**Errors and Exception Handling**

In this lecture we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

In [1]:



print('Hello)

**File "<ipython-input-1-db8c9988558c>", 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](https://docs.python.org/3/library/exceptions.html). Now let's learn how to handle errors and exceptions in our own code.

In [ ]:



x **=** num1**/**num2

num2 **=**0

**What Is an Exception?**

In Python, everything is a type of object, including integers, strings, booleans and indeed Exceptions and Errors. In Python the Exception/Error types are defined in a class hierarchy with the root of this hierarchy being the BaseException type. All built-in errors and exceptions eventually extend from the BaseException type. It has a subclass Exception which is the root of all user defined exceptions (as well as many built-in exceptions). In turn ArithmeticException is the base class for all built-in exceptions associated with arithmetic errors.



The following table illustrates terminology typically used with exception/error

handling in Python

​

Exception                   An error which is generated at runtime

Raising an exception         Generating a new exception

Throwing an exception       Triggering a generated exception

Handling an exception         Processing code that deals with the error

Handler                     The code that deals with the error (referred to as the catch block)

Signal                       A particular type of exception (such as out of bounds or divide by zero)

Different types of error produce different types of exception. For example, if the error is caused by dividing an integer by zero, then the exception is a arithmetic exception. The type of exception is identified by objects and can be caught and processed by exception handlers. Each handler can deal with exceptions associated with its class of error or exception (and its subclasses). An exception is instantiated when it is raised. The system searches back up the execution stack (the set of functions or methods that have been invoked in reverse order) until it finds a handler which can deal with the exception. The associated handler then processes the exception. This may involve performing some remedial action or terminating the current execution in a controlled manner. In some cases, it may be possible to restart executing the code. As a handler can only deal with an exception of a specified class (or subclass), an exception may pass through a number of handler blocks before it finds one that can process it.

**Handling an Exception**

You can catch an exception by implementing the try—except construct. This construct is broken into three parts:

• try block. The try block indicates the code which is to be monitored for the exceptions listed in the except expressions.

• except clause. You can use an optional except clause to indicate what to do when certain classes of exception/error occur (e.g. resolve the problem or generate a warning message). There can be any number of except clauses in sequence checking for different types of error/exceptions.

• else clause. This is an optional clause which will be run if and only if no exception was thrown in the try block. It is useful for code that must be executed if the try clause does not raise an exception.

• finally clause. The optional finally clause runs after the try block exits (whether or not this is due to an exception being raised). You can use it to clean up any resources, close files, etc.

**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** IndexOutbounds:

*# This will only check for an IOError exception and then execute this print statement*

print("Error: Could not find file or read data")

**except** IOError:

**pass**

**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'):

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 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 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:

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 except. Let's see a new example that will take into account a user providing the wrong input:

In [1]:



**def** askint():

**try**:

val **=** int(input("Please enter an integer: ")) *#it is integer variable , exception has been raised*

*# print(type(val))*

**except**:

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

​

**finally**:

print("Finally, I executed!")

*# print(val) #the exception has occured and come here*

In [2]:



askint()

Please enter an integer: 5

Finally, I executed!

In [3]:



askint()

Please enter an integer: five

Looks like you did not enter an integer!

Finally, I executed!

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 [4]:



**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 [5]:



askint()

Please enter an integer: five

Looks like you did not enter an integer!

Try again-Please enter an integer: 5

Finally, I executed!

5

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 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!

In [5]:



**import** sys

**try**: *#we write anything in suspition that it might fail*

num1**=**int(input("Please enter first number : "))

num2**=**int(input("Please enter second number : "))

ans**=**num1**/**num2

**except** ZeroDivisionError: *#if anything fails in try, control comes here*

print("sorry you cannot divide by zero")

print("exception name is : ", sys.exc\_info()[0])

print("exception description is : ", sys.exc\_info()[1])

**except** ValueError:

print("sorry you cannot divide strings")

print("exception name is : ", sys.exc\_info()[0])

print("exception description is : ", sys.exc\_info()[1])

**except**:

print("something went wrong")

print("exception name is : ", sys.exc\_info()[0])

print("exception description is : ", sys.exc\_info()[1])

**else**: *#control comes when everything passes in try*

print("answer of division is ", ans)

**finally**: *#surely execute irrespective of try passing or failing*

print("bye")

*#else and finally are optional blocks but with try except is mandatory*

*#default except has to be in the last*

Please enter first number : 10

Please enter second number : st

sorry you cannot divide strings

exception name is : <class 'ValueError'>

exception description is : invalid literal for int() with base 10: 'st'

bye

In [ ]:



​

In [ ]:



​



Raising an Exception

​

An error or exception is raised using the keyword raise. The syntax of this is

For example:

  def function\_bang():

      print('function\_bang in')

      raise ValueError('Bang!')

      print('function\_bang')

In the above function the second statement in the function body will create a new

instance of the ValueError class and then raise it so that it is thrown allowing it

to be caught by any exception handlers that have been defined.

