

# **TITLE:** Python Programming: Problem Solving, Packages and Libraries

## **Edition**

---

Lecture PPT Chapter 15: File operations  
in Python

# Learning Objectives - File Operations in Python

## LEARNING OBJECTIVES

*After studying this chapter, you will be able to:*

- LO 1** Differentiate between a binary file and a text file and also know about file objects, their attributes and methods.
- LO 2** Know about the basic file operations: opening, reading/writing and closing.
- LO 3** Use the common methods in reading/writing files.
- LO 4** Appreciate the need for “automatic file closure” using the “with” statement.
- LO 5** Use file “pickling/serializing” for “persistence”.

# Basics of a file (text versus binary file)

**The characteristics of a “file” can be given as follows:**

- **It is a location on disk to store information and has a name.**
- **Files in Python can be broadly categorized into “text” and “binary” files.**
- **A text file is a sequence of lines. The end of each line is indicated by the “newline” or End of Line (EOL) “escape sequence”, i.e., “\n”. Each line in turn consists of characters.**
- **On the other hand, a “binary” file is a file “other than” a text file. Binary files can generally be interpreted by an application which “understands” its file structure.**
- **To read a file, you need to open it first, then perform operations on it (read or write) and then close it.**

# A short note on opening a file

**Note on file opening:** To open a file, you need to create a file object or a handle to a file. In Python, whether you want to read, write or append to a file you must create a file object, by using the in-built `open()` function. Once you have created a file object (or a file handle) using the `open` function then you can get two types of options or operations on this file object:

- The first type is file object **methods** which you can use on the file object or file handle using the dot operator and the method name. Note the method name will always have two brackets, i.e., `()`.
- The second type is file object **attributes** which are also accessed using the dot operator but without the brackets.

Suppose you create a file object using `open` method and call this file object `fileObj`. Now you can use the `read()` method on this file object by using the syntax `fileObj.read()`. Similarly you can get the attributes of a file object by using the various file object attributes. For example, each file object has a `name` attribute. Hence, you can get this `name` attribute using `fileObj.name`

# Opening and closing files -- 1

**The syntax of the open() function is as follows:**

```
1 file_object = open(file_name [, access_mode][, buffering])
```

**Note:** Square brackets indicate optional parameters to the function. So the open method has three parameters out of which the first, i.e., file\_name is mandatory, while the other two have default values and are therefore optional.

The built-in function open() can be used to open a file. The return value of this function is a “file object” sometimes also called “file handle”. This file object can then be used to perform various operations on the file.

There are three arguments which may be given to the open() function, out of which first is compulsory and other two are optional. They are:

- 1. Argument file\_name: (must be given):** The argument file\_name must be given. It is form of a Python string giving the name of the file to be opened. It may include the “path” to the file also. The path can be relative or absolute. The relative path is from the current working directory (cwd), i.e., the directory you are present in.

**[Continued on next slide]**

# Opening and closing files -- 2

[Continued from previous slide]

2. **Argument `access_mode`: (optional parameter):** The second argument is `access_mode`. This argument determines the “mode” in which the file has to be opened. This is also a Python string. Some common “modes” are as follows:
  1. 'r' if the file is to be opened “read only”. This is the default/ optional mode. If the second parameter is omitted, the interpreter presumes that it is 'r' and opens the file for reading only.
  2. 'w' if the file is to be opened for “writing”. However, if a file with same name already exists, then it will be deleted.
  3. 'a' if the data is to be “appended”, i.e., the data is “added to the end”.
  4. Note: Using 'r+' will open the file for both reading and writing.
3. **Buffering: (optional Parameter):** This parameter can largely be ignored.

# Text mode versus binary mode

- **Text mode:** When a file is opened in *text mode*, the contents are automatically decoded and the file contents are returned as a string, i.e., `str`.
- **Binary files:** A file can be opened (using the `open()` method) in *binary mode* by adding a “b” (lowercase only) to the argument for the mode. (For example, if file is to be in read mode binary, then argument will be ‘rb’). When you read a file in “binary mode”, you read the data “as it is” without any transformations. For example, in text mode the sequence `\n` would be read not as characters, but as a “new line”. However, in binary mode `\n` will be read as `\n` and nothing else.

