Exception Handling

When an error occurs, or exception as we call it, Python will normally stop and generate an error message.

These exceptions can be handled using the `try` statement:

```
try:
  print(x)
except:
  print("An exception occurred")
```

The `try` block lets you test a block of code for errors. The `except` block lets you handle the error. The `finally` block lets you execute code, regardless of the result of the try- and except blocks.

Since the try block raises an error, the except block will be executed. Without the try block, the program will crash and raise an error. You can define as many exception blocks as you want, e.g. if you want to execute a special block of code for a special kind of error:

```
try:
  print(x)
except NameError:
  print("Variable x is not defined")
except:
  print("Something else went wrong")
```

Catching Specific Exceptions in Python

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clause to handle them differently but only one will be executed in case an exception occurs.

```
try:
   # do something
   pass

except ValueError:
   # handle ValueError exception
   pass

except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass

```

```
except:
  # handle all other exceptions
  pass
```

Else

You can use the else keyword to define a block of code to be executed if no errors were raised:

```
try:
  print("Hello")
except:
  print("Something went wrong")
else:
  print("Nothing went wrong")
```

The finally block, if specified, will be executed regardless if the try block raises an error or not

```
try:
  print(x)
except:
  print("Something went wrong")
finally:
  print("The 'try except' is finished")
```

This can be useful to close objects and clean up resources:

```
try:
  f = open("demofile.txt")
  f.write("Lorum Ipsum")
except:
  print("Something went wrong when writing to the file")
finally:
  f.close()
```

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	**ZeroDivisionError** 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	**IOError** 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.

Exception → RuntimeError → Arithmatic → ValueError

Exception → IOError →

Argument of an Exception

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, Argument:

  You can print value of Argument here...
```

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 above function here.

temp_convert("xyz");
```

Raising Exceptions

In Python programming, exceptions are raised when corresponding errors occur at run time, but we can forcefully raise it using the keyword **raise**.

We can also optionally pass in value to the exception to clarify why that exception was raised.

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

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

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

```
def checkAge(age):
    if(age < 21):
        raise InvalidAgeError("age More than 21 required")
```

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 related to *RuntimeError*. Here, a class is created that is subclassed from *RuntimeError*. 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 and caught in the except block. The variable e is used to create an instance of the class **InvalidAgeError**.

```
class InvalidAgeError(RuntimeError):

    def __init__(self, errorMessage):
        self.errorMessage = errorMessage


try:
    checkAge(15)
except InvalidAgeError as error:
    print("Age should be more than 21", error)
```