FreeCAD Scripted Objects Modern API

META	VALUE	
generated	2025-04-22 08:45:55.383971	
author	Frank David Martínez Muñoz	
copyright	(c) 2024 Frank David Martínez Muñoz.	
license	LGPL 2.1	
version	1.0.0-beta6	
min_python	3.10	
min_freecad	0.22	
contributors	Gaël Écorchard	

TABLE OF CONTENTS

- FreeCAD Scripted Objects Modern API
- Preliminaries
 - Disclaimer
 - Audience
 - Goals
 - Non Goals
- Features
- General Scripted Object Architecture
 - Scripted Object Overview diagram
 - App::DocumentObject
 - Gui::ViewProvider
 - DataProxy
 - ViewProxy
 - Object creation and binding
- DataProxy Lifecycle
 - All possible states
 - Lifecycle diagram
 - State event listeners

- on attach
- on_create
- on_start
- on_remove
- on restore
- Persistence events listeners
 - on_serialize
 - on_deserialize
- Active event handlers
 - on_execute
 - on_change
 - on_before_change
 - Direct property change listeners
- Other listeners
 - on_extension
- Hooks
 - is_active
 - is_dirty
- Properties
 - Declaring properties
 - Property constructors
 - Property modes
 - Property Editor modes
 - Property change listener
 - Property access
 - Creating properties programmatically
- ViewProxy listeners
 - on_attach
 - on_start
 - on_edit_start
 - on_edit_end
 - on_dbl_click
 - on_context_menu
 - on_delete
 - on_claim_children
 - on_drag_object
 - on_drop_object
 - on_object_change

- ViewProxy hooks
 - can_drag_objects
 - can_drop_objects
 - can_drag_object
 - can_drop_object
 - icon
 - Edit Modes
- Main Decorators
 - @proxy
 - @view proxy
 - Display Modes
 - Function: DisplayMode
- Extensions
 - Available Object Extensions
 - Available View extensions
- Migrations
 - @migrations decorator
 - Function: migrations
- FreeCAD Preferences
 - Preferences API
 - Preference
 - @Preference.subscribe
- Utility functions reference
 - Global functions in fpo module
 - Function: get_selection
 - Function: set_immutable_prop
 - Function: message_box
 - Function: confirm_box
 - Function: print_log
 - Function: print_err
 - Function: get_pd_active_body
 - Function: set_pd_shape
- Compatibility notes
 - Serialization and deserialization
 - Official documentation sources:
- Code examples
- Quick setup

- Usage from a Workbench
- Usage from Macros
- Quick and dirty setup
- Examples setup

Preliminaries

Disclaimer

All of the following information is the result of my own research and usage of the FreeCAD's Python APIs along several years. It reflects my very own view, coding style and limited understanding of FreeCAD internals. All the content is based on official docs, forum discussions, development of my own extensions, reading code of existing extensions and FreeCAD sources.

This document does not cover 100% of the API yet because there are still some obscure methods that can be overridden from the Python Proxies but there is no enough documentation of them, I have never used them or I have not found usage examples. My goal is to cover all of the supported features but it will take time.

Audience

This is a technical document for developers of FreeCAD extensions commonly known as Feature Python Objects or more generally Scripted Objects.

General programming experience, some basic FreeCAD know-how and a minimalistic comprehension of Python are sufficient, as long as you can search the internet for a basic grasp of classes, functions, decorators, type hints, etc...;)

It is also expected that the readers are FreeCAD users, and have a good understanding of the basic usage of it.

Goals

- The API must be developer friendly, consistent, maintainable and compatible with FC 0.21+
- The API must be an overlay on top of the existing API, so it must not conflict with existing code.
- The API must be 100% documented.
- Old code and new code can be mixed, so existing projects can be upgraded gradually if desired.
- Include a tiny documentation generator to produce compact, nice and readable documentation of this API for developers.

Non Goals

- It is not intended to replace anything in the existing FreeCAD APIs.
- It is not intended to require any refactoring of existing Python code.
- It is not intended to require any refactoring of existing C/C++ code.
- The documentation generator script is not for general use.

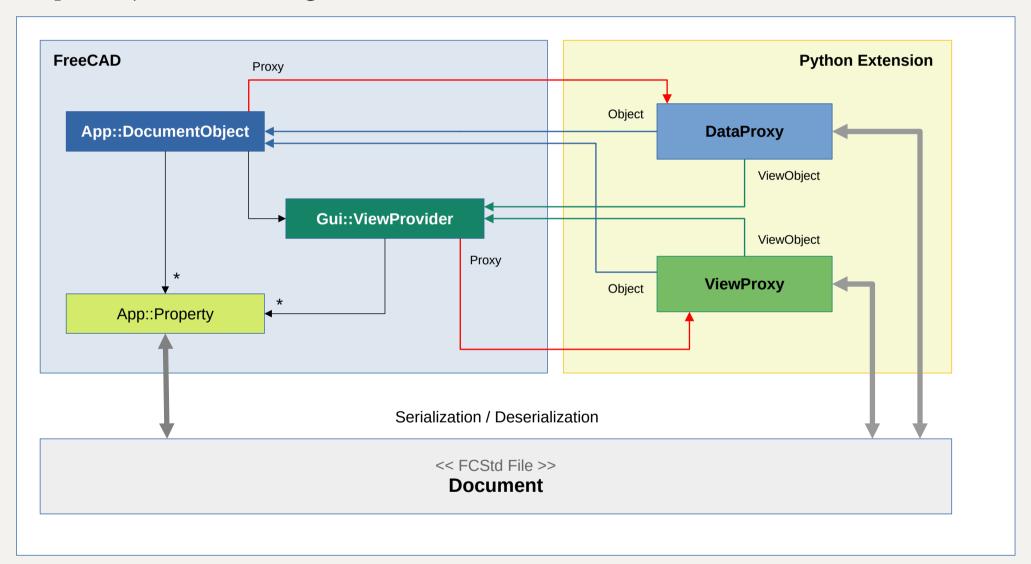
Features



General Scripted Object Architecture

Despite the widespread use of the name *FeaturePythonObject*, this is a concept and not a specific class in the Python API. Maybe a better name should be <code>ScriptedObject</code>. Every <code>ScriptedObject</code> has two main components: The Data component and the View component. Each main component is also divided in two parts: the <code>FreeCAD</code> object and the Python Proxy object. All these 4 pieces conforms the <code>ScriptedObject</code> concept. It is more clear in the following diagram:

