Processing files

Accessing files from Python code

One of the most common issues in the developer's job is to **process data stored in files** while the files are usually physically stored using storage devices - hard, optical, network, or solid-state disks.

It's easy to imagine a program that sorts 20 numbers, and it's equally easy to imagine the user of this program entering these twenty numbers directly from the keyboard.

It's much harder to imagine the same task when there are 20,000 numbers to be sorted, and there isn't a single user who is able to enter these numbers without making a mistake.

It's much easier to imagine that these numbers are stored in the disk file which is read by the program. The program sorts the numbers and doesn't send them to the screen, but instead creates a new file and saves the sorted sequence of numbers there.

If we want to implement a simple database, the only way to store the information between program runs is to save it into a file (or files if your database is more complex).

In principle, any non-simple programming problem relies on the use of files, whether it processes images (stored in files), multiplies matrices (stored in files), or calculates wages and taxes (reading data stored in files).

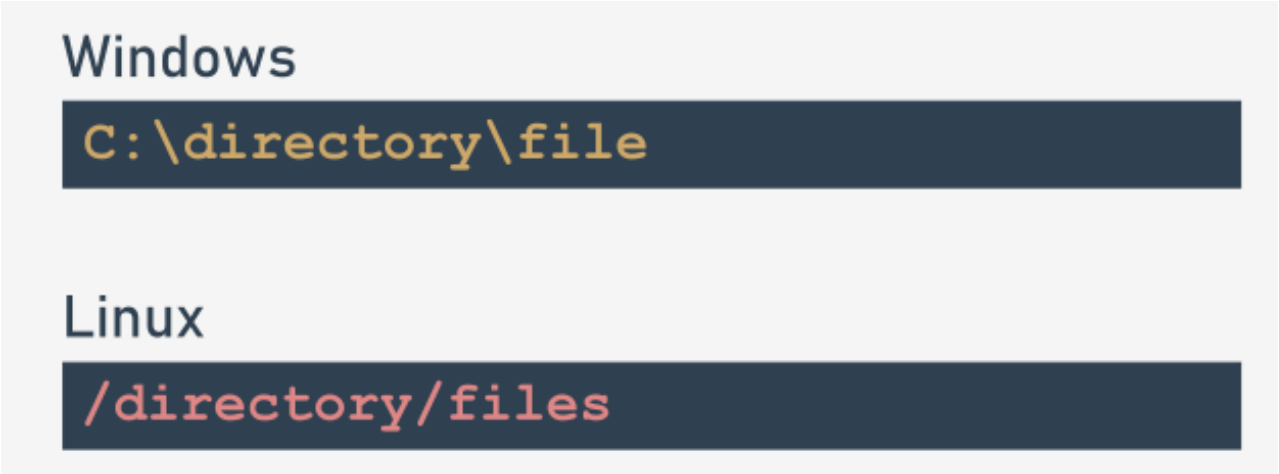
You may ask why we have waited until now to show you these issues.

The answer is very simple - Python's way of accessing and processing files is implemented using a consistent set of objects. There is no better moment to talk about it.

File names

Different operating systems can treat the files in different ways. For example, Windows uses a different naming convention than the one adopted in Unix/Linux systems.

If we use the notion of a canonical file name (a name which uniquely defines the location of the file regardless of its level in the directory tree) we can realize that these names look different in Windows and in Unix/Linux:



As you can see, systems derived from Unix/Linux don't use the disk drive letter (e.g., `C:`) and all the directories grow from one root directory called `/`, while Windows systems recognize the root directory as `\`.

In addition, Unix/Linux system file names are case-sensitive. Windows systems store the case of letters used in the file name, but don't distinguish between their cases at all.

This means that these two strings:

`ThisIsTheNameOfTheFile`
and

`thisisthenameofthefile`
describe two different files in Unix/Linux systems, but are the same name for just one file in Windows systems.

The main and most striking difference is that you have to use **two different separators for the directory names**: `\` in Windows, and `/` in Unix/Linux.

