Lecture 13 – Exception

So far you have encountered several kinds of run-time errors, such as integer division by zero and accessing a list with an out-of-range index.

To this point, all of our run-time errors have resulted in **the program's termination.**

Python provides a standard mechanism **called exception handling** that allows programmers to deal with these kinds of run-time errors and many more.

Rather than always terminating the program's execution, a program can detect the problem and execute code to correct the issue or manage it in other ways.

Many of these potential problems can be handled by the algorithm itself.

For example, **an if statement** can test to see if a list index is within the bounds of the list. However, if the list is accessed at many different places within a function, the large number of conditionals in place to ensure the list access safety can quickly obscure the overall logic of the function. Fortunately, specific Python exceptions are available to cover problems such as these.

So, Python handles all errors with exceptions.

An **exception** is a signal that an error or other unusual condition has occurred. There are a number of built-in exceptions, which indicate conditions like reading past the end of a file, or dividing by zero.

You can also define your own exceptions.

Raising exceptions

Whenever your program attempts to do something erroneous or meaningless, Python raises exception to such conduct:

```
>>> 1 / 0

Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
```

This *traceback* indicates that the ZeroDivisionError exception is being raised. This is a built-in exception -- see below for a list of all the other ones.

Example

```
x = int(input("\nPlease enter a small positive integer: "))
print("x =", x)
```

Sample Run

```
Please enter a small positive integer: Five

Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    x = int(input("Please enter a small positive integer: "))
ValueError: invalid literal for int() with base 10: 'Five'
```

Catching exceptions

In order to handle errors, you can set up exception handling blocks in your code. You can wrap the code in a try/except construct.

The keywords **try and except** are used to catch exceptions.

When an error occurs within the try block, Python looks for a matching except block to handle it. If there is one, execution jumps there.

Example

```
try:
  print (1/0)
except ZeroDivisionError:
  print ("You can't divide by zero.  ")
```

Sample Run

```
You can't divide by zero.
```

Example

```
try:
   x = int(input("\nPlease enter a small positive integer:   "))
   print("x =", x)
except ValueError:
   print("Input cannot be parsed as an integer")
```

Sample run

```
Please enter a small positive integer:   five
Input cannot be parsed as an integer
```

If you don't specify an exception type on the except line, it will catch all exceptions.

Exceptions can propagate up the call stack.

Example

```
def f(x):
  return g(x) + 1

def g(x):
  if x < 0:
    raise ValueError
    print("I can't cope with a negative number here.")
  else: return 5
try:
  print (f(-6))
except ValueError:
  print ("That value was invalid.")
```

In this code, the print statement calls the function f.
That function calls the function g, which will raise an exception of type ValueError.

Neither f nor g 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. This code prints:

```
That value was invalid.
```

Recovering and continuing with finally

Exceptions could lead to a situation where, after raising an exception, the code block where the exception occurred might not be revisited. In some cases, this might leave external resources used by the program in an unknown state.

`finally` clause allows programmers to close such resources in case of an exception.

Example

```
try:
  result = None
  try:
    result = 1/0
  except ZeroDivisionError:
    print ("division by zero!" )
  print ("result is ", result )

finally:
  print ("executing finally clause")
```

Sample run

```
division by zero!
result is  None
executing finally clause


//   when  1 / 3

result is  0.3333333333333333
executing finally clause
```

Example

```
try:
  result = 1 / 0
except ZeroDivisionError:
  print ("division by zero!")
else:
  print ("result is", result )
finally:
  print ("executing finally clause" )
```

Sample Run

```
division by zero!
executing finally clause


//   when  1 / 3


result is 0.3333333333333333
executing finally clause
>>>
```

Examples Of Exceptions :

StandardError

The base class for built-in exceptions. All built-in exceptions are derived from this class, which is itself derived from the root class `Exception`.

ArithmeticError

The base class for those built-in exceptions that are raised for various arithmetic errors: **OverflowError, ZeroDivisionError, FloatingPointError.**

LookupError

The base class for those exceptions that are raised when a key or index used on a mapping or sequence is invalid: **IndexError, KeyError**.

IOError

Raised when an I/O operation (such as a `print` statement, the built-in `open()` function or a method of a file object) fails for an I/O-related reason, e.g., ``file not found" or ``disk full".

NameError

Raised when a local or global name is not found. This applies only to unqualified names. The associated value is the name that could not be found.

OverflowError

Raised when the result of an arithmetic operation is too large to be represented. This cannot occur for long integers (which would rather raise `MemoryError` than give up). Because of the lack of standardization of floating point exception handling in C, most floating point operations also aren't checked. For plain integers, all operations that can overflow are checked except left shift, where typical applications prefer to drop bits than raise an exception.

ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero. The associated value is a string indicating the type of the operands and the operation.

What is the output of the following program

```
for x in range(10):
    if x > 5:
        raise ValueError("'x'should not exceed 5.")
    print(x)
```

What is the output of the following program

```
for x in range(10):
    try:
        if x > 5:
            raise ValueError
        print(x)
    except ValueError:
        print("'x'should not exceed 5.")
```

What is the output of the following program

```
a = [12, 25, 39]
try:
    print ("Second Item = " ,  (a[1]))
    print ("Fourth Item = " ,  (a[3]))
except:
    print ("An error occurred")
```

What is the output of the following program if the user entered W followed by Z followed by 12

```
def main():
    x = get_int()
    print("x is " , x)


def get_int():
    while True:
        try:
            return int(input("What's x? "))
        except ValueError:
            print("x is not an integer")



main()
```