11. Context manager. Work with files

Python open function

Built-in function ***open()***

```
def open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True):
    pass
```

Text file example:

```
  GNU nano 4.8                          my_file.txt
Now I can see a text inside this file
Hello world
```

Open file and return a corresponding file object. If the file cannot be opened, an **OSError** is raised.

```
my_file = open("my_file.txt")
print(my_file)
```

```
<_io.TextIOWrapper name='my_file.txt' mode='r' encoding='UTF-8'>
```

File object has a **read()** method:

```
my_file = open("my_file.txt")
print(my_file.read())
```

```
Now I can see a text inside this file
Hello world
```

Python read and readline methods

***read()** method:*

```
    @abstractmethod
    def read(self, n: int = -1) -> AnyStr:
        pass
```

read() method with argument (**int** type) allow you specify how many *characters* you want to return:

```
my_file = open("my_file.txt")
print(my_file.read(3))
```

readline() *method:*

```
    @abstractmethod
    def readline(self, limit: int = -1) -> AnyStr:
        pass
```

readline() method with argument (**int** type) have the same logic as **read()**

read() allow you to get FULL text. r**eadline()** allow you to get text line by line

```
my_file = open("my_file.txt")
print(my_file.readline())
```

```
Now I can see a text inside this file
```

```
my_file = open("my_file.txt")
print(my_file.read())
```

```
Now I can see a text inside this file
Hello world
```

Looping through the lines

Looping through the whole file*:*

```
my_file = open("my_file.txt")
for line in my_file:
    print(line)
```

```
Now I can see a text inside this file

Hello world

```

```
my_file = open("my_file.txt")
for line in my_file:
    print(line, end='')
```

```
Now I can see a text inside this file
Hello world
```

Looping with **readline()** method:

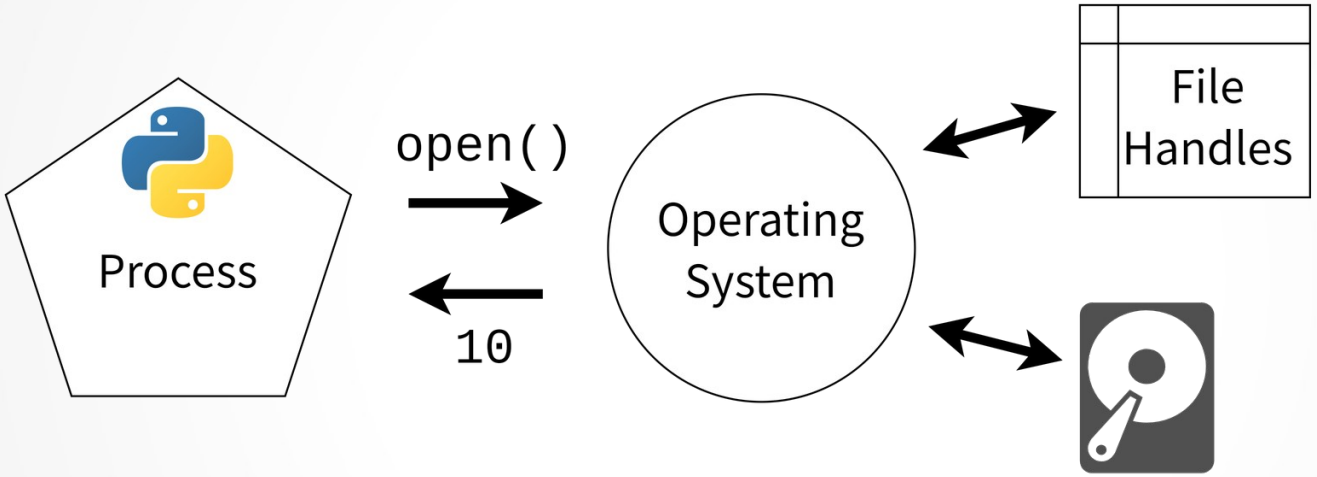
```
my_file = open("my_file.txt")
line = my_file.readline()
while line:
    print(line, end='')
    line = my_file.readline()
```

Python close function

Call **close()** to close the file and immediately free up any system resources used by it.

```
my_file = open("my_file.txt")
line = my_file.readline()
my_file.close()
```

Why we need to close the files?



