



# TECHNION

Azrieli Continuing Education and  
External Studies Division

## Module 5.6.1: Files

ALL RIGHTS RESERVED © COPYRIGHT 2022  
DO NOT DISTRIBUTE WITHOUT WRITTEN PERMISSION

# What are Files?

Files are a long sequence of bytes, saved on the hard disk.

Sometimes we will want to read them, and sometimes to write them.

Thankfully, this is very easy to do in Python!

# So what are files?

Files are a long sequence of bytes.

Not characters.

Not letters.

**Bytes!**

So what do we read from files?

**Bytes.**

What do we write to files?

**Bytes!**

# How do we create bytes?

In order to create bytes, we need to **encode** strings.

What is an encoding?

Let's remind ourselves what ASCII is.

# Character Set

- A character set is a group of all possible characters in a certain encoding.
- An encoding is a way to turn a string into a series of bits (but more on that in a future lesson).
- The most common character set is ASCII – in which all characters are encoded into 1 byte (8 bits).

# ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

# So to *Encode* is to What?

To ***encode*** is to translate a string into the bytes that represent each letter.

ASCII is the easiest encoding, useful only with English. We will learn other encodings in the future.

For example, encoding 'A' using ASCII is to translate 'A' into the hexadecimal byte **41**, written in Python as `'\x41'`.

This turns a ***string*** object into a ***bytes*** object.

# Creating Bytes in Python

```
In [1]: 'A'.encode()  
Out[1]: b'A'
```

It's as easy as that.

The **b** before the quotes shows that these are bytes, not a string.

This is what we can use in order to write strings to files!

We can also

```
In [2]: type(b'Hello, World!')  
Out[2]: bytes
```



# Converting Bytes into Strings

The same can be done the other way around.

Bytes can be converted into the strings that they represent by using the **.decode()** string method:

```
In [3]: alphabet = b'abcdefg'
```

```
In [4]: type(alphabet)
```

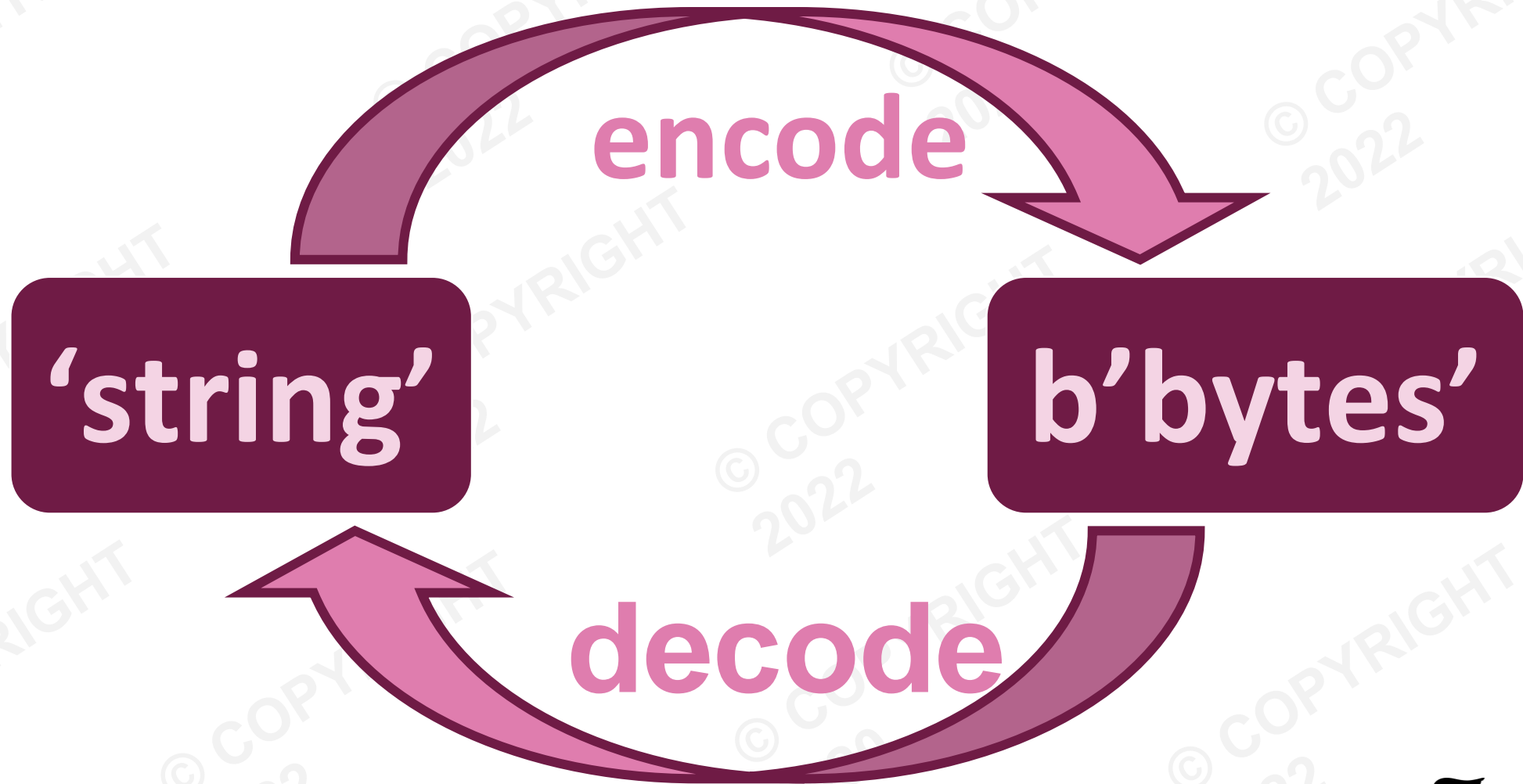
```
Out[4]: bytes
```

```
In [5]: alphabet = alphabet.decode()
```

```
In [6]: type(alphabet)
```

```
Out[6]: str
```

