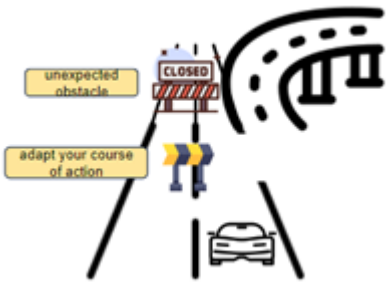
Exception handling

Exception Handling in Python is a way to handle errors and unexpected situations in a more manageable manner.

As programmers, we've all experienced it before. You create a fantastic piece of code, carefully testing each part, but then, out of nowhere, a surprising error pops up and stops your program. It can be really frustrating, right? Don't worry!

Exception handling is like a safety net for your code. It helps you deal with unexpected problems in a smooth and helpful way, keeping your program running without crashing and providing useful feedback to users. It's like a superhero that saves your code from potential disasters!

Imagine you are driving to work, and suddenly you come across a road closure due to construction work. Instead of panicking or giving up, you follow the detour signs, taking an alternative route to reach your destination safely and on time. This real-life example illustrates the concept of exception handling, where you encounter an unexpected obstacle (road closure) and quickly adapt your course of action (following the detour) to overcome the issue and continue with your journey smoothly.



Similarly, in programming, exception handling allows the code to detect and respond to unforeseen errors or issues, ensuring the

program can handle unexpected situations gracefully and continue functioning without crashing or causing disruptions.

Some common scenarios where we need to apply exception handling:

File Handling

: File handling is about working with files on a computer, like opening, reading, or saving information. Sometimes, we may face unexpected issues like the file not being there, someone else using it, or the file being damaged. Exception handling helps us handle these unexpected situations smoothly, so our program doesn't crash and can keep working properly. It's like having a backup plan for when something goes wrong with the files we want to use in our program.

User Input

: When users provide input to our program, they might make mistakes or leave important fields blank. With exception handling, we can handle these errors and ask the user to provide valid input instead of crashing.

Mathematical Errors

: Sometimes, our program may try to do math that doesn't make sense, like dividing by zero or dealing with really large or small numbers. Exception handling helps us avoid these problems.

In each of these cases, exception handling allows the program to handle errors gracefully, prevent crashes, and provide better error messages or alternative solutions to the user, improving the overall reliability and user experience of the program.

Here are some of the most common types of exceptions in Python:

The first one is a Syntax error.

1. **SyntaxError**

   : This error occurs when the Python interpreter encounters a mistake in the structure of your code. It typically happens due to misspelled keywords, missing colons, unbalanced parentheses, or other syntax-related issues.

Let’s understand this with an example:

- print("Hello World")

The provided example illustrates a common syntax error in Python, which occurs when a parenthesis is not closed properly. The provided code attempts to print "Hello World" using the print function, but it contains a syntax error. The issue is a missing closing parenthesis after the string. In Python, functions require parentheses to enclose their arguments. The corrected version is:

- print("Hello World")

1. **TypeError**

   : A TypeError in Python occurs when you try to perform an operation or use a function with a value incompatible with the expected type, such as trying to add a string to an integer.

To better understand, let me demonstrate with an example.

- result = int("5") + 3

Here, we have converted 5 from a string to an integer and hence the operation is now working without an error.

1. **ValueError**

   : This error occurs when a function or method receives an argument of the correct type but with an invalid value. For example, if you're trying to convert a string to a number, but the string doesn't represent a valid number, Python may raise a ValueError.

Let's understand this with a simple example of adding two numbers entered by the user:

- num1 = int(input("Enter the first number: "))

num2 = int(input("Enter the second number: "))

result = num1+num2

print(f"The result of sum is: result")

ValueError: invalid literal for int() with base 10: 'six' and show it for all the errors

Consequently, the program crashes and stops executing, and the following lines of code are not executed.