This difference is not very important to the normal user, but is **very important when writing programs in Python**.

To understand why, try to recall the very specific role played by the `\` inside Python strings.

Suppose you're interested in a particular file located in the directory `dir`, and named `file`.

Suppose also that you want to assign a string containing the name of the file.

In Unix/Linux systems, it may look as follows:

```
name = "/dir/file"
```

But if you try to code it for the Windows system:

```
name = "\dir\file"
```

you'll get an unpleasant surprise: either Python will generate an error, or the execution of the program will behave strangely, as if the file name has been distorted in some way.

In fact, it's not strange at all, but quite obvious and natural. Python uses the `\` as an escape character (like `\n`).

This means that Windows file names must be written as follows:

```
name = "\\dir\\file"
```

Fortunately, there is also one more solution. Python is smart enough to be able to convert slashes into backslashes each time it discovers that it's required by the OS.

This means that any the following assignments:

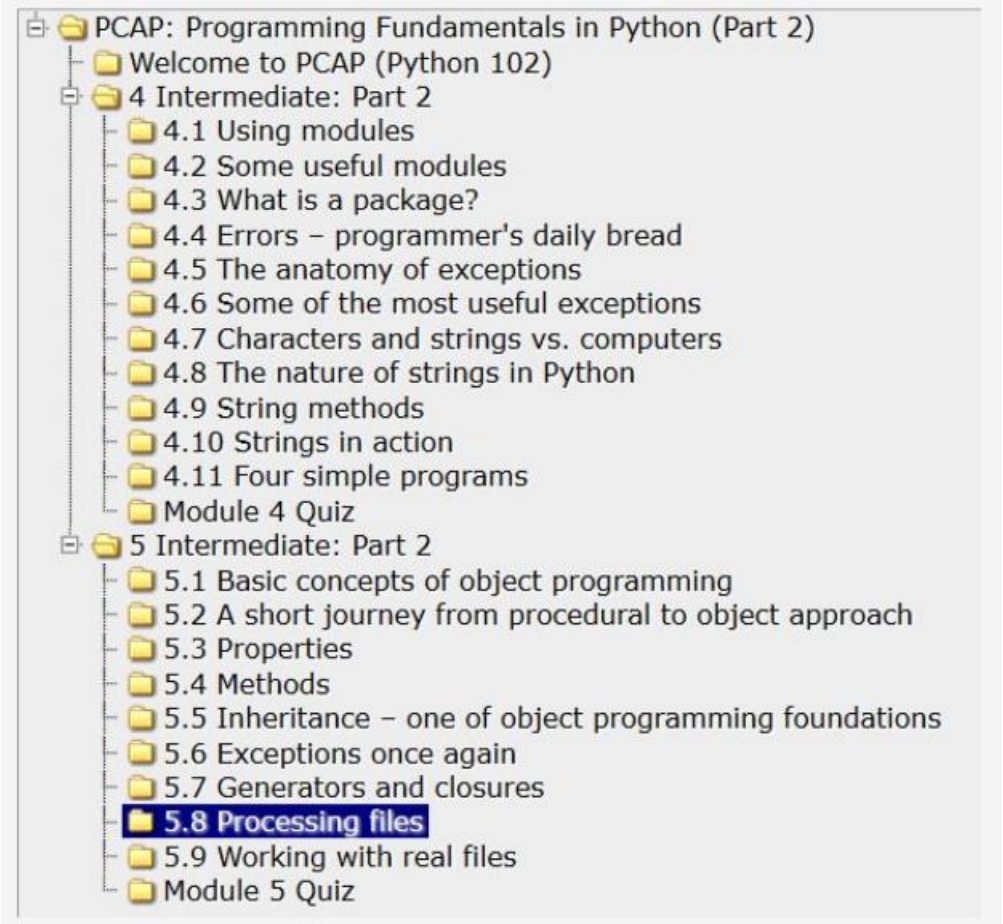
```
name = "/dir/file"
name = "c:/dir/file"
```

will work with Windows, too.

