09 - Exception handling

Understanding Exceptions

Understanding Exceptions

What Are Exceptions?

- Exceptions are errors that occur during the execution of a program (at runtime).
- They disrupt the normal flow of code and need to be handled to prevent crashes.

Understanding Exceptions

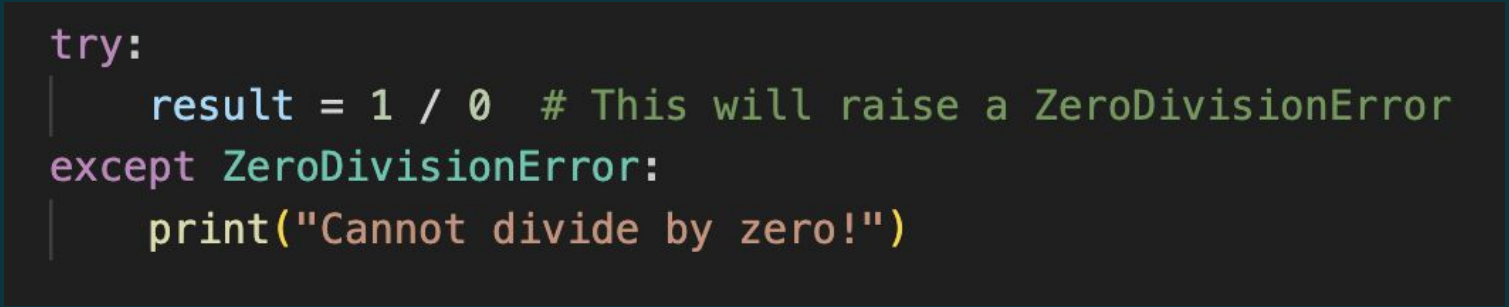
Common Exceptions:

- **ZeroDivisionError**: Division by zero.
- **IndexError**: Raised when you try to access an invalid index in a list.
- **KeyError:** Happens when you try to access a non-existent key in a dictionary.
- **TypeError:** Raised when you apply an operation or function to an object of an inappropriate type.
- **ValueError**: Raised when a function receives an argument of the correct type but an inappropriate value.
- **FileNotFoundError**: Trying to open a file that doesn't exist.

