PyQModManager Documentation

Release 1.0.0-alpha11

bicobus

May 12, 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

Simple tool to manage a set of archives. Written to manage mods available for the game Lilith's Throne.

- Track the state of the different files bundled with each archives
- Unpack into a designated directory

1.1.1 Installation and Requirements

Windows

If you are running windows, a self contained binary is provided at each release. Please check the releases page to download the archive. Running the application is a simple as double clicking the .exe file present in the archive.

Linux

You'll need to install python3 with whatever package manager your distribution provides you. This application has been developped using python 3.7, any prior version might work, but untested. If you feel adventurous, you can checkout pyenv.

This software being a python script, it doesn't need to be installed. However requirements are needed for that purpose.

You'll need whatever is presesent in the requirements.txt. As Linux distributions such as debian are commonly out of date, I'd recommend installing the requirement locally, under your user directory, with the following command.

```
~$ pip install --user -r requirements.txt
```

You could also opt to install in a virtual environment through pipenv. `pipenv install` will use the Pipfile already present in the repository.

```
~$ pip install pipenv
~$ pipenv install
```

Running the app

If you chose pipenv, you can then start the application using the following, provided you are in the same folder as the run.py file.

```
~$ pipenv run ./run.py
```

If you opted for the simple pip install, simply execute run.py.

```
~$ python run.py
```

1.1.2 Known issues

The software is currently being rewritten, as such no known limitation exists. This might change once we get out of alpha.

1.1.3 Hacking

If you want to hack around, you only require the dependencies listed in the requirements.txt file. The Pipfile has a list of dev-packages, which is unneeded to actually run and develop the software. They're helper tools, like pylint or flake8

Documentation generator

The documentation is currently available at https://qmodmanager.rtfd.io/

The generation of the documentation on readthedocs.org necessitate some extra steps in order to successfuly generate the api documentation. We have to generate ui files at build time, which are not included in the repository nor available to rtd. Therefore a script apidoc has been provided as helper.

The apidoc script will forcefully generate the required files for the api. In addition of that, it will also parse the various UI files present in the resources folder and generate stubs in the _ext folder. Those stubs needs to be regenerated whenever a UI file is added to the resources folder.

1.2 Usage

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

1.2.1 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.

1.2.2 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.

1.2.3 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.

1.2.4 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.

1.2.5 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.

1.2.6 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.

1.3 Version History

1.3.1 1.0.0-alpha11 - 20-05-12

Added

- Color code each managed item based on their status
  - Each line has a dual color: left and right
  - Right side can either be transparent or red, to show existing conflicts.
  - Left side can either be green, blue or yellow
    - Yellow is for missing files
    - Blue is for mismatched files
    - Green is when every files of the archive matches on the drive.
  - Greyed out text means the archive contains nothing that can be installed
  - The Help buttons will send users to the readthedocs website.

Changed

- Each file is now beautifully displayed in a tree instead of using a TextInput
- Files are color coded depending on their states.
- The conflicts tab details where a file as been found as duplicate: *GameFile* or *Archive*

Fixed

- Fix crash related to file system watch (watchdog)

1.3.2 1.0.0-alpha10

- Same as alpha9, but working.

1.3.3 1.0.0-alpha9

- Send archives to the trashbin instead of a full removal from the hard drive.
- Foundations for the internationalisation (l10n) of the software through gettext
- A Watchdog to monitor both the module's repository and the game's module path
  - The software will automatically add whatever archive dropped in the module's repository
  - The software will automatically determine if the game's module directory has been modified and regenerate it's database the next time the application gain focus
  - A checkbox exists to disable this behavior if unchecked.
- Internal dev stuff: changes of libraries used, reworking codebase, etc

1.4 qmm

1.4.1 qmm package

qmm.**get_base_path**()

qmm.**get_data_path**(*relpath*)

qmm.**is_frozen**()

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.

Parameters

- **crc** – CRC32 of the represented file, 0 or empty if file is a folder.
- **path** – relative path to the represented file.
- **attributes** – 'D' for folder, 'F' otherwise.
- **modified** – timestamp of the last modification of the file.
- **isfrom** – either 'TYPE_GAMEFILE' or 'TYPE_LOOSEFILE'

as_dict()

Return this object as a dict (kinda).

attributes

crc

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

modified

origin

path

path_as_posix()

Return 'pathlib.PurePosixPath' with self._Path as value.

split()

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

qmm.bucket.**as_gamefile**(*crc: Crc32, value: Union[pathlib.Path, pathlib.PurePath]*)
Add to the gamefiles a path indexed to its target CRC32.

qmm.bucket.**as_loosefile**(*crc: Crc32*, *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: Crc32 = 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** (*str*) – 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 a list of valid suffixes.

