# pyfatfs

*Release 1.1.0*

**Nathan-J. Hirschauer <nathanhi <at> deepserve.info**

**Mar 04, 2024**

# CONTENTS


**1 DosDateTime** **3**

**2 EightDotThree** **5**

**3 FATDirectoryEntry** **7**

**4 FatIO** **15**

**5 PyFat** **17**

**6 PyFatFS** **21**

**7 PyFatFSOpener** **25**

**8 Exceptions** **27**

**9 Contributing to pyfat** **29**

**10 Getting started** **31**

10.1 PyFilesystem2 quickstart . . . 31

10.1.1 Parameters . . . 31

10.1.1.1 encoding . . . 31

10.1.1.2 offset . . . 31

10.1.1.3 preserve_case . . . 32

10.1.1.4 read_only . . . 32

10.1.1.5 utc . . . 32

10.1.1.6 lazy_load . . . 32

**11 Indices and tables** **33**

**Python Module Index** **35**

**Index** **37**

Python FAT filesystem module with PyFilesystem2 compatibility.

pyfatfs allows interaction with FAT12/16/32 filesystems, either via PyFilesystem2 for file-level abstraction or direct interaction with the filesystem for low-level access.

CHAPTER

ONE

# DOSDATETIME

Enhancement of datetime for DOS date/time format compatibility.

**class** pyfatfs.DosDateTime.**DosDateTime**

Bases: datetime

DOS-specific date/time format serialization.

**static deserialize_date**(*dt: int*) → *DosDateTime*

Convert a DOS date format to a Python object.

**static deserialize_time**(*tm: int*) → time

Convert a DOS time format to a Python object.

**fromtimestamp**(***kwargs*) → *DosDateTime*

timestamp[, tz] -> tz's local time from POSIX timestamp.

**now**(***kwargs*) → *DosDateTime*

Returns new datetime object representing current time local to tz.

**tz**

Timezone object.

If no tz is specified, uses local timezone.

**serialize_date**() → int

Convert current datetime to FAT date.

**serialize_time**() → int

Convert current datetime to FAT time.

CHAPTER

TWO

# EIGHTDOTTHREE

8DOT3 file name helper class & functions.

**class** pyfatfs.EightDotThree.**EightDotThree**(*encoding: str = 'ibm437'*)

Bases: `object`

8DOT3 filename representation.

Offer 8DOT3 filename operation.

**Parameters**

**encoding** – Codepage for the 8.3 filename. Defaults to *FAT_OEM_ENCODING* as per FAT spec.

**INVALID_CHARACTERS = [range(0, 32), 34, 42, 43, 44, 46, 47, 58, 59, 60, 61, 62, 63, 91, 92, 93, 124]**

Invalid characters for 8.3 file names

**SFN_LENGTH = 11**

Length of the byte representation in a directory entry header

**checksum**(***kwargs*)

**get_unpadded_filename**(***kwargs*)

**static is_8dot3_conform**(*entry_name: str*, *encoding: str = 'ibm437'*)

Indicate conformance of given entries name to 8.3 standard.

**Parameters**

- **entry_name** – Name of entry to check
- **encoding** – `str`: Encoding for SFN

**Returns**

bool indicating conformance of name to 8.3 standard

**static make_8dot3_name**(*dir_name: str*, *parent_dir_entry*) → str

Generate filename based on 8.3 rules out of a long file name.

In 8.3 notation we try to use the first 6 characters and fill the rest with a tilde, followed by a number (starting at 1). If that entry is already given, we increment this number and try again until all possibilities are exhausted (i.e. A~999999.TXT).

**Parameters**

- **dir_name** – Long name of directory entry.
- **parent_dir_entry** – *FATDirectoryEntry*: Dir entry of parent dir.

**Returns**
*str*: 8DOT3 compliant filename.

**Raises**
PyFATException: If parent dir is not a directory or all name generation possibilities are exhausted

`set_byte_name`(*name: bytes*)

Set the name as byte input from a directory entry header.

**Parameters**
`name` – *bytes*: Padded (must be 11 bytes) 8dot3 name

`set_str_name`(*name: str*)

Set the name as string from user input (i.e. folder creation).

CHAPTER

THREE

# FATDIRECTORYENTRY

Directory entry operations with PyFAT.

**class** pyfatfs.FATDirectoryEntry.**FATDirectoryEntry**(*DIR_Name: EightDotThree, DIR_Attr: int, DIR_NTRes: int, DIR_CrtTimeTenth: int, DIR_CrtTime: int, DIR_CrtDate: int, DIR_LstAccessDate: int, DIR_FstClusHI: int, DIR_WrtTime: int, DIR_WrtDate: int, DIR_FstClusLO: int, DIR_FileSize: int, encoding: str = 'ibm437', fs: pyfatfs.PyFat.PyFat = None, lazy_load: bool = False, lfn_entry=None*)

Bases: `object`

Represents directory entries in FAT (files & directories).

FAT directory entry constructor.