# Some common “modes” of opening a file in Python

Modes	Description
<b>r</b>	The file is opened for reading only. It is the default mode.
<b>rb</b>	The file is opened for reading but in “binary” mode.
<b>r+</b>	The file is opened for both reading and writing.
<b>rb+</b>	The file is opened for both reading and writing. In “binary” mode.
<b>w</b>	The file is opened for writing only. If a file by same name already exists, then it is overwritten without warning. However, if a file by the given name does not exist, then it is created for writing.
<b>wb</b>	The file is opened for writing only but in binary mode. If a file by same name already exists, then it is overwritten without warning. However, if a file by the given name does not exist, then it is created for writing.
<b>w+</b>	It opens a file for both reading and writing. If a file by same name already exists, then it is overwritten without warning. However, if a file by the given name does not exist, then it is created for writing.
<b>wb+</b>	It opens a file for both reading and writing but in “binary” mode. If a file by same name already exists, then it is overwritten without warning. However, if a file by the given name does not exist, then it is created for writing.
<b>a</b>	It opens a file for appending. If a file by given name exists, then the “file pointer” points to the end of file. However, if no file of given name exists, then a new empty file of given name is created.
<b>ab</b>	It opens a file for appending but in binary format. If a file by given name exists, then the “file pointer” points to the end of file. However, if no file of given name exists, then a new empty file of given name is created.
<b>a+</b>	It opens a file for both reading as well as appending. If a file by given name exists, then the “file pointer” points to the end of file. However, if no file of given name exists, then a new empty file of given name is created and you can both read as well as write in this newly created file.
<b>ab+</b>	It opens a file for both reading as well as appending but in binary mode. If a file by given name exists, then the “file pointer” points to the end of file. However, if no file of given name exists, then a new empty file of given name is created and you can both read as well as write in this newly created file.



# The file object attributes

After opening a file using `open()`, you get a “file object”. The file object has a number of “attributes” which can be accessed using the dot (.) notation. Table 15.2 gives some common “file attributes”:.(assuming `fHandle` is a “file object” or a “file handle”).

Attribute	Description
<code>fHandle.closed</code>	Gives a value of True if file is closed, Else returns False.
<code>fHandle.mode</code>	Gives the “access mode” in which the file was opened.
<code>fHandle.name</code>	Gives name of the file.

# Reading and writing a file

The common methods used in reading/ writing files are as follows:

- `read()`,
- `readline()`,
- `readlines()`,
- `write()`,
- Methods for getting file positions, i.e., `seek()` and `tell()`

[These methods are discussed in the following slides]

# The read() and tell() methods -- 1

- Once a file has been opened using the open() method, then the read() method can be used to “read” it.
- Note that the file must have been opened in a mode which “permits” reading.
- For example, the “w” mode does not permit reading, so you cannot use the read() method on a file object which has been opened in “w” mode.
- When you read a file using the read() method, there are two important things, namely:
  - The place from where to start reading.
  - How much to read.

**[-- continued]**

# The read() and tell() methods -- 2

The syntax of the read() method is as follows:

```
1 fileObject.read([count]) #count is optional
```

- The method read() has one “optional” parameter, i.e., “count”.
- If you don’t give any “count” parameter, then the read() function tries to “read as much as possible”.
- But “how much” data a read() method can read also depends upon the “location of the cursor”, i.e., the point at which the “cursor” is.
- Note that the “current position” within the file is sometimes called “cursor”.
- So if the “cursor” is at the beginning of file, the read() method (without any count parameter) will try to read the “complete file”, i.e., till the end of the file.
- But if the “cursor” is at the end of the file, then the read() method will return an “empty string”.
- However, if the “count” parameter is given, then the read(count) will try to read that many bytes of data.

# The read() and tell() methods -- 3

The fHandle or file object has a method called `fObj.tell()`. Using this method, you can know the current position of the cursor. The “current position” in many texts is called “cursor” or “pointer”. The format of `tell()` method of `fObj` is as follows:

```
1 fObj.tell() → integer #The (→ integer) indicates that return value
2 is an integer
```

Note that the `fObj.tell()` method returns an integer

# seek(offset[,from\_what]) method

- As pointed out earlier, a “file object” has a method `fobj.tell()` which gives the “current position” of the “cursor” or “pointer”. The `fobj`, i.e., the Python file object also provides a method `seek()` to “change” the location of the cursor to a desired location. Some important aspects of the `seek(offset[, from])` method are as follows:
- It has two parameters: “`offset`” and “`from_what`”. The “`offset`” parameter is mandatory, while the “`from_what`” is optional.
- 1. The “`offset`” parameter indicates the “number of bytes” by which the cursor is to be moved.

[.. Continued to next slide]

**[Continued from previous slide ....]**

2. The “from\_what” parameter is the “anchor” or the point from where the count of “offset” begins.
- By default **from\_what** is 0 meaning that by default the count of “offset” is done from “start” of the opened file.
  - Similarly a value of 1 indicates that the “offset” is to be done from the “current position of the cursor”.
  - A value of 2 indicates that the count of “offset” is to be done from the “end” of the file.
  - These three values are also stored in following constants of the os module:
    - (1) **os.SEEK\_SET** is equivalent to the value of **from\_what = 0**;
    - (2) **os.SEEK\_CUR** is equivalent to **from\_what = 1**; and
    - (3) **os.SEEK\_END** is equivalent to **from\_what = 2**.

