Exception Handling:

Python indicates errors and exceptional conditions by raising exceptions,

Catching and Raising Exceptions

Exceptions are caught using **try** …**except** blocks, whose general syntax is:

```
try:
    try_suite
except exception_group1 as variable1:
    except_suite1
…
except exception_groupN as variableN:
    except_suiteN
else:
    else_suite
finally:
    finally_suite
```

There must be at least one **except** block, but both the **else** and the **finally** blocks are optional.

The **else** block's suite is executed when the **try** block's suite has finished normally—but it is not executed if an **exception** occurs.

If there is a **finally** block, it is always executed at the end.

Each **except** clause's exception group can be a single exception or a parenthesized Tuple of exceptions.

For each group, the **as** *variable* part is optional; if used, the variable contains the exception that occurred, and can be accessed in the exception block's suite.

If an exception occurs in the **try** block's suite, each **except** clause is tried in turn. If the exception matches an exception group, the corresponding suite is executed. To match an exception group, the exception must be of the same type as the exception type as listed in the group.

Raising Exceptions |manually

Exception Handling:

Exceptions provide a useful means of changing the flow of control. We can take advantage of this either by using the built-in exceptions, or by creating our own, raising either kind when we want to.

There are three syntaxes for raising exceptions:

raise *exception*(*args*)

raise *exception*(*args*) from *original_exception*

raise

When the first syntax is used the exception that is specified should be either one of the built-in exceptions, or a custom exception that is derived from Exception.

If we give the exception some **text as its argument**, this text will be output if the exception is printed when it is caught.

The second syntax is a variation of the first—the Chained exceptions exception is raised as a chained exception that includes the *original_exception* exception, so this syntax is used inside except suites.

When the third syntax is used, that is, when no exception is specified, raise will re-raise **the currently active exception**—and if there isn't one it will raise a TypeError.

To trigger an exception manually, simply run a raise statement. User-triggered exceptions are caught the same way as those Python raises.:

```
>>> try:
        raise IndexError          # Trigger exception manually
except IndexError:
        print('got IndexError exception')
...
got exception
```

As usual, if they're not caught, user-triggered exceptions are propagated up to the toplevel default exception handler and terminate the program with a standard error message:

```
>>> raise IndexError
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError
```

EXCEPTIONS PROPAGATE UP

In the below example: How exception could propagate up the call stack.

```
#this example show the propagation of error up the call stack

>>>def add(x):
        return validate(x) + 1

>>>def validate(x):
        if x < 0: raise ValueError
        else: return x

>>>try:
        a = int(input ("Enter a Negative Number to raise the error:"))
        y=(add(a))
        print("The result is %d" %y)
except ValueError:
        print("You entered an invalid number.")
```

The output is as follows:

```
>>>  Enter a Negative Number to raise the error:10
11
The result is 10
>>>  Enter a Negative Number to raise the error:-5
You entered an invalid number.
>>>
```

In the above example,

Exception Handling:

 the print statement calls the function add(x). That function calls the function validate(x), which will raise an exception of type ValueError.

 Neither add(x) nor validate(x) has a try/except block to handle ValueError. So the exception raised propagates out to the main code, where there is an exception handling block waiting for it.

 So when the user had entered -5 the code prints: That value was invalid.

Consider the following program, which has a “divide-byzero” error.

```
>>>def spam(divideBy):
        return 42 / divideBy

>>>print(spam(2))
>>>print(spam(12))
>>>print(spam(0))
>>>print(spam(1))
```

We’ve defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5

Traceback (most recent call last):
File "C:/zeroDivide.py", line 6, in <module>
print(spam(0))
File "C:/zeroDivide.py", line 2, in spam
return 42 / divideBy
ZeroDivisionError: division by zero
```

A ZeroDivisionError happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the return statement in spam() is causing an error.

Exception Handling:

FINALLY

Defining Clean-up Actions: The try statement has another optional clause "***finally***" which is intended to define clean-up actions that must be executed under all circumstances. A **finally** clause is always executed before leaving the **try** statement, whether an exception has occurred or not. When an **exception** has occurred in the **try** clause and has not been handled by an **except** clause (or it has occurred in a **except** or **else** clause), it is re-raised after the finally clause has been executed. The **finally** clause is also executed "on the way out" when any other clause of the try statement is left via a break, continue or return statement.

Let' see an example:

```
>>>def divide(x, y):
        try:
        result = x / y
        except ZeroDivisionError:
        print("Division by zero!")
        else:
        print("result is", result)
        finally:
        print("Executing finally clause")


>>>divide(2, 0)
```

The output is as follows:

```
>>>
Division by zero!
Executing finally clause
```

Exception Handling:

```
try:
   x= int(input( 'positive number ?... '))
   if x<=0:
      raise ValueError

except ValueError:
   print('should be positive ..')
else:
   print('..else block..')


print('-----------')
print('remaining statements')
print('-----------')
```

```
try:
   x= int(input( 'positive number ?... '))
   if x<=0:
      raise ValueError

except ValueError:
   print('should be positive ..')
else:
   print('..else block..')
finally:
   print('...finally block...')


print('-----------')
print('remaining statements')
print('-----------')
```