Scripted Object Overview diagram



☐ Important

So to develop your own ScriptedObject, you need to create at least one class for the DataProxy and optionally an additional class for the ViewProxy.

(i) Note

View component is optional and only required for GUI part of the object.

App::DocumentObject

Document objects are classes that define the data, geometry and logic of the features in the document. These classes are internal FreeCAD C++ classes and are instantiated from Python using document.addObject(...)

See: https://wiki.freecad.org/Scripted objects

Path::FeatureAreaViewPython

App::FeaturePython and Part::FeaturePython are the most common DocumentObject classes used to create custom objects in FreeCAD, but there are many more types supported:

OBJECT TYPE	DESCRIPTION
App::DocumentObjectGroupPython	
App::FeaturePython	Typical Scripted Object
App::GeometryPython	
App::LinkElementPython	
App::LinkGroupPython	
App::LinkPython	
App::MaterialObjectPython	
App::PlacementPython	
Fem::ConstraintPython	
Fem::FeaturePython	
Fem::FemAnalysisPython	
Fem::FemMeshObjectPython	
Fem::FemResultObjectPython	
Fem::FemSolverObjectPython	
Mesh::FeaturePython	
Part::CustomFeaturePython	
Part::FeaturePython	Typical Scripted object with Shape
Part::Part2DObjectPython	
PartDesign::FeatureAddSubPython	Additive/Subtractive PD Shape
PartDesign::FeatureAdditivePython	Additive PD Shape
PartDesign::FeaturePython	Base PD Feature
PartDesign::FeatureSubtractivePython	Subtractive PD Shape
PartDesign::SubShapeBinderPython	
Path::FeatureAreaPython	

OBJECT TYPE DESCRIPTION

Path::FeatureCompoundPython

Path::FeaturePython

Path::FeatureShapePython

Points::FeaturePython

Sketcher::SketchObjectPython

Spreadsheet::SheetPython

TechDraw::DrawComplexSectionPython

TechDraw::DrawLeaderLinePython

TechDraw::DrawPagePython

TechDraw::DrawRichAnnoPython

TechDraw::DrawTemplatePython

TechDraw::DrawTilePython

TechDraw::DrawTileWeldPython

TechDraw::DrawViewPartPython

TechDraw::DrawViewPython

TechDraw::DrawViewSectionPython

TechDraw::DrawViewSymbolPython

TechDraw::DrawWeldSymbolPython

• Wiki Source: https://wiki.freecad.org/Scripted_objects#Available_object_types

• Forum Source: https://forum.freecad.org/viewtopic.php?t=86414&start=10#p752318

(i) Note

There is an official class diagram, but it does not include scriptable objects apart from App::FeaturePython. See https://wiki.freecad.org/File:FreeCAD_core_objects.svg

Gui::ViewProvider

ViewProviders are classes that define the way objects will look like in the tree view and the 3D view, and how they will interact with certain graphical actions such as selection.

Source: https://wiki.freecad.org/Viewprovider

DataProxy

This is a Python class responsible for managing all the data logic of your ScriptedObject, it creates the data properties and executes the required code on *document recompute*. We will see the details later.

The parametric effect of the ScriptedObject is not achieved by inheritance but rather by "parallel collaboration".

The FreeCAD object has a member, Proxy, that is the instance of the DataProxy class that was created by passing the object itself as argument.

Properties belong to the object and trigger callbacks in the proxy, if any. This is the main way scripted objects work.

There exist also FreeCAD generated events that trigger callbacks in the proxy.

This class is also responsible for serializing/deserializing its own internal state (i.e. not object's properties) from/to the document.

Inside the proxy methods, the object is accessible via self.Object.

To define a DataProxy class, just define a class and decorate it with @proxy decorator.

```
from fpo import proxy, PropertyLength, print_log

@proxy()

class MyCustomObjectProxy:
    length = PropertyLength(default=5)

def on_execute(self):
    print_log("length=", self.length)
    ...

# -- usage

pobj = MyCustomObjectProxy.create(name="MyThing")
```

The name of the class is irrelevant, but using *Proxy* as suffix looks like a good naming convention.

ViewProxy

This is a Python class responsible for managing all the presentation logic of your ScriptedObject, it creates the presentation properties and executes the required code to display the ScriptedObject in the Tree and in the 3D scene. We will see the details later. Analogously to the DataProxy, the ViewProxy is not inherited but rather works in parallel with the ViewObject (accessible via object.ViewObject). Changes in properties of the ViewObject and FreeCAD GUI events trigger callbacks in the ViewProxy.

Moreover, there is a callback (on_object_change) for changes in the Object itself.

This class is also responsible for serializing/deserializing is own internal state from/to the document.

Inside the proxy methods, the object and the view provider are accessible via self.Object and self.ViewObject, respectively.

