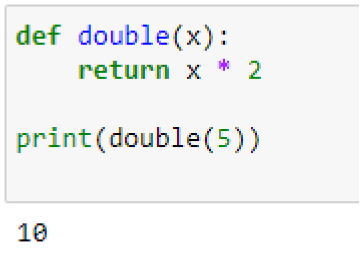
13.1 Lambda Functions

- ➢ Lambda or Anonymous functions are the functions defined without a name.
- ➢ The normal functions are defined with the ‘def’ keyword whereas the lambda functions are defined with the ‘lambda’ keyword.
- ➢ Lambda functions are used extensively along with the built-in functions.

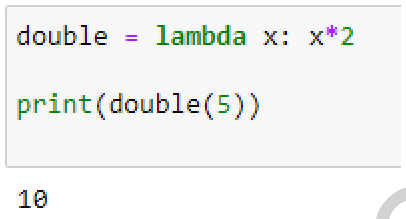
The below example was discussed at the timestamp 1:00.



```
def double(x):
    return x * 2

print(double(5))
```

```
10
```

In this example, we are defining the function double() which multiplies the given input number with ‘2’.
If we want to define the same function using the ‘lambda’ keyword, then it would be written as



```
double = lambda x: x*2

print(double(5))
```

```
10
```

In this way, we are defining the whole function in a single line.
Here **‘double’** is the function name, the ‘x’ present to the left side of colon(:) is an argument and **x*2** is the return value.
For example, if we want to compute the doubled value for number 10, then the syntax for it would be **double(10)**.

Below are the examples that were discussed starting from the timestamp 2:52 illustrating the usage of lambda functions along with the map(), filter() and reduce() functions.

Example 1

```
#Example use with filter()
lst = [1, 2, 3, 4, 5]
even_lst = list(filter(lambda x: (x%2 == 0), lst))
print(even_lst)
```

```
[2, 4]
```

In the above example, we are passing a list as an input and want only the even numbers out of it. So we are defining a lambda function **x:x%2==0** which means for the element ‘x’ it returns **True**, only if it is divisible by 2. If x%2==0, then it will be taken into the new list ‘even_lst’. Otherwise the element ‘x’ will be ignored.

Example 2

```
#Example use with map()
lst = [1, 2, 3, 4, 5]
new_lst = list(map(lambda x: x ** 2, lst))
print(new_lst)
```

```
[1, 4, 9, 16, 25]
```

In this example, we are passing a list as an input and want to apply square operation on all the elements in the list and get the result. So here we are defining a lambda function **x:x**2** which means for the element ‘x’, the square operation is performed. The map() function applied this lambda function to every element in the list ‘lst’ and returns the result in the form of a ‘map’ object. If we want the result in a list format, then we have to apply list() function on the ‘map’ object.

Example 3

```
#Example use with reduce()
from functools import reduce

lst = [1, 2, 3, 4, 5]
product_lst = reduce(lambda x, y: x*y, lst)
print(product_lst)
```

```
120
```

In this example, we are defining a lambda function that performs the product of two numbers. So here we are passing a list ‘lst’ and the reduce() picks the first 2 elements in the list and computes the product and then computes the product of the obtained result with the third element. This result is multiplied with the 4th element and this continues till the end.

13.2 File Handling in Python

File I/O

- ➢ File is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk).
- ➢ Since, random access memory (RAM) is volatile which loses its data when the computer is turned off, we use files for future use of the data.
- ➢ When we want to read from or write to a file we need to open it first. When we are done, it needs to be closed, so that resources that are tied with the file are freed.

File Operations

1) Opening a file
2) Read or Write
3) Closing a file

Opening a file

Python has a built-in function open() to open a file. This function returns a file object, also called a handle, as it is used to read or modify the file accordingly.

Syntax

f = open(file_name, mode)

Here 'file_name' is the name of the file that has to be opened. 'Mode' indicates whether to read from the file or write the data to the file or append the data to the existing data in the file.
'F' is a file object used to handle the operations on the file. It is also called **'Handle'**.

Example

f = open('example.txt','r')

Here it first checks if the file 'example.txt' is present in the current directory. If yes, then this file will be opened in read mode. If the file is not present, then it throws an error.

We also have to specify the mode while opening a file. In mode, we specify whether we want to read 'r', write 'w' or append 'a' to the file. We also can specify if we want to open the file in text mode or binary mode.

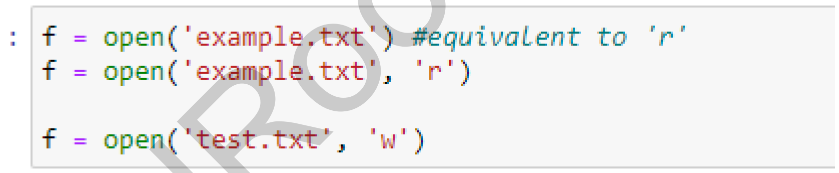
If we do not specify the mode, then the default mode will be the reading mode.

File Modes

‘r’	→	Opens a file for reading. (default)
'w'	→	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
'x'	→	Opens a file for exclusive creation. If the file already exists, the operation fails.
'a'	→	Open for appending at the end of the file without truncating it. Creates a new file if it does not exist.
't'	→	Open in text mode. (default)
'b'	→	Open in binary mode.
'+'	→	Open a file for updating (reading and writing)

➢ The default encoding is platform dependent. In windows, it is **'cp1252'** but **'utf-8'** in Linux.
➢ So, we must not also rely on the default encoding or else our code will behave differently in different platforms.
➢ Hence, when working with files in text mode, it is highly recommended to specify the encoding type.

Below is an example that shows how to open a file



```
f = open('example.txt') #equivalent to 'r'
f = open('example.txt', 'r')

f = open('test.txt', 'w')
```

In the first statement, we are opening the file directly without specifying the mode. But still the default mode is read(‘r’). So the file opens in read mode.

In the second statement, the file opens in read mode, as we have specified the mode as ‘r’.

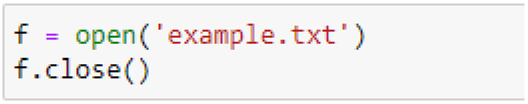
In both these statements, it first checks if the specified file is present in the current directory or not. If the file is present, then it will go ahead with opening the file. If the file is not present, then it will throw an error (FileNotFoundError).

In the third statement, as we want to open the file in write mode, it first checks if the file is present in the current directory. If the file is present, then all the contents in the file will be deleted and this file will be opened for writing purpose. If the file is not present, then it creates a file and then opens it for writing purpose.

Closing a file

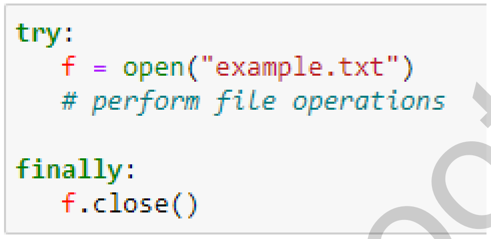
- ➢ Closing a file will free up the resources that were tied with the file and is done using the close() method.
- ➢ Python has a garbage collector to clean up unreferenced objects but, we must not rely on it to close the file.

