pyfomod Documentation

Release 1.2.1

Daniel Nunes

Nov 02, 2019

Contents

pyfomod is a high-level fomod library written in python. No prior knowledge of xml or fomod itself is required.

To start off, we'll import *pyfomod* as usual:

```
>>> import pyfomod
```

Fomod installers are packaged under a **fomod** subfolder and include a **moduleconfig.xml** file and optionally a **info.xml** file.

parse(*source*[, *warnings*], *lineno=False*)

This function is used to parse either a package or loose files:

```
>>> os.listdir("package")
['fomod', 'readme.txt', 'content']
>>> os.listdir("package/fomod")
['info.xml', 'moduleconfig.xml']
>>> root = pyfomod.parse("package")

>>> payload = ("package/fomod/info.xml", "package/fomod/moduleconfig.xml")
>>> root = pyfomod.parse(payload)
```

As seen above, *source* can be either a path-like object with the path to a package root or a tuple of (**path/to/info.xml**, **path/to/moduleconfig.xml**). If **info.xml** does not exist, `None` should be passed as its path.

warnings is used to collect warnings related to parsing. See *Validation* for more information on this.

lineno is a boolean on whether to load the line numbers of the original package into each element of the tree. This is by default `False` due to the increased performance burden this option places on the parser.

The returned *root* is the root of the fomod tree, see *Root*.

pyfomod can safely ignore most errors present in the physical files. Please note, however, that *pyfomod* does not support comments and they are not parsed at all.

write(*root*, *path*)

When you are finished reading and/or modifying your tree, write it:

```
>>> pyfomod.write(root, "package")
```

root should be the *Root* of the fomod tree you're writing. *path* takes the same arguments types as *source* in *parse()*.

CHAPTER 1

Root

class Root

Represents the root of the entire tree. An instance of this class is used to represent the entire tree in *parse()*/*write()* functions.

Users looking to create a new tree should intantiate this class.

All the following properties are read/write.

name
A string with the package's name.

image
A string with the path to the package's image.

author
A string with the package's author.

version
A string with the package's version.

description
A string with the package's description.

website
A string with the package's website.

conditions
A *Conditions* instance where it is checked whether the installer can start.

files
A *Files* instance that contains files/folders that will always be installed.

pages
A *Pages* instance that holds a list of installer pages.

file_patterns
A *FilePatterns* instance that contains a list of patterns that install files based on conditions.

installer(*[path[, game_version[, file_type]]]*)
A shortcut to creating an installer for this tree. For information on the arguments, see *Installer*.

CHAPTER 2

Conditions

class Conditions

This class contains a list of codnitions. The fulfillment of these conditions leads to some action described in the containing class.

There are four possible conditions:

- flag condition - checks whether a flag has a specific value. See *Flags*;
- file condition - checks whether a file is missing or otherwise;
- version condition - checks whether the game is at least the specified version;
- nested conditions - a *Conditions* objectm allowing for nested conditions.

Instances of this class are dict-like objects, but hashable. Conditions held by the instance are defined by the key and value used.

To add a version condition, the key must be *None* and the value a string with the version:

```
>>> conditions[None] = "1.0.0"
```

To add a flag condition, the key is a string with the flag name and value is a string with flag value:

```
>>> conditions["flag_name"] = "flag_value"
```

To add a file condition, the key is a string with the file path and the value is an enum *FileType* - this enum has *ACTIVE*, *INACTIVE* and *MISSING*:

```
>>> conditions["file_path"] = FileType.MISSING
```

Finally, to add a nested condition, the key is the object and the value is *None*:

```
>>> nested = Conditions()
>>> conditions[nested] = None
```

type

This property accepts the enum *ConditionType*. This enum has either *AND* and *OR*. If *AND*, then all the conditions must be true to fulfill this instance, if *OR* only one condition needs to be true.

CHAPTER 3

Files

class Files

The *Files* class is a container of files and folders to install. It produces dict-like objects that map file/folder sources to destination folder paths relative to the target folder (this target folder may vary per game/manager).

To add a file is simple:

```
>>> files["file_path"] = "dest"
```

to add a folder, however, you must add a trailing slash to the key:

```
>>> files["folder_path/"] = "dest"
```

CHAPTER 4

Pages

class Pages

This class produces list-like objects that hold *Page* instances.

order

This controls the order in which the *Page* objects appear. This property is an enum, *Order*, that has the values *ASCENDING*, *DESCENDING* and *EXPLICIT*. This orders the pages in this object according to their name. To note that only *EXPLICIT* preserves the order in this list.

