**Files.** Ink was made in the ancient world. It was applied to scrolls to print words and pictures. Today we persist our information on computers.  
**With files,** a form of long-term storage, we persist data. We use the open() method to access files. Methods like readlines() handle their data.  
**A program.** There are some tricks to handling text files. Even if we just want to display all the lines from a file, newlines must be handled. We read all lines from a file with readlines().

**Tip:**We first must create a new file object. And then we loop over the list returned by readlines().

**Raw string:**This program uses the path syntax a Windows system. We start the string with "r" to avoid errors with backslashes.

**Based on:** Python 3

**Python program that reads all lines**

*# Open a file on the disk.*

f = open(*r"C:\perls.txt"*, "r")

*# Print all its lines.*

for line in f.readlines():

*# Modify the end argument.*

print(line, end="")

**Output**

Line 1

Line 2

**Paths.** Please change the path to an existing text file (or create a "perls.txt" file at the required location). The end="" parameter to print() modifies the behavior of print.[Path](https://www.dotnetperls.com/path-python)

**End:**When end is assigned an empty string, the trailing newline is not printed to the console. This results in only one newline.

[Strings](https://www.dotnetperls.com/string-python)[Console, Print](https://www.dotnetperls.com/console-python)

**Note:**The problem occurs because the line itself has a newline, and print also adds a newline of its own.

**With.** This statement cleans up resources. It makes simpler the task of freeing system resources. It is used with file handling: open() is a common call. It improves readability.

**First:**We use "with" in this simple program. The program opens and reads from a file.

**Tip:**This statement makes sure the system resources are cleaned up properly. The with statement is similar to a try-finally statement.

**Python program that uses with statement**

name = *r"C:\perls.txt"*

*# Open the file in a with statement.*

with open(name) as f:

print(f.readline(), end="")

*# Repeat.*

with open(name) as f:

print(f.readline(), end="")

**Output**

First line

First line

**Pickle, list.** Often we need to store objects. With pickle, we write collections such as lists to a data file. It supports many objects. The with statement improves resource cleanup.[List](https://www.dotnetperls.com/list-python)

**However:**In this example, we create a list. We pass this list to pickle.dump().

**Dump:**This writes the list contents in binary form to the file f.pickle. The extension (pickle) has no importance.

**Then:**After we call pickle.dump(), we ignore the original list in memory. We load that same data back from the disk with pickle.load().

**Python program that uses pickle, list**

import pickle

*# Input list data.*

list = ["one", "two", "three"]

print("before:", list)

*# Open the file and call pickle.dump.*

with open("f.pickle", "wb") as f:

pickle.dump(list, f)

*# Open the file and call pickle.load.*

with open("f.pickle", "rb") as f:

data = pickle.load(f)

print("after:", data)

**Output**

before: ['one', 'two', 'three']

after: ['one', 'two', 'three']

**New, empty file.** The second argument to open() is a string containing "mode" flag characters. The "w" specifies write-only mode—no appending or reading is done.

**Erased:**If the file happens to exist, it is erased. So be careful when developing programs with this call.

**Python program that creates new, empty file**

*# Create new empty file.*

*# ... If the file exists, it will be cleared of content.*

f = open(*"C:\\programs\\test.file"*, "w")

**Write lines.** This program writes lines to a file. It first creates an empty file for writing. It specifies the "w" mode to create an empty file. Then it writes two lines.

**Tip:**The line separators (newline chars) are needed. There is no "writeline" method available.

**Python program that uses write**

*# Create an empty file for writing.*

with open(*"C:\\programs\\test.file"*, "w") as f:

*# Write two lines to the file.*

f.write("cat\n")

f.write("bird\n")

**Result: test.file**

cat

bird

**Count character frequencies.** This program opens a file and counts each character using a frequency dictionary. It combines open(), readlines, and dictionary's get().