Return type 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.

load(*filename=None*)

save(*filename=None*)

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

Contains a bunch of helper function to display Qt's dialogs.

class qmm.dialogs.**SplashProgress**(*parent*, *title*, *message*)
Bases: QDialog, qmm.ui_qprogress.Ui_Dialog

progress(*text: str*, *category: str = None*)

qmm.dialogs.**qError**(*message*, ***kwargs*)
Helper function to show an error dialog.

qmm.dialogs.**qInformation**(*message*, ***kwargs*)
Helper function to show an informational dialog.

qmm.dialogs.**qWarning**(*message*, ***kwargs*)
Helper function to show a warning dialog.

qmm.dialogs.**qWarningYesNo**(*message*, ***kwargs*)
Helper function to show an Y/N warning dialog.

qmm.filehandler module

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

class qmm.filehandler.**ArchiveInstance**(*archive_name: str, file_list: List[qmm.bucket.FileMetadata]*)
Bases: object

Represent an archive and its content already analyzed and ready for display.

all_ignored
Value is *True* if all files of the archive are of status *FILE_IGNORED*.

all_matching
Return *True* if all files in the archive matches on the drive.

conflicts()
Yield file metadata of conflicting entries of the archive.

files(*exclude_directories=False*) → Generator[qmm.bucket.FileMetadata, None, None]

folders() → Generator[qmm.bucket.FileMetadata, None, None]
Yield folders present in the archive

get_status(*file*)

has_conflicts
Value is *True* if conflicts exists for this archive.

has_ignored
Value is *True* if a file of the archive is of status *FILE_IGNORED*.

has_matched
Return True if a file of the archive is of status *FILE_MATCHED*.

has_mismatched
Value is *True* if a file of the archive is of status *FILE_MISMATCHED*.

has_missing
Value is *True* if a file of the archive is of status *FILE_MISSING*.

ignored() → Iterable[qmm.bucket.FileMetadata]
Yield file metadata of ignored entries of the archive.

install_info()
Return a several lists useful to the installation process.

The content in matched and ignored key will be compiled into a set of exclude flags, whereas the content of mismatched key will be overridden

See also:

install_archive()

matched() → Generator[qmm.bucket.FileMetadata, None, None]
Yield file metadata of matched entries of the archive.

mismatched() → Generator[qmm.bucket.FileMetadata, None, None]
Yield file metadata of mismatched entries of the archive.

missing() → Generator[qmm.bucket.FileMetadata, None, None]
Yield file metadata of missing entries of the archive.

reset_conflicts()
Generate a list of conflicting files, either from in the game folders or in other archives, for each file present in this archive.

reset_status()
Called whenever the state of an archive becomes dirty, which is also the default state.

Populate 'self._meta' with tuples containing the 'FileMetadata' object of each individual file alongside the current status of that file. The status can be either 'FILE_MATCHED', 'FILE_MISMATCHED', 'FILE_IGNORED' or 'FILE_MISSING'.

status() → Generator[Tuple[qmm.bucket.FileMetadata, int], None, None]

uninstall_info()
Informations necessary to the uninstall function

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

FileAdded = 1
State of a file yielded through *refresh()*

FileRemoved = 2
State of a file yielded through *refresh()*

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.

build_archives_list(*progress*, *rebuild=False*)

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)

hashsums(*key*)

initiate_conflicts_detection()

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 (Union[*FileAdded*, *FileRemoved*], str) –

State and name of the file

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.

stat(*key*)

qmm.filehandler.**FILE_IGNORED = 4**
Indicate that the file will be ignored by the software.

qmm.filehandler.**FILE_MATCHED = 1**
Indicate that the file is found on the drive, and match in content.

qmm.filehandler.**FILE_MISMATCHED = 3**
Indicate that the file to exists on drive, but not matching in content.

qmm.filehandler.**FILE_MISSING = 2**
Indicate that the file is absent from the drive.

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

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

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.**extract7z**(*file_archive: pathlib.Path*, *output_path: pathlib.Path*, *exclude_list=None*, *progress=None*) → Union[List[qmm.bucket.FileMetadata], bool]

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.**file_status**(*file: qmm.bucket.FileMetadata*) → int

qmm.filehandler.**generate_conflicts_between_archives**(*archives_lists: qmm.filehandler.ArchivesCollection*, *progress=None*)

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

  The content in matched and ignored key will be compiled into a set of exclude flags, whereas the content of mismatched key will be overridden. See *ArchiveInstance.install_info()*

Returns Output of function *extract7z()* or False

qmm.filehandler.**list7z**(*file_path: Union[str, pathlib.Path], progress=None*) → List[qmm.bucket.FileMetadata]

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 a string if successful, otherwise None

Return type str or None

qmm.filehandler.**uninstall_files**(*file_list: List[qmm.bucket.FileMetadata]*)
Removes a list of files and directory from the filesystem.

