Working with real files

Processing text files

In this lesson we're going to prepare a simple text file with some short, simple content.

We're going to show you some basic techniques you can utilize to **read the file contents** in order to process them.

The processing will be very simple - you're going to copy the file's contents to the console, and count all the characters the program has read in.

But 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.).

That's why you should avoid creating the file using any advanced text processor like MS Word, LibreOffice Writer, or something like this. Use the very basics your OS offers: Notepad, vim, gedit, etc.

If your text files contain some national characters not covered by the standard ASCII charset, you may need an additional step. Your `open()` function invocation may require an argument denoting specific text encoding.

For example, if you're using a Unix/Linux OS configured to use UTF-8 as a system-wide setting, the `open()` function may look as follows:

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

where the encoding argument has to be set to a value which is a string representing proper text encoding (UTF-8, here).

Consult your OS documentation to find an encoding name adequate to your environment.

INFORMATION

For the purposes of our experiments with file processing carried out in this section, we're going to use a pre-uploaded set of files (e.g., `tzop.txt`, or `text.txt` files) which you'll be able to work with. If you'd like to work with your own files locally on your machine, we strongly encourage you to do so, and to use IDLE to carry out your own tests.

Reading a text file's contents can be performed using several different methods - none of them is any better or worse than any other. It's up to you which of them you prefer and like.

Some of them will sometimes be handier, and sometimes more troublesome. Be flexible. Don't be afraid to change your preferences.

The most basic of these methods is the one offered by the `read()` function, which you were able to see in action in the previous lesson.

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.

We'll start with the simplest variant and use a file named `text.txt`. The file has the following contents:

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
```

Now look at the code below, and let's analyze it.

```
1  from os import strerror
2
3  try:
4      cnt = 0
5      s = open('text.txt', "rt")
6      ch = s.read(1)
7      while ch != '':
8          print(ch, end='')
9          cnt += 1
10         ch = s.read(1)
11     s.close()
12     print("\n\nCharacters in file:", cnt)
13 except IOError as e:
14     print("I/O error occurred: ", strerr(e.errno))
```

The routine is rather simple:

- use the try-except mechanism and open the file of the predetermined name (`text.txt` in our case)
- try to read the very first character from the file (`ch = s.read(1)`)
- if you succeed (this is proven by a positive result of the `while` condition check), output the character (note the `end=` argument - it's important! You don't want to skip to a new line after every character!);
- update the counter (`cnt`), too;
- try to read the next character, and the process repeats.

If you're absolutely sure that the file's length is safe and you can read the whole file to the memory at once, you can do it - the `read()` function, invoked without any arguments or with an argument that evaluates to `None`, will do the job for you.

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

Don't expect miracles - computer memory isn't stretchable.

Look at the code below.

```
from os import strerror

try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        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: ", strerr(e.errno))
```

What do you think of it?

Let's analyze it:

- open the file as previously;
- read its contents by one `read()` function invocation;
- next, process the text, iterating through it with a regular `for` loop, and updating the counter value at each turn of the loop;

The result will be exactly the same as previously.

Processing text files: `readline()`

If you want to treat the file's contents **as a set of lines**, not a bunch of characters, the `readline()` method will help you with that.

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

This opens up new opportunities - now you can also count lines easily, not only characters.

Let's make use of it. Look at the code.

```
from os import strerror

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:", strerr(e.errno))
```

As you can see, the general idea is exactly the same as in both previous examples.

Processing text files: `readlines()`

Another method, which treats text file as a set of lines, not characters, is `readlines()`.

The `readlines()` method, when invoked without arguments, 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).

Feel free to experiment with this example code to understand how the `readlines()` method works.

```
s = open("text.txt")

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

print(s.readlines(20))

s.close()
```

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.

Note: when there is nothing to read from the file, the method returns an empty list. Use it to detect the end of the file.

To the extent of the buffer's size, you can expect that increasing it may improve input performance, but there is no golden rule for it - try to find the optimal values yourself.

Look at this code.