Below is an example discussed at the timestamp 5:45



```
f = open('example.txt')
f.close()
```

This method of closing a file is not at all safe because if any exception occurs while performing any operations, then the code exits without closing the file.
Hence the best and the safe way is given below



```
try:
   f = open("example.txt")
   # perform file operations

finally:
   f.close()
```

In this example, we are opening the file and performing file operations. This block of code is enclosed in the ‘try’ block. In case, if any exception occurs, then the control comes out of it and executes the ‘finally’ block in which we are closing the file.
This way, we are guaranteed that the file is properly closed even if an exception is raised, causing the program flow to stop.

The best way to do this is using the **with** statement. This ensures that the file is closed when the **‘with’** block is exited. We don't need to explicitly call the close() method. It is done internally.

The syntax to open a file ‘example.txt’ without using the ‘with’ keyword is
f = open(‘example.txt’)

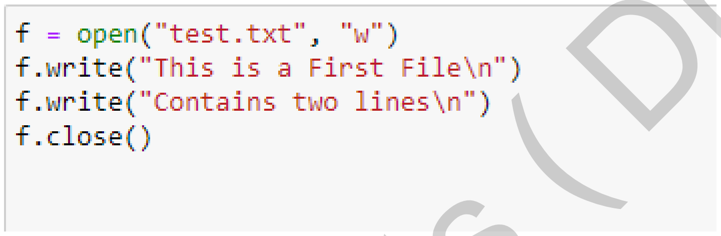
The syntax to open the same file using the ‘with’ statement is

with open("example.txt",encoding = 'utf-8') as f:

Writing to a file

- ➢ In order to write the data into a file, we have to open a file in write (‘w’), append (‘a’) or exclusive creation (‘x’) mode.
- ➢ Whenever we want to write the data into a file, if there is any data already present in it, then all the data will be erased.
- ➢ Writing a string or sequence of bytes (for binary files) is done using **write()** method. This method returns the number of characters written to the file.

Below is an example of writing data to a file and it has been discussed at the timestamp 8:30



```
f = open("test.txt", "w")
f.write("This is a First File\n")
f.write("Contains two lines\n")
f.close()
```

In this example, in the first statement, we are opening the file ‘test.txt’ in write mode. If this file is already present, then it would be opened in write mode. If it is not present in the current directory, then a new file is created with the name ‘test.txt’ and is opened in the write mode.

In the 2nd and the 3rd statements, we are writing the data to the file. In case, if any data is already existing in the file, then all that data will be erased and this new data gets added into the file.

In the 4th statement, we are closing the file.

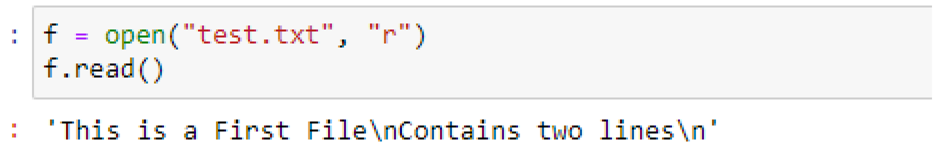
Reading from a file

There are various methods available to read the data from the files.

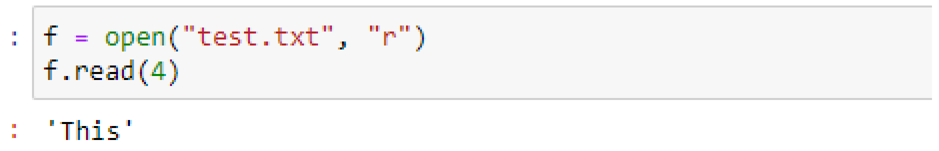
read(size)	→	reads the specified ‘size’ number of characters.
readline()	→	reads only one line of the data
read()	→	reads the entire file

In read() method if a value is not specified for the ‘size’ parameter, then it reads the entire file.

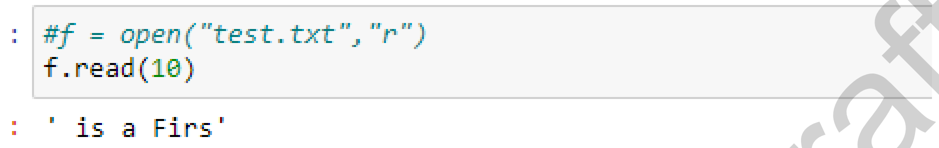
The below examples were discussed starting from the timestamp 9:35



```
: f = open("test.txt", "r")
  f.read()

: 'This is a First File\nContains two lines\n'
```



```
: f = open("test.txt", "r")
  f.read(4)

: 'This'
```



```
: #f = open("test.txt","r")
  f.read(10)

: ' is a Firs'
```

In the first example, we are opening the file and are reading the entire data from the file.

In the second example, we are opening the file and as we have specified the size as 4 in **read()** method, only the first 4 characters will be read from the file.

In the third cell, as we are not opening the file again and also we haven’t closed the file in the previous example, the first 4 characters were already read. So the cursor is now at position 4. When we pass the size as 10 in the **read()** method, from the 5th position, till the 10th position all the characters will be read.

We can change the file cursor position using the **seek()** method. We can find out the current position of our file cursor using **tell()** method. The below examples were discussed in the video starting from the timestamp 12:00.

```
f.tell()
```

```
14
```

```
f.seek(0) #bring the file cursor to initial position
```

```
0
```

```
print(f.read()) #read the entire file
```

```
This is a First File
Contains two lines
```

Here after executing the previously discussed examples, the file cursor has moved to the 14th position. So when we run the **f.tell()** method, it is showing the current cursor position as 14.

As we wanted the cursor to move back to the 0th position, we are calling the **f.seek(0)** method. The value in the parentheses of **seek()** method indicates the position to which the cursor has to be moved.

Again from here we are running **f.read()** to read the entire file.

Apart from reading the files character wise, we also can read them line by line. Using a loop, we can read a file line by line and this is the fastest and efficient approach. Below are the examples illustrating the file reading line by line.

```
:  f.seek(0)
   for line in f:
       print(line)
```

```
This is a First File

Contains two lines
```

In this example, we are running a loop, starting from the position 0 till the end. Here the contents are read until it encounters the next line character (ie., \n). This way all the lines are read.

```
f = open("test.txt", "r")
f.readline()
```

```
'This is a First File\n'
```

```
f.readline()
```

```
'Contains two lines\n'
```

```
f.readline()
```

```
''
```

In this above example, we are reading line by line using the readline() method. Here while reading the characters from the cursor position, if it encounters a newline character(ie., \n), then it the data till there would be read and is returned. Again when we execute the readline() method, then from the position of the new line character that was encountered, till the next new line character, the data would be read and is returned. This way, using the readline() method, we can read all the contents present in the file.

```
f.seek(0)
f.readlines()
```

```
['This is a First File\n', 'Contains two lines\n']
```

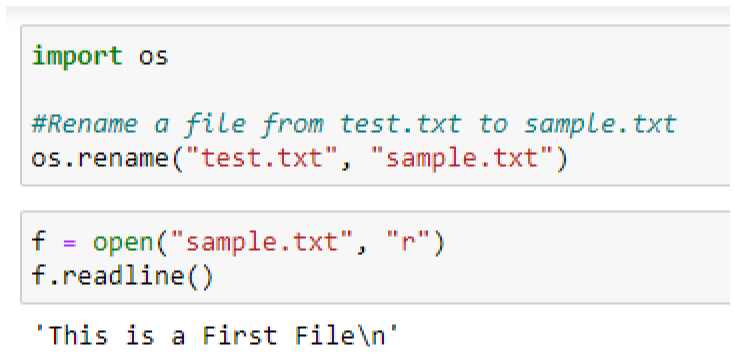
In this above code, all the lines are read at once starting from the cursor position(here it is 0) till the end of the file and are returned as a list of strings.

