Documentation for OS

(operating system).

Introduction about operating system:

- ✓ An **operating system** (**OS**) is system software that manages computer hardware , software resources, and provides common service for computer program .
  Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting software for cost allocation of processor time, mass storage , printing, and other resources.
- ✓ For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and frequently makes system calls to an OS function or is interrupted by it. Operating systems are found on many devices that contain a computer – from cellular phones and video game consoles to web services and supercomputers.
- ✓ Some operating systems require installation or may come pre-installed with purchased computers (OME-installation), whereas others may run directly from media (i.e. live CD) or flash memory (i.e. USB stick).

simple Clarify about our project:

i. Design and implement a basic shell interface that supports the execution of a series of commands. The shell should be robust (e.g., it should not crash under any circumstance)

ii. In this project you will implement a Mini-FAT simple file system. Your file system will be able to allow users to browse the directory structure, create and delete new files and directories, etc

iii. Purpose : The purpose of this project is to familiarize you with Shell and File system

The steps is to implement the project:

- At this point, the prompt should indicate that the shell is ready to accept input from the user. It also shows useful information, such as the current directory.
- For now, you just need to implement a simple prompt.
- The prompt should look like the following:
    - H: \>
- Before the shell can begin executing commands, it needs to extract the command name and the arguments into “tokens”.
- In the shell, the first token will always be the name of the command we wish to execute, and all remaining tokens will be arguments to that command.
- Once the shell understands what commands to execute it is time to implement the execution of simple commands.
- The shell must support the following internal commands:
    1. cd - Change the current default directory to . If the argument is not present, report the current directory. If the directory does not exist an appropriate error should be reported.
    2. cls - Clear the screen.
    3. dir - List the contents of directory .
    4. quit - Quit the shell.
    5. copy - Copies one or more files to another location
    6. del - Deletes one or more files.
    7. help -Provides Help information for commands.
    8. md - Creates a directory.
    9. rd- Removes a directory.
    10. rename - Renames a file.
    11. type - Displays the contents of a text file.
    12. import – import text file(s) from your computer
    13. export – export text file(s) to your computer

Implementation with code:

writing a simple file system which will use a regular file as the "virtual disk". The structure of the file system is based on FAT, simplified, and called mini-FAT

- Mini-FAT description:-

  The mini-FAT file system makes the following simplifications to FAT:

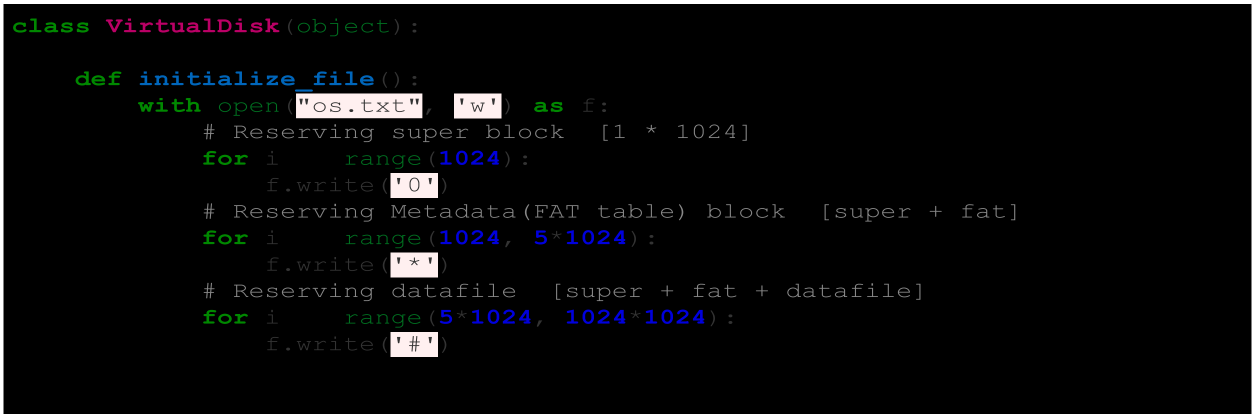
  - ✓ There are 1024 bytes in a cluster, and 1024 clusters in the file system.
  - ✓ The super-block occupies cluster 0 and has no purpose. It can contain all zeros.
  - ✓ The meta-data region, which consists only of the FAT table. The FAT table is an array of 32 bit integers, indexed by cluster numbers, 0 through 1023. The table occupies clusters 1 through 4

  - ○ Creation class with name (virtual disk)that have size 1024*1024 (1MB) blocks separating the file into three parts

    I. super block that take first block with value (0)

    II. mini fat that take from second blocks to fifth blocks 4*1024 with value (*)

    III. datafile that take the rest of the blocks that are empty 1019*1024 with take value (#)



```
class VirtualDisk(object):

    def initialize_file():
        with open("os.txt", 'w') as f:
            # Reserving super block  [1 * 1024]
            for i    range(1024):
                f.write('0')
            # Reserving Metadata(FAT table) block  [super + fat]
            for i    range(1024, 5*1024):
                f.write('*')
            # Reserving datafile  [super + fat + datafile]
            for i    range(5*1024, 1024*1024):
                f.write('#')
```

- Creation class with name (Mini fat) that have size 4*1024
  - ✓ Inside this class there is eight methods

I. Initialize :

create constructor to initiate size fat table(1024).
creates our fat table in a type of data structure called Data frames to store as dictionary.

II. write fat table :

Firstly skip first block (super block)
Secondly convert list of int TO list of bytes.
Finally Store data frame in file .

III. get fat table:

Return data frame .

IV. Print fat:

Print fat table for debugging or testing.

V. get available block:

Get first available block in data frame that take zero value.