**Parameters**

- **DIR_Name** – *EightDotThree* class instance
- **DIR_Attr** – Attributes of directory
- **DIR_NTRes** – Reserved attributes of directory entry
- **DIR_CrtTimeTenth** – Milliseconds at file creation
- **DIR_CrtTime** – Creation timestamp of entry
- **DIR_CrtDate** – Creation date of entry
- **DIR_LstAccessDate** – Last access date of entry
- **DIR_FstClusHI** – High cluster value of entry data
- **DIR_WrtTime** – Modification timestamp of entry
- **DIR_WrtDate** – Modification date of entry
- **DIR_FstClusLO** – Low cluster value of entry data
- **DIR_FileSize** – File size in bytes
- **encoding** – Encoding of filename
- **lfn_entry** – FATLongDirectoryEntry instance or None

**ATTR_ARCHIVE = 32**

Bit set in DIR_Attr if entry is an archive

**ATTR_DIRECTORY = 16**

Bit set in DIR_Attr if entry is a directory

**ATTR_HIDDEN = 2**

Bit set in DIR_Attr if entry is hidden

**ATTR_LONG_NAME = 15**

Bits set in DIR_Attr if entry is an LFN entry

**ATTR_LONG_NAME_MASK = 63**

Bitmask to check if entry is an LFN entry

**ATTR_READ_ONLY = 1**

Bit set in DIR_Attr if entry is read-only

**ATTR_SYSTEM = 4**

Bit set in DIR_Attr if entry is a system file

**ATTR_VOLUME_ID = 8**

Bit set in DIR_Attr if entry is a volume id descriptor

**FAT_DIRECTORY_HEADER_SIZE = 32**

Size of a directory entry header in bytes

**FAT_DIRECTORY_LAYOUT = '<11sBBBHHHHHHHL'**

Directory entry header layout in struct formatted string

**FAT_DIRECTORY_VARS = ['DIR_Name', 'DIR_Attr', 'DIR_NTRes', 'DIR_CrtTimeTenth', 'DIR_CrtTime', 'DIR_CrtDate', 'DIR_LstAccessDate', 'DIR_FstClusHI', 'DIR_WrtTime', 'DIR_WrtDate', 'DIR_FstClusLO', 'DIR_FileSize']**

Directory entry headers

**FREE_DIR_ENTRY_MARK = 229**

Marks a directory entry as empty

**LAST_DIR_ENTRY_MARK = 0**

Marks all directory entries after this one as empty

**MAX_FILE_SIZE = 4294967295**

Maximum allowed file size, dictated by size of DIR_FileSize

**add_subdirectory**(*dir_entry*, *recursive: bool = True*)

Register a subdirectory in current directory entry.

**Parameters**

**dir_entry** – FATDirectoryEntry

**Raises**

PyFATException: If current entry is not a directory or given directory entry already has a parent directory set

**property filesize**

Size of the file in bytes.

**Getter**

Get the currently set filesize in bytes

**Setter**

Set new filesize. FAT chain must be extended separately. Raises *PyFATException* with *errno=E2BIG* if filesize is larger than *FATDirectoryEntry.MAX_FILE_SIZE*.

**Type**
int

`get_atime()` → *DosDateTime*

Get dentry access time.

`get_checksum()` → int

Get calculated checksum of this directory entry.

**Returns**
Checksum as int

`get_cluster()`

Get cluster address of directory entry.

**Returns**
Cluster address of entry

`get_ctime()` → *DosDateTime*

Get dentry creation time.

`get_entries()`

Get entries of directory.

**Raises**
PyFatException: If entry is not a directory

**Returns**
tuple: root (current path, full), dirs (all dirs), files (all files)

`get_entry(path: str)`

Get sub-entry if current entry is a directory.

**Parameters**
`path` – Relative path of entry to get

**Raises**
PyFATException: If entry cannot be found

**Returns**
FATDirectoryEntry: Found entry

`get_entry_size()`

Get size of directory entry.

**Returns**
Entry size in bytes as int

`get_full_path()`

Iterate all parents up and join them by "/".

`get_long_name()`

Get long name of directory entry.

**Raises**
NotAnLFNEntryException: If entry has no long file name

**Returns**
str: Long file name of directory entry

**get_mtime()** → *DosDateTime*

Get dentry modification time.

**get_parent_dir()**

Get the parent directory entry.

**get_short_name()**

Get short name of directory entry.

**Returns**
str: Name of directory entry

**get_size()**

Get filesize or directory entry size.

**Returns**
Filesize or directory entry size in bytes as int

**is_archive()**

Determine if dir entry has archive attribute set.

**Returns**
Boolean value indicating archive attribute is set

**is_directory()**

Determine if dir entry has directory attribute set.

**Returns**
Boolean value indicating directory attribute is set

**is_empty()**

Determine if directory does not contain any directories.

**is_hidden()**

Determine if dir entry has the hidden attribute set.

**Returns**
Boolean value indicating hidden attribute is set

**is_read_only()**

Determine if dir entry has read-only attribute set.

**Returns**
Boolean value indicating read-only attribute is set

**is_special()**

Determine if dir entry is a dot or dotdot entry.

**Returns**
Boolean value whether or not entry is a dot or dotdot entry

**is_system()**

Determine if dir entry has the system file attribute set.

