File Handling

1. Opening a file

Syntax : fileobject = open(filename, [accessmode])

-open() returns an object of type file on a success, error otherwise.
- different access modes – r, w, a . . .
- The returned file object does not hold the file contents, rather a `window' through which file name can be viewed

2. Reading

fileobject.read()
fileobject.readlines()

3. Writing

fileobject.write(msg)
fileobject.writeliens(msgseq)

4. Close file

fileobject.close()

5. other Useful Methods -

fileobject.seek(offset, [from])
- Move <offset> bytes <from> location

fileobject.tell()
- Current position in file

ERRORS & EXCEPTIONS

Exceptions
Events that can modify the control flow through a program.

Error
A bug in a program that causes it to operate incorrectly,

Types of Error -
Parsing (Syntax) errors
Logical errors
Runtime errors

1. Why need to rectify errors/exceptions?
- Exceptions are inevitable and could be fatal.
- Secure Programming (a.k.a Defensive Programming)

2. Exceptions

- Are generated automatically on errors.
- Built-in (Standard) vs User-defined
- Can be triggered and handled by our code
- Generally, a two-phase process:
1. Detection of exception condition – raising an exception implicitly or explicitly
2. Exception handling
e.g. Ignore error, log error, abort program, remedial actions, etc

3. Types of Errors in Python

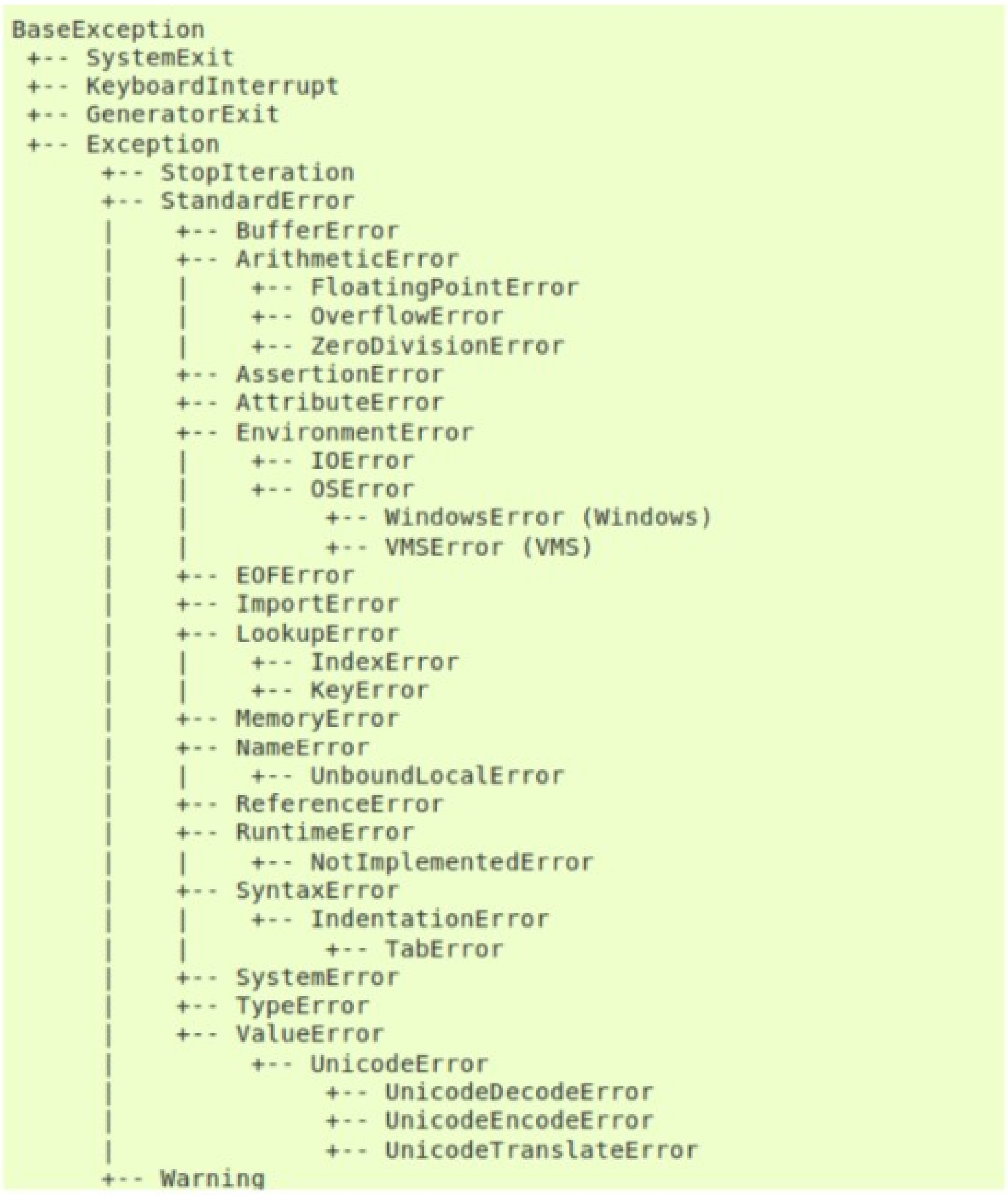
Some standard exceptions you’ve probably encountered:
- NameError - access uninitialized variable
- SyntaxError
- ZeroDivisionError
- KeyError - access non-existing dictionary key
- IndexError - access out-of-range index
- IOError - input/output (e.g. in file read/write)
- TypeError - operations with invalid type.

