In the name of Allah

Hello Everybody

Let’s get started!

Error Handling

Error handling in Python is a programming technique that allows you to gracefully handle exceptions or errors that might occur during the execution of your code. Exceptions are unexpected events or conditions that can disrupt the normal flow of your program.

Type of built-in error in Python

Python has a number of built-in exceptions, such as the well-known errors SyntaxError, NameError, and TypeError.

SyntaxError

This error occurs when there is a problem with the syntax of your Python code, such as a missing colon at the end of a statement, an indentation error, or a misspelled keyword.

```
# SyntaxError
# Missing colon at the end of a statement
if True
    print("Hello, world!")

File "C:\Users\Matin\Desktop\test.py",
line 3
    if True
           ^
SyntaxError: expected ':'
```

IndentationError

This is a subtype of SyntaxError and occurs when there are issues with the indentation of your code, such as inconsistent use of tabs and spaces.

```
# IndentationError
# Inconsistent use of tabs and spaces
def indentation_error():
    print("Indented with spaces")
    print("Indented with tabs")
```

```
File "C:\Users\Matin\Desktop\test.py",
line 5
    print("Indented with tabs")
TabError: inconsistent use of tabs and
spaces in indentation
```

NameError

This error occurs when you try to use a variable or a function that is not defined in the current scope.

```
# NameError
# Using an undefined variable
print(undefined_variable)
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    print(undefined_variable)
NameError: name 'undefined_variable' is
not defined
```

TypeError

This error occurs when an operation or function is applied to an object of inappropriate type. For example, trying to divide a string by an integer would raise a TypeError.

```python
# TypeError
# Performing an inappropriate operation
result = 10 / "2"
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    result = 10 / "2"
TypeError: unsupported operand type(s) for
/: 'int' and 'str'
```

ZeroDivisionError

Raised when you attempt to divide by zero, which is mathematically undefined.

```python
# ZeroDivisionError
# Dividing by zero
result = 5 / 0
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    result = 5 / 0
ZeroDivisionError: division by zero
```

ValueError

Raised when a function receives an argument of the correct type but an inappropriate value. For example, trying to convert a non-numeric string to an integer using int().

```
# ValueError
# Converting an inappropriate value
value = int("abc")
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    value = int("abc")
ValueError: invalid literal for int() with
base 10: 'abc'
```

IndexError

Raised when you try to access an index that is out of range in a sequence (e.g., a list or a string).

```python
# IndexError
# Accessing an index out of range
my_list = [1, 2, 3]
element = my_list[5]
```

```
File "C:\Users\Matin\Desktop\test.py",
line 4, in <module>
    element = my_list[5]
IndexError: list index out of range
```

KeyError

Raised when you attempt to access a dictionary key that does not exist.

```
# KeyError
# Accessing a non-existent key in a
dictionary
my_dict = {'name': 'Alice'}
value = my_dict['age']
```

```
File "C:\Users\Matin\Desktop\test.py",
line 4, in <module>
    value = my_dict['age']
KeyError: 'age'
```

MemoryError

Raised when the Python interpreter runs out of memory.

```
# MemoryError
# Running out of memory
big_list = [0] * 100**9  # This will
consume a massive amount of memory
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    big_list = [0] * 100**9  # This will
consume a massive amount of memory
MemoryError
```

FileNotFoundError

Raised when you attempt to open a file that doesn't exist.

```python
# FileNotFoundError
# Trying to open a non-existent file
with open('non_existent_file.txt ',
'r') as file:
    content = file.read()
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    with open('non_existent_file.txt',
'r') as file:
FileNotFoundError: [Errno 2] No such file
or directory: 'non_existent_file.txt'
```

ImportError

Raised when there is an issue with importing a module. This can include the module not existing or having an import error within the module itself.

```
# ImportError
# Importing a module that doesn't exist
import non_existent_module
```

```
File "C:\Users\Matin\Desktop\test.py",
line 3, in <module>
    import non_existent_module
ModuleNotFoundError: No module named
'non_existent_module'
```

Raising error in Python

You can raise an error in Python using the raise statement. The raise statement allows you to raise exceptions or errors explicitly in your code when a specific condition or situation warrants it.

Raise Error

raise: The keyword used to raise an exception.

SomeException: The specific exception type you want to raise. This can be any built-in exception type (e.g., ValueError, TypeError, CustomError) or a custom exception you've defined by creating a new exception class.

"Error message": An optional error message that provides additional information about the error. This message will be included in the exception's error description.

```
raise SomeException("Error message")
```

Dealing with errors

Dealing with errors in Python involves using error handling mechanisms to handle exceptions gracefully, allowing your program to continue executing or providing informative feedback to the user when unexpected situations occur.

Try and Except Blocks

Use try and except blocks to handle exceptions. The try block contains the code that might raise an exception, and the except block contains code to handle the exception if it occurs.

```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to handle the exception
```

Try and Except Blocks

Use try and except blocks to handle exceptions. The try block contains the code that might raise an exception, and the except block contains code to handle the exception if it occurs.

```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to handle the exception

# For example, to handle a
ZeroDivisionError
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Multiple Except Blocks

You can use multiple except blocks to handle different types of exceptions or execute different code for each exception type.

```
try:
    # Code that might raise an exception
except ZeroDivisionError:
    # Code to handle a ZeroDivisionError
except ValueError:
    # Code to handle a ValueError
```

Generic Exception Handling

You can use a generic except block without specifying a particular exception type to catch any unexpected exceptions. However, this should be used sparingly, and specific exception handling is recommended for robust error management.

```
try:
    # Code that might raise an exception
except:
    # Code to handle any exception
```

Finally Block

You can use a finally block to specify code that should be executed regardless of whether an exception was raised or not. This is often used for cleanup operations.

```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to handle the exception
finally:
    # Code to execute regardless of
whether an exception was raised
```

Finally Block

You can use a finally block to specify code that should be executed regardless of whether an exception was raised or not. This is often used for cleanup operations.

```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to handle the exception
finally:
    # Code to execute regardless of
whether an exception was raised
```

Else Block

You can use an else block to specify code that should be executed when the code within the try block runs successfully without raising any exceptions.

```
try:
    # Code that might raise an exception
except SomeExceptionType:
    # Code to handle the exception
else:
    # Code to execute when no exception was raised
finally:
    # Code to execute regardless of whether an exception was raised
```

Example

In this example, the code in the try block attempts to get user input and perform a division operation. If the user enters invalid input or tries to divide by zero, the corresponding except block is executed. If no exception occurs, the else block is executed. The finally block always executes, whether or not an exception occurred, and is typically used for cleanup operations.

```python
try:
    dividend = int(input("Enter a number:"))
    result = 10 / dividend
except ZeroDivisionError:
    print("You cannot divide by zero.")
except ValueError:
    print("Invalid input. Please enter a
valid number.")
else:
    print(f"The result is {result}")
finally:
    print("Execution complete, whether an
exception occurred or not.")
```

That’s all for today!