Exception Handling

What is Exception Handling?

Exception handling in Python allows a program to deal with runtime errors (exceptions) and prevent crashes. It helps in gracefully handling errors and ensuring the program continues execution where possible.

Errors in Python

There are two types of errors in Python:

1. **Syntax Errors**: Errors due to incorrect syntax, such as missing colons, incorrect indentation, or misspelling keywords.

   ```
   print 'hello world'  # SyntaxError: Missing parentheses in call to 'print'
   ```

2. **Exceptions**: Errors that occur during execution, such as dividing by zero or accessing an invalid index.

Common Built-in Exceptions

Some common exceptions in Python include:

- **ZeroDivisionError**: Division by zero.
- **TypeError**: Operation on incompatible data types.
- **ValueError**: Incorrect value passed to a function.
- **IndexError**: Accessing an invalid list index.

  ```
  L = [1,2,3]
  print(L[100])  # IndexError
  ```

- **KeyError**: Accessing a non-existent key in a dictionary.

```
d = {'name': 'Nitish'}
print(d['age'])  # KeyError
```

- **ModuleNotFoundError**: Importing a non-existent module.

```
import mathi  # ModuleNotFoundError
```

- **FileNotFoundError**: Trying to open a non-existent file.
- **ImportError**: Failure to import a module.

Try-Except Block

The `try` block contains code that might raise an exception, while the `except` block handles it.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
    print("Result:", result)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input! Please enter a number.")
```

Handling Multiple Exceptions

Multiple exceptions can be handled using multiple `except` blocks or by grouping them.

```
try:
    x = int("abc")
except (ValueError, TypeError) as e:
    print("An error occurred:", e)
```

Using Else and Finally

- **else**: Executes if no exception occurs.
- **finally**: Always executes, useful for resource cleanup.

```
try:
    f = open("test.txt", "r")
    content = f.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
finally:
    if 'f' in locals():
        f.close()
```

Raising Exceptions

The `raise` keyword allows manual exception raising.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or above!")
    print("Valid age.")

check_age(16)  # Raises ValueError
```

Custom Exceptions

Custom exceptions can be created by inheriting from the `Exception` class.

```
class MyCustomError(Exception):
    pass

try:
    raise MyCustomError("Something went wrong!")
```

```
except MyCustomError as e:
    print("Caught custom exception:", e)
```

Summary

- Use `try-except` to handle exceptions and prevent program crashes.
- Use `else` for code execution when no error occurs.
- Use `finally` for cleanup operations.
- Raise custom exceptions using `raise` and define your own exception classes.

Proper exception handling improves program reliability and user experience.

Python has a variety of built-in exceptions. Here's a comprehensive list:

Built-in Exceptions in Python

1. **ArithmeticError** – Base class for arithmetic-related errors.
    - `ZeroDivisionError` – Division or modulo by zero.
    - `OverflowError` – Numeric operation result too large to be represented.
    - `FloatingPointError` – Floating-point operation failure (rarely raised).
2. **AssertionError** – Raised when an `assert` statement fails.
3. **AttributeError** – Raised when an invalid attribute reference occurs.
4. **BufferError** – Raised when buffer-related operations fail.
5. **EOFError** – Raised when `input()` reaches end-of-file condition (EOF).
6. **ImportError** – Raised when an import statement fails.
    - `ModuleNotFoundError` – Raised when a module could not be found.
7. **IndexError** – Raised when trying to access an invalid index in a sequence.
8. **KeyError** – Raised when trying to access a non-existent dictionary key.

9. **KeyboardInterrupt** – Raised when the user interrupts program execution (Ctrl+C).
10. **MemoryError** – Raised when an operation runs out of memory.
11. **NameError** – Raised when a variable is not found in the local or global scope.
    - `UnboundLocalError` – Raised when a local variable is referenced before being assigned.
12. **NotImplementedError** – Raised when an abstract method requires overriding.
13. **OSError** – Base class for operating system-related errors.
    - `FileNotFoundError` – Raised when a file or directory is requested but does not exist.
    - `PermissionError` – Raised when a file operation lacks proper permission.
    - `BlockingIOError` – Raised when an operation would block on an I/O stream.
    - `TimeoutError` – Raised when a system function times out.
    - `IsADirectoryError` – Raised when a file operation is requested on a directory.
14. **ReferenceError** – Raised when a weak reference proxy is used after the referent is garbage collected.
15. **RuntimeError** – Raised when an error does not belong to any other category.
    - `RecursionError` – Raised when the maximum recursion depth is exceeded.
16. **StopIteration** – Raised to signal the end of an iterator.
17. **SyntaxError** – Raised when incorrect syntax is encountered.
    - `IndentationError` – Raised when there's an issue with indentation.
        - `TabError` – Raised when indentation uses inconsistent tabs and spaces.
18. **SystemError** – Raised when an internal error occurs in the interpreter.
19. **TypeError** – Raised when an operation is performed on an incorrect data type.
20. **ValueError** – Raised when an operation receives an argument with the correct type but invalid value.
21. **UnicodeError** – Base class for Unicode-related errors.