To define a ViewProxy class just define a class and annotate it with @view proxy decorator.

```
from fpo import view_proxy, DisplayMode

@view_proxy(icon='self:my-icon.svg')

class MyCustomObjectViewProxy:

wireframe = DisplayMode(name='Wireframe')

shaded = DisplayMode(name='Shaded', is_default=True)
```

The name of the class is irrelevant, but using *ViewProxy* as suffix looks like a good naming convention.

To bind the Proxy and the ViewProxy together, you specify the ViewProxy as an argument of the @proxy decorator.

```
from fpo import proxy, view_proxy

wiew_proxy()

class MyCustomViewProxy:

...

proxy(view_proxy=MyCustomViewProxy) # <-- associate DataProxy with ViewProxy

class MyCustomObjectProxy:

...

# ----

by MyCustomObjectProxy.create(name="MyThing")</pre>
```

Object creation and binding

Once the classes are defined, you can create your objects using the create static method.

```
1 def create(name: str = None, label: str = None, doc: Document = None)
```

The create method takes care of adding the DocumentObject to the *Document* and binding the proxies, view providers, etc...

ARGUMENT	TYPE	DESCRIPTION
name	str	Internal name of the object
label	str	Label of the object used in UI.
doc	Document	Document, if omitted, current document will be used, if there is not current document, a new one will be created.

Example:

1 obj = MyCustomObjectProxy.create(name="MyThing")

DataProxy Lifecycle

Every DataProxy object has a lifecycle. You can observe state changes using the appropriate event listeners to add your custom logic.

All possible states

STATE	DESCRIPTION
NonExistent	(virtual) the object does not exists
Creating	(hidden) proxy instance exists but it is not initialized
Created	Proxy is created, properties and extensions are initialized
Active	Everything is initialized and the object is in consistent state. Objects and Proxies are bound together
Serialized	(virtual) the object is passivated in the FCStd file.
Restoring	(hidden) restoring everything from FCStd file.
Restored	Object is fully restored and migrations are applied if any
Removed	(virtual) object was removed from the document
Attaching	(virtual) FreeCAD is creating and binding the objects
Migrating	(virtual) Migration code is running

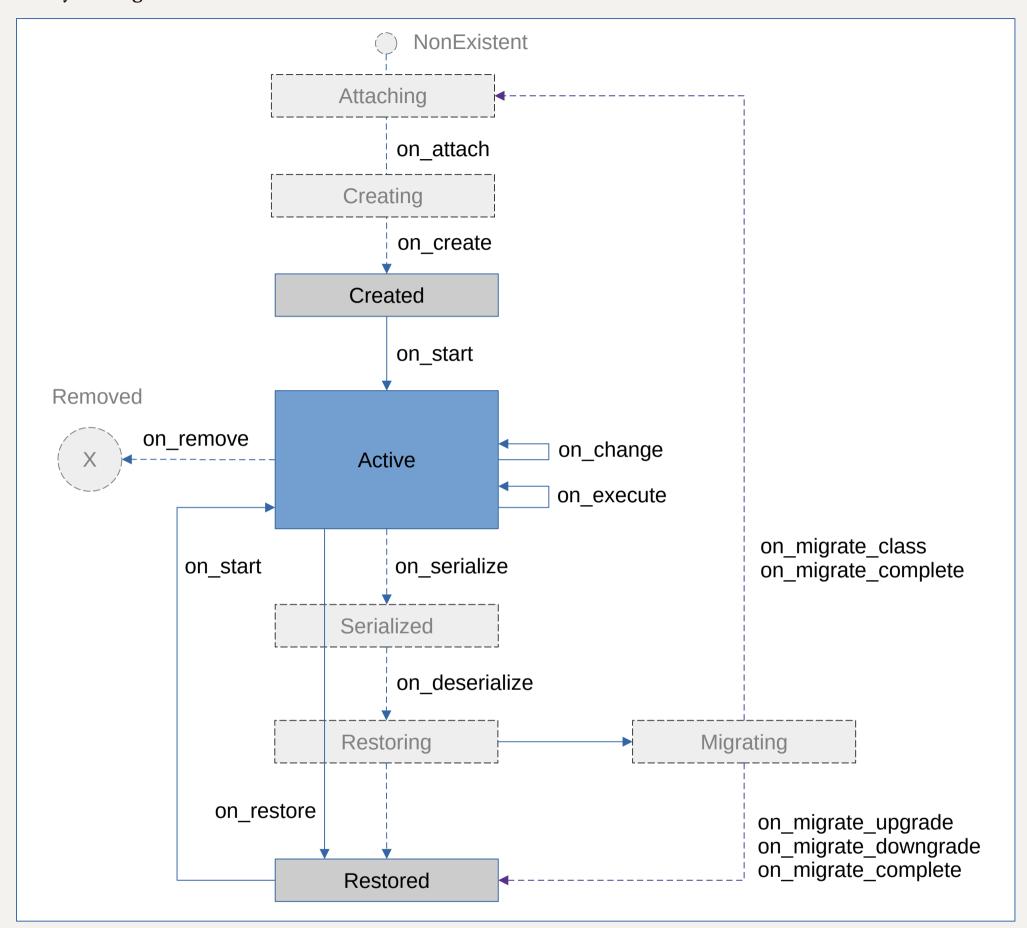
i Note

Virtual states are pure conceptual, they are not present in the code but helps to understand the lifecycle.

Hidden states are not observable (there are no event handlers for them)

The following diagram shows the complete lifecycle:

Lifecycle diagram



State event listeners

To listen to a specific state change event, you create an event handler for that specific event. That is a simple method in your class with the correct signature. All listeners are of course optional.

Using state change listeners you can inject any custom logic in the right place.