Any program written in Python (and not only in Python, because that convention applies to virtually all programming languages) does not communicate with the files directly, but through some abstract entities that are named differently in different languages or environments - the most-used terms are **handles** or **streams** (we'll use them as synonyms here).

The programmer, having a more- or less-rich set of functions/methods, is able to perform certain operations on the stream, which affect the real files using mechanisms contained in the operating system kernel.

In this way, you can implement the process of accessing any file, even when the name of the file is unknown at the time of writing the program.



The operations performed with the abstract stream reflect the activities related to the physical file.

To connect (bind) the stream with the file, it's necessary to perform an explicit operation.

The operation of connecting the stream with a file is called **opening the file**, while disconnecting this link is named **closing the file**.

Hence, the conclusion is that the very first operation performed on the stream is always `open` and the last one is `close`. The program, in effect, is free to manipulate the stream between these two events and to handle the associated file.

This freedom is limited, of course, by the physical characteristics of the file and the way in which the file has been opened.

Let us say again that the opening of the stream can fail, and it may happen due to several reasons: the most common is the lack of a file with a specified name.

It can also happen that the physical file exists, but the program is not allowed to open it. There's also the risk that the program has opened too many streams, and the specific operating system may not allow the simultaneous opening of more than `n` files (e.g., 200).

A well-written program should detect these failed openings, and react accordingly.

File streams

The opening of the stream is not only associated with the file, but should also declare the manner in which the stream will be processed. This declaration is called an **open mode**.

If the opening is successful, the **program will be allowed to perform only the operations which are consistent with the declared open mode**.

There are two basic operations performed on the stream:

- **read** from the stream: the portions of the data are retrieved from the file and placed in a memory area managed by the program (e.g., a variable);
- **write** to the stream: the portions of the data from the memory (e.g., a variable) are transferred to the file.

There are three basic modes used to open the stream:

- **read mode**: a stream opened in this mode allows **read operations only**; trying to write to the stream will cause an exception (the exception is named `UnsupportedOperation`, which inherits `OSError` and `ValueError`, and comes from the `io` module);
- **write mode**: a stream opened in this mode allows **write operations only**; attempting to read the stream will cause the exception mentioned above;
- **update mode**: a stream opened in this mode allows **both writes and reads**.

Before we discuss how to manipulate the streams, we owe you some explanation. **The stream behaves almost like a tape recorder**.

When you read something from a stream, a virtual head moves over the stream according to the number of bytes transferred from the stream.

When you write something to the stream, the same head moves along the stream recording the data from the memory.

Whenever we talk about reading from and writing to the stream, try to imagine this analogy. The programming books refer to this mechanism as the **current file position**, and we'll also use this term.

It's necessary now to show you the object responsible for representing streams in programs.

File handles

Python assumes that **every file is hidden behind an object of an adequate class**.

Of course, it's hard not to ask how to interpret the word *adequate*.

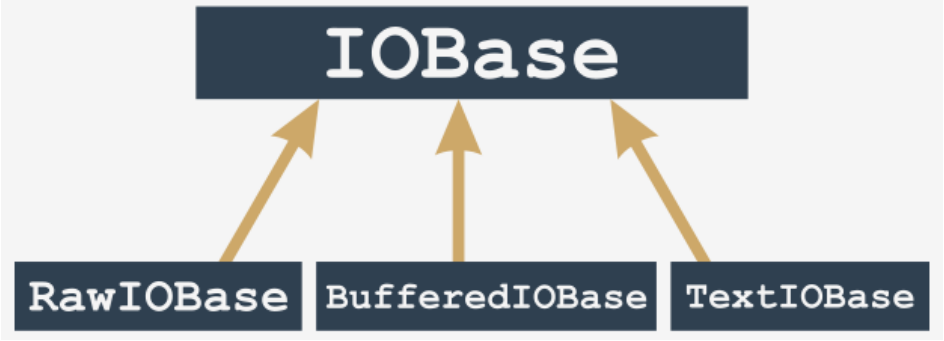
Files can be processed in many different ways - some of them depend on the file's contents, some on the programmer's intentions.

