Skip to content

Utilities (bpy.utils)

This module contains utility functions specific to blender but not associated with blenders internal data.

SUBMODULES

bpy.utils submodule (bpy.utils.previews)

bpy.utils submodule (bpy.utils.units)

bpy.utils.**blend_paths(absolute=False, packed=False, local=False)**

Returns a list of paths to external files referenced by the loaded .blend file.

PARAMETERS:

- **absolute** (*bool*) – When true the paths returned are made absolute.
- **packed** (*bool*) – When true skip file paths for packed data.
- **local** (*bool*) – When true skip linked library paths.

RETURNS:

path list.

RETURN TYPE:

list[str]

bpy.utils.**escape_identifier(string)**

Simple string escaping function used for animation paths.

PARAMETERS:

string (*str*) – text

RETURNS:

The escaped string.

RETURN TYPE:

str

bpy.utils.**flip_name(name, strip_digits=False)**

Flip a name between left/right sides, useful for mirroring bone names.

PARAMETERS:

- **name** (*str*) – Bone name to flip.
- **strip_digits** (*bool*) – Whether to remove `.###` suffix.

RETURNS:

The flipped name.

RETURN TYPE:

str

bpy.utils.**unescape_identifier(string)**

Simple string un-escape function used for animation paths. This performs the reverse of `escape_identifier()` .

PARAMETERS:

string (*str*) – text

RETURNS:

The un-escaped string.

RETURN TYPE:

str

bpy.utils.**register_class(cls)**

Register a subclass of a Blender type class.

PARAMETERS:

cls (type[`bpy.types.Panel` | `bpy.types.UIList` | `bpy.types.Menu` | `bpy.types.Header` | `bpy.types.Operator` | `bpy.types.KeyingSetInfo` | `bpy.types.RenderEngine` | `bpy.types.AssetShelf` | `bpy.types.FileHandler` | `bpy.types.PropertyGroup` | `bpy.types.AddonPreferences`]) – Registerable Blender class type.

RAISES:

ValueError – if the class is not a subclass of a registerable blender class.

Note

If the class has a *register* class method it will be called before registration.

bpy.utils.**register_cli_command(id, execute)**

Register a command, accessible via the (`-c` / `--command`) command-line argument.

PARAMETERS:

- **id** (*str*) –

  The command identifier (must pass an `str.isidentifier` check).

  If the `id` is already registered, a warning is printed and the command is inaccessible to prevent accidents invoking the wrong command.

- **execute** (*callable*) – Callback, taking a single list of strings and returns an int. The arguments are built from all command-line arguments following the command id. The return value should be 0 for success, 1 on failure (specific error codes from the `os` module can also be used)

RETURNS:

The command handle which can be passed to `unregister_cli_command()` .

RETURN TYPE:

capsule

Custom Commands

Registering commands makes it possible to conveniently expose command line functionality via commands passed to (`-c` / `--command`).

```
import os

import bpy


def sysinfo_print():
    """
    Report basic system information.
    """

    import pprint
    import platform
    import textwrap

    width = 80
    indent = 2

    print("Blender {:s}".format(bpy.app.version_string))
    print("Running on: {:s}-{:s}".format(platform.platform(), platform.machine()))
    print("Processors: {!r}".format(os.cpu_count()))
    print()
    # Dump `bpy.app`.
    for attr in dir(bpy.app):
        if attr.startswith("_"):
            continue
        # Overly verbose.
        if attr in {"handlers", "build_cflags", "build_cxxflags"}:
            continue

        value = getattr(bpy.app, attr)
        if attr.startswith("build_"):
            pass
        elif isinstance(value, tuple):
            pass
        else:
            # Otherwise ignore.
            continue

        if isinstance(value, bytes):
            value = value.decode("utf-8", errors="ignore")

        if isinstance(value, str):
            pass
        elif isinstance(value, tuple) and hasattr(value, "__dir__"):
            value = {
                attr_sub: value_sub
                for attr_sub in dir(value)
                # Exclude built-ins.
                if not attr_sub.startswith(("_", "n_"))
                # Exclude methods.
                if not callable(value_sub := getattr(value, attr_sub))
            }
            value = pprint.pformat(value, indent=0, width=width)
        else:
            value = pprint.pformat(value, indent=0, width=width)

        print("{:s}:\n{:s}\n".format(attr, textwrap.indent(value, "  " * indent)))


def sysinfo_command(argv):
    if argv and argv[0] == "--help":
        print("Print system information & exit!")
        return 0

    sysinfo_print()
    return 0


cli_commands = []


def register():
    cli_commands.append(bpy.utils.register_cli_command("sysinfo", sysinfo_command))
```

```
def unregister():
    for cmd in cli_commands:
        bpy.utils.unregister_cli_command(cmd)
    cli_commands.clear()


if __name__ == "__main__":
    register()
```

