Programming technologies

Exception Handling

Agenda

- Exceptions introduction
- Handling Exceptions
- Raising Exceptions
- User-defined Exceptions
- Assertions
- LBYL & EAFP

Errors and Exceptions

When a Python program meets an error, it stops the execution of the rest of the program. An error in Python might be either an error in the syntax of an expression or a Python exception.

An exception in Python is an incident that happens while executing a program that causes the regular course of the program's commands to be disrupted. When a Python code comes across a condition it can't handle, it raises an exception. An object in Python that describes an error is called an exception.

When a Python code throws an exception, it has two options: handle the exception immediately or stop and quit.

Errors and Exceptions

- Syntax errors

Also known as parsing errors. Syntax error is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language. The error is caused by (or at least detected at) the token (statement) preceding the arrow ^ symbol.

- Exceptions

Also known as runtime errors. Whenever the interpreter has a problem it notifies the user/programmer by raising an exception. By default, the interpreter handles exceptions by stopping the program and printing an error message. However, we can override this behavior by catching the exception.

Errors and Exceptions

```
string = "Python Exceptions"

for s in string:
    if (s != o):
        print(s)
```

```
string = "Python Exceptions"

for s in string:
    if (s != o):
        print(s)
```

Statement “try…except”

The try...except block is used to handle exceptions in Python. Here's the syntax of try...except block:

```
try:
    # code that may cause exception
except:
    # code to run when exception occurs
```

Here, we have placed the code that might generate an exception inside the try block. Every try block is followed by an except block.
When an exception occurs, it is caught by the except block. The except block cannot be used without the try block.

Statement “try…except”

The try statement works as follows.

- First, the try clause (the statement(s) between the try and except keywords) is executed.
- If no exception occurs, the except clause is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the except keyword, the except clause is executed, and then execution continues after the try statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer try statements; if no handler is found, it is an unhandled exception and execution stops program with an error.

Case statement

- Ignore exception

If we do nothing, the program stops. We can do this in two ways—don't use a try statement in the first place, or don't have

- Log exception

We can write a message and let it propagate; generally this will stop the program.

- Recover from exception

We can write an except clause to do some recovery action to undo the effects of something that was only partially completed in the try clause. We can take this a step further and wrap the try statement in a while statement and keep retrying until it succeeds.

- Silence exception

If we do nothing (that is, pass) then processing is resumed after the try statement. This silences the exception.

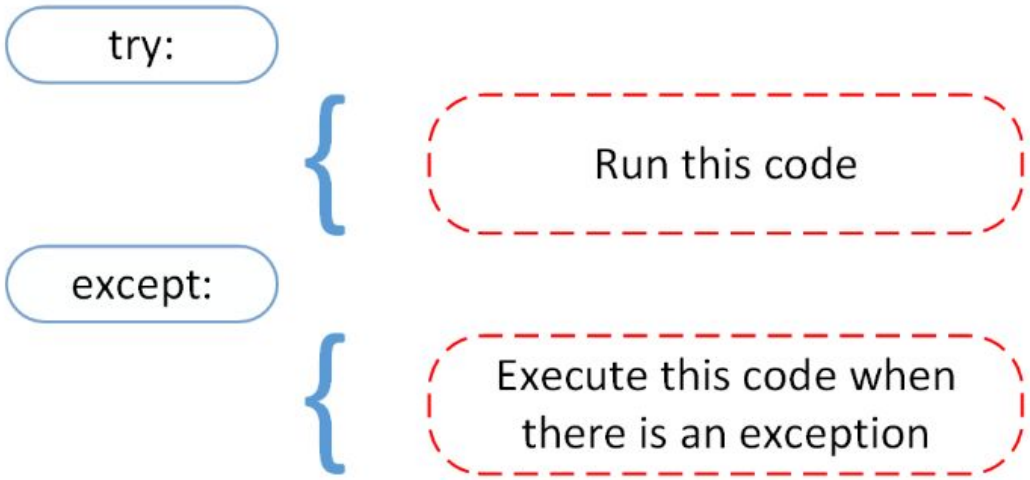
- Rewrite exception

We can raise a different exception. The original exception becomes a context for the newly-raised exception.