## Your Handy Guide



# Saving Paths

Let's remember what we learned in our “strings” lesson...

# Raw Strings - No Escapes Allowed

- Using **raw strings**, Python ignores all backslashes, not treating them as escape characters anymore.
- To define a raw string, add 'r' before defining your string.

```
>>> file_path = 'C:\temp\newfile.txt'
>>> print(file_path)
C:      emp
ewfile.txt
>>> escaped_file_path = 'C:\\temp\\newfile.txt'
>>> print(escaped_file_path)
C:\temp\newfile.txt
>>> raw_file_path = r'C:\temp\newfile.txt'
>>> print(raw_file_path)
C:\temp\newfile.txt
```

# Accessing a File

To access a file, we will use the ***open*** function.

***Open*** receives two inputs: the file path, and the mode.

We will use one of two modes:

- “rb” – used for reading files
- “wb” – used for writing files

# Closing the File

When we access a file and open it, our Operating System locks it only for us. This means no one else can access it while the file is open.

This means that once we are done using the file (reading/writing), we will **always** want to close it. If we don't others will not be able to access it.

We close the file using the **.close()** method.

# Reading a File

Step 1 – Opening the file – using the **open()** function, and giving the path, and the mode **'rb'**

Step 2 – Reading the data – using the method **.read()**

Step 3 – Closing the file – using the method **.close()**

## Reading a File (cont.)

```
In [7]: file = open(r"C:\python\hello.txt", 'rb')
```

```
In [8]: text_bytes = file.read()
```

```
In [9]: file.close()
```

```
In [10]: text_bytes
```

```
Out[10]: b'Hello, World!'
```

```
In [11]: text = text_bytes.decode()
```

```
In [12]: print(text)
```

```
Hello, World!
```



# Writing a File

Step 1 – Opening the file – using the **open()** function, and giving the path, and the mode **'wb'**

Step 2 – Writing the data – using the method **.write()**, and giving the bytes we want to write.

Step 3 – Closing the file – using the method **.close()**

## Writing a File (cont.)

```
In [21]: text = 'Agent Bell, reporting for duty!'
```

```
In [22]: text_bytes = my_text.encode()
```

```
In [23]: text_bytes
```

```
Out[23]: b'Agent Bell, reporting for duty!'
```

```
In [24]: file = open(r'C:\Python\agent.txt', 'wb')
```

```
In [25]: file.write(text_bytes)
```

```
Out[25]: 31
```

```
In [26]: file.close()
```

# The Path of the File to Write

The path you give the ***open()*** function when writing can be an existent file, or an inexistent file.