VI. get available blocks:

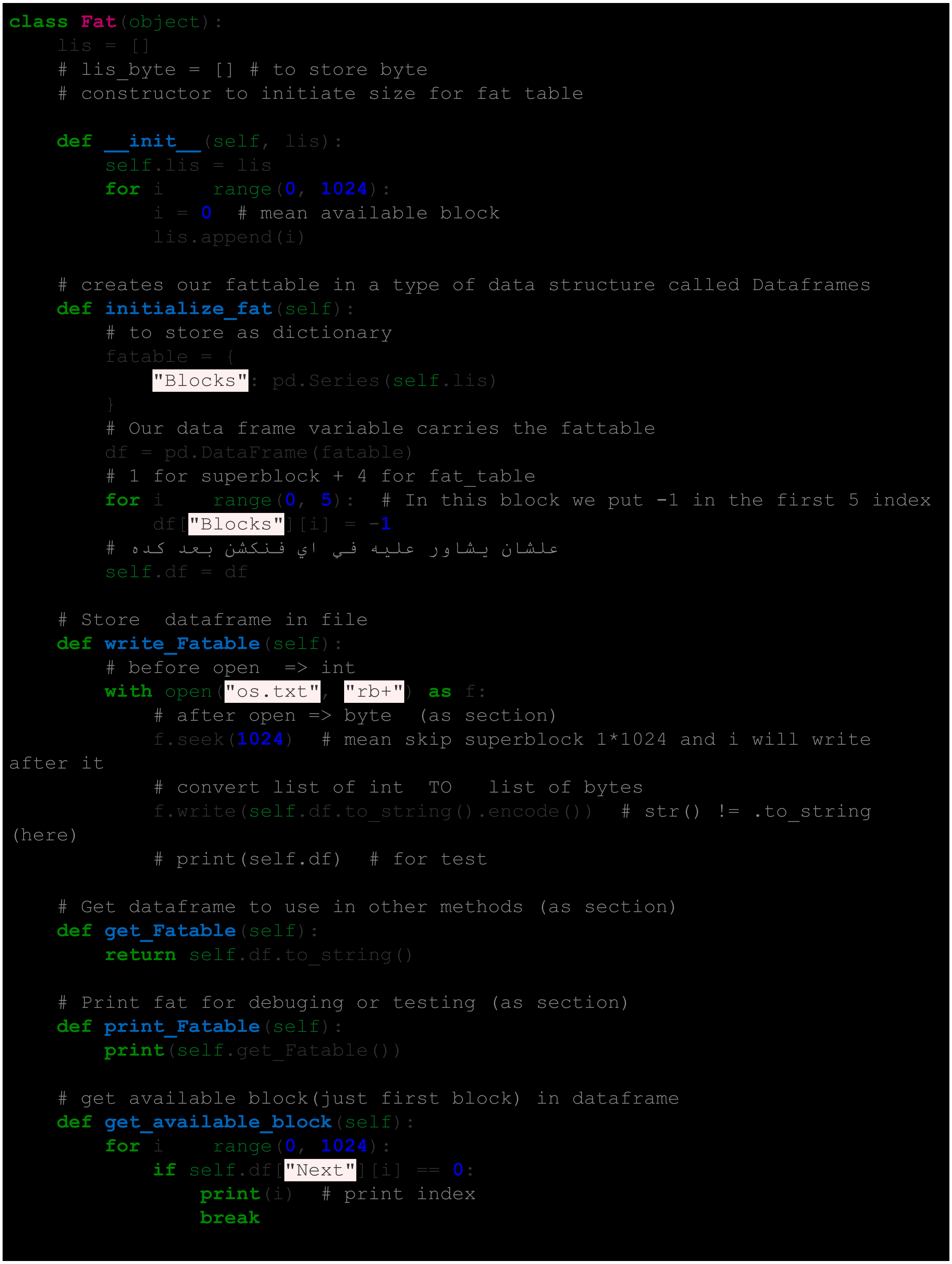
Get all available blocks.

VII. get next:

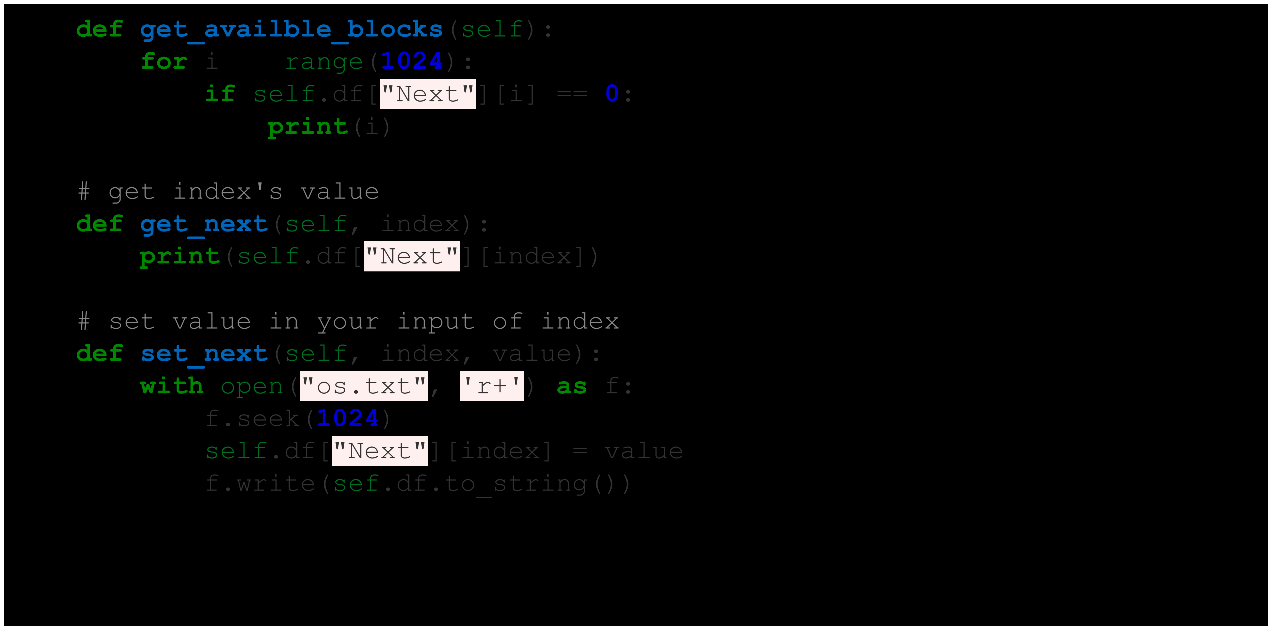
Pass the index and return the value that opposite it.

