Operator(bpy_struct)

Basic Operator Example

This script shows simple operator which prints a message.

Since the operator only has an `Operator.execute` function it takes no user input.

The function should return `{'FINISHED'}` or `{'CANCELLED'}`, the latter meaning that operator execution was aborted without making any changes, and that no undo step will created (see next example for more info about undo).

Note

Operator subclasses must be registered before accessing them from blender.

```
import bpy


class HelloWorldOperator(bpy.types.Operator):
    bl_idname = "wm.hello_world"
    bl_label = "Minimal Operator"

    def execute(self, context):
        print("Hello World")
        return {'FINISHED'}


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(HelloWorldOperator.bl_idname, text="Hello World Operator")


# Register and add to the view menu (required to also use F3 search "Hello World Operator"
bpy.utils.register_class(HelloWorldOperator)
bpy.types.VIEW3D_MT_view.append(menu_func)

# Test call to the newly defined operator.
bpy.ops.wm.hello_world()
```

Modifying Blender Data & Undo

Any operator modifying Blender data should enable the `'UNDO'` option. This will make Blender automatically create an undo step when the operator finishes its `execute` (or `invoke`, see below) functions, and returns `{'FINISHED'}`.

Otherwise, no undo step will be created, which will at best corrupt the undo stack and confuse the user (since modifications done by the operator may either not be undoable, or be undone together with other edits done before). In many cases, this can even lead to data corruption and crashes.

Note that when an operator returns `{'CANCELLED'}`, no undo step will be created. This means that if an error occurs *after* modifying some data already, it is better to return `{'FINISHED'}`, unless it is possible to fully undo the changes before returning.

Note

Most examples in this page do not do any edit to Blender data, which is why it is safe to keep the default `bl_options` value for these operators.

Note

In some complex cases, the automatic undo step created on operator exit may not be enough. For example, if the operator does mode switching, or calls other operators that should create an extra undo step, etc.

Such manual undo push is possible using the `bpy.ops.ed.undo_push` function. Be careful though, this is considered an advanced feature and requires some understanding of the actual undo system in Blender code.

```
import bpy


class DataEditOperator(bpy.types.Operator):
    bl_idname = "object.data_edit"
    bl_label = "Data Editing Operator"
    # The default value is only 'REGISTER', 'UNDO' is mandatory when Blender data is modif
    # (and does require 'REGISTER' as well).
    bl_options = {'REGISTER', 'UNDO'}

    def execute(self, context):
        context.object.location.x += 1.0
        return {'FINISHED'}


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(DataEditOperator.bl_idname, text="Blender Data Editing Operator")


# Register.
bpy.utils.register_class(DataEditOperator)
bpy.types.VIEW3D_MT_view.append(menu_func)

# Test call to the newly defined operator.
bpy.ops.object.data_edit()
```

Invoke Function

`Operator.invoke` is used to initialize the operator from the context at the moment the operator is called. invoke() is typically used to assign properties which are then used by execute(). Some operators don't have an execute() function, removing the ability to be repeated from a script or macr

This example shows how to define an operator which gets mouse input to execute a function and that this operator can be invoked or executed from the python api.

Also notice this operator defines its own properties, these are different to typical class properties because blender registers them with the operator, to us as arguments when called, saved for operator undo/redo and automatically added into the user interface.

```
import bpy


class SimpleMouseOperator(bpy.types.Operator):
    """ This operator shows the mouse location,
        this string is used for the tooltip and API docs
    """
    bl_idname = "wm.mouse_position"
    bl_label = "Invoke Mouse Operator"

    x: bpy.props.IntProperty()
    y: bpy.props.IntProperty()
    def execute(self, context):
        # rather than printing, use the report function,
        # this way the message appears in the header,
        self.report({'INFO'}, "Mouse coords are {:d} {:d}".format(self.x, self.y))
        return {'FINISHED'}

    def invoke(self, context, event):
        self.x = event.mouse_x
        self.y = event.mouse_y
        return self.execute(context)


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(SimpleMouseOperator.bl_idname, text="Simple Mouse Operator")


# Register and add to the view menu (required to also use F3 search "Simple Mouse Operator
bpy.utils.register_class(SimpleMouseOperator)
bpy.types.VIEW3D_MT_view.append(menu_func)

# Test call to the newly defined operator.
# Here we call the operator and invoke it, meaning that the settings are taken
# from the mouse.
bpy.ops.wm.mouse_position('INVOKE_DEFAULT')

# Another test call, this time call execute() directly with pre-defined settings.
bpy.ops.wm.mouse_position('EXEC_DEFAULT', x=20, y=66)
```

Calling a File Selector

This example shows how an operator can use the file selector.

Notice the invoke function calls a window manager method and returns `{'RUNNING_MODAL'}`, this means the file selector stays open and the operator does not exit immediately after invoke finishes.

The file selector runs the operator, calling `Operator.execute` when the user confirms.

