blender_addon_utils

Release 0.1

30350n

Mar 31, 2023

GETTING STARTED

Welcome to the blender_addon_utils documenation.

This is a utility module for blender addons to help automatically installing addon dependencies and checking for updates.

CHAPTER

ONE

INSTALLATION

Add the blender_addon_utils respository to your addon as a submodule:

```
$ git submodule add https://github.com/30350n/blender_addon_utils my_addon/blender_addon_
↪utils/
```

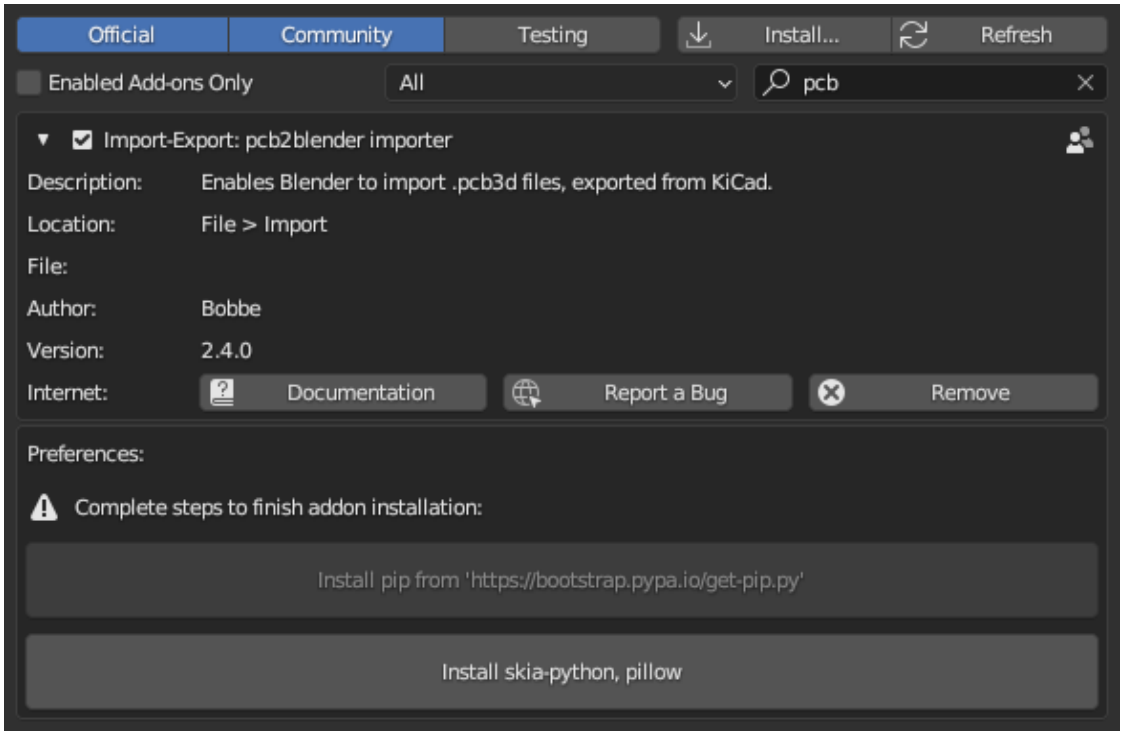
CHAPTER

TWO

HOW IT WORKS

You can declare required dependencies for your addon via `add_dependencies()`.

When you generate your registration code via `register_modules_factory()`, the function will automatically check if all previously declared dependencies are available.
If not, the resulting registration functions, won't register your modules, but the end-user will instead be presented with a panel like this, in the addon preferences:



The panel first offers the option to install pip and then the required dependencies.
After dependency installation is complete, your addon will be automatically reloaded.

By default, dependencies will be installed to a `site-packages` directory inside the root directory of the package that's calling the `add_dependencies()` function.

So if you call `add_dependencies()` in `my_addon/__init__.py`, dependencies will be installed to `my_addon/site-packages` (make sure your vcs ignores this folder).

CHAPTER

THREE

USAGE

To use the `add_dependencies()` and `register_modules_factory()` functionality, all your addon code will have to be arranged into submodules, which each submodule having its own `register()` and `unregister()` functions.

Your `__init__.py` should only contain minimal setup code, like this:

```
bl_info = {
    ...
}

from .blender_addon_utils import add_dependencies, register_modules_factory

deps = {
    "pip_package_name": "module_name",
}
add_dependencies(deps)

modules = ["my_submodule"]
register, unregister = register_modules_factory(modules)
```

Example `my_submodule.py` file:

```
import bpy

...

classes = (
    ...
)

def register():
    for cls in classes:
        bpy.utils.register_class(cls)

def unregister():
    for cls in reversed(classes):
        bpy.utils.unregister_class(cls)
```

CHAPTER

FOUR

REGISTER

register_modules_factory(*modules: Iterable[str]*) → Tuple[Callable, Callable]

Register addon submodules and handle dependency installation (if dependencies were added via *add_dependencies()*).

Parameters

deps (*list*) – List of module file names to register. Each file has to define its own `register()` and `unregister()` functions.

Returns

`register()` and `unregister()` function

add_dependencies(*deps: dict*, *path: Path | None = None*, *no_extra_deps: bool = False*)

Add dependencies to be installed via pip during registration.

Parameters

- **deps** (*dict*) – Pairs of pip package and module names (can be the same)
- **path** (*pathlib.Path*) – Directory to install the dependencies to (defaults to `calling_package_root/site_packages`)
- **no_extra_deps** (*bool*) – Ignore any sub-dependencies required (pip –no-deps option)

Example

CHAPTER

FIVE

ERROR_HELPER

class ErrorHelper

Mixin class for operators to simplify error/warning handling.

error(*msg: AnyStr*) → set

Report and print error message.

Parameters
msg (*str*) – The error message

Returns
{"CANCELLED"}

warning(*msg: AnyStr*)

Report and print warning.

Parameters
msg (*str*) – The warning message

Example:

```
class MY_ADDON_OT_my_operator(bpy.types.Operator, ErrorHelper):
    bl_idname = "my_addon.my_operator"
    bl_label = "My Operator"

    def execute(self, context):
        if context.mode == "EDIT_MESH":
            self.warning("we are in edit mode")
        elif context.mode == "POSE":
            return self.error("operator doesn't work in pose mode")
        return {"FINISHED"}
```

CHAPTER

SIX

DEBUG

class Timer(*name: AnyStr | None = None*)

Context manager for timing code snippets.

Parameters

name (*str*) – if specified, print it together with the elapsed time when exiting the context

Variables

time (*float*) – elapsed time (only available after exiting the context)

Example:

```
>>> with Timer("100000 divisions"):
...     result = [i / 42 for i in range(100000)]
...
    4112.60 us [100000 divisions]
```

PYTHON MODULE INDEX

INDEX