- Chain exception

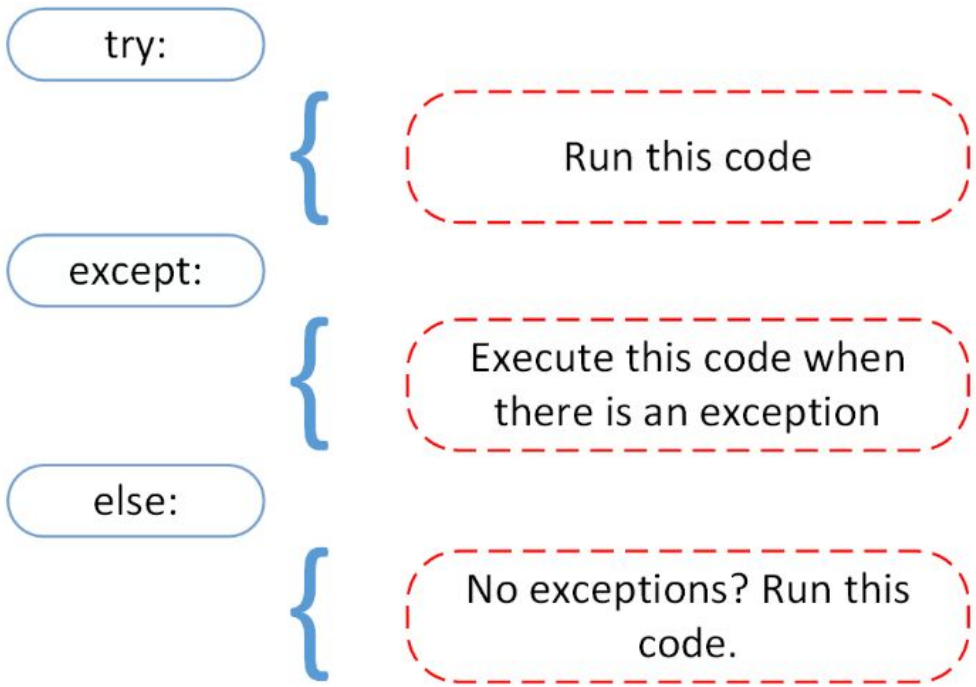
We chain a different exception to the original exception a matching except clause in the try statement.

