Department of Computer Science and Engineering (UG Studies)
PES University, Bangalore, India

Introduction to Computing using Python (UE19CS101)

Mr. Prakash C O
Asst. Professor,
Dept. of CSE, PESU
coprakasha@pes.edu

Files in Python

The variables we create in our program remain in memory until the end of the program. They are lost when the program terminates. **We require files to make the information permanent. We call this feature persistence**. The files persist not only after the program termination, but also after rebooting or after restarting a computer.

The files belong to the operating system that runs the computer. The naming convention depends on the operating system.

- **In Microsoft Windows**, The total file name (called the path) has the drive name, then ":", then \directory name – this could repeat number of times – then the filename.
- **In linux**, we have no drive name. The file name (path) has /directory name – repeats number of times -then the filename.

File Types

In Python, a file is categorized as either **text** or **binary**.

Text files **(Examples: .txt, .rtf, .csv, .py, …)** are structured as a sequence of lines, where each line includes a sequence of characters. Each line is terminated with a special character, called the EOL or End of Line character. There are several types, but the most common is the comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

Text files are humanly readable which is displayed in a sequence of characters that can be easily interpreted by humans as it presented in the textual form.

A binary file **(Examples: .png, .jpg …)** is any type of file that is not a text file. In binary files data is displayed in some encoded format (using 0's and 1's) instead of plain characters. Typically they contain the sequence of bytes. Because of their nature, binary files can only be processed by an application that know or understand the file's structure. In other words, they must be applications that can read and interpret binary.

Files on most modern file systems are composed of three main parts:

1. **`Header:`** metadata about the contents of the file (file name, size, type, and so on)
2. **`Data`**: contents of the file as written by the creator or editor
3. **`End of file (EOF)`**: special character that indicates the end of the file

File Objects

A file object allows us to use, access and manipulate all the user accessible files.

To use a file, in Python, we make an association of an entity called file object with the file. This file object refers to some data structures outside of our program which in turn refers to the file. We call this referred file as a resource.

We get the file resource (when we need) using a function called *open*. We should release this file resource whenever we do not require any more. This is like taking loan from a bank and repaying it.
We release the file resource by calling a function called *close* on the file object.

A file may be

- used for reading if it already exists.
- created or overwritten.
- used for reading as well as writing.

We indicate our intention by specifying what is called the mode - 'r' for read, 'w' for write. etc..

The open() method

Python has a built-in function ***`open()`*** to open a file. This function creates a file object.

Syntax:

```
f = open(file_name, access_mode)
```

Where,

- **`file_name`** = name of the file to be opened
- **`access_mode`** = mode in which the file is to be opened. It can be read, write etc. By default, it opens the file in reading mode (r) if not specified explicitly. Here is the list of different modes in which a file can be opened.

Mode	Description
r	**Opens a file for reading mode only.**
w	**Opens a file for writing mode only. Creates a new file for writing, if the file doesn't exist.**
rb	Opens a binary file for reading mode only.
wb	Opens a binary file for writing mode only. Creates a new file for writing it the file doesn't exist.
r+	**Opens a file for reading and writing mode.**
rb+	Opens a binary file for reading and writing mode.

w+	**Opens a file for writing and reading mode. Creates a new file for writing, if the file doesn't exist.**
wb+	Opens a binary file for writing and reading mode. Creates a new file for writing it the file doesn't exist.
a	**Opens a file for appending. Creates a new file for writing it the file doesn't exist.**
a+	**Opens a file for appending and reading. Creates a new file for reading and writing it the file doesn't exist.**
ab	Opens a binary file for appending. Creates a new file for writing it the file doesn't exist.
ab+	Opens a binary file for reading and writing mode. Creates a new file for reading and writing it the file doesn't exist.

