Python Exceptions

Python Logical Errors

Errors that occur at runtime (after passing the syntax test) are called **exceptions** or **logical errors**.

For instance, they occur when we try to open a file(for reading) that does not exist, try to divide a number by zero , or try to import a module that does not exist.

Whenever these types of runtime errors occur, Python creates an exception object.

If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

Exceptions in Python

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

If we don’t handle the exceptions, an error message is displayed and our program comes to a sudden unexpected halt.

Catching Exceptions

In Python, exceptions can be handled using a ‘try’ statement.

The critical operation which can raise an exception is placed inside the ‘try’ clause. The code that handles the exceptions is written in the ‘except’ clause.

We can thus choose what operations to perform once we have caught the exception.

```
try:
    #block of code
except Exception2:
    #block of code
except:
    #block of code
```

Catching Exceptions

```
try:
x = input('Enter the first number: ')
y = input('Enter the second number: ')
print x/y
except ZeroDivisionError:
print "The second number can't be zero!"
```

More Than One except Clause

If you run the program from the previous section again and enter a nonnumeric value at the prompt, another exception occurs:

```
Enter the first number: 10
Enter the second number: "Hello, world!"
Traceback (most recent call last):
File "exceptions.py", line 4, in ?
print x/y
TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

More Than One except Clause

```
try:
x = input('Enter the first number: ')
y = input('Enter the second number: ')
print x/y
except ZeroDivisionError:
print "The second number can't be zero!"
except TypeError:
print "That wasn't a number, was it?"
```

Catching Two Exceptions with One Block

If you want to catch more than one exception type with one block, you can specify them all in

a tuple

```
try:
x = input('Enter the first number: ')
y = input('Enter the second number: ')
print x/y
except (ZeroDivisionError, TypeError, NameError):
print 'Your numbers were bogus...'
```

Raising Exceptions

In Python programming, exceptions are raised when errors occur at runtime.

We can also manually raise exceptions using the raise keyword.

We can optionally pass values to the exception to clarify why that exception was raised.

```
>>> try:
...     a = int(input("Enter a positive integer: "))
...     if a <= 0:
...         raise ValueError("That is not a positive number!")
... except ValueError as ve:
...     print(ve)
```

Python try with else clause

In some situations, you might want to run a certain block of code if the code block inside try ran without any errors.

For these cases, you can use the optional else keyword with the try statement.

```
try:
    num = int(input("Enter a number: "))
    assert num % 2 == 0
except:
    print("Not an even number!")
else:
    reciprocal = 1/num
    print(reciprocal)
```

try...finally

- The try statement in Python can have an optional finally clause.
- This clause is executed no matter what, and is generally used to release external resources.
- For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).
- In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not.
- These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

try...finally

```
try:
1/0
except NameError:
print "Unknown variable"
else:
print "That went well!"
finally:
print "Cleaning up."
```

Creating Custom Exceptions

- In Python, users can define custom exceptions by creating a new class.
- This exception class has to be derived, either directly or indirectly, from the built-in Exception class.

```
>>> class CustomError(Exception):
...     pass
>>> raise CustomError
Traceback (most recent call last):
...
__main__.CustomError
```

Custom Exceptions

When we are developing a large Python program, it is a good practice to place all the user-defined exceptions that our program raises in a separate file.

Many standard modules do this. They define their exceptions separately as exceptions.py or errors.py .

User-defined exception class can implement everything a normal class can do, but we generally make them simple and concise.

Most implementations declare a custom base class and derive others exception classes from this base class.