In any case, different files may require different sets of operations, and behave in different ways.

An object of an adequate class is **created when you open the file and annihilate it at the time of closing**.

Between these two events, you can use the object to specify what operations should be performed on a particular stream. The operations you're allowed to use are imposed by **the way in which you've opened the file**.

In general, the object comes from one of the classes shown here:



Note: you never use constructors to bring these objects to life. The only way you **obtain them is to invoke the function named** `open()`.

The function analyses the arguments you've provided, and automatically creates the required object.

If you want to **get rid of the object, you invoke the method named** `close()`.

The invocation will sever the connection to the object, and the file and will remove the object.

For our purposes, we'll concern ourselves only with streams represented by `BufferIOBase` and `TextIOBase` objects. You'll understand why soon.

Due to the type of the stream's contents, **all the streams are divided into text and binary streams**.

The text streams ones are structured in lines; that is, they contain typographical characters (letters, digits, punctuation, etc.) arranged in rows (lines), as seen with the naked eye when you look at the contents of the file in the editor.

This file is written (or read) mostly character by character, or line by line.

The binary streams don't contain text but a sequence of bytes of any value. This sequence can be, for example, an executable program, an image, an audio or a video clip, a database file, etc.

Because these files don't contain lines, the reads and writes relate to portions of data of any size. Hence the data is read/written byte by byte, or block by block, where the size of the block usually ranges from one to an arbitrarily chosen value.

Then comes a subtle problem. In Unix/Linux systems, the line ends are marked by a single character named `LF` (ASCII code 10) designated in Python programs as `\n`.

Other operating systems, especially these derived from the prehistoric CP/M system (which applies to Windows family systems, too) use a different convention: the end of line is marked by a pair of characters, `CR` and `LF` (ASCII codes 13 and 10) which can be encoded as `\r\n`.

This ambiguity can cause various unpleasant consequences.

If you create a program responsible for processing a text file, and it is written for Windows, you can recognize the ends of the lines by finding the `\r\n` characters, but the same program running in a Unix/Linux environment will be completely useless, and vice versa: the program written for Unix/Linux systems might be useless in Windows.

Such undesirable features of the program, which prevent or hinder the use of the program in different environments, are called **non-portability**.

Similarly, the trait of the program allowing execution in different environments is called **portability**. A program endowed with such a trait is called a **portable program**.

Since portability issues were (and still are) very serious, a decision was made to definitely resolve the issue in a way that doesn't engage the developer's attention.

It was done at the level of classes, which are responsible for reading and writing characters to and from the stream. It works in the following way:

- when the stream is open and it's advised that the data in the associated file will be processed as text (or there is no such advisory at all), it is **switched into text mode**;
- during reading/writing of lines from/to the associated file, nothing special occurs in the Unix environment, but when the same operations are performed in the Windows environment, a process called a **translation of newline characters** occurs: when you read a line from the file, every pair of `\r\n` characters is replaced with a single `\n` character, and vice versa; during write operations, every `\n` character is replaced with a pair of `\r\n` characters;
- the mechanism is completely **transparent** to the program, which can be written as if it was intended for processing Unix/Linux text files only; the source code run in a Windows environment will work properly, too;

- when the stream is open and it's advised to do so, its contents are taken as-is, **without any conversion** - no bytes are added or omitted.

Opening the streams

The **opening of the stream** is performed by a function which can be invoked in the following way:
`stream = open(file, mode = 'r', encoding = None)`

Let's analyze it:

- the name of the function (`open`) speaks for itself; if the opening is successful, the function returns a stream object; otherwise, an exception is raised (e.g., `FileNotFoundError` **if the file you're going to read doesn't exist**);
- the first parameter of the function (`file`) specifies the name of the file to be associated with the stream;
- the second parameter (`mode`) specifies the open mode used for the stream; it's a string filled with a sequence of characters, and each of them has its own special meaning (more details soon);
- the third parameter (`encoding`) specifies the encoding type (e.g., UTF-8 when working with text files)
- the opening must be the very first operation performed on the stream.