Note: Normally, files are opened in ***text mode***, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent. 'b' appended to the mode opens the file in ***binary mode***: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

A file opening may not always succeed. If it fails, it throws an exception. A few possible exceptions:

- opening a non-existent file for reading
- opening a file for writing where directory does not allow us to create files

Let us examine an example for opening a non-existent file for reading.

```
# filename : 1_file.py

# We specify the physical filename - the name by which the operating system knows the
  file - in a function called open.
# We open file in different modes - reading(r), writing(w), appending(a) ...
# We get a file handle as the result of opening.
# We use the file handle from that point onwards

# gives a runtime error if the file does not exist and we try to open the file for reading
f = open("junk.txt", "r")
```

Output:

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

In the above code, program terminates with the exception FileNotFoundError, because the file ***junk.txt*** doesn't exist.

```
# Let us consider an existing file for opening.
# An open file uses the resources of the operating system.
f = open("t.txt", "r")
# We return the resources by calling a function called close on the open file handle.
f.close()
```

Observe the signature of the function open. The first argument is the filename as per the rules of the operating system. It is a string. The second argument is the mode. The function returns a handle to the file – file object – if it succeeds.

What if we open a file which exists for reading? We get a file object.

```
f = open("t.txt", "r")
print(f, type(f)) # do not worry about this output!
f.close()
```

Output:

```
# <_io.TextIOWrapper name='t.txt' mode='r' encoding='cp1252'> <class '_io.TextIOWrapper'>
```

The object **f** is said to be a wrapper for something. Let us ignore it.
The last statement `f.close()` releases the resources associated with the file object.

Reading and Writing Opened Files

Once you've opened up a file, you'll want to read or write to the file. There are multiple methods that can be called on a file object to help you out:

Method	What It Does
`.readline(size=-1)`	This reads at most size number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.
`.readlines()`	This reads the remaining lines from the file object and returns them as a list.
`.read(size=-1)`	This reads from the file based on the number of size bytes. If no argument is passed or None or -1 is passed, then the entire file is read.

We can read an existing file in a number of ways. We shall examine these in the next few sections. We shall consider the file t.txt for experimenting.

t.txt

```
do not
trouble trouble
till trouble
troubles you
```

We shall now examine parts of the file 2_file_read.py.

```
f = open("t.txt", "r")

line1 = f.readline()
print("first line : ", line1) # do not

line2 = f.readline()
print("second line : ", line2) # trouble trouble
```

```
line1 = line1.strip(); line2 = line2.strip()
print("File contents after removing newline character by strip() method:")
print("first line : ", line1)
print("second line : ", line2)
f.close()
```

Output:

```
first line :  do not

second line :  trouble trouble

File contents after removing newline character by strip() method:
first line :  do not
second line :  trouble trouble
```

In the above code, The statement **`line1 = f.readline()`** reads the first line of the file with which f is associated by the open statement. **The function readline reads a line from the file including the newline character which exists at the end of the line in a file**. **Observe the difference with respect to input(). The function input does not read the newline**.

`print("first line : ", line1)`
In the output, we will see a blank line as line1 has a new line at the end and print by default outputs a newline.

`line2 = f.readline()`
The readline also causes the cursor or offset – the position from the file where the next operation would happen – to change to the location past the end of the line in the file. This readline reads the next line in the file into the variable.

The presence of newline in each variable read from a file causes lots of problem in processing. We would prefer to get them removed. This is the way to remove any white space at either end of a string.

`line1 = line1.strip(); line2 = line2.strip()`

To read a file completely, we may want to use readline in a loop.
This code is taken from the file **2_file_read.py.**

```
f = open("t.txt", "r")
line = f.readline()
while line:
      line = line.strip() # remember to assign back to line!!
      print(line)
      line = f.readline()
f.close()
```

Output:

```
do not
trouble trouble
till trouble
troubles you
```

Observe the presence of the code **`line = f.readline()`** before the loop as well as the last statement of the loop. We read a line before we enter the loop and in the loop, we process that line and then again read one more line at the end of the loop. As we keep reading from a file, we would reach the end of the file at some point. **When we reach the end of the file, there is nothing more to read and readline returns an empty string**. So, now you can understand the condition of the while. Would this code work if the input file is empty?

An equivalent code for program 2_file_read.py without using strip() function

```
f = open("t.txt", "r")
line = f.readline()
while line:
    print(line, end='')
    line = f.readline()
f.close()
```

Output:

```
do not
trouble trouble
till trouble
troubles you
```

Note: In the above code, the statement ***`print(line, end='')`*** **is having the argument** ***`end=''`***, that is used to prevent Python from adding an additional newline to the text that is being printed and only print what is being read from the file.