```
from os import strerror

try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    lines = s.readlines(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:", strerr(e.errno))
```

We've modified it to show you how to use `readlines()`.

We've decided to use a 15-byte-long buffer. Don't think it's a recommendation.

We've used such a value to avoid the situation in which the first `readlines()` invocation consumes the whole file.

We want the method to be forced to work harder, and to demonstrate its capabilities.

There are **two nested loops in the code**: the outer one uses `readlines()`'s result to iterate through it, while the inner one prints the lines character by character.

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

We 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.

Look at the code below and see how simple and clear the code has now become.

```
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: ", strerr(e.errno))
```

Dealing with text files: `write()`

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

p>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.

The example below 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.

```
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: ", strerr(e.errno))
```

The string to be recorded consists of the word line, followed by the line number. We'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.

We just wanted to show you that `write()` is able to operate on single characters.

The code creates a file filled with the following text:

```
line #1
line #2
```

```
line #3
line #4
line #5
line #6
line #7
line #8
line #9
line #10
```

We encourage you to test the behavior of the `write()` method locally on your machine.

Look at the example.

```
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: ", strerr(e.errno))
```

We've modified the previous code to write whole lines to the text file.

The contents of the newly created file are the same.

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")
```

What is a 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.

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.

Take a look:

```
data = bytearray(100)
```

Such an invocation creates a bytearray object able to store ten bytes.

Note: such a constructor **fills the whole array 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 in the example below.

```
1  data = bytearray(10)
2
3  for i in range(len(data)):
4      data[i] = 10 - i
5
6  for b in data:
7      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.

Now we're going to show you **how to write a byte array to a binary file** - binary, as we don't want to save its readable representation - we want to write a one-to-one copy of the physical memory content, byte by byte.

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

Look at the code.

```
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:", strerr(e.errno))
```

Let's analyze it:

- 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 analyze the contents of the newly created output file.

You're going to use it in the next step.

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.

Note:

- 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)

Look at the complete code below:

```
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:", strerr(e.errno))
```

Let's analyze it:

- 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 - are they the same as you expected?

Run the code and check if it's working.

An alternative way of reading the contents of a binary file is offered by the method named `read()`.

Invoked without arguments, it tries to **read all the contents of the file into the memory**, making them a part of a newly created object of the bytes class.

This class has some similarities to `bytearray`, with the exception of one significant difference - it's **immutable**.

Fortunately, there are no obstacles to creating a byte array by taking its initial value directly from the bytes object, just like here:

```
from os import strerror

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

	for b in data:
		print(hex(b), end=' ')

except IOError as e:
	print("I/O error occurred:", strerr(e.errno))
```

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.

You can use the method just like here:

```
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:", strerr(e.errno))
```

Note: the first five bytes of the file have been read by the code - the next five are still waiting to be processed.

Copying files - a simple and functional tool

Now you're going to amalgamate all this new knowledge, add some fresh elements to it, and use it to write a real code which is able to actually copy a file's contents.

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.

Look at the code below.

```
1  from os import strerror
2
3  srcname = input("Source file name?: ")
4  try:
5      src = open(srcname, 'rb')
6  except IOError as e:
7      print("Cannot open source file: ", strerror(e.errno))
8      exit(e.errno)
9  dstname = input("Destination file name?: ")
10 try:
11     dst = open(dstname, 'wb')
12 except Exception as e:
13     print("Cannot create destination file: ", strerr(e.errno))
14     src.close()
15     exit(e.errno)
16
17 buffer = bytearray(65536)
18 total  = 0
19 try:
20     readin = src.readinto(buffer)
21     while readin > 0:
22         written = dst.write(buffer[:readin])
23         total += written
24         readin = src.readinto(buffer)
25 except IOError as e:
26     print("Cannot create destination file: ", strerr(e.errno))
27     exit(e.errno)
28
29 print(total,'byte(s) succesfully written')
30 src.close()
31 dst.close()
```

Let's analyze it:

- 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 9 through 15: repeat nearly the same action, but this time for the output file;
- line 17: 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 18: count the bytes copied - this is the counter and its initial value;
- line 20: try to fill the buffer for the very first time;
- line 21: as long as you get a non-zero number of bytes, repeat the same actions;
- line 22: 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 23: update the counter;
- line 24: read the next file chunk;
- lines 29 through 31: some final cleaning - the job is done.