1. **IndexError:**

   Imagine you have a list of items, like a row of boxes. Each box has a number on it, starting from 0. If you ask Python to give you the item in a box that doesn't exist (for example, asking for the item in box number 5 when you only have 3 boxes), Python will get confused and tell you about an IndexError. So if we are trying to access an index that is outside the valid range for a list, tuple, or other sequence types then Index error occurs.

Let’s use a basic example to help you understand.

- my_list = [10, 20, 30]

Try to access an item in a box that doesn't exist (index 3)

value = my_list[3] # Try to access an item in a box that doesn't exist (index 3)

In this example, my_list has three boxes with indices 0, 1, and 2. However, we're trying to access the item in the box with index 3, which doesn't exist. Python will raise an IndexError because we are asking for something beyond the valid range of indices for the list. It's like asking for the content of a box that hasn't been created, causing Python to get confused and report an IndexError.

1. **ZeroDivisionError**

   : This error occurs when you attempt to divide a number by zero. Here's an easy example:

- result = 10 / 0

In this example, we are attempting to divide the number 10 by zero. Since dividing by zero is not a valid mathematical operation, Python will raise a ZeroDivisionError.

1. **IndentationError:**

   This error occurs when there is an issue with the indentation of your code. Python relies on indentation to define code blocks, and inconsistent indentation can lead to errors. For example:

- if True:

print("Indented incorrectly") # not indented properly.

This will result in an IndentationError since the print statement is not indented properly.

1. **AttributeError:**

   AttributeError error occurs when you try to access an attribute or method that doesn't exist for a given object. In simple terms, it occurs when you're trying to use a feature or do something with a program, but the program doesn't recognize or have that particular feature you're asking for.

Let's consider a simple example:

Suppose you have a variable 'name', but accidentally try to use it as if it's a function.

- name = "John"

name.length # This will raise an AttributeError

In this code snippet, we have a variable ‘name’ storing the string "John." Subsequently, we attempt to retrieve the length of the name using the .length attribute, assuming it exists.

An error would be raised which is an AttributeError. This occurs because the string class in Python uses the len() function to get the

length, not the .length attribute. Therefore, when we try to access a non-existent attribute like

.length

, Python raises an AttributeError indicating that the attribute does not exist

(All the errors mentioned above show the error while coding)

These are just a few examples of the many types of exceptions that can occur in Python. It's essential for Python developers to be familiar with these errors as they play a crucial role in debugging and improving code quality.

The primary construct for handling exceptions in Python is the

try-except block

. This block allows you to enclose code that might raise an exception within a

try block

, and then specify how to handle the exception in the corresponding

except block

. Here's a simple example:

- try:

Code that may raise an exception

result = 10 / 0

except:

Handle any exception

print("An unexpected error occurred.")

In this example:

The code inside the try block is where an exception might occur. Here, we attempt to perform a division operation (10 / 0), which will raise a ZeroDivisionError since dividing by zero is undefined.

If an exception occurs within the try block, the control is transferred to the corresponding except block. In this case, the except block is catching any exception without specifying a particular type. It serves as a generic catch-all for any exception that might be raised.

In the event of an exception, the program will execute the code inside this block, printing the message "An unexpected error occurred."

This generic approach can be useful during the development and debugging phases when you want to identify and address any unexpected errors quickly. However, in a live working environment, it's often beneficial to catch specific types of exceptions to provide more targeted and meaningful error messages, aiding in quicker issue resolution.

As mentioned earlier, catching specific exceptions allows for a more structured and controlled approach to error handling. The next step might involve modifying the code to handle the specific exception, as shown in the earlier example with except ZeroDivisionError.

Handling a Specific Exception

Now, let's consider an example where the except block is catching a specific exception type, which is ZeroDivisionError:

- try:

Code that may raise an exception

result = 10 / 0

except ZeroDivisionError:

Handle the specific exception which is division by zero

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

In this case, the except block is tailored to handle only ZeroDivisionError. If any other type of exception occurs, this block will not catch it.