Using Python Argument Parsing

This example shows how the Python `argparse` module can be used with a custom command.

Using `argparse` is generally recommended as it has many useful utilities and generates a `--help` message for your command.

```
import os
import sys

import bpy


def argparse_create():
    import argparse

    parser = argparse.ArgumentParser(
        prog=os.path.basename(sys.argv[0]) + " --command keyconfig_export",
        description="Write key-configuration to a file.",
    )

    parser.add_argument(
        "-o", "--output",
        dest="output",
        metavar='OUTPUT',
        type=str,
        help="The path to write the keymap to.",
        required=True,
    )

    parser.add_argument(
        "-a", "--all",
        dest="all",
        action="store_true",
        help="Write all key-maps (not only customized key-maps).",
        required=False,
    )

    return parser


def keyconfig_export(argv):
    parser = argparse_create()
    args = parser.parse_args(argv)

    # Ensure the key configuration is loaded in background mode.
    bpy.utils.keyconfig_init()
    bpy.ops.preferences.keyconfig_export(
        filepath=args.output,
        all=args.all,
    )

    return 0


cli_commands = []


def register():
    cli_commands.append(bpy.utils.register_cli_command("keyconfig_export", keyconfig_ex


def unregister():
    for cmd in cli_commands:
        bpy.utils.unregister_cli_command(cmd)
    cli_commands.clear()


if __name__ == "__main__":
    register()
```

bpy.utils.**unregister_cli_command(handle)**

Unregister a CLI command.

PARAMETERS:

handle (*capsule*) – The return value of `register_cli_command()` .

bpy.utils.**resource_path(type, major=bpy.app.version[0], minor=bpy.app.version[1])**

Return the base path for storing system files.

PARAMETERS:

- **type** (*str*) – string in [‘USER’, ‘LOCAL’, ‘SYSTEM’].
- **major** (*int*) – major version, defaults to current.
- **minor** (*str*) – minor version, defaults to current.

RETURNS:

the resource path (not necessarily existing).

RETURN TYPE:

str

bpy.utils.**unregister_class(cls)**

Unload the Python class from blender.

PARAMETERS:

cls (type[`bpy.types.Panel` | `bpy.types.UIList` | `bpy.types.Menu` | `bpy.types.Header` | `bpy.types.Operator` | `bpy.types.KeyingSetInfo` | `bpy.types.RenderEngine` | `bpy.types.AssetShelf` | `bpy.types.FileHandler` | `bpy.types.PropertyGroup` | `bpy.types.AddonPreferences`]) – Blender type class, see `bpy.utils.register_class` for classes which can be registered.

Note

If the class has an *unregister* class method it will be called before unregistering.

bpy.utils.**keyconfig_init()**

bpy.utils.**keyconfig_set(filepath, *, report=None)**

bpy.utils.**load_scripts(*, reload_scripts=False, refresh_scripts=False, extensions=True)**

Load scripts and run each modules register function.

