Accessing Files in Python

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).

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:

Windows:

$C:\backslash directory\backslash file$

Linux

$/directory/files$

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 $file$.

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

$$name = "/dir/file"$$

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

$$name = "\backslash\backslash dir\backslash\backslash file"$$

Mind the \ escape character here.

Inconvenient, isn't it.

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 me 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**.

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:

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** $close()$.

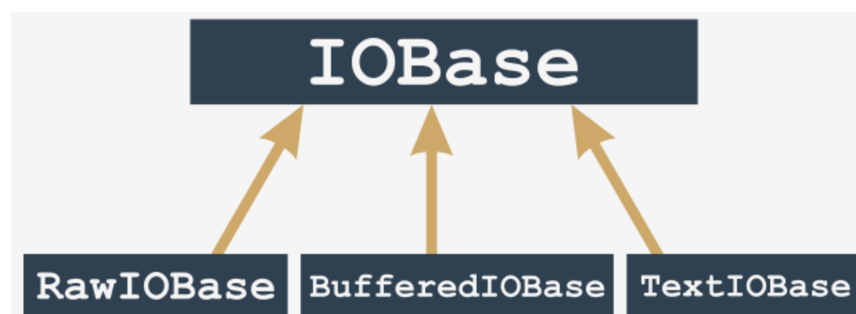

Image source: Cisco/Python Institute

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

Due to the type of the stream's contents, all the streams are divided into **text (words) and**

binary streams (an executable file, an image, an audio or a video clip or a database file, etc).

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 $\backslash n$.

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 $\backslash r \backslash n$.

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 $\backslash r \backslash n$ 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 $\backslash r \backslash n$ characters is replaced with a single $\backslash n$ character, and vice versa; during write operations, every $\backslash n$ character is replaced with a pair of $\backslash r \backslash n$ 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

```
stream = open(file, mode = 'r', encoding = None)
```

- 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.

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.

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

Modes associated with read must **exist**, but files associated with writes modes doesn't need to exist.

The $write$ mode will erase all elements in the original files if it originally exists.

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.

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

Here is an example of reading from stream and closing it.

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

$sys.stdin$, $sys.stdout$ and $sys.stderr$.

$sys.stdin$:

- Standard input
- 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.

$sys.stdout$:

- Standard output
- 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.

$sys.stderr$:

- Standard error output
- 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.

- 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. The operation system handbook will provide more information on these issues.

```
stream.close()
```

The function expects exactly no arguments.

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.

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 e:
    print(e.errno)
```

Some useful errors:

$error.EBADF$ -> 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.
$errno.ERBIG$ -> 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 of 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.

$strerror()$, it comes from the os module and expects just one argument – an error number.

```python
from os import strerror

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

Here is how it can be used.

Processing text files

Remember - our understanding of a text file is very strict. In our sense, it's a plain text file - it may contain only text, without any additional decorations (formatting, different fonts, etc.).

Use the very basics your OS offers, for example Notepad.

```python
stream = open("file.txt", 'rt', encoding = 'utf-8')
```

Reading a text file's contents can be performed using several different methods, the most basic of these methods is the one offered by the $read()$ function.

If applied to a text file, the function is able to:

- Read a desired number of characters (including just one) from the file, and return them as a string;
- Read all the file contents, and return them as a string;
- If there is nothing more to read (the virtual reading head reaches the end of the file), the function returns an empty string.

Read

```python
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    ch = s.read(1)
    while ch != '':
        print(ch, end = '')
        cnt += 1
        ch = s.read(1)
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

The code here reads an entire file (one character by one character) and prints the number of characters inside.

```python
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end = '')
        cnt += 1
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

The code here does the same job, but reads the entire file.

Remember - **reading a terabyte-long file using this method may corrupt your OS**.

$readline()$ Function

```python
from os import strerrror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()

    while line != '':
        lcnt += 1

        for ch in line:
            print(ch, end = '')
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file: ", ccnt)
    print("Lines in file: ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

The method tries to **read a complete line of text from the file**. It returns it as a string in the case of success. Otherwise, it returns an empty string.

$readlines()$ Function

It tries to read all the file contents, and returns **a list of strings**, one element per file line.

If you're not sure if the file size is small enough and don't want to test the OS, you can

convince the $readlines()$ method to read not more than a specified number of bytes at once (the returning value remains the same - it's a list of a string).

