05-Exception Handling (Defensive programming)

October 31, 2020

Coursework delivered by: Alison Mukoma

1 Exceptions

When something goes wrong an exception is raised. For example, if you try to divide by zero, `ZeroDivisionError` is raised or if you try to access a nonexistent key in a dictionary, `KeyError` is raised.

[ ]:
```
empty_dict = {}
# empty_dict['key']  # Uncomment to see the traceback
```

1.1 `try-except` structure

If you know that a block of code can fail in some manner, you can use `try-except` structure to handle potential exceptions in a desired way.

[ ]:
```
# Let's try to open a file that does not exist
file_name = 'not_existing.txt'

try:
    with open(file_name, 'r') as my_file:
        print('File is successfully open')

except FileNotFoundError as e:
    print('Uups, file: {} not found'.format(file_name))
    print('Exception: {} was raised'.format(e))
```

If you don't know the type of exceptions that a code block can possibly raise, you can use `Exception` which catches all exceptions. In addition, you can have multiple `except` statements.

[ ]:
```
def calculate_division(var1, var2):
    result = 0

    try:
```

```
        result = var1 / var2
    except ZeroDivisionError as ex1:
        print("Can't divide by zero")
    except Exception as ex2:
        print('Exception: {}'.format(ex2))

    return result

result1 = calculate_division(3, 3)
print('result1: {}'.format(result1))

result2 = calculate_division(3, '3')
print('result2: {}'.format(result2))

result3 = calculate_division(3, 0)
print('result3: {}'.format(result3))
```

`try-except` can be also in outer scope:

[ ]:
```
def calculate_division(var1, var2):
    return var1 / var2

try:
    result = calculate_division(3, '3')
except Exception as e:
    print(e)
```

[1]:
```
print 'Hello
```

```
  File "<ipython-input-1-23e01f0d17c8>", line 1
    print 'Hello
                ^
SyntaxError: EOL while scanning string literal
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. 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 and are not unconditionally fatal.

You can check out the full list of built-in exceptions here. now lets learn how to handle errors and exceptions in our own code.

##try and except

The basic terminology and syntax used to handle errors in Python is the **try** and **except** statements. The code which can cause an exception to occue is put in the *try* block and the handling of the exception is the implemented in the *except* block of code. The syntax form is:

```
try:
   You do your operations here...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ...
else:
   If there is no exception then execute this block.
```

We can also just check for any exception with just using except: To get a better understanding of all this lets check out an example: We will look at some code that opens and writes a file:

[11]:
```
try:
    f = open('testfile','w')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print␣
 ↪statement
   print "Error: Could not find file or read data"
else:
   print "Content written successfully"
   f.close()
```

```
Content written successfully
```

Now lets see what would happen if we did not have write permission (opening only with 'r'):

[14]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print␣
 ↪statement
   print "Error: Could not find file or read data"
else:
   print "Content written successfully"
   f.close()
```

```
Error: Could not find file or read data
```

Great! Notice how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said except: if we weren't sure what exception would occur. For example:

[13]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print statement
   print "Error: Could not find file or read data"
else:
   print "Content written successfully"
   f.close()
```

```
Error: Could not find file or read data
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where **finally** comes in. ##finally The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

For example:

[16]:
```
try:
   f = open("testfile", "w")
   f.write("Test write statement")
finally:
   print "Always execute finally code blocks"
```

```
Always execute finally code blocks
```

We can use this in conjunction with except. Lets see a new example that will take into account a user putting in the wrong input:

[33]:
```
def askint():
        try:
            val = int(raw_input("Please enter an integer: "))
        except:
            print "Looks like you did not enter an integer!"

        finally:
            print "Finally, I executed!"
        print val
```

[35]:
```
askint()
```

```
Please enter an integer: 5
Finally, I executed!
5
```

[36]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-36-6ee53d339e7e> in <module>()
----> 1 askint()

<ipython-input-33-728ec4c542c2> in askint()
      7         finally:
      8             print "Finally, I executed!"
----> 9         print val

UnboundLocalError: local variable 'val' referenced before assignment
```

Notice how we got an error when trying to print val (because it was never properly assigned) Lets remedy this by asking the user and checking to make sure the input type is an integer:

[39]:
```
def askint():
        try:
            val = int(raw_input("Please enter an integer: "))
        except:
            print "Looks like you did not enter an integer!"
            val = int(raw_input("Try again-Please enter an integer: "))
        finally:
            print "Finally, I executed!"
        print val
```

[40]:
```
askint()
```

```
Please enter an integer: f
Looks like you did not enter an integer!
Try again-Please enter an integer: f
Finally, I executed!
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-40-6ee53d339e7e> in <module>()
----> 1 askint()
```

```
<ipython-input-39-e540976abf48> in askint()
      4         except:
      5             print "Looks like you did not enter an integer!"
----> 6             val = int(raw_input("Try again-Please enter an integer: "))
      7         finally:
      8             print "Finally, I executed!"

