**Ch\_12. Python - Syntax Errors**

**Generally, three types of errors appear in a computer program:**

1. Syntax errors,
2. logical errors and
3. runtime errors.
4. **Syntax errors:**

* It is are the most common type of errors one faces while writing a program, whether you are new to programming or an experienced programmer.
* Syntax errors are basically related to the rules of grammar of a certain language.
* Syntax errors occur whenever the rules laid down by the language are not followed.
* In Python, there are well defined rules for giving name to an identifier, that is, a variable, a function, a class, a module or any Python object.
* Similarly, Python keywords should be used as per the syntax defined.
* Whenever these rules are not followed, Python interpreter displays a syntax error message.
* A simple example of declaring a variable in Python interactive shell is given below.

>>> name="Python

File "<stdin>", line 1

name="Python

^

SyntaxError: unterminated string literal (detected at line 1)

1. **logical errors:**

* Syntax errors are easy to identify and rectify.
* The IDE such as VS Code makes it easy.
* However, sometimes, your code doesn't show any syntax errors, but still the output of the program is not what you anticipate.
* Such errors are logical errors.
* They are hard to detect, as the error lies in the logic used in the code.
* You learn by experience how to correct logical errors.
* VS Code and other IDEs have features such as watches and breakpoints to trap these errors.

1. **runtime error:**

* Third type of error is a runtime error also called exception.
* There is no syntax error nor there is any logical error in your program.
* Most of the times, the program gives desired output, but in some specific situations you get abnormal behaviour of the program, such as the program abnormally terminates or gives some absurd result.
* The factors causing exceptions are generally external to the program.
* For example incorrect input, type conversion or malfunction IO device etc.

## **What is Exception?**

* An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
* In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.
* When a Python script raises an exception, it must either handle the exception immediately otherwise it terminates and quits.
* Python's standard library defines standard exception classes. As with other Python classes, Exceptions are also subclasses of Object class. Following is the object hierarchy of Python's Exceptions.

object

BaseException

Exception

ArithmeticError

FloatingPointError

OverflowError

ZeroDivisionError

AssertionError

AttributeError

BufferError

EOFError

ImportError

ModuleNotFoundError

LookupError

IndexError

KeyError

MemoryError

NameError

OSError

ReferenceError

RuntimeError

StopAsyncIteration

StopIteration

SyntaxError

# 12.1 Python - Exceptions Handling

If you have some suspicious code that may raise an exception, you can defend your program by placing the suspicious code in a **try**: block. After the **try**: block, include an **except**: statement, followed by a block of code which handles the problem as elegantly as possible.

* The **try**: block contains statements which are susceptible for exception
* If exception occurs, the program jumps to the **except**: block.
* If no exception in the **try**: block, the **except**: block is skipped.

### **Syntax**

Here is the simple syntax of **try...except...else** blocks −

try:

You do your operations here

......................

except ExceptionI:

If there is ExceptionI, then execute this block.

except ExceptionII:

If there is ExceptionII, then execute this block.

......................

else:

If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax −

* A single **try** statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.
* You can also provide a generic **except** clause, which handles any exception.
* After the except clause(s), you can include an **else** clause. The code in the **else** block executes if the code in the try: block does not raise an exception.
* The **else** block is a good place for code that does not need the try: block's protection.

### **Example**

This example opens a file, writes content in the file and comes out gracefully because there is no problem at all.

#Example 1

try:

    fh=open("testfile","w")

    fh.write("This is my test file for exception handling")

except IOError:

    print("Error.can\' find file or read data")

else:

    print("Written content in the file successfully")

    fh.close()

**12.2 Python - The try-except Block**

You can also use the **except** statement with no exceptions defined as follows −

try:

You do your operations here

......................

except:

If there is any exception, then execute this block.

......................

else:

If there is no exception then execute this block.

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

You can also use the same except statement to handle multiple exceptions as follows −

try:

You do your operations here

......................

except(Exception1[, Exception2[,...ExceptionN]]]):

If there is any exception from the given exception list,

then execute this block.

......................

else:

If there is no exception then execute this block.

**12.3 Python - The try-finally Block**

You can use a **finally**: block along with a **try**: block. The **finally**: block is a place to put any code that must execute, whether the try-block raised an exception or not.

The syntax of the **try-finally** statement is this −

try:

You do your operations here;

......................

Due to any exception, this may be skipped.

finally:

This would always be executed.

......................

**Note** − You can provide except clause(s), or a finally clause, but not both. You cannot use else clause as well along with a finally clause.

#Example 1

try:

    fh=open("testfile",'w')

    fh.write("This is the second file for exception handling")

finally:

    print("Error.can\' finde the file or read data")

    fh.close()

"""

If you do not have permission to open the file in writing mode, then it will produce the following output −

Error: can't find file or read data

"""

#The same example can be written more cleanly as follows −

try:

    fh=open("testfile1.txt","w")

    try:

        fh.write("This is the second file for exception handling")

    finally:

        print("Going to close the file")

        fh.close()

except IOError:

    print("Error.can\' finde the file or read data")

When an exception is thrown in the try block, the execution immediately passes to the **finally** block. After all the statements in the **finally** block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

## **Exception with Arguments**

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows −

try:

You do your operations here

......................

except ExceptionType as Argument:

You can print value of Argument here...