```
from os import strerrror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readline(20)

    while len(lines) != 0:
        for line in lines:
            lcnt += 1

            for ch in line:
                print(ch, end = '')
                ccnt += 1

        lines = s.readlines(10)
    s.close()
    print("\n\nCharacters in file: ", ccnt)
    print("Lines in file: ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

The maximum accepted input buffer size is passed to the method as its argument.

You may expect that $readlines()$ can process a file's contents more effectively than $readline()$, as it may need to be invoked fewer times.

When there is nothing to read from the file, the method returns an empty list.

The last example I want to present shows a very interesting trait of the object returned by the $open()$ function in text mode.

I think it may surprise you - **the object is an instance of the iterable class**.

Strange? Not at all. Usable? Yes, absolutely.

The **iteration protocol defined for the file object** is very simple - its __next__ method just **returns the next line read in from the file**.

Moreover, you can expect that the object automatically invokes $close()$ when any of the file reads reaches the end of the file.

```
from os import strerror

try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
```

```
        for ch in line:
            print(ch, end = '')
            ccnt += 1

    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

Write

Writing text files seems to be simpler, as in fact there is one method that can be used to perform such a task.

The method is named $write()$ and it expects just one argument - a string that will be transferred to an open file (don't forget - the open mode should reflect the way in which the data is transferred - **writing a file opened in read mode won't succeed**).

No newline character is added to the $write()$'s argument, so you have to add it yourself if you want the file to be filled with a number of lines.

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.
    for i in range(10):
        s = "line #" + str(i+1) + "\n"
        for ch in s:
            fo.write(ch)
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

The example here shows a very simple code that creates a file named $newtext.txt$ (note: the open mode w ensures that the **file will be created from scratch**, even if it exists and contains data) and then puts ten lines into it.

The string to be recorded consists of the word $line$, followed by the line number. I've decided to write the string's contents character by character (this is done by the inner for loop) but you're not obliged to do it in this way.

Expected output:

```
line #1
line #2
line #3
line #4
line #5
line #6
```

```
line #7
line #8
line #9
line #10
```

```
from os import strerror

try:
    fo = open('newtext.txt', 'wt')
    for i in range(10):
        fo.write("line #" + str(i+1) + "\n")
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

The code here does the same thing, put printing as strings.

Note: You can use the same method to write to the *stderr* stream, but don't try to open it, as it's always open implicitly.

For example, if you want to send a message string to *stderr* to distinguish it from normal program output, it may look like this:

```
import sys
sys.stderr.write("Error message")
```

Bytearray

Before we start talking about binary files, we have to tell you about one of the **specialized classes Python uses to store amorphous data**.

Amorphous data is data which have no specific shape or form - they are just a series of bytes.

This doesn't mean that these bytes cannot have their own meaning, or cannot represent any useful object, e.g., bitmap graphics.

The most important aspect of this is that in the place where we have contact with the data, we are not able to, or simply don't want to, know anything about it.

Amorphous data cannot be stored using any of the previously presented means - they are neither strings nor lists.

Therefore, there should be a special container able to handle such data.

Python has more than one such container - one of them is a **specialized class name** *bytearray* - as the name suggests, it's **an array containing (amorphous) bytes**.

If you want to have such a container, e.g., in order to read in a bitmap image and process it in any way, you need to create it explicitly, using one of available constructors.

```
data = bytearray(10)
```

Such an invocation creates a *bytearray* object able to store ten bytes, filled with zeros.

Bytearrays resemble lists in many respects. For example, they are **mutable**, they're a subject of the *len*() function, and you can access any of their elements using conventional

indexing.

There is one important limitation - **you mustn't set any byte array elements with a value which is not an integer** (violating this rule will cause a $TypeError$ exception) and you're **not allowed to assign a value that doesn't come from the range 0 to 255 inclusive** (unless you want to provoke a $ValueError$ exception).

You can treat any byte array elements as **integer** values, just like here:

```
data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 - i

for b in data:
    print(hex(b))
```

Note: We've used two methods to iterate the byte arrays, and made use of the $hex()$ function to see the elements printed as hexadecimal values.