try…except



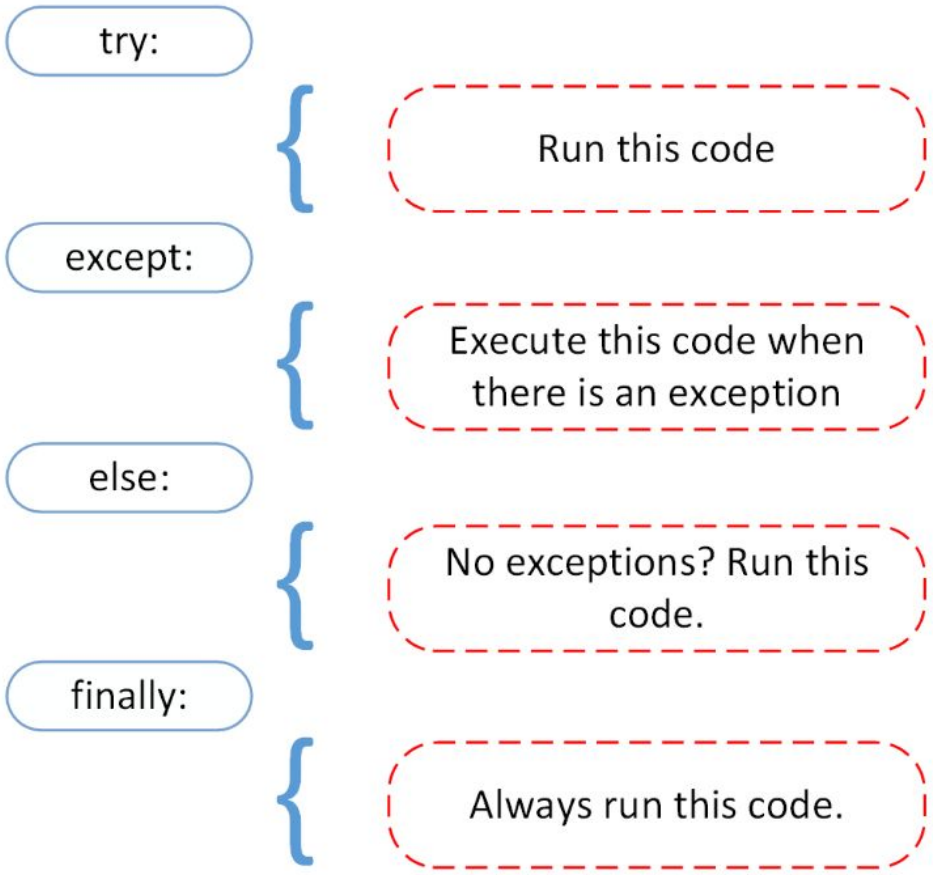

```
while True:
    try:
        x = int(input('Please enter a number: '))
        break
    except ValueError as error:
        print('Oops! Try again...', error, sep = '\n')
```

try…except…else




```
while True:
    try:
        x = input('Please enter a file name: ')
        f = open(x, 'r')
    except OSError as error:
        print ('Oops! Cannot open file. Try again...', error, sep = '\n')
    else:
        print(f'{x} has {len(f.readlines())} lines')
        f.close()
        break
```

try...except...else...finally




```
f = open('test.txt', 'r')
int_lines = []
i = 0
try:
    for item in f:
        int_lines.append(int(item))
        i += 1
except ValueError as error:
    print (f'Oops! {i+1}-th line in file is not Integer Number')
else:
    print(f'File hase {len(int_lines)} Integer Numbers')
finally:
    f.close()
```

Handling Multiple Exceptions

```
import sys

try:
    f = open('test.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print(f'OS error: {err}')
except ValueError:
    print('Could not convert data to an integer')
except:
    print('Unexpected error', sys.exc_info()[0])
    raise
```

Handling Multiple Exceptions

```
import sys

try:
    f = open('test.txt')
    s = f.readline()
    i = int(s.strip())
except (OSError, ValueError, RuntimeError) as err:
    print(f'Error: {err}')
```