**Returns**
Boolean value indicating system attribute is set

**is_volume_id()**

Determine if dir entry has the volume ID attribute set.

**Returns**
Boolean value indicating volume ID attribute is set

**mark_empty()**

Mark this directory entry as empty.

**static new**(*name: EightDotThree, tz: 0, encoding: str, attr: int = 0, ntres: int = 0, cluster: int = 0, filesize: int = 0*) → *FATDirectoryEntry*

Create a new directory entry with sane defaults.

**Parameters**

- **name** – `EightDotThree`: SFN of new dentry
- **tz** – `timezone`: Timezone value to use for new timestamp
- **encoding** – `str`: Encoding for SFN
- **attr** – `int`: Directory attributes
- **ntres** – `int`: Reserved NT directory attributes
- **cluster** – `int`: Cluster number of dentry
- **filesize** – `int`: Size of file referenced by dentry

**Returns**
`FATDirectoryEntry` instance

**remove_dir_entry**(*name*)

Remove given dir_entry from dir list.

**NOTE:** This will also remove special entries such as ».«, »..« and volume labels!

**set_cluster**(*first_cluster*)

Set low and high cluster address in directory headers.

**set_lfn_entry**(*lfn_entry*)

Set LFN entry for current directory entry.

**Param**
lfn_entry: Can be either of type *FATLongDirectoryEntry* or *None*.

**set_size**(*size: int*)

Set filesize.

**Parameters**
**size** – *int*: File size in bytes

**walk()**

Walk all directory entries recursively.

**Returns**
tuple: root (current path, full), dirs (all dirs), files (all files)

**class** `pyfatfs.FATDirectoryEntry.`**FATLongDirectoryEntry**

Bases: `object`

Represents long file name (LFN) entries.

Initialize empty LFN directory entry object.

**FAT_LONG_DIRECTORY_LAYOUT = '<B10sBBB12sH4s'**

LFN entry header layout in struct formatted string

**FAT_LONG_DIRECTORY_VARS = ['LDIR_Ord', 'LDIR_Name1', 'LDIR_Attr', 'LDIR_Type', 'LDIR_Chksum', 'LDIR_Name2', 'LDIR_FstClusLO', 'LDIR_Name3']**

LFN header fields when extracted with *FAT_LONG_DIRECTORY_LAYOUT*

**LAST_LONG_ENTRY = 64**

Ordinance of last LFN entry in a chain

**LFN_ENTRY_LENGTH = 26**

Length for long file name in bytes per entry

**add_lfn_entry**(*LDIR_Ord*, *LDIR_Name1*, *LDIR_Attr*, *LDIR_Type*, *LDIR_Chksum*, *LDIR_Name2*, *LDIR_FstClusLO*, *LDIR_Name3*)

Add LFN entry to this instances chain.

**Parameters**

- **LDIR_Ord** – Ordinance of LFN entry
- **LDIR_Name1** – First name field of LFN entry
- **LDIR_Attr** – Attributes of LFN entry
- **LDIR_Type** – Type of LFN entry
- **LDIR_Chksum** – Checksum value of following 8dot3 entry
- **LDIR_Name2** – Second name field of LFN entry
- **LDIR_FstClusLO** – Cluster address of LFN entry. Always zero.
- **LDIR_Name3** – Third name field of LFN entry

**get_entries**(*reverse: bool = False*)

Get LFS entries in correct order (based on *LDIR_Ord*).

**Parameters**

**reverse** – *bool*: Returns LFN entries in reversed order. This is required for byte representation.

**static is_lfn_entry**(*LDIR_Ord*, *LDIR_Attr*)

Verify that entry is an LFN entry.

**Parameters**

- **LDIR_Ord** – First byte of the directory header, ordinance
- **LDIR_Attr** – Attributes segment of directory header

**Returns**

*True* if entry is a valid LFN entry

**is_lfn_entry_complete**()

Verify that LFN object forms a complete chain.

**Returns**

*True* if *LAST_LONG_ENTRY* is found

**mark_empty**()

Mark LFN entry as empty.

`pyfatfs.FATDirectoryEntry.make_lfn_entry(dir_name: str, short_name: EightDotThree)`

Generate a *FATLongDirectoryEntry* instance from directory name.

**Parameters**

- **dir_name** – Long name of directory
- **short_name** – *EightDotThree* class instance

**Raises**

*PyFATException* if entry name does not require an LFN entry or the name exceeds the FAT limitation of 255 characters

CHAPTER

# FOUR

# FATIO

Implementation of *FatIO* for basic I/O.

**class** `pyfatfs.FatIO.`**`FatIO`**(*fs:* PyFat, *path: str*, *mode: fs.mode.Mode = fs.mode.Mode*)

Bases: `RawIOBase`

Wrap basic I/O operations for PyFat.

Wrap basic I/O operations for PyFat. **Currently read-only**.

**Parameters**

- **fs** – *PyFat*: Instance of opened filesystem
- **path** – *str*: Path to file. If *mode* is *r*, the file must exist.
- **mode** – *Mode*: Mode to open file in.

**close**() → None

Close open file handles assuming lock handle.

**read**(*size: int = -1*) → bytes | None

Read given bytes from file.

**readable**() → bool

Determine whether the file is readable.