The `Operator.poll` function is optional, used to check if the operator can run.

```
import bpy


class ExportSomeData(bpy.types.Operator):
    """Test exporter which just writes hello world"""
    bl_idname = "export.some_data"
    bl_label = "Export Some Data"

    filepath: bpy.props.StringProperty(subtype="FILE_PATH")

    @classmethod
    def poll(cls, context):
        return context.object is not None

    def execute(self, context):
        file = open(self.filepath, 'w')
        file.write("Hello World " + context.object.name)
        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator_context = 'INVOKE_DEFAULT'
    self.layout.operator(ExportSomeData.bl_idname, text="Text Export Operator")


# Register and add to the file selector (required to also use F3 search "Text Export Opera
bpy.utils.register_class(ExportSomeData)
bpy.types.TOPBAR_MT_file_export.append(menu_func)


# test call
bpy.ops.export.some_data('INVOKE_DEFAULT')
```

Dialog Box

This operator uses its `Operator.invoke` function to call a popup.

```
import bpy


class DialogOperator(bpy.types.Operator):
    bl_idname = "object.dialog_operator"
    bl_label = "Simple Dialog Operator"

    my_float: bpy.props.FloatProperty(name="Some Floating Point")
    my_bool: bpy.props.BoolProperty(name="Toggle Option")
    my_string: bpy.props.StringProperty(name="String Value")

    def execute(self, context):
        message = "Popup Values: {:f}, {:d}, '{:s}'".format(
            self.my_float, self.my_bool, self.my_string,
        )
        self.report({'INFO'}, message)
        return {'FINISHED'}

    def invoke(self, context, event):
        wm = context.window_manager
        return wm.invoke_props_dialog(self)


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(DialogOperator.bl_idname, text="Dialog Operator")
```

```
# Register and add to the object menu (required to also use F3 search "Dialog Operator" fo
bpy.utils.register_class(DialogOperator)
bpy.types.VIEW3D_MT_object.append(menu_func)

# Test call.
bpy.ops.object.dialog_operator('INVOKE_DEFAULT')
```

Custom Drawing

By default operator properties use an automatic user interface layout. If you need more control you can create your own layout with a `Operator.draw` function.

This works like the `Panel` and `Menu` draw functions, its used for dialogs and file selectors.

```
import bpy


class CustomDrawOperator(bpy.types.Operator):
    bl_idname = "object.custom_draw"
    bl_label = "Simple Modal Operator"

    filepath: bpy.props.StringProperty(subtype="FILE_PATH")

    my_float: bpy.props.FloatProperty(name="Float")
    my_bool: bpy.props.BoolProperty(name="Toggle Option")
    my_string: bpy.props.StringProperty(name="String Value")

    def execute(self, context):
        print("Test", self)
        return {'FINISHED'}

    def invoke(self, context, event):
        wm = context.window_manager
        return wm.invoke_props_dialog(self)

    def draw(self, context):
        layout = self.layout
        col = layout.column()
        col.label(text="Custom Interface!")

        row = col.row()
        row.prop(self, "my_float")
        row.prop(self, "my_bool")

        col.prop(self, "my_string")


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(CustomDrawOperator.bl_idname, text="Custom Draw Operator")


# Register and add to the object menu (required to also use F3 search "Custom Draw Operato
bpy.utils.register_class(CustomDrawOperator)
bpy.types.VIEW3D_MT_object.append(menu_func)

# test call
bpy.ops.object.custom_draw('INVOKE_DEFAULT')
```

Modal Execution

This operator defines a `Operator.modal` function that will keep being run to handle events until it returns `{'FINISHED'}` or `{'CANCELLED'}`.

Modal operators run every time a new event is detected, such as a mouse click or key press. Conversely, when no new events are detected, the modal operator will not run. Modal operators are especially useful for interactive tools, an operator can have its own state where keys toggle options as the operator runs. Grab, Rotate, Scale, and Fly-Mode are examples of modal operators.

`Operator.invoke` is used to initialize the operator as being active by returning `{'RUNNING_MODAL'}`, initializing the modal loop.

Notice `__init__()` and `__del__()` are declared. For other operator types they are not useful but for modal operators they will be called befo the `Operator.invoke` and after the operator finishes. Also see the class construction and destruction section

```
import bpy


class ModalOperator(bpy.types.Operator):
    bl_idname = "object.modal_operator"
    bl_label = "Simple Modal Operator"
    bl_options = {'REGISTER', 'UNDO'}

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print("Start")

    def __del__(self):
        print("End")
        super().__del__()

    def execute(self, context):
        context.object.location.x = self.value / 100.0
        return {'FINISHED'}

    def modal(self, context, event):
        if event.type == 'MOUSEMOVE':  # Apply
            self.value = event.mouse_x
            self.execute(context)
        elif event.type == 'LEFTMOUSE':  # Confirm
            return {'FINISHED'}
        elif event.type in {'RIGHTMOUSE', 'ESC'}:  # Cancel
            # Revert all changes that have been made
            context.object.location.x = self.init_loc_x
            return {'CANCELLED'}

        return {'RUNNING_MODAL'}

    def invoke(self, context, event):
        self.init_loc_x = context.object.location.x
        self.value = event.mouse_x
        self.execute(context)
        context.window_manager.modal_handler_add(self)
        return {'RUNNING_MODAL'}


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(ModalOperator.bl_idname, text="Modal Operator")


# Register and add to the object menu (required to also use F3 search "Modal Operator" for
bpy.utils.register_class(ModalOperator)
bpy.types.VIEW3D_MT_object.append(menu_func)

# test call
bpy.ops.object.modal_operator('INVOKE_DEFAULT')
```