Example:

```
from fpo import proxy
3 @proxy()
   class MyCustomObjectProxy:
5
       def on_create(self, event: fpo.events.CreateEvent) -> None:
6
            print(f"{event.source.Label} ({event.source.Name}) was created")
8
       def on_attach(self, event: fpo.events.AttachEvent) -> None:
9
            print(f"{event.source.Label} was attached to the document")
10
11
12
       def on_start(self, event: fpo.events.StartEvent) -> None:
            print(f"{event.source.Label} is ready")
13
14
15
       def on_remove(self, event: fpo.events.RemoveEvent) -> None:
            print(f"{event.source.Name} was removed")
16
17
18
       def on_restore(self, event: fpo.events.DocumentRestoredEvent) -> None:
            print(f"{event.source.Name} ({event.source.Name}) was loaded from the file")
19
20
21
```

on_attach

```
def on_attach(self: Proxy) -> None
def on_attach(self: Proxy, event: fpo.events.AttachEvent) -> None
```

Called when the proxy is just bound to the DocumentObject and attached to the Document.

```
class AttachEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject | None = None
```

on_create

```
def on_create(self: Proxy) -> None
def on_create(self: Proxy, event: fpo.events.CreateEvent) -> None
```

Called when the object is created and after all properties are created and all migrations are applied.

```
class CreateEvent:
source: DocumentObject
```

on start

```
def on_start(self: Proxy) -> None
def on_start(self: Proxy, event: fpo.events.StartEvent) -> None
```

Called after the object is created the first time or after restored from the document. Usually any custom initialization logic must be done here.

```
class StartEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject | None = None
```

on_remove

```
def on_remove(self: Proxy) -> None
def on_remove(self: Proxy, event: fpo.events.RemoveEvent) -> None
```

Called before the object is removed from the Document

```
class RemoveEvent:
source: DocumentObject
```

on restore

```
def on_restore(self: Proxy) -> None
def on_restore(self: Proxy, event: fpo.events.DocumentRestoredEvent) -> None
```

Called when the object is restored (read) from the FCStd file

```
1 class DocumentRestoredEvent:
2    source: DocumentObject
```

Persistence events listeners

FreeCAD is responsible for managing the persistence of the DocumentObjects and ViewProviders (that means objects' properties) but your Proxy classes are responsible for persisting/loading its own internal state from/to the document.

on serialize

```
def on_serialize(self: Proxy) -> None
def on_serialize(self: Proxy, event: fpo.events.SerializeEvent) -> None
```

This method is called to collect data from your object and store it in the document, all that you have to do is include your state into the state dictionary.

```
class SerializeEvent:
source: DocumentObject
state: dict[str, Any]
view_provider: ViewProviderDocumentObject | None = None
```

```
from fpo import proxy
3 @proxy()
4 class MyCustomObjectProxy:
       var1: str
       var2: int
6
       def __init__(self, fp):
8
           self.var1 = 'Hello'
9
           self.var2 = 5
10
11
12
       def on_serialize(self, event: fpo.events.SerializeEvent) -> None:
           event.state['my_value_1'] = self.var1
13
14
           event.state['my_value_1'] = self.var2
15
16
```

on_deserialize

```
1 def on_deserialize(self: Proxy) -> None
2 def on_deserialize(self, event: fpo.events.DeserializeEvent) -> None
```

This method is called to give you data from the document, all that you have to do is read the values from the state dict.

```
class DeserializeEvent:
source: DocumentObject
state: dict[str, Any]
view_provider: ViewProviderDocumentObject | None = None
```

```
from fpo import proxy

@proxy()

class MyCustomObjectProxy:

var1: str

var2: int

def on_deserialize(self, event: fpo.events.DeserializeEvent) -> None:

self.var1 = events.state.get('my_value_1', '')

self.var2 = events.state.get('my_value_2', 0)

...
```

Active event handlers

When your ScriptedObject is active, all your work is performed in on_execute and on_change event listeners.

To query the state of the object, you can call the <code>is_active</code> method:

```
1 def is_active(self: Proxy) -> bool
```

You can also override the method to provide your own logic.

on_execute

```
def on_execute(self: Proxy) -> None
def on_execute(self: Proxy, event: events.ExecuteEvent) -> None
```

This method is where you place the main scripting code of your ScriptedObject, this is called on document recompute if the object is marked as dirty (changed).

```
class ExecuteEvent:
source: DocumentObject
```

```
from fpo import proxy, PropertyLength
import Part

proxy(object_type='Part::FeaturePython')

class CustomBoxProxy:
    width = PropertyLength(default=5.0, description='Width of the box')
    length = PropertyLength(default=5.0, description='Length of the box')
    height = PropertyLength(default=5.0, description='Height of the box')

def on_execute(self):
```

```
# Your magic happens here
self.Object.Shape = Part.makeBox(self.length, self.width, self.height)

# Your magic happens here
self.Object.Shape = Part.makeBox(self.length, self.width, self.height)

# **The self.Object.Shape = Part.makeBox(self.length, self.width, sel
```

on_change

```
1 def on_change(self: Proxy, event: events.PropertyChangedEvent) -> None
```

Called after any property has changed.