**readinto**(*_FatIO__buffer: bytearray*) → int | None

Read data "directly" into bytearray.

**seek**(*offset: int*, *whence: int = 0*) → int

Seek to a given offset in the file.

**Parameters**

- **offset** – `int`: offset in bytes in the file
- **whence** – `int`: offset position: - `0`: absolute - `1`: relative to current position - `2`: relative to file end

**Returns**

New position in bytes in the file

**seekable**() → bool

FAT I/O driver is able to seek in files.

**Returns**

*True*

**truncate**(*size: int | None = 0*) → int

Truncate file to given size.

**Parameters**

**size** – *int*: Size to truncate to, defaults to 0.

**Returns**

*int*: Truncated size

**writable**() → bool

Determine whether the file is writable.

**write**(*_FatIO__b: bytes | bytearray*) → int | None

Write given bytes to file.

CHAPTER

FIVE

# PYFAT

FAT and BPB parsing for files.

**class** pyfatfs.PyFat.**PyFat**(*encoding: str = 'ibm437'*, *offset: int = 0*, *lazy_load: bool = True*)

Bases: object

PyFAT base class, parses generic filesystem information.

Set up PyFat class instance.

**Parameters**

- **encoding** (*str*) – Define encoding to use for filenames
- **offset** (*int*) – Offset of the FAT partition in the given file

**FAT12_CLUSTER_VALUES = {'BAD_CLUSTER': 4087, 'END_OF_CLUSTER_MAX': 4095, 'END_OF_CLUSTER_MIN': 4088, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 4079, 'MIN_DATA_CLUSTER': 2}**

Possible cluster values for FAT12 partitions

**FAT12_SPECIAL_EOC = 4080**

**FAT16_CLEAN_SHUTDOWN_BIT_MASK = 32768**

FAT16 bit mask for clean shutdown bit

**FAT16_CLUSTER_VALUES = {'BAD_CLUSTER': 65527, 'END_OF_CLUSTER_MAX': 65535, 'END_OF_CLUSTER_MIN': 65528, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 65519, 'MIN_DATA_CLUSTER': 2}**

Possible cluster values for FAT16 partitions

**FAT16_DRIVE_ERROR_BIT_MASK = 16384**

FAT16 bit mask for volume error bit

**FAT32_CLEAN_SHUTDOWN_BIT_MASK = 134217728**

FAT32 bit mask for clean shutdown bit

**FAT32_CLUSTER_VALUES = {'BAD_CLUSTER': 268435447, 'END_OF_CLUSTER_MAX': 268435455, 'END_OF_CLUSTER_MIN': 268435448, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 268435439, 'MIN_DATA_CLUSTER': 2}**

Possible cluster values for FAT32 partitions

**FAT32_DRIVE_ERROR_BIT_MASK = 67108864**

FAT32 bit mask for volume error bit

**FAT_CLUSTER_VALUES = {12: {'BAD_CLUSTER': 4087, 'END_OF_CLUSTER_MAX': 4095, 'END_OF_CLUSTER_MIN': 4088, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 4079, 'MIN_DATA_CLUSTER': 2}, 16: {'BAD_CLUSTER': 65527, 'END_OF_CLUSTER_MAX': 65535, 'END_OF_CLUSTER_MIN': 65528, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 65519, 'MIN_DATA_CLUSTER': 2}, 32: {'BAD_CLUSTER': 268435447, 'END_OF_CLUSTER_MAX': 268435455, 'END_OF_CLUSTER_MIN': 268435448, 'FREE_CLUSTER': 0, 'MAX_DATA_CLUSTER': 268435439, 'MIN_DATA_CLUSTER': 2}}**

Maps fat_type to possible cluster values

**FAT_DIRTY_BIT_MASK = 1**

Dirty bit in FAT header

**FAT_TYPE_FAT12 = 12**

Used as fat_type if FAT12 fs has been detected

**FAT_TYPE_FAT16 = 16**

Used as fat_type if FAT16 fs has been detected

**FAT_TYPE_FAT32 = 32**

Used as fat_type if FAT32 fs has been detected

**FAT_TYPE_UNKNOWN = 0**

Used as fat_type if unable to detect FAT type

**FS_TYPES = {0: b'FAT ', 12: b'FAT12 ', 16: b'FAT16 ', 32: b'FAT32 '}**

Maps fat_type to BS_FilSysType from FS header information

**allocate_bytes**(***kwargs*)

**calc_num_clusters**(*size: int = 0*) → int

Calculate the number of required clusters.

**Parameters**
**size** – *int*: required bytes to allocate

**Returns**
Number of required clusters

**close**(***kwargs*)

**flush_fat**(***kwargs*)

**free_cluster_chain**(***kwargs*)

**get_cluster_chain**(***kwargs*)

**get_data_cluster_address**(*cluster: int*) → int

Get offset of given cluster in bytes.

**Parameters**
**cluster** – Cluster number as *int*

**Returns**
Bytes address location of cluster

**get_fs_location**(***kwargs*)

**mkfs**(*filename: str*, *fat_type: 12 | 16 | 32*, *size: int = None*, *sector_size: int = 512*, *number_of_fats: int = 2*, *label: str = 'NO NAME'*, *volume_id: int = None*, *media_type: int = 248*)

Create a new FAT filesystem.

