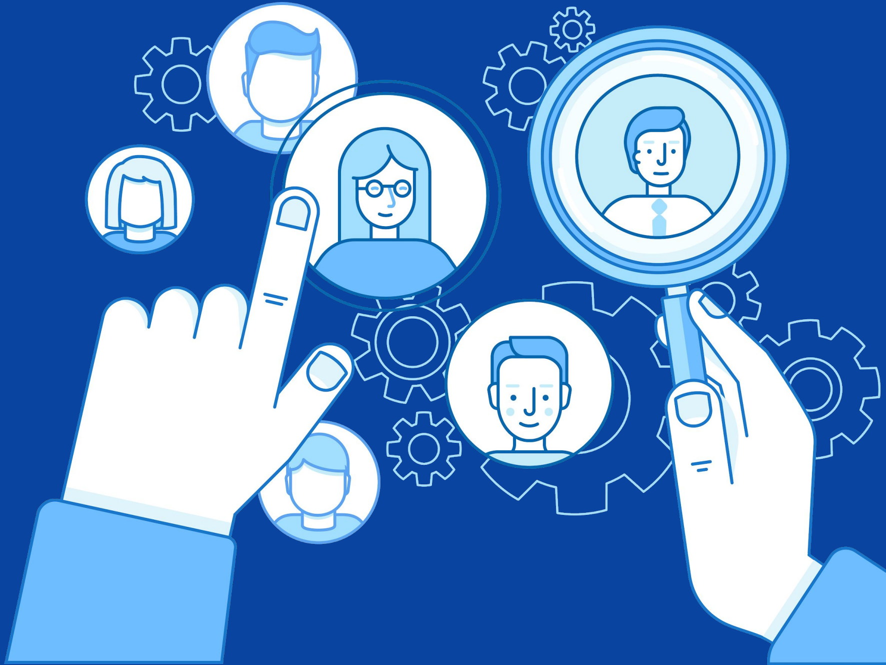
Digital Career Institute

Python Course: Input & Output



Digital Career Institute

Goal of the Submodule

The goal of this submodule is to help students learn different methods of data input and output. By the end of this submodule, the learners will be able to understand:

- How to input/output data to a text and binary file.
- The difference between the **`bytes`** and **`bytearray`** types.
- What are I/O streams and most of their methods and properties.
- How to work with directories and files.
- Different ways to capture user input and output data to the user.