Note that you would normally not implement this override but rather use listeners described below.

```
class PropertyChangedEvent(Generic[PT]):
    source: DocumentObject
    property_name: str
    old_value: events.PT | None
    new_value: events.PT | None
    view_provider: ViewProviderDocumentObject | None = None
```

on_before_change

```
def on_before_change(self: Proxy) -> None
def on_before_change(self: Proxy, event: events.PropertyWillChangeEvent) -> None
```

Called before a property change is performed.

```
class PropertyWillChangeEvent(Generic[PT]):
    source: DocumentObject
    property_name: str
    value: events.PT | None
    view_provider: ViewProviderDocumentObject | None = None
```

Direct property change listeners

You can listen to changes on specific properties using the <code>@{prop}.observer</code> decorator:

```
@proxy(object_type='Part::FeaturePython')
class CustomBoxProxy:
   width = PropertyFloat(default=5.0, description='Width of the box')
   length = PropertyFloat(default=5.0, description='Length of the box')
   height = PropertyFloat(default=5.0, description='Height of the box')
```

```
# Your magic happens here

def on_execute(self):
    self.Object.Shape = Part.makeBox(self.length, self.width, self.height)

@length.observer

def length_changed(self, event: events.PropertyChangedEvent) -> None:
    print(f"Hey! length has changed from {event.old_value} to {event.new_value}")
```

They are compatible with the on_change event handler, so you can use both. on_change() is called before or after calling the property listener.

Other listeners

on_extension

```
1 def on_extension(self: Proxy, event: events.ExtensionEvent) -> None
```

Called when an extension is added to the object. Extensions add predefined behaviors to the <code>DocumentObject</code>, making it behave like a group, link, attachable, etc... see: extensions.

```
class ExtensionEvent:
source: DocumentObject
name: str
```

Hooks

There are other optional methods called by FreeCAD to get some info from the Proxy.

is_active

```
1 def is_active(self) -> bool
```

Return True if your DataProxy in the "ready" state.

is_dirty

```
1 def is_dirty(self) -> bool
```

Return True if your DataProxy in a state that requires *recompute*

Properties

The main interaction between your ScriptedObject and the user is by managing property values, the user sets the property values and you do something useful with that.

Properties are declared using special property constructors, there is one constructor per property type.

For a reference of all property types, check the official docs:

• https://wiki.freecad.org/FeaturePython Custom Properties

Declaring properties

Each property can be declared with a proxy attribute.

For example, to create an Integer property:

```
1 @proxy()
2 class MyMagicProxy:
3    my_property = PropertyInteger(section="Basic", default=5)
4    # Optional listener
6    @my_property.observer
7    def my_property_obs(self, event: events.PropertyChangedEvent):
8         print(f"MyProperty has changed from {event.old_value} to {event.new_value}")
9
```

Property constructors

```
def Property{__property_type__}}(
2
            name: str = None,
3
            section: str = 'Data',
            default: Any = None,
4
            description: str = '',
            mode: PropertyMode = PropertyMode.Default,
6
7
            observer_func: Callable = None,
8
            link_property: str = None,
            enum: Enum = None,
9
            options: Callable[[], List[str]] = None)
10
```

ARGUMENT	DESCRIPTION
name	Name of the property, deduced from the attribute if missing
section	Subsection in the property editor
default	Default value of the property
description	Tooltip text
mode	A combination of PropertyMode flags
enum	Only valid for PropertyEnumeration. The enum type.
options	Only valid for PropertyOptions. A function that returns the list of options
link_property	Key of the Link property (see extensions) App::LinkExtensionPython
observer_func	Function to listen for property changes. You can also use the observer decorator. Signature function(event: events.PropertyChangedEvent)

Examples

```
from fpo import PropertyInteger, PropertyLength, PropertyAngle, proxy

@proxy()

class MyProxy:
    x = PropertyInteger(default=5)
    y = PropertyLength(default=0)
    w = PropertyAngle(default=30)
```

Property modes

On property declaration, you can specify a mode or a combination of them. Supported modes are the following:

MODE	DESCRIPTION
PropertyMode.Default	No special property type
PropertyMode.ReadOnly	Property is read-only in the editor
PropertyMode.Transient	Property won't be saved to file
PropertyMode.Hidden	Property won't appear in the editor
PropertyMode.Output	Modified property doesn't touch its parent container
PropertyMode.NoRecompute	Modified property doesn't touch its container for recompute
PropertyMode.NoPersist	Property won't be saved to file at all

Property Editor modes

Editor modes for properties are different than actual property modes and are transient:

MODE	DESCRIPTION
PropertyEditorMode.Default	read/write access in the editor
PropertyEditorMode.ReadOnly	Property is read-only in the editor
PropertyEditorMode.Hidden	Property won't appear in the editor

Property change listener

A function can be subscribed to the property to listen for change events. The argument of the property listener is optional

```
1 @proxy()
2 class MyMagicProxy
3
       my_prop1 = PropertyInteger(section="Basic", default=5)
       my_prop2 = PropertyInteger(section="Basic", default=5)
4
5
       @my_prop1.observer
6
       def listener1(self, event: events.PropertyChangedEvent):
7
            print(f"MyProperty1 has changed from {event.old_value} to {event.new_value}")
9
10
       @my_prop2.observer
       def listener2(self):
11
            print("MyProperty2 has changed")
12
13
```

Important

Only one observer (listener) method can be attached to each property.

Property access

All properties can be accessed from the Proxy object using the declared property name.

It is internally proxyfied to the actual DocumentObject.

```
1 @proxy()
2 class MyMagicProxy
3    my_property1 = PropertyInteger(section="Basic", default=5)
4    def on_execute(self):
6    # Transparently access the property from the remote object
```

```
8
            # The returned value is a float when the property is a quantity
9
            # (length, angle, etc...).
            x = self.my_property1
10
11
12
            # You can also access the property from the object.
            # Remember that the name is automatically camel-cased if not specified
13
14
            # in the constructor.
            # This time, xx is an `App.Units.Quantity` where applicable.
15
            xx = self.Object.MyProperty1
16
17
            # Transparently update the property from the remote object
18
19
            self.my_property1 = 10
20
```

(i) Note

Properties are only proxies of the actual properties in the internal FreeCAD object (DocumentObject).

So persistence is managed by FreeCAD.

Additional state of your proxy object must be serialized/deserialized by you in on_serialize / on_deserialize listeners.

