Exception Handling

Monday, January 3, 2022 2:53 PM

EXCEPTIONS

Using exceptions, one can handle potential errors that can occur in an operation. This is called exception Rasing.

An exception is an object that contains a description of what went wrong and the location of where the problem occurred referred to as a trackback.

Commonly syntax errors occur, the following is an example of a syntax error when trying to write Hello World 5 times:

```
In [1]: for i in range(5)
            print("Hello World!")

          File "C:\Users\Joe\AppData\Local\Temp/ipykernel_6924/1742204242.py", line 1
            for i in range(5)
                           ^
        SyntaxError: invalid syntax
```

As shown, the last line shows the type of error in this case being syntax.
The above traceback shows the line and location, being a chevron on that line where the error occurred.
The missing piece of code is a colon that should go after the (5).

Another common type of exception is related to arithmetic. The following example is trying to divide 1/0.

```
In [3]: 1/0

        ---------------------------------------------------------------------------
        ZeroDivisionError                         Traceback (most recent call last)
        ~\AppData\Local\Temp/ipykernel_6924/2354412189.py in <module>
        ----> 1 1/0

        ZeroDivisionError: division by zero
```

This returns a zero division error because the python interpreter knows that this operation is impossible, you cannot divide by 0.

Something similar occurs when trying to add values of different data types, see example where two numbers are integers and one is a string:

```
In [5]: 1 + 2 + "three"

        ---------------------------------------------------------------------------
        TypeError                                 Traceback (most recent call last)
        ~\AppData\Local\Temp/ipykernel_6924/3095229709.py in <module>
        ----> 1 1 + 2 + "three"

        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

This raises a type error. The error description states that you cannot add and integer and a string together.

Another illustration, import the math module and then take the square route of –1, see example:

```
In [6]: import math

        print(math.sqrt(-1))

        ---------------------------------------------------------------------------
        ValueError                                Traceback (most recent call last)
        ~\AppData\Local\Temp/ipykernel_6924/1489513527.py in <module>
              1 import math
              2
        ----> 3 print(math.sqrt(-1))

        ValueError: math domain error
```

A value error is raised because the math function expects a number but cannot handle negative values.
Note that a type error was not raised because the correct value type was provided.

Another exception is the file not found error. This is raised when trying to access a file that doesn't exist, see example:

```
In [4]: with open("x_files.txt") as xf:
            the_truth = xf.read()

        print(the_truth)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_6924/747505930.py in <module>
----> 1 with open("x_files.txt") as xf:
      2     the_truth = xf.read()
      3
      4 print(the_truth)

FileNotFoundError: [Errno 2] No such file or directory: 'x_files.txt'
```

EXCEPTION CLAUSES

The way to handle exceptions is to use the
`try:`
`pass`
`except:`
`pass`
`else:`
`pass`
`finally:`
`pass`
Constructions.

Multiple `except:` clauses can exists if necessary which gives the ability to respond to different exceptions differently.

Basically, the python interpreter executes the code in the `try:` block where it jumps to the `except:` block if an error is occurred and executes. In the event of no error being occurred, after the `try:` block is run, the python interpreter will skip all the `except:` blocks and run the `else:` code. The `finally:` block will always run regardless of error or no error.

This example is a simple exception by trying to open a .txt file that doesn't exist. In the `except:` block we specify an exception is what we are looking for.

```
In [7]: try:
            f = open("testfile1.txt")
        except Exception:
            print("Sorry this file does not exist")
```

```
Sorry this file does not exist
```

As you can see, instead of raising an error, it runs our custom code via the `except:` block. But this example is very vague and the exception can be modified to catch a specific type of error. The `except:` block can be modified to trigger when a FileNotFoundError is raised, see example:

```
In [8]: try:
            f = open("testfile1.txt")
        except FileNotFoundError:
            print("Sorry this file does not exist")
```

```
Sorry this file does not exist
```

This code return the same result however only when a specified error is found.

Next, the `else:` block will run if the `try:` block doesn't raise an exception.

In the next code example, the correct file name will be provided meaning the `try:` block will skip the `except:` block and run the `else:` block. The contents of the file will be printed out and then closed:

```
try:
    f = open("testfile.txt")
except FileNotFoundError:
    print("Sorry this file does not exist")
else:
    print(f.read())
    f.close()
```

```
Hello World
```

The contents of the file in this case being Hello World.

The `finally:` block will run regardless and it is good to include this to ensure the code releases system resources and makes sure the operation is shutdown regardless if it was successful or not. An example is in databases, the `finally:` block will ensure that the database is closed down.
In this example a `print()` method can be added:

```
try:
    f = open("testfile.txt")
except FileNotFoundError:
    print("Sorry this file does not exist")
else:
    print(f.read())
    f.close()
finally:
    print("Shutting down code")
```

```
Hello World
Shutting down code
```

As shown, the contents of the files is read and the `finally:` block is also run as well.