VIII. set next:

set value in your input of index.



```
class Fat(object):
    lis = []
    # lis_byte = [] # to store byte
    # constructor to initiate size for fat table

    def __init__(self, lis):
        self.lis = lis
        for i    range(0, 1024):
            i = 0  # mean available block
            lis.append(i)

    # creates our fattable in a type of data structure called Dataframes
    def initialize_fat(self):
        # to store as dictionary
        fatable = {
            "Blocks": pd.Series(self.lis)
        }
        # Our data frame variable carries the fattable
        df = pd.DataFrame(fatable)
        # 1 for superblock + 4 for fat_table
        for i    range(0, 5):  # In this block we put -1 in the first 5 index
            df["Blocks"][i] = -1
        # علشان يشاور عليه في اي فنكشن بعد كده
        self.df = df

    # Store  dataframe in file
    def write_Fatable(self):
        # before open  => int
        with open("os.txt", "rb+") as f:
            # after open => byte  (as section)
            f.seek(1024)  # mean skip superblock 1*1024 and i will write
after it
            # convert list of int  TO  list of bytes
            f.write(self.df.to_string().encode())  # str() != .to_string
(here)
            # print(self.df)  # for test

    # Get dataframe to use in other methods (as section)
    def get_Fatable(self):
        return self.df.to_string()

    # Print fat for debuging or testing (as section)
    def print_Fatable(self):
        print(self.get_Fatable())

    # get available block(just first block) in dataframe
    def get_available_block(self):
        for i    range(0, 1024):
            if self.df["Next"][i] == 0:
                print(i)  # print index
                break
```



```
def get_availble_blocks(self):
    for i    range(1024):
        if self.df["Next"][i] == 0:
            print(i)

# get index's value
def get_next(self, index):
    print(self.df["Next"][index])

# set value in your input of index
def set_next(self, index, value):
    with open("os.txt", 'r+') as f:
        f.seek(1024)
        self.df["Next"][index] = value
        f.write(sef.df.to_string())
```

Creation a class with name (directory entry):

That is responsible for collect all attribute about the file that it’s track.

- Directory Entry:
  - ✓ Constructor ():

    The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created.

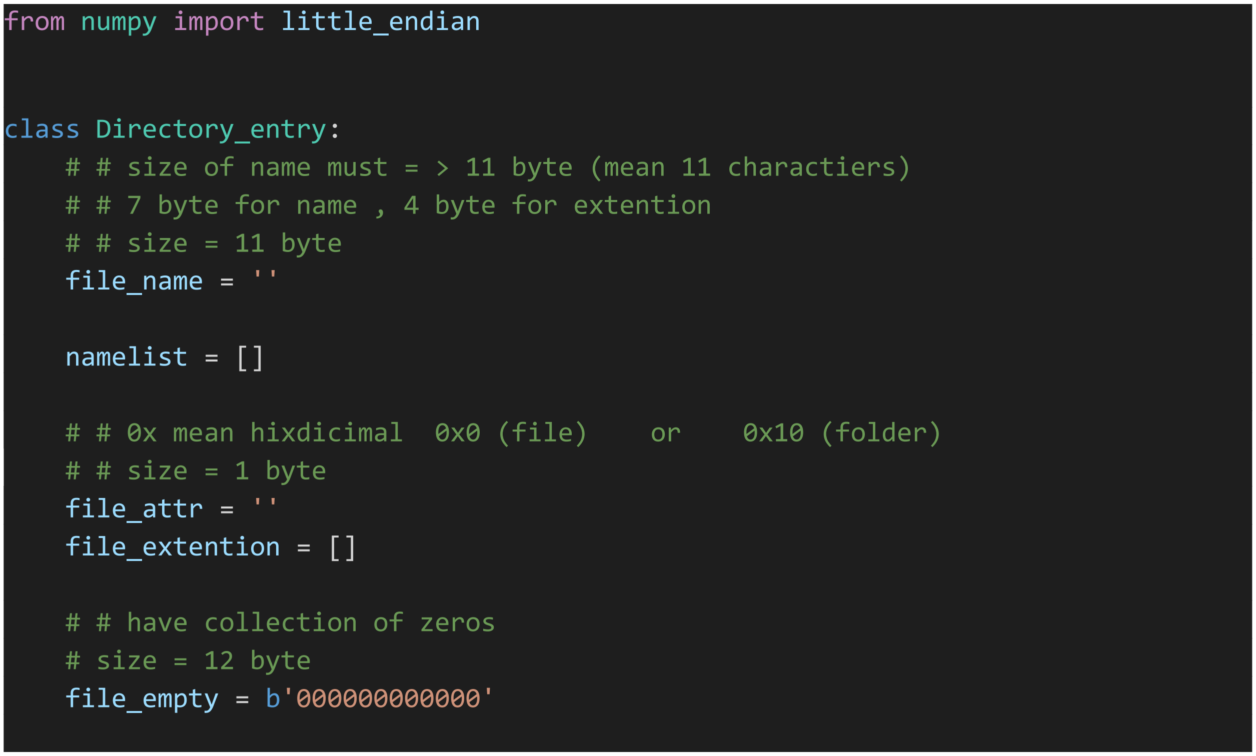
    [file name, file attribute, file size, file empty, first cluster].