Try out :

i) 10 * (1/0) which type of exception raised?

ii) 4 + spam*3 which type of exception raised?

Iii) '2' + 2 which type of exception raised?



```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
      |    +-- BufferError
      |    +-- ArithmeticError
      |    |    +-- FloatingPointError
      |    |    +-- OverflowError
      |    |    +-- ZeroDivisionError
      |    +-- AssertionError
      |    +-- AttributeError
      |    +-- EnvironmentError
      |    |    +-- IOError
      |    |    +-- OSError
      |    |         +-- WindowsError (Windows)
      |    |         +-- VMSError (VMS)
      |    +-- EOFError
      |    +-- ImportError
      |    +-- LookupError
      |    |    +-- IndexError
      |    |    +-- KeyError
      |    +-- MemoryError
      |    +-- NameError
      |    |    +-- UnboundLocalError
      |    +-- ReferenceError
      |    +-- RuntimeError
      |    |    +-- NotImplementedError
      |    +-- SyntaxError
      |    |    +-- IndentationError
      |    |         +-- TabError
      |    +-- SystemError
      |    +-- TypeError
      |    +-- ValueError
      |         +-- UnicodeError
      |              +-- UnicodeDecodeError
      |              +-- UnicodeEncodeError
      |              +-- UnicodeTranslateError
      +-- Warning
```

4. Exception Handling

- try...except...[else]

- try...finally

Useful to specify cleanup actions that must occur, regardless of exception.
e.g. File close, server disconnects, etc

Syntax :
```
try:
        <statements>
except <e1>:
        <statements>
except (e2, e3, ...eN):
        <statements>
except:
        <statements>
else:
        <statements>
```

Eg.

-> Prog1 -

```
try:
        f = open('IDoNotExist.txt')
except IOError:
        print 'Unable to open the file'
```

-> Prog2 -

```
try:
        float('this is test')
        float([1,2])
except(ValueError, TypeError):
        print 'Invalid Argument Encountered'
else:
        print 'No exception occured!'
```

try ... finally SYNTAX:
```
try:
        <statements>
finally:
	<statements>			# Always run this code
```

Prog 3 -

```
try:
	n = float(raw_input(‘Enter your number:’))
	double = 2 * n
finally:
	print ‘Who can stop me from executing?’
	print "Double=", double
```

5. Raising Exceptions

To explicitly raise exceptions, use the raise statement.

SYNTAX : raise <exception to be raised> [, args]

If no exception supplied with the raise statement, the last exception (if any) in the current try block is re-raised;
otherwise, TypeError (no exception to re-raise).

Eg.

```
try:
	raise NameError
except NameError:
	print ‘Exception ocurred!’
	raise
```

Eg

```
try:
    raise Exception('spam', 'eggs')
except Exception as inst:
    print(type(inst))    # the exception instance
    print(inst.args)     # arguments stored in .args
    print(inst)          # __str__ allows args to be printed directly,
                         # but may be overridden in exception subclasses
    x, y = inst.args     # unpack args
    print('x =', x)
    print('y =', y)
```

6. Assertions

Are diagnostic predicates which must evaluate to true. If false, an AssertionError exception is thrown.
Think of them as conditional raise i.e. raise-if/raise-if-not

SYNTAX : assert <test>

Eg.

```
def f(n):
        assert n>0                    # must be positive
        return math.sqrt(n)
```