Python

Python readline()

```
File = open ("filename. txt", "t")

all_lines = file.readlines ()
print (all_lines)
```

Read a file line by line in Python

Prerequisites:

- Open a file
- Access modes
- Close a file

Python provides inbuilt functions for creating, writing, and reading files. There are two types of files that can be handled in python, normal text files and binary files (written in binary language, 0s, and 1s). In this article, we are going to study reading line by line from a file.

Method 1: Read a File Line by Line using readlines()

readlines() is used to read all the lines at a single go and then return them as each line a string element in a list. This function can be used for small files, as it reads the whole file content to the memory, then split it into separate lines. We can iterate over the list and strip the newline '\n' character using strip() function.

Example:

Python3

```
# Python code to
# demonstrate readlines()
L = ["Geeks\n", "for\n", "Geeks\n"]

# writing to file
file1 = open('myfile.txt', 'w')
file1.writelines(L)
file1.close()

# Using readlines()
file1 = open('myfile.txt', 'r')
Lines = file1.readlines()

count = 0
# Strips the newline character
for line in Lines:
    count += 1
    print('Line{}: {}'.format(count, line.strip()))
```

Output:

```
Line1: Geeks
Line2: for
Line3: Geeks
```

The time complexity of the given Python code is O(n), where n is the total number of lines in the file

The auxiliary space complexity of the code is O(n), where n is the total number of lines in the file.

Method 2: Read a File Line by Line using readline()

readline() function reads a line of the file and return it in the form of the string. It takes a parameter n, which specifies the maximum number of bytes that will be read. However, does not reads more than one line, even if n exceeds the length of the line. It will be efficient when reading a large file because instead of fetching all the data in one go, it fetches line by line. readline() returns the next line of the file which contains a newline character in the end. Also, if the end of the file is reached, it will return an empty string.

Example:

STEP BY STEP APPROACH :

1. Create a list L with three string elements containing newline characters.
2. Open a file named myfile.txt in write mode and assign it to the variable file1.
3. Write the contents of the list L to the file using the writelines() method of the file object file1.
4. Close the file object file1 using the close() method.
5. Open the file named myfile.txt in read mode and assign it to the variable file1.
6. Initialize a variable count to 0.
7. Start an infinite loop.
8. Increment the variable count by 1 in each iteration of the loop.
9. Read the next line from the file object file1 using the readline() method and assign it to the variable line.
10. Check if the line variable is empty. If it is empty, it means that the end of the file has been reached. In that case, break out of the loop using the break statement.
11. Print the value of count and the contents of line using the format() method of the string class. The strip() method is used to remove the newline character at the end of the line.
12. Close the file object file1 using the close() method.

```
# Python program to
# demonstrate readline()

L = ["Geeks\n", "for\n", "Geeks\n"]

# Writing to a file
file1 = open('myfile.txt', 'w')
file1.writelines(L)
file1.close()

# Using readline()
file1 = open('myfile.txt', 'r')
count = 0

while True:
    count += 1

    # Get next line from file
    line = file1.readline()

    # if line is empty
    # end of file is reached
    if not line:
        break
    print("Line{}: {}".format(count, line.strip()))

file1.close()
```

Output:

```
Line1 Geeks

Line2 for

Line3 Geeks
```

Method 3: Read a File Line by Line using for loop

An iterable object is returned by open() function while opening a file. This final way of reading a file line-by-line includes iterating over a file object in for a loop. In doing this we are taking advantage of a built-in Python function that allows us to iterate over the file object implicitly using a for loop in a combination with using the iterable object. This approach takes fewer lines of code, which is always the best practice worthy of following.

```
# Python program to
# demonstrate reading files
# using for loop

L = ["Geeks\n", 'for\n', 'Geeks\n']

# Writing to file
file1 = open('myfile.txt', 'w')
file1.writelines(L)
file1.close()

# Opening file
file1 = open('myfile.txt', 'r')
count = 0

# Using for loop
print("Using for loop")
for line in file1:
    count += 1
    print('Line{}: {}'.format(count, line.strip()))

# Closing files
file1.close()
```

Output:

```
Using for loop

Line1: Geeks

Line2: for

Line3: Geeks
```

Method 4: Read a File Line by Line using for loop and list comprehension

A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element. Here, we will read the text file and print the raw data including the new line character in another output we removed all the new line characters from the list

Example

Python3

```
with open('myfile.txt') as f:
    lines = [line for line in f]
print(lines)
# removing the new line characters
with open('myfile.txt') as f:
    lines = [line.rstrip() for line in f]
print(lines)
```

Output:

```
['Geeks\n', 'For\n', 'Geeks']
['Geeks', 'For', 'Geeks']
```

With statement

In the above approaches, every time the file is opened it is needed to be closed explicitly. If one forgets to close the file, it may introduce several bugs in the code, i.e. many changes in files do not go into effect until the file is properly closed. To prevent this with statement can be used. The With statement in Python is used in exception handling to make the code cleaner and much more readable. It simplifies the management of common resources like file streams. Observe the following code example on how the use of with statement makes the code cleaner. There is no need to call file.close() when using with the statement. The with the statement itself ensures proper acquisition and release of resources.

```python
# Python program to
# demonstrate with
# statement
L = ["Geeks\n", "for\n", "Geeks\n"]

# Writing to file
with open('myfile.txt', "w") as fp:
    fp.writelines(L)

# using readlines()
count = 0
print("Using readlines()")
with open('myfile.txt') as fp:
    Lines = fp.readlines()
    for line in Lines:
        count += 1
        print("Line{}: {}".format(count, line.strip()))

# Using readline()
count = 0
print("\nUsing readline()")
with open('myfile.txt') as fp:
    while True:
        count += 1
        line = fp.readline()

        if not line:
            break
        print("Line{}: {}".format(count, line.strip()))

# Using for loop
count = 0
print("\nUsing for loop")
with open('myfile.txt') as fp:
    for line in fp:
        count += 1
        print("Line{}: {}".format(count, line.strip()))
```

Output:

```
Using readlines()
Line1: Geeks
Line2: for
Line3: Geeks

Using readline()
Line1: Geeks
Line2: for
Line3: Geeks

Using for loop
Line1: Geeks
Line2: for
Line3: Geeks
```