    Methods:
  - ✓ Check type:
  - Is responsible for determine is file or folder.
  - If value of attribute = 0x0, that mean file.
  - If value of attribute = 0x10, that mean folder.
  - And we here also size of file, if value of name = 11 char of bytes, that mean and this is what is required.
  - ✓ Get bytes:

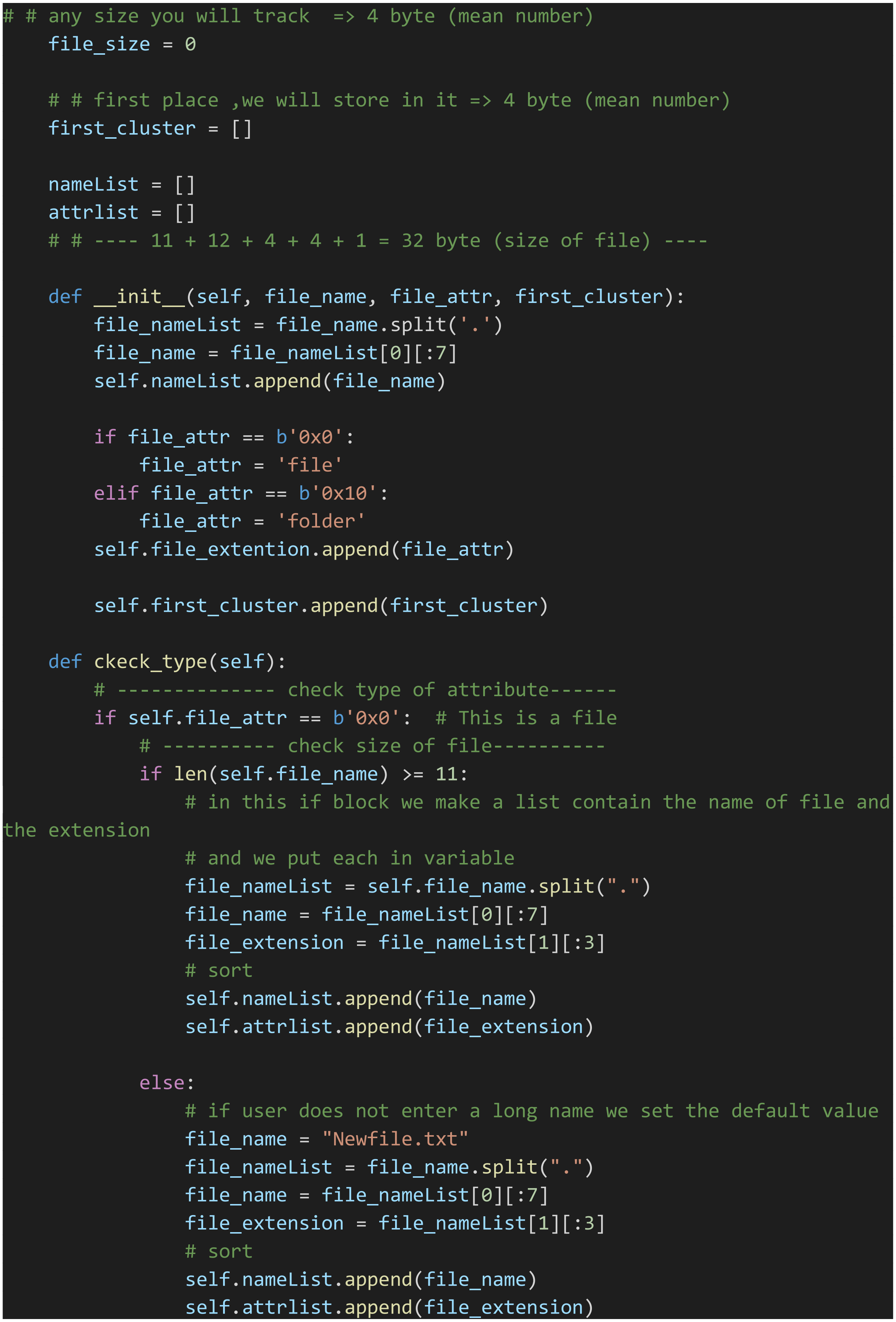
- Take Directory entry as parameter.
- Return Array of bytes, that have all records of directory table as byte array.

✓ Get directory:

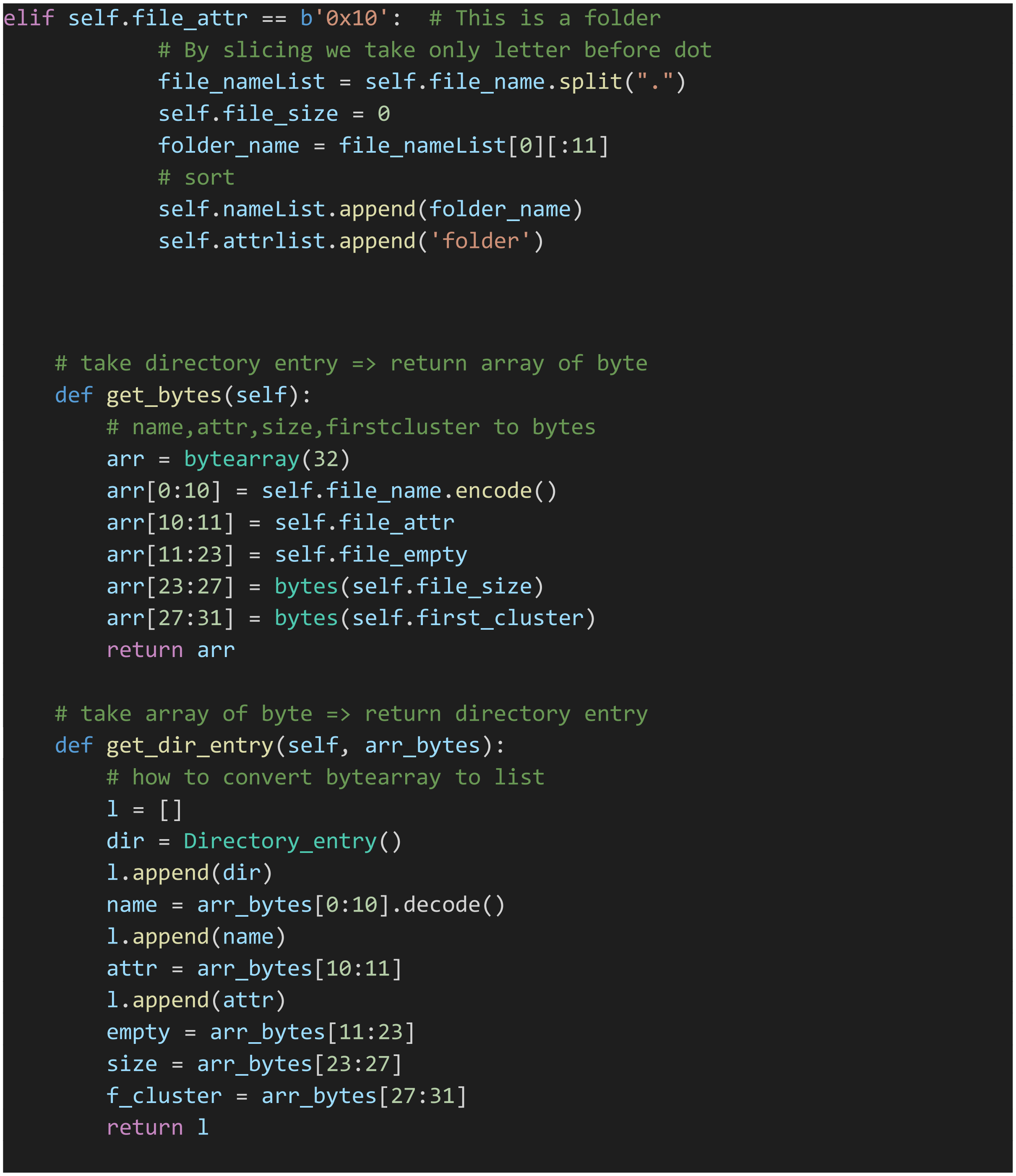
- Take Array of byes as parameter.
- Return Directory entry, that have all records of array of byte, but we convert as string to sorted as records.



```
from numpy import little_endian


class Directory_entry:
    # # size of name must = > 11 byte (mean 11 charactiers)
    # # 7 byte for name , 4 byte for extention
    # # size = 11 byte
    file_name = ''

    namelist = []

    # # 0x mean hixdicimal  0x0 (file)    or    0x10 (folder)
    # # size = 1 byte
    file_attr = ''
    file_extention = []

    # # have collection of zeros
    # size = 12 byte
    file_empty = b'000000000000'
```



```
# # any size you will track  => 4 byte (mean number)
    file_size = 0

    # # first place ,we will store in it => 4 byte (mean number)
    first_cluster = []

    nameList = []
    attrlist = []
    # # ---- 11 + 12 + 4 + 4 + 1 = 32 byte (size of file) ----

    def __init__(self, file_name, file_attr, first_cluster):
        file_nameList = file_name.split('.')
        file_name = file_nameList[0][:7]
        self.nameList.append(file_name)

        if file_attr == b'0x0':
            file_attr = 'file'
        elif file_attr == b'0x10':
            file_attr = 'folder'
        self.file_extention.append(file_attr)

        self.first_cluster.append(first_cluster)

    def ckeck_type(self):
        # -------------- check type of attribute------
        if self.file_attr == b'0x0':  # This is a file
            # ---------- check size of file----------
            if len(self.file_name) >= 11:
                # in this if block we make a list contain the name of file and
the extension
                # and we put each in variable
                file_nameList = self.file_name.split(".")
                file_name = file_nameList[0][:7]
                file_extension = file_nameList[1][:3]
                # sort
                self.nameList.append(file_name)
                self.attrlist.append(file_extension)

            else:
                # if user does not enter a long name we set the default value
                file_name = "Newfile.txt"
                file_nameList = file_name.split(".")
                file_name = file_nameList[0][:7]
                file_extension = file_nameList[1][:3]
                # sort
                self.nameList.append(file_name)
                self.attrlist.append(file_extension)
```



```
elif self.file_attr == b'0x10':  # This is a folder
            # By slicing we take only letter before dot
            file_nameList = self.file_name.split(".")
            self.file_size = 0
            folder_name = file_nameList[0][:11]
            # sort
            self.nameList.append(folder_name)
            self.attrlist.append('folder')



    # take directory entry => return array of byte
    def get_bytes(self):
        # name,attr,size,firstcluster to bytes
        arr = bytearray(32)
        arr[0:10] = self.file_name.encode()
        arr[10:11] = self.file_attr
        arr[11:23] = self.file_empty
        arr[23:27] = bytes(self.file_size)
        arr[27:31] = bytes(self.first_cluster)
        return arr

    # take array of byte => return directory entry
    def get_dir_entry(self, arr_bytes):
        # how to convert bytearray to list
        l = []
        dir = Directory_entry()
        l.append(dir)
        name = arr_bytes[0:10].decode()
        l.append(name)
        attr = arr_bytes[10:11]
        l.append(attr)
        empty = arr_bytes[11:23]
        size = arr_bytes[23:27]
        f_cluster = arr_bytes[27:31]
        return l
```