All the above discussed methods will stop reading/fetching the data and return empty values once if the end of the file is reached.

Renaming and Deleting Files in Python

Along with read/write operations, we also can perform rename/delete operations on the files. But in order to perform these operations, we have to import the ‘os’ module.

Below examples were discussed starting from the timestamp 14:50

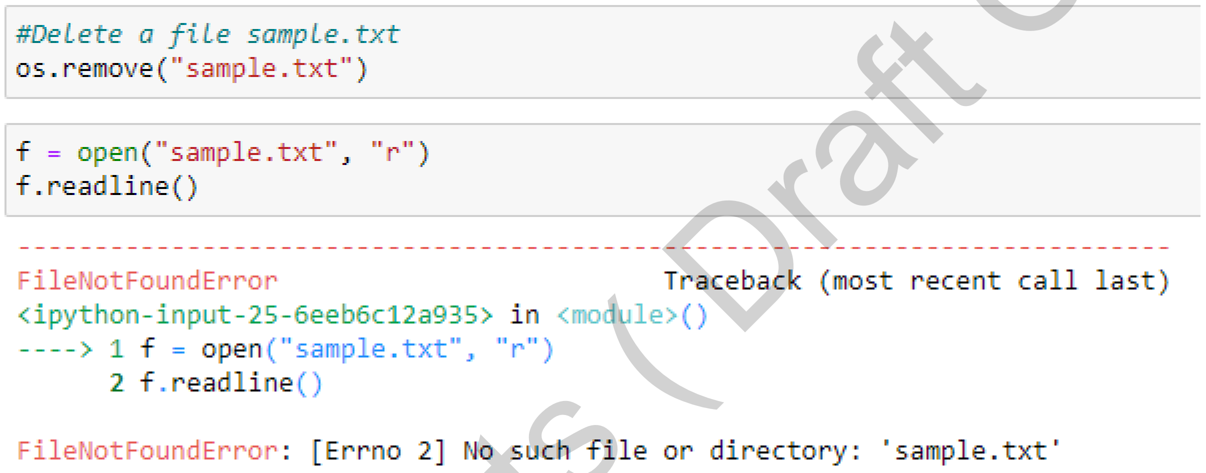


```
import os

#Rename a file from test.txt to sample.txt
os.rename("test.txt", "sample.txt")
```

```
f = open("sample.txt", "r")
f.readline()
```

```
'This is a First File\n'
```

In the above example, we are renaming the ‘test.txt’ file to ‘sample.txt’ and then we are reading the contents from the ‘sample.txt’ file and it all went successfully.



```
#Delete a file sample.txt
os.remove("sample.txt")
```

```
f = open("sample.txt", "r")
f.readline()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-25-6eeb6c12a935> in <module>()
----> 1 f = open("sample.txt", "r")
      2 f.readline()

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

In this example, we are deleting the ‘sample.txt’ file and are trying to open and read the file contents. But we are unsuccessful in opening and reading the file, as it was already deleted.

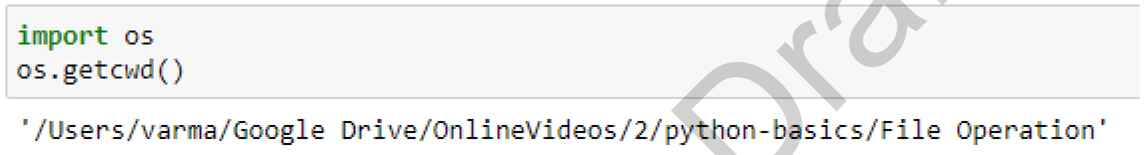
Python Directory and File Management

- ➢ If there are a large number of files to handle in your Python program, you can arrange your code within different directories to make things more manageable.
- ➢ A directory or folder is a collection of files and subdirectories. Python has the os module, which provides us with many useful methods to work with directories (and files as well).

Getting the Current Directory

We can get the name of the current directory using the **getcwd()** method of the ‘os’ module. This method returns the directory name in the form of a string.

The below example was discussed at the timestamp 16:10



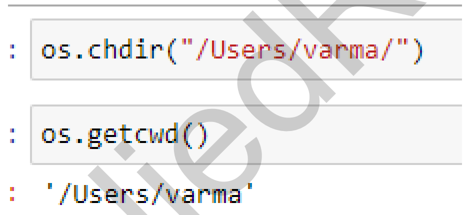
```
import os
os.getcwd()
```

```
'/Users/varma/Google Drive/OnlineVideos/2/python-basics/File Operation'
```

Changing the Directory

We can change the current working directory using the **chdir()** method. The new path that we want to change to must be supplied as a string to this method. We can use both forward slash (/) or the backward slash (\) to separate path elements.

The below example was discussed at the timestamp 16:40



```
: os.chdir("/Users/varma/")
```

```
: os.getcwd()
```

```
: '/Users/varma'
```

List Directories and Files

We can get a list of all the directories and files using the **listdir()** method. We have to specify the path of the directory as an argument for **listdir()** method and it returns a list of all directories and files in that directory.

If we do not specify any path, then by default it gives a list of all directories and files in the current directory.
Below is the syntax to find out the list of all directories and files in the current directory.

```
os.listdir(os.getcwd())
```

Making New Directory

- ➢ We can make a new directory using the **mkdir()** method.
- ➢ This method takes in the path of the new directory. If the full path is not specified, the new directory is created in the current working directory.
- ➢ If we want to remove an empty directory, then we have to use the **rmdir()** method. This method works only on empty directories.
- ➢ If we want to remove non-empty directories, then we have to use **rmtree()** method in the '**shutil**' module.

  The below example was discussed at the timestamp 19:20 illustrating all the above discussed operations.