CHAPTER 5

Page

class Page

This class produces list-like objects of *Group* instances.

These objects are the pages the user will eventually see when installing the mod.

name

A string with the page's name/title.

order

See *Pages.order*.

CHAPTER 6

Group

class Group

Another list-like class, of *Option* objects. Each of this class' instances represent a named section of a *Page*.

name

A string with the group's name/title.

type

This property controls which options the user may select in this section. It is controlled via enum, *GroupType*, and each value is quite self-explanatory: *ALL*, *ANY*, *ATLEASTONE*, *ATMOSTONE*, *EXACTLYONE*.

order

See *Pages.order*.

CHAPTER 7

Option

class Option

This class represents a single option the user can select.

name

A string with the option's short text.

description

A string with the option's long description.

image

A string with a relative path to an image.

files

A *Files* object with the files/folders to install if this option is selected.

flags

A *Flags* object with the flags to set if this option is selected.

type

This property controls the option type. It's usually an *OptionType* enum (and is recommended) which supports these values: *OPTIONAL*, *REQUIRED*, *RECOMMENDED*, *NOTUSABLE* or *COULDBEUSABLE*.

This property could also be a *Type* object, which allows for more complex type selection based on conditions.

CHAPTER 8

Flags

class Flags

This class produces dict-like objects that map flag names to values:

```
>>> flags = pyfomod.Flags()
>>> flags['name'] = 'value'
>>> dict(flags)
{'name': 'value'}
```

CHAPTER 9

Type

class Type

This class produces dict-like objects that map `Conditions` to *OptionType* (see `Option.type`) values. This class is used to find an appropriate type for an option - each pair's conditions are evaluated until one is met, which is the type used. If no conditions are evaluated to true, `default` is used as the option type.

default

The default *OptionType* (see `Option.type`) used in case no other suitable types are found.

CHAPTER 10

FilePatterns

class FilePatterns

This class produces dict-like objects that map *Conditions* to *Files*. This class is used after *Pages* when installing and installs the corresponding files for any conditions that were met.

CHAPTER 11

Validation

pyfomod allows the user to validate the fomod tree - it checks for common mistakes and incorrect values that, while valid, may lead to unexpected behaviour during user installation.

```
>>> with open("example/fomod/moduleconfig.xml", "r") as fp:
...     print(fp.read())
...
<config xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"␣
↪xsi:noNamespaceSchemaLocation="http://qconsulting.ca/fo3/ModConfig5.0.xsd">
<moduleName></moduleName>
<requiredInstallFiles>
    <file/>
</requiredInstallFiles>
</config>
```

You can check for warnings during parsing by passing a list to *parse()*:

```
>>> warnings = []
>>> pyfomod.parse("example", warnings)
<pyfomod.fomod.Root at 0x1f98dde2a88>
>>> for warning in warnings:
...     print(f"Title: {warning.title}")
...     print(f"Message: {warning.msg}")
...     print(f"Critical: {warning.critical}\n")
...
Title: Missing Source Attribute
Message: The 'source' attribute on the 'file' tag is required. This tag will be␣
↪skipped.
Critical: True
```

You can also check for warnings during runtime by calling the *validate()* method on any fomod object. Note that the possible errors produced in these two situations are different, so if you want to find every possible warning be sure to use both:

```
>>> root = pyfomod.parse("example")
>>> for warning in root.validate():
...     print(f"Title: {warning.title}")
...     print(f"Message: {warning.msg}")
...     print(f"Critical: {warning.critical}\n")
...
Title: Missing Installer Name
Message: This fomod does not have a name.
Critical: False

Title: Empty Fomod Tree
Message: This fomod is empty, nothing will be installed.
Critical: False
```

validate(***callbacks*)

This method validates the object and all its children, recursively. It returns a list of *ValidationWarning* with the errors it found.

The *callbacks* argument is a dict that maps strings to function objects. The keys of this dict should be *pyfomod* class names and the function objects should take a single argument - the instance the function is being run on - and return a list of *ValidationWarning* objects.

