Skip to content

FileHandler(bpy_struct)

Basic FileHandler for Operator that imports just one file

When creating a `Operator` that imports files, you may want to add them ‘drag-and-drop’ support, File Handlers helps to define a set of files extensions (`FileHandler.bl_file_extensions`) that the `Operator` support and a `FileHandler.poll_drop` function that can be used to check in what specific context the `Operator` can be invoked with ‘drag-and-drop’ filepath data.

Same as operators that uses the file select window, this operators required a set of properties, when the `Operator` can import just one file per execution it needs to define the following property:

```
filepath: bpy.props.StringProperty(subtype='FILE_PATH')
```

This `filepath` property now will be used by the `FileHandler` to set the ‘drag-and-drop’ filepath data.

```
import bpy


class CurveTextImport(bpy.types.Operator):
    """ Test importer that creates a text object from a .txt file """
    bl_idname = "curve.text_import"
    bl_label = "Import a text file as text object"

    """
    This Operator supports import one .txt file at the time, we need the
    following filepath property that the file handler will use to set file path data.
    """
    filepath: bpy.props.StringProperty(subtype='FILE_PATH', options={'SKIP_SAVE'})

    @classmethod
    def poll(cls, context):
        return (context.area and context.area.type == "VIEW_3D")

    def execute(self, context):
        """ Calls to this Operator can set unfiltered filepaths, ensure the file extension
        if not self.filepath or not self.filepath.endswith(".txt"):
            return {'CANCELLED'}

        with open(self.filepath) as file:
            text_curve = bpy.data.curves.new(type="FONT", name="Text")
            text_curve.body = ''.join(file.readlines())
            text_object = bpy.data.objects.new(name="Text", object_data=text_curve)
            bpy.context.scene.collection.objects.link(text_object)
        return {'FINISHED'}

    """
    By default the file handler invokes the operator with the filepath property set.
    In this example if this property is set the operator is executed, if not the
    file select window is invoked.
    This depends on setting ``options={'SKIP_SAVE'}`` to the property options to avoid
    to reuse filepath data between operator calls.
    """
    def invoke(self, context, event):
        if self.filepath:
            return self.execute(context)
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}


class CURVE_FH_text_import(bpy.types.FileHandler):
    bl_idname = "CURVE_FH_text_import"
    bl_label = "File handler for curve text object import"
    bl_import_operator = "curve.text_import"
    bl_file_extensions = ".txt"

    @classmethod
    def poll_drop(cls, context):
        return (context.area and context.area.type == 'VIEW_3D')


bpy.utils.register_class(CurveTextImport)
bpy.utils.register_class(CURVE_FH_text_import)
```

Basic FileHandler for Operator that imports multiple files

Also operators can be invoked with multiple files from ‘drag-and-drop’, but for this it is require to define the following properties:

```
directory: StringProperty(subtype='FILE_PATH')
files: CollectionProperty(type=bpy.types.OperatorFileListElement)
```

This `directory` and `files` properties now will be used by the `FileHandler` to set ‘drag-and-drop’ filepath data.

```
import bpy
from bpy_extras.io_utils import ImportHelper
from mathutils import Vector


class ShaderScriptImport(bpy.types.Operator, ImportHelper):
    """Test importer that creates scripts nodes from .txt files"""
    bl_idname = "shader.script_import"
    bl_label = "Import a text file as a script node"

    """
    This Operator can import multiple .txt files, we need following directory and files
    properties that the file handler will use to set files path data
    """
    directory: bpy.props.StringProperty(subtype='FILE_PATH', options={'SKIP_SAVE', 'HIDDEN
    files: bpy.props.CollectionProperty(type=bpy.types.OperatorFileListElement, options={'

    """Allow the user to select if the node's label is set or not"""
    set_label: bpy.props.BoolProperty(name="Set Label", default=False)

    @classmethod
    def poll(cls, context):
        return (
            context.region and context.region.type == 'WINDOW' and
```

```python
            context.area and context.area.ui_type == 'ShaderNodeTree' and
            context.object and context.object.type == 'MESH' and
            context.material
        )

    def execute(self, context):
        """ The directory property need to be set. """
        if not self.directory:
            return {'CANCELLED'}
        x = 0.0
        y = 0.0
        for file in self.files:
            """
            Calls to the operator can set unfiltered file names,
            ensure the file extension is .txt
            """
            if file.name.endswith(".txt"):
                node_tree = context.material.node_tree
                text_node = node_tree.nodes.new(type="ShaderNodeScript")
                text_node.mode = 'EXTERNAL'
                import os
                filepath = os.path.join(self.directory, file.name)
                text_node.filepath = filepath
                text_node.location = Vector((x, y))

                # Set the node's title to the file name
                if self.set_label:
                    text_node.label = file.name

                x += 20.0
                y -= 20.0
        return {'FINISHED'}

    # Use ImportHelper's invoke_popup() to handle the invocation so that this operator's p
    # are shown in a popup. This allows the user to configure additional settings on the c
    # the `set_label` property. Consider having a draw() method on the operator in order t
    # properties in the UI appropriately.
    #
    # If filepath information is not provided the file select window will be invoked inste

    def invoke(self, context, event):
        return self.invoke_popup(context)


class SHADER_FH_script_import(bpy.types.FileHandler):
    bl_idname = "SHADER_FH_script_import"
    bl_label = "File handler for shader script node import"
    bl_import_operator = "shader.script_import"
    bl_file_extensions = ".txt"

    @classmethod
    def poll_drop(cls, context):
        return (
            context.region and context.region.type == 'WINDOW' and
            context.area and context.area.ui_type == 'ShaderNodeTree'
        )


bpy.utils.register_class(ShaderScriptImport)
bpy.utils.register_class(SHADER_FH_script_import)
```