```python
import shutil

os.mkdir('test')
os.chdir('./test')
f = open("testfile.txt",'w')
f.write("Hello World")
os.chdir("../")
os.rmdir('test')
```

```
---------------------------------------------------------------------------
OSError                                   Traceback (most recent call last)
<ipython-input-39-a990945f349d> in <module>()
      6 f.write("Hello World")
      7 os.chdir("../")
----> 8 os.rmdir('test')
      9 
     10 

OSError: [Errno 66] Directory not empty: 'test'
```

```python
# remove an non-empty directory
shutil.rmtree('test')
```

```python
os.getcwd()
```

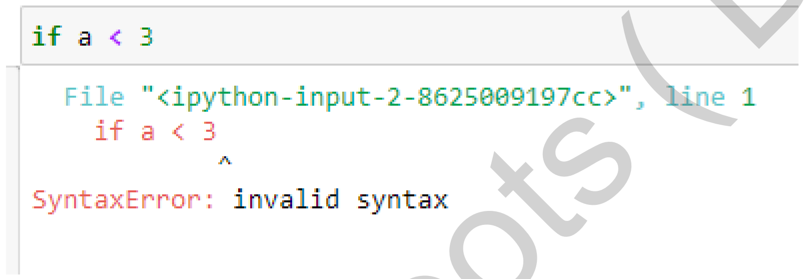
```
'/Users/varma'
```

13.3 Exception Handling

Python Errors and Built-in Exceptions

- ➢ We sometimes do encounter errors while writing the programs. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
- ➢ Errors can also occur at runtime and these are called **exceptions**.
- ➢ Few examples are when a file we try to open does not exist (FileNotFoundError), dividing a number by zero (ZeroDivisionError), module we try to import is not found (ImportError) etc.
- ➢ Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

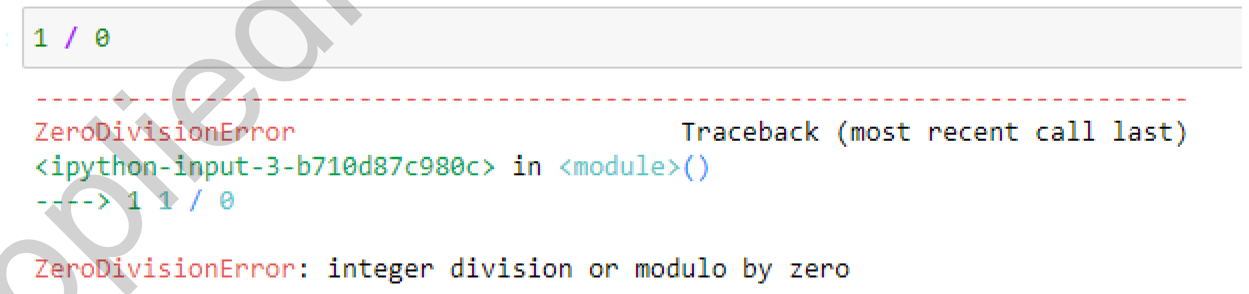
Below are a few examples of errors that were discussed starting from the timestamp 0:15



```
if a < 3
```

```
  File "<ipython-input-2-8625009197cc>", line 1
    if a < 3
            ^
SyntaxError: invalid syntax
```

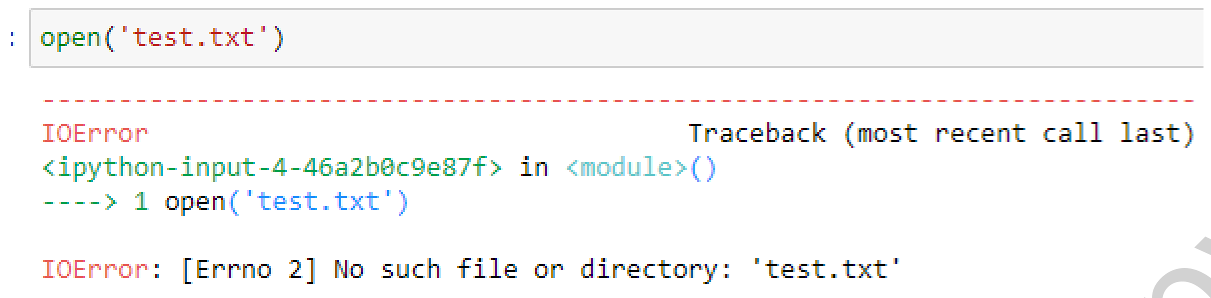
Here we haven't used the colon symbol after the condition, hence it has raised a syntax error.



```
1 / 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-3-b710d87c980c> in <module>()
----> 1 1 / 0

ZeroDivisionError: integer division or modulo by zero
```

Here we are trying to divide a number and hence it has raised the ZeroDivisionError.



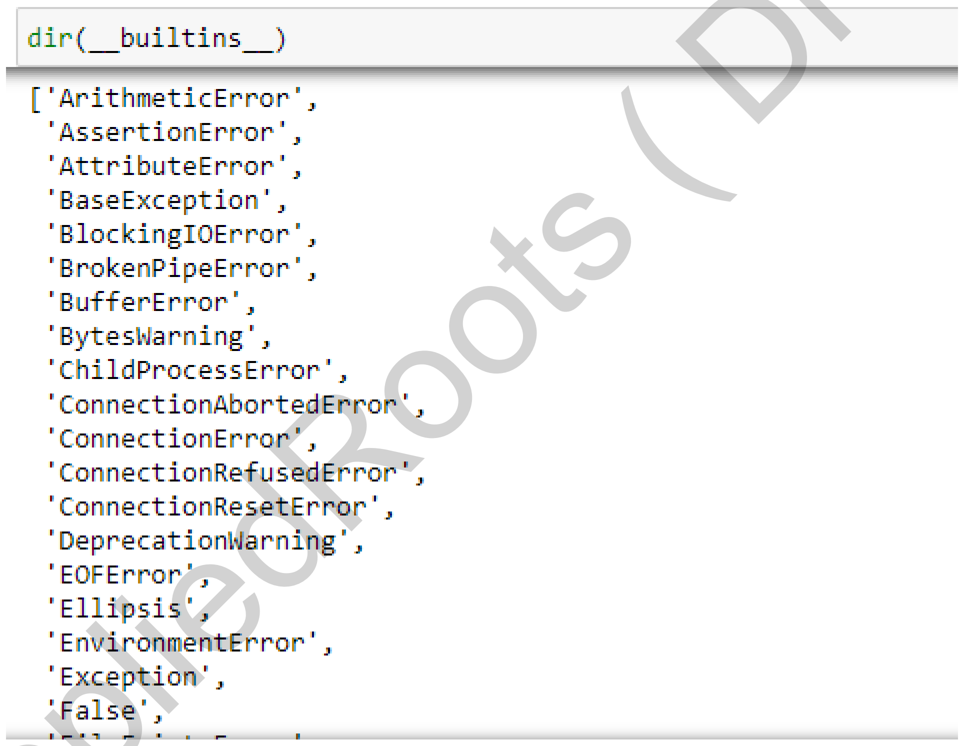
```
: open('test.txt')

---------------------------------------------------------------------------
IOError                                   Traceback (most recent call last)
<ipython-input-4-46a2b0c9e87f> in <module>()
----> 1 open('test.txt')

IOError: [Errno 2] No such file or directory: 'test.txt'
```

As there is no file called 'test.txt', it has raised the IOError.

Python Built-in Exceptions

The dir() gives a list of all the built-in exceptions that are handled by Python.

The syntax of the dir() has been explained at the timestamp 2:00



```
dir(__builtins__)