# readline([size]) and readlines([sizehint]) -- 1

- The readline([size]) method has one optional parameter “size”.
- If the “size” parameter is absent, the method returns 1 line of text in the form of a string.
- The newline, i.e., \n at the end of the line is also returned as part of the string returned by the method.
- If the optional “size” is present, then the parameter “size” indicates the “number of bytes” to be returned.
- When End of File (EOF) is reached, readline() will return an empty string.
- Note that since readline() reads only 1 line at a time, it is “memory efficient”.
- The syntax for the readlines() is

```
1 fileObject.readlines( [sizehint] );# Square brackets indicate  
2 optional parameter
```



# readline([size]) and readlines([sizehint]) -- 2

The syntax for the readlines() is

```
1 fileObject.readlines( [sizehint] );# Square brackets indicate
2 optional parameter
```

- The method **readlines()** reads all the lines in one go until the EOF is reached.
- Another important difference is that the **readlines()** method does not return a single string but rather a “list of strings”.
- So each line in the file forms one string in the list.
- For example, if a file has say 5 lines, then **readlines()** will give a list of 5 strings.
- If the optional “**sizehint**” parameter is present, then instead of reading upto EOF, **readlines([sizehint])** reads approximately upto number of bytes indicated by the parameter “**sizehint**”.
- But in any case a line will be taken as a whole and not left “in between a line” because of sizehint.

# Writing to files using write() method

- For writing to a file, there are following two possible modes:
  1. "Append" mode. It adds data to the end of the file.
  2. "write" mode. It simply overwrite the file.
- In both the modes, if a file by the given name does not exist from before, then it is automatically created.
- Note that in write mode, the existing text is overwritten and in append mode the new text is added to the existing text in the file.
- Also note that the method for both writing and appending is write() but the modes are different.
- For writing, i.e., overwriting you use "w" and for appending, i.e., adding you use "a" mode.
- Note that in append mode, the addition is always done at the end of the file irrespective of where the file pointer might be.

# writelines(aList) method of file object

- A file object also has a writelines() method.
- The method writelines() writes a “sequence” of strings to the opened file.
- The sequence can be in form of an “iterable” object like a “list of strings”.
- The method writelines() also has a return value of None.

The syntax is as follows:

```
1 fileObject.writelines( someSequence )
```

Where **someSequence** is typically a list.

Note that newlines (i.e. \n) are not added

# Some more advanced concepts in file operations

Some additional advanced concepts discussed in the book are:

- Using the “with” statement to close file automatically
- Python IO stream object
- Pickling and unpickling
- Pickle module and steps in unpickling

# Using the “with” statement to close file automatically -- 1

- Python provides a method whereby you ensure that files are automatically closed after opening.
- Stream objects such as file objects (or file handles) have a method `close()` to explicitly close an opened file.
- In case a programmer does not explicitly close an opened file using `close()` function, then the garbage collector (gc) will eventually close it, but this might take some time.
- Till such time that the gc does close the opened file, the opened file will continue to reside in RAM and utilize resources.
- So one should always explicitly close an opened file using `close()` explicitly.

However, there may be another problem. Consider the following scenario:

- You open a file using `open()`.
- However, before you can close the file using `close()`, the script crashes due to some bug.
- Then you will not get a chance to explicitly close the file using `close()`.

# Using the “with” statement to close file automatically -- 2

- **There are two possible solutions for this:**
  - First: Enclose the file opening code in a try-finally block so that in case of an exception you can close the opened file. This method was followed on Python 2.x and can also be done in Python 3.x
  - However, Python 3.x provides another solution, i.e., using the “with” statement for opening and automatically closing a file.

**The following script shows the use of “with” statement to close a file automatically:**

1	path = 'C:\Python34\myScripts\derozio.txt'
2	with open(path, 'r+') as fObj:
3	for line in fObj:
4	print('*',line)
5	# OUTPUT
6	* My country! In thy days of glory past
7	* A beauteous halo circled round thy brow
8	* and worshipped as a deity thou wast—

# Python IO stream object

- The standard output stream (also called stdout) is where the text output of the program is sent by default.
- Whenever a script is started in Python, three data connections are created automatically: standard input, standard output and standard error.

# Pickling and unpickling

- **Pickling is also called “serialization” for “persistence”. In some text, it is also called “marshalling,” or “flattening”.**
- **But “pickling” is not to be confused with “compression”.**
- **Following issues are witnessed while storing a Python “object” on the hard disk:**
  - **During pickling, the Python interpreter when storing the object needs to “convert the object” into a “byte stream” which can then be “stored” in memory.**
  - **During “unpickling”, the Python interpreter does the reverse of pickling and reads the “byte stream” from the memory and “reconstructs the object” from this byte stream.**



# What does “Converting an object hierarchy to a byte stream” mean?

- In Python everything is an object and therefore it is representation of some class.
- “Objects” in Python have attributes and methods and the attributes have values.
- So when a Python object is to be stored in a file, it has to be first converted into a “stream of bytes”.
- The pickled byte stream which is stored on the hard disk, is used by the interpreter to re-construct the original object hierarchy by unpickling the stream.
- This process is somewhat to object serialization in other programming languages like Java, though the protocol followed is different.
- So what has been “pickled” in Python cannot be “unpickled” in Java.