base class — `bpy_struct`

subclasses — `IMAGE_FH_drop_handler`, `IO_FH_fbx`, `IO_FH_gltf2`, `NODE_FH_image_node`, `SEQUENCER_FH_image_strip`, `SEQUENCER_FH_movie_strip`, `SEQUENCER_FH_sound_strip`, `VIEW3D_FH_camera_background_image`, `VIEW3D_FH_empty_image`, `VIEW3D_FH_vdb_volume`

class bpy.types.**FileHandler(bpy_struct)**

Extends functionality to operators that manages files, such as adding drag and drop support

bl_export_operator

Operator that can handle export for files with the extensions given in bl_file_extensions

TYPE:

string, default “”, (never None)

bl_file_extensions

Formatted string of file extensions supported by the file handler, each extension should start with a “.” and be separated by “;”. For Example: *“.blend;.ble”*

TYPE:

string, default “”, (never None)

bl_idname

If this is set, the file handler gets a custom ID, otherwise it takes the name of the class used to define the file handler (for example, if the class name is “OBJECT_FH_hello”, and bl_idname is not set by the script, then bl_idname = “OBJECT_FH_hello”)

TYPE:

string, default “”, (never None)

bl_import_operator

Operator that can handle import for files with the extensions given in bl_file_extensions

TYPE:

string, default “”, (never None)

bl_label

The file handler label

TYPE:

string, default “”, (never None)

classmethod poll_drop(context)

If this method returns True, can be used to handle the drop of a drag-and-drop action

RETURN TYPE:

boolean

classmethod bl_rna_get_subclass(id, default=None)

PARAMETERS:

id (*str*) – The RNA type identifier.

RETURNS:

The RNA type or default when not found.

RETURN TYPE:

`bpy.types.Struct` subclass

classmethod bl_rna_get_subclass_py(id, default=None)

PARAMETERS:

id (*str*) – The RNA type identifier.

RETURNS:

The class or default when not found.

RETURN TYPE:

type

Inherited Properties

- `bpy_struct.id_data`

Inherited Functions

- `bpy_struct.as_pointer`
- `bpy_struct.driver_add`
- `bpy_struct.driver_remove`
- `bpy_struct.get`
- `bpy_struct.id_properties_clear`
- `bpy_struct.id_properties_ensure`
- `bpy_struct.id_properties_ui`
- `bpy_struct.is_property_hidden`
- `bpy_struct.is_property_overridable_library`
- `bpy_struct.is_property_readonly`
- `bpy_struct.is_property_set`
- `bpy_struct.items`
- `bpy_struct.keyframe_delete`
- `bpy_struct.keyframe_insert`
- `bpy_struct.keys`
- `bpy_struct.path_from_id`
- `bpy_struct.path_resolve`
- `bpy_struct.pop`
- `bpy_struct.property_overridable_library_set`
- `bpy_struct.property_unset`
- `bpy_struct.type_recast`
- `bpy_struct.values`