SNS COLLEGE OF ENGINEERING

Kurumbapalayam(Po), Coimbatore – 641 107
Accredited by NAAC-UGC with 'A' Grade
Approved by AICTE, Recognized by UGC & Affiliated to Anna University, Chennai

Department of Artificial Intelligence and Data Science

Course Name – Computational Thinking and Python Programming

I Year / I Semester

Unit 5-Files

EXCEPTION

Python (interpreter) raises exceptions when it encounters errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

```
>>> if a < 3
File "<interactive input>", line 1
if a < 3
SyntaxError: invalid syntax
```

Errors can also occur at runtime and these are called exceptions. They occur, for example, when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc. Whenever these type of runtime error occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

```
>>> 1 / 0
Traceback (most recent call last):
File "<string>", line 301, in runcode
File "<interactive input>", line 1, in <module>
ZeroDivisionError: division by zero

>>> open("imaginary.txt")
Traceback (most recent call last):
File "<string>", line 301, in runcode
File "<interactive input>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'
```

Python Built-in Exceptions

Illegal operations can raise exceptions. There are plenty of built-in exceptions in Python that are raised when corresponding errors occur. We can view all the built-in exceptions using the local() built-in functions as follows.

>>> locals()['__builtins__']

This will return us a dictionary of built-in exceptions, functions and attributes. Some of the common built-in exceptions in Python programming along with the error that cause then are tabulated below.

Python Built-in Exceptions

Exception : Cause of Error

AssertionError : Raised when assert statement fails.
AttributeError Raised when attribute assignment or reference fails.

FloatingPointError : Raised when a floating point operation fails.
GeneratorExit : Raise when a generator's close() method is called.
ImportError : Raised when the imported module is not found.
IndexError : Raised when index of a sequence is out of range.
KeyError : Raised when a key is not found in a dictionary.

KeyboardInterrupt : Raised when the user hits interrupt key (Ctrl+c or delete).
MemoryError : Raised when an operation runs out of memory.
NameError : Raised when a variable is not found in local or global scope.
NotImplementedError : Raised by abstract methods.
OSError : Raised when system operation causes system related error.
OverflowError : Raised when result of an arithmetic operation is too large to be represented.
ReferenceError : Raised when a weak reference proxy is used to access a garbage collected referent.
RuntimeError : Raised when an error does not fall under any other category.
StopIteration : Raised by next() function to indicate that there is no further item to be returned by iterator.
SyntaxError : Raised by parser when syntax error is encountered.
IndentationError : Raised when there is incorrect indentation.
TabError : Raised when indentation consists of inconsistent tabs and spaces.
SystemError : Raised when interpreter detects internal error.
SystemExit : Raised by sys.exit() function.
TypeError : Raised when a function or operation is applied to an object of incorrect type.
UnboundLocalError : Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.
UnicodeError : Raised when a Unicode-related encoding or decoding error occurs.
UnicodeEncodeError : Raised when a Unicode-related error occurs during encoding.

UnicodeDecodeError : Raised when a Unicode-related error occurs during decoding.

UnicodeTranslateError : Raised when a Unicode-related error occurs during translating.
ValueError : Raised when a function gets argument of correct type but improper value.
ZeroDivisionError : Raised when second operand of division or modulo operation is zero.

We can handle these built-in and user-defined exceptions in Python using try, except and finally statements.

Python Exception Handling

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong.
When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.
For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.
If never handled, an error message is spit out and our program come to a sudden, unexpected halt.

Catching Exceptions in Python

In Python, exceptions can be handled using a try statement.
A critical operation which can raise exception is placed inside the try clause and the code that handles exception is written in except clause.
It is up to us, what operations we perform once we have caught the exception. Here is a simple example.

```
#import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]
for entry in randomList:
try:
print("The entry is", entry)
r = 1/int(entry)
break
except:
print("Oops!",sys.exc_info()[0],"occured.")
print("Next entry.")
print()
print("The reciprocal of",entry,"is",r)
```

Output

The entry is a
Oops! <class 'ValueError'> occured.
Next entry.

The entry is 0
Oops! <class 'ZeroDivisionError' > occured.
Next entry.

The entry is 2
The reciprocal of 2 is 0.5

In this program, we loop until the user enters an integer that has a valid reciprocal. The portion that can cause exception is placed inside try block.
If no exception occurs, except block is skipped and normal flow continues. But if any exception occurs, it is caught by the except block.
Here, we print the name of the exception using ex_info() function inside sys module and ask the user to try again. We can see that the values 'a' and '1.3' causes ValueError and '0' causes ZeroDivisionError.

try...finally

The try statement in Python can have an optional finally clause. This clause is executed no matter what, and is generally used to release external resources.
For example, we may be connected to a remote data center through the network or working with a file or working with a Graphical User Interface (GUI).
In all these circumstances, we must clean up the resource once used, whether it was successful or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee execution. Here is an example of file operations to illustrate this.

```
try:
f= open("test.txt",encoding = 'utf-8')
# perform file operations
finally:
f.close()
```