# Python pickling versus JSON

- One of the most common used format for “data exchange” as well as “pickling” is JSON (JavaScript Object Notation).
- But JSON is different from Python pickling. The major differences are as follows:
  1. JSON is a format for “text serialization in unicode”. Python pickling is a “binary serialization format”.
  2. JSON being a “text serialization format” is “human readable”. Python pickling is not human readable.
  3. JSON is almost a “universal standard” and used / understood by many applications. Python pickling is very specific to Python and therefore not inter-operable.
  4. The limitation of JSON is that while it is easy to serialize “simple objects”, it may be difficult to serialize “objects with complex structures”. On the other hand, Python pickling can be used with almost any type of Python objects.

# Pickle module and steps in unpickling

Steps in unpickling are

- Import pickle module
- Open the file and create a handle to the file or a file object from which to unpickle. Open the file in 'rb' mode, i.e., “for reading in binary mode”.
- Use the method `pickle.load(input_file)`. This method of the pickle module reads the saved data in the form of a stream. Thereafter, it “re-constructs” the object from the data stream.

The following example script shows how “pickling” and “unpickling” is done

```
1 >>> fileObj = open('C:\Python34\myScripts\myPick.pkl', 'wb')
2 >>> import pickle
3 >>> myDict = {'a':1, 'b':2}
4 >>> pickle.dump(myDict, fileObj)
5 >>> fileObj.close()
6 >>> f = open('C:\Python34\myScripts\myPick.pkl', 'rb')
7 >>> myObj = pickle.load(f)
8 >>> myObj
9 {'a': 1, 'b': 2}
10 >>> f.seek(0,0)
11 0
12 >>> f.read()
13 b'\x80\x03}q\x00(X\x01\x00\x00\x00aq\x01K\x01X\x01\x00\x00\x00bq\x02K\x02u.'
14 >>>
```

# Some useful methods of the os module

Python has an “os” module for dealing with those “functionalities” which depend on the “operating system”. This

module also provides methods for “file processing” like renaming and deleting files.

Some important/ useful methods of the os module are as follows:

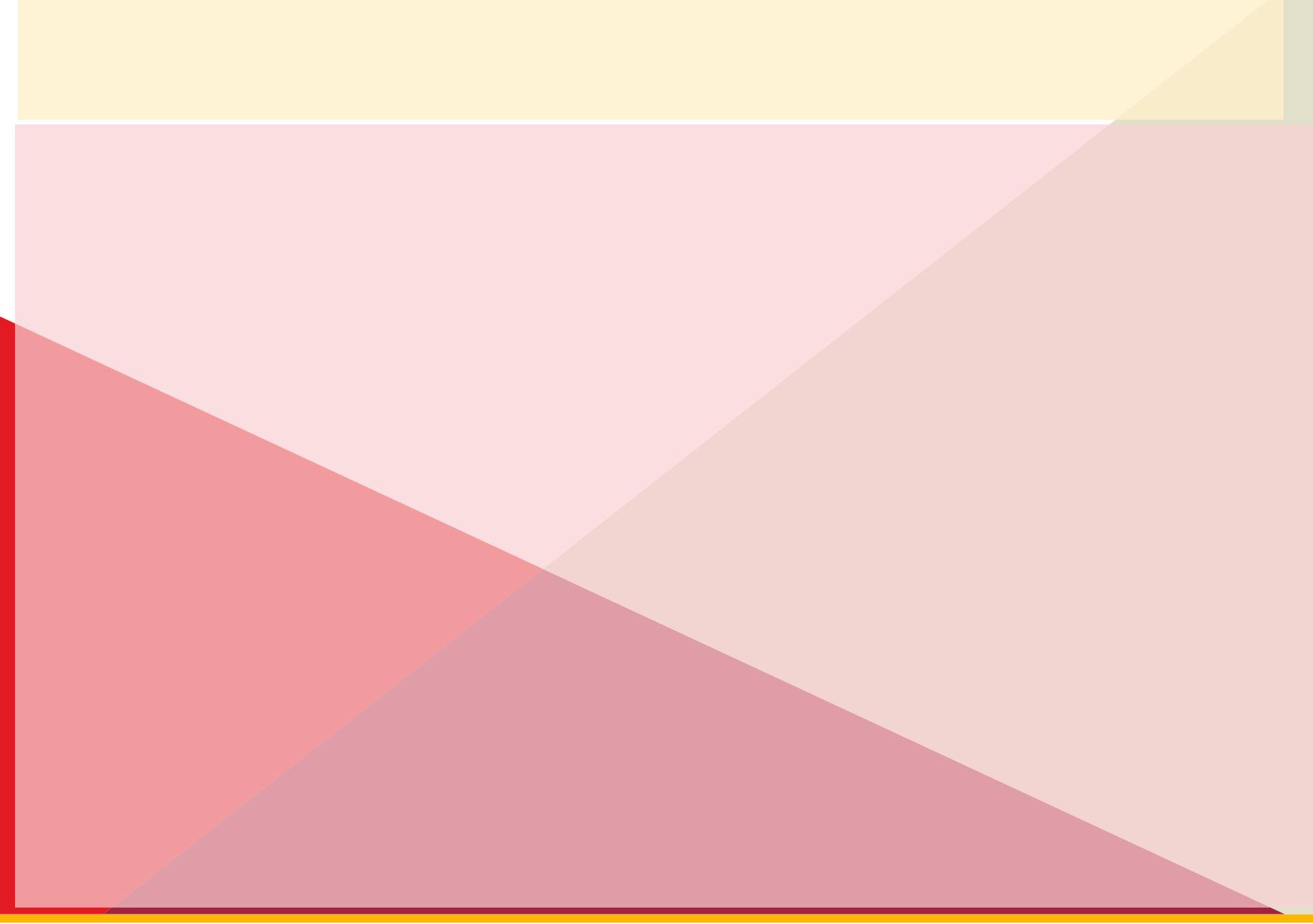
The `rename(current_filename, new_filename)` method:

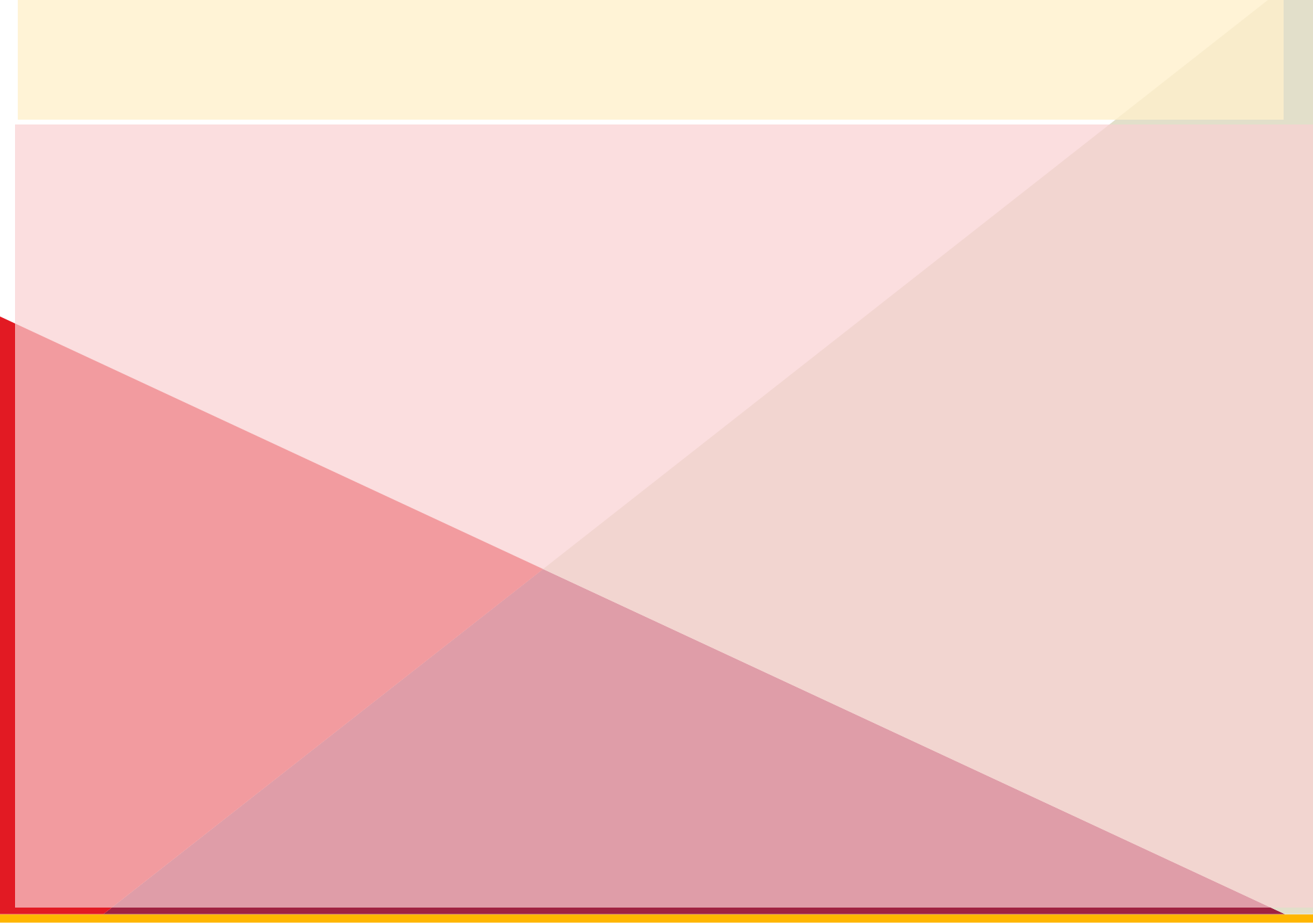
The `rename(current_filename, new_filename)` method takes two arguments, the `current_filename` and the `new_filename`. The syntax of the method is as follows:

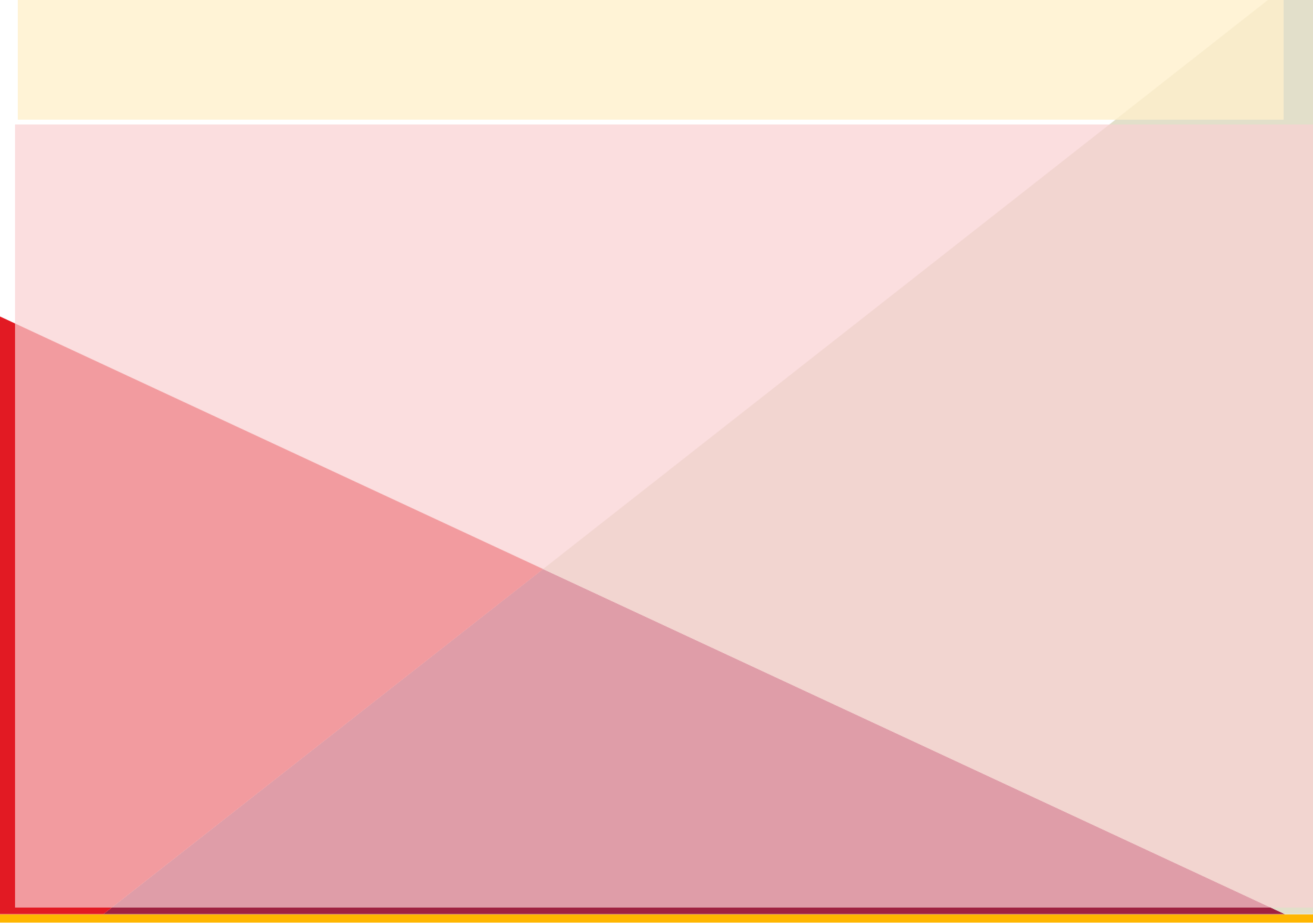
```
1 os.rename(current_file_name, new_file_name)
```

