ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Errors in programming are inevitable, but Python provides powerful tools to manage and handle them gracefully.

Effective error handling ensures that your program can recover from unexpected situations without crashing. Debugging, on the other hand, helps you identify and fix issues in your code.

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Syntax Errors: Occur when the Python interpreter encounters code that doesn’t follow the correct syntax.

Example:

```python
if True
    print("Hello")  # Missing colon at the end of the if statement
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Fix: Ensure the syntax is correct.

```python
if True:
    print("Hello")
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Runtime Errors: Occur during program execution and can cause the program to terminate unexpectedly.

- **Example:**

```python
print(1 / 0)  # Division by zero is a runtime error
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Logical Errors: Occur when the code runs without crashing but produces incorrect results. These errors are harder to detect and require debugging.

- **Example:**

```python
def multiply(a, b):
    return a + b  # This should be a * b
```

ikSaan.com

Error Handling and Debugging in Python

Exception Handling with try, except, and finally

Python provides a mechanism to handle runtime errors using try-except blocks.

The try block contains the code that might throw an exception, and the except block catches and handles the exception.

ikSaan.com

Error Handling and Debugging in Python

Exception Handling with try, except, and finally

Basic Syntax:

```python
try:
    # code that may raise an exception
except SomeException:
    # code to handle the exception
```

ikSaan.com

Error Handling and Debugging in Python

Exception Handling with try, except, and finally

Example:

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print(result)
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Handling Multiple Exceptions: You can catch multiple exceptions using multiple except blocks or a tuple of exceptions.

```python
try:
    result = 10 / 0
except (ValueError, ZeroDivisionError) as e:
    print(f"An error occurred: {e}")
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

- **else: Executed if no exceptions occur.**
- **finally: Always executed, whether an exception occurs or not. Typically used for cleanup actions like closing files or network connections.**

```python
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful, result is:", result)
finally:
    print("This will always execute.")
```

ikSaan.com

Error Handling and Debugging in Python

Error Handling and Debugging in Python

Example with File Handling:

```python
try:
    file = open("myfile.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()  # Ensures the file is closed even if an error occurs
```

ikSaan.com

Error Handling and Debugging in Python

Raising Exceptions

You can manually raise exceptions using the raise keyword if you want to signal that an error has occurred in your code.

```python
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    return age

try:
    check_age(-1)
except ValueError as e:
    print(e)  # Output: Age cannot be negative!
```

ikSaan.com

Error Handling and Debugging in Python

Custom Exceptions

You can define your own exceptions by creating a class that inherits from Python's built-in Exception class.

```python
class NegativeNumberError(Exception):
    """Custom exception for negative numbers."""
    pass

def check_positive(number):
    if number < 0:
        raise NegativeNumberError("Negative numbers are not allowed!")

try:
    check_positive(-5)
except NegativeNumberError as e:
    print(e)  # Output: Negative numbers are not allowed!
```

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Effective debugging helps identify the source of errors in your code.

Print Debugging

The simplest way to debug is by printing intermediate values to track the flow of the program.

python

```python
def sum_numbers(numbers):
    total = 0
    for number in numbers:
        print(f"Adding {number} to total")  # Debugging output
        total += number
    return total
```

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Using the assert Statement

The assert statement is used for debugging and testing assumptions. If the condition evaluates to False, Python raises an AssertionError.

```python
def divide(a, b):
    assert b != 0, "b cannot be zero!"
    return a / b
```

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Using a Debugger (pdb)

Python has a built-in debugger called pdb that allows you to step through code, inspect variables, and understand program flow.

Basic Commands:

- n: Execute the next line of code.
- s: Step into a function.
- p variable: Print the value of a variable.
- c: Continue execution until the next breakpoint.

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Example:

```python
import pdb

def calculate_area(length, width):
    pdb.set_trace()  # Sets a breakpoint here
    return length * width

area = calculate_area(5, 0)
```

Running this script will enter an interactive debugging session where you can inspect variables and control the flow of execution.

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Using Logging for Debugging

The logging module is more flexible than print() statements because you can control what level of information gets displayed or saved to a file.

ikSaan.com

Error Handling and Debugging in Python

Debugging Techniques

Using Logging for Debugging

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def multiply(a, b):
    logging.debug(f"Multiplying {a} by {b}")
    return a * b

result = multiply(5, 10)
logging.info(f"Result: {result}")
```

ikSaan.com

Error Handling and Debugging in Python

Common Exceptions

Here are some common built-in exceptions and their typical use cases:

- ValueError: Raised when a function receives an argument of the right type but an inappropriate value.
  - Example: int("abc") raises a ValueError because "abc" is not a valid integer.
- TypeError: Raised when an operation is performed on an inappropriate type.
  - Example: 1 + "2" raises a TypeError because you cannot add an integer and a string.
- IndexError: Raised when an index is out of range.
  - Example: Accessing an index in a list that doesn’t exist.
- KeyError: Raised when trying to access a dictionary key that doesn't exist.
  - Example: my_dict['nonexistent_key'].
- ZeroDivisionError: Raised when dividing by zero.
  - Example: 10 / 0.

ikSaan.com

Error Handling and Debugging in Python

Best Practices for Error Handling and Debugging

Use Specific Exceptions: Catch the most specific exception possible. Catching all exceptions can make it harder to identify the real issue.

```python
try:
    # risky operation
except ValueError:
    # handle ValueError specifically
```

ikSaan.com

Error Handling and Debugging in Python

Best Practices for Error Handling and Debugging

Handle Exceptions Gracefully: Avoid crashing the program, and provide meaningful messages or actions when an error occurs.

```python
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("That was not a valid number.")
```

ikSaan.com

Error Handling and Debugging in Python

Best Practices for Error Handling and Debugging

Don’t Silence Exceptions: Avoid using a bare except clause unless absolutely necessary, as it can hide real issues in the code.

```python
try:
    # risky operation
except Exception as e:
    print(f"An error occurred: {e}")
```

ikSaan.com

Error Handling and Debugging in Python

Best Practices for Error Handling and Debugging

Use finally for Cleanup: Always close resources (e.g., files, network connections) in the finally block to ensure they are properly released.

python

```python
try:
    file = open("data.txt")
    # Perform operations
finally:
    file.close()  # Always close the file
```