Handling Exceptions: Try, except

Handling Exceptions: Try, except

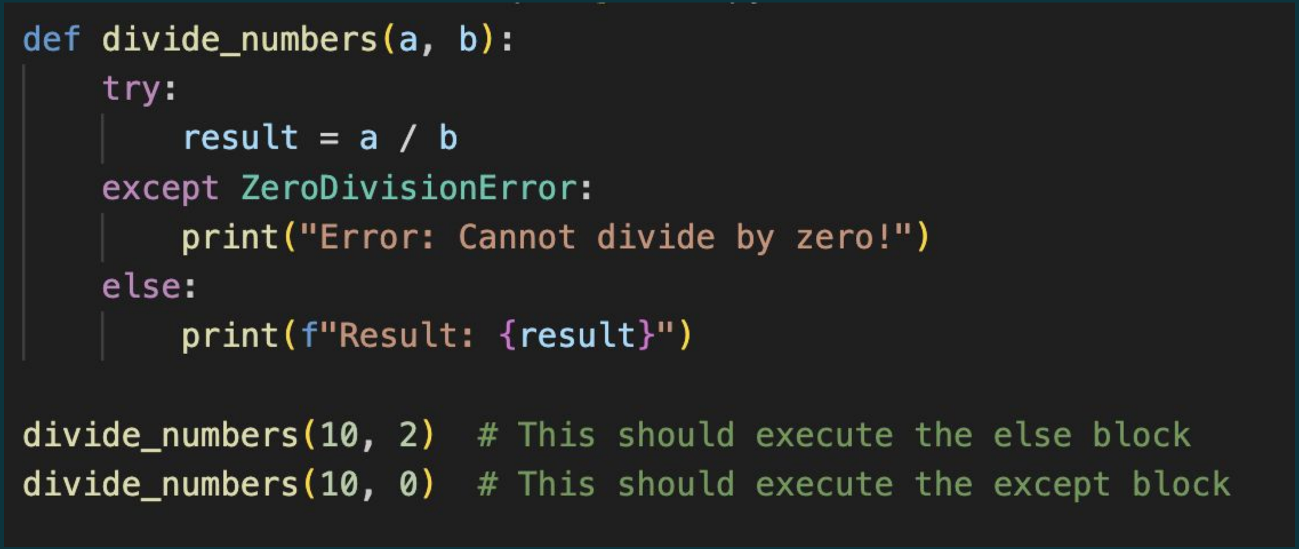
The try and except blocks in Python are used for handling exceptions (errors) that may occur during the execution of a program.



```python
try:
    result = 1 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Try, except, else

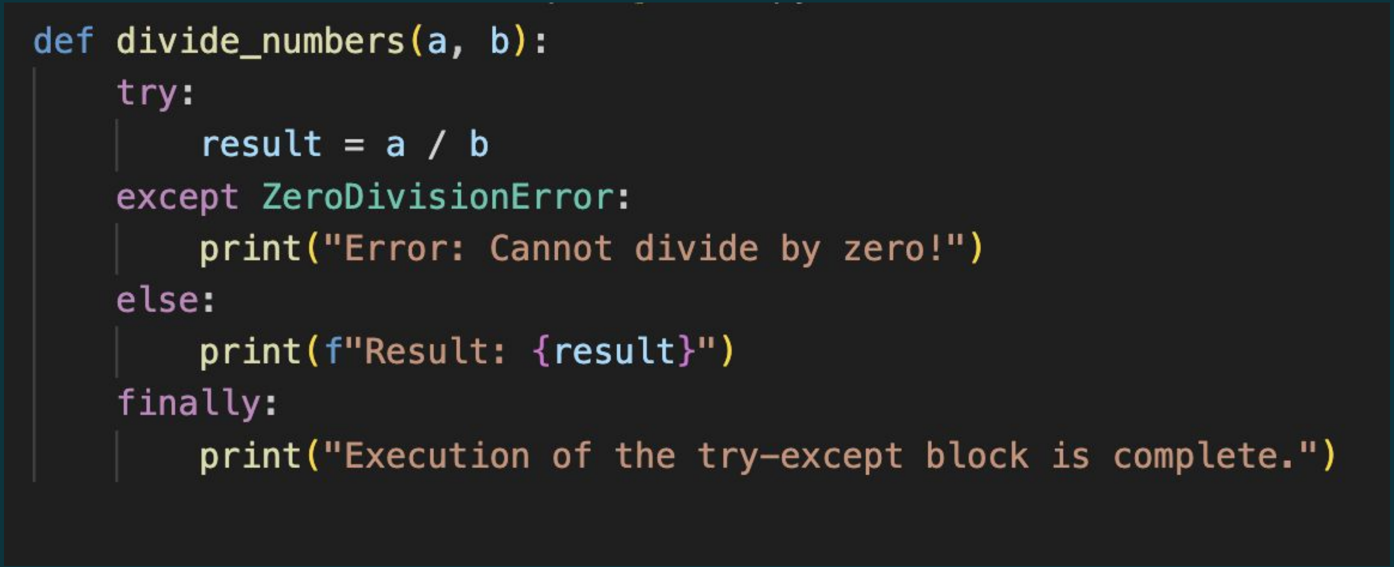
The else block in a try statement is executed only if no exceptions are raised in the try block.



```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    else:
        print(f"Result: {result}")

