PyQModManager Documentation

Release 1.0.0-alpha9

bicobus

Apr 28, 2020

Contents:

A kind of archive manager for easy handling of game modules.

Works on Python 3.7+.

CHAPTER 1

User's Guide

1.1 PyQModManager

In this document will inform you about various elements of the graphical user interface of PyQModManager.

1.1.1 Settings

The software needs to know two on disk location: where the game is located and a space to store the modules you've downloaded.

The game location should be the folder containing the .jar or .exe of the game. The repository for your module should be a random *empty* folder of your choice.

At the time I write these lines, the settings window is still a WiP.

1.1.2 Adding modules to be tracked by the software

The software keeps track of 3 types of archives: Rar files, 7z files and Zip files. Two ways exists to have the game track modules:

1. Drop an archive in the repository folder.
2. Use the button from the toolbar.

If you use , the archive file will be copied over the repository folder leaving the original file untouched.

1.1.3 Removing an archive

Select the archive you wish to delete and click on the trashbin () button. Alternatively, right-click on the archive and select the appropriate option. PyQModManager sends all removed archives to your trashbin, which you will need to empty manually.

1.1.4 Installing a module

Select the archive you wish to install then click on the install () button of the toolbar. Alternatively, you can right-click the archive and select the install option.

1.1.5 Uninstalling a module

Select the archive you wish to uninstall then click on the uninstall () button of the toolbar. Alternatively, you can right-click the archive and select the uninstall option.

1.1.6 Monitoring of the hard drive

The software will monitor filesystem changes on both the game's module folder and the folder you designated as repository. The software will automatically scan any archive dropped in your repository folder, as well as making sure the game's module folder remains known even if you unpack files through other means than PyQModManager.

You can toggle off that behavior through the auto-refresh checkbox located above the list of your available modules. Doing so will activate a refresh button, located right next to the checkbox, which will allow you to manually refresh the internal database if you make changes to the file system.

The monitoring of the filesystem is designed to be as lightweight as possible. It disable itself whenever PyQModManager becomes inactive (alt-tab or minimized), and reactive itself whenever the software gain focus. Gaining back focus will force a refresh of the database on a needed basis: if nothing has changed, nothing is done.

1.2 qmm

1.2.1 qmm package

Submodules

qmm.bucket module

Buckets of dicts with a set of helpers function.

This module serves has a stand-in database, any function or method it contain would be facilitator to either access or transform the data. This module is necessary in order to keep track of the state of the different files and make that specific state available globally within the other modules.

class qmm.bucket.**FileMetadata**(*crc, path: Union[str, pathlib.Path], attributes, modified, isfrom*)

Bases: object

Representation of a file.

Can handle game files, mod files or file information comming from an archive.

as_dict()

Return this object as a dict (kinda).

exists()

Check if the file exists on the disk

is_dir()

Check if the represented item is a directory

is_file()
Check if the represented item is a file

qmm.bucket.**as_conflict**(*key: str*, *value*)
Append and item to the conflicts bucket

qmm.bucket.**as_loosefile**(*crc: int*, *filepath: pathlib.Path*)
Adds filepath to the loosefiles bucket, indexed on given CRC.

qmm.bucket.**file_crc_in_loosefiles**(*filemd: qmm.bucket.FileMetadata*) → bool
Check if a file's crc exists in loosefile's index.

qmm.bucket.**file_path_in_loosefiles**(*filemd: qmm.bucket.FileMetadata*) → bool
Check if a file's path exists within the different loosefile lists.

qmm.bucket.**remove_item_from_loosefiles**(*file: qmm.bucket.FileMetadata*)
Removes the reference to file if it is found in loosefiles

qmm.bucket.**with_conflict**(*path: str*) → bool
Check if path exists in conflicts's keys.

The conflicts bucket purpose is to list issues in-between archives only.

Parameters **path** (*str*) – Simple string, should be a path pointing to a file

Returns True if path exist in conflicts's keys

Return type bool

qmm.bucket.**with_gamefiles**(*crc: int = None*, *path: str = None*)
First check if a CRC32 exist within the gamefiles bucket, if no CRC is given or the check fails, will then check if a path is present in the gamefiles's bucket values. :param crc: CRC32 as integer :type crc: int :param path: the relative pathlike string of a file :type path: str

Returns True if either CRC32 or path are found

Return type bool

qmm.common module

qmm.common.**settings_are_set**()
Returns False if either 'local_repository' or 'game_folder' isn't set.

qmm.common.**timestamp_to_string**(*timestamp*)
Takes a UNIX timestamp and return a vernacular date.

qmm.common.**tools_path**()
Returns the path to the 7z executable.

TODO: needs a better name

qmm.common.**valid_suffixes**(*output_format='qfiledialog'*) → Union[List[str], Tuple[str, str, str], bool]
Properly format a list of filters for QFileDialog.

Parameters **output_format** – Accepts either 'qfiledialog' or 'pathlib'. 'pathlib' returns a simple list of suffixes, whereas 'qfiledialog' format the output to be an acceptable filter for QFileDialog.

Returns: list

qmm.config module

class qmm.config.**Config**(*filename*, *config_dir=None*, *defaults=None*, *compress=False*)
Bases: collections.abc.MutableMapping

Influenced by deluge's config object.

delayed_save(*msec=5000*)
Schedule a save in the future if one isn't already planned.

exception qmm.config.**SettingsNotSetError**
Bases: Exception

qmm.config.**get_config_dir**(*filename=None*, *extra_directories=None*) → str
Return the full path of the user config dir.