['ArithmeticError',
 'AssertionError',
 'AttributeError',
 'BaseException',
 'BlockingIOError',
 'BrokenPipeError',
 'BufferError',
 'BytesWarning',
 'ChildProcessError',
 'ConnectionAbortedError',
 'ConnectionError',
 'ConnectionRefusedError',
 'ConnectionResetError',
 'DeprecationWarning',
 'EOFError',
 'Ellipsis',
 'EnvironmentError',
 'Exception',
 'False',
```

Python Exception Handling - Try, Except and Finally blocks

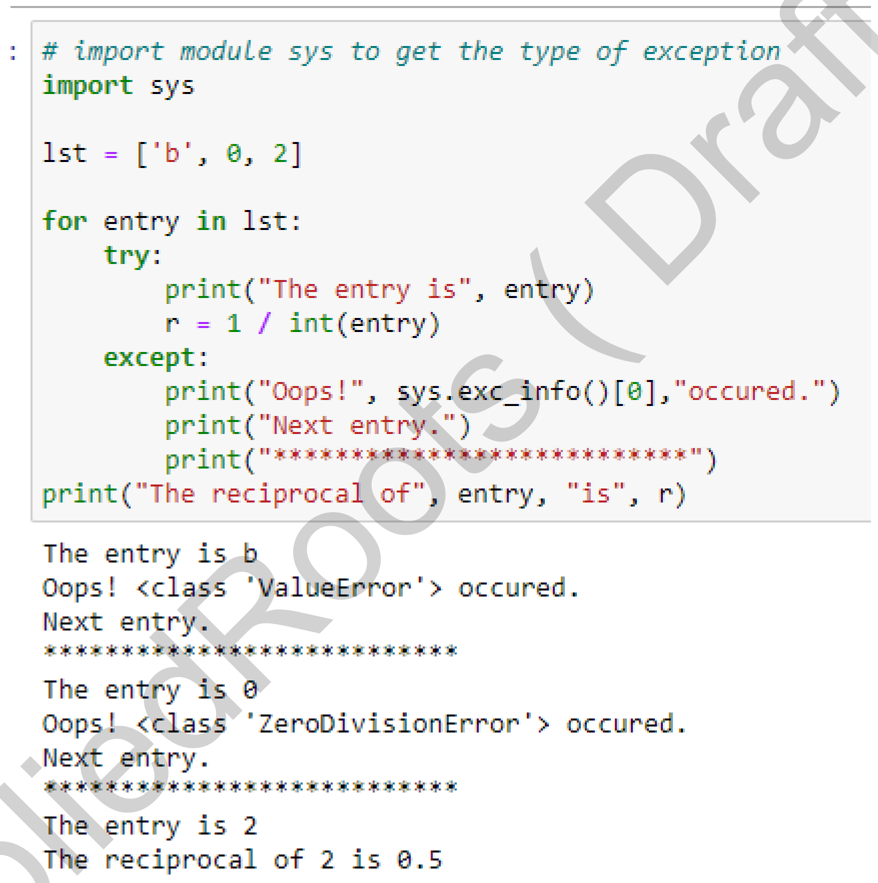
- ➢ Python has many built-in exceptions which forces the program to output an error when something in it goes wrong.
- ➢ When these exceptions occur, it causes the current process to stop and passes it to the calling process until it is handled. If not handled, our program will crash.

- ➢ For example, if function A calls function B which in turn calls function C and an exception occurs in function C. If it is not handled in C, the exception passes to B and then to A.
- ➢ If never handled, an error message is spit out and the program comes to a sudden, unexpected halt.

Catching Exceptions in Python

- ➢ In Python, the exceptions are handled using the ‘try’ statement.
- ➢ A critical operation which can raise an exception is placed inside the try clause and the code that handles exception is written in the except clause.

Below is an example discussed at the timestamp 2:35 in the video.



```python
# import module sys to get the type of exception
import sys

lst = ['b', 0, 2]

for entry in lst:
    try:
        print("The entry is", entry)
        r = 1 / int(entry)
    except:
        print("Oops!", sys.exc_info()[0],"occured.")
        print("Next entry.")
        print("***************************")
print("The reciprocal of", entry, "is", r)
```

```
The entry is b
Oops! <class 'ValueError'> occured.
Next entry.
***************************
The entry is 0
Oops! <class 'ZeroDivisionError'> occured.
Next entry.
***************************
The entry is 2
The reciprocal of 2 is 0.5
```

Here in this example, we are looping through the list ‘lst’ and are trying to divide the number 1 by each of the elements in the list ‘lst’.

When we divide the number 1 by character ‘b’, it throws ValueError.

When we divide the number 1 by the number ‘0’, it throws ZeroDivisionError.

When we divide the number 1 by the number 2, then it gives the result as 0.5

Note: We can have only one ‘try’ block and one or more ‘except’ clauses. But the except clause(s) execute only if an exception occurs in the ‘try’ block. Otherwise the ‘except’ block doesn’t execute at all.
The name of the exception occurred is obtained by the function **sys.exc_info()**

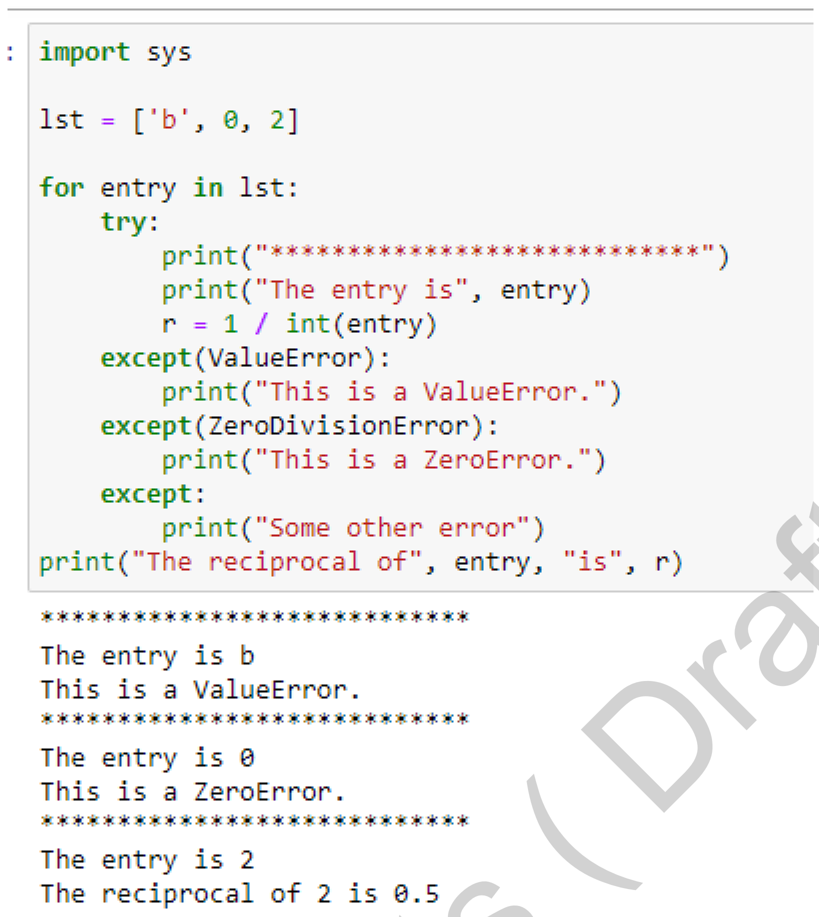
Catching Specific Exceptions in Python

In the above example, we did not mention any exception in the except clause.

This is not a good programming practice as it will catch all exceptions and handle every case in the same way. We can specify which exceptions an except clause will catch.

A try clause can have any number of except clauses to handle them differently but only one will be executed in case an exception occurs.

The below example was discussed at the timestamp 7:45



```
import sys

lst = ['b', 0, 2]

for entry in lst:
    try:
        print("****************************")
        print("The entry is", entry)
        r = 1 / int(entry)
    except(ValueError):
        print("This is a ValueError.")
    except(ZeroDivisionError):
        print("This is a ZeroError.")
    except:
        print("Some other error")
print("The reciprocal of", entry, "is", r)
```

```
****************************
The entry is b
This is a ValueError.
****************************
The entry is 0
This is a ZeroError.
****************************
The entry is 2
The reciprocal of 2 is 0.5
```

In this example, we are handling the ZeroDivisionError and ValueError exceptions separately by writing separate 'except' clauses for them. After this we have again defined an 'except' clause that handles other exceptions.

Raising Exceptions

➢ In Python, exceptions are raised whenever the corresponding errors occur at the runtime. But we also can forcefully raise exceptions using the 'raise' keyword.
➢ We can also optionally pass in value to the exception to clarify why that exception was raised.

The below examples illustrate how to manually raise exceptions. These examples were discussed starting from the timestamp 9:30.