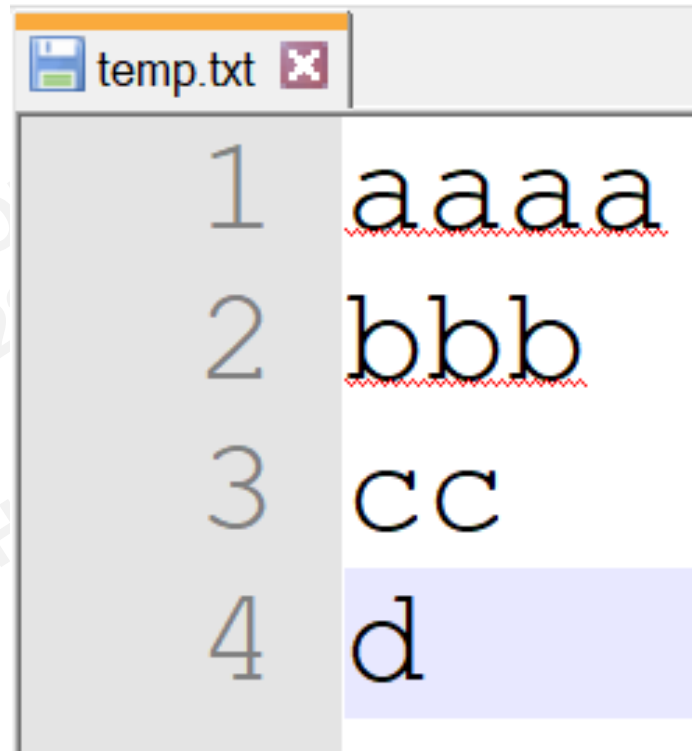
If the file doesn't exist – a new file will be created.

If the file exists – the existing file will be erased and replaced.

But, if the directory the file is located in doesn't exist – an error will be raised.

# Reading

Let's go back to reading a file. We'll use this file, named temp.txt:



1	aaaa
2	bbbb
3	cc
4	d

## Reading

```
In [1]: file = open(r'C:\Python\temp.txt', 'rb')
```

```
In [2]: data_bytes = file.read()
```

```
In [3]: file.close()
```

```
In [4]: data = data_bytes.decode()
```

```
In [5]: data
```

```
Out[5]: 'aaaa\r\nbbb\r\ncc\r\n'
```

```
In [6]: print(data)
```

```
aaaa
```

```
bbb
```

```
cc
```

```
d
```

# Splitting Text into Lines

We can use the `.split()` method, with the newline character as the delimiter.

```
In [7]: data_lines = data.split('\r\n')
```

```
In [8]: data_lines
```

```
Out[8]: ['aaaa', 'bbb', 'cc', 'd']
```

We can also use the `.splitlines()` method, which is much easier!

```
In [9]: data_lines = data.splitlines()
```

```
In [10]: data_lines
```

```
Out[10]: ['aaaa', 'bbb', 'cc', 'd']
```

# Reading Lines from a File

```
In [11]: file = open(r'C:\Python\temp.txt', 'rb')
```

```
In [12]: data = file.read().decode()
```

```
In [13]: file.close()
```

```
In [14]: for line in data.splitlines():  
        ...:     print(line)  
        ...:
```

```
aaaa  
bbb  
cc  
d
```

## The .read() Method

**Read**, when executed without an input, reads all of the bytes of the file. What happens if we use it twice?

```
In [15]: file = open(r'C:\Python\alphabet.txt', 'rb')
```

```
In [16]: file.read()
```

```
Out[16]: b'abcdefghijklmnopqrstuvwxyz'
```

```
In [17]: file.read()
```

```
Out[17]: b''
```



# File Pointer

Just like in variables, our pointer shows us where in the file we read from.

When the file has just been opened, the file pointer points to the beginning of the file. As the file is being read, the pointer advances over the bytes of the file.