PARAMETERS:

- **reload_scripts** (*bool*) – Causes all scripts to have their unregister method called before loading.
- **refresh_scripts** (*bool*) – only load scripts which are not already loaded as modules.
- **extensions** (*bool*) – Loads additional scripts (add-ons & app-templates).

bpy.utils.**modules_from_path(path, loaded_modules)**

Load all modules in a path and return them as a list.

PARAMETERS:

- **path** (*str*) – this path is scanned for scripts and packages.
- **loaded_modules** (*set[ModuleType]*) – already loaded module names, files matching these names will be ignored.

RETURNS:

all loaded modules.

RETURN TYPE:

list[ModuleType]

bpy.utils.**preset_find(name, preset_path, *, display_name=False, ext='.py')**

bpy.utils.**preset_paths(subdir)**

Returns a list of paths for a specific preset.

PARAMETERS:

subdir (*str*) – preset subdirectory (must not be an absolute path).

RETURNS:

Script paths.

RETURN TYPE:

list[str]

bpy.utils.**refresh_script_paths()**

Run this after creating new script paths to update sys.path

bpy.utils.**app_template_paths(*, path=None)**

Returns valid application template paths.

PARAMETERS:

path (*str*) – Optional subdir.

RETURNS:

App template paths.

RETURN TYPE:

Iterator[str]

bpy.utils.**time_from_frame(frame, *, fps=None, fps_base=None)**

Returns the time from a frame number .

If *fps* and *fps_base* are not given the current scene is used.

PARAMETERS:
frame (*int* | *float*) – number.

RETURNS:
the time in seconds.

RETURN TYPE:
datetime.timedelta

bpy.utils.**register_manual_map(manual_hook)**

bpy.utils.**unregister_manual_map(manual_hook)**

bpy.utils.**register_preset_path(path)**

Register a preset search path.

PARAMETERS:
path (*str*) –

preset directory (must be an absolute path).

This path must contain a "presets" subdirectory which will typically contain presets for add-ons.

You may call `bpy.utils.register_preset_path(os.path.dirname(__file__))` from an add-ons `__init__.py` file. When the `__init__.py` is in the same location as a `presets` directory. For example an operators preset would be located under: `presets/operator/{operator.id}/` where `operator.id` is the `bl_idname` of the operatc

RETURNS:
success

RETURN TYPE:
bool

bpy.utils.**unregister_preset_path(path)**

Unregister a preset search path.

PARAMETERS:
path (*str*) –

preset directory (must be an absolute path).

This must match the registered path exactly.

RETURNS:
success

RETURN TYPE:
bool

bpy.utils.**register_classes_factory(classes)**

Utility function to create register and unregister functions which simply registers and unregisters a sequence of classes.

bpy.utils.**register_submodule_factory(module_name, submodule_names)**

Utility function to create register and unregister functions which simply load submodules, calling their register & unregister functions.

Note

Modules are registered in the order given, unregistered in reverse order.

PARAMETERS:
- **module_name** (*str*) – The module name, typically `__name__`

- **module_name** (*str*) – The module name, typically `__name__`.
- **submodule_names** (*list[str]*) – List of submodule names to load and unload.

RETURNS:

register and unregister functions.

RETURN TYPE:

tuple[Callable[[], None], Callable[[], None]]

bpy.utils.**register_tool(tool_cls, *, after=None, separator=False, group=False)**

Register a tool in the toolbar.

PARAMETERS:

- **tool_cls** (type[`bpy.types.WorkSpaceTool`]) – A tool subclass.
- **after** (*Sequence[str]* | *set[str]* | *None*) – Optional identifiers this tool will be added after.
- **separator** (*bool*) – When true, add a separator before this tool.
- **group** (*bool*) – When true, add a new nested group of tools.

bpy.utils.**make_rna_paths(struct_name, prop_name, enum_name)**

Create RNA "paths" from given names.