ValueError: invalid literal for int() with base 10: 'f'
```

Hmmm...that only did one check. How can we continually keep checking? We can use a while loop!

[41]:
```
def askint():
    while True:
        try:
            val = int(raw_input("Please enter an integer: "))
        except:
            print "Looks like you did not enter an integer!"
            continue
        else:
            print 'Yep thats an integer!'
            break
        finally:
            print "Finally, I executed!"
        print val
```

[42]:
```
askint()
```

```
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: five
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: four
Looks like you did not enter an integer!
Finally, I executed!
Please enter an integer: 4
Yep thats an integer!
Finally, I executed!
```

[ ]:
```
# Let's try to open a file that does not exist
file_name = 'not_existing.txt'

try:
    with open(file_name, 'r') as my_file:
        print('File is successfully open')
```

```
except FileNotFoundError as e:
    print('Uups, file: {} not found'.format(file_name))
    print('Exception: {} was raised'.format(e))
```

If you don't know the type of exceptions that a code block can possibly raise, you can use `Exception` which catches all exceptions. In addition, you can have multiple `except` statements.

[ ]:
```
def calculate_division(var1, var2):
    result = 0

    try:
        result = var1 / var2
    except ZeroDivisionError as ex1:
        print("Can't divide by zero")
    except Exception as ex2:
        print('Exception: {}'.format(ex2))

    return result

result1 = calculate_division(3, 3)
print('result1: {}'.format(result1))

result2 = calculate_division(3, '3')
print('result2: {}'.format(result2))

result3 = calculate_division(3, 0)
print('result3: {}'.format(result3))
```

`try-except` can be also in outer scope:

[ ]:
```
def calculate_division(var1, var2):
    return var1 / var2

try:
    result = calculate_division(3, '3')
except Exception as e:
    print(e)
```

1.1.1 Please note the following when using defensive programming as bellow

this code uses the else clause which will ONLY be reached and executed if the code in try block succeeds without errors

```
try:
   this code is the entry point and will be executed first
except Exception as e:
   manage the error here
   (this code in here will always be reached and run ONLY when when
```

```
        the code in the try block fails)
    else:
        This code will always run ONLY when the code in the try block successeds without errors
        that is because we are using try-except-else (take note of the else here)
```

For example below

[4]:
```
try:
    with open("my_file.txt", "r") as file:
        content = file.read()
        print(content)

except Exception as e:
    print(f"I Failed to run properly, something went wrong {e}")

else:
    print("I will ONLY run when the code in try succeeds")
```

```
I Failed to run properly, something went wrong [Errno 2] No such file or
directory: 'my_file.txt'
```

this code uses the try-except-finally clause which will ALWAYS be reached and executed whether the code in try block succeeds or fails

```
try:
    this code is the entry point and will be executed first
except Exception as e:
    manage the error here
    (this code in here will always be reached and run ONLY when when
     the code in the try block fails)
finally:
    This code will always be reached and run even if there were errors in the try block
    that is because we are using try-except-finally (take note of the finally here)
```

For example

[ ]:
```
try:
    with open("my_file.txt", "r") as file:
        content = file.read()
        print(content)

except Exception as e:
    print(f"I Failed to run properly, something went wrong {e")

finally:
    print("I will ALWAYS run even when the code in try fails")
```

[ ]:
```
when the code in the try block executes successfuly without errors
Then the code in the else
db_location = "/Users/username/Desktop/chinook.db"
```

```
get_all_employees = """SELECT firstname, lastname, title, email FROM employees;
 ↪"""

try:
    connection = sqlite3.connect(db_location)
    executioner = connection.cursor()
    all_employees = executioner.execute(get_all_employees)
    for employee in all_employees:
        first_name = employee[0]
        last_name = employee[1]
        title = employee[2]
        email = employee[3]
        print(f"Full name: {first_name} {last_name} \nEmail address: {email}␣
 ↪\nJob Title: {title} \n")
    connection.commit()

except sqlite3.Error as e:
    print(f"Something went wrong with db operation as below \n{e}")

else:
    executioner.close()
```

1.2 Creating your custom exceptions

In your own applications, you can use custom exceptions for signaling users about errors which occur during your application run time.

[ ]:
```
import math

# Define your own exception
class NegativeNumbersNotSupported(Exception):
    pass

# Dummy example how to use your custom exception
def secret_calculation(number1, number2):
    if number1 < 0 or number2 < 0:
        msg = 'Negative number in at least one of the parameters: {}, {}'.
 ↪format(
            number1, number2)
        raise NegativeNumbersNotSupported(msg)

    return math.sqrt(number1) + math.sqrt(number2)

# Uncomment to see the traceback
# result = secret_calculation(-1, 1)
```