**Parameters**

- **filename** – *str*: Name of file to create filesystem in
- **fat_type** – *FAT_TYPE_FAT{12,16,32}*: FAT type
- **size** – *int*: Size of new filesystem in bytes
- **sector_size** – *int*: Size of a sector in bytes
- **number_of_fats** – *int*: Number of FATs on the disk
- **label** – *str*: Volume label
- **volume_id** – *bytes*: Volume id (4 bytes)
- **media_type** – *int*: Media type (0xF{0,8-F})

**open**(*filename: str | PathLike*, *read_only: bool = False*)

Open filesystem for usage with PyFat.

**Parameters**

- **filename** – *str*: Name of file to open for usage with PyFat.
- **read_only** – *bool*: Force read-only mode of filesystem.

**static open_fs**(*filename: str*, *offset: int = 0*, *encoding='ibm437'*)

Context manager for direct use of PyFAT.

**parse_dir_entries_in_address**(*address: int = 0*, *max_address: int = 0*, *tmp_lfn_entry:* FATLongDirectoryEntry *= None*)

Parse directory entries in address range.

**parse_dir_entries_in_cluster_chain**(*cluster*) → list

Parse directory entries while following given cluster chain.

**parse_header**()

Parse BPB & FAT headers in opened file.

**parse_lfn_entry**(*lfn_entry:* FATLongDirectoryEntry *= None*, *address: int = 0*)

Parse LFN entry at given address.

**parse_root_dir**()

Parse root directory entry.

**read_cluster_contents**(***kwargs*)

**set_fp**(*fp: BytesIO | IOBase*)

Open a filesystem from a valid file pointer.

This allows using in-memory filesystems (e.g., BytesIO).

**Parameters**

**fp** – *FileIO*: Valid *FileIO* object

**update_directory_entry**(***kwargs*)

**write_data_to_cluster**(***kwargs*)

CHAPTER

SIX

# PYFATFS

PyFilesystem2 implementation of PyFAT.

**class** pyfatfs.PyFatFS.**PyFatBytesIOFS**(**args: Any*, ***kwargs: Any*)

Bases: *PyFatFS*

Provide PyFatFS functionality for BytesIO or IOBase streams.

PyFilesystem2 FAT constructor, initializes self.fs with BytesIO.

**Parameters**

- **fp** – *BytesIO | IOBase*: Open file, either in-memory or already open handle.
- **encoding** – *str*: Valid Python standard encoding.
- **offset** – *int*: Offset from file start to filesystem start in bytes.
- **preserve_case** – *bool*: By default 8DOT3 filenames do not support casing. If preserve_case is set to *True*, it will create an LFN entry if the casing does not conform to 8DOT3.
- **utc** – *bool*: Store timestamps in UTC rather than the local time. This applies to dentry creation, modification and last access time.
- **lazy_load** – *bool*: Load directory entries on-demand instead of parsing the entire directory listing on mount.

**class** pyfatfs.PyFatFS.**PyFatFS**(**args: Any*, ***kwargs: Any*)

Bases: FS

PyFilesystem2 extension.

PyFilesystem2 FAT constructor, initializes self.fs.

**Parameters**

- **filename** – *str*: Name of file/device to open as FAT partition.
- **encoding** – *str*: Valid Python standard encoding.
- **offset** – *int*: Offset from file start to filesystem start in bytes.
- **preserve_case** – *bool*: By default 8DOT3 filenames do not support casing. If preserve_case is set to *True*, it will create an LFN entry if the casing does not conform to 8DOT3.
- **read_only** – *bool*: If set to true, the filesystem is mounted in read-only mode, not allowing any modifications.
- **utc** – *bool*: Store timestamps in UTC rather than the local time. This applies to dentry creation, modification and last access time.

- **lazy_load** – *bool*: Load directory entries on-demand instead of parsing the entire directory listing on mount.

**close**()

Clean up open handles.

**create**(*path: str*, *wipe: bool = False*) → bool

Create a new file.

**Parameters**

- **path** – Path of new file on filesystem
- **wipe** – Overwrite existing file contents

**exists**(*path: str*)

Verify if given path exists on filesystem.

**Parameters**
**path** – Path to file or directory on filesystem

**Returns**
Boolean value indicating entries existence

**getinfo**(*path: str*, *namespaces=None*)

Generate PyFilesystem2's *Info* struct.

**Parameters**

- **path** – Path to file or directory on filesystem
- **namespaces** – Info namespaces to query, *NotImplemented*

**Returns**
*Info*

**getmeta**(*namespace='standard'*)

Get generic filesystem metadata.

**Parameters**
**namespace** – Namespace to query, only *standard* supported

**Returns**
*dict* with file system meta data

**getsize**(*path: str*)

Get size of file in bytes.

**Parameters**
**path** – Path to file or directory on filesystem

**Returns**
Size in bytes as *int*

**gettype**(*path: str*)

Get type of file as *ResourceType*.

