**Python File Processing & Pickling**

Python file handling also referred to as File Input and Output I/O is one of the essential topics for programmers , automation engineers and testers. It is a requirement to work with files for either writing to a file or read data from a file.

Optimizing a single file operation can help us produce a high-performing application or a robust solution for an automated software testing process.

Let’s take an example, say, we’re going to create a large project in Python that contains a number of workflows. Then, it’s imperative for us to create a log file. By doing so, we shall be doing both the read/write operations on the log files.

Log files are a great tool to debug large programs. It’s always better to think about a scalable design from the beginning, as you won’t regret it.

### **What is the meaning of file handling?**

Here is a basic definition of file handling in Python.

*File is a named location on the system storage which records data for later access. It enables persistent storage in a non-volatile memory i.e. Hard disk.*

In Python, file processing takes place in the following order.

* Open a file that returns a file handle.
* Use the handle to perform read or write action.
* Close the file handle.

Before you do a read or write operation to a file in Python, you need to open it first. As the read/write transaction completes, you should close it to free the resources tied by the processing of the file.

In the next sections, we’ll touch upon all the Python file handling topics one by one. Since it’s an example-driven Python tutorial, it will be nice to open a Python console to test-run the code as we go along.

Python has several functions for creating, reading, updating, and deleting files. The key function for working with files in Python is the open() function.

The open() function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

### **Open a file in Python**

To read or write to a file, we need to open it first. To open a file in Python, we use its built-in open() function. This function returns a file object, i.e., a handle. We can also use it to read or modify the file.

**There are four different methods (modes) for opening a file:**

**"r"** - Read - Default value. Opens a file for reading, error if the file does not exist

**"a"** - Append - Opens a file for appending, creates the file if it does not exist

**"w"** - Write - Opens a file for writing, creates the file if it does not exist

**"x"** - Create - Creates the specified file, returns an error if the file exists

**In addition we can specify if the file should be handled as binary or text mode**

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

# 

# 

# **Operations on a File in Python**

The operations performed on a file are called the file operation or processing. Following is the list of operations that can be applied on a file in python.

1. Opening or Creating a File
2. Writing to a File
3. Reading from a File
4. Closing a File
5. Renaming a File
6. Deleting a File

# **Files Types in Python**

Even though we have various file formats in our computer, the in-built methods in Python can handle two major types of files. The types are text files and binary files. The text and binary files have **.txt** and **.bin** as their extensions respectively.

We have to understand both types of files. Text files are used to store human readable information whereas the binary files contain computer readable information that are written using binary digits 0s and 1s.

The file processing in python is done by two different types of approaches.

a) One is processing the content in the file.

b) Two is processing the directory.

The first one can be done by the built in functions. The second one can be processed by the **os** module methods. In this lesson we shall concentrate only on the file content manipulation.

# 

# **Opening or Creating a New File in Python**

The very first process to do in file handling is the creation of a file. The method open() is used to open an existing file or create a new file. If the complete directory is not given then the file will be created in the directory in which the Python file is stored. The syntax for using open() method is given below.

**Syntax:** file\_object = *open*( file\_name, “Access Mode”, **Buffering** )

Below are the parameter details.

**<access\_mode>** - It’s an integer representing the file opening mode, e.g., read, write, append, etc. It’s an optional parameter. By default, it is set to read-only **<r>**. In this mode, we get data in text format after reading from the file.

On the other hand, the binary mode returns bytes. It’s preferable for accessing the non-text files like an image or the Exe files. See the table below for details. It lists down the available access modes.

**<buffering>** - The default value is **0**, which means buffering won’t happen. If the value is **1**, then line buffering will take place while accessing the file. If it’s higher than **1**, then the buffering action will run as per the buffer size. In the case of a negative value, the default behavior is considered.

**<file\_name>** - It’s a string representing the name of the file you want to access.

**File Open Modes**

|  |  |
| --- | --- |
| **Modes** | **Description** |
| **<r>** | It opens a file in read-only mode while the file offset stays at the root. |
| **<rb>** | It opens a file in (binary + read-only) modes. And the offset remains at the root level. |
| **<r+>** | It opens the file in both (read + write) modes while the file offset is again at the root level. |
| **<rb+>** | It opens the file in (read + write + binary) modes. The file offset is again at the root level. |
| **<w>** | It allows write-level access to a file. If the file already exists, then it’ll get overwritten. It’ll create a new file if the same doesn’t exist. |
| **Modes** | **Description** |
| **<wb>** | Use it to open a file for writing in binary format. Same behavior as for write-only mode. |
| **<w+>** | It opens a file in both (read + write) modes. Same behavior as for write-only mode. |
| **<wb+>** | It opens a file in (read + write + binary) modes. Same behavior as for write-only mode. |
| **<a>** | It opens the file in append mode. The offset goes to the end of the file. If the file doesn’t exist, then it gets created. |
| **<ab>** | It opens a file in (append + binary) modes. Same behavior as for append mode. |
| **<a+>** | It opens a file in (append + read) modes. Same behavior as for append mode. |
| **<ab+>** | It opens a file in (append + read + binary) modes. Same behavior as for append mode. |