PARAMETERS:

- **struct_name** (*str*) – Name of a RNA struct (like e.g. "Scene").
- **prop_name** (*str*) – Name of a RNA struct's property.
- **enum_name** (*str*) – Name of a RNA enum identifier.

RETURNS:

A triple of three "RNA paths" (most_complete_path, "struct.prop", "struct.prop:'enum'"). If no enum_name is given, the third element will always be void.

RETURN TYPE:

tuple[str, str, str]

bpy.utils.**manual_map()**

bpy.utils.**manual_language_code(default='en')**

RETURNS:

The language code used for user manual URL component based on the current language user-preference, falling back to the `default` whe
unavailable.

RETURN TYPE:

str

bpy.utils.**script_path_user()**

returns the env var and falls back to home dir or None

bpy.utils.**extension_path_user(package, *, path='', create=False)**

Return a user writable directory associated with an extension.

> Note
>
> This allows each extension to have it's own user directory to store files.
>
> The location of the extension it self is not a suitable place to store files because it is cleared each upgrade and the users may not have write permissions to the repository (typically "System" repositories).

PARAMETERS:

- **package** (*str*) – The `__package__` of the extension.
- **path** (*str*) – Optional subdirectory.

- **create** (*bool*) – Treat the path as a directory and create it if its not existing.

RETURNS:
a path.

RETURN TYPE:
str

bpy.utils.**script_paths(*, subdir=None, user_pref=True, check_all=False, use_user=True, use_system_environment=True)**

Returns a list of valid script paths.

PARAMETERS:

- **subdir** (*str*) – Optional subdir.
- **user_pref** (*bool*) – Include the user preference script paths.
- **check_all** (*bool*) – Include local, user and system paths rather just the paths Blender uses.
- **use_user** (*bool*) – Include user paths
- **use_system_environment** (*bool*) – Include BLENDER_SYSTEM_SCRIPTS variable path

RETURNS:
script paths.

RETURN TYPE:
list[str]

bpy.utils.**smpte_from_frame(frame, *, fps=None, fps_base=None)**

Returns an SMPTE formatted string from the *frame*: `HH:MM:SS:FF`.

If *fps* and *fps_base* are not given the current scene is used.

PARAMETERS:
frame (*int* | *float*) – frame number.

RETURNS:
the frame string.

RETURN TYPE:
str

bpy.utils.**smpte_from_seconds(time, *, fps=None, fps_base=None)**

Returns an SMPTE formatted string from the *time*: `HH:MM:SS:FF`.

If *fps* and *fps_base* are not given the current scene is used.

PARAMETERS:
time (*int* | *float* | *datetime.timedelta*) – time in seconds.

RETURNS:
the frame string.

RETURN TYPE:
str

bpy.utils.**unregister_tool(tool_cls)**

bpy.utils.**user_resource(resource_type, *, path='', create=False)**

Return a user resource path (normally from the users home directory).

PARAMETERS:

- **resource_type** (*str*) – Resource type in [‘DATAFILES’, ‘CONFIG’, ‘SCRIPTS’, ‘EXTENSIONS’].
- **path** (*str*) – Optional subdirectory.

- **create** (*bool*) – Treat the path as a directory and create it if its not existing.

RETURNS:

a path.

RETURN TYPE:

str

bpy.utils.**execfile(filepath, *, mod=None)**

Execute a file path as a Python script.

PARAMETERS:

- **filepath** (*str*) – Path of the script to execute.
- **mod** (*ModuleType* | *None*) – Optional cached module, the result of a previous execution.

RETURNS:

The module which can be passed back in as `mod`.

RETURN TYPE:

ModuleType

bpy.utils.**expose_bundled_modules()**

For Blender as a Python module, add bundled VFX library python bindings to `sys.path`. These may be used instead of dedicated packages, t
ensure the libraries are compatible with Blender.

Copyright © Blender Authors
Made with Furo