Topics

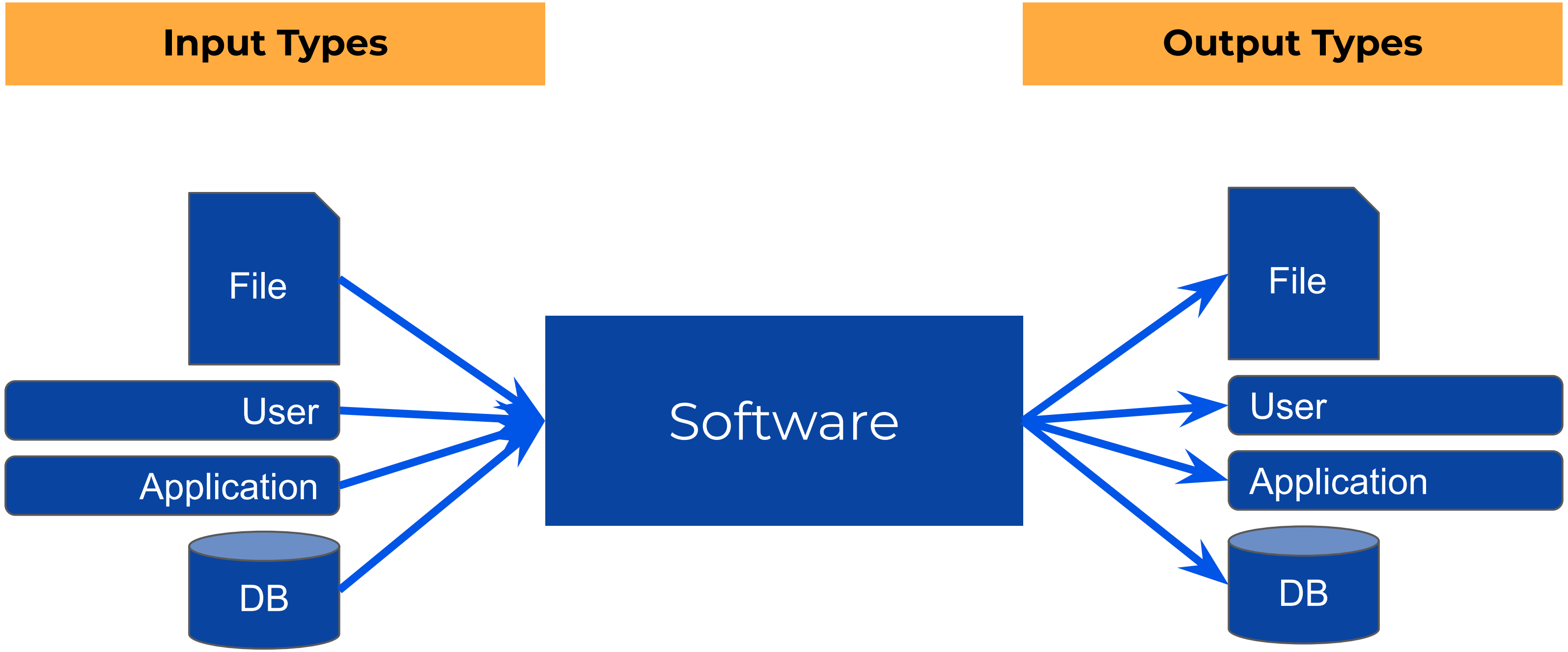
- The `open` function and reading a file.
- The `with` operator.
- Writing files.
- File opening modes.
- Bytes-like objects and binary files.
- Streams.
- Using the file system.
- User I/O.
- Application I/O.

Digital Career Institute

Input & Output

Digital Career Institute

Data Input & Output



File I/O: open()

```
>>> file = open("the_hobbit.txt")
```

The **open** function is a built-in Python function. It does not need to be imported from any package.

```
>>> print(open.__doc__)
```

Opening Files: path

```
>>> file = open("the_hobbit.txt")
```

```
>>> file = open("books/the_hobbit.txt")
```

```
>>> file = open("/home/user/the_hobbit.txt")
```

We can use **relative** and **absolute** paths.

File Input

Digital Career Institute

Reading Text Files

```
>>> file = open("the_hobbit.txt")
>>> print(file)
<_io.TextIOWrapper name='test.txt'
mode='r' encoding='UTF-8'>
```

The **`open`** function returns a **`File Object`**.

Reading Text Files: read()

```
>>> file = open("the_hobbit.txt")
>>> print(file.read())
In a hole in the ground there lived a
hobbit...
```

The **`file`** object has a **`read`** method to access the contents of a text file.

Reading Text Files: read()

```
>>> file = open("the_hobbit.txt")
>>> print(file.read(10))
In a hole
```

The **`read`** method has a parameter **`size`** to limit the length of the reading.

Defaults to `EOF` (End Of File).

Reading Text Files: readlines()

```
>>> file = open("the_hobbit.txt")
>>> for line in file.readlines():
...     print(line)
In a hole in the ground there lived a
hobbit.

Not a nasty, dirty, wet hole, filled...
```

The **`file`** object has a **`readlines`** method to access the contents of a text file as a **list of string lines**.

Iterating Lines

The **`file`** object
is an **iterable**.

```
>>> for line in file.readlines():
...     print(line)
```

=

```
>>> for line in file:
...     print(line)
```

```
>>> lines = file.readlines():
```

=

```
>>> lines = list(file)
```

Reading Text Files: readline()

```
>>> file = open("the_hobbit.txt")
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>>
```

The **`file`** object has a **`readline`** method to access the contents of a **single line** in a text file.

`readline` also has a parameter **`size`** that defaults to **`EOL`** (End Of Line).

Reading Text Files: The Position

```
>>> file = open("the_hobbit.txt")
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>> print(file.readline())
Not a nasty, dirty, wet hole, filled
with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole
with nothing in it to sit down on or to
eat: it was a hobbit-hole, and that
means comfort.
```

Repeating again the call to `readline` will return the **next line**.