For more information on file attributes, please run down through the below table.

|  |  |
| --- | --- |
| **Attributes** | **Description** |
| **<file.closed>** | For a closed file, it returns true whereas false otherwise. |
| **<file.mode>** | It returns the access mode used to open a file. |
| **<file.name>** | Returns the name of a file |
| **<file.softspace>** | It returns a boolean to suggest if a space char will get added before printing another value in the output of a <print> command. |

**A typical examples of opening a file for writing to binary**

#### #Open a file in write and binary mode.

fileobject = open("app.log", "wb")

#Display file name.

print ("File name: ", fob.name)

#Display state of the file.

print ("File state: ", fob.closed)

#Print the opening mode.

print ("Opening mode: ", fob.mode)

#Output the softspace value.

print ("Softspace flag: ", fob.softspace)

Below is the display of the above.

Last login: Tue Sep 8 13:05:44 on console

gndede@M4519 ~ % python3

Python 3.7.7 (default, Mar 10 2020, 15:43:33)

[Clang 11.0.0 (clang-1100.0.33.17)] on darwin

Type "help", "copyright", "credits" or "license" for more information.

>>> fob = open("app.log", "wb")

>>> print ("File name: ", fob.name)

File name: app.log

>>> print ("File state: ", fob.closed)

File state: False

>>> print ("Opening mode: ", fob.mode)

Opening mode: wb

>>> print ("Softspace flag: ", fob.softspace)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: '\_io.BufferedWriter' object has no attribute 'softspace'

>>>

In Python 3.x, there is a clear difference between strings (text) and a byte (8-bits). It states that the char ‘a’ doesn’t represent the ASCII value 97 until we specify it like that.

So, while we want to use a file in text mode, we better mention the correct encoding type beforehand.

Also, Python stores a file in the form of bytes on the disk, so you need to decode them in strings before reading. Similarly, encode them while writing texts to the file.

For a note, Python enables platform-dependent encoding by default. Hence, if we don’t change it, then it’s set to <cp1252> for Windows and <utf-8> for Linux.

Thus, the documentation says to quote the desired encoding while opening a file in Python. See the Python code snippet.

**fileObject = open('app.log', mode = 'r', encoding = 'utf-8')**

For note, you should import the <io> module in Python 2.x to enable the encoding feature. Python 3.x does it implicitly.

### **Close a file in Python**

It’s always best practice to close a file when work gets finished. However, Python runs a garbage collector to clean up the unused objects. But we must exclusively do the closing of a file instead of leaving it for the Python interpreter to do that..

#### **The close() file method**

Python provides the **<close()>** method to close a file.

While closing a file, the system frees up all resources allocated to it. That is rather easier to achieve.

##### 

##### 

##### 

##### **Close operation in Python**

**The most basic way is to call the Python close() method.**

**fileObject = open("app.log", encoding = 'utf-8')**

**# do file operations.**

**fileobject.close()**

Or we can close it gracefully using try-catch using an exception while performing other operations on the file. In this way, the code exits without closing the file. As a result, it would be better to put code inside a **<try-finally>** block**.**

**try:**

**fileobject = open('app.log', encoding = 'utf-8')**

**# do file operations.**

**finally:**

**fileobject.close()**

So, even if there comes an exception, the above code will make sure our file gets appropriately closed.

Another way to close a file is by using the **WITH** clause. It ensures that the file gets closed when the block inside the **WITH** clause executes. The beauty of this method is that it doesn’t require to call the close() method explicitly.

**with open('app.log', encoding = 'utf-8') as file:**

**#do any file operation.**

**Example:**

>>> with open('/content/presidentsofUsa.txt', 'r') as reader:

>>> # Read & print the entire file

>>> print(reader.read())

**George Washington**

**Abraham Lincoln**

**John F. Kennedy**

**Gerald Ford**

**Lyndon Johnson**

**Jimmy Carter**

**George W. Bush**

**Bill Clinton**

**Barack Obama**

**Joe Biden**

## Syntax

To open a file for reading it is enough to specify the name of the file:

File = open(“testfile.txt”) or file = open(“testfile.txt”, “rt”)

Because “r” is for read, and “t” for text are the default values, we don’t need to specify them.