**Parameters**
**path** – Path to file or directory on filesystem

**Returns**
*ResourceType.directory* or *ResourceType.file*

**listdir**(*path: str*)

List contents of given directory entry.

**Parameters**
**path** – Path to directory on filesystem

**makedir**(*path: str*, *permissions: fs.permissions.Permissions = None*, *recreate: bool = False*)

Create directory on filesystem.

**Parameters**

- **path** – Path of new directory on filesystem
- **permissions** – Currently not implemented
- **recreate** – Ignore if directory already exists

**openbin**(*path: str*, *mode: str = 'r'*, *buffering: int = -1*, ***options*)

Open file from filesystem.

**Parameters**

- **path** – Path to file on filesystem
- **mode** – Mode to open file in
- **buffering** – TBD

**Returns**
*BinaryIO* stream

**opendir**(*path: str*, *factory=None*) → fs.subfs.SubFS

Get a filesystem object for a sub-directory.

**Parameters**
**path** – str: Path to a directory on the filesystem.

**remove**(*path: str*)

Remove a file from the filesystem.

**Parameters**
**path** – *str*: Path of file to remove

**removedir**(*path: str*)

Remove empty directories from the filesystem.

**Parameters**
**path** – *str*: Directory to remove

**removetree**(*dir_path: str*)

Recursively remove the contents of a directory.

**Parameters**
**dir_path** – `str`: Path to a directory on the filesystem.

**setinfo**(*path: str*, *info*)

Set file meta information such as timestamps.

CHAPTER

SEVEN

# PYFATFSOPENER

Registers the PyFatFSOpener used by PyFilesystem2.

**class** pyfatfs.PyFatFSOpener.**PyFatFSOpener**

Bases: Opener

Registers fat:// protocol for PyFilesystem2.

**open_fs**(*fs_url: str*, *parse_result: ParseResult*, *create: bool*, *cwd: str*, *writeable: bool = True*)

Handle PyFilesystem2's protocol opening interface.

**protocols: List[Text] = ['fat']**

CHAPTER

EIGHT

# EXCEPTIONS

**exception** pyfatfs._exceptions.**PyFATException**(*msg: str*, *errno=None*)

Bases: Exception

Generic PyFAT Exceptions.

Construct base class for PyFAT exceptions.

**Parameters**

- **msg** – Exception message describing what happened
- **errno** – Error number, mostly based on POSIX errno where feasible

**exception** pyfatfs._exceptions.**NotAnLFNEntryException**(*msg: str*, *errno=None*)

Bases: *PyFATException*

Indicates that given dir entry cannot be interpreted as LFN entry.

Construct base class for PyFAT exceptions.

**Parameters**

- **msg** – Exception message describing what happened
- **errno** – Error number, mostly based on POSIX errno where feasible

CHAPTER

NINE

# CONTRIBUTING TO PYFAT

Feel free to contribute improvements to this package via pull request.

The preferred method of development is a test driven approach and following the nvie git flow branching model. Please be so kind to bear these things in mind when handing in improvements.

Thank you very much!

CHAPTER

TEN

# GETTING STARTED

## 10.1 PyFilesystem2 quickstart

Use fs.open_fs to open a filesystem with a FAT FS URL:

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1")
```

### 10.1.1 Parameters

It is possible to supply query parameters to the URI of the PyFilesystem2 opener to influence certain behavior; it can be compared to mount options. Multiple parameters can be supplied by separating them via `ampersand (&)`.

#### 10.1.1.1 encoding

pyfatfs offers an encoding parameter to allow overriding the default encoding of ibm437 for file names, which was mainly used by DOS and still is the default on Linux.

Any encoding known by Python can be used as value for this parameter, but keep in mind that this might affect interoperability with other systems, especially when the selected encoding/codepage is not native or supported.

Please note that this only affects encoding of the 8DOT3 short file names, not long file names of the VFAT extension, as LFN are always stored as UTF-16-LE.

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1?encoding=cp1252")
```

#### 10.1.1.2 offset

Specify an offset in bytes to skip when accessing the file. That way even complete disk images can be read if the location of the partition is known:

```
import fs
my_fs = fs.open_fs("fat:///dev/sda?offset=32256")
```

#### 10.1.1.3 preserve_case

Preserve case when creating files. This will force LFN entries for all created files that do not match the 8DOT3 rules. This defaults to `true` but can be disabled by setting preserve_case to `false`:

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1?preserve_case=false")
```

#### 10.1.1.4 read_only

Open filesystem in read-only mode and thus don't allow writes/modifications. This defaults to false but can be enabled by setting read_only to `true`:

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1?read_only=true")
```

#### 10.1.1.5 utc

Create all timestamps on the filesystem in UTC time rather than local time. Affects all directory entries' creation, modification and access times.

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1?utc=true")
```

#### 10.1.1.6 lazy_load

If set to `true` (default), the directory entries are loaded only when accessed to increase performance with larger filesystems and resilience against recursion / directory loops.