When you open a file with **open()**, you make a system call to the operating system to locate that file on the hard drive and prepare it for reading or writing. The operating system will then return an **unsigned integer** called a file handle on Windows and a file descriptor on UNIX-like systems, including Linux and macOS:

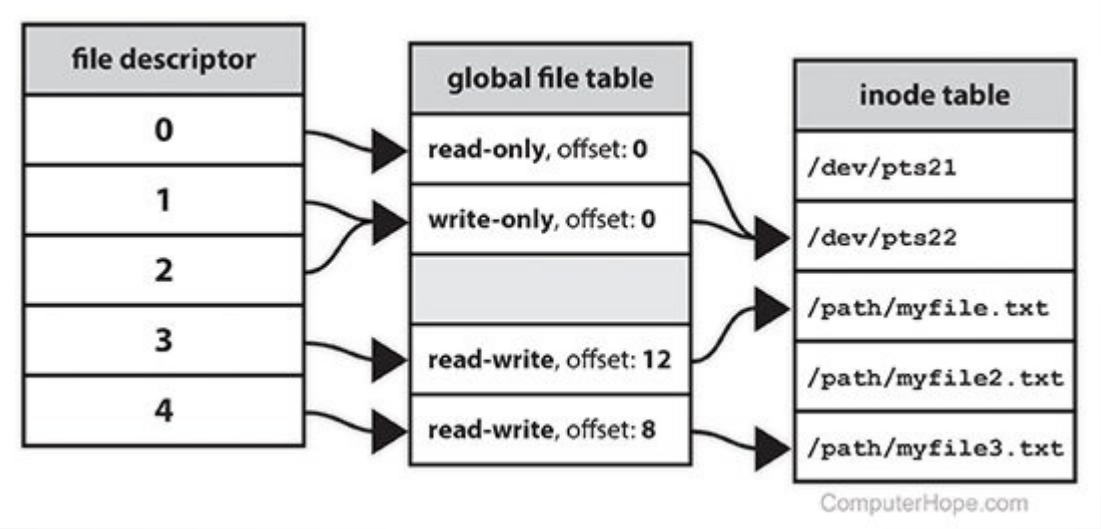
Once you have the number associated with the file, you're ready to do read or write operations. Whenever Python wants to read, write, or close the file, it'll make another system call, providing the file handle number.

File descriptors

The Python file object has a **fileno()** method that you can use to find the file handle/descriptor:

```
my_file = open("my_file.txt")
print(my_file.fileno())
```

When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream (read-only, write-only, etc.).



File close

What if some Exception happened?

```
my_file = open("my_file.txt")
raise Exception()
my_file.close()
print('Is file closed?')
```

Better way to close file

```
my_file = open("my_file.txt")
try:
    print(my_file.read())
finally:
    my_file.close()
    print('Is file closed?')
```

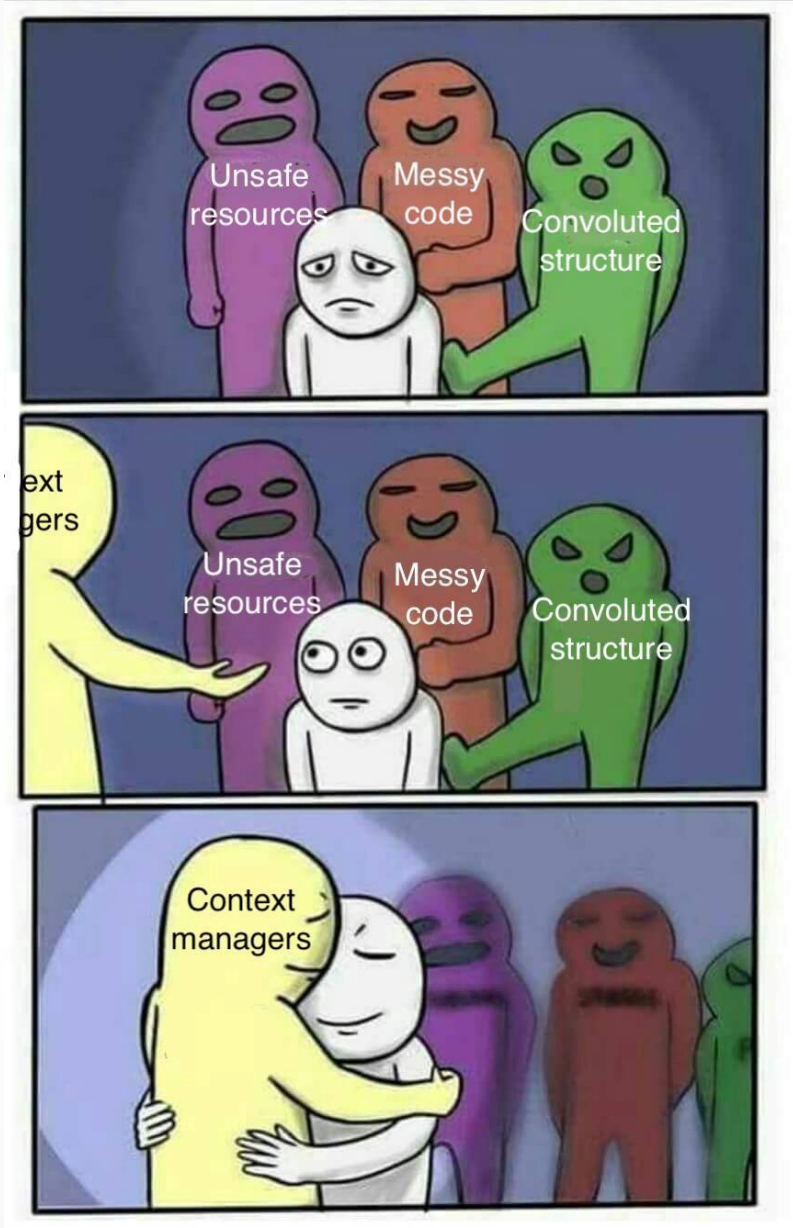
Context manager

Context managers allow you to **allocate** and **release** resources precisely when you want to.

```
my_file = open("my_file.txt")
try:
    print(my_file.read())
finally:
    my_file.close()
    print('Is file closed?')
```

