**Type Casting & Exceptions**

martes, 28 de marzo de 2023

6:25 p. m.

Type Casting:

Type conversion: <https://www.programiz.com/python-programming/type-conversion-and-casting>

<https://www.geeksforgeeks.org/type-casting-in-python-implicit-and-explicit-with-examples/>

Exceptions - Error Handling:

Python Doc. (8) - Errors & Exceptions: <https://docs.python.org/3/tutorial/errors.html>

Exceptions and Errors Handling

Intro - <https://realpython.com/python-exceptions/>

Raise works to stop the execution if a certain condition is met.

x = 10

if x > 5:

    raise Exception('number higher than 5')

Terminal:

**Exception has occurred: Exception**

number higher than 5

File "C:\Users\USUARIO\GR\Software Development\Learning\Python\Exceptions & Errors Handling\Errors, try & except.py", line 5, in <module> raise Exception('number higher than 5') Exception: number higher than 5

Assert, Raise and Try

37 
raise - raise an exception. 
assert - raise an exception if a given condition is (or isn't) true. 
try 
Share 
- execute some code that might raise an exception, and if so, catch it. 
Edit Follow 
answered Oct21, 2016 at 18:15 
John Gordon 
27.8k .7 .32 .55 

Here are the key takeaways: 
• A try clause is executed up until the point where the first exception is encountered. 
• Inside the except clause, or the exception handler, you determine how the program 
responds to the exception. 
• You can anticipate multiple exceptions and differentiate how the program should 
respond to them. 
Avoid using bare except clauses. 

Assert

Assert that a condition is met: 
assert: 
Test if condition is True 

Try Except

try: 
except: 
Run this code 
Execute this code when 
there is an exception 

Else after Try Except

The else Clause 
In Python, using the else statement, you can instruct a program to execute a certain block Of 
code only in the absence of exceptions. 
try: 
except: 
else: 
Run this code 
Execute this code when 
there is an exception 
No exceptions? Run this 
code. 

And finally, 'Finally'

Cleaning Up After Using finally 
Imagine that you always had to implement some sort Of action to clean up after executing 
your code. Python enables you to do so using the finally clause. 
try: 
except: 
else: 
finally: 
Run this code 
Execute this code when 
there is an exception 
No exceptions? Run this 
code. 
Always run this code. 

Have a look at the following example: 
Python 
try: 
linux_interaction() 
except AssertionError as error: 
print(error) 
else: 
try: 
with open('file.log') as file: 
read_data = file. read() 
except FileNotFoundError as fnf_error: 
print (fnf _error) 
finally: 
print('C1eaning up, irrespective of any exceptions. 
In the previous code, everything in the finally clause will be executed. It does not matter if 
you encounter an exception somewhere in the try or else clauses. Running the previous 
code on a Windows machine would output the following: 
Shell 
Function can 
Cleaning up, 
only run on Linux systems. 
irrespective of any exceptions. 

Closing comments

Summing Up 
After seeing the difference between syntax errors and exceptions, you learned about various 
ways to raise, catch, and handle exceptions in Python. In this article, you saw the following 
options: 
• raise allows you to throw an exception at any time. 
• assert enables you to verify if a certain condition is met and throw an exception if it 
isn't. 
• In the Euclause, all statements are executed until an exception is encountered. 
• exceot is used to catch and handle the exception(s) that are encountered in the 
clause. 
• else lets you code sections that should run only when no exceptions are encountered 
in the Lyclause. 
• finall enables you to execute sections Of code that should always run, with or 
without any previously encountered exceptions. 

* Anecdotes from not paying attending to catching exceptions: <https://realpython.com/the-most-diabolical-python-antipattern/>

Corey's Tutorial

C. Schafer - [Python Tutorial: Using Try/Except Blocks for Error Handling](https://www.youtube.com/watch?v=NIWwJbo-9_8)

The use exceptions and error handling is to write code that can assertively know what went wrong, so let say we have this:

l = [1,2,3]

i = l[4]

When executed it'll throw this error message:

**Exception has occurred: IndexError**

list index out of range

File "C:\Users\USUARIO\GR\Software Development\Learning\Python\Exceptions & Errors Handling\Errors, try & except.py", line 13, in <module> i = l[4] ~^^^ IndexError: list index out of range

Now if we have the same but between a try/except block, it will be possible to custom made error messages or to let it pass all along:

try:

   l = [1,2,3]

   i = l[4]

except Exception:

    pass

This will let it pass everything within the try block, since it says 'Anything that throw an error, let it pass', this is particularly dangerous since when debugging it will be virtually impossible to know why a code is not working as intended, so the general recommendation is to be as specific as possible when naming the exceptions, which is possible as follows:

try:

   l = [1,2,3]

   i = l[4]

except IndexError as e:

    print(e)

Console: list index out of range

If handling multiple exceptions, is preferable to have the more specific errors at the top and the more general at the bottom, since if a except Exception is above a except IndexError everything will be caught at the first one.

Now, if is not known what kind of exception could be arose but we will have to catch it and read it as it is, the next tweak will solve it:

try:

   l = [1,2,3,4,5]

   i = l[4]     #Now, this was corrected

   var = bad\_var

except IndexError as e:

    print(e)

except Exception as e:

    print(e)

Console: list index out of range This is an exception caught by having not named the variable "var" correctly.

Another part of the try/except expression is the 'else' statement, which for exceptions works as '… if no exceptions are thrown, execute this …'

try:

   l = [1,2,3,4,5]

   i = l[4]

   var = 'bad\_var'     #Now, this was corrected

except IndexError as e:

    print(e)

except Exception as e:

    print(e)

else:

    print(var)

Console: bad\_var

The last part of the try/except expression is the 'finally' statement, which works as '… regardless there's exceptions or nor, execute this …'

try:

   l = [1,2,3,4,5]

   i = l[4]

   var = bad\_var    #intentionally it was made wrong to show finally

except IndexError as e:

    print(e)

except Exception as e:

else:

    print(var)

finally:

    print('This is the end of the code')

Console:

name 'bad\_var' is not defined

This is the end of the code

This finally makes sense when there is resources or things to be close at the end of the execution, like databases.

Assertions

The assert reserved word works similar to a try/except expression, but merged, meaning that assert executes a code and raise a message if a certain condition were met.

The exception thrown is an **AssertionError.**

If a program is being executed in optimized mode the assertions are excluded.

def age\_check(n):

    assert n > 0, 'Non positives ages are not accepted'

age\_check(-4)

Console: Exception has occurred: AssertionError Non positives ages are not accepted

File "C:\Users\USUARIO\GR\Software Development\Learning\Python\Exceptions & Errors Handling\Errors, try & except.py", line 92, in age\_check assert n > 0, 'Non positives ages are not accepted' File "C:\Users\USUARIO\GR\Software Development\Learning\Python\Exceptions & Errors Handling\Errors, try & except.py", line 95, in <module> age\_check(-4) AssertionError: Non positives ages are not accepted