Python keeps track of the **position** of the reading.

We can only read forward!

Reading Text Files: tell()

```
>>> file = open("the_hobbit.txt")
>>> print(file.tell())
0
>>> print(file.readline())
In a hole in the ground there lived a
hobbit.
>>> print(file.tell())
46
```

The **`file`** object has a method **`tell`** that returns the current **position**.

Reading Text Files: seek()

```
>>> file = open("the_hobbit.txt")
>>> print(file.tell())
0
>>> file.seek(46)
>>> print(file.tell())
46
>>> print(file.read(41))
Not a nasty, dirty, wet hole, filled
with
>>> print(file.tell())
87
```

The `file` object has a method **seek** that changes the current **position**.

Closing Files: closed()

```
>>> file = open("the_hobbit.txt")
>>> print(file.closed)
False
>>> file.close()
>>> print(file.closed)
True
>>> print(file.read())
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    print(file.fileno())
ValueError: I/O operation on closed file
```

Files must **always** be closed when we don't need them any more.

The `file` object has a method `close` to close the file.

The file object also has a property `closed` that returns `True` if the file has been closed, and `False` otherwise.

Closing Files: with

```
>>> with open("the_hobbit.txt") as file:
...     print(file.closed)
...
False
>>> print(file.closed)
True
```

The `with` statement will automatically close the file, **even if there is an error exception** occurring in the code.

It is always preferable to open files with the **`with`** statement.

File Output

Digital Career Institute

File Output: write()

```
>>> file.write("Some text")
```

The **`write`** method writes some text into the file.

Writing Text Files

```
>>> with open("todo_list.txt") as file:
...     file.write("My ToDos’ Header")
...
Traceback (most recent call last):
  File "test.py", line 18, in <module>
    file.write("My ToDos’ Header")
io.UnsupportedOperation: not writable
```

Writing to files is, by default, **denied**.

The **open** function, by default, opens files for **reading** purposes **only**.

Opening Mode

The **open** function has a **mode** parameter to indicate which mode should be used when opening the file.

The default value of ***mode*** *is "read a text file".*

```
>>> file = open("the_hobbit.txt")
>>> print(file)
<_io.TextIOWrapper name='test.txt'
mode='r' encoding='UTF-8'>
```

Opening Mode: Reading

```
>>> file = open("the_hobbit.txt")
```

=

```
>>> file = open("the_hobbit.txt", mode="r")
```

Opening a text file for **reading** purposes.

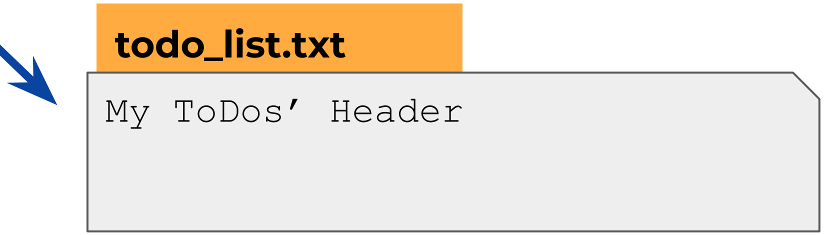
Opening Mode: Writing

```
>>> file = open("the_hobbit.txt", mode="w")
```

Opening a text file for **writing** purposes.

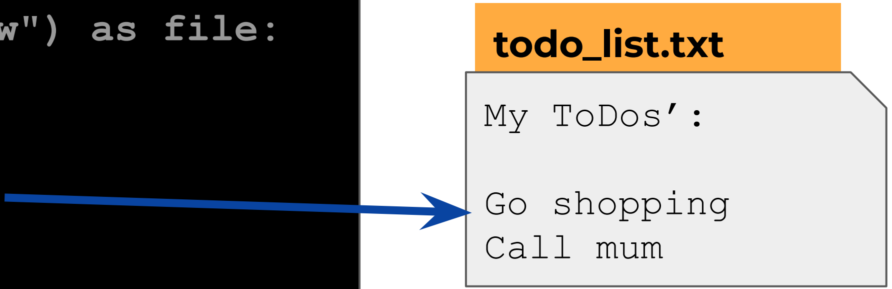
Writing Files

```
>>> with open("todo_list.txt", "w") as file:
...       file.write("My ToDos' Header")
...
>>>
```