We can handle this exception by writing a try block with an except clause

for the ValueError class. For example:

  try:

      function\_bang()

      except ValueError as ve:

      print(ve)

This generates the output

​

function\_bang in

Bang!



Note that if you just want to raise an exception without providing any constructor arguments, then you can just provide the name of the exception class to the

raise keyword:

  raise ValueError # short hand for raise ValueError()

​

You can also re-raise an error or an exception; this can be useful if you merely

want to note that an error has occurred and then re throw it so that it can be handled

further up in your application:

  try:

      function\_bang()

      except ValueError:

          print('oops')

          raise

This will re raise the ValueError caught by the except clause. Note here we

did not even bind it to a variable; however, we could have done this if required.

​

try:

  function\_bang()

  except ValueError as ve:

  print(ve)

  raise

In [7]:



'''

raise - raises an exception blindly without checking the condition

assert - raises an exception conditionally, it checks the condition and then raises an exception

'''

​

**try**:

num1**=**int(input("Please enter any number "))

**if** num1**%**2**==**0:

**raise** ValueError

**except** ValueError:

print("only odd numbers are allowed for even numbers exception is raised")

**else**:

print("success")

Please enter any number 2

only odd numbers are allowed for even numbers exception is raised

In [3]:



*#assert raises an AssertionError if condition fails, if condition passes exception is not raised*

**import** sys

**try**:

num1**=**int(input("Please enter any number "))

**assert** num1**%**2**!=**0, "only odd numbers are allowed for even numbers exception is raised"

**assert** num1**>**20, "only odd numbers are allowed which are greater than 20"

**except** AssertionError:

print(sys.exc\_info()[0],sys.exc\_info()[1]) *#first value is for name of the exception , second one is for description of exception*

​

**else**:

print("success")

Please enter any number 19

<class 'AssertionError'> only odd numbers are allowed which are greater than 20

In [ ]:



​



Defining an Custom Exception

You can define your own Errors and Exceptions, which can give you more control

over what happens in particular circumstances. To define an exception, you create a

subclass of the Exception class or one of its subclasses.

For example, to define a InvalidAgeException, we can extend the

Exception class and generate an appropriate message:

  class InvalidAgeException(Exception):

""" Valid Ages must be between 0 and 120 """

This class can be used to explicitly represent an issue when an age is set on a

Person which is not within the acceptable age range.

raise ValueError # short hand for raise ValueError()

In [5]:



**class** InvalidAgeException(Exception):

""" Valid Ages must be between 0 and 120 """

**def** \_\_init\_\_(self, value):

self.value **=** value

**def** \_\_str\_\_(self):

**return** 'InvalidAgeException(' **+** str(self.value) **+** ')'

**class** Person:

**def** \_\_init\_\_(self, name, age):

self.\_name **=** name

self.\_age **=** age

@property

**def** age(self):

""" The docstring for the age property """

print('In age method')

**return** self.\_age

@age.setter

**def** age(self, value):

print('In set\_age method(', value, ')')

**if** isinstance(value, int) **&** (value **>** 0 **&** value **<** 120):

self.\_age **=** value

**else**:

**raise** InvalidAgeException(value)

@property

**def** name(self):

print('In name')

**return** self.\_name

@name.deleter

**def** name(self):

**del** self.\_name

**def** \_\_str\_\_(self):

**return** 'Person[' **+** str(self.\_name) **+** '] is ' **+** self.\_age

​

In [7]:



**try**:

p **=** Person('Adam', 21)

p.age **=** **-**1

**except** InvalidAgeException **as** e: *#print as object Invalid exception class is being called*

print(e)

In set\_age method( -1 )

InvalidAgeException(-1)

In [ ]:



'''

write a function to return a dictionary. in that function create dictionary

where key is emp id and value is name.