**Note:** Before even we attempt to read a file, we must make sure the file exists, otherwise we may encounter an error while trying to read it.

## 

## 

## Open a File on the Server

Assume we have the following file, located in the same folder as Python:

File name: testfile.txt

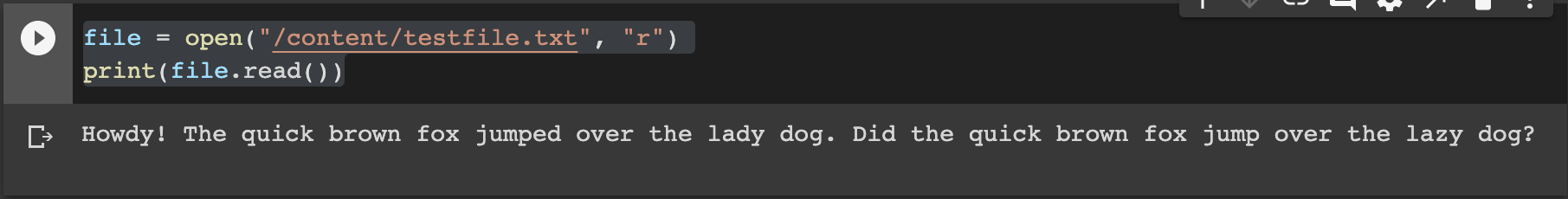
Howdy! The quick brown fox jumped over the lady dog. Did the quick brown fox jump over the lazy dog?

Let’s open the file using the built-in open() function. The open() function returns file object, which has a read() method for reading the content of the file:

### Example

file = open("/content/testfile.txt", "r")

print(file.read())

If the file is located in a different location, you will have to specify the file path, like this:

### Example

Open a file on a different location:

file = open("/content/testfile.txt", "r")

print(file.read())

## Read Only Parts of the File

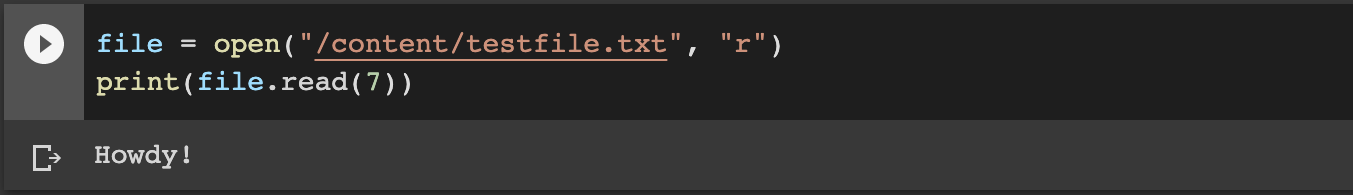
By default the read() method returns the whole text, but you can also specify how many characters you want to return:

### Example

Return the 7 first characters of the file:

file = open("/content/testfile.txt", "r")

print(file.read(7))



## Read Lines

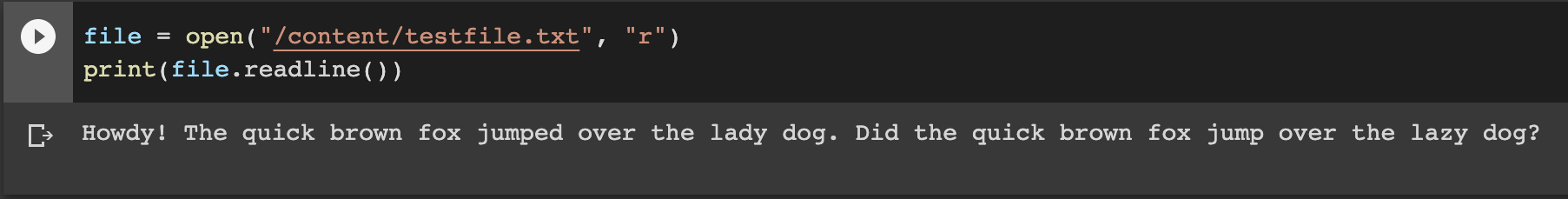
We can also return one line by using the readline() method:

### Example

Read one line of the file:

file = open("/content/testfile.txt", "r")

print(file.readline())



By calling readline() two times, you can read the two first lines:

### 

### 

### 

### 

### 

### 

### 

### 

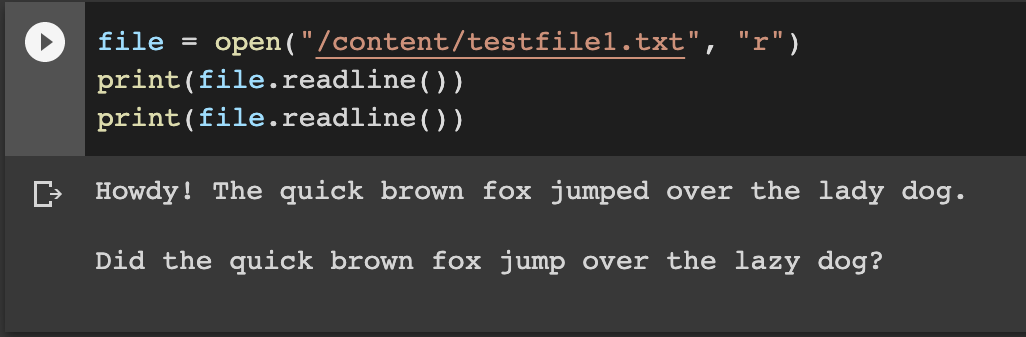
### Example

**Read two lines of the file:**

**file = open("/content/testfile1.txt", "r")**

**print(file.readline())**

**print(file.readline())**

****

By looping through the lines of the file, you can read the whole file, line by line:

### We can use the readline() method to read individual lines of a file. This method reads a file till the newline, including the newline character.

Lastly, the readlines() method returns a list of remaining lines of the entire file. All these reading methods return empty values when the end of file (EOF) is reached.

### 

### 

### 

### 

### 

### 

### 

### 

### 

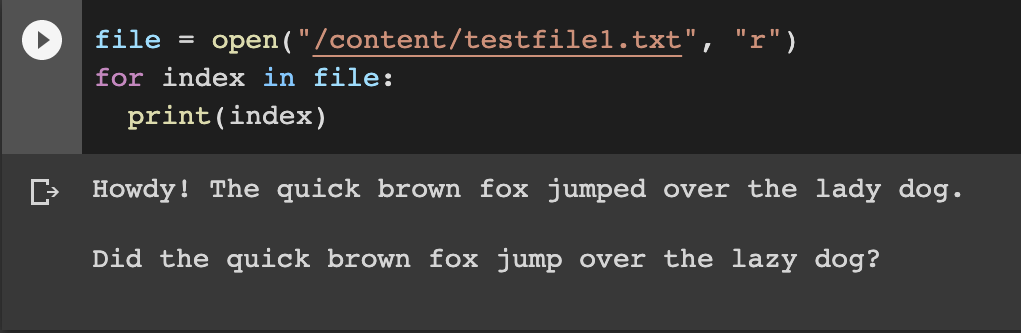
### **Example**

Loop through the file line by line:

file = open("/content/testfile1.txt", "r")

for index in file:

print(index)



## Close Files

It is a good practice to always close the file when we are done with it.

### Example

Close the file when you are finish with it:

file = open("/content/testfile.txt", "r")

print(file.readline())

file.close()

**Note:** You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

## Write to an Existing File

To write to an existing file, we must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

### Example

Open the file "demofile2.txt" and append content to the file:

file = open("/content/testfile2.txt", "a")

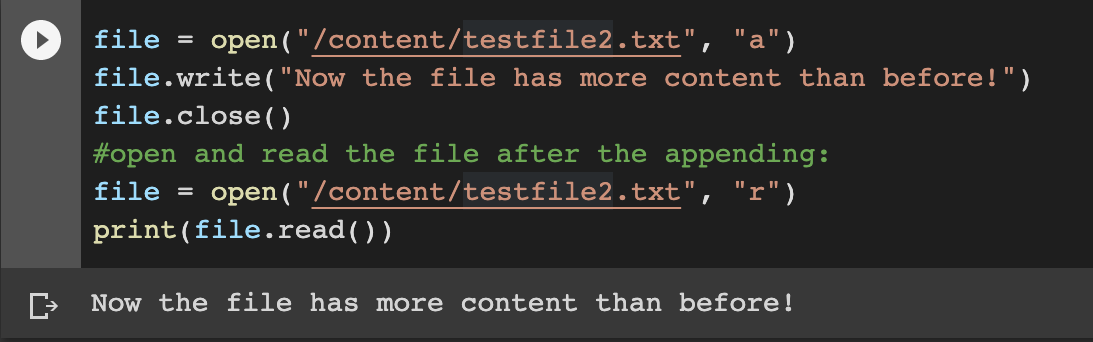
file.write("Now the file has more content than before!")

file.close()

#open and read the file after the appending:

file = open("/content/testfile2.txt", "r")

print(file.read())



### Example

Open the file "testfile3.txt" and overwrite the content:

file = open("/content/textfile3.txt", "w")