- `UnicodeEncodeError` – Raised when encoding a Unicode string fails.
- `UnicodeDecodeError` – Raised when decoding a Unicode string fails.
- `UnicodeTranslateError` – Raised when translating a Unicode string fails.

22. **Warning Categories** – These are used for generating warnings.
    - `DeprecationWarning`
    - `FutureWarning`
    - `UserWarning`
    - `SyntaxWarning`
    - `RuntimeWarning`
    - `PendingDeprecationWarning`
    - `ResourceWarning`

These exceptions help handle various errors that may arise in Python programs. Would you like examples for any specific ones?

Here are examples for each built-in exception in Python:

---

1. ArithmeticError (Base Class)

`ZeroDivisionError`

```
print(10 / 0)  # ZeroDivisionError: division by zero
```

`OverflowError`

```
import math
print(math.exp(1000))  # OverflowError: math range error
```

`FloatingPointError`

```
import sys
sys.set_float_format("IEEE754")  # May trigger FloatingPointError in some con
figurations
```

2. AssertionError

```
x = 5
assert x > 10, "x is not greater than 10"  # AssertionError: x is not greater than
10
```

3. AttributeError

```
class Test:
    pass

obj = Test()
print(obj.value)  # AttributeError: 'Test' object has no attribute 'value'
```

4. BufferError

```
byte_array = memoryview(bytearray(5))
byte_array.release()  # This might raise BufferError if accessed after release
```

5. EOFError

```
input()  # If input() reaches end-of-file (EOF), it raises EOFError
```

6. ImportError & ModuleNotFoundError

```
import nonexistent_module  # ModuleNotFoundError: No module named 'none
xistent_module'
```

7. IndexError

```
arr = [1, 2, 3]
print(arr[5])  # IndexError: list index out of range
```

8. KeyError

```
my_dict = {"name": "John"}
print(my_dict["age"])  # KeyError: 'age'
```

9. KeyboardInterrupt

```
while True:
    pass  # Press Ctrl+C to trigger KeyboardInterrupt
```

10. MemoryError

```
big_list = [1] * (10**10)  # MemoryError: Out of memory
```

11. NameError & UnboundLocalError

```
print(undeclared_variable)  # NameError: name 'undeclared_variable' is not de
fined
```

`UnboundLocalError`

```
def func():
    print(x)  # UnboundLocalError: local variable 'x' referenced before assignm
ent
    x = 10
func()
```

12. NotImplementedError

```
class Base:
    def method(self):
        raise NotImplementedError("This method should be overridden")

class Derived(Base):
    pass

obj = Derived()
obj.method()  # NotImplementedError: This method should be overridden
```

13. OSError & Its Subclasses

FileNotFoundError

```
open("nonexistent_file.txt", "r")  # FileNotFoundError: No such file or directory
```

PermissionError

```
open("/root/secret.txt", "w")  # PermissionError: [Errno 13] Permission denied
```

TimeoutError

```
import time
raise TimeoutError("Operation took too long")  # TimeoutError: Operation took
```

```
too long
```

14. ReferenceError

```
import weakref

class Test:
    pass

obj = Test()
weak_ref = weakref.ref(obj)
del obj
print(weak_ref())  # None (object was garbage collected)
```

15. RuntimeError & RecursionError

RuntimeError

```
def faulty_function():
    return faulty_function()

faulty_function()  # RuntimeError: maximum recursion depth exceeded
```

RecursionError

```
import sys
sys.setrecursionlimit(10)
def infinite_recursion():
    infinite_recursion()

infinite_recursion()  # RecursionError: maximum recursion depth exceeded
```

16. StopIteration

```
it = iter([1, 2, 3])
print(next(it))
print(next(it))
print(next(it))
print(next(it))  # StopIteration
```

17. SyntaxError & IndentationError

SyntaxError

```
print "Hello"  # SyntaxError: Missing parentheses in call to 'print'
```

IndentationError

```
def func():
print("Hello")  # IndentationError: expected an indented block
```

TabError

```
def func():
    print("Hello")
    print("World")  # TabError: inconsistent use of tabs and spaces in indentatio
n
```

18. SystemError

```
raise SystemError("Internal system error occurred")  # SystemError: Internal s
ystem error occurred
```

19. TypeError

```
print("Hello" + 5)  # TypeError: can only concatenate str (not "int") to str
```

20. ValueError

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

21. UnicodeError & Its Subclasses

UnicodeEncodeError

```
s = "你好"
s.encode("ascii")  # UnicodeEncodeError: 'ascii' codec can't encode characte
rs
```

UnicodeDecodeError

```
b = b"\xff\xfeA\x00"
b.decode("ascii")  # UnicodeDecodeError: 'ascii' codec can't decode byte 0xf
f
```

UnicodeTranslateError

```
s = "é"
s.translate({ord("é"): None})  # UnicodeTranslateError: character mapping mu
st return integer or None
```

22. Warning Categories

DeprecationWarning

```
import warnings
```

```
warnings.warn("This is deprecated", DeprecationWarning)
```

FutureWarning

```
warnings.warn("This feature will be removed in future versions", FutureWarnin
g)
```

These examples cover all major exceptions in Python. Let me know if you need further explanations! 🚀