Raising Exceptions

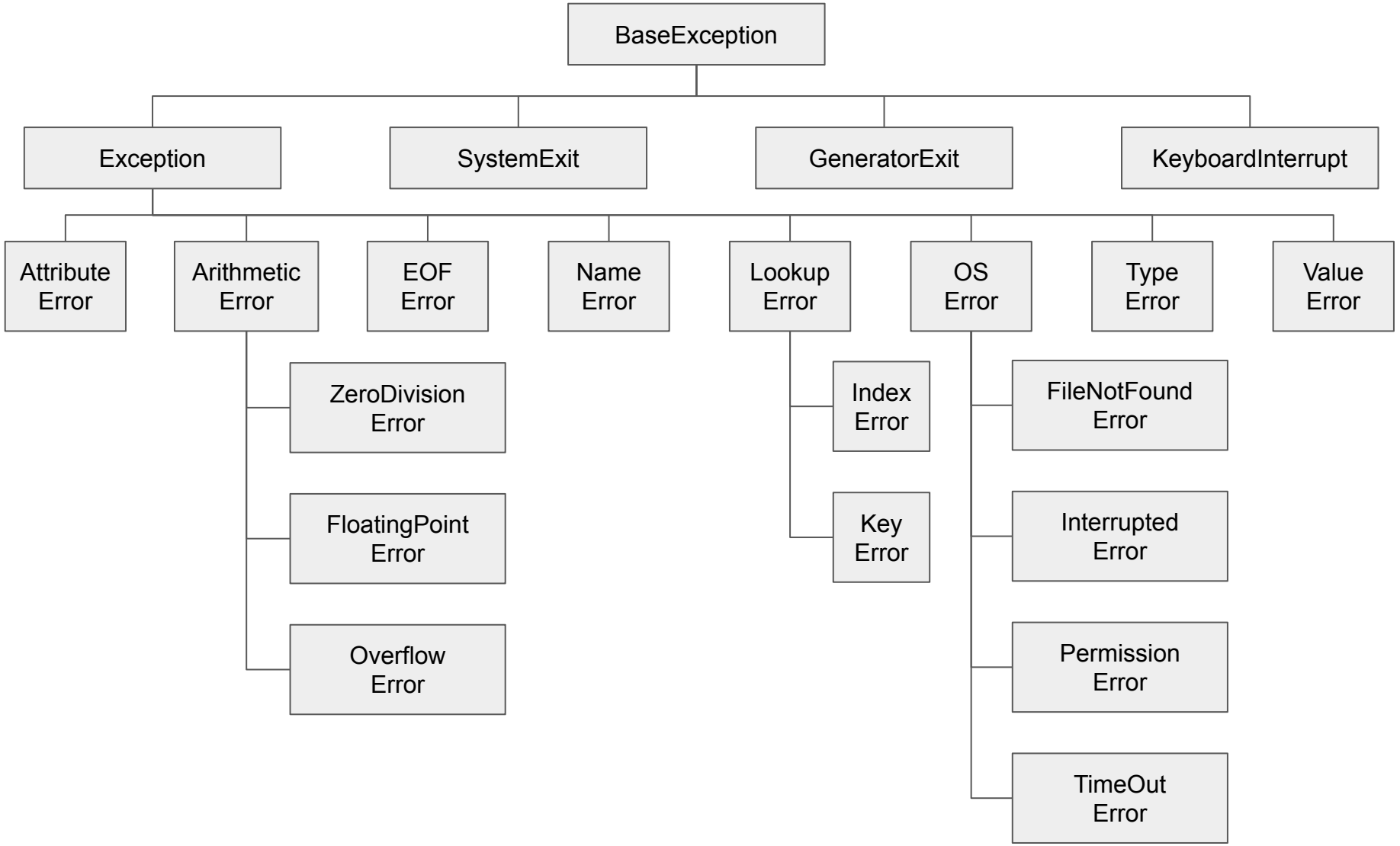
If a condition does not meet our criteria but is correct according to the Python interpreter, we can intentionally raise an exception using the raise keyword. We can use a customized exception in conjunction with the statement.

If we wish to use raise to generate an exception when a given condition happens, we may do so as follows:

```
num = [3, 4, 5, 7]
if len(num) > 3:
    raise Exception( f"Length of list must be <= 3 but is {len(num)}" )
```

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeException('Something bad happened') from exc
```

Python Exceptions List




```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

Python Exceptions List

Exception	All exceptions of Python have a base class.
StopIteration	If the next() method returns null for an iterator, this exception is raised.
SystemExit	The sys.exit() procedure raises this value.
StandardError	Excluding the StopIteration and SystemExit, this is the base class for all Python built-in exceptions.
ArithmeticError	All mathematical computation errors belong to this base class.
OverflowError	This exception is raised when a computation surpasses the numeric data type's maximum limit.
FloatingPointError	If a floating-point operation fails, this exception is raised.
ZeroDivisionError	For all numeric data types, its value is raised whenever a number is attempted to be divided by zero.