```
: raise KeyboardInterrupt
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-15-7d145351408f> in <module>()
----> 1 raise KeyboardInterrupt

KeyboardInterrupt:
```

```
: raise MemoryError("This is memory Error")
```

```
---------------------------------------------------------------------------
MemoryError                               Traceback (most recent call last)
<ipython-input-16-0ce8140e6e3c> in <module>()
----> 1 raise MemoryError("This is memory Error")

MemoryError: This is memory Error
```

```
: try:
      num = int(input("Enter a positive integer:"))
      if num <= 0:
          raise ValueError("Error:Entered negative number")
  except ValueError as e:
      print(e)
```

```
Enter a positive integer:-10
Error:Entered negative number
```

In this above example, we are checking if the given input number is less than or equal to 0. If yes, then we are raising the ‘ValueError’ exception and in the ‘except’ clause we are printing the value we passed to the exception to clearly indicate why the error has occurred.

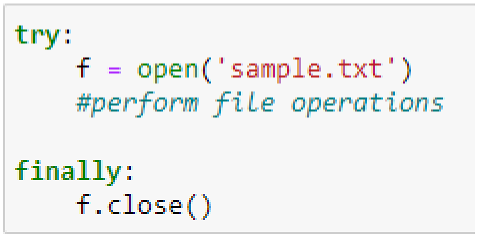
As the value entered is less than 0, the ‘ValueError’ got raised and the message is getting printed.

Try-finally

- The ‘**try**’ statement in Python can have an optional ‘**finally**’ clause.
- Irrespective of whether an exception occurs or not, the ‘finally’ block gets executed.
- In case, if any exception occurs in the **‘try’** block, then the corresponding **‘except’** block gets executed and then the **‘finally’** block gets executed.

- ➢ In case, if there are no exceptions occurred in the **‘try’** block, then immediately the **‘finally’** block gets executed.
- ➢ This clause is executed no matter what, and is generally used to release external resources.The **‘finally’** block is always present after the **‘try’** and all the **‘except’** blocks.
- ➢ It is always a good programming practice to specify the **‘finally’** block whenever we are using the **‘try’** and **’except’** blocks in our code.

The below example was discussed at the timestamp 12:45 that illustrates the usage of ‘**finally**’ block.



```
try:
    f = open('sample.txt')
    #perform file operations

finally:
    f.close()
```

In this example, we are performing the file closing operation in the **‘finally’** block because sometimes whenever we open a file and are working on it and if any exception occurs, then the code execution stops without closing the file. In order to release the resources occupied, we can use **‘finally’** block.

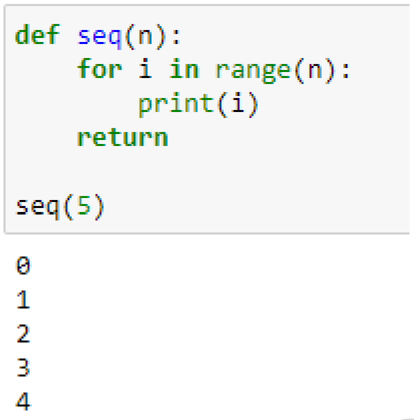
13.4 Debugging in Python

Debugging is about understanding where the bugs are in our program. **pdb** implements an interactive debugging environment for Python programs. It includes features to let you pause your program, look at the values of variables, and watch program execution step-by-step, so you can understand what your program actually does and find bugs in the logic.

Starting the Debugger

From the Command Line

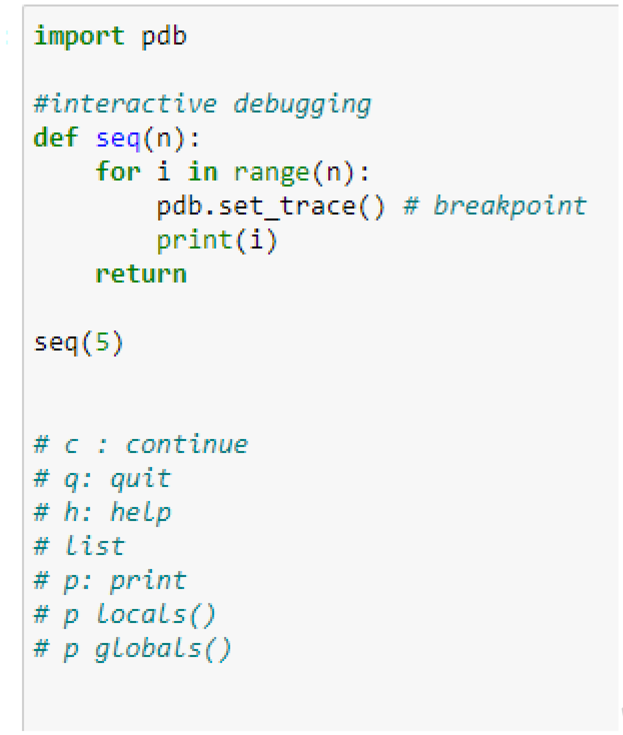
The below examples were discussed starting from the timestamp 0:52



```python
def seq(n):
    for i in range(n):
        print(i)
    return

seq(5)
```

```
0
1
2
3
4
```

From within the program



```
import pdb

#interactive debugging
def seq(n):
    for i in range(n):
        pdb.set_trace() # breakpoint
        print(i)
    return

seq(5)