Writing Files: writelines()

```
>>> to_dos = ["Go shopping\n", "Call mum\n"]
>>> with open("todo_list.txt", "w") as file:
...     file.write("My ToDos:")
...     file.write("\n\n")
...     file.writelines(to_dos)
...
>>>
```


todo_list.txt

```
My ToDos’:

Go shopping
Call mum
```

The methods **`write`** and **`writeline`** do not add a line break automatically. In this example it is added with **`\n`**.

Other Opening Modes

Digital Career Institute

Opening Modes

```
>>> file = open("todo_list.txt", mode="r")
```

Opening Modes	
r	Reading only
w	Writing only (overwriting)
x	Creating only (fails if already exists)
a	Appending only

r, **w** and **a** accept a modifier **+** to update the file.

Opening Modes

Properties by mode	r	r+	w	w+	a	a+
Can read	✓	✓		✓		✓
Can write		✓	✓	✓	✓	✓
Can write after seek		✓	✓	✓		
Can create			✓	✓	✓	✓
Can truncate			✓	✓		
Starting position	start	start	start	start	end	end

*To **truncate** is to remove all the content of the file without deleting the file.*

Opening Modes

Rules of thumb

r	Only reading is required.
w	Only writing is required.
w+	Both reading and writing are required.

Creating Files

The `x` *mode is used if we only want to write the file when it does not exist.*

```
>>> with open("todo_list.txt", mode="x") as file:
...     file.write("My ToDos:")
```