Expected output:

```
0xa
0x9
0x8
0x7
0x6
0x5
0x4
0x3
0x2
0x1
```

So, how do we write a byte array to a binary file?

```
from os import strerror

data = bytearray(10)

for i in range(len(data)):
    data[i] = 10 + i

try:
    bf = open('file.bin', 'wb')
    bf.write(data)
    bf.close()
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

- First, we initialize $bytearray$ with subsequent values starting from 10. If you want the file's contents to be clearly readable, replace 10 with something like ord('a') - this will produce bytes containing values corresponding to the alphabetical part of the ASCII code (don't think it will make the file a text file - it's still binary, as it was created with a wb flag).
- Then, we create the file using the $open()$ function - the only difference compared to the previous variants is the open mode containing the b flag.

- The $write()$ method takes its argument ($bytearray$) and sends it (as a whole) to the file.
- The stream is then closed in a routine way.

The $write()$ method returns a number of successfully written bytes.

If the values differ from the length of the method's arguments, it may announce some write errors.

In this case, we haven't made use of the result - this may not be appropriate in every case.

Try to run the code and analyse the contents of the newly created output file.

How to read bytes from a stream

Reading from a binary file requires use of a specialized method name $readinto()$, as the method doesn't create a new byte array object, but fills a previously created one with the values taken from the binary file.

The method returns the number of successfully read bytes.

The method tries to fill the whole space available inside its argument; if there are more data in the file than space in the argument, the read operation will stop before the end of the file; otherwise, the method's result may indicate that the byte array has only been filled fragmentarily (the result will show you that, too, and the part of the array not being used by the newly read contents remains untouched).

```
from os import strerror

data = bytearray(10)

try:
    bf = open('file.bin', 'rb')
    bf.readinto(data)
    bf.close()

    for b in data:
        print(hex(b), end = ' ')
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

First, we open the file (the one you created using the previous code) with the mode described as rb.

Then, we read its contents into the byte array named $data$, of size ten bytes.

Finally, we print the byte array contents.

Be careful - **don't use this kind of read if you're not sure that the file's contents will fit the available memory**.

If the $read()$ method is invoked with an argument, it specifies the **maximum number of bytes to be read**.

The method tries to read the desired number of bytes from the file, and the length of the returned object can be used to determine the number of bytes actually read.

```
from os import strerror

try:
```

```
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

It can be done just like this.

The first five bytes of the file have been read by the code - the others are still waiting to be processed.

Copying files

```
from os import strerror

srcname = input("Enter the source file name: ")
try:
    src = open(srcname, 'rb')
except IOError as e:
    print("Cannot open the source file: ", strerror(e.errno))
    exit(e.errno)

dstname = input("Enter the destination file name: ")
try:
    dst = open(dstname, 'wb')
except Exception as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    src.close()
    exit(e.errno)

buffer = bytearray(65536)
total = 0
try:
    readin = src.readinto(buffer)
    while readin > 0:
        written = dst.write(buffer[:readin])
        total += written
        readin = src.readinto(buffer)
except IOError as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    exit(e.errno)

print(total,'byte(s) succesfully written')
src.close()
dst.close()
```

Let's analyse the code here.

- Lines 3 through 8: ask the user for the name of the file to copy, and try to open it to read. Terminate the program execution if the open fails. Note: use the $exit()$ function to stop program execution and to pass the completion code to the OS. Any

completion code other than 0 says that the program has encountered some problems. Use the $errno$ value to specify the nature of the issue.

- Lines 10 through 16: repeat nearly the same action, but this time for the output file.
- Line 18: prepare a piece of memory for transferring data from the source file to the target one. Such a transfer area is often called a buffer, hence the name of the variable. The size of the buffer is arbitrary - in this case, we decided to use 64 kilobytes. Technically, a larger buffer is faster at copying items, as a larger buffer means fewer I/O operations. Actually, there is always a limit, the crossing of which renders no further improvements, test it yourself if you want.
- Line 19: count the bytes copied - this is the counter and its initial value.
- Line 21: try to fill the buffer for the very first time.
- Line 22: as long as you get a non-zero number of bytes, repeat the same actions.
- Line 23: write the buffer's contents to the output file (note: we've used a slice to limit the number of bytes being written, as $write()$ always prefer to write the whole buffer).
- Line 24: update the counter.
- Line 25: read the next file chunk.
- Lines 30 through 32: some final cleaning - the job is done.

Of course, the purpose is not to make a better replacement for commands like $copy$ (MS Windows) or cp (Unix/Linux) but to see one possible way of creating a working tool, even if nobody wants to use it.