- Directory:
  - Constructor

    The task of constructors is to initialize (assign values) to the data members of the class when an object of the class is created.
    [file name, file attribute, first cluster].
  - Dir table
    - Take the values from constructor and sort in data frame.
    - Sort date entry as bytes .
    - Take the values from constructors and convert it to byte and sort in data frame.
  - Write Directory:

    Write directory is store direct tables in fat table as cluster value.
  - Read dire

    Get directory.
  - Search directory

    Take file name as parameter and return index after search in data frame.
  - Read file name

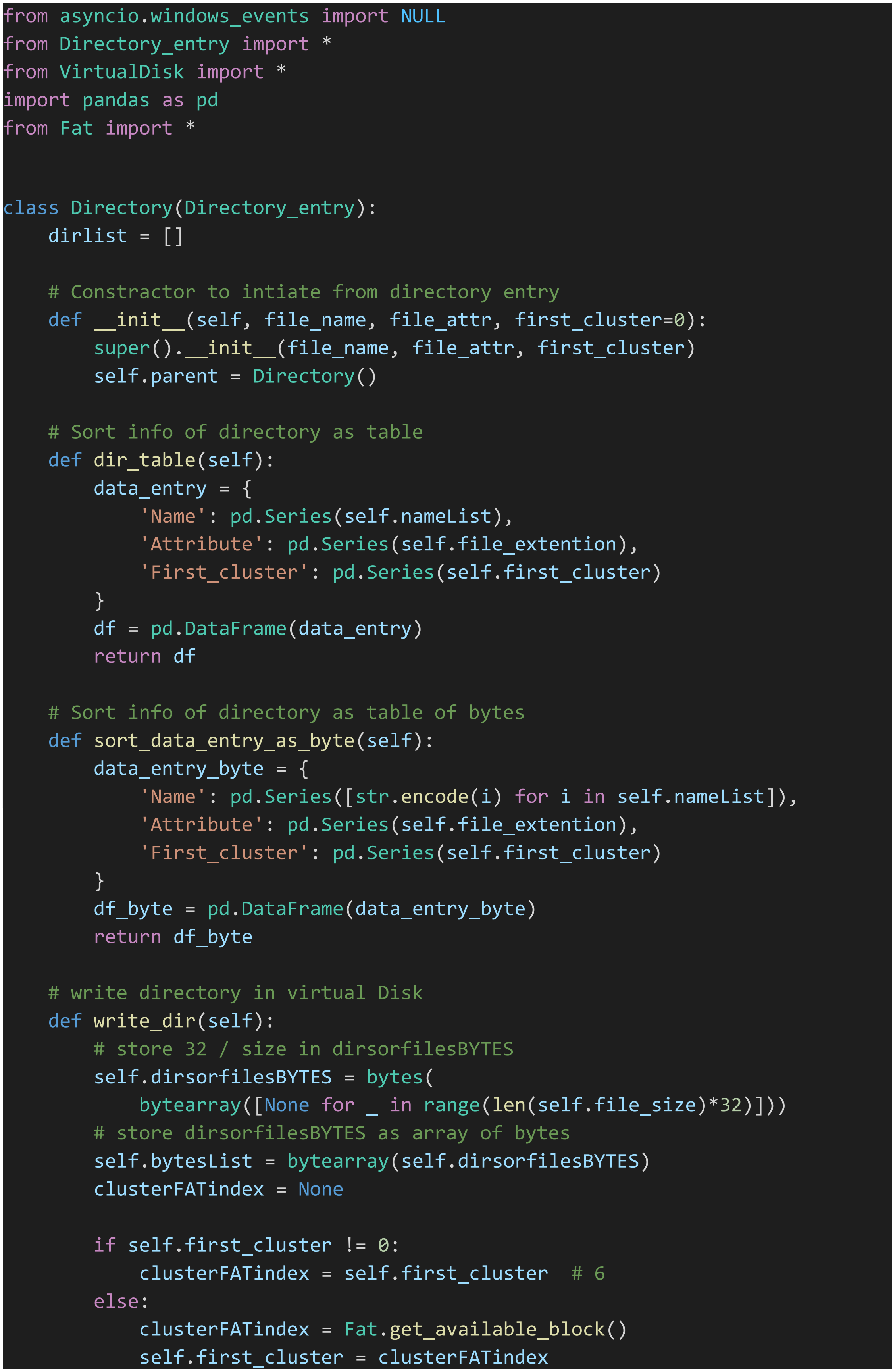
    Return list of names from directory table.
  - Update content
    1. read directory table .
    2. search in it by file name as parameter .
    3. if found it remove record and store a new one.
  - Clear directory size

    Clear directory size from all index cluster of fat by first cluster .

