatible type. 'hello' + 5 raises TypeError because you can't add a string and an integer.

ZeroDivisionError — raised when you try to divide by zero.

FileNotFoundError — raised when you try to open a file that doesn't exist.

IndexError — raised when you access a list index that doesn't exist.

KeyError — raised when you access a dictionary key that doesn't exist.

> **Without Handling, Any of These Crash Your Program**
> *Without exception handling, any one of these errors will crash your program — printing a traceback and stopping immediately. Exception handling gives you the ability to catch these errors, respond to them, and keep the program running.*

2. The try-except Block

The try-except block is Python's mechanism for catching exceptions. The code you want to protect goes inside the try block. If an exception occurs, Python jumps to the except block instead of crashing:

```
try:
    # Code that might raise an exception
except ExceptionType:
    # Code that runs if that exception occurs
```

Compare code with and without exception handling:

```
# WITH exception handling — handles bad input gracefully
try:
    num = int(input("Enter a number: "))
    print(f"Double: {num * 2}")
except ValueError:
    print("That was not a valid number. Please enter digits
only.")

print("Program continues after the try-except block.")
```

Running this with a valid number like 15 produces:

```
Enter a number: 15
Double: 30
Program continues after the try-except block.
```

Running it again with invalid input like 'twenty' produces:

```
Enter a number: twenty
That was not a valid number. Please enter digits only.
Program continues after the try-except block.
```

With the try-except block in place, the program does not crash on invalid input. It catches the ValueError, prints a friendly message, and continues executing. The final print statement runs in both cases — the program recovers from the error and moves on.

> **The Exception Does Not Propagate**
> *When Python catches an exception in an except block, the exception is considered handled. Execution continues normally after the try-except block. Without the except block, the exception propagates upward, crashing the function and ultimately the program.*

3. Catching Specific Exceptions

Catching every possible exception with a bare except: clause is bad practice — it hides bugs and makes programs harder to debug. Always catch the specific exception type you expect:

```
def safe_divide(num1, num2):
    try:
        return num1 / num2
    except ZeroDivisionError:
        return "Error: Cannot divide by zero."

def safe_convert(value):
    try:
        return int(value)
    except ValueError:
        return f"Error: Cannot convert '{value}' to an
integer."

def safe_list_access(items, index):
    try:
        return items[index]
    except IndexError:
        return f"Error: Index {index} is out of range."

print(safe_divide(10, 2))
print(safe_divide(10, 0))
print(safe_convert("42"))
```

```
print(safe_convert("forty-two"))
```

Each function catches only the exception it expects. If an unexpected exception occurs — one not listed in the except clause — it is not caught and propagates normally, which is the correct behavior. You want to see unexpected errors, not hide them.

4. Catching Multiple Exception Types

A single try block can have multiple except clauses — one for each exception type you anticipate. Python checks them in order and executes the first matching one:

```
def process_input(value, index, data):
    try:
        number = int(value)
        item = data[index]
        return number / len(item)
    except ValueError:
        return "Error: Value could not be converted to
integer."
    except IndexError:
        return "Error: Index out of range."
    except ZeroDivisionError:
        return "Error: Division by zero."
```

You can also catch multiple exception types in a single except clause using a tuple:

```
try:
    result = int(input("Number: ")) / int(input("Divisor: "))
except (ValueError, ZeroDivisionError) as e:
    print(f"Input error: {e}")
```

The as e part gives you access to the exception object — using it in an f-string gives you Python's built-in error message, which is often informative enough to show directly to the user.

Key Takeaways

An exception is a runtime error — distinct from a syntax error caught before the program starts — and the try-except block is how Python lets you catch one instead of crashing. Always catch the specific exception type you expect rather than a bare except:, since hiding unexpected errors makes bugs harder to find. Multiple except clauses handle different exception types from the same try block, checked in order, and a single except clause can catch several types at once using a tuple. The next lesson adds else and finally to the structure, and shows how to raise exceptions of your own to enforce rules in the code you write.