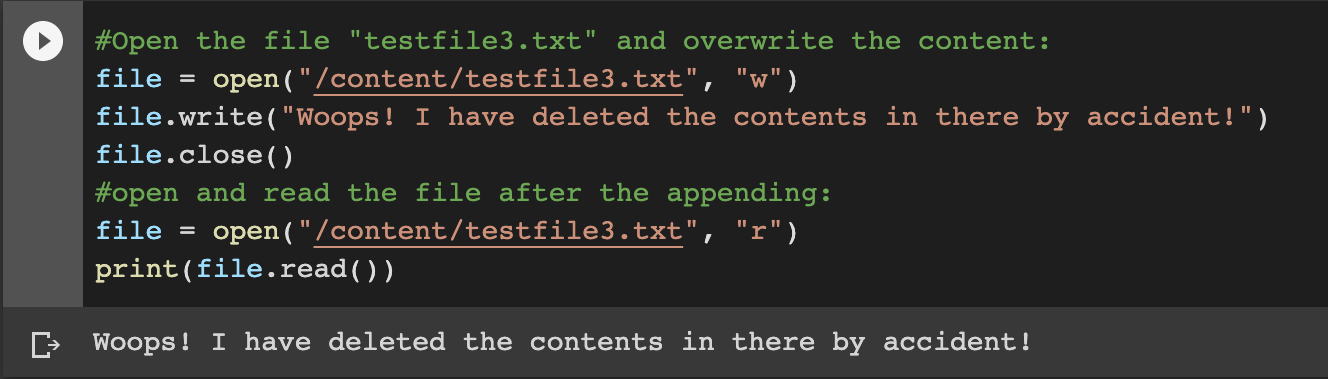
file.write("Woops! I have deleted the contents in there by accident!")

file.close()

#open and read the file after the appending:

file = open("/content/textfile3.txt", "r")

print(file.read())



**Note:** the "w" method will overwrite the entire file.

## Create a New File

To create a new file in Python, we use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

Example

Create a file called "myfile.txt":

file = open("/content/mytextfile.txt", "x")

Result: a new empty file is created!

### 

### 

### Example

Create a new file if it does not exist:

f = open("/content/mytextfile.txt", "w")

## Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

### Example

Remove the file "testfile.txt":

import os

os.remove("/content/testfile.txt")

## Let’s check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

### 

### 

### 

### 

### 

### 

### 

### Example

Check if file exists, *then* delete it:

import os

if os.path.exists("/content/testfile.txt"):

os.remove("/content/testfilefile.txt")

else:

print("The file does not exist")



## Delete Folder

To delete an entire folder, use the os.rmdir() method:

### Example

Remove the folder "myfolder":

import os

os.rmdir("/content/myfolder")

**Note:** We can only remove *empty* folders.

### **Set File offset in Python**

## Description

#### The method tell() returns the current position of the file read/write pointer within the file.

#### **Tell() Method**

This method gives you the current offset of the file pointer in a file.

**Syntax:**

**Following is the syntax for tell() method −**

file.tell()

The tell() method doesn’t require any argument.

## Parameters

NA

## 

## 

## 

## 

## 

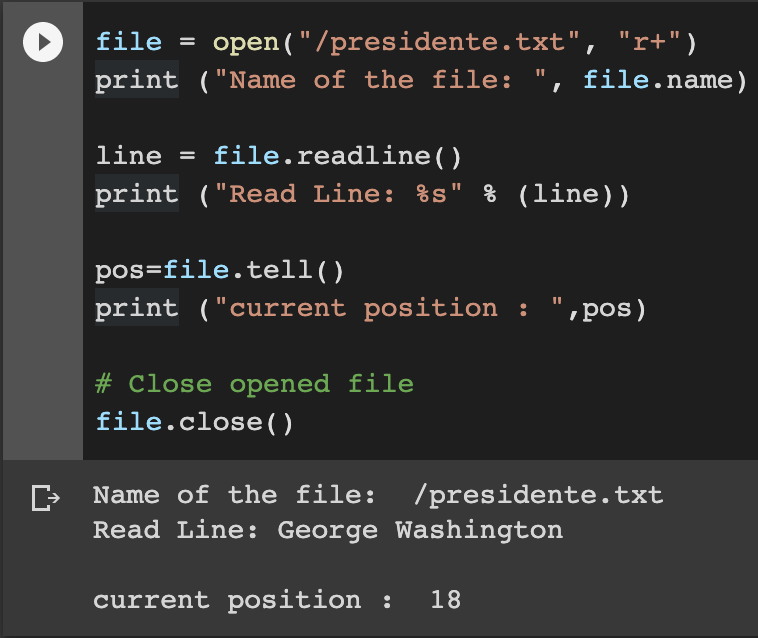
## 

## 

## 

## Return Value

This method returns the current position of the file read/write pointer within the file.



