Exception Handling

Types of error

- **Compile-time Error or Syntax Error**

Syntax errors are the most essential kind of mistake and they emerge when the Python parser cannot comprehend a line of code. Then, it features where the sentence structure has an error. Most syntactical blunders are grammatical errors, wrong space, or off base contentions.

Example-1:

Code	Output

```
i = 5
if i< 10
    print (i)
```
	```
	if i<10
	           ^
	SyntaxError: invalid syntax
	```

Here, in Example-1, we can see from the output that there is a ":"(colon) missing at the end of the if statement, so this program will show "SyntaxError: invalid syntax" by making a pointer at the missed colon.

- **Logical Error**

Logical errors are the most difficult to fix as they produce an incorrect result due to incorrect program **logic while** running the program without crashing/ showing any errors.

Example-2:

Code	Output

```
num = 5
if num % 2 != 0:
     print ("even")
```
	`even`

Here, in the output red messages are shown, but this code has a logical error. As we know if a number is divisible by 2, then it is an even number. Otherwise, it is an odd number. But, the conditional statement in Example-2 is implying if a number is not divisible by 2, then it is even. So, we are seeing even as output for a given number 5, which is actually odd.

- **Runtime Error**

Runtime errors occur during the execution of one of the statements in the program, which causes the interpreter to stop executing the program and display an error message.

Example-3:

Code	Output

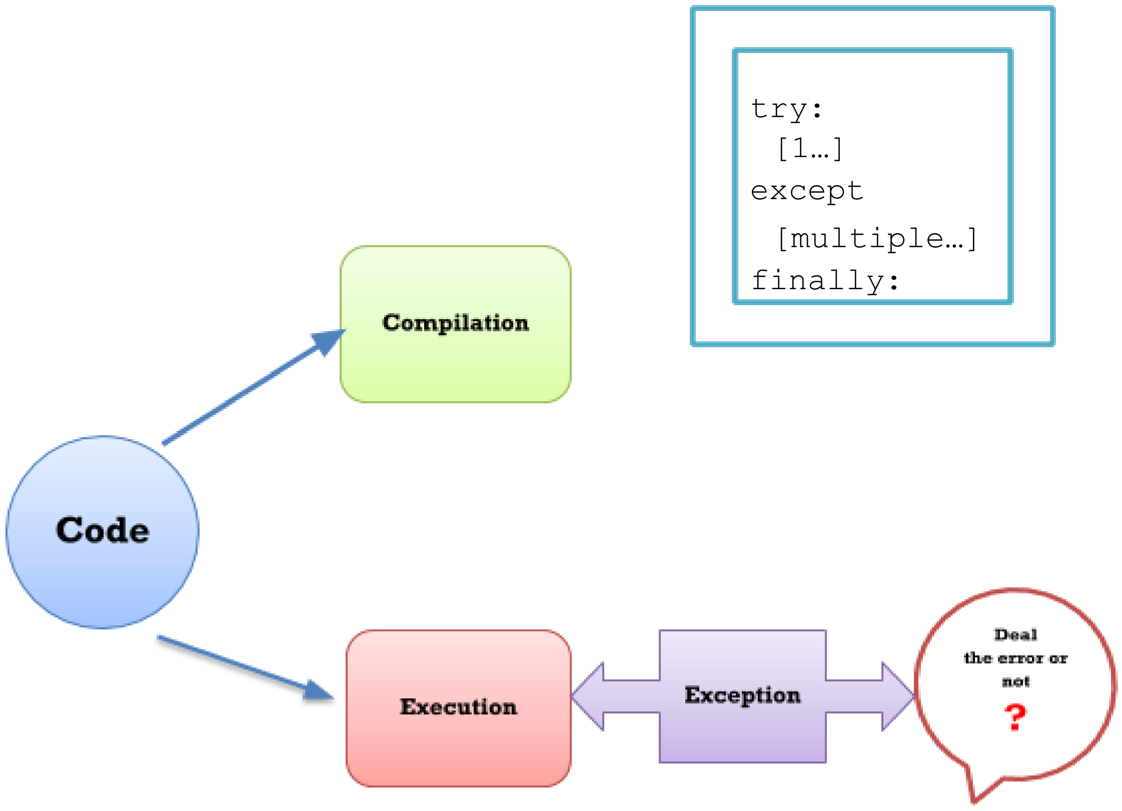
`a = 5`
`b = 0`
`x = a/b`
`print (x)`
	`ZeroDivisionError`

Here, in Example-3, we are dividing 5 by zero, so this program will show ZeroDivisionError.
Exception handles this type of error!

Exception:

An exception is an event that occurs **during the execution** of a program, that disrupts the normal flow of the program. So, rather than terminating the program by showing an error, exception handles the error and terminates or continues the program according to the instruction given in the program.

Diagrammatic Overview:



```
try:
  [1…]
except
  [multiple…]
finally:
```

Try block & Except block:

Try block: The portion of code that has a possibility of generating runtime error is written in the “try block”. This try block comes in pairs similar to if-else blocks. Each try block can have multiple “except blocks”. So, when an error occurs in the try block, the program can continue its execution by searching for a suitable except block in the try-except branching, that matches with the exception generated.