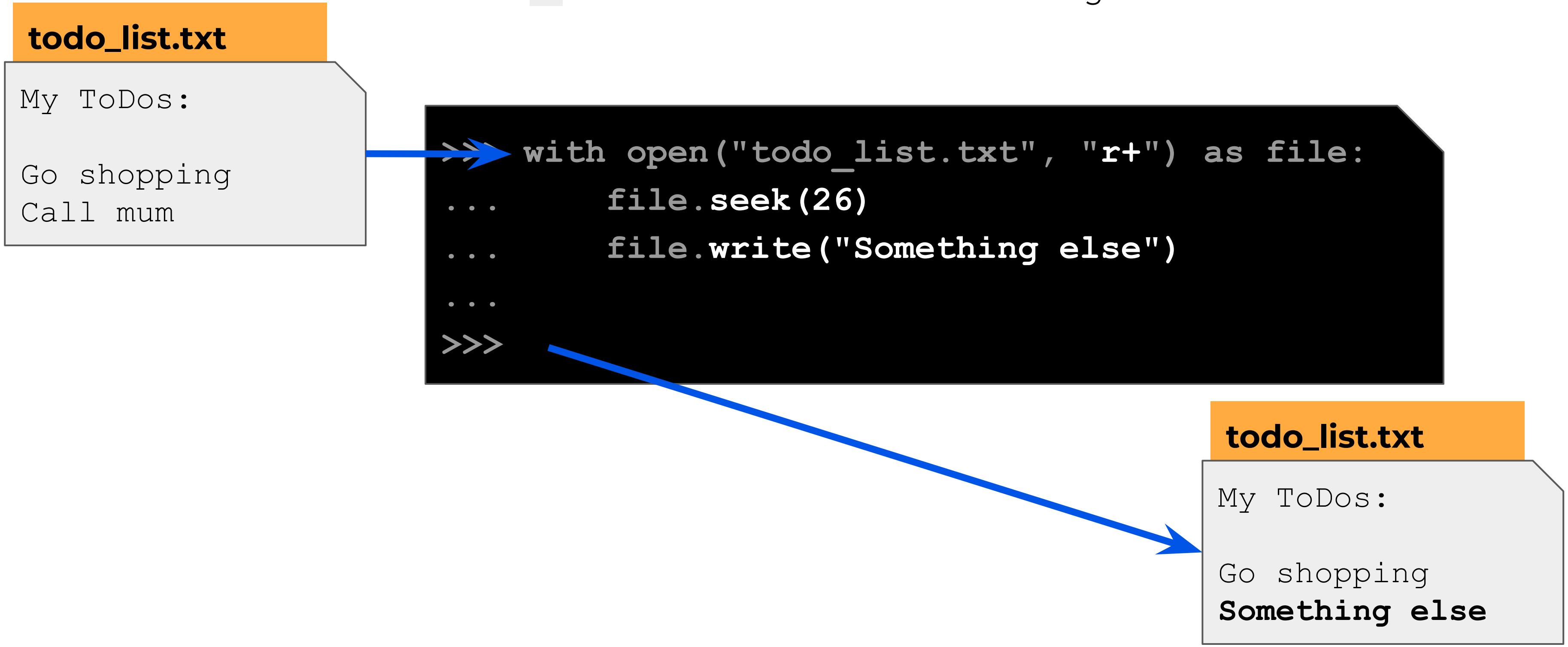
If the file exists raises an exception

Overwrites or creates the file

```
>>> with open("todo_list.txt", mode="w") as file:
...     file.write("My ToDos:")
```

Writing Files

The `r+` will let us overwrite without truncating.



todo_list.txt

```
My ToDos:

Go shopping
Call mum
```

```
>>> with open("todo_list.txt", "r+") as file:
...     file.seek(26)
...     file.write("Something else")
...
>>>
```

todo_list.txt

```
My ToDos:

Go shopping
Something else
```

Writing Files

The `a` *mode will start at the end of the file.*

```
>>> with open("todo_list.txt", "a") as file:
...     print(file.tell())
35
...     file.seek(0)
...     print(file.tell())
0
...     file.write("Something else\n")
...     print(file.tell())
49
>>>
```

todo_list.txt

```
My ToDos:

Go shopping
Call mum
Something else
```

Binary Files

Digital Career Institute

The open() Modes

```
>>> file = open("the_hobbit.txt")
```

=

```
>>> file = open("the_hobbit.txt", mode="r")
```

=

```
>>> file = open("the_hobbit.txt", mode="rt")
```

Format Mode

Opening Mode

Opening a **t**ext file for **r**eading.

Format Modes

```
>>> file = open("test.txt", mode="rt")
```

Opening Modes	
r	Reading only
w	Writing (overwriting)
x	Creating only (fails if already exists)
a	Appending

Format Modes	
t	Text (default)
b	Binary

Text and binary files have the same properties and methods, but they are created differently.

Writing Binary Files

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = "In a hole in the ground
there lived a hobbit.")
...     file.write(content)
...
Traceback (most recent call last):
  File "test.py", line 24, in <module>
    file.write(content)
TypeError: a bytes-like object is required,
not 'str'
```

Normal text (of type **`str`**) cannot be written into binary files.

The file in binary mode requires a **byte-like object**.

Byte-like Objects

DLI

bytes

&

bytearray

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = bytes("In a hole in the ground
there lived a hobbit.\nNot a nasty, dirty, wet
hole, filled with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole with
nothing in it to sit down on or to eat: it was a
hobbit-hole, and that means comfort.", "utf-8")
...     file.write(content)
...
>>>
```

The function `bytes` can also be used to convert the string.

The type `bytes` is the equivalent of the type `str` for binary data.

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = "In a hole in the ground there
lived a hobbit.\nNot a nasty, dirty, wet hole,
filled with the ends of worms and an oozy smell,
nor yet a dry, bare, sandy hole with nothing in it
to sit down on or to eat: it was a hobbit-hole,
and that means comfort.".encode("utf-8")
...     file.write(content)
...
>>>
```

The method **`encode`** can also be used to convert to **`bytes`**.

String to bytes

```
>>> with open("the_hobbit.bin", "wb") as file:
...     content = b"In a hole in the ground
there lived a hobbit.\nNot a nasty, dirty, wet
hole, filled with the ends of worms and an oozy
smell, nor yet a dry, bare, sandy hole with
nothing in it to sit down on or to eat: it was a
hobbit-hole, and that means comfort."
...     file.write(content)
...
>>>
```

Prepending `b` to the string also works.

Bytes to Strings

```
>>> content = b"Hello."
>>> print(type(content))
<class 'bytes'>
>>> print(type(content.decode("utf-8")))
<class 'str'>
```

The reverse (converting a `bytes` object to `str`) can be done with the `decode` method.

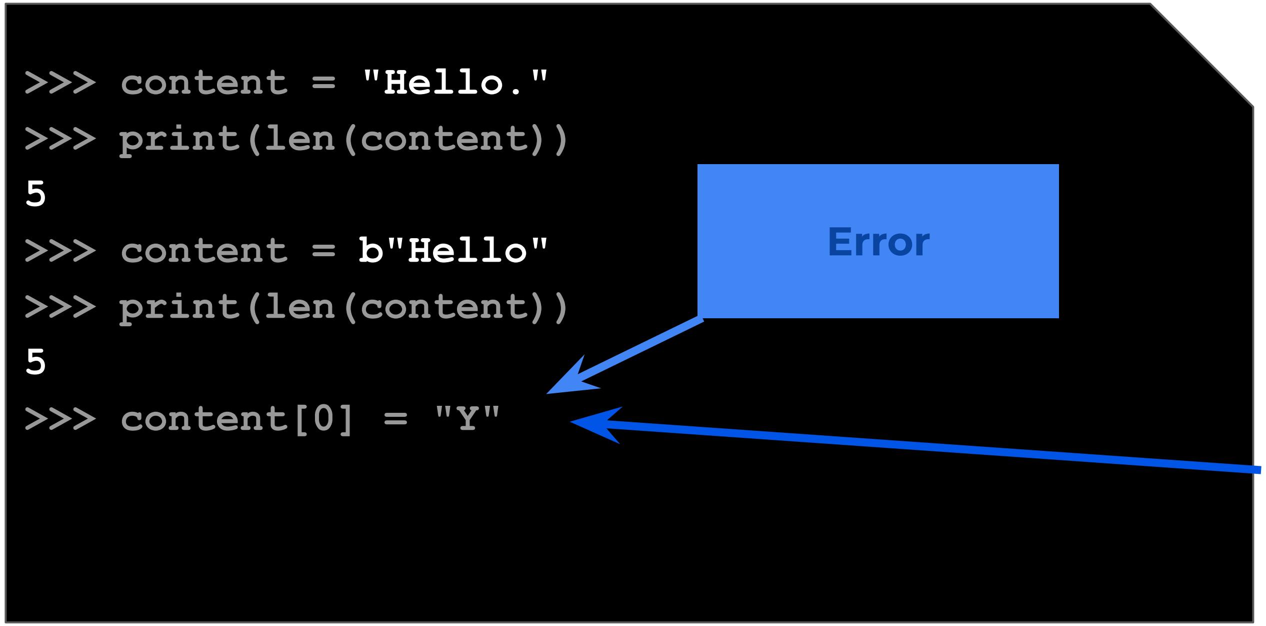
Using the Type bytes

```
>>> content = "Hello."
>>> print(content[0])
H
>>> content = b"Hello"
>>> print(content[0])
72
```

The type **`bytes`**, like **`str`**, is a sequence.

Each item corresponds to a letter but, in this case, is represented as a decimal integer.

Using the Type bytes



```
>>> content = "Hello."
>>> print(len(content))
5
>>> content = b"Hello"
>>> print(len(content))
5
>>> content[0] = "Y"
```

As sequences, they both can use similar functions.

The **`bytes`** type is also immutable.

The Type bytearray

A **`bytearray`** *is a mutable* **`bytes`***.*

```
>>> content = bytearray("Hello", "utf-8")
>>> print(type(content))
<class 'bytearray'>
```

The type **`bytearray`** is the equivalent of the type **`list`** for binary data.

Using the Type bytearray

```
>>> content = bytearray(b"Hello")
>>> print(content.decode("utf-8"))
Hello
>>> content[0:1] = b"Y"
>>> content.append(ord("w"))
>>> print(content.decode("utf-8"))
Yellow
```

Like in a `list`, we can change one of the items by specifying an index.

`ord` returns the decimal that represents the letter `w` and is being appended to the `bytearray`.

Summary of Data Types

	Immutable	*Mutable*
text	**str**	**list**
binary	**bytes**	**bytearray**