The code below taken from the file **2_file_read.py** outputs the lines with line number. It is left to you to follow the program!

```
f = open("t.txt", "r")
line = f.readline()
i = 0
while line:
      line = line.strip() # remember to assign back to line!!
      i += 1
      print(i, ":", line)
      line = f.readline()
f.close()
```

Output:

```
1 :  do not
2 :  trouble trouble
3 :  till trouble
4 :  troubles you
```

There are a number of ways reading using a file object.

a) `readline` : reads a line

b) `readlines` : reads the whole file into a list – where each element of the list is a line of the file.

This code segment is from the file 3_file_read.py.

```
f = open("t.txt", "r")
lines = f.readlines()
f.close()
# print(lines) #['do not\n', 'trouble trouble\n', 'till trouble\n', 'troubles you\n']

for line in lines:
      line = line.strip()
      print(line)
```

Output:

```
do not
trouble trouble
till trouble
troubles you
```

Note: The strip() method removes whitespace by default, so there is no need to call it with parameters like '\t' or '\n'. However, strings in Python are immutable and can't be modified. The result is a new string which is returned by the call.

Simplified and equivalent code of the above program 3_file_read.py

```
f = open("t.txt", "r")

for line in f.readlines():
      print(line, end='')
f.close()
```

As reading the whole file is read into the memory, the file should not be very big.

The following code counts the number of occurrences of a particular word. We leave it to you to walk through the logic.

```
# let us count the number of troubles
f = open("t.txt", "r")
lines = f.readlines()
f.close()
# print(lines) #['do not\n', 'trouble trouble\n', 'till trouble\n', 'troubles you\n']

wcount = 0
for line in lines:
      line = line.strip()
      wcount += line.count('trouble')
print("Total trouble : ", wcount)
```

Output:

```
Total trouble :  4
```

There are a number of ways reading using a file object.

a) `readline` : reads a line

b) `readlines` : reads the whole file into a list – where each element of the list is a line of the file

c) `read` : reads a number of bytes from the file. By default reads the whole file into a string.

This code segment is from the file 4_file_read.py.

```
f = open("t.txt", "r")
all = f.read()
print(all, type(all))
f.close()
```

Output:

```
do not
trouble trouble
till trouble
troubles you
<class 'str'>
```

As reading the whole file is read into the memory, the file should not be very big.

 # The program below reads from the file based on the number of size bytes mentioned

```
f = open("t.txt", "r")
firstnbytes = f.read(10)
print(firstnbytes)
f.close()
```

Output:

```
do not
tro
```

The following code finds the unique words in the file. We leave it to you to walk through the logic.

butter.txt: This is the data file used in the following programs.

```
betty botter bought some butter
but the butter was bitter
betty botter bought some better butter
to make the bitter butter better
```

```
# display unique words in the file
f = open("butter.txt", "r")
all = f.read()
f.close()

wordset = set()
for word in all.split():
      wordset.add(word)

for word in sorted(wordset):
      print(word, end=' ')
```

Output:

```
better betty bitter botter bought but butter make some the to was
```

There are a number of ways reading using a file object.

a) `readline` : reads a line

b) `readlines` : reads the whole file into a list – where each element of the list is a line of the file

c) `read` : reads a number of bytes from the file. By default reads the whole file into a string.

d) use the file object as an iterable.

One of the most interesting feature of Python is the for loop. **The for loop can step through any object which is iterable – which supports a function called __iter__ to create an iterator which in turn should support the function __next__ get the next element. The file object is iterable. Each time we iterate through the for loop, we get a line of the file.**

This code segment is from the file 5_file_read.py.

```
f = open("butter.txt", "r")
for line in f :
      line = line.strip()
      print(line)
f.close()
```

Output:

```
betty botter bought some butter
but the butter was bitter
betty botter bought some better butter
to make the bitter butter better
```

When we reach the end of the file, the for loop terminates.

