try and except[¶](http://localhost:8888/notebooks/Downloads/Complete-Python-3-Bootcamp-master/Complete-Python-3-Bootcamp-master/07-Errors%20and%20Exception%20Handling/01-Errors%20and%20Exceptions%20Handling.ipynb#try-and-except)

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

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 let's check out an example: We will look at some code that opens and writes a file:

In [2]:

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

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

In [3]:

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 <code>except:</code> if we weren't sure what exception would occur. For example:

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:

In [4]:

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 <code>finally</code> comes in.

**## finally**

The <code>finally:</code> block of code will always be run regardless if there was an exception in the <code>try</code> 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:

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[¶](http://localhost:8888/notebooks/Downloads/Complete-Python-3-Bootcamp-master/Complete-Python-3-Bootcamp-master/07-Errors%20and%20Exception%20Handling/01-Errors%20and%20Exceptions%20Handling.ipynb" \l "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:

In [5]:

try:

f = open("testfile", "w")

f.write("Test write statement")

f.close()

finally:

print("Always execute finally code blocks")

Always execute finally code blocks

We can use this in conjunction with <code>except</code>. Let's see a new example that will take into account a user providing the wrong input:

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

In [6]:

def askint():

try:

val = int(input("Please enter an integer: "))

except:

print("Looks like you did not enter an integer!")

​

finally:

print("Finally, I executed!")

print(val)

. . .

In [7]:

askint()

Please enter an integer: 5

Finally, I executed!

5

. . .

In [8]:

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-8-cc291aa76c10> in <module>()

----> 1 askint()

<ipython-input-6-c97dd1c75d24> 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). Let's remedy this by asking the user and checking to make sure the input type is an integer:

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

In [9]:

def askint():

try:

val = int(input("Please enter an integer: "))

except:

print("Looks like you did not enter an integer!")

val = int(input("Try again-Please enter an integer: "))

finally:

print("Finally, I executed!")

print(val)

. . .

In [10]:

askint()

Please enter an integer: five

Looks like you did not enter an integer!

Try again-Please enter an integer: four

Finally, I executed!

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-9-92b5f751eb01> in askint()

2 try:

----> 3 val = int(input("Please enter an integer: "))

4 except:

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

During handling of the above exception, another exception occurred:

ValueError Traceback (most recent call last)

<ipython-input-10-cc291aa76c10> in <module>()

----> 1 askint()

<ipython-input-9-92b5f751eb01> in askint()

4 except:

5 print("Looks like you did not enter an integer!")

----> 6 val = int(input("Try again-Please enter an integer: "))

7 finally:

8 print("Finally, I executed!")

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

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

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

In [11]:

def askint():

while True:

try:

val = int(input("Please enter an integer: "))

except:

print("Looks like you did not enter an integer!")

continue

else:

print("Yep that's an integer!")

break

finally:

print("Finally, I executed!")

print(val)

. . .

In [12]:

askint()

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: 3

Yep that's an integer!

Finally, I executed!

So why did our function print "Finally, I executed!" after each trial, yet it never printed `val` itself? This is because with a try/except/finally clause, any <code>continue</code> or <code>break</code> statements are reserved until *\*after\** the try clause is completed. This means that even though a successful input of **\*\*3\*\*** brought us to the <code>else:</code> block, and a <code>break</code> statement was thrown, the try clause continued through to <code>finally:</code> before breaking out of the while loop. And since <code>print(val)</code> was outside the try clause, the <code>break</code> statement prevented it from running.

​

Let's make one final adjustment:

So why did our function print "Finally, I executed!" after each trial, yet it never printed val itself? This is because with a try/except/finally clause, any continue or break statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the else: block, and a break statement was thrown, the try clause continued through to finally: before breaking out of the while loop. And since print(val) was outside the try clause, the break statement prevented it from running.

Let's make one final adjustment:

In [13]:

def askint():

while True:

try:

val = int(input("Please enter an integer: "))

except:

print("Looks like you did not enter an integer!")

continue

else:

print("Yep that's an integer!")

print(val)

break

finally:

print("Finally, I executed!")

. . .

In [14]:

askint()

Please enter an integer: six

Looks like you did not enter an integer!

Finally, I executed!

Please enter an integer: 6

Yep that's an integer!

6

Finally, I executed!