Note: the mode and encoding arguments may be omitted - their default values are assumed then. The default opening mode is reading in text mode, while the default encoding depends on the platform used.

Let us now present you with the most important and useful open modes. Ready?

Opening the streams: modes

`r` open mode: read

- the stream will be opened in **read mode**;
- the file associated with the stream **must exist** and has to be readable, otherwise the `open()` function raises an exception.

`w` open mode: write

- the stream will be opened in **write mode**;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist it will be created; if it exists, it will be truncated to the length of zero (erased); if the creation isn't possible (e.g., due to system permissions) the `open()` function raises an exception.

`a` open mode: append

- the stream will be opened in **append mode**;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; if it exists the virtual recording head will be set at the end of the file (the previous content of the file remains untouched.)

`r+` open mode: read and update

- the stream will be opened in **read and update mode**;
- the file associated with the stream **must exist and has to be writeable**, otherwise the `open()` function raises an exception;
- both read and write operations are allowed for the stream.

`w+` open mode: write and update

- the stream will be opened in **write and update** mode;
- the file associated with the stream **doesn't need to exist**; if it doesn't exist, it will be created; the previous content of the file remains untouched;
- both read and write operations are allowed for the stream.

Selecting text and binary modes

If there is a letter `b` at the end of the mode string it means that the stream is to be opened in the **binary mode**.

If the mode string ends with a letter `t` the stream is opened in the **text mode**.

Text mode is the default behaviour assumed when no binary/text mode specifier is used.

Finally, the successful opening of the file will set the current file position (the virtual reading/writing head) before the first byte of the file **if the mode is not** `a` and after the last byte of file **if the mode is set to** `a`.

Text mode	Binary mode	Description
`rt`	`rb`	read
`wt`	`wb`	write
`at`	`ab`	append
`r+t`	`r+b`	read and update
`w+t`	`w+b`	write and update

EXTRA

You can also open a file for its exclusive creation. You can do this using the `x` open mode. If the file already exists, the `open()` function will raise an exception.

Opening the stream for the first time

Imagine that we want to develop a program that reads content of the text file named: `C:\Users\User\Desktop\file.txt`.

How to open that file for reading? Here's the relevant snippet of the code:

```
try:
	stream = open("C:\Users\User\Desktop\file.txt", "rt")
	# processing goes here
	stream.close()
except Exception as exc:
	print("Cannot open the file:", exc)
```

What's going on here?

- we open the try-except block as we want to handle runtime errors softly;
- we use the `open()` function to try to open the specified file (note the way we've specified the file name)
- the open mode is defined as text to read (as **text is the default setting**, we can skip the `t` in mode string)
- in case of success we get an object from the `open()` function and we assign it to the stream variable;
- if `open()` fails, we handle the exception printing full error information (it's definitely good to know what exactly happened)

Pre-opened streams

We said earlier that any stream operation must be preceded by the `open()` function invocation. There are three well-defined exceptions to the rule.

When our program starts, the three streams are already opened and don't require any extra preparations. What's more, your program can use these streams explicitly if you take care to import the `sys` module:

```
import sys
```

because that's where the declaration of the three streams is placed.

The names of these streams are: `sys.stdin`, `sys.stdout`, and `sys.stderr`.

Let's analyze them:

- `sys.stdin`
  - `stdin` (as *standard input*)
  - the `stdin` stream is normally associated with the keyboard, pre-open for reading and regarded as the primary data source for the running programs;
  - the well-known `input()` function reads data from `stdin` by default.
- `sys.stdout`
  - `stdout` (as *standard output*)
  - the `stdout` stream is normally associated with the screen, pre-open for writing, regarded as the primary target for outputting data by the running program;
  - the well-known `print()` function outputs the data to the `stdout` stream.
- `sys.stderr`
  - `stderr` (as *standard error output*)

- the `stderr` stream is normally associated with the screen, pre-open for writing, regarded as the primary place where the running program should send information on the errors encountered during its work;
- we haven't presented any method to send the data to this stream (we will do it soon, we promise)
- the separation of `stdout` (useful results produced by the program) from the `stderr` (error messages, undeniably useful but does not provide results) gives the possibility of redirecting these two types of information to the different targets. More extensive discussion of this issue is beyond the scope of our course. The operation system handbook will provide more information on these issues.

