Files

A common programming task is to retrieve input from a file using the built-in ***open()*** function instead of using keyboard entry.

The open() built-in function requires a single argument that specifies the path to the file. Ex: open('myfile.txt') opens myfile.txt located in the same directory as the executing script. Full path names can also be specified, as in open('C:\\Users\\BWayne\\tax\_return.txt'). The ***file.close()*** method closes the file, after which no more reads or writes to the file are allowed.

Complete the statement to open the file "readme.txt" for reading.

my\_file =  open('readme.txt')

#The open() function opens the specified file and returns a new file object.

Complete the statement to read up to 500 bytes from "readme.txt" into the contents variable.

my\_file = open('readme.txt')

contents = my\_file.read(500)

Complete the program by echoing the second line of "readme.txt"

my\_file = open('readme.txt')   
lines = my\_file.readlines()

print(lines[1])

# Read file contents

print ('Reading in data....')

f = open('mydata.txt')

lines = f.readlines()

f.close()

# Iterate over each line

print('\nCalculating average....')

total = 0

for ln in lines:

total += int(ln)

# Compute result

avg = total/len(lines)

print(f'Average value: {avg}')

One of the most common programming tasks is to read data from a file and then process that data to produce a useful result.

Thus, a program commonly 1) reads the contents of a file, 2) iterates over each line to process data values, and 3) computes some value, such as the average value.

Iterating over each line of a file is so common that file objects support iteration using the for .... in syntax. The below example echoes the contents of a file:

"""Echo the contents of a file."""

f = open('myfile.txt')

for line in f:

print(line, end="")

f.close()

**Writing to a file**

Programs write to a file to store data permanently. The ***file.write()*** method writes a string argument to a file.

The write() method accepts a string argument only. Integers and floating-point values must first be converted using str(), as in f.write(str(5.75)).

Numeric values must be converted to strings.

or adjust the buffering to add text to file every 100 bytes wth: Ex:f = open('myfile.txt', 'w', buffering=100)

num1 = 5

num2 = 7.5

num3 = num1 + num2

f = open('myfile.txt', 'w')

f.write(str(num1))

f.write(' + ')

f.write(str(num2))

f.write(' = ')

f.write(str(num3))

f.close()

Final contents of myfile.txt:

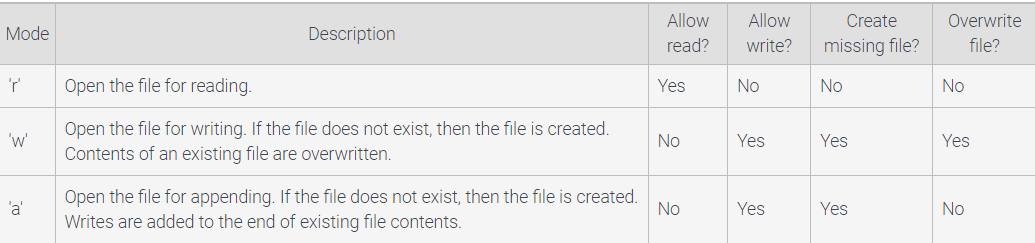
5 + 7.5 = 12.5

The write() method accepts a string argument only. Integers and floating-point values must first be converted using str(), as in f.write(str(5.75)).

When writing to a file, the mode of the file must be explicitly set in the open() function call.

Mode: indicates how a file is opened

writing allowed (y/n)

if content already, append or overwrite.

Statement myfile.write('\n') executes. The interpreter stores '\n' in a buffer. Writing a newline causes the buffer to be written to the file, so 'Num5' is placed in myfile.txt.

Graphical user interface, text, application, Word

Description automatically generated

Or use buffering control. The below code, sets buffer to transfer to file every 100 bytes

Ex:f = open('myfile.txt', 'w', buffering=100)

Or flush() to force an output buffer to write to disk.

import os

*# Open a file with default line-buffering.*

f = open('myfile.txt', 'w')

*# No newline character, so not written to disk immediately*

f.write('Write me to a file, please!')

*# Force output buffer to be written to disk*

f.flush()

os.fsync(f.fileno())

# **Interacting with file systems**

os.path module contains portable functions for handling file paths so they work across different Os’s.

~~literal path = "subdir\\bat\_mobile.jpg~~

instead write:

path = os.path.join('subdir', 'bat\_mobile.jpg')

* subdir\\bat\_mobile.jpg (in Windows)
* subdir/bat\_mobile.jpg (in Linux)

tokens = 'C:\\Users\\BWayne\\tax\_return.txt'.split(os.path.sep)

* ['C:', 'Users', 'BWayne', 'tax\_return.txt']

# **The 'with' statement**

with open('myfile.txt', 'r') as myfile:

*# Statement-1*

*# Statement-2*

*# ....*

*# Statement-N*

A ***with statement*** can be used to open a file, execute a block of statements, and automatically close the file when complete.

 The with statement creates a ***context manager***, which manages the use of a resource, such as a file, by performing set-up and teardown operations. Forgetting to close a file can sometimes cause the file incapable of being written to by other programs.

print('Opening myfile.txt')

*# Open a file for reading and appending*

with open('myfile.txt', 'r+') as f:

*# Read in two integers*

num1 = int(f.readline())

num2 = int(f.readline())

product = num1 \* num2

*# Write back result on own line*

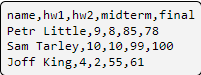
f.write('\n')

f.write(str(product))

*# No need to call f.close()*

*Good practice is to use a with statement when opening files to guarantee that the file is closed when no longer needed.*

# **12.7 Comma separated values files**

A ***comma separated values*** (csv) file is a simple text-based file format that uses commas to separate data items, called ***fields***.

import csv

with open('grades.csv', 'r') as csvfile:

grades\_reader = csv.reader(csvfile, delimiter=',')

row\_num = 1

for row in grades\_reader:

print(f'Row #{row\_num}: {row}')

row\_num += 1

A picture containing text

Description automatically generated