## Result

#### **When we run the above program, it produces the following result −**

**Name of the file: /presidente.txt**

**Read Line: George Washington**

**current position : 18**

#### 

### 

### **Example**

#### **Find the current file position:**

#### **file = open("testfile.txt", "r")**

#### **print(file.tell())**

## **Definition and Usage**

#### The tell() method returns the current file position in a file stream.

#### Tip: You can change the current file position with the [seek()](https://www.w3schools.com/python/ref_file_tell.asp) method.

### Example

#### **Return the current file position after reading the first line:**

#### **file = open("testfile.txt", "r")**

#### **print(file.readline())**

#### **print(file.tell())**

#### 

#### 

#### **Seek() Method**

**Python** file **method seek()** sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means **seek** relative to the current position and 2 means **seek** relative to the file's end.

### Example

Change the current file position to 4, and return the rest of the line:

file = open("testfile.txt", "r")

file.seek(4)

print(file.readline())

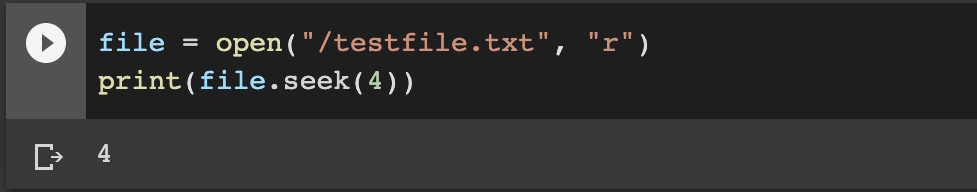
## 

### Example

Return the new position:

f = open("demofile.txt", "r")

print(f.seek(4))



## Definition and Usage

The seek() method sets the current file position in a file stream.

The seek() method also returns the new postion.

This method can help us change the position of a file pointer in a file.

file = open("testfile.txt", "r+")

print ("Name of the file: ", file.name)

line = file.readline()

print ("Read Line: %s" % (line))

pos=file.tell()

print ("current position : ",pos)

# Close opened file

file.close()

## **Python File Methods**

There are various methods available with the file object. Some of them have been used in the above examples.

Here is the complete list of methods in text mode with a brief description:

|  |  |
| --- | --- |
| Method | Description |
| **close()** | Closes an opened file. It has no effect if the file is already closed. |
| **detach()** | Separates the underlying binary buffer from the TextIOBase and returns it. |
| **fileno()** | Returns an integer number (file descriptor) of the file. |
| **flush()** | Flushes the write buffer of the file stream. |
| **isatty()** | Returns True if the file stream is interactive. |
| **read(n)** | Reads at most n characters from the file. Reads till end of file if it is negative or None. |
| **readable()** | Returns True if the file stream can be read from. |
| **readline(n=-1)** | Reads and returns a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset, from = SEE\_SET | Changes the file position to offset bytes, in reference to from (start, current, end). |
| seekable() | Returns True if the file stream supports random access. |
| tell() | Returns the current file location. |
| truncate(size=None) | Resizes the file stream to size bytes. If size is not specified, resizes to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Writes the string s to the file and returns the number of characters written. |
| writelines(lines) | Writes a list of lines to the file. |

**Pickle Module in Python**

The purpose of pickling **is** to translate data into a format that can be transferred from RAM to disk. The pickle module implements binary protocols for serializing and de-serializing a Python object structure. *“Pickling”* is the process whereby a Python object hierarchy is converted into a byte stream, and *“unpickling”* is the inverse operation, whereby a byte stream (from a binary file or bytes-like object) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” 1 or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

**Warning:** The pickle module is not secure. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

The [**pickle**](https://docs.python.org/2/library/pickle.html#module-pickle) module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy.

### **What Can You Do With Pickle?**

Pickling is useful for applications where you need some degree of persistence in your data. Your program's state data can be saved to disk, so you can continue working on it later on. It can also be used to send data over a Transmission Control Protocol (TCP) or socket connection, or to store python objects in a database. Pickle is very useful for when you're working with machine learning algorithms, where you want to save them to be able to make new predictions at a later time, without having to rewrite everything or train the model all over again.

### **When Not To Use pickle**

If you want to use data across different programming languages, pickle is not recommended. Its protocol is specific to Python, thus, cross-language compatibility is not guaranteed. The same holds for different versions of Python itself. Unpickling a file that was pickled in a different version of Python may not always work properly, so you have to make sure that you're using the same version and perform an update if necessary. You should also try not to unpickle data from an untrusted source. Malicious code inside the file might be executed upon unpickling.

## **Storing data with pickle**

### **What can be pickled?**

You can pickle objects with the following data types:

* Booleans,
* Integers,
* Floats,
* Complex numbers,
* (normal and Unicode) Strings,
* Tuples,
* Lists,
* Sets, and
* Dictionaries that contain picklable objects.