​

once dictionary is created ask user any emp id and fetch the name of employee

if emp id does not exist, exception should be trapped and valid message should be printed.

do not use if else etc loops

'''

​

**def** return\_dict():

d **=** {}

empid**=**0

**while** empid **!=** **-**999:

empid **=** int(input('Enter an emp id: '))

**if** empid **!=-**999:

name **=** input('Enter a name: ')

d[empid]**=**name

**return** d

​

**def** check\_empid(emp\_dict):

**try**:

empid **=** int(input('Enter an emp id to search: '))

print("employee name is ", emp\_dict[empid])

**except**:

print("sorry employee does not exist")

​

emp\_dict **=** return\_dict()

print('Final dictonary is: ', emp\_dict)

check\_empid(emp\_dict)

In [ ]:



*# you need to authenticate credit card number*

*# card number should be of 16 digits exactly*

*# card number should not contain capital letters*

*# card number should not contain lower case letters*

*# card number should not contain special chars*

*# card number should not contain spaces*

*# card number should contain only digits*

*# card number should not start with 0*

*# if all these conditions are true, card number is a valid card number*

​

**try**:

ccno**=**input("Please enter the credit card number ")

**assert** len(ccno)**==**4, "card number should be exactly of 4 digits"

**assert** ccno[0]**!=**'0', "card number should not start with 0"

upper,lower,digit,space,special **=** 0,0,0,0,0

**for** x **in** ccno:

**if** x.isupper():

upper**+=**1

**elif** x.islower():

lower**+=**1

**elif** x.isdigit():

digit**+=**1

**elif** x.isspace():

space**+=**1

**else**:

special**+=**1

**assert** upper**==**0, "card number should not contain upper case letters"

**assert** lower**==**0, "card number should not contain lower case letters"

**assert** special**==**0, "card number should not contain special charcaters"

**assert** space**==**0, "card number should not contain spaces"

**except** AssertionError:

print(sys.exc\_info()[1])

**else**:

print("Success :- you entered valid card number")

In [ ]:



question **for** homework

​

authencate the password creation process :**-**

password should contain atleast 8 chars

password should contain atleast one upper case letter

password should contain atleast one lower case letter

password should contain atleast one digit

password should contain atleast one special char

password should contain atleast one space

**try** using the way we did **for** credit card authentication

**try** using any function to do the same

**try** using **lambda** function **for** each condition

In [ ]:



**import** sys

**from** string **import** ascii\_letters,digits

string **=** "Aassha 2345@"

​

**def** passw\_checking(string):

length,upperlt,lowerlt,diglt,spacelt,txt **=** 0,0,0,0,0,0

**if** len(string) **<** 8:

length **=** 0

**else**:

length **=** len(string)

textspecial **=** set(string).difference(ascii\_letters**+**digits)

print("special characters are",textspecial)

**if** len(list(textspecial))**>**0:

txt**+=**1

**for** st **in** string:

**if** (st.isupper()):

upperlt**+=**1

**if** (st.islower()):

lowerlt**+=**1

**if** (st.isdigit()):

diglt**+=**1

**if** (st.isspace()):

spacelt**+=**1

**return** length,upperlt,lowerlt,diglt,spacelt,txt

​

​

length,uper,lowr,digt,spc,septxt **=** passw\_checking(string)

print("length = {} , Uppercase = {}, lowercase = {},digit = {},spaces = {},specialtxt = {}".format(length,uper,lowr,digt,spc,septxt))

**try**:

​

lt **=** **lambda** length : **True** **if**(length **>** 8) **else** **False**

**assert** lt(length), "Length of the password should be greater than 8"

​

up **=** **lambda** uper : **True** **if**(uper**>**0)**else** **False**

**assert** up(uper),"There should be atleast one upper case letter"

​

​

lw **=** **lambda** lowr : **True** **if**(lowr**>**0)**else** **False**

**assert** lw(lowr),"There should be atleast one Lower case letter"

​

dt **=** **lambda** digt : **True** **if**(digt**>**0)**else** **False**

**assert** dt(digt),"There should be atleast one digit "

​

sept **=** **lambda** septxt : **True** **if**(septxt**>**0)**else** **False**

**assert** sept(septxt),"There should be atleast one special case letter"

​

sp **=** **lambda** spc : **True** **if**(spc**>**0)**else** **False**

**assert** sp(spc),"There should be atleast one space "

**except**:

print(sys.exc\_info()[1])

**else**:

print("The password is valid")

​

​