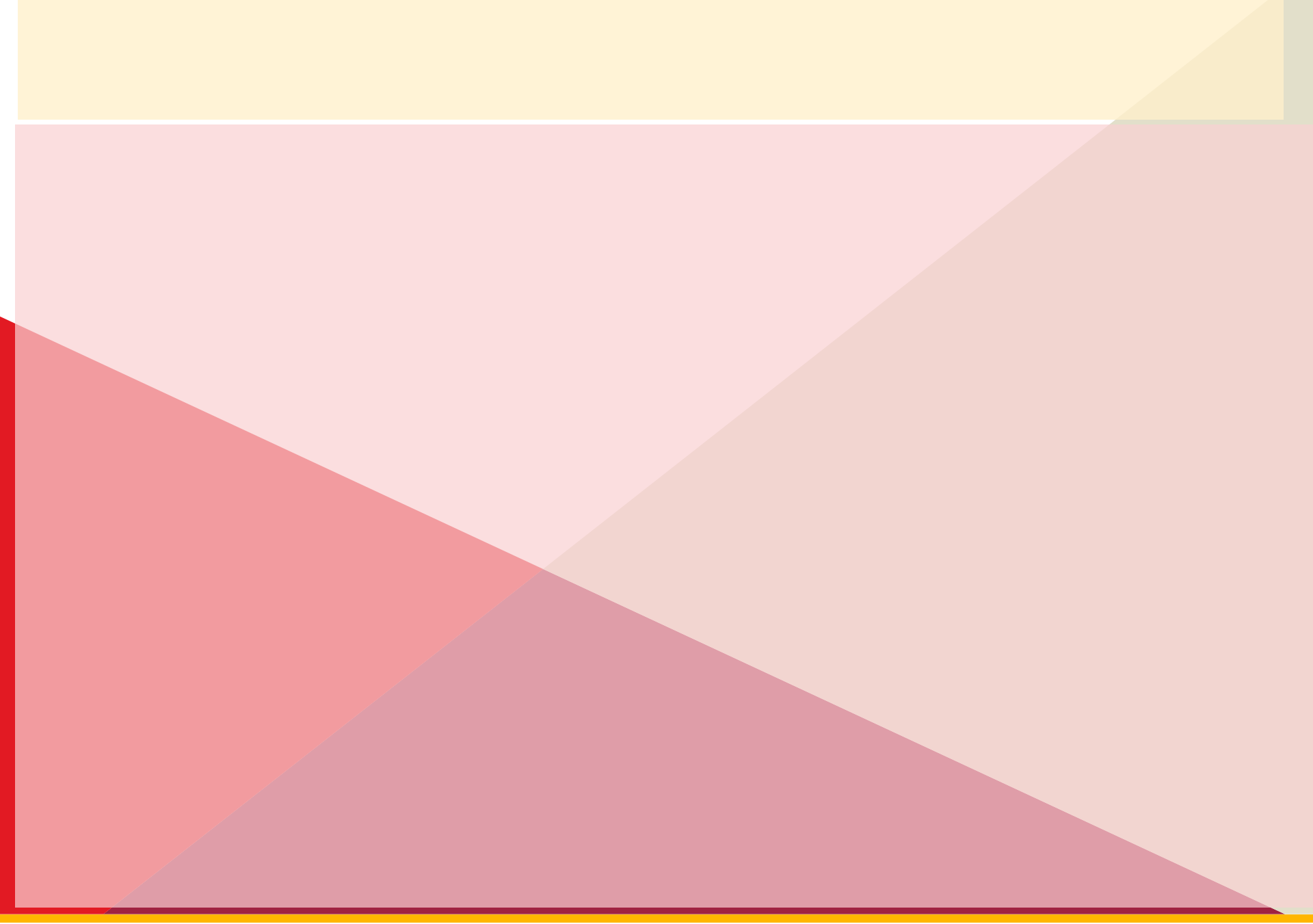
Similarly there are other functions of the os module.

