Chapter 09: Exception Handling

Exceptions

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*

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.

For example, let us consider a program where we have a calculation in which you are performing a division operation, say a/b. What happens to the program if b=0?. This is an error situation and needs to be handled.
If never handled, an error message is displayed and our program comes to a sudden unexpected halt. If you want to continue with the processing, you need to handle this situation using Exception handling.

Catching Exceptions in Python

Let us first see how the program behaves when exception conditions are ignored(Not handled). Here is a sample program

```
randomList = ['a', 0, 2]

for entry in randomList:

        print("The entry is", entry)
        r = 1/int(entry)

print("The reciprocal of", entry, "is", r)
```

In the above program we are repeatedly dividing 1 by each value of list. The first value ‘a’ will result in an exception condition as 1/a has no meaning in math. The program, when encounters the situation, it will show the error and will exit abruptly. But if you want to

continue with valid values and show error messages for invalid values, you need to handle the 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.
Here is a simple example.

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)

    except:
        print("Oops!", sys.exc_info()[0], "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

In this program, we loop through the values of the `randomList` list. As previously mentioned, the portion that can cause an exception is placed inside the `try` block.
If no exception occurs, the `except` block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the `except` block (first and second values).
Here, we print the name of the exception using the exc_info() function inside sysmodule.
We can see that a causes ValueError and 0 causes ZeroDivisionError.
We can also catch exception as follows

```
randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except Exception as e:
```

```
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", entry, "is", r)
```

Catching Specific Exceptions in Python

In the above example, we did not mention any specific exception in the `except`clause. This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an `except` clause should catch. A `try` clause can have any number of `except` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

```
try:
   # do something
   pass


except ValueError:
   # handle ValueError exception
   pass

except (TypeError, ZeroDivisionError):
   # handle multiple exceptions
   # TypeError and ZeroDivisionError
   pass
except:
   # handle all other exceptions
   pass
```

The above program can be better implemented by handling the specific exceptions

```
# import module sys to get the type of exception
from decimal import DivisionByZero

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        print("The reciprocal of", entry, "is", r)
```

```
    except(ValueError):
        print("Oops! Value Error", "occurred.")
        print("Next entry.")
        print()
    except(ZeroDivisionError):
        print("Oops! Division By Zero Error", "occurred.")
        print("Next entry.")
        print()
```

Raising Exceptions in Python

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: "))
    num % 2 == 0
    break
except:
    print("Not an even number!")
else:
    try:
        reciprocal = 1/num
        print(reciprocal)
    except:
        print("ERROR")
```

Python 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 database

In 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, disconnecting from database) are performed in the `finally` clause to guarantee the execution.

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Custom / User-defined Exceptions

Sometimes we have to define and `raise` exceptions explicitly to indicate that something goes wrong. Such a type of exception is called a **user-defined exception** or **customized exception.**

The user can define custom exceptions by creating a new class. This new exception class has to derive either directly or indirectly from the built-in class Exception. In Python, most of the built-in exceptions also derived from the Exception class.

Example:

```
class SalaryNotInRangeError(Exception):
    """Exception raised for errors in the input salary.

    Attributes:
        salary -- input salary which caused the error
        message -- explanation of the error
    """

    def __init__(self, salary,message):
        self.salary = salary
        self.message = message.format(salary)
        super().__init__(self.message)


salary = int(input("Enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary , "Salary {} is not in (5000, 15000) range")
```

Exercises

Problem : Ask user to enter amount, term and interest(%) and calculate simple interest. If the user enter an interest rate of more than 20% raise an exception(System defined) and inform the user and exit the program

Problem: You're going to write an interactive calculator! User input is assumed to be a formula that consist of a number, an operator (at least `+` and `-`), and another number, separated by white space (e.g. `1 + 1`). Split user input using str.split() and check whether the resulting `list` is valid:

- If the input does not consist of 3 elements, raise a `FormulaError`, which is a custom `Exception`.
- Try to convert the first and third input to a `float` (like so: `float_value = float(str_value)`). Catch any `ValueError` that occurs, and instead raise a `FormulaError`
- If the second input is not `'+'` or `'-'`, again raise a `FormulaError`

If the input is valid, perform the calculation and print out the result. The user is then prompted to provide new input, and so on, until the user enters `quit`.