Multiple except Blocks

Now, let's extend the example to demonstrate how the program could handle a scenario where the user enters 'zero' as input, leading

to a ValueError. We'll use multiple except blocks to handle different types of exceptions:

- try:

Code that may raise an exception

user_input = input("Enter a number: ")

result = 10 / int(user_input)

except ZeroDivisionError:

Handle division by zero

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

except ValueError:

Handle invalid input (e.g., user enters 'zero')

print("Error: Invalid input. Please enter a valid number.")

except:

Catch any other unexpected exceptions

print("An unexpected error occurred.")

(Enter zero for Enter a number)

In this extended example:

In try Block the user is prompted to enter a number using the input() function. The code then attempts to perform a division operation (10 / int(user_input)).

In this example, if the user enters '0', a ZeroDivisionError will be raised, and this block will handle the specific exception by printing "Error: Cannot divide by zero."

But If the user enters something other than a valid number (e.g., 'zero'), then the code attempts to convert the string "zero" to an integer, a ValueError will be raised, and the appropriate except block will be executed.

The last except block will handle any other unexpected exceptions that might occur during execution. It prints a generic error message, "An unexpected error occurred."

By incorporating these multiple except blocks, the program becomes more robust and user-friendly. It provides distinct error messages based on the type of exception encountered, allowing for a more informative response to the user and facilitating easier debugging during development.

In addition to using multiple except blocks to handle different types of exceptions, you can further enhance the robustness and user-friendliness of your program by incorporating the else block within the try-except structure.

The else block is executed only if no exceptions are raised in the corresponding try block. Here's the general structure of a try-except block with an else clause:

- try:

Code that may raise an exception

except SomeException:

Code to handle the specific exception

else:

- # Code to be executed if no exceptions are raised

If the code inside the try block does not raise any exceptions, the code inside the else block will be executed. This is useful for situations where you want to perform certain actions only if the try block is complete without any exceptions.

Here's an example to illustrate the use of the else clause in a try-except block:

- try:

x = int(input("Enter a number: "))

result = 10 / x

except ValueError:

print("Invalid input. Please enter a valid number.")

except ZeroDivisionError:

print("Cannot divide by zero.")

else:

print("Division result:", result)

In this example, if the user enters a valid number and doesn't cause a ValueError or a ZeroDivisionError, the code in the else block will be executed, printing the division result. If an exception occurs, the appropriate except block will handle it, and the code in the else block won't be executed.

And so, using the else clause in conjunction with the try-except structure allows you to organize your code to separate the normal flow from exception handling, making it more readable and maintaining a clean structure.

When discussing the try-except structure with multiple except blocks and an else clause, it's essential also to introduce the final block.

The final block provides a way to specify a set of statements that will be executed regardless of whether an exception occurred.

Let's look at an example that includes the try block with multiple except blocks, an else block, and the final block:

- try:

x = int(input("Enter a number: "))

result = 10 / x

except ValueError:

print("ValueError: Invalid input. Please enter a valid number.")

except ZeroDivisionError:

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

else:

print("Division result:", result)

finally:

print("Finally block: This always gets executed.")

Now let’s suppose the user enters a 0.

Output:

- Enter a number: 0

ZeroDivisionError: Cannot divide by zero.

Finally block: This always gets executed.

In this case, entering zero leads to a ZeroDivisionError, and the corresponding except ZeroDivisionError block is executed. The "Finally block: This always gets executed." statement ensures that the finally block is also executed.

In the second scenario, suppose the user enters a non-zero number (e.g., 2), and the code in the else block is executed, printing the division result. As before, the finally block also gets executed.

And this is the output we get:

- Enter a number: 2

Division result: 5.0

Finally block: This always gets executed.

In conclusion, exception handling is a crucial programming aspect that enables developers to write robust and reliable code. It acts as a safety net, allowing programs to gracefully handle unexpected errors and disruptions, preventing crashes and improving the overall user experience.