# c : continue
# q: quit
# h: help
# list
# p: print
# p locals()
# p globals()
```

Debugger Commands

1) **h(elp)**

   Without argument, print the list of available commands. With a command as argument, print help about that command. help pdb displays the full documentation (the docstring of the pdb module). Since the command argument must be an identifier, help exec must be entered to get help on the ! command.

2) **w(here)**

   Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame, which determines the context of most commands.

3) **d(own)**

   Move the current frame count (default one) levels down in the stack trace (to a newer frame).

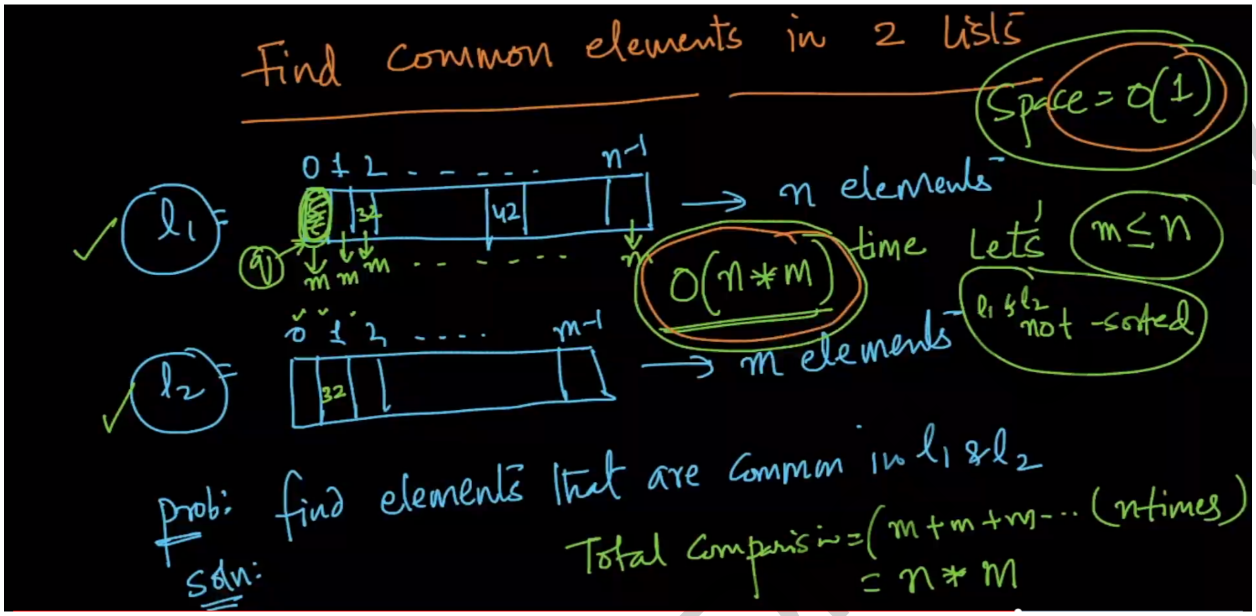
4) **c(ont(inue))**

Continue execution, only stop when a breakpoint is encountered.

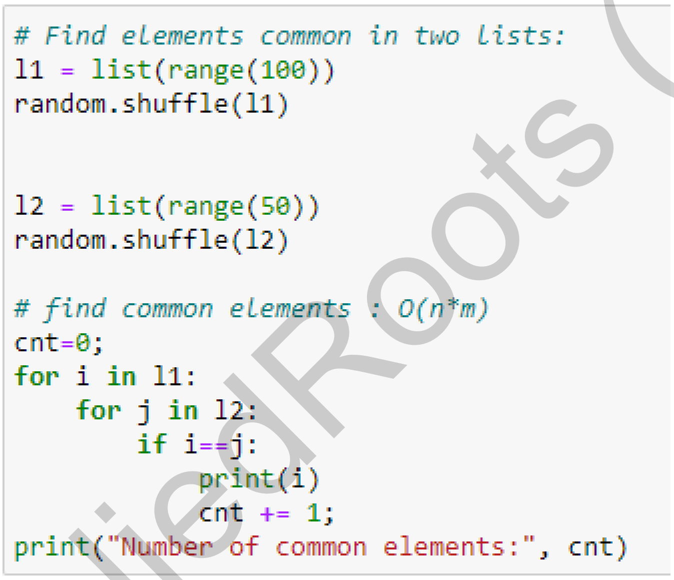
5) **q(uit)**

Quit from the debugger. The program being executed is aborted.

13.5 Solved Problem: Finding the elements common in two lists



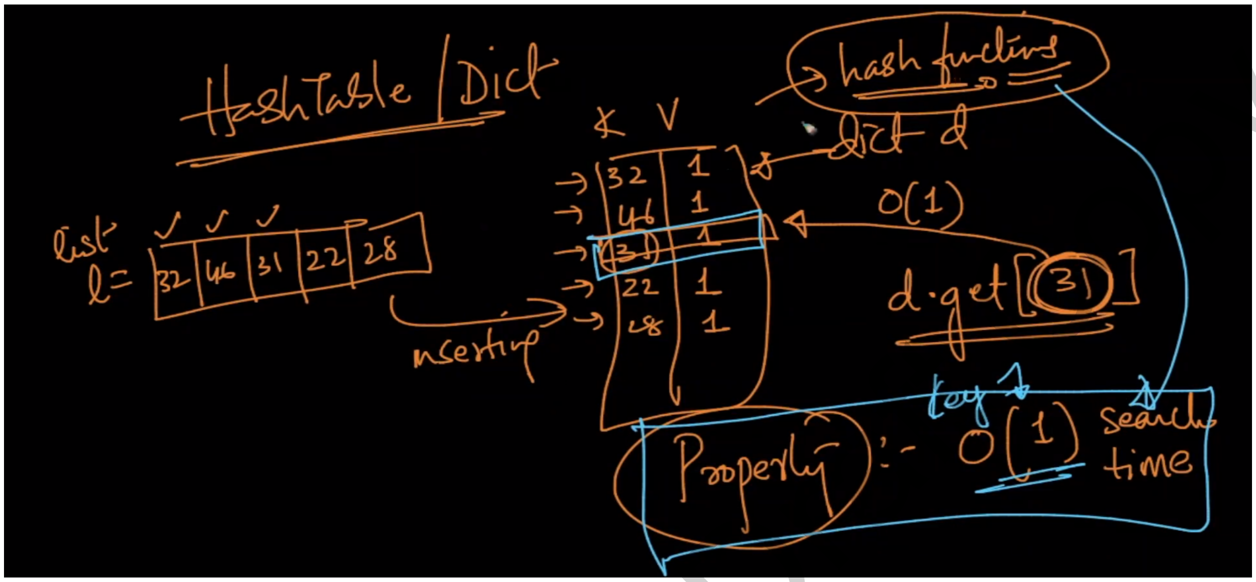
Below is the code snippet that was discussed at the timestamp 4:20

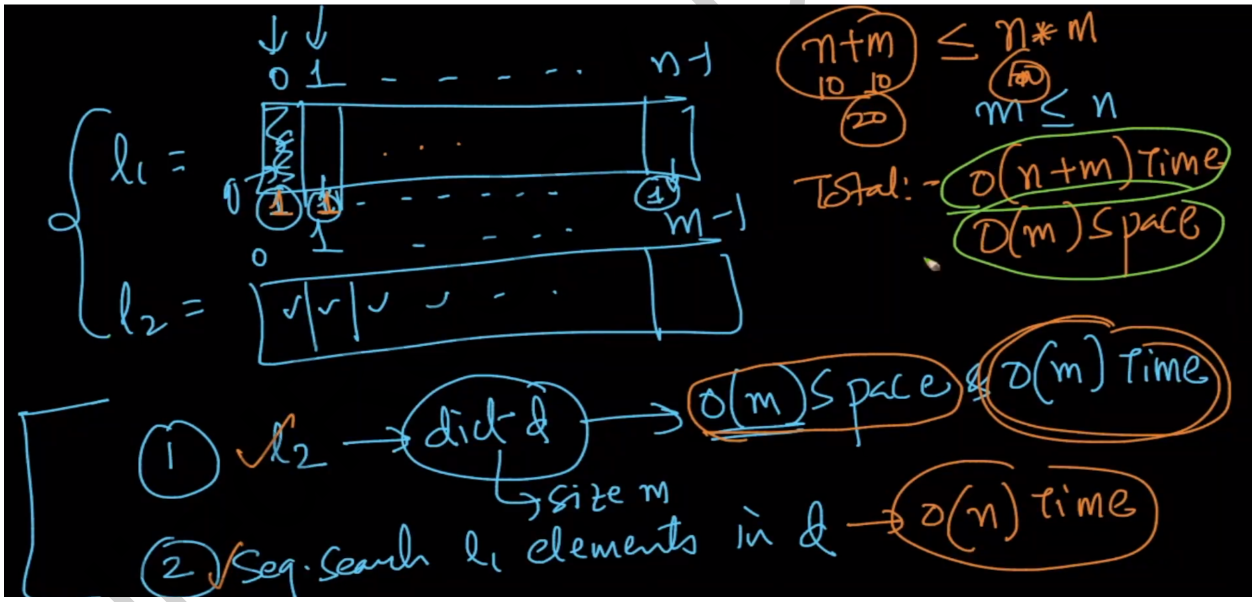