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

#Example 3 Following is an example for a single exception −

#define a function here

def temp\_convert(var):

    try:

        return int(var)

    except ValueError as Arg:

        print("The argument does not contain numbers\n",Arg)

#call the above function

temp\_convert('cyz')

"""It will produce the following output −

The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'"

# 12.4 Python - Raising Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows −

### **Syntax**

raise [Exception [, args [, traceback]]]

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

### **Example**

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. class. Defining new exceptions is quite easy and can be done as follows −

def functionName( level ):

if level <1:

raise Exception(level)

# The code below to this would not be executed

# if we raise the exception

return level

**Note** − In order to catch an exception, an "except" clause must refer to the same exception thrown either as a class object or a simple string. For example, to capture the above exception, we must write the except clause as follows −

try:

Business Logic here...

except Exception as e:

Exception handling here using e.args...

else:

Rest of the code here...

The following example illustrates the use of raising an exception −

def functionName(level):

    if level<1:

        raise Exception(level)

     # The code below to this would not be executed

      # if we raise the exception

    return level

try:

    l=functionName(-10)

    print('level=',1)

except Exception as e:

    print("Error in level arg",e.args[0])

"""

This will produce the following output −

error in level argument -10

    """

# 12.5 Python - Exception Chaining

Exception chaining is a technique of handling exceptions by re-throwing a caught exception after wrapping it inside a new exception. The original exception is saved as a property (such as cause) of the new exception.

During the handling of one exception 'A', it is possible that another exception 'B' may occur. It is useful to know about both exceptions in order to debug the problem. Sometimes it is useful for an exception handler to deliberately re-raise an exception, either to provide extra information or to translate an exception to another type.

In Python 3.x, it is possible to implement exception chaining. If there is any unhandled exception inside an except section, it will have the exception being handled attached to it and included in the error message.

### **Example**

In the following code snippet, trying to open a non-existent file raises FileNotFoundError. It is detected by the except block. While handling another exception is raised.

#Example 1

try:

    open("Nofile.txt")

except OSError:

    raise RuntimeError("unable to handle error")

Traceback (most recent call last):

File "/home/cg/root/64afcad39c651/main.py", line 2, in <module>

open("nofile.txt")

FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):

File "/home/cg/root/64afcad39c651/main.py", line 4, in <module>

raise RuntimeError("unable to handle error")

RuntimeError: unable to handle error

### **raise . . from**

If you use an optional from clause in the raise statement, it indicates that an exception is a direct consequence of another. This can be useful when you are transforming exceptions. The token after from keyword should be the exception object.

# 12.6 Python - Nested try Block

In a Python program, if there is another **try-except** construct either inside either a **try** block or inside its **except** block, it is known as a nested-try block. This is needed when different blocks like outer and inner may cause different errors. To handle them, we need nested try blocks.

We start with an example having a single "try − except − finally" construct. If the statements inside try encounter exception, it is handled by except block. With or without exception occurred, the finally block is always executed.

### **Example 1**

Here, the **try** block has "division by 0" situation, hence the **except** block comes into play. It is equipped to handle the generic exception with Exception class.

#Example 1

a=10

b=0

try:

    print(a/b)

except Exception:

    print("General Exception")

finally:

    print("inside outer finally block")

"""

It will produce the following output −

General Exception

inside outer finally block

"""

#Example 2

"""

Let us now see how to nest the try constructs.

We put another "try − except − finally" blocks inside the existing try block.

The except keyword for inner try now handles generic Exception,

while we ask the except block of outer try to handle ZeroDivisionError.

Since exception doesn't occur in the inner try block, its corresponding generic Except isn't called. The division by 0 situation is handled by outer except clause."""

print("\nExample 2")

try:

    print(a/b)

    try:

       print("This is inner try block")

    except Exception:

       print("General Exception")

    finally:

       print("inside inner finally block")

except ZeroDivisionError:

    print("Division by 0")

finally:

    print("inside Outer finally block ")

"""

It will produce the following output −

Division by 0

inside outer finally block

"""

#Example 3

"""

Now we reverse the situation. Out of the nested try blocks,

the outer one doesn't have any exception raised, but the statement causing division by 0 is inside inner try,

and hence the exception handled by inner except block. Obviously, the except part corresponding to outer try: will not be called upon."""

print("\nExample 3")

try:

     print("this is outer try block")

     try:

         print(a/b)

     except ZeroDivisionError:

        print("Division by 0")

     finally:

        print("Inside the inner finally block")

except Exception:

    print("General Exception")

finally:

    print("inside Outer finally block ")

"""

It will produce the following output −

This is outer try block

Division by 0

inside inner finally block

inside outer finally block

    """

**12.7 Python - User-Defined Exceptions**

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example that has a user-defined MyException class. Here, a class is created that is subclassed from base Exception class. This is useful when you need to display more specific information when an exception is caught.

In the **try** block, the user-defined exception is raised whenever value of num variable is less than 0 or more than 100 and caught in the **except** block. The variable e is used to create an instance of the class MyException.