[The book gives details of some important functions of the os module. They are pretty straightforward and don't need detailed explanation]

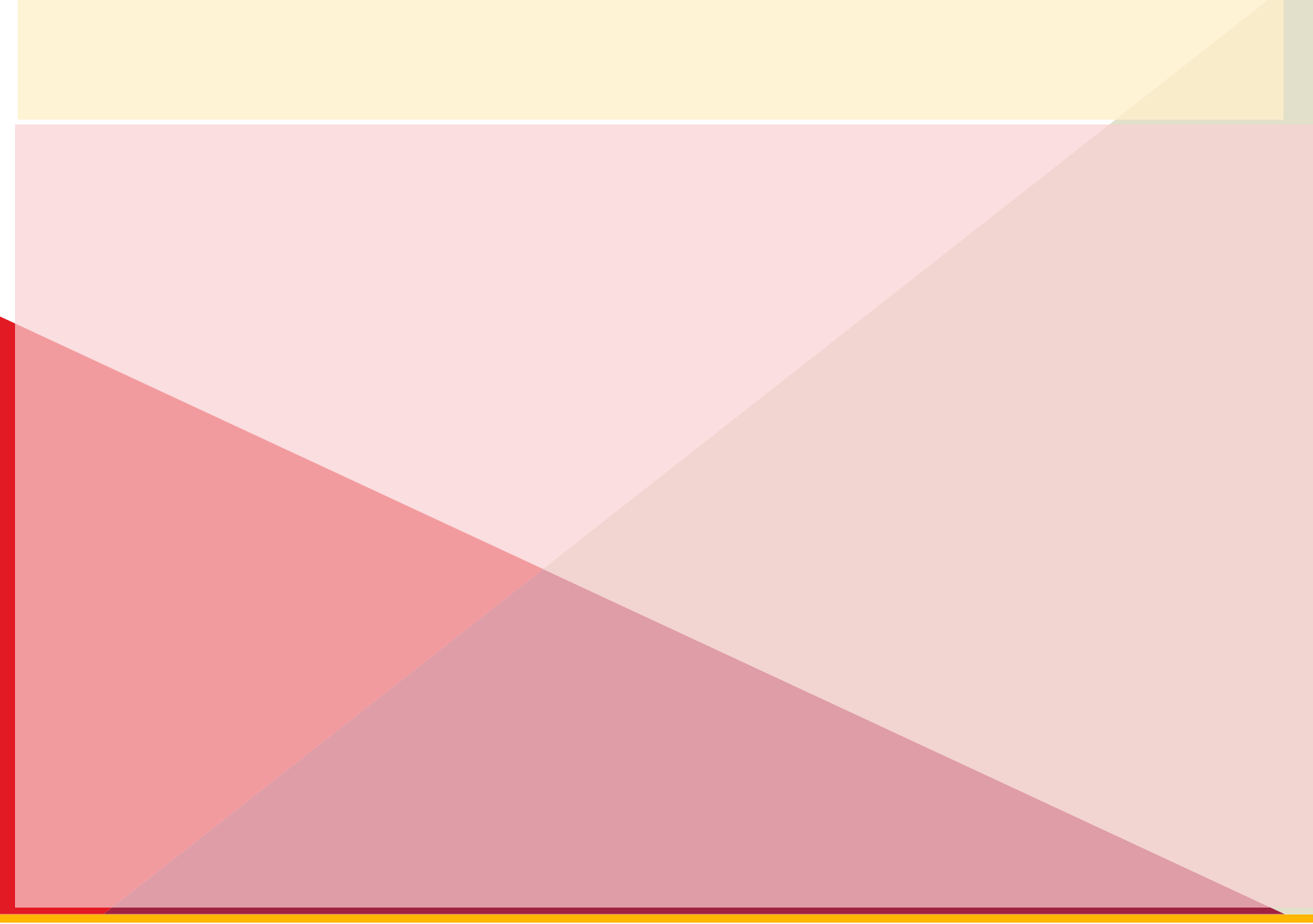


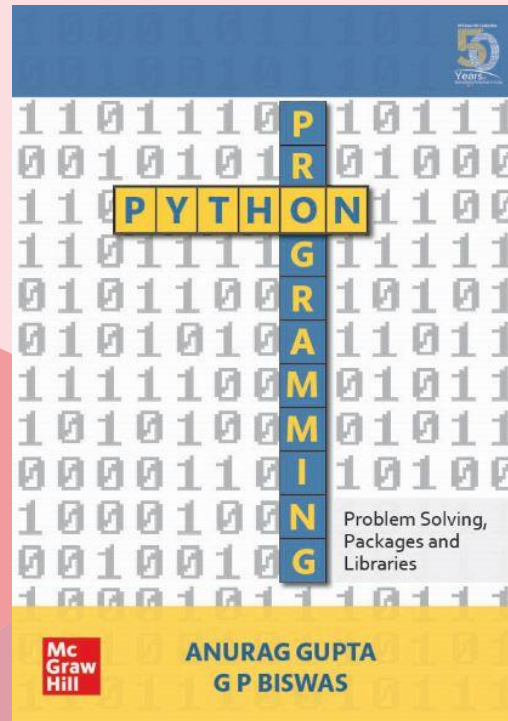















Because learning changes everything.®

# Thank You!

---

**For any queries or feedback contact us at:**

 support.india@mheducation.com

 1800-103-5875

 [www.mheducation.co.in](http://www.mheducation.co.in)

in