All the above can be pickled, but you can also do the same for classes and functions, for example, if they are defined at the top level of a module.

Not everything can be pickled (easily), though: examples of this are **generators**, **inner classes**, **lambda functions** and **defaultdicts**. In the case of lambda functions, you need to use an additional package named **dill**. With default dicts, you need to create them with a module-level function.

### **Pickle vs JSON**

JSON stands for JavaScript Object Notation. It's a lightweight format for data-interchange, that is easily readable by humans. Although it was derived from JavaScript, JSON is standardized and language-independent. This is a serious advantage over pickle. It's also more secure and much faster than pickle.

However, if you only need to use Python, then the pickle module is still a good choice for its ease of use and ability to reconstruct complete Python objects.

An alternative is cPickle. It is nearly identical to pickle, but written in C, which makes it up to 1000 times faster. For small files, however, you won't notice the difference in speed. Both produce the same data streams, which means that Pickle and cPickle can use the same files.

### **Pickling files**

To use pickle, start by importing it in Python.

import pickle

For this tutorial, you will be pickling a simple dictionary. A dictionary is a list of

key : value elements. You will save it to a file and then load again. Declare the dictionary as such:

**pets\_dict = { 'Bob': 3, 'Jimmy': 2, 'Laika': 3, 'Jimmy': 10, 'Jack': 3, 'Stella': 3, 'Nzinga': 7 }**

To pickle this dictionary, you first need to specify the name of the file you will write it to, which is pets in this case.

**Note** that the file does not have an extension.

To open the file for writing, simply we use the open() function. The first argument should be the name of our file. The second argument is '**wb**'. The **w** means that we'll be writing to the file, and **b** refers to binary mode. This means that the data will be written in the form of byte objects.

If we forget the **b**, a TypeError: must be str, not bytes will be returned. We may sometimes come across a slightly different notation; **w+b**, but don't worry, it provides the same functionality.

**filename = 'pets'**

**outfile = open(filename, 'wb')**

Once the file is opened for writing, we can use pickle.dump(), which takes two arguments: the object we want to pickle and the file to which the object has to be saved. In this case, the former will be pets\_dict, while the latter will be outfile.

Don't forget to close the file with close()!

**pickle.dump(pets\_dict, outfile)**

**outfile.close()**

Now, a new file named pets should have appeared in the same directory as your Python script (unless we specified a file path as file name).

### **Unpickling files**

The process of loading a pickled file back into a Python program is similar to the one we saw previously: using the open() function again, but this time with '**rb**' as second argument (instead of **wb**). The **r** stands for read mode and the **b** stands for binary mode.

We'll be reading a binary file. Assign this to infile. Next, we use pickle.load(), with infile as argument, and assign it to new\_dict. The contents of the file are now assigned to this new variable. Again, we'll need to close the file at the end of the process.

**infile = open(filename,'rb')**

**new\_dict = pickle.load(infile)**

**infile.close()**

To make sure that you successfully unpickled it, we can print the dictionary, compare it to the previous dictionary and check its type with type().

**print(new\_dict)**

**print(new\_dict==pets\_dict)**

**print(type(new\_dict))**

**pets\_dict = { 'Bob': 3, 'Jimmy': 2, 'Laika': 3, 'Jimmy': 10, 'Jack': 3, 'Stella': 3, 'Nzinga': 7 }**

**True**

**<class 'dict'>**

### **Compressing pickle files**

If we’re saving a large dataset and our pickled file takes up a lot of space, we may want to compress it. This can be done using **bzip2** or **gzip**. They both compress files, but **bzip2** is a bit slower. **gzip**, however, produces files about twice as large as **bzip2**. We'll be using bzip2 in this lesson.

Remember that compression and serialization is not the same! We can go back to the beginning of this lesson to refresh our memory about that.

Start by importing bzip2 with import bz2. Importing pickle is done the same way as in the beginning of this tutorial.

**import bz2**

**import pickle**

**serialized\_file = bz2.BZ2File('smallerfile', 'w')**

**pickle.dump(pets\_dict, serialized\_file)**

A new file named smallerfile should have appeared. Keep in mind that the difference in file size compared to an uncompressed version will not be noticeable with small object structures.



### **Unpickling Python 2 objects in Python 3**

We may sometimes come across objects that were pickled in Python 2 while running Python 3. This can be a hassle to unpickle.

We could either unpickle it by running Python 2, or do it in Python 3 with encoding='latin1' in the load() function.