```
import fs
my_fs = fs.open_fs("fat:///dev/sda1?lazy_load=false")
```

CHAPTER

# ELEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## p

pyfatfs, **??**
pyfatfs.DosDateTime, 3
pyfatfs.EightDotThree, 5
pyfatfs.FATDirectoryEntry, 7
pyfatfs.FatIO, 15
pyfatfs.PyFat, 17
pyfatfs.PyFatFS, 21
pyfatfs.PyFatFSOpener, 25

# INDEX


## A

add_lfn_entry() (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry method*), 12
add_subdirectory() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 8
allocate_bytes() (*pyfatfs.PyFat.PyFat method*), 18
ATTR_ARCHIVE (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 7
ATTR_DIRECTORY (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 7
ATTR_HIDDEN (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
ATTR_LONG_NAME (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
ATTR_LONG_NAME_MASK (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
ATTR_READ_ONLY (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
ATTR_SYSTEM (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
ATTR_VOLUME_ID (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8

## C

calc_num_clusters() (*pyfatfs.PyFat.PyFat method*), 18
checksum() (*pyfatfs.EightDotThree.EightDotThree method*), 5
close() (*pyfatfs.FatIO.FatIO method*), 15
close() (*pyfatfs.PyFat.PyFat method*), 18
close() (*pyfatfs.PyFatFS.PyFatFS method*), 22
create() (*pyfatfs.PyFatFS.PyFatFS method*), 22

## D

deserialize_date() (*pyfatfs.DosDateTime.DosDateTime static method*), 3
deserialize_time() (*pyfatfs.DosDateTime.DosDateTime static method*), 3
DosDateTime (*class in pyfatfs.DosDateTime*), 3

## E

EightDotThree (*class in pyfatfs.EightDotThree*), 5
exists() (*pyfatfs.PyFatFS.PyFatFS method*), 22

## F

FAT12_CLUSTER_VALUES (*pyfatfs.PyFat.PyFat attribute*), 17
FAT12_SPECIAL_EOC (*pyfatfs.PyFat.PyFat attribute*), 17
FAT16_CLEAN_SHUTDOWN_BIT_MASK (*pyfatfs.PyFat.PyFat attribute*), 17
FAT16_CLUSTER_VALUES (*pyfatfs.PyFat.PyFat attribute*), 17
FAT16_DRIVE_ERROR_BIT_MASK (*pyfatfs.PyFat.PyFat attribute*), 17
FAT32_CLEAN_SHUTDOWN_BIT_MASK (*pyfatfs.PyFat.PyFat attribute*), 17
FAT32_CLUSTER_VALUES (*pyfatfs.PyFat.PyFat attribute*), 17
FAT32_DRIVE_ERROR_BIT_MASK (*pyfatfs.PyFat.PyFat attribute*), 17
FAT_CLUSTER_VALUES (*pyfatfs.PyFat.PyFat attribute*), 17
FAT_DIRECTORY_HEADER_SIZE (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
FAT_DIRECTORY_LAYOUT (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
FAT_DIRECTORY_VARS (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
FAT_DIRTY_BIT_MASK (*pyfatfs.PyFat.PyFat attribute*), 18
FAT_LONG_DIRECTORY_LAYOUT (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry attribute*), 11
FAT_LONG_DIRECTORY_VARS (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry attribute*), 12

FAT_TYPE_FAT12 (*pyfatfs.PyFat.PyFat attribute*), 18
FAT_TYPE_FAT16 (*pyfatfs.PyFat.PyFat attribute*), 18
FAT_TYPE_FAT32 (*pyfatfs.PyFat.PyFat attribute*), 18
FAT_TYPE_UNKNOWN (*pyfatfs.PyFat.PyFat attribute*), 18
FATDirectoryEntry (*class in pyfatfs.FATDirectoryEntry*), 7
FatIO (*class in pyfatfs.FatIO*), 15
FATLongDirectoryEntry (*class in pyfatfs.FATDirectoryEntry*), 11
filesize (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry property*), 8
flush_fat() (*pyfatfs.PyFat.PyFat method*), 18
free_cluster_chain() (*pyfatfs.PyFat.PyFat method*), 18
FREE_DIR_ENTRY_MARK (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
fromtimestamp() (*pyfatfs.DosDateTime.DosDateTime method*), 3
FS_TYPES (*pyfatfs.PyFat.PyFat attribute*), 18

## G

get_atime() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_checksum() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_cluster() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_cluster_chain() (*pyfatfs.PyFat.PyFat method*), 18
get_ctime() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_data_cluster_address() (*pyfatfs.PyFat.PyFat method*), 18
get_entries() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_entries() (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry method*), 12
get_entry() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_entry_size() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_fs_location() (*pyfatfs.PyFat.PyFat method*), 18
get_full_path() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_long_name() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_mtime() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 9
get_parent_dir() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
get_short_name() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
get_size() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
get_unpadded_filename() (*pyfatfs.EightDotThree.EightDotThree method*), 5
getinfo() (*pyfatfs.PyFatFS.PyFatFS method*), 22