The following program code taken from the file **5_file_read.py** displays the words in the decreasing order of frequency. We leave it to you to walk through the logic.

```
# create a frequency of each word in the file.
# display in the decreasing order of frequency

f = open("butter.txt", "r")
wordfreq = {}
for line in f :
      line = line.strip()
      for word in line.split():
            if word not in wordfreq :
                  wordfreq[word] = 0
            wordfreq[word] += 1
f.close()

#The sorted() function returns a sorted list of the specified iterable object.
for word in sorted(wordfreq, reverse = True, key = lambda k : wordfreq[k]):
      print(word, wordfreq[word])
```

Output:

```
butter 4
betty 2
botter 2
bought 2
some 2
the 2
bitter 2
better 2
```

```
but 1
was 1
to 1
make 1
```

Let us look at a small program to create a file.
This code is taken from the file **6_file_write.py**

```
fout = open("out.txt", "w")
print(fout, type(fout))
print("hello world", file = fout)
fout.close()
```

Observe that the file is opened in write mode.

There are a couple of ways to write to a filename

a) print with the keyword parameter file having the file object as the argument

b) write on file object – is not discussed in this course.

This code is taken from the file **6_file_write.py.**

This code creates a new file with the lines having the word trouble in the file t.txt. We leave it to you to walk through the logic.

```
# pick all lines containing trouble and write to another file

fin = open("t.txt")
fout = open("t_new.txt", "w")
for line in fin:
      if 'trouble' in line :
            line = line.strip()
            print(line, file = fout)
fin.close()
fout.close()
```

Using *with* statement with files

The with statement was introduced in python 2.5. The with statement is useful in the case of manipulating the files. The with statement is used in the scenario where a pair of statements is to be executed with a block of code in between.

The syntax to open a file using with statement is given below.

```
with open(<file name>, <access mode>) as <file-pointer>:
      #statement suite
```

The advantage of using with statement is that it provides the guarantee to close the file regardless of how the nested block exits.

It is always suggestible to use the with statement in the case of files because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file. It doesn't let the file to be corrupted.

Example

```
with open("file.txt",'r') as f:
      content = f.read();
      print(content)
```

Output:

```
 Python is the modern day language. It makes things so simple.
```

What Is a CSV File?

A CSV file (Comma Separated Values file) is a type of plain text file that uses specific structuring to arrange tabular data. Because it's a plain text file, it can contain only actual text data—in other words, printable ASCII or Unicode characters.

The structure of a CSV file is given away by its name. Normally, CSV files use a comma to separate each specific data value. Here's what that structure looks like:

Example: employee_birthday.txt

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Where Do CSV Files Come From?

CSV files are normally created by programs that handle large amounts of data. They are a convenient way to export data from spreadsheets and databases as well as import or use it in other programs. For example, you might export the results of a data mining program to a CSV file and then import that into a spreadsheet to analyze the data, generate graphs for a presentation, or prepare a report for publication.

Parsing CSV Files With Python's Built-in CSV Library

The csv library provides functionality to both read from and write to CSV files. Designed to work out of the box with Excel-generated CSV files, it is easily adapted to work with a variety of CSV formats. The csv library contains objects and other code to read, write, and process data from and to CSV files.

Reading CSV Files With csv

Reading from a CSV file is done using the reader object. The CSV file is opened as a text file with Python's built-in open() function, which returns a file object. This is then passed to the reader, which does the heavy lifting.

Here's the employee_birthday.txt file:

```
name,department,birthday month
John Smith,Accounting,November
Erica Meyers,IT,March
```

Here's code to read it:

```
import csv

with open('employee_birthday.txt') as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    line_count = 0
    for row in csv_reader:
        if line_count == 0:
            print(f'Column names are {", ".join(row)}')
            line_count += 1
        else:
            print(f'\t{row[0]} works in the {row[1]} department, and was born in
{row[2]}.')
            line_count += 1
    print(f'Processed {line_count} lines.')
```

Output:

```
Column names are name, department, birthday month
    John Smith works in the Accounting department, and was born in November.
    Erica Meyers works in the IT department, and was born in March.
Processed 3 lines.
```

Each row returned by the reader is a list of String elements containing the data found by removing the delimiters. The first row returned contains the column names, which is handled in a special way.

References:

1. 17_File.pdf – Prof. N S Kumar, Dept. of CSE, PES University.
2. https://realpython.com/courses/reading-and-writing-csv-files/
3. https://realpython.com/working-with-files-in-python/
4. https://realpython.com/python-csv/
5. http://www.trytoprogram.com/python-programming/python-files-io
6. https://docs.python.org/3/tutorial/inputoutput.html