Code above is the same as bellow:

```
with open("my_file.txt") as my_file:
    print(my_file.read())
```



Context manager

Classic structure of context manager:

```
class ContextManager():
    def __init__(self):
        print('init method called')

    def __enter__(self):
        print('enter method called')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('exit method called')


with ContextManager() as manager:
    print('with statement called')
```

```
init method called
enter method called
with statement called
exit method called
```

Context manager

open() is also context manager. Example of the **open()** implementation

```
class custom_open_file:
    def __init__(self, filename, mode='r'):
        self.file = open(filename, mode)
        print('init method called')

    def __enter__(self):
        print('enter method called')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.file.close()
        print('exit method called')


with custom_open_file('my_file.txt') as my_file:
    print(my_file.read())
```

What should we use?

```
my_file = open("my_file.txt")
print(my_file.read())
my_file.close()
```

```
my_file = open("my_file.txt")
try:
    print(my_file.read())
finally:
    my_file.close()
```

```
with open('my_file.txt', 'r') as my_file:
    print(my_file.read())
```

File modes

```
def open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True):
    pass
```

Possible file modes:

Mode	Meaning
'r'	Reading (default)
'w'	Writing
'a'	Appending
'b'	Binary data
'+'	Updating (reading and writing)
'x'	Exclusive creation, failing if file exists

Should we use default mode or we can skip it?

```
with open('my_file.txt', 'r') as my_file:
    print(my_file.read())
```

Remember 2-nd rule of Python Zen:

Explicit is better than implicit.

File modes (multiple modes)

You can do read and write operations into the file with mode **‘+’**

```
with open('my_file.txt', 'w+') as my_file:
    my_file.write('Hello world\n')
    my_file.write('Ok')
```

REMEMBER: You cursor will be at the end of file (**a+** mode) or file will be truncated (**w+** mode).
You can change your cursor position with a **seek()** method:

```
with open('./my_file.txt', 'a+') as my_file:
    my_file.seek(0)
    print(my_file.read())
    my_file.write('Begin\n')
```

To check the current position you could use **tell()** method:

```
with open('./my_file.txt', 'a+') as my_file:
    my_file.seek(0)
    print(my_file.read())
    print(my_file.tell())
    my_file.write('Begin\n')
```

File modes (delete)

You can delete the file with using module **os**. If the file doesn't exist you retrieve an error

```
import os
os.remove("my_file.txt")
```

To avoid getting an error, you could check if file exists with **os.path.exists()**

```
import os

if os.path.exists("my_file.txt"):
    os.remove("my_file.txt")
else:
    print("The file does not exist")
```

To delete an entire folder, use the **os.rmdir()** method:

```
import os
os.rmdir("my_folder")
```

CSV module

Example of csv file with ‘,’ delimiter:

```
a,b,c
1,2,3
```

Read such file:

```
import csv
with open('file.csv', newline='') as csvfile:
    reader = csv.reader(csvfile, delimiter=',')
    for row in reader:
        print(', '.join(row))
```

Write into the file:

```
import csv
with open('file.csv', 'w') as csvfile:
    writer = csv.writer(csvfile, delimiter=',')
    writer.writerow(['a', 'b', 'c'])
    writer.writerow([3, 4, 5])
```

JSON

Convert in JSON object

```
import json
personal_info = {
    'name': 'Oleksii',
    'age': 28,
    'languages': ['English', 'Ukrainian']
}
json_data = json.dumps(personal_info)
print(json_data)
```

To work with data from JSON you should **loads()** it first:

```
import json
personal_info = {
    'name': 'Oleksii',
    'age': 28,
    'languages': ['English', 'Ukrainian']
}
json_data = json.dumps(personal_info)

python_data = json.loads(json_data)
print(python_data['languages'])
```

JSON dump and load into file

To save data as json file you could use **dump()** method from **json** module:

```
def dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True,
        allow_nan=True, cls=None, indent=None, separators=None,
        default=None, sort_keys=False, **kw):
```

Example:

```
with open('example.json', 'w') as file:
    json.dump(personal_info, file, indent=4)
```

To load data from json file you could use **load()** method from **json** module:

```
def load(fp, *, cls=None, object_hook=None, parse_float=None,
        parse_int=None, parse_constant=None, object_pairs_hook=None, **kw):
```

Example:

```
with open('example.json', 'r') as file:
    data_from_json = json.load(file)
    print(data_from_json)
```