Parameters

- **filename** – If provided, gets added at the end of the string.
- **extra_directories** – If provided, extends on the returned path.

qmm.dialogs module

qmm.filehandler module

exception qmm.filehandler.**ArchiveException**
Bases: *qmm.filehandler.FileHandlerException*

class qmm.filehandler.**ArchivesCollection**
Bases: collections.abc.MutableMapping, typing.Generic

add_archive(*path*, *hashsum: str = None*, *progress=None*)
Add an archive to the list of managed archives.

This method should be used over __setitem__ as it setup the different metadata required by the UI.

find(*archive_name: str = None*, *hashsum: str = None*)
Find a member based on the name or hashsum of the archive.

If archiveName is not None, will check if archiveName exists in the keys of the collection. If hashsum is not None, will check if the value exists in the self._hashsums dict. If all checks fails, returns False.

Parameters

- **archive_name** – filename of the archive, suffix included (default None)
- **hashsum** – sha256sum of the file (default None)

refresh() → Iterable[Tuple[int, str]]
Scan the local repository to add or remove archives as needed.

This is a companion method to use with WatchDog whenever something changes on the filesystem.

Yields event_type, name of archive, List[Filemetadata]

rename_archive(*src_path*, *dest_path*)
Rename the key pointing to an archive

Whenever an archive on the drive gets renamed, we need to do the same with the key under which the parsed data is stored.

exception qmm.filehandler.**FileHandlerException**
Bases: Exception

qmm.filehandler.**archive_analysis**(*file_list: List[qmm.bucket.FileMetadata]*) → List[Tuple[qmm.bucket.FileMetadata, int]]
Return a list of tuples representing the archive.

The tuples contains the item of an archive alongside it's status. The status can be either FILE_MATCHED, FILE_MISMATCHED, FILE_IGNORED or FILE_MISSING.

Parameters **file_list** – a list containing every items of an archive, must be bucket.FileMetadata

Returns A list of tuples, each tuple being (FileMetadata, int)

qmm.filehandler.**build_game_files_crc32**(*progress=None*)
Compute the CRC32 value of all the game files then add them to a bucket.

The paths returned by this function are non-existent due to a difference between the mods and the game folder structure. It is needed to be that way in order to compare the mod files with the existing game files.

Parameters **progress** (*dialogs.qProgress.progress*) – Callback to a method accepting strings as argument.

qmm.filehandler.**build_loose_files_crc32**(*progress=None*)
Build the CRC32 value of all loose files.

Parameters **progress** (*dialogs.qProgress.progress*) – Callback to a method accepting strings as argument.

Returns None

qmm.filehandler.**conflicts_process_files**(*files*, *archives_list*, *current_archive*, *processed*)
Process an archive, verify that each of its files are unique.

qmm.filehandler.**copy_archive_to_repository**(*filename*)
Copy an archive to the manager's repository

qmm.filehandler.**delete_archive**(*filepath*)
Delete an archive from the filesystem.

qmm.filehandler.**file_in_other_archives**(*file: qmm.bucket.FileMetadata, archives: qmm.filehandler.ArchivesCollection, ignore: List[T]*) → List[T]
Search for existence of file in other archives.

Parameters

- **file** (FileMetadata) – file to be found
- **archives** (ArchivesCollection) – instance of ArchivesCollection
- **ignore** (*list*) – list of archives to ignore, for example already parsed archives

Returns List of archives containing the same file.

Return type List

qmm.filehandler.**get_mod_folder**(*with_file: str = None*, *prepend_modpath=False*) → pathlib.Path
Return the path to the game folder.

Parameters

- **with_file** – append 'with_file' to the path
- **prepend_modpath** – if True, adds the module path before 'with_file'

Returns PathLike structure representing the game folder.

qmm.filehandler.**ignore_patterns**(*seven_flag=False*)
Output a tuple of patterns to ignore.

Parameters **seven_flag** (*bool*) – Patterns format following 7z exclude switch.

qmm.filehandler.**install_archive**(*file_to_extract: str, file_context: Dict[str, List[qmm.bucket.FileMetadata]]*) → Union[bool, List[qmm.bucket.FileMetadata]]
Install the content of an archive into the game mod folder.

Parameters

- **file_to_extract** – path to the archive to extract.
- **file_context** – A dict containing the keys matched, mismatched, ignored. Each of these entries point to a list containing FileMetadata objects.

Returns Output of function extract7z() or False

qmm.filehandler.**reErrorMatch**()
Matches zero or more characters at the beginning of the string.

qmm.filehandler.**reExtractMatch**()
Matches zero or more characters at the beginning of the string.

qmm.filehandler.**reListMatch**()
Matches zero or more characters at the beginning of the string.

qmm.filehandler.**sha256hash**(*filename: Union[IO, str]*) → Optional[str]
Returns the 256 hash of the managed archive.

Parameters **filename** – path to the file to hash

Returns if successful None: if not successful

Return type string

qmm.filehandler.**uninstall_files**(*file_list: list*)
Removes a list of files and directory from the filesystem.

qmm.lang module

qmm.manager module

qmm.version module

qmm.widgets module

CHAPTER 2

Contribute

Did you find a bug or have a feature request for PyQModManager? You can file an issue ticket at the issue tracker. You can also ask questions at the Lilith's Throne official discord.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

T

U

V

W