EXCEPTION HANDLING

- **Syntax Errors**

Syntax errors – usually the easiest to spot, syntax errors occur when you make a typo. Not ending an **if** statement with the colon is an example of an syntax error, as is misspelling a Python keyword (e.g. using **whille** instead of **while**). Syntax error usually appear at compile time and are reported by the interpreter.

Example:

```
x = int(input('Enter a number: '))


while x%2 == 0:
    print('You have entered an even number.')
else:
    print ('You have entered an odd number.')
```

output:

```
File "<ipython-input-7-7d22d202663a>",line 3
while x%2 == 0:
^ SyntaxError: invalid syntax
```

Note: Here, that the keyword **whille** is misspelled. If we try to run the program

- **Runtime Errors**

  If a program is free of syntax errors, it will be run by the Python interpreter. However, the program may exit if it encounters a runtime error – a problem that went undetected when the program was parsed, but is only revealed when the code is executed.

- **What is BUG?**

  a bug is an error in a software program.Bug means which found by the testing teams & it's name as bug or issue

- **What is Exception?**

  Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* .

- **Need of Exception handling**

An exception is an error that occurred during program execution , and disrupts the normal flow of program. To handle such errors by restricting them to harm the program to great extent, exception handling is used. These exception handlers tackle three type of errors:-

• Checked or compile time exception

• Unchecked or Runtime exception

• Exception or logical errors

- Predefined Exceptions
- **exception BaseException**
  This is the base class for all built-in exceptions. It is not meant to be directly

inherited by user-defined classes. For, user-defined classes, Exception is used. This class is responsible for creating a string representation of the exception using str() using the arguments passed. An empty string is returned if there are no arguments.

args : The args are the tuple of arguments given to the exception constructor.

with_traceback(tb) : This method is usually used in exception handling. This method sets tb as the new traceback for the exception and returns the exception object.

Code :

```
try:
    ...
except SomeException:
    tb = sys.exc_info()[2]
    raise OtherException(...).with_traceback(tb)
```

- **exception Exception**
  This is the base class for all built-in non-system-exiting exceptions. All user-defined exceptions should also be derived from this class.
- **exception ArithmeticError**
  This class is the base class for those built-in exceptions that are raised for various arithmetic errors such as :
  - OverflowError
  - ZeroDivisionError
  - FloatingPointError

Example:

```
try:
    a = 10/0
    print(a)
except ArithmeticError:
        print("This statement is raising an arithmetic exception.")
else:
    print("Success.")
```

Output:

```
This statement is raising an arithmetic exception
```

- **exception BufferError**
  This exception is raised when buffer related operations cannot be performed.
- **exception LookupError**
  This is the base class for those exceptions that are raised when a key or index used on a mapping or sequence is invalid or not found. The exceptions raised are :
  - KeyError
  - IndexError

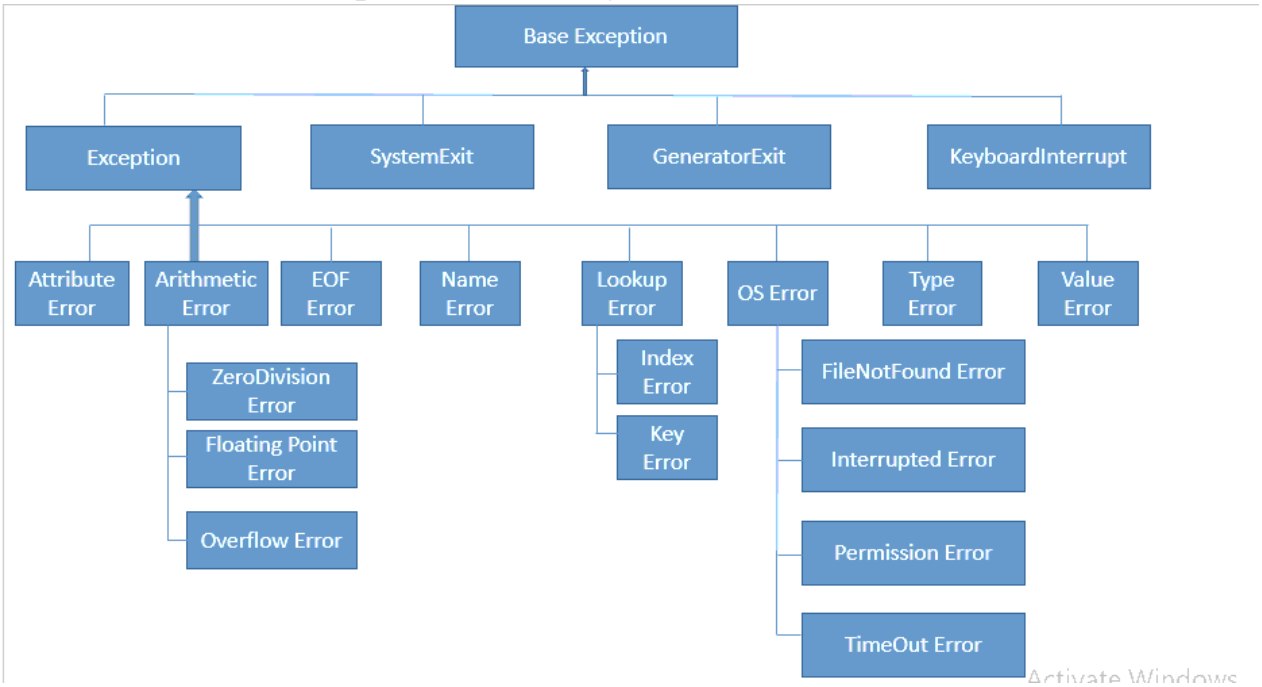
Example:

```
try:
    a = [1, 2, 3]
    print(a[3])
except LookupError:
    print("Index out of bound error.")
else:
    print("Success")
```