Python Exceptions List

AssertionError	If the Assert statement fails, this exception is raised.
AttributeError	This exception is raised if a variable reference or assigning a value fails.
EOFError	When the endpoint of the file is approached, and the interpreter didn't get any input value by raw_input() or input() functions, this exception is raised.
ImportError	This exception is raised if using the import keyword to import a module fails.
KeyboardInterrupt	If the user interrupts the execution of a program, generally by hitting Ctrl+C, this exception is raised.
LookupError	LookupErrorBase is the base class for all search errors.
IndexError	This exception is raised when the index attempted to be accessed is not found.
KeyError	When the given key is not found in the dictionary to be found in, this exception is raised.

Python Exceptions List

NameError	This exception is raised when a variable isn't located in either local or global namespace.
UnboundLocalError	This exception is raised when we try to access a local variable inside a function, and the variable has not been assigned any value.
EnvironmentError	All exceptions that arise beyond the Python environment have this base class.
IOError	If an input or output action fails, like when using the print command or the open() function to access a file that does not exist, this exception is raised.
SyntaxError	This exception is raised whenever a syntax error occurs in our program.
IndentationError	This exception was raised when we made an improper indentation.
SystemExit	This exception is raised when the sys.exit() method is used to terminate the Python interpreter. The parser exits if the situation is not addressed within the code.
TypeError	This exception is raised whenever a data type-incompatible action or function is tried to be executed.
ValueError	This exception is raised if the parameters for a built-in method for a particular data type are of the correct type but have been given the wrong values.
RuntimeError	This exception is raised when an error that occurred during the program's execution cannot be classified.
NotImplementedError	If an abstract function that the user must define in an inherited class is not defined, this exception is raised.

User-defined Exception

- Programs may name their own exceptions by creating a new exception class
- Exceptions should typically be derived from the Exception class, either directly or indirectly
- Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception
- When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions

User-defined Exception

```
class InputError(Exception):
    """Exception raised for errors in the table/

    Attributes:
        expression -- input expression in which the error occured
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

x = int(input('Enter a integer number between 1 and 10: '))
try:
    if x not in range(1, 11):
        raise InputError(x, 'Invalid input entered')
except InputError as err:
    print(err)
```

The assert Statement

Python examines the adjacent expression, preferably true when it finds an assert statement. Python throws an AssertionError exception if the result of the expression is false.

Python uses ArgumentException, if the assertion fails, as the argument for the AssertionError. We can use the try-except clause to catch and handle AssertionError exceptions, but if they aren't, the program will stop, and the Python interpreter will generate a traceback.

- Assertions are for debugging – not errors
- An assertion is conditionally raising an exception - AssertionError.
- You can choose to catch it or not, inside a try or not.
- Don’t use assert statements to guard against pieces of code that a user shouldn’t access

The assert Statement

```
import sys

try:
    number = int(input('Please input a positive number less than 10: '))
    assert number > 0 and number < 10, 'Number out of range'
except ValueError:
    print('You don`t khow what a number is!')
    sys.exit(1)
except AssertionError as err:
    print(str(err))
```

EAFP

LBYL (Look Before You Leap)

```
if key in some_dict:
    return some_dict[key]
else:
    return None
# do something else
```

EAFP (Easier to Ask for Forgiveness than Permission)

```
try:
    return some_dict[key]
except KeyError:
    return None
    # do something else
```

- Code is more readable
- Python's exceptions work fast enough
- Guard you against a race condition

EAFP

EAFP (Easier to Ask for Forgiveness than Permission):

- IO operations (Hard drive and Networking)
- Actions that will almost always be successful
- Database operations (when dealing with transactions and can rollback)
- Fast prototyping in a throw away environment

LBYL (Look Before You Leap):

- Irrevocable actions, or anything that may have a side effect
- Operation that may fail more times than succeed
- When an exception that needs special attention could be easily caught beforehand