Python Exception Handling

Python has many built-in exceptions that are raised when a program encounters an error, and most external libraries, like the popular Requests, include his own custom exceptions that we will need to deal to.

Basic exception handling

You can't divide by zero, that is a mathematical true, and if you try to do it in Python, the interpreter will raise the built-in exception ZeroDivisionError:

```
>>> def divide(dividend , divisor):
...     print(dividend / divisor)
...
>>> divide(dividend=10, divisor=5)
# 5

>>> divide(dividend=10, divisor=0)
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# ZeroDivisionError: division by zero
```

Let's say we don't want our program to stop its execution or show the user an output he will not understand. Say we want to print a useful and clear message, then we need to ***handle*** the exception with the `try` and `except` keywords:

```
>>> def divide(dividend , divisor):
...     try:
...         print(dividend / divisor)
...     except ZeroDivisionError:
...         print('You can not divide by 0')
...
>>> divide(dividend=10, divisor=5)
# 5

>>> divide(dividend=10, divisor=0)
# You can not divide by 0
```

Final code in exception handling

The code inside the `finally` section is always executed, no matter if an exception has been raised or not:

```
>>> def divide(dividend , divisor):
...     try:
...         print(dividend / divisor)
...     except ZeroDivisionError:
...         print('You can not divide by 0')
...     finally:
...         print('Execution finished')
...
>>> divide(dividend=10, divisor=5)
```

```
# 5
# Execution finished

>>> divide(dividend=10, divisor=0)
# You can not divide by 0
# Execution finished
```

Custom Exceptions

Custom exceptions initialize by creating a `class` that inherits from the base `Exception` class of Python, and are raised using the `raise` keyword:

```
>>> class MyCustomException(Exception):
...     pass
...
>>> raise MyCustomException
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# __main__.MyCustomException
```

To declare a custom exception message, you can pass it as a parameter:

```
>>> class MyCustomException(Exception):
...     pass
...
>>> raise MyCustomException('A custom message for my custom exception')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# __main__.MyCustomException: A custom message for my custom exception
```

Handling a custom exception is the same as any other:

```
>>> try:
...     raise MyCustomException('A custom message for my custom exception')
>>> except MyCustomException:
...     print('My custom exception was raised')
...
# My custom exception was raised
```