This argument is useful for adding more warnings to check for or even for running an arbitrary function recursively on the tree:

```
>>> # this callback will create a warning for all 'moduleName' tags
>>> def example_callback(instance):
...     title = "Example Title"
...     msg = "This is an example validation message!"
...     return [pyfomod.ValidationWarning(title, msg, instance)]
...
>>> root = pyfomod.parse("example")
>>> for warning in root.validate(Name=[example_callback]):
...     print(f"Title: {warning.title}")
...     print(f"Message: {warning.msg}")
...     print(f"Critical: {warning.critical}\n")
...
Title: Example Title
Message: This is an example validation message!
Critical: False

Title: Missing Installer Name
Message: This fomod does not have a name.
Critical: False

Title: Empty Fomod Tree
Message: This fomod is empty, nothing will be installed.
Critical: False
```

class ValidationWarning

The base class of all pyfomod warnings. Each instance of this class refers to an error present in the fomod tree.

title

A string with a suitable title.

msg

A string describing the error.

elem

The fomod object this error refers to.

critical
A boolean on whether this error refers to something that may interfere with the installation process or is merely aesthetic.

Each warning returned by pyfomod is a specific subclass to allow for better filtering.

class warnings.**InvalidEnumWarning**
Critical warning for when the fomod file has an attribute that evaluates to an enum, like *operator* attribute in the *dependencies* tag, and the value of this attribute does not match any of the possibilities.

class warnings.**DefaultAttributeWarning**
Warning for when an attribute is required but a sensible default can be used.

class warnings.**RequiredAttributeWarning**
Warning for when an attribute is required but a default *cannot* be found for it (such as file paths). This warning occurs during parsing and so this tag will *not* be parsed.

class warnings.**CommentsPresentWarning**
Warning for when comments exists in the fomod files. This exists because those comments will be deleted if the parsed tree is written.

class warnings.**InvalidSyntaxWarning**
Warning for generic fomod syntax errors that are not better covered by other warnings.

class warnings.**MissingInfoWarning**
Warning for when there is no *info.xml* file.

class warnings.**EmptyTreeWarning**
Warning for when a tree is empty - meaning nothing will be installed.

class warnings.**ImpossibleFlagWarning**
Warning for when there is a flag dependency for a flag that is never created.

class warnings.**InstallerNameWarning**
Warning for fomod trees without a name.

class warnings.**EmptyConditionsWarning**
Warning for empty conditions - these will not be written to prevent errors.

class warnings.**VersionDependencyWarning**
Warning for version dependencies that do not specify a version. These will not be written to prevent errors.

class warnings.**FileDependencyWarning**
Warning for file dependencies that do not specify a file to depend on. These will not be written to prevent errors.

class warnings.**UselessFlagsWarning**
Warning for flag dependencies used in *moduleDependencies* - these can never be created and so will always fail.

class warnings.**EmptySourceWarning**
Warning for files or folders that have no source path - this may lead to problems when installing.

class warnings.**MissingDestinationWarning**
Warning for files or folders that do not explicit specify the destination path. When omitted, the destination is assumed to be the same as the source path.

class warnings.**OrderWarning**
The *order* attribute in the *installSteps*, *optionalFileGroups* and *plugins* tags defines the order in which their children appear in the installer. Any value but *Explicit* will reorder the children to something different than the user specified, which is generally not intended. An omitted *order* attribute defaults to *Ascending*.

class warnings.**EmptyPageWarning**
Warning for when there are pages without groups.

class warnings.**PageNameWarning**
Pages without name.

class warnings.**EmptyGroupWarning**
Warning for when there are groups without options.

class warnings.**GroupNameWarning**
Groups without name.

class warnings.**AtLeastOneWarning**
A group that has the type *SelectAtLeastOne* but none of the options are selectable.

class warnings.**ExactlyOneMissingWarning**
A group that has the type *SelectExactlyOne* but none of the options are selectable.

class warnings.**ExactlyOneRequiredWarning**
A group that has the type *SelectExactlyOne* but at least two of the options are required.

class warnings.**AtMostOneWarning**
A group that has the type *SelectAtMostOne* but at least two of the options are required.

class warnings.**OptionNameWarning**
Options without name.

class warnings.**OptionDescriptionWarning**
Options without description.

class warnings.**EmptyOptionWarning**
Options that don't install files or set flags and therefore are useless.

class warnings.**EmptyTypeWarning**
A *dependencyType* tag that has no children and therefore cannot set a type.

CHAPTER 12

Fomod Installer

New in version 1.0.0.