Creating properties programmatically

It is also possible to create properties programmatically using the original FreeCAD API, but in that case you manage them directly from the object.

```
1 @proxy()
   class MyMagicProxy
3
4
       def on_start(self) -> None:
5
            if not hasattr(self.Object, 'Length'):
                self.Object.addProperty(
6
                    "App::PropertyLength",
7
                    "Length", "Box", "Length of the box").Length = 1.0
8
9
       def on_execute(self, event: events.ExecuteEvent) -> None:
10
            # read (a Quantity here)
11
           x = self.Object.Length
12
            # write
13
            self.Object.Length = 10
14
```

15

ViewProxy listeners

ViewProxy has a lightly lifecycle compared to DataProxy but has a lot of listeners and methods to interface with FreeCAD GUI.

on_attach

```
def on_attach(self) -> None
def on_attach(self, event: events.AttachEvent) -> None
```

Called when the ViewObject is attached to the Document.

Usually the initialization logic is here.

on_start

```
def on_start(self) -> None
def on_start(self, event: events.StartEvent) -> None
```

Called when the ViewObject is attached to the Document, all declared properties are created and all declared Display Modes are created.

on_edit_start

```
1 def on_edit_start(self, event: events.EditStartEvent) -> bool | None
```

Called when the user requests edit.

See edit modes

```
class EditStartEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
mode: int
```

on_edit_end

```
1 def on_edit_end(self, event: events.EditEndEvent) -> bool | None
```

Called when the user terminates editing. See edit modes

```
class EditEndEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
mode: int
```

on_dbl_click

```
1 def on_dbl_click(self, event: events.DoubleClickEvent) -> bool
```

Called when the user double clicks the Tree Node. Return True to tell the core system that you handled the action already.

```
class DoubleClickEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
```

on_context_menu

```
1 def on_context_menu(self, event: events.ContextMenuEvent) -> None
```

Called to populate the context menu. You can add actions to the menu object:

```
class ContextMenuEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
menu: QtGui.QMenu
```

```
def on_context_menu(self, event: events.ContextMenuEvent) -> None:
    event.menu.addAction(...)
```

on_delete

```
1 def on_delete(self, event: events.DeleteEvent) -> bool
```

Called when the ViewObject is deleted.

Usually to re-expose the child nodes.

```
class DeleteEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
sub_elements: Any
```

on_claim_children

```
1 def on_claim_children(self, event: events.ClaimChildrenEvent) -> list[DocumentObject]
```

Returns a list of Document Objects that need to be shown as child nodes of this ScriptedObject.

```
class ClaimChildrenEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
```

on_drag_object

```
1 def on_drag_object(self, event: events.DragAndDropEvent) -> None
```

Called if the obj was allowed to be dragged. You perform the drag logic here.

```
class DragAndDropEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
dragged_object: DocumentObject
```

on_drop_object

```
1 def on_drop_object(self, event: events.DragAndDropEvent) -> None
```

Called if the dropped obj was accepted. You perform the drop logic here.

on_object_change

```
1 def on_object_change(self, event: events.DataChangedEvent) -> None
```

Called when a property changes on the associated DocumentObject (Not the ViewObject).

```
class DataChangedEvent:
source: DocumentObject
view_provider: ViewProviderDocumentObject
property_name: str
```

ViewProxy hooks

can_drag_objects

```
1 def can_drag_objects(self) -> bool
```

Returns True if this VP accepts dragging of sub-elements.

can_drop_objects

```
1 def can_drop_objects(self) -> bool
```

Returns True if this VP accepts dropping of sub-elements.

can_drag_object

```
1 def can_drag_object(self, event: events.DragAndDropEvent) -> bool
```

Returns True if this VP accepts dragging of the dragged obj.

can_drop_object

```
1 def can_drop_object(self, event: events.DragAndDropEvent) -> bool
```

Returns True if this VP accepts dropping of the incoming obj.

icon

```
1 def icon(self) -> str | None
```

Returns the path of the icon (Tree Node Icon).

If the returned value is prefixed with 'self:' the path will be resolved relatively to the file where the class is declared.

Edit Modes

MODE	DESCRIPTION
Default(0)	The object will be edited using the mode defined internally to be the most appropriate for the object type
Transform(1)	The object will have its placement editable with the Std TransformManip command
Cutting(2)	This edit mode is implemented as available but currently does not seem to be used by any object
Color(3)	The object will have the color of its individual faces editable with the Part FaceColors command

Main Decorators

There are two entry points for the API, the DataProxy and the ViewProxy. Both of them are created decorating a class with the corresponding decorators @proxy and @view_proxy respectively.

@proxy

```
1 @proxy(
2    object_type: str = 'App::FeaturePython',
3    subtype: str = None,
4    view_proxy: ViewProxy = None,
5    extensions: Iterable[str] = None,
6    view_provider_name_override: str = None,
7    version: int = 1)
```

Converts a user-defined class into a full blown DataProxy with all of the lifecycle management, versioning, proxyfied properties, extensions, etc...

ARGUMENT	DESCRIPTION
object_type *	One of the supported Python feature types. This will be used to create the FC Object using addObject(). by default it is App::FeaturePython
subtype	The handler name of your ScriptedObject, by default it is the name of your class. Saved as Proxy. Type
view_proxy	A reference to the view proxy class
extensions *	A list of extensions to be added to the ScriptedObject
version	Current version of the class. (Used by migrations)
view_provider_name_override	Forced ViewProvider name

@view_proxy

Converts a user-defined class into a full blown ViewProxy with all of the lifecycle management, proxyfied properties, extensions, display mode builders, etc...

```
1 @view_proxy(
2    view_provider_name_override: str = None,
3    extensions: Iterable[str] = None,
4    icon: str = None)
```

ARGUMENT	DESCRIPTION
view_provider_name_override	ViewProvider internal type name, empty by default so FreeCAD will decide the value
icon	Path of the icon for the Tree. If prefixed with 'self:' the path is relative to the file where the class is declared. i.e. 'self:my_icon.svg' will be resolved in the same folder as the file that declares your class.
extensions *	A list of extensions to be added to the VP

Display Modes

Display modes are named zones in the 3D view that have specific presentation attributes.