# Define a custom exception class named myException that inherits from the built-in Exception class

class myException(Exception):

    'invalid marks'

    pass

# Assign a value to the variable 'num'

num = 110

# Use a try-except block to handle potential exceptions

try:

    # Check if the value of 'num' is outside the valid range (less than 0 or greater than 100)

    if num < 0 or num > 100:

        # If the condition is met, raise the custom exception 'myException'

        raise myException

except myException as e:

    # Catch and handle the custom exception by printing an error message along with the invalid 'num' value

    print("invalid marks", num)

else:

    # If no exception is raised, execute this block and print a message indicating valid 'num' value

    print("Marks obtained", num)

**12.9 Python - Built-in Exceptions**

Here is a list of Standard Exceptions available in Python −

|  |  |
| --- | --- |
| **Sr.No.** | **Exception Name & Description** |
| 1 | **Exception**  Base class for all exceptions |
| 2 | **StopIteration**  Raised when the next() method of an iterator does not point to any object. |
| 3 | **SystemExit**  Raised by the sys.exit() function. |
| 4 | **StandardError**  Base class for all built-in exceptions except StopIteration and SystemExit. |
| 5 | **ArithmeticError**  Base class for all errors that occur for numeric calculation. |
| 6 | **OverflowError**  Raised when a calculation exceeds maximum limit for a numeric type. |
| 7 | **FloatingPointError**  Raised when a floating point calculation fails. |
| 8 | **ZeroDivisonError**  Raised when division or modulo by zero takes place for all numeric types. |
| 9 | **AssertionError**  Raised in case of failure of the Assert statement. |
| 10 | **AttributeError**  Raised in case of failure of attribute reference or assignment. |
| 11 | **EOFError**  Raised when there is no input from either the raw\_input() or input() function and the end of file is reached. |
| 12 | **ImportError**  Raised when an import statement fails. |
| 13 | **KeyboardInterrupt**  Raised when the user interrupts program execution, usually by pressing Ctrl+C. |
| 14 | **LookupError**  Base class for all lookup errors. |
| 15 | **IndexError**  Raised when an index is not found in a sequence. |
| 16 | **KeyError**  Raised when the specified key is not found in the dictionary. |
| 17 | **NameError**  Raised when an identifier is not found in the local or global namespace. |
| 18 | **UnboundLocalError**  Raised when trying to access a local variable in a function or method but no value has been assigned to it. |
| 19 | **EnvironmentError**  Base class for all exceptions that occur outside the Python environment. |
| 20 | **IOError**  Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist. |
| 21 | **OSError**  Raised for operating system-related errors. |
| 22 | **SyntaxError**  Raised when there is an error in Python syntax. |
| 23 | **IndentationError**  Raised when indentation is not specified properly. |
| 24 | **SystemError**  Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit. |
| 25 | **SystemExit**  Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit. |
| 26 | **TypeError**  Raised when an operation or function is attempted that is invalid for the specified data type. |
| 27 | **ValueError**  Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified. |
| 28 | **RuntimeError**  Raised when a generated error does not fall into any category. |
| 29 | **NotImplementedError**  Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented. |

#Example 1 IndexError

#It is shown when trying to access item at invalid index.

num=[10,20,30,40]

#for n in (range(5)):#IndexError: list index out of range

for n in range(4):

    print(num[n])

#Example 2 ModuleNotFoundError

#This is displayed when module could not be found.

"""

import notamodule

Traceback (most recent call last):

   import notamodule

ModuleNotFoundError: No module named 'notamodule'

"""

#Example 3 KeyError

#It occurs as dictionary key is not found.

d1={1:'aa',2:'bb',3:'cc'}

#print(d1['4'])

"""

Traceback (most recent call last):

   D1['4']

KeyError: '4'

    """

#Example 4 ImportError

#It is shown when specified function is not available for import.

"""

from math import cube

Traceback (most recent call last):

   from math import cube

ImportError: cannot import name 'cube'"""

#Example 5 StopIteration error

#This error appears when next() function is called after iterator stream exhausts.

it=iter([1,2,3])

print(next(it))

print(next(it))

print(next(it))

#print(next(it))

"""

Traceback (most recent call last):

   next(it)

StopIteration

    """

#Example 6 TypeError

#This is shown when operator or function is applied to an object of inappropriate type.

#print('2'+2)

#TypeError: can only concatenate str (not "int") to str

#Example 7 ValueError

#It is displayed when function's argument is of inappropriate type.

#print(int('xyz'))

#ValueError: invalid literal for int() with base 10: 'xyz'

#Example 8 NameError

#This is encountered when object could not be found.

#print(age)

#NameError: This is encountered when object could not be found.

#Example 9 ZeroDivisionError

#It is shown when second operator in division is zero.

#x=100/0

#ZeroDivisionError: It is shown when second operator in division is zero.

#Example 9 KeyboardInterrupt

#When user hits the interrupt key normally Control-C during execution of program.

"""

name=input('enter your name')

enter your name^c

Traceback (most recent call last):

   name=input('enter your name')

KeyboardInterrupt

    """