You can start a non-gui installer from *pyfomod*. This will not actually install any files or modify your filesystem in any way. It follows the same format and conventions as the rest of *pyfomod* with one notable exception - the **priority** xml attribute that is listed as ignored in *Ignored Tags and Attributes* is used in determining which files to install.

You can continue to freely modify the fomod tree you passed to the installer with the exception of removing objects. These changes will be reflected live. In order to ensure maximum compatibility, you should use the objects the installer returns from its `Installer.next()` and `Installer.previous()` methods instead of `Page`, `Group` or `Option` - these are read-only equivalent to the corresponding classes in *pyfomod*.

To start, create an instance of `Installer`.

class Installer(*root*[, *path*[, *game_version*[, *file_type*]]])

Each instance of this class represents an ongoing installation. You can instance as many of these objects as you want, but keep in mind that modifications to a tree will be reflected on all installers that share them.

root is a required argument that represents the root of the fomod tree. You can pass a `Root` object which will be used directly by the installer. Any other than this will be passed along to `parse()` to produce a `Root` object.

If *path* is given, it will act as the root path for the fomod tree. Source lookups will be done using this path, although *pyfomod* will never modify any files. These lookups will allow `files()` to provide the user with a complete dictionary of file sources and destinations, sorted acording to priority (meaning folders will be walked recursively for files and empty folders). Otherwise only logical path computations will be made.

If *path* is not given but a string is passed as *root* then this will be assumed to be a root path for the fomod tree.

game_version should be a string with the current version of the game you're running this installer for.

file_type should be a function object that takes in a file name and returns a `FileType` concerning the file's presence in the target folder.

During instancing of this class, if the conditions in `Root.conditions` are not met, a `FailedCondition` exception might be raised. To get the first visible page, run `next()` with no arguments.

next([*selected_options*])

Use this method to get the next page of the installer. Pass a list of selected options as *selected_options*.

This will return an `InstallerPage` instance.

This returns `None` when the installer is finished.

previous()
Use this method to return to a previous page. Returns a tuple of `(InstallerPage, [previously_selected_options])`.

This returns `None` when the installer is at the start.

files()
Returns a dictionary that maps file sources (strings) to file destinations (strings). If *path* is provided to the installer in a manner described above then actual files (or folders if they are empty) are used in the deictionary, otherwise only logical operations are made with the folders in the fomod tree.

This should be called once the installer is finished but can be called at any time.

flags()
Returns a dictionary that maps flag values (strings) to current flag values (strings). Although this does not impact the installation the user may debug installers by calling this during the installation.

CHAPTER 13

Low-Level Access

Although *pyfomod* is a high-level library all data is preserved and is accessible through a private interface. This access is not recommended, may break pyfomod's normal use if mishandled and may change at any point with no deprecation or grace period.

All classes, regardless of whether they're mentioned above or referred here as "hidden", can be validated individually or written to a string via the *to_string* method.

All classes used in *pyfomod* that have a corresponding xml element hold data in similar ways:

- All initial attributes when parsing are stored in *self._attrib* - these may be overwritten when serializing the object;
- All unused children are stored in *self._children* - this is a dictionary of "tag" -> ({attribute dictionary}, "text")
- The line number of the original element is stored in *self.lineno* if the initial `parse()` function was passed the keyword argument *lineno=True*. Otherwise, *self.lineno* is *None*

The **info.xml** file's root is stored apart from **moduleconfig.xml**'s root, at *root._info*, where *root* is the object returned by `parse()`. Since there is no consensus on what the **info.xml** file should contain or even the format/schema, *pyfomod* assumes the user knows what it's loading and will respect the tag's case. The *root._info* object belongs to the *Info* class. This class has two methods that handle extracting and modifying information on this file: *get_text* and *set_text*. These assume the information is stored in the text of children of the <*fomod*> root element and search for a case-insensitive tag. The user is free to extract or modify information using the *_attrib* and *_children* attributes in the object.

13.1 Ignored Tags and Attributes

Some of the tags and attributes present in the fomod schema are ignored by the API both because they're either not very useful or have fallen out of use or in order to streamline user experience.

These are not removed or lost however, they're both accessible as described above.

The following tags are ignored:

- *fommDependency*

The following attributes are ignored:

- *position*, *colour* - [**moduleName**]
- *showImage*, *showFade*, *height* - [**moduleImage**]
- *alwaysInstall*, *installIfUsable*, *priority* - [**file**, **folder**]

Index