getmeta() (*pyfatfs.PyFatFS.PyFatFS method*), 22
getsize() (*pyfatfs.PyFatFS.PyFatFS method*), 22
gettype() (*pyfatfs.PyFatFS.PyFatFS method*), 22

## I

INVALID_CHARACTERS (*pyfatfs.EightDotThree.EightDotThree attribute*), 5
is_8dot3_conform() (*pyfatfs.EightDotThree.EightDotThree static method*), 5
is_archive() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_directory() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_empty() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_hidden() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_lfn_entry() (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry static method*), 12
is_lfn_entry_complete() (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry method*), 12
is_read_only() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_special() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_system() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10
is_volume_id() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 10

## L

LAST_DIR_ENTRY_MARK (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
LAST_LONG_ENTRY (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry attribute*), 12
LFN_ENTRY_LENGTH (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry attribute*), 12
listdir() (*pyfatfs.PyFatFS.PyFatFS method*), 22

## M


make_8dot3_name() (*pyfatfs.EightDotThree.EightDotThree static method*), 5
make_lfn_entry() (*in module pyfatfs.FATDirectoryEntry*), 12
makedir() (*pyfatfs.PyFatFS.PyFatFS method*), 23
mark_empty() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
mark_empty() (*pyfatfs.FATDirectoryEntry.FATLongDirectoryEntry method*), 12
MAX_FILE_SIZE (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry attribute*), 8
mkfs() (*pyfatfs.PyFat.PyFat method*), 18
module
    pyfatfs, 1
    pyfatfs.DosDateTime, 3
    pyfatfs.EightDotThree, 5
    pyfatfs.FATDirectoryEntry, 7
    pyfatfs.FatIO, 15
    pyfatfs.PyFat, 17
    pyfatfs.PyFatFS, 21
    pyfatfs.PyFatFSOpener, 25


## N


new() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry static method*), 11
NotAnLFNEntryException, 27
now() (*pyfatfs.DosDateTime.DosDateTime method*), 3


## O


open() (*pyfatfs.PyFat.PyFat method*), 19
open_fs() (*pyfatfs.PyFat.PyFat static method*), 19
open_fs() (*pyfatfs.PyFatFSOpener.PyFatFSOpener method*), 25
openbin() (*pyfatfs.PyFatFS.PyFatFS method*), 23
opendir() (*pyfatfs.PyFatFS.PyFatFS method*), 23


## P


parse_dir_entries_in_address() (*pyfatfs.PyFat.PyFat method*), 19
parse_dir_entries_in_cluster_chain() (*pyfatfs.PyFat.PyFat method*), 19
parse_header() (*pyfatfs.PyFat.PyFat method*), 19
parse_lfn_entry() (*pyfatfs.PyFat.PyFat method*), 19
parse_root_dir() (*pyfatfs.PyFat.PyFat method*), 19
protocols (*pyfatfs.PyFatFSOpener.PyFatFSOpener attribute*), 25
PyFat (*class in pyfatfs.PyFat*), 17
PyFatBytesIOFS (*class in pyfatfs.PyFatFS*), 21
PyFATException, 27
pyfatfs
    module, 1
PyFatFS (*class in pyfatfs.PyFatFS*), 21
pyfatfs.DosDateTime
    module, 3
pyfatfs.EightDotThree
    module, 5
pyfatfs.FATDirectoryEntry
    module, 7
pyfatfs.FatIO
    module, 15
pyfatfs.PyFat
    module, 17
pyfatfs.PyFatFS
    module, 21
pyfatfs.PyFatFSOpener
    module, 25
PyFatFSOpener (*class in pyfatfs.PyFatFSOpener*), 25


## R


read() (*pyfatfs.FatIO.FatIO method*), 15
read_cluster_contents() (*pyfatfs.PyFat.PyFat method*), 19
readable() (*pyfatfs.FatIO.FatIO method*), 15
readinto() (*pyfatfs.FatIO.FatIO method*), 15
remove() (*pyfatfs.PyFatFS.PyFatFS method*), 23
remove_dir_entry() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
removedir() (*pyfatfs.PyFatFS.PyFatFS method*), 23
removetree() (*pyfatfs.PyFatFS.PyFatFS method*), 23


## S


seek() (*pyfatfs.FatIO.FatIO method*), 15
seekable() (*pyfatfs.FatIO.FatIO method*), 15
serialize_date() (*pyfatfs.DosDateTime.DosDateTime method*), 3
serialize_time() (*pyfatfs.DosDateTime.DosDateTime method*), 3
set_byte_name() (*pyfatfs.EightDotThree.EightDotThree method*), 6
set_cluster() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
set_fp() (*pyfatfs.PyFat.PyFat method*), 19
set_lfn_entry() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
set_size() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
set_str_name() (*pyfatfs.EightDotThree.EightDotThree method*), 6
setinfo() (*pyfatfs.PyFatFS.PyFatFS method*), 23

SFN_LENGTH (*pyfatfs.EightDotThree.EightDotThree attribute*), 5

## T

truncate() (*pyfatfs.FatIO.FatIO method*), 15

## U

update_directory_entry() (*pyfatfs.PyFat.PyFat method*), 19

## W

walk() (*pyfatfs.FATDirectoryEntry.FATDirectoryEntry method*), 11
writable() (*pyfatfs.FatIO.FatIO method*), 16
write() (*pyfatfs.FatIO.FatIO method*), 16
write_data_to_cluster() (*pyfatfs.PyFat.PyFat method*), 19