Parameters **file_list** (*list[FileMetadata]*) – A list of *FileMetadata* objects.

Returns True on success, False if an error occurred during the deleting process.

Return type bool

Notes

Any error will be logged silently to the application configured facility.

qmm.lang module

qmm.lang.**get_locale**()

qmm.lang.**list_available_languages**()

qmm.lang.**normalize_locale**(*loc: str*)

qmm.lang.**set_gettext**(*install=True*)

qmm.manager module

Handles the Qt main window.

class qmm.manager.**ArchiveAddedEventHandler**(*moved_cb*, *created_cb*, *deleted_cb*, *modified_cb*)

Bases: *qmm.manager.QmmWdEventHandler*, watchdog.events.PatternMatchingEventHandler, QObject

on_created(*event*)

Called when a file or directory is created.

Parameters **event** (DirCreatedEvent or FileCreatedEvent) – Event representing file/directory creation.

on_deleted(*event*)

Called when a file or directory is deleted.

Parameters **event** (DirDeletedEvent or FileDeletedEvent) – Event representing file/directory deletion.

on_modified(*event*)

Called when a file or directory is modified.

Parameters **event** (DirModifiedEvent or FileModifiedEvent) – Event representing file/directory modification.

on_moved(*event*)

Called when a file or a directory is moved or renamed.

Parameters **event** (DirMovedEvent or FileMovedEvent) – Event representing file/directory movement.

class qmm.manager.**CustomMenu**

Bases: object

setup_menu(*obj*)

Register self as obj's context menu

class qmm.manager.**GameModEventHandler**(*moved_cb*, *created_cb*, *deleted_cb*, *modified_cb*)

Bases: *qmm.manager.QmmWdEventHandler*, watchdog.events.PatternMatchingEventHandler, QObject

on_any_event(*event*)

Catch-all event handler.

Parameters **event** (FileSystemEvent) – The event object representing the file system event.

on_created(*event*)

Called when a file or directory is created.

Parameters **event** (DirCreatedEvent or FileCreatedEvent) – Event representing file/directory creation.

on_deleted(*event*)

Called when a file or directory is deleted.

Parameters **event** (DirDeletedEvent or FileDeletedEvent) – Event representing file/directory deletion.

on_modified(*event*)

Called when a file or directory is modified.

Parameters **event** (DirModifiedEvent or FileModifiedEvent) – Event representing file/directory modification.

on_moved(*event*)
Called when a file or a directory is moved or renamed.

Parameters **event** (DirMovedEvent or FileMovedEvent) – Event representing file/directory movement.

class qmm.manager.**MainWindow**
Bases: QMainWindow, *qmm.manager.QEventFilter*, *qmm.manager.CustomMenu*, qmm.ui_mainwindow.Ui_MainWindow

callback_at_show(*item*)

do_about()
Show the about window.

do_settings(*first_launch=False*)
Show the settings window.

Parameters **first_launch** (*bool*) – If true, disable cancel button and bind the save button to qmm.MainWindow._init_mods

fswatch_clear(*QString*, *QString*)

fswatch_ignore(*QString*, *QString*)

get_row_index_by_name(*name*)
Return row if name is found in the list.

Parameters **name** (*str*) – Filename of the archive to find, matches content of the ListRowItem text method.

Returns index of item found, *None* if *name* matches nothing.

Return type int or None

is_mod_repo_dirty

on_window_activate()

on_window_deactivate()

set_tab_color(*index*, *color: PyQt5.QtGui.QColor = None*) → None
Manage tab text color.

Helper to MainWindow._on_selection_change.

Store the default text color of a tab in order to restore it whenever the selected element in the linked list changes.

Parameters

- **index** (*int*) – index of the tab
- **color** (*QColor*) – new color of the text

setup_schedulers()

show(*self*)

class qmm.manager.**QAppEventFilter**
Bases: QObject

Detect if the application is active then triggers to appropriate events

The purpose of this object is to enable or disable WatchDog related procedures. We want to disable file system watch on the modules directory when the window is inactive (user has alt-tabbed outside of it or minimized the application), as such delay any activity until the user comes back to the application itself. The intent is to minimize uneeded operations as the user could move and rename multiple files in the folder. We only need to scan the module's repository once the user has finished, thus once the application becomes active.

