**https://youtu.be/V\_NXT2-QIlE?si=QkMvjAAp2jY9b4Xv**

**What is Exception Handling?**

Exception handling in Python is a mechanism that allows you to manage and respond to **errors** and **unexpected events** that occur during the execution of your program. These events, called **exceptions**, disrupt the normal flow of the program. Instead of letting your program crash, exception handling allows you to gracefully handle the error, provide feedback to the user, and continue the program's execution.

**The try, except, else, and finally Blocks**

Exception handling is built around four main keywords: try, except, else, and finally.

* **try**: This block of code contains the statements that might raise an exception. You put the risky code here.
* **except**: This block is executed only if an exception occurs within the try block. You can specify which type of exception to handle.
* **else**: The else block is executed if and only if the try block completes successfully without raising any exceptions.
* **finally**: This block is always executed, regardless of whether an exception occurred or not. It's often used for cleanup actions, like closing files or network connections.

Here's an example that uses all four:

Python

def safe\_division(numerator, denominator):

try:

# Code that might raise a ZeroDivisionError

result = numerator / denominator

except ZeroDivisionError:

print("Error: Cannot divide by zero.")

return None

except TypeError:

print("Error: Inputs must be numbers.")

return None

else:

print("Division successful.")

return result

finally:

print("This block always runs.")

print(safe\_division(10, 2)) # Successful case

print("-" \* 20)

print(safe\_division(10, 0)) # Exception case

print("-" \* 20)

print(safe\_division("10", 2)) # Different exception case

**Key Concepts**

* **Handling Specific Exceptions**: It's good practice to handle specific exceptions rather than a generic except block. This makes your code more precise and prevents you from accidentally catching errors you didn't expect. You can handle multiple exceptions with a single except block using a tuple: except (TypeError, ValueError):.
* **Raising Exceptions**: You can manually raise an exception in your code using the raise keyword. This is useful for signaling an error condition that your function cannot handle.

Python

def check\_age(age):

if age < 0:

raise ValueError("Age cannot be negative.")

print(f"Age is {age}.")

* **Polymorphism in Exceptions**: All built-in exceptions in Python are classes that inherit from BaseException. This means you can handle a parent exception class to catch all its child classes. For example, except Exception: will catch most standard errors. However, this is generally not recommended as it can hide bugs.

Exception handling is a powerful tool for creating robust and reliable Python programs. It allows your code to anticipate problems and respond to them in a controlled and predictable way.

Errors in Python can be broadly categorized into two types: **Syntax Errors** and **Exceptions**.

**1. Syntax Errors**

**Syntax errors**, also known as **parsing errors**, are the most basic type of error. They occur when the Python interpreter cannot understand your code because it violates the rules of the Python language. They are typically detected before the program even begins to run. Think of them as grammatical mistakes in a human language.

**Common causes of syntax errors:**

* **Missing or mismatched parentheses, brackets, or braces:** For example, print("hello" instead of print("hello").
* **Incorrect use of keywords:** For example, using whil instead of while.
* **Incorrect indentation:** Python relies on indentation to define code blocks, so an IndentationError is a common syntax error.
* **Missing colons:** For example, if condition instead of if condition:.

**Example:**

Python

# SyntaxError: EOL (end of line) while scanning string literal

print("Hello, world!)

The interpreter stops and points to the line where the error occurred, often with a ^ symbol to indicate the exact location.

**2. Exceptions**

**Exceptions** are errors that occur during the **execution** of a program. Unlike syntax errors, the code itself is syntactically correct, but something goes wrong when it's being run. This could be due to invalid input, a file not being found, or a mathematical operation that's impossible to perform.

When an exception occurs, the normal flow of the program is interrupted, and Python raises an exception object. If this exception is not handled, the program will terminate and show a **traceback**—a detailed report of what went wrong.

Python has many built-in exception types, and it's also possible to create custom ones. Here are some common examples:

* **TypeError**: Raised when an operation is performed on an object of an inappropriate type.

Python

"hello" + 5 # TypeError: can only concatenate str (not "int") to str

* **NameError**: Raised when a local or global name is not found.

Python

print(my\_variable) # NameError: name 'my\_variable' is not defined

* **ZeroDivisionError**: Raised when the second operand of a division or modulo operation is zero.

Python

10 / 0 # ZeroDivisionError: division by zero

* **FileNotFoundError**: Raised when a file or directory is requested but does not exist.

Python

with open("non\_existent\_file.txt") as f:

pass # FileNotFoundError: [Errno 2] No such file or directory

* **IndexError**: Raised when a sequence (like a list or tuple) index is out of range.

Python

my\_list = [1, 2, 3]

print(my\_list[5]) # IndexError: list index out of range

* **ValueError**: Raised when a function receives an argument that has the right type but an invalid value.

Python

int("abc") # ValueError: invalid literal for int() with base 10: 'abc'

Handling these exceptions using try...except blocks is crucial for writing robust and reliable code that doesn't crash unexpectedly.