Closing streams

The last operation performed on a stream (this doesn't include the `stdin`, `stdout`, and `stderr` streams which don't require it) should be **closing**.

That action is performed by a method invoked from within open stream object: `stream.close()`.

- the name of the function is definitely self-commenting (`close()`)
- the function expects exactly no arguments; the stream doesn't need to be opened
- the function returns nothing but raises `IOError` exception in case of error;
- most developers believe that the `close()` function always succeeds and thus there is no need to check if it's done its task properly.

  This belief is only partly justified. If the stream was opened for writing and then a series of write operations were performed, it may happen that the data sent to the stream has not been transferred to the physical device yet (due to mechanism called **caching** or **buffering**). Since the closing of the stream forces the buffers to flush them, it may be that the flushes fail and therefore the `close()` fails too.

We have already mentioned failures caused by functions operating with streams but not mentioned a word how exactly we can identify the cause of the failure.

The possibility of making a diagnosis exists and is provided by one of streams' exception component which we are going to tell you about just now.

Diagnosing stream problems

The `IOError` object is equipped with a property named `errno` (the name comes from the phrase *error number*) and you can access it as follows:

```
try:
    # some stream operations
except IOError as exc:
    print(exc.errno)
```

The value of the `errno` attribute can be compared with one of the predefined symbolic constants defined in the `errno` module.

Let's take a look at some selected **constants useful for detecting stream errors**:

`errno.EACCES` → `Permission denied`

The error occurs when you try, for example, to open a file with the *read only* attribute for writing.

`errno.EBADF` → `Bad file number`

The error occurs when you try, for example, to operate with an unopened stream.

`errno.EEXIST` → `File exists`

The error occurs when you try, for example, to rename a file with its previous name.

`errno.EFBIG` → `File too large`

The error occurs when you try to create a file that is larger than the maximum allowed by the operating system.

`errno.EISDIR` → `Is a directory`

The error occurs when you try to treat a directory name as the name of an ordinary file.

`errno.EMFILE` → `Too many open files`

The error occurs when you try to simultaneously open more streams than acceptable for your operating system.

`errno.ENOENT` → `No such file or directory`

The error occurs when you try to access a non-existent file/directory.

`errno.ENOSPC` → `No space left on device`

The error occurs when there is no free space on the media.

The complete list is much longer (it includes also some error codes not related to the stream processing.)

If you are a very careful programmer, you may feel the need to use the sequence of statements similar to those presented below:

```
import errno
try:
	s = open("c:/users/user/Desktop/file.txt", "rt")
	# actual processing goes here
	s.close()
except Exception as exc:
	if exc.errno == errno.ENOENT:
		print("The file doesn't exist.")
	elif exc.errno == errno.EMFILE:
		print("You've opened too many files.")
	else:
		printf("The error number is:", exc.errno)
```

Fortunately, there is a function that can dramatically **simplify the error handling code**. Its name is `strerror()`, and it comes from the `os` module and **expects just one argument - an error number**.

Its role is simple: you give an error number and get a string describing the meaning of the error.

Note: if you pass a non-existent error code (a number which is not bound to any actual error), the function will raise `ValueError` exception.

Now we can simplify our code in the following way:

```
from os import strerror
try:
	s = open("c:/users/user/Desktop/file.txt", "rt")
	# actual processing goes here
	s.close()
except Exception as exc:
	print("The file could not be opened:", strerror(exc.errno));
```

Okay. Now it's time to deal with text files and get familiar with some basic techniques you can use to process them.