**infile = open(filename,'rb')**

**new\_dict = pickle.load(infile, encoding='latin1')**

This will not work if your objects contain NumPy arrays. In that case, you could also try using encoding='bytes':

**infile = open(filename,'rb')**

**new\_dict = pickle.load(infile, encoding='bytes')**

First, import pickle to use it, then we define an example dictionary, which is a Python object.

Next, we open a file (note that we open to write bytes in Python 3+), then we use pickle.dump() to put the dict into the opened file, then close.

**import pickle**

**pets\_dict = { 'Bob': 3, 'Jimmy': 2, 'Laika': 3, 'Tobi': 10, 'Jack': 3, 'Stella': 3, 'Nzinga': 7 }**

#Below code will save the pickle file for us, now we need to cover how to access the pickled file:

**pickle\_output = open("dict.pickle","wb")**

**pickle.dump(pets\_dict, pickle\_output)**

**pickle\_output.close()**

#Open the pickle file

#Use pickle.load() to load it to a var.

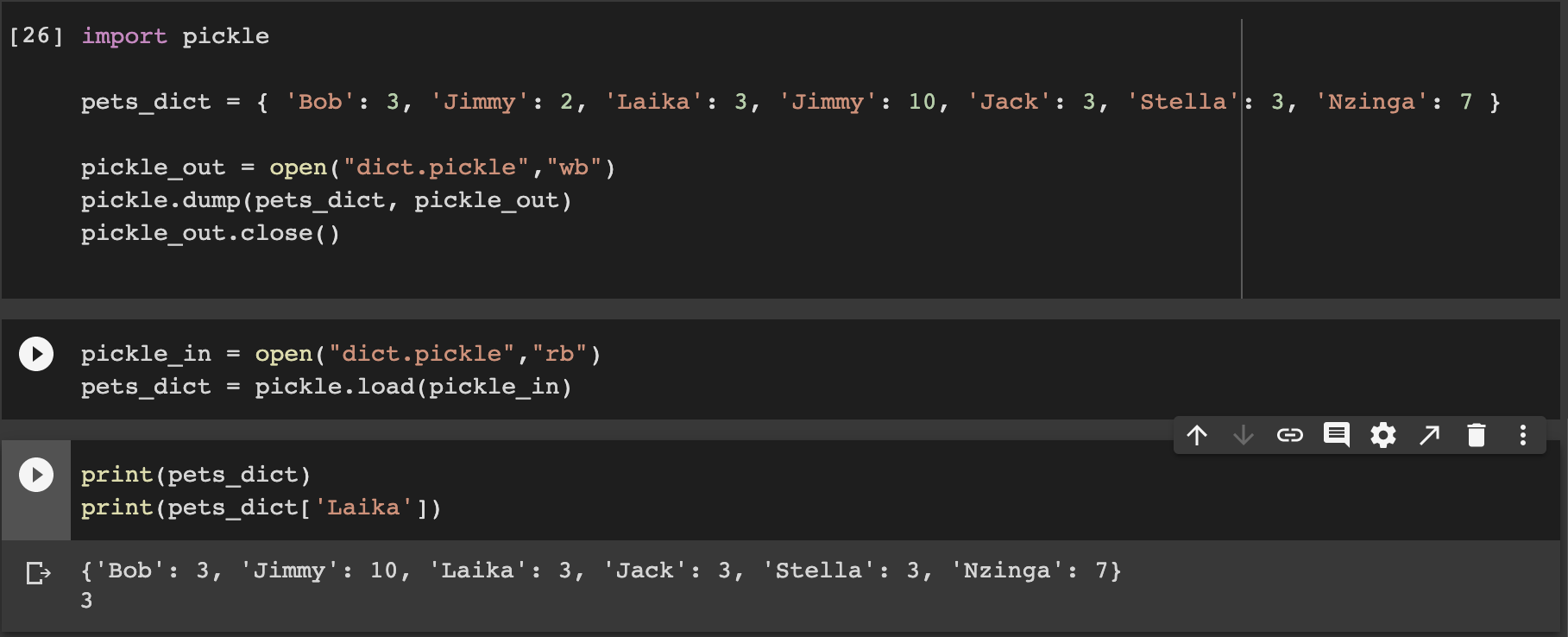
**pickle\_input = open("dict.pickle","rb")**

**example\_dict = pickle.load(pickle\_in)**

#This shows that we've retained the dict data-type.

**print(example\_dict)**

**print(example\_dict[‘Laika’])**

****

**Conclusion**

Congratulations! We're now ready to start file processing using Python. We'll be able to open files for reading, append, delete or for writing and also for display.

We're now ready to start pickling and unpickling files with Python. We'll be able to save your machine learning models and resume work on them later on. If we would like to learn more on how to build such predictive models in Python.