The detection of activity needs to be done at the Session Manager, namely `QApplication` (`QGuiApplication` or `QCoreApplication`). That object handles every window and widgets of the application. Each of those window and widgets could become inactive regardless of the status of the whole application. Inactivity could be defined as whenever the application loose focus (keyboard input). This loss also happen whenever the window is being dragged around by the user, which means we need to make sure to not trigger any refresh of the database for those user cases. To achieve that we track the geometry and coordinates of the window and trigger the callback only if those parameters remains the same between an inactive and active event.

Callbacks are `on_window_activate` and `on_window_deactivate`.

eventFilter(*self*, *QObject*, *QEvent*) → bool

get_coords() → Tuple[int, int]
Return the coordinates of the top window.

get_geometry() → Tuple[int, int]
Return the geometry of the top window.

set_coords()

set_geometry()

set_top_window(*window: qmm.manager.MainWindow*)
Define the widget that is considered as top window.

class qmm.manager.**QEventFilter**
Bases: `object`

eventFilter(*o*, *e*)

setup_filters(*objects*)

class qmm.manager.**QmmWdEventHandler**(*moved_cb*, *created_cb*, *deleted_cb*, *modified_cb*)
Bases: `object`

clear(*src_path*, *event_type*)
Remove a path from the event's ignore tuple.

ignore(*src_path*, *event_type*)
Ignore an event if path is found in it's ignore tuple.

sgn_created(*PyQt_PyObject*)

sgn_deleted(*PyQt_PyObject*)

sgn_modified(*PyQt_PyObject*)

sgn_moved(*PyQt_PyObject*)

exception qmm.manager.**UnknownContext**
Bases: `Exception`

qmm.manager.**main**()
Start the application proper.

qmm.version module

qmm.widgets module

Contains various Qt Widgets used internally by the application.

class qmm.widgets.**ListRowItem**(*filename: str*, *archive_manager: qmm.filehandler.ArchivesCollection*)

Bases: QListWidgetItem

ListWidgetItem representing one single archive.

filename
Returns the name of the archive filename, suitable for path manipulations.

hashsum
Returns the sha256 hashsum of the archive.

modified
Return last modified time for an archive, usually time of creation.

name
Return the name of the archive, formatted for GUI usage.

Transfrom the '_' character into space.

refresh_strings()
Called when the game's folder state changed.

Reinitialize the widget's strings, recompute the conflicts then redo all triaging and formatting.

set_gradients()

set_text_color()

class qmm.widgets.**QAbout**(*parent=None*)

Bases: QWidget, qmm.ui_about.Ui_About

About window displaying various informations about the software.

class qmm.widgets.**QSettings**

Bases: QMainWindow

connect_to_savebutton(*callback*)

disconnect_from_savebutton(*callback*)

set_mode(*first_run=False*)

setup_ui()

show()
Show the window and assign internal variables.

class qmm.widgets.**QSettingsCentralWidget**(*parent=None*)

Bases: QWidget, qmm.ui_settings.Ui_Settings

Define the settings windows.

on_cancel_button_clicked()
Simply hide the window.

The default values are being defined within the show method, thus there is nothing here for us to do.

Returns void

qmm.widgets.**autoresize_columns**(*tree_widget: PyQt5.QtWidgets.QTreeWidget*)
Resize all columns of a QTreeWidget to fit content.

qmm.widgets.**build_conflict_tree_widget**(*container: PyQt5.QtWidgets.QTreeWidget*, *archive_instance: qmm.filehandler.ArchiveInstance*)

qmm.widgets.**build_ignored_tree_widget**(*tree_widget: PyQt5.QtWidgets.QTreeWidget*, *ignored_iter: Iterable[qmm.bucket.FileMetadata]*)

qmm.widgets.**build_tree_from_path**(*item: qmm.bucket.FileMetadata*, *parent: PyQt5.QtWidgets.QTreeWidget*, *folders*, *color=None*, *extra_column=None*)
Generate a set of related `PyQt5.QtWidgets.QTreeWidgetItem()` based on a file path.

Parameters

- **item** – a *qmm.bucket.FileMetadata* object.
- **parent** – The container widget to anchor the first node to.
- **folders** – A dict containing the parents widgets.
- **color** – Background color value for the widget.
- **extra_column** – Extra values to pass down to `_create_treewidget()`

Returns A dictionnary containing the folders ancestry.

Return type dict

qmm.widgets.**build_tree_widget**(*container: PyQt5.QtWidgets.QTreeWidget*, *archive_instance: qmm.filehandler.ArchiveInstance*)

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

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W