Exception Handling

- 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.

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:

Example

#The try block will generate an exception, because x is not defined:

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

output:

An exception occurred

Many Exceptions :

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:

Example

Print one message if the try block raises a NameError and another for other cd Desktop

cd

```
try:
```

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

output:

Variable x is not defined

Else :

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

Example

In this example, the try block does not generate any error:

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

output

Hello

Nothing went wrong

Finally :

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

Example

```
try:
```

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

output:

Something went wrong

The 'try except' is finished

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

Example

Try to open and write to a file that is not writable:

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

output:

Something went wrong when writing to the file

Raising Exception:

The raise statement allows the programmer to force a specific exception to occur. The sole argument in raise indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from Exception).

example:

```
# Program to depict Raising Exception
```

```
try:
    raise NameError("Hi there")  # Raise Error
except NameError:
    print ("An exception")
    raise  # To determine whether the exception was raised or not
```

The output of the above code will simply line printed as “An exception” but a Runtime error will also occur in the last due to raise statement in the last line. So, the output on your command line will look like

```
Traceback (most recent call last):
  File "003dff3d748c75816b7f849be98b91b8.py", line 4, in
    raise NameError("Hi there") # Raise Error
NameError: Hi there
```

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.