**Strip:**The program strips each line because we do not want to bother with newline characters.

**Get:**The code uses the two-argument form of get. If a value exists, it is returned—otherwise, 0 is returned.

[Dictionary: get](https://www.dotnetperls.com/dictionary-python)

**Example text, file.txt: Python**

aaaa

bbbbb

aaaa

bbbbb

aaaa bbbbb

CCcc

xx

y y y y y

Z

**Python program that counts characters in file**

*# Open a file.*

f = open(*r"C:\programs\file.txt"*, "r")

*# Stores character counts.*

chars = {}

*# Loop over file and increment a key for each char.*

for line in f.readlines():

for c in line.strip():

*# Get existing value for this char or a default of zero.*

*# ... Add one and store that.*

chars[c] = chars.get(c, 0) + 1

*# Print character counts.*

for item in chars.items():

print(item)

**Output**

('a', 12)

(' ', 5)

('C', 2)

('b', 15)

('c', 2)

('y', 5)

('x', 2)

('Z', 1)

**Benchmarking readlines, read.** There is significant overhead in accessing a file for a read. I tested readlines against read() on a file with about 1000 lines.

**And:**It was far faster to read the entire file in a single call with the read() method. Using readlines was slower.

**Analysis:**Readlines likely causes a disk access, or more disk accesses than read. For some files, using read() is a better choice.

**File, line repeated 1000 times: test.file**

This is an interesting file.

This is an interesting file.

...

**Python program that times readlines, read**

import time

print(time.time())

*# Version 1: use readlines.*

i = 0

while i < 10000:

with open("C:\\programs\\test.file", "r") as f:

count = 0

for line in f.readlines():

count += len(line)

i += 1

print(time.time())

*# Version 2: use read.*

i = 0

while i < 10000:

with open("C:\\programs\\test.file", "r") as f:

count = 0

data = f.read()

count = len(data)

i += 1

print(time.time())

**Output**

1406148416.003978

1406148423.383404 readlines = *7.38 s*

1406148425.989555 read = *2.61 s*

**IOError.** File handling is an error-prone task. Sometimes a file is moved without our knowledge. Sometimes even a hardware error can occur. We cannot prevent this.

**So:**We must handle IOError in important programs. We can use exception handling, like try and except.

[IOError](https://www.dotnetperls.com/ioerror-python)[Error](https://www.dotnetperls.com/error-python)  
**Read binary data.** A Python program can read binary data from a file. We must add a "b" at the end of the mode argument. We call read() to read the entire file into a bytes object.[Bytes, bytearray](https://www.dotnetperls.com/bytes-python)

**Here:**A file on the local disk is read. This is a gzip file, which has special bytes at its start.

**Python that reads binary data**

*# Read file in binary form.*

*# ... Specify "b" for binary read and write.*

f = open(r"C:\stage-perls-cf\file-python", *"rb"*)

*# Read the entire file.*

data = f.read()

*# Print length of result bytes object.*

*# ... Print first three bytes (which are gzip).*

print(len(data))

print(data[0])

print(data[1])

print(data[2])

**Output**

42078

31

139

8

**Formats.** Markup files are often used in computer programs. We handle HTML and XML files. There are many ways to parse or scan these formats. I show HTMLParser and Expat.[HTML: HTMLParser](https://www.dotnetperls.com/html-python)[XML: Expat](https://www.dotnetperls.com/xml-python)

**CSV files:**Parsing CSV files is important. It is tedious. We introduce the csv module to help make it easier.

**Textwrap:**The textwrap module can be to rewrap text files. This can improve the formatting of files.

[Textwrap](https://www.dotnetperls.com/textwrap-python)  
**Modes.** The default mode for the file open method in Python is "r." This means "read." The Python documentation has more details on possible modes.

The mode argument is optional; 'r' will be assumed if it's omitted.

[Input and Output: Python.org](http://docs.python.org/3/tutorial/inputoutput.html)  
**Complexity.** Files are a source of complexity in programs. We must process known file formats. And sometimes we also must handle invalid or corrupted files.  
**A review.** File handling is an important yet error-prone aspect of program development. It is essential. It gives us data persistence.