```
# Find elements common in two lists:
l1 = list(range(100))
random.shuffle(l1)


l2 = list(range(50))
random.shuffle(l2)

# find common elements : O(n*m)
cnt=0;
for i in l1:
    for j in l2:
        if i==j:
            print(i)
            cnt += 1;
print("Number of common elements:", cnt)
```

- ➢ In the above example, we are generating two lists 'l1' and 'l2'.
- ➢ We are initializing a variable 'cnt' to 0.
- ➢ We are looping through each of the lists using two 'for' loops and comparing each element in 'l1' with each element in 'l2'.
- ➢ If we find any element present in both the lists, then we have to increment 'cnt' by 1.
- ➢ If 'n' is the number of elements in 'l1' and 'm' is the number of elements in 'l2', then the time complexity would be O(n*m). As we are using only one variable (ie., 'cnt'), the space complexity is O(1).

13.6 Solved Problem: Finding the elements common in two lists using Hashtable/Dict




0 1 n-1
l1 =
m-1
l2 =
n+m ≤ n*m
m ≤ n
Total :- O(n+m) Time
O(m) Space
1 l2 → dict d → O(m) Space & O(m) Time
size m
2 Seq. Search l1 elements in d → O(n) time

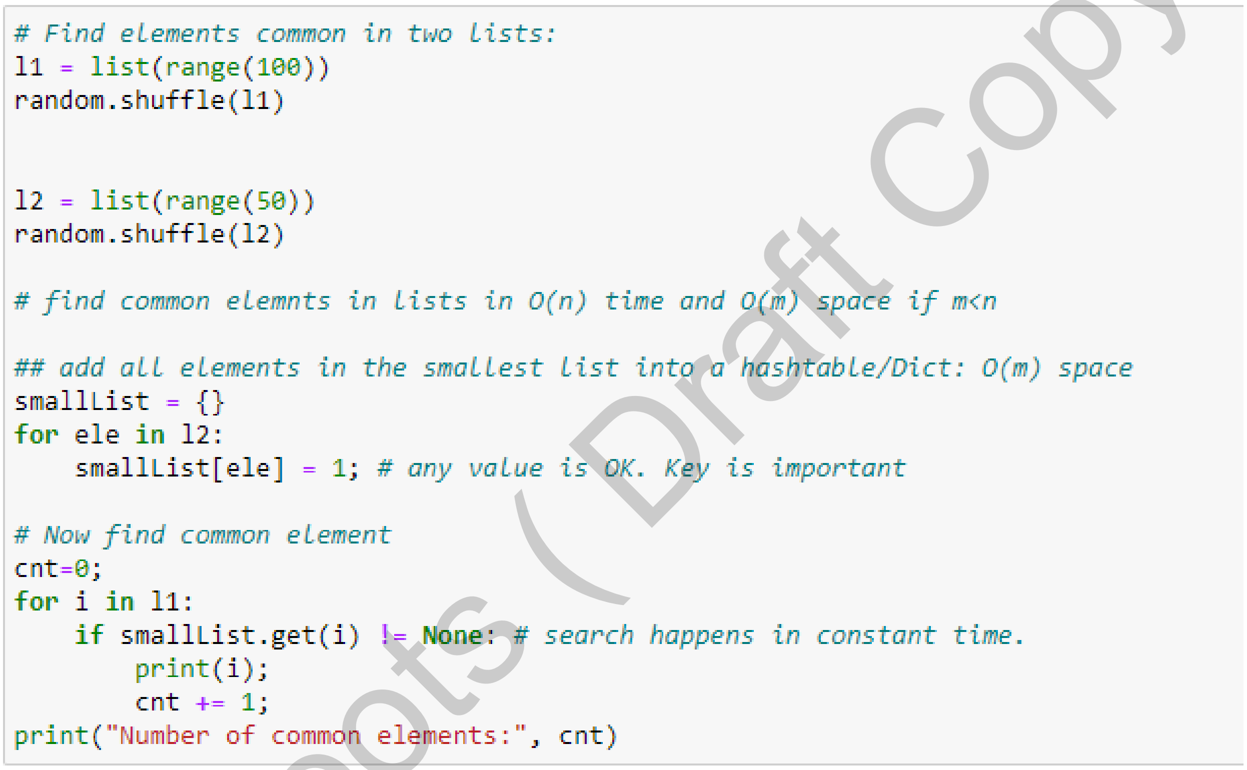
n→ number of elements in list ‘l1’
m→ number of elements in list ‘l2’

In the previous scenario, we have see the time complexity was O(n*m) and the space complexity was O(1). In order to improve the time complexity, we go with another approach. In this approach, we are using a dictionary/hashtable to store the elements of the list ‘l2’. It takes O(m) time complexity to store all the elements of ‘l2’ in the dictionary.

After storing the elements of ‘l2’ in the dictionary, we are checking if each element of ‘l1’ is present in the dictionary as a key. For this comparison purpose, the time complexity is O(n). The space complexity used to store the ‘m’ elements in the dictionary is O(m).

So for the addition of ‘m’ elements into the dictionary, the time complexity is O(n) and the time complexity for comparison of ‘n’ elements with the ‘m’ keys is O(m). So the total time complexity is O(m+n) and the total space complexity is O(m).

The below example was discussed at the timestamp 9:30 in the video.



```
# Find elements common in two lists:
l1 = list(range(100))
random.shuffle(l1)


l2 = list(range(50))
random.shuffle(l2)

# find common elemnts in lists in O(n) time and O(m) space if m<n

## add all elements in the smallest list into a hashtable/Dict: O(m) space
smallList = {}
for ele in l2:
    smallList[ele] = 1; # any value is OK. Key is important

# Now find common element
cnt=0;
for i in l1:
    if smallList.get(i) != None: # search happens in constant time.
        print(i);
        cnt += 1;
print("Number of common elements:", cnt)
```

Time Complexity for storing the ‘m’ elements in the dictionary = O(m)

Time Complexity for checking if all the ‘n’ elements are present in the dictionary = O(n)

The total time complexity of this program = O(n+m)

Space Complexity for this program = O(m) (as we are storing the ‘m’ values in the dictionary)