Except block: Handles the errors generated in the Try block.

ex1.py

Code

```
a = 6
b = 0
try:
     x = a/b [causes error]
     print (x)
except Exception:
     print ("cannot divide by zero")
```

Output

```
cannot divide by zero
```

Here, in **ex1.py** we have divided “a” by “b” where a = 6 and b = 0 and kept the result in “x”. It causes a divide by zero error that is defined in python by “ZeroDivisionError”. The **`x = a/b,`** this portion of code will cause a runtime error that will force our program to stop the execution, which we do not want in real life. So, this possible exception generating code is written in “try block”. The “ZeroDivisionError '' error caused in the try block is being accepted by the except block that contains the Exception class. So, “cannot divide by zero” got printed as an output.
Here, the Exception class can accept any type of errors. But, in real life, we might need to deal with multiple errors in a given block of code. So, we need to specify different classes as different options in the except blocks which will add under the try block.

```
try:
     [error code]
except A_class:
     [specific action for A type error]
except B_class:
     [specific action for B type error]
except C_class:
     [specific action for C type error]
...
...
...
```

Link to exception hierarchy:

https://docs.python.org/3/library/exceptions.html#exception-hierarchy

ex2.py

Code	Output

`a = 6`
`b = 2`
`try:`
`     x = a/b`
`     print (x)`
`except Exception:`
`     print ("cannot divide by zero")`
	`3.0`
	
	Here in **ex2.py**, no error occurred. So, the program didn't go to the "except block".

Finally block:

It is written at the end of a try-except block ladder. It can be used only once in a try-except branch. Whether any runtime error is raised or not in the try block, this "finally block" will always get executed. So, this block of the code will execute with or without exception regenerating in the try block. It is usually used to show a common or default message or to perform default actions at any stake.

ex3.py

Code	Output

`a = 6`
`b = 0`
`try:`
`     print ("start procedure")`
`     x = a/b`
`     print(x)`
`except Exception:`
`     print ("cannot divide by zero")`
`finally:`
`     print("end procedure")`
	`start procedure`
	`cannot divide by zero`
	`end procedure`

Here, in ex3.py, we can see that a new block named "**finally** " has been introduced. After the except block, the "finally block" is getting executed. So we can see the "`end procedure`" message in the output.

Else block:

It is written at the end of a try-except block ladder, before the finally block. If there is no exception in the try block, then this block of code is executed.

ex4.py

Code	Output

```
a = 6
b = 2
try:
     print ("start procedure")
     x = a/b
     print(x)
except Exception:
     print ("cannot divide by zero")
else:
     print("no exception occurred")
finally:
     print("end procedure")
```
	```
	start procedure
	3.0
	no exception occurred
	end procedure
	```

Here, in **ex4.py,** even though there is no error in the “try” block, the “finally” block is getting executed. Additionally, since there is no error in the code, the “except block” is not working and its neighboring else block will work as the next valid statement.

Major types of error:

Exception	Cause of Error
NameError	Raised when a variable is not found in local or global scope.
IndentationError	Raised when there is incorrect indentation.
ZeroDivisionError	Raised when the second operand of division or modulo operation is zero.
TypeError	Raised when a function or operation is applied to an object of incorrect type
RuntimeError	Raised when an error does not fall under any other category.
KeyError	Raised when a key is not found in a dictionary.
IndexError	Raised when the index of a sequence is out of range.

Multiple exceptions

file1.py

Code	Input	Output

`try:`
    `a = int(input("Enter value of a: "))`
    `b = int(input("Enter value of b: "))`
    `print ("start procedure")`
    `x = a/b`
    `print("X", x)`
`except ZeroDivisionError:`
    `print ("cannot divide by 0")`
`except ValueError:`
    `print("please enter an int")`
`finally:`
    `print("end procedure")`
	12
	4
		start procedure
		X 3.0
		end procedure

	"p"	please enter an int end procedure
	30 0	start procedure cannot divide by 0 end procedure

Raising exception

We can choose to throw an exception if a condition occurs. For throwing or to "**raise**" an exception, we need to use the "**raise"** keyword.

file2.py

Code	Input	Output

`try:`
    `a = int(input("Enter value of a: "))`
    `b = int(input("Enter value of b: "))`
    `print ("start procedure")`
 	
    `if b == 0:`
        `raise ZeroDivisionError("b cannot be 0")`
 	
    `x = a/b`
    `print("X", x)`
`except ValueError:`
    `print("please enter an int")`
`finally:`
    `print("end procedure")`
	16
	2
		start procedure
		X 8.0
		end procedure

	"p"	please enter an int end procedure
	8 0	start procedure end procedure **ZeroDivisionError: b cannot be 0**

Here, in **file2.py,** for b=0, the above example is raising the "ZeroDivisionError" error with a customized message. For this reason, we are seeing different outputs for a=8 and b=0 with a custom error message lastly "`ZeroDivisionError:` b cannot be zero".