So, objects placed in each zone are rendered with the zone's attributes.

Display modes are implemented by FreeCAD using coin objects, usually SoGroup or SoSeparator.

Each display mode has a name, an optional method builder that builds the coin object and optionally can be marked as default mode.

Function: DisplayMode

Signature / DisplayMode

```
def DisplayMode(
    name: str | None=None,
    *,
    is_default: bool=False,
    builder: Callable | None=None
    ) -> None: ...
```

Docs / DisplayMode

ARGUMENT	ТҮРЕ	DESCRIPTION
name	str	Name of the display mode
is_default	bool	Configure the DM as default, defaults to False
builder	Callable[[ViewObject], coin.SoGroup]	Method to build the coin object

Declare a display mode.

Declarator of Display Modes, allows to configure a mode and optionally a builder method to create and register the coin object.

Example:

Extensions

Extensions are predefined behaviors that can be added to the DocumentObject or ViewObject to add functionality.

Available Object Extensions

- App::GeoFeatureGroupExtensionPython
- App::GroupExtensionPython
- App::LinkBaseExtensionPython
 - App::LinkExtensionPython (ref)
- App::OriginGroupExtensionPython
- App::SuppressibleExtensionPython
- Part::AttachExtensionPython
- TechDraw::CosmeticExtensionPython

Available View extensions

- Gui::ViewProviderSuppressibleExtensionPython
- Gui::ViewProviderExtensionPython
- Gui::ViewProviderGeoFeatureGroupExtensionPython
- Gui::ViewProviderGroupExtensionPython
- Gui::ViewProviderOriginGroupExtensionPython
- PartGui::ViewProviderAttachExtensionPython
- PartGui::ViewProviderSplineExtensionPython

Migrations

Migrating old versions of your ScriptedObject to maintain backwards compatibility with old files is a complex topic.

You can read all the low level details in this extensive official wiki:

https://wiki.freecad.org/Scripted_objects_migration

In this API, migrations are way more simple as you only have to add the migrations decorator and implement the corresponding methods

Migrations using the same DataProxy Class

```
1
2 @migrations()
3 @proxy(version=2)
4 class FpoClass:
5
        def on_migrate_complete(self, event: events.MigrationEvent) -> None:
6
            # Called after all migrations are applied
 7
8
        def on_migrate_upgrade(self, event: events.MigrationEvent) -> None:
9
            # Called if version is less than current version
10
            # Do any required migration code here
11
12
13
        def on_migrate_downgrade(self, event: events.MigrationEvent) -> None:
            # Called if version is greater than current version
14
15
            # Do any required migration code here
16
        def on_migrate_error(self, event: events.MigrationEvent) -> None:
17
18
            # Called if migration fails
19
20
21
```

```
class MigrationEvent:
source: DocumentObject
from_version: int
to_version: int
```

Migrations using a different DataProxy class

Some times you refactor your code and move the file that declares your DataProxy class, in this scenario FreeCAD fails to find your class as its old module is persisted in the FCStd file.

In this situation you need to do a redirection from the old file to the new one.

Suppose that your old DataProxy was defined as a class named OriginalFpo in a file named original.py and you decided to move it to a file named better.py and renamed your class to BetterFpo.

You need to redirect calls from the old file/class to the new one, and also apply some migration logic to convert the old version into the new version.

In your new file better.py you have your new class, nothing special is required there.

```
# file: better.py

@view_proxy()

class BetterFpoViewProvider:
    # ....

@proxy(view_provider=BetterFpoViewProvider)

class BetterFpo:
    # All the new stuff
```

Now you have to redirect old calls from the old file to the new one. So just create a class with the old name but make it into a migration, the migration will take care of calling your logic and redirecting to the new class after it.

```
# Then rebind to the new class
BetterFpo.rebind(fp) # Reinitialize fp as the new Fpo
# Then rebind to the new class
14
# Reinitialize fp as the new Fpo
15
```

@migrations decorator

Function: migrations

Signature / migrations

```
1 def migrations(current: type | None=None) -> Callable[[type], type]: ...
```

Docs / migrations

ARGUMENT	ТҮРЕ	DESCRIPTION
current	ProxyClass	most recent class, if omitted, the same class is used.

Install migrations management into the class.

Example:

```
1 @migrations()
2 @proxy(version=5)
3 class MyScriptedObjectClass:
4 ...
```

FreeCAD Preferences

You can read and write FreeCAD preferences from your code using a simple API. Use it to save/load configurations.