Enum Search Popup

You may want to have an operator prompt the user to select an item from a search field, this can be done using `bpy.types.Operator.invoke_search_popup`.

```
import bpy
from bpy.props import EnumProperty


class SearchEnumOperator(bpy.types.Operator):
    bl_idname = "object.search_enum_operator"
    bl_label = "Search Enum Operator"
    bl_property = "my_search"

    my_search: EnumProperty(
        name="My Search",
        items=(
            ('FOO', "Foo", ""),
            ('BAR', "Bar", ""),
            ('BAZ', "Baz", ""),
        ),
    )

    def execute(self, context):
        self.report({'INFO'}, "Selected:" + self.my_search)
        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.invoke_search_popup(self)
        return {'RUNNING_MODAL'}


# Only needed if you want to add into a dynamic menu.
def menu_func(self, context):
    self.layout.operator(SearchEnumOperator.bl_idname, text="Search Enum Operator")
```

```
# Register and add to the object menu (required to also use F3 search "Search Enum Operato
bpy.utils.register_class(SearchEnumOperator)
bpy.types.VIEW3D_MT_object.append(menu_func)

# test call
bpy.ops.object.search_enum_operator('INVOKE_DEFAULT')
```

base class — `bpy_struct`

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

Storage of an operator being executed, or registered after execution

bl_cursor_pending

Cursor to use when waiting for the user to select a location to activate the operator (when `bl_options` has `DEPENDS_ON_CURSOR` set)

TYPE:

enum in Window Cursor Items, default 'DEFAULT'

bl_description

TYPE:

string, default "", (never None)

bl_idname

TYPE:

string, default "", (never None)

bl_label

TYPE:

string, default "", (never None)

bl_options

Options for this operator type

TYPE:

enum set in Operator Type Flag Items, default {'REGISTER'}

bl_translation_context

TYPE:

string, default "Operator", (never None)

bl_undo_group

TYPE:

string, default "", (never None)

has_reports

Operator has a set of reports (warnings and errors) from last execution

TYPE:

boolean, default False, (readonly)

layout

TYPE:

`UILayout`, (readonly)

macros

TYPE:

`bpy_prop_collection` of `Macro`, (readonly)

name

TYPE:

string, default "", (readonly, never None)

options

Runtime options

TYPE:

`OperatorOptions`, (readonly, never None)

properties

TYPE:

`OperatorProperties`, (readonly, never None)

bl_property

The name of a property to use as this operators primary property. Currently this is only used to select the default property when expanding an operator into a menu.

TYPE:

str

report(type, message)

report

PARAMETERS:

- **type** (enum set in Wm Report Items) – Type
- **message** (*string, (never None)*) – Report Message

is_repeat()

is_repeat

RETURNS:

result

RETURN TYPE:

boolean

classmethod poll(context)

Test if the operator can be called or not

RETURN TYPE:

boolean

execute(context)

Execute the operator

RETURNS:

result

RETURN TYPE:

enum set in Operator Return Items

check(context)

Check the operator settings, return True to signal a change to redraw

RETURNS:
result

RETURN TYPE:
boolean

invoke(context, event)

Invoke the operator

RETURNS:
result

RETURN TYPE:
enum set in Operator Return Items

modal(context, event)

Modal operator function

RETURNS:
result

RETURN TYPE:
enum set in Operator Return Items

draw(context)

Draw function for the operator

cancel(context)

Called when the operator is canceled

classmethod description(context, properties)

Compute a description string that depends on parameters

RETURNS:
result

RETURN TYPE:
string

as_keywords(*, ignore=())

Return a copy of the properties as a dictionary

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

The class or default when not found.

RETURN TYPE:

type

classmethod poll_message_set(message, *args)

Set the message to show in the tool-tip when poll fails.

When message is callable, additional user defined positional arguments are passed to the message function.

PARAMETERS:

message (*str* | *Callable[[Any, ...], str* | *None]*) – The message or a function that returns the message.

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`

References

- `bpy.context.active_operator`
- `SpaceFileBrowser.active_operator`
- `SpaceFileBrowser.operator`
- `Window.modal_operators`
- `WindowManager.fileselect_add`
- `WindowManager.invoke_confirm`
- `WindowManager.invoke_popup`
- `WindowManager.invoke_props_dialog`
- `WindowManager.invoke_props_popup`
- `WindowManager.invoke_search_popup`
- `WindowManager.modal_handler_add`
- `WindowManager.operators`