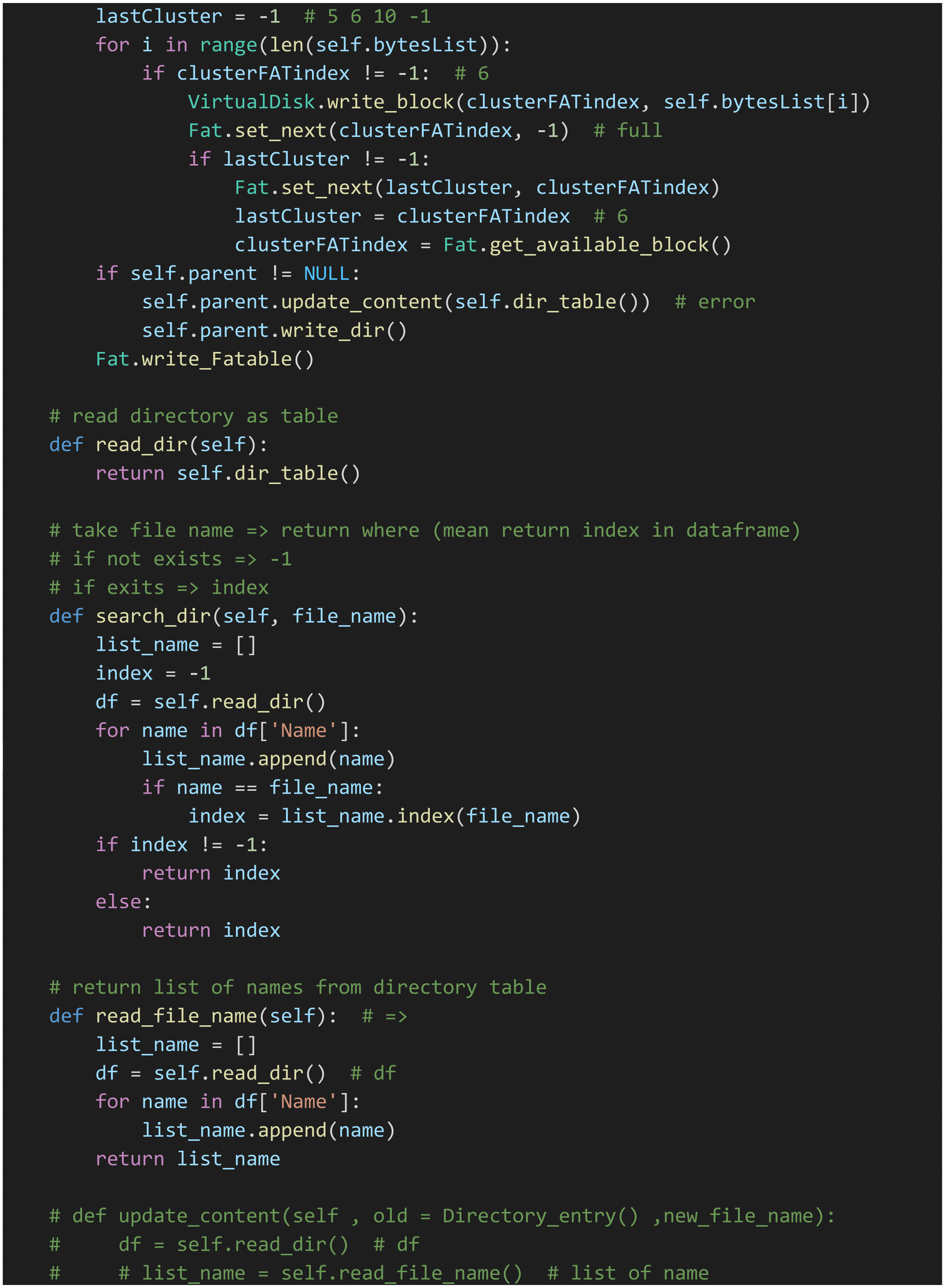
(if first cluster = 0 that mean empty, -1 mean full size storge n number that mean next cluster).