divide_numbers(10, 2)  # This should execute the else block
divide_numbers(10, 0)  # This should execute the except block
```

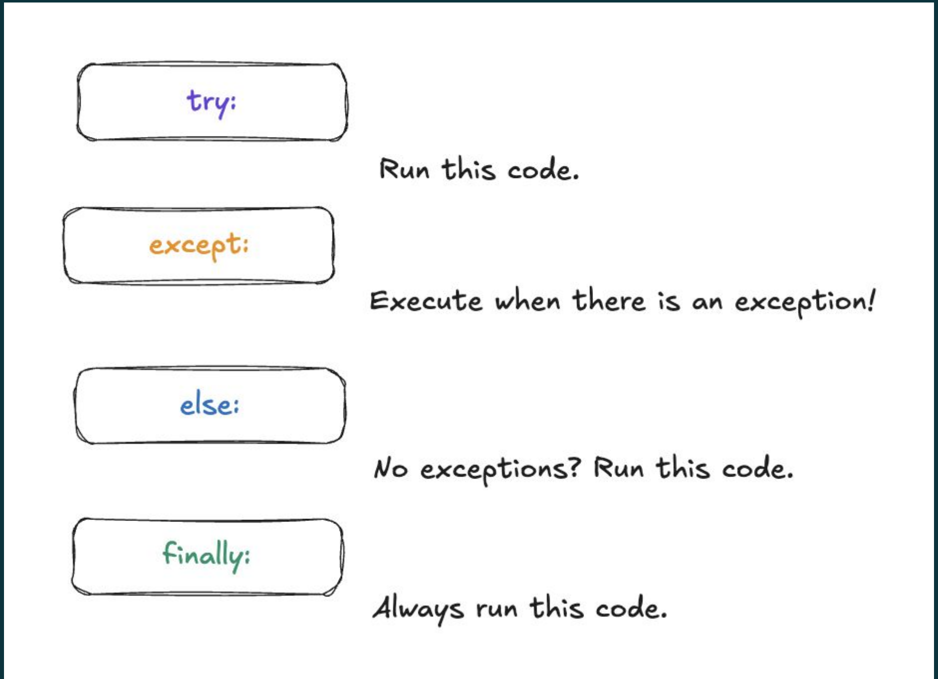
Try, except, else, finally

The finally block is particularly useful for cleanup actions that must be executed under all circumstances, such as closing a file or releasing resources.