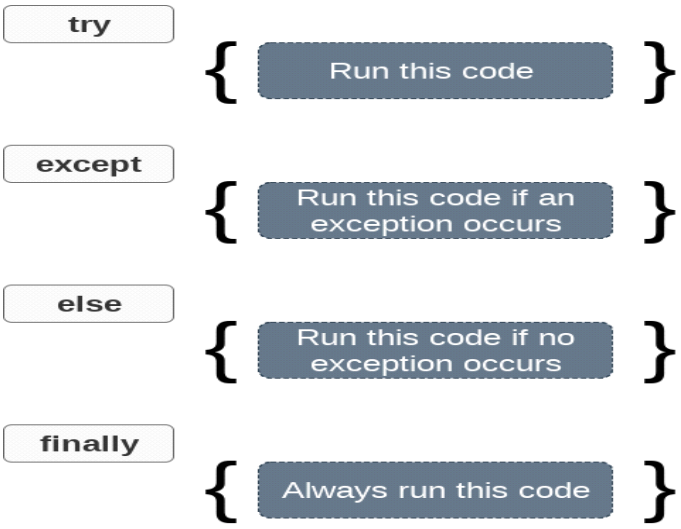
output:

```
Index out of bound error
```

- **Predefined Exceptions Hierarchy**



- **try, except and finally clauses**



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:

How try() works?

- First **try** clause is executed i.e. the code between **try** and **except** clause.
- If there is no exception, then only **try** clause will run, **except** clause is finished.
- If any exception occured, **try** clause will be skipped and **except** clause will run.
- If any exception occurs, but the **except** clause within the code doesn't handle it, it is passed on to the outer **try** statements. If the exception left unhandled, then the execution stops.
- A **try** statement can have more than one **except** clause
  Example:

```
try:
    def fun(x,y):
        z=x/y
        print("division",z)
```

```
        fun(10,0)

    except ZeroDivisionError as e:
        print("Error:",e)

    except ValueError as e:
        print("ValueError exception caught:",e)
    except TypeError as e:
        print("TypeError Error:",e)
    except Exception as e:
        print("exception:",e)
    finally:
        print("finally successfully executed")

    output:
    Error: division by zero
    finally successfully executed
```

- **Handling Multiple Exceptions**

• Given a piece of code that can throw any of several different exceptions, and one needs to account for all of the potential exceptions that could be raised without creating duplicate code or long, meandering code passages.

• If you can handle different exceptions all using a single block of code, they can be grouped together in a tuple

Example:

```
try:
    client_obj.get_url(url)
except (URLError, ValueError, SocketTimeout):
    client_obj.remove_url(url)
```

The `remove_url()` method will be called if any of the listed exceptions occurs. If, on the other hand, if one of the exceptions has to be handled differently, then put it into its own except clause as shown in the code given below :

- **Nested try, except and finally blocks**

We can take try-except-finally blocks inside try or except or finally. Hence nesting of try-except-finally blocks is possible.

Generally risky code we have to take inside outer try block and too much risky code we have to take inside inner try block.

Inside inner try block if an exception raised then inner except block is responsible to handle. If it is unable to handle then outer except block is responsible to handle.

EXAMPLE:

```
try:
    print("Outer try block")
    #print(10/0)
    try:
        print("Inner try block")
        print(10/0)

    except ValueError:
        print("inner except block")
    finally:
        print("inner finally block")

except:
    print("outer except block")

finally:
    print("outer finally block")
```

output:

Outer try block
Inner try block
inner finally block
outer except block
outer finally block