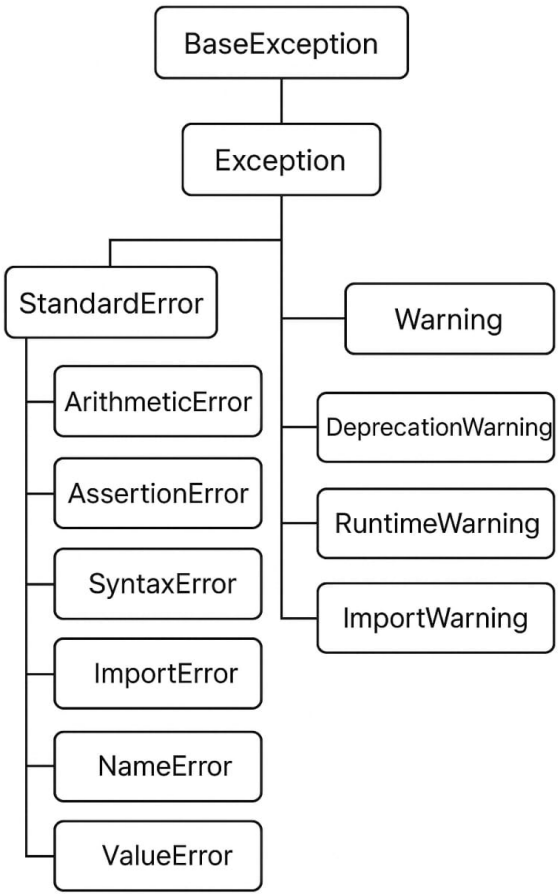
EXCEPTION HANDLING

DEFINITION AND EXPLANATION

a. The errors which are occurred at runtime are called runtime errors
b. Runtime errors are also called as exceptions
c. These errors are mainly occurred when we pass the wrong input values
d. These errors are identified by the Python Virtual Machine (PVM)
e. Whenever the runtime error occurs, the python will create an exception object. If it is not handled then python prints a traceback to that error with description about that error like why that error occurred
f.



g. All the exceptions are the classes in python
h. The base class for all the built in exceptions is "BaseException" class
i. If we don't handle the exception the program will not be executed and it leads to abnormal condition
j. A warning represents a caution to the user, even though it is not handled the program will be executed
k. TYPES OF EXCEPTIONS
    i. Built-in exceptions: The exceptions which are raised automatically by the PVM are called Built-in exceptions
    ii. User-defined exceptions: The exceptions which are raised manually by the user by using "raise" keyword are called User-defined exceptions
l. If you want to create the user-defined exception then you have to extend the "Exception" class not the "BaseException" class

HANDLING THE EXCEPTION

We can handle the exception by using try, except, finally and else blocks

1. TRY BLOCK
    a. The code which may raise an exception that will be in try block
    b. In simple words, in try block we write exception related code
    c. If an exception occurs in try block, the code execution will be transferred to the except block
    d. Syntax:

```
try:
    stmts/exception_code
```

2. EXCEPT BLOCK
   a. This block is used to handle the exception that occurred in the try block
   b. When an exception occurs, python looks for the first except block whether it can handle the exception or not if it handles, the code inside the expect block will be executed, otherwise it goes for next except block and so on. If no except block will handle the exception the program will be terminated with the error message
   c. Syntax:

```
except:
    stmts/exception_handlers
```

3. ELSE BLOCK
   a. This block will be executed if no exception occurs in try block
   b. This block is optional
   c. Syntax:

```
else:
    stmts
```

4. FINALLY BLOCK
   a. This block will be executed whether the exception occurs or not
   b. This block is used to perform the cleanup actions like closing the file, database connections, etc
   c. In simple words, in this block we write the memory related
   d. In all cases the finally block will be execute but in one case it won't be execute i.e., if we manually exit from the program

e. Syntax:

```
finally:
    stmts
```

IMPORTANT POINTS:

- ⇨ Try block can be alone
- ⇨ Except block can't be alone it must follow the try block
- ⇨ Else and finally block is optional
- ⇨ Finally block can follow after try and except block
- ⇨ For one try block there should be only one finally block

EXAMPLE FOR EXCEPTION HANDLING:

```
try:
    num1=eval(input("enter the number 1"))
    num2=eval(input("enter the number 2"))
    print(num1/num2)
except ZeroDivisionError as e:
    print(e)
else:
    print("i am else block")
finally:
    print("iam finally block")
```

- ⇨ In this example "e" holds the description about the exception. It is the default message/description
- ⇨ We can also write our own message

  ex:

```
except ZeroDivisionError:
    print("second number cannot be zero")
```

HANDLING THE EXCEPTION IN DIFFERENT WAYS

⇨ We can handle the exception:

Syntax:

```
except Exceptionclassname:
        stmts
```

ex:

```
except ZeroDivisionError:
    print("second number cannot be zero")
```

⇨ We can handle the exception as an obj that contains the description about the exception

Syntax:

```
except Exception_class_name as obj:
        print(e)
```

ex:

```
except ZeroDivisionError as e:
    print(e)
```

⇨ We can handle the multiple exceptions by passing the all exceptions as a tuple

Syntax:

```
except(exceptionclass1, exceptionclass2….):
```

ex:

```
except(ZeroDivisionError,ValueError) as e:
        print(e)
```

⇨ We can handle the all the exceptions without mentioning the exception classes

- But it is not recommended

Syntax:

expect Exception as e:

print(e)

- ⇨ It can handle all the exceptions
- ⇨ It is called as default exception
- ⇨ It should be placed at last of the program

RAISE STATEMENT

- ⇨ The raise statement is used to raise the exception manually
- ⇨ By using raise statement, we can raise the built-in and user-defined exceptions
- ⇨ Syntax:

raise Exception_class_name(“error message”)

ex:

```
num1=eval(input("enter the number1"))
num2=eval(input("enter the number2"))
def divide(num1,num2):
    if num2==0:
        raise ZeroDivisionError("second number cannot
be zero")
    else:
        return num1/num2
try:
  result=divide(num1,num2)
  print(result)
except ZeroDivisionError as e:
  print(e)
```

USER DEFINED EXCEPTION

- ⇨ The exceptions which are created by the user/programmer are called user defined exceptions
- ⇨ If you want to create the user-defined exception then you have to extend the “Exception” class not the “BaseException” class
- ⇨ Syntax:

```
Class Exception_class_name(Exception):
        def __init__(self,msg):
            self.msg=msg
```

ex:

```
class TooYoung(Exception):
    def __init__(self,msg):
            self.msg=msg
class TooOld(Exception):
    def __init__(self,msg):
            self.msg=msg
age=int(input("enter the age"))
if age<18:
    raise TooYoung("ur too young")
elif age>60:
    raise TooOld("ur too old")
```