```python
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
    else:
        print(f"Result: {result}")
    finally:
        print("Execution of the try-except block is complete.")
```

Summary



try:

Run this code.

except:

Execute when there is an exception!

else:

No exceptions? Run this code.

finally:

Always run this code.

Custom Exceptions

Custom Exceptions

Custom exceptions are user-defined classes that extend Python’s built-in **Exception** class.

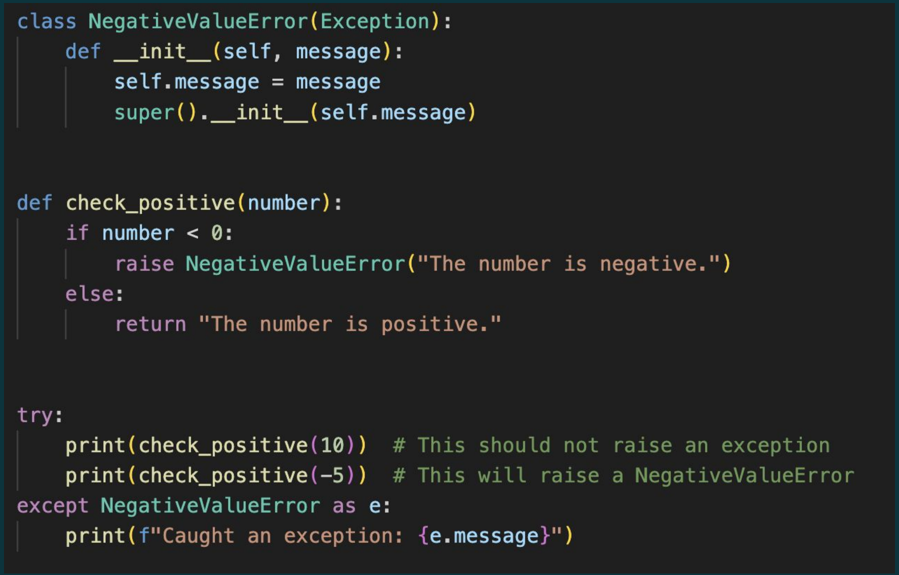
They allow you to create more meaningful and specific error messages tailored to your application’s needs.

Benefits of custom exceptions:

- Improves code readability and maintainability by defining exceptions specific to your application's domain.
- Make error handling more precise and informative.

Defining a Custom Exception

1. Inherit from the built-in **Exception** class.
2. **Optionally**, override the __init__ method to accept custom parameters.
3. Use the **raise** keyword to trigger the custom exception in your code.
4. Use **try** and **except** blocks to catch and handle the custom exception.



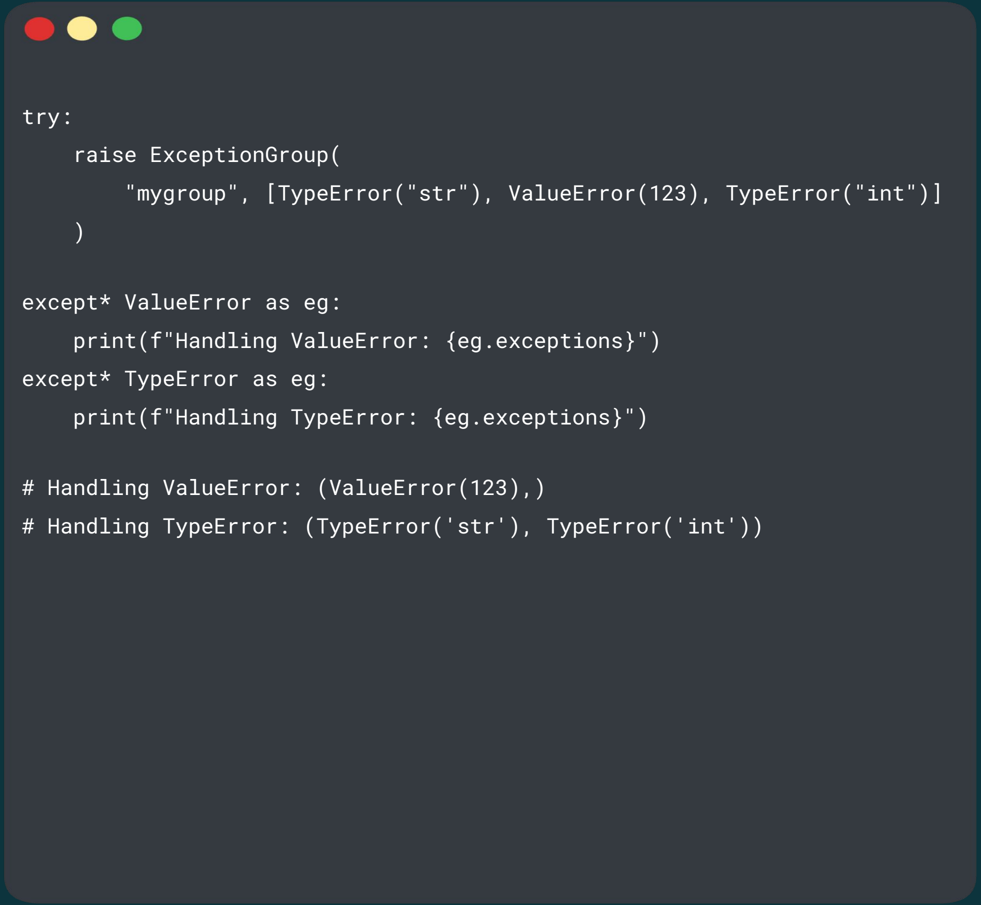
```python
class NegativeValueError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)


def check_positive(number):
    if number < 0:
        raise NegativeValueError("The number is negative.")
    else:
        return "The number is positive."


try:
    print(check_positive(10))  # This should not raise an exception
    print(check_positive(-5))  # This will raise a NegativeValueError
except NegativeValueError as e:
    print(f"Caught an exception: {e.message}")
```

Exception Groups

Exception Groups

- It provides a way to group unrelated exceptions together, and it comes with a new **except*** syntax for handling them.
- A collection or group of different kinds of Exceptions.
- Exception Groups introduced in **Python 3.11**

`ExceptionGroup` Example



```
try:
    raise ExceptionGroup(
        "mygroup", [TypeError("str"), ValueError(123), TypeError("int")]
    )

except* ValueError as eg:
    print(f"Handling ValueError: {eg.exceptions}")
except* TypeError as eg:
    print(f"Handling TypeError: {eg.exceptions}")

# Handling ValueError: (ValueError(123),)
# Handling TypeError: (TypeError('str'), TypeError('int'))
```

Conclusion

- What is exception handling
  - Handle exceptions to maintain program stability.
  - Use try, except, else, and finally for robust error management.
  - Raise exceptions to enforce error conditions explicitly.
- Custom Exceptions
- Exception Groups