In FreeCAD, preferences are saved in a Tree of groups/entries like this:

- Every Group is under a root parent, in the example above the root is BaseApp
- Every Entry is under a Group, in the example above the group is MyExtension/My Group
- Every Entry has one value of type int, float, str, bool

To access them for read/write, you just need to create a proxy for the preference:

```
1
2 #----
3 # file: preferences.py
4 # Declare preferences wherever you want,
5 # but usually in some `preferences.py` module
6 # so you can reuse from everywhere.
7 from fpo import Preference
8 config_x = Preference(group="MyExtension/My Group", name="My Param X", default=10)
   config_y = Preference(group="MyExtension/My Group", name="My Param Y", default=10)
10 config_z = Preference(group="MyExtension/My Group", name="My Param Z", default=10)
11
12 #----
13 # file: whatever.py
14 import preferences as pref
15
16 # read values
17 print(f"X = {pref.config_x()}")
18 print(f"Y = {pref.config_y()}")
  print(f"Z = {pref.config_z()}")
19
20
21 # write values
```

```
pref.config_x(150)
pref.config_y(100)
pref.config_z(210)

# subscribe/observe to changes in preferences with listeners
from fpo import Preference

@Preference.subscribe(group="MyExtension/My Group")
def on_preference_change(group, value_type, name, value):
    print(f"Preference changed: {group}, {value_type}, {name}, {value}")
```

Preferences API

Preference

```
1 Preference(group:str, name:str, default:Any=None, value_type:type=str, root:str="BaseApp")
```

ARGUMENT	DESCRIPTION
group	Group path
name	Entry name
default	Default value returned if Entry does not exists
value_type	Type of the value, if not provided, type(default) is used, if no default. str is used
root	Tree root, default is BaseApp

@Preference.subscribe

```
1 @Preference.subscribe(group:str, root="BaseApp")
```

Creates and attach a preference listener, you can observe changes in a group.

ARGUMENT	DESCRIPTION
group	Group path to observe
root	Tree root, default is BaseApp

```
from fpo import Preference

preference.subscribe(group="MyExtension/My Group")

def on_preference_change(group, value_type, name, value):
    print(f"Preference changed: {group}, {value_type}, {name}, {value}")

# You can remove the observer subscription later:
    on_preference_change.unsubscribe()
```

Utility functions reference

There are also few global functions that are frequently used in ScriptedObject development.

So I include them here for quick reference because they are used in the examples.

Global functions in fpo module

Function: get_selection

Signature / get_selection

```
1 def get_selection(*args: tuple) -> tuple: ...
```

Docs / get_selection

RETURN TYPE	DESCRIPTION
List[DocumentObject]	If no arguments are supplied, the list of selected objects
bool, *List[DocumentObject]	If arguments are supplied, the first element returned says if the selection matches the patterns, the rest are the selected objects in order

ARGUMENT	ТҮРЕ	DESCRIPTION
*args	*[str re.Pattern]	List of patterns

Return current selection in specific order and matching specific types.

case 1: no args, just returns selection as list:

```
1 sel = get_selection()
```

case 2: args are the required selection types:

```
1 ok, axis, obj = get_selection('PartDesign::Line'. '*')
2 if ok:
3 ...
```

regex are supported for type matching and '*' is a general wildcard:

```
ok, axis, part, other = get_selection('PartDesign::Line', re.compile('Part::.*'), '*')
if ok:
...
```

this will parse selection for three elements, first one must be a

PartDesign::Line, second one must be any object from the Part namespace,
the last one can be anything. The user can select them
in any order but they will be returned in the specified order. Take
into account that wildcards will match anything so it is better to
specify patterns from more specific to least specific.

In this invocation schema, the first element of the returned tuple is a boolean that indicate if the selection matches the patterns.

Function: set_immutable_prop

Signature / set_immutable_prop

```
1 def set_immutable_prop(obj: ObjectRef, name: str, value: Any) -> None: ...
```

Docs / set_immutable_prop

ARGUMENT	ТҮРЕ	DESCRIPTION
obj	ObjectRef	remote FreeCAD object
name	str	property
value	Any	the value

Force update a property with Immutable status.

Temporarily removes the immutable flag, sets the value and restore the flag if required.

Function: message_box

Signature / message_box

```
1 def message_box(message: str, title: str='Message', details: str='') -> None: ...
```

Docs / message_box

ARGUMENT	ТҮРЕ	DESCRIPTION
message	str	summary
title	str	box title, defaults to "Message"
details	str	expandable text, defaults to None

Show a basic message dialog (modal).

If App.GuiUp is False, prints to the console.

Function: confirm_box

Signature / confirm_box

```
1 def confirm_box(message: str, title: str='Message', details: str='') -> bool: ...
```

Docs / confirm_box

RETURN TYPE	DESCRIPTION
bool	True if user accepts

ARGUMENT	ТҮРЕ	DESCRIPTION
message	str	summary
title	str	box title, defaults to "Message"
details	str	expandable text, defaults to None

Ask for a confirmation with a basic dialog.

Requires App.GuiUp == True

Function: print_log

Signature / print_log

```
1 def print_log(*args: tuple) -> None: ...
```

Docs / print_log

Print into the FreeCAD console with info level.

Function: print_err

Signature / print_err

```
1 def print_err(*args: tuple) -> None: ...
```

Docs / print_err

Print into the FreeCAD console with error level.

Function: get_pd_active_body

Signature / get_pd_active_body

```
1 def get_pd_active_body() -> DocumentObject | tuple[DocumentObject, DocumentObject, str] | None: ...
```

Docs / get_pd_active_body

RETURN TYPE	DESCRIPTION
PartDesign.Body	Active Body

Retrieve the active PartDesign Body if any.

Function: set_pd_shape

Signature / set_pd_shape

```
1 def set_pd_shape(fp: DocumentObject, shape: Shape) -> None: ...
```

Docs / set_pd_shape

Prepare the shape for usage in PartDesign and sets Shape and AddSubShape.

Compatibility notes

Serialization and deserialization

State serialization process used to be managed by methods named __getstate__ / __setstate__ in older versions of FreeCAD but they were renamed to dumps / loads in recent versions due to conflicts with Python 3.11+.

This backwards compatibility issue is transparently managed by this API, but it also was fixed in master recently:

- https://github.com/FreeCAD/FreeCAD/pull/12243
- https://github.com/FreeCAD/FreeCAD/pull/10769