When all bytes have been read, the pointer is already at the end of the file, so executing the **.read()** method again will read 0 more bytes.

## File Pointer in Action

abcdefghijklmnopqrstuvwxyz

In [15]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [16]: file.read()

Out[16]: b'abcdefghijklmnopqrstuvwxyz'

In [17]: file.read()

Out[17]: b''

## **.read(num\_of\_bytes)**

Instead of using **.read()** to read the whole file, we can give it the number of bytes to read.

The pointer will walk along with it!

abcdefghijklmnopqrstuvwxyz

↑  
In [18]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [19]: file.read(2)

Out[19]: b'ab'

In [20]: file.read(4)

Out[20]: b'cdef'

In [21]: file.read(8)

Out[21]: b'ghijklmn'

In [22]: file.read()

Out[22]: b'opqrstuvwxyz'

## Where is My Pointer Now? - .tell()

We can use the `.tell()` method to ask where the pointer is currently placed:

```
In [23]: file = open(r'C:\Python\alphabet.txt', 'rb')
```

```
In [24]: file.read(3)
```

```
Out[24]: b'abc'
```

```
In [25]: file.read(5)
```

```
Out[25]: b'defgh'
```

```
In [26]: file.tell()
```

```
Out[26]: 8
```

**Where is the pointer now?  
8 bytes in!**

# Changing the Pointer Position

I can **change** the pointer's position by using the method **.seek()**, and giving the position to change to.

abcdefghijklmnopqrstuvwxyz

In [27]: file = open(r'C:\Python\alphabet.txt', 'rb')

In [28]: file.read(5)

Out[28]: b'abcde'

In [29]: file.seek(2)

Out[29]: 2

In [30]: file.read(3)

Out[30]: b'cde'

In [31]: file.seek(14)

Out[31]: 14

In [32]: file.read()

Out[32]: b'opqrstuvwxyz'

# Reading the File Twice

So, in order for us to reread the file (use the `.read()` method twice), all we need to do is to reset the pointer to the beginning of the file!

```
In [7]: file = open(r'C:\Python\alphabet.txt', 'rb')
```

```
In [8]: file.read()
```

```
Out[8]: b'abcdefghijklmnopqrstuvwxyz'
```

```
In [9]: file.seek(0)
```

```
Out[9]: 0
```

```
In [10]: file.read()
```

```
Out[10]: b'abcdefghijklmnopqrstuvwxyz'
```

```
In [11]: file.close()
```



## The with Keyword

Up until now, opening a file took a few lines, and you had to make sure you remember to close the file.

```
def read_file_content(file_path):  
    file = open(file_path, 'rb')  
    data = file.read().decode()  
    file.close()  
return data
```

## The with Keyword (cont.)

But what if you forget to close the file?

Or what if an error gets raised, before the **.close()** method was executed?

# The with Keyword - Reading

Using **with**, we can handle files easier!

```
def read_file_content(file_path):  
    with open(file_path, 'rb') as f:  
        return f.read().decode()
```

The **with** creates the **open()** object and saves it in the variable **f**.

It also makes sure to execute **.close()** for you, even if an error happens!

## The with Keyword - Writing

```
def write_file_content(file_path, string_to_write):  
    with open(file_path, 'wb') as f:  
        f.write(string_to_write.encode())
```

## Modes Without 'b'

Apart from modes 'rb' and 'wb', there are the modes 'r' and 'w'.

Repeat after me: We will never use modes 'r' and 'w'!

Explanation:

- Whenever we read or write English text files, there are certain bytes that make the writing/reading **stop** when the pointer reaches them.
- We will usually write and read files that are not English text, which means that these bytes should be ignored.
- The 'b' in the mode stands for *binary*. This means that we should write and read our files, while ignoring the stopping bytes.
- This is why we will **always** use the binary modes!

# Summary

- Files are a sequence of bytes
- The **byte** type
- Encoding and Decoding
- Reading Files (mode '**rb**')
  - Writing Files (mode '**wb**')
    - Reading Multi-line Text
    - File Pointers – the **read**, **tell** and **seek** Methods
    - The **with** Keyword