Python Errors and Built-in-Exceptions

When writing a program, we, more often than not, will encounter errors.

Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.

In [2]:
```
if a < 3
```

```
  File "<ipython-input-2-8625009197cc>", 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.

In [3]:
```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-b710d87c980c> in <module>()
----> 1 1 / 0

ZeroDivisionError: integer division or modulo by zero
```

In [4]:
```
open('test.txt')
```

```
---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-4-46a2b0c9e87f> in <module>()
----> 1 open('test.txt')

IOError: [Errno 2] No such file or directory: 'test.txt'
```

Python Built-in Exceptions

In [1]:
```
dir(__builtins__)
```

```
Out[1]: ['ArithmeticError',
         'AssertionError',
         'AttributeError',
         'BaseException',
         'BlockingIOError',
         'BrokenPipeError',
         'BufferError',
         'BytesWarning',
         'ChildProcessError',
         'ConnectionAbortedError',
         'ConnectionError',
         'ConnectionRefusedError',
         'ConnectionResetError',
         'DeprecationWarning',
         'EOFError',
         'Ellipsis',
         'EnvironmentError',
         'Exception',
         'False',
         'FileExistsError',
         'FileNotFoundError',
         'FloatingPointError',
         'FutureWarning',
         'GeneratorExit',
         'IOError',
         'ImportError',
         'ImportWarning',
         'IndentationError',
         'IndexError',
         'InterruptedError',
         'IsADirectoryError',
         'KeyError',
         'KeyboardInterrupt',
         'LookupError',
         'MemoryError',
         'ModuleNotFoundError',
         'NameError',
         'None',
         'NotADirectoryError',
         'NotImplemented',
         'NotImplementedError',
         'OSError',
         'OverflowError',
         'PendingDeprecationWarning',
         'PermissionError',
         'ProcessLookupError',
         'RecursionError',
         'ReferenceError',
         'ResourceWarning',
         'RuntimeError',
         'RuntimeWarning',
         'StopAsyncIteration',
         'StopIteration',
         'SyntaxError',
         'SyntaxWarning',
         'SystemError',
         'SystemExit',
         'TabError',
         'TimeoutError',
         'True',
         'TypeError',
         'UnboundLocalError',
         'UnicodeDecodeError',
         'UnicodeEncodeError',
```

Python Exception Handling - Try, Except and Finally

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.

In [6]:
```
# import module sys to get the type of exception
import sys

lst = ['b', 0, 2]

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

```
The entry is b
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
```

Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the except clause.

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.

In [10]:
```
import sys

lst = ['b', 0, 2]

for entry in lst:
    try:
        print("****************************")
        print("The entry is", entry)
        r = 1 / int(entry)
    except(ValueError):
        print("This is a ValueError.")
    except(ZeroDivisionError):
        print("This is a ZeroError.")
    except:
        print("Some other error")
print("The reciprocal of", entry, "is", r)
```

```
****************************
The entry is b
This is a ValueError.
****************************
The entry is 0
This is a ZeroError.
****************************
The entry is 2
The reciprocal of 2 is 0.5
```

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.

In [15]:
```
raise KeyboardInterrupt
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-15-7d145351408f> in <module>()
----> 1 raise KeyboardInterrupt

KeyboardInterrupt:
```

In [16]:
```
raise MemoryError("This is memory Error")
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-16-0ce8140e6e3c> in <module>()
----> 1 raise MemoryError("This is memory Error")

MemoryError: This is memory Error
```

In [13]:
```
try:
    num = int(input("Enter a positive integer:"))
    if num <= 0:
        raise ValueError("Error:Entered negative number")
except ValueError as e:
    print(e)
```

```
Enter a positive integer:-10
Error:Entered negative number
```

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.

In [22]:
```
try:
    f = open('sample.txt')
    #perform file operations

finally:
    f.close()
```