Contents

1. Errors and Exception

Error in Python can be of two types i.e. Syntax errors and Exceptions.

Syntax error: This error is caused by the wrong syntax in the code. It leads to the termination of the program.

```
a = 10
if(a > 5)
print("a is greater than 5")
```

```
Output:
    if(a > 5)
             ^
SyntaxError: invalid syntax
```

```
a = 10
if(a > 5):
print("a is greater than 5")
```

```
    print("a is greater than 5")
    ^
IndentationError: expected an indented block
```

Exceptions: Exceptions are raised when the program is syntactically correct, but the code results in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
marks = 10000
a = marks / 0
print(a)
```

```
  File "e:\python_programming\test.py", line 31, in <module>
    b = a/0
ZeroDivisionError: division by zero
```

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

- **SyntaxError:** This exception is raised when the interpreter encounters a syntax error in the code, such as a misspelled keyword, a missing colon, or an unbalanced parenthesis.
- **TypeError**: This exception is raised when an operation or function is applied to an object of the wrong type, such as adding a string to an integer.
- **NameError**: This exception is raised when a variable or function name is not found in the current scope.
- **IndexError**: This exception is raised when an index is out of range for a list, tuple, or other sequence types.
- **KeyError**: This exception is raised when a key is not found in a dictionary.
- **ValueError**: This exception is raised when a function or method is called with an invalid argument or input, such as trying to convert a string to an integer when the string does not represent a valid integer.
- **AttributeError**: This exception is raised when an attribute or method is not found on an object, such as trying to access a non-existent attribute of a class instance.
- **IOError**: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.
- **ZeroDivisionError**: This exception is raised when an attempt is made to divide a number by zero.
- **ImportError**: This exception is raised when an import statement fails to find or load a module.

Code	**Exceptions**
print(10 + "Hello")	print(10 + "Hello") TypeError: unsupported operand type(s) for +: 'int' and 'str'
print(hello())	print(hello()) NameError: name 'hello' is not defined

numbers = [1, 2, 3, 4]
print(numbers[4])
	print(numbers[4])
	IndexError: list index out of range

2. Catching and handling exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Python try...except Block

```
Syntax:
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Example:

```
a = 10
b = int(input("Enter the denominator"))
try:
    result = a / b
    print(result)
except:
    print("Error: denominator cannot be 0.")
```

```
Enter the denominator0
Error: denominator cannot be 0.
```

Catching Specific Exceptions in Python

Example:

```
a = 10
numbers = [1, 2, 3, 4]
b = int(input("Enter the denominator"))
index = int(input("Enter the index"))
try:
    result = a / b
    print(result)
    data = numbers[index]
    print(data)
except ZeroDivisionError:
    print("Error: denominator cannot be 0.")

except IndexError:
    print("Error: index out of bound.")
```

```
Enter the denominator2
Enter the index5
5.0
Error: index out of bound.
```

Python try with else clause

```
Syntax:
try:
    # code that may cause exception
except:
    # code to run when exception occurs
else:
    # code to run if try runs without errors
```

Example:

```
a = 10
b = int(input("Enter the denominator: "))
try:
    result = a / b
except ZeroDivisionError:
    print("Error: denominator cannot be 0.")
else:
    print(result)
```

```
Enter the denominator: 5
2.0
```

Python try...finally

```
Syntax:
try:
    # code that may cause exception
except:
    # code to run when exception occurs
else:
    # code that always executes
```

Example:

```
a = 10
b = int(input("Enter the denominator: "))
try:
    result = a / b
except ZeroDivisionError:
    print("Error: denominator cannot be 0.")
else:
    print(result)
finally:
    print("Execution Completed.")
```

```
Enter the denominator: 0
Error: denominator cannot be 0.
Execution Completed.
```

```
Enter the denominator: 5
```

```
2.0
Execution Completed.
```

3. User-defined exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

Here's the syntax to define custom exceptions,

```
class CustomError(Exception):
    ...
    pass

try:
   ...
   raise CustomError()

except CustomError:
    ...
```

Example:

```
class InvalidAgeException(Exception):
    pass

try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise InvalidAgeException()
    else:
        print("You can vote")
```

```
except InvalidAgeException:
    print("You cannot vote")
```

```
Enter your age: 10
You cannot vote
```

Example:

```
class InvalidAgeException(Exception):
    def __init__(self, message = "Age cannot be less than 18."):
        super().__init__(message)

age = int(input("Enter your age: "))
if age < 18:
    raise InvalidAgeException
```

```
Enter your age: 10
Traceback (most recent call last):
  File "e:\python_programming\tempCodeRunnerFile.py", line 8, in <module>
    raise InvalidAgeException
__main__.InvalidAgeException: Age cannot be less than 18.
```

Example:

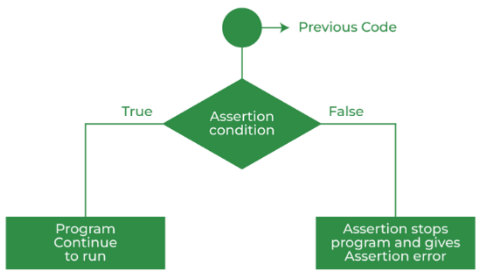
```
class InvalidAgeException(Exception):
    def __init__(self, message = "Age cannot be less than 18."):
        super().__init__(message)

try:
    age = int(input("Enter your age: "))
    if age < 18:
        raise InvalidAgeException()

except InvalidAgeException as e:
    print(e)
```

```
Enter your age: 10
Age cannot be less than 18.
```

4. Debugging programs with the assert statement

Python Assertions are the debugging tools that help in the smooth flow of code. Assertions are mainly assumptions that a programmer knows or always wants to be true and hence puts them in code so that failure of these doesn't allow the code to execute further.

The assert statement in Python is used for debugging purposes. It allows you to test if a condition in your code returns True. If the condition evaluates to False, an AssertionError exception is raised, which helps in identifying bugs or issues in your program.



Syntax :

assert condition, error_message(optional)

```
a = 10
b = 0
assert b != 0
print(a / b)
```

```
Traceback (most recent call last):
  File "e:\python_programming\assert_1.py", line 3, in <module>
    assert b != 0
AssertionError
```

Chapter 6: Exceptions and File Handling in Python [5 hrs.]