- ✓ Delete

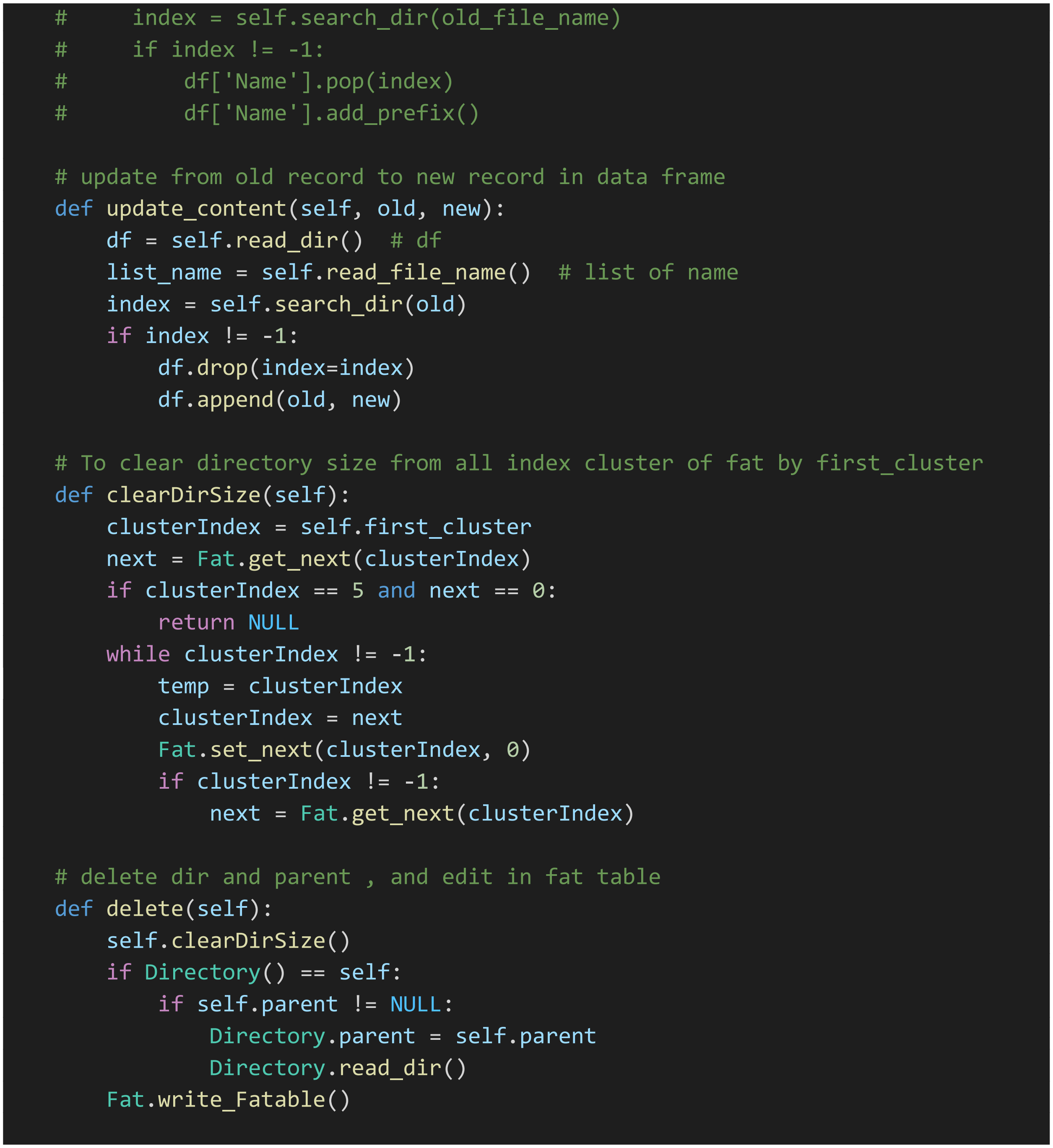
  before clear directory size mean all directory we have remove directories child if exist and write fat table after clear.



```python
from asyncio.windows_events import NULL
from Directory_entry import *
from VirtualDisk import *
import pandas as pd
from Fat import *


class Directory(Directory_entry):
    dirlist = []

    # Constractor to intiate from directory entry
    def __init__(self, file_name, file_attr, first_cluster=0):
        super().__init__(file_name, file_attr, first_cluster)
        self.parent = Directory()

    # Sort info of directory as table
    def dir_table(self):
        data_entry = {
            'Name': pd.Series(self.nameList),
            'Attribute': pd.Series(self.file_extention),
            'First_cluster': pd.Series(self.first_cluster)
        }
        df = pd.DataFrame(data_entry)
        return df

    # Sort info of directory as table of bytes
    def sort_data_entry_as_byte(self):
        data_entry_byte = {
            'Name': pd.Series([str.encode(i) for i in self.nameList]),
            'Attribute': pd.Series(self.file_extention),
            'First_cluster': pd.Series(self.first_cluster)
        }
        df_byte = pd.DataFrame(data_entry_byte)
        return df_byte

    # write directory in virtual Disk
    def write_dir(self):
        # store 32 / size in dirsorfilesBYTES
        self.dirsorfilesBYTES = bytes(
            bytearray([None for _ in range(len(self.file_size)*32)]))
        # store dirsorfilesBYTES as array of bytes
        self.bytesList = bytearray(self.dirsorfilesBYTES)
        clusterFATindex = None

        if self.first_cluster != 0:
            clusterFATindex = self.first_cluster  # 6
        else:
            clusterFATindex = Fat.get_available_block()
            self.first_cluster = clusterFATindex
```



```
        lastCluster = -1  # 5 6 10 -1
        for i in range(len(self.bytesList)):
            if clusterFATindex != -1:  # 6
                VirtualDisk.write_block(clusterFATindex, self.bytesList[i])
                Fat.set_next(clusterFATindex, -1)  # full
                if lastCluster != -1:
                    Fat.set_next(lastCluster, clusterFATindex)
                    lastCluster = clusterFATindex  # 6
                    clusterFATindex = Fat.get_available_block()
        if self.parent != NULL:
            self.parent.update_content(self.dir_table())  # error
            self.parent.write_dir()
        Fat.write_Fatable()

    # read directory as table
    def read_dir(self):
        return self.dir_table()

    # take file name => return where (mean return index in dataframe)
    # if not exists => -1
    # if exits => index
    def search_dir(self, file_name):
        list_name = []
        index = -1
        df = self.read_dir()
        for name in df['Name']:
            list_name.append(name)
            if name == file_name:
                index = list_name.index(file_name)
        if index != -1:
            return index
        else:
            return index

    # return list of names from directory table
    def read_file_name(self):  # =>
        list_name = []
        df = self.read_dir()  # df
        for name in df['Name']:
            list_name.append(name)
        return list_name

    # def update_content(self , old = Directory_entry() ,new_file_name):
    #     df = self.read_dir()  # df
    #     # list_name = self.read_file_name()  # list of name
```



```python
#     index = self.search_dir(old_file_name)
#     if index != -1:
#         df['Name'].pop(index)
#         df['Name'].add_prefix()

# update from old record to new record in data frame
def update_content(self, old, new):
    df = self.read_dir()  # df
    list_name = self.read_file_name()  # list of name
    index = self.search_dir(old)
    if index != -1:
        df.drop(index=index)
        df.append(old, new)

# To clear directory size from all index cluster of fat by first_cluster
def clearDirSize(self):
    clusterIndex = self.first_cluster
    next = Fat.get_next(clusterIndex)
    if clusterIndex == 5 and next == 0:
        return NULL
    while clusterIndex != -1:
        temp = clusterIndex
        clusterIndex = next
        Fat.set_next(clusterIndex, 0)
        if clusterIndex != -1:
            next = Fat.get_next(clusterIndex)

# delete dir and parent , and edit in fat table
def delete(self):
    self.clearDirSize()
    if Directory() == self:
        if self.parent != NULL:
            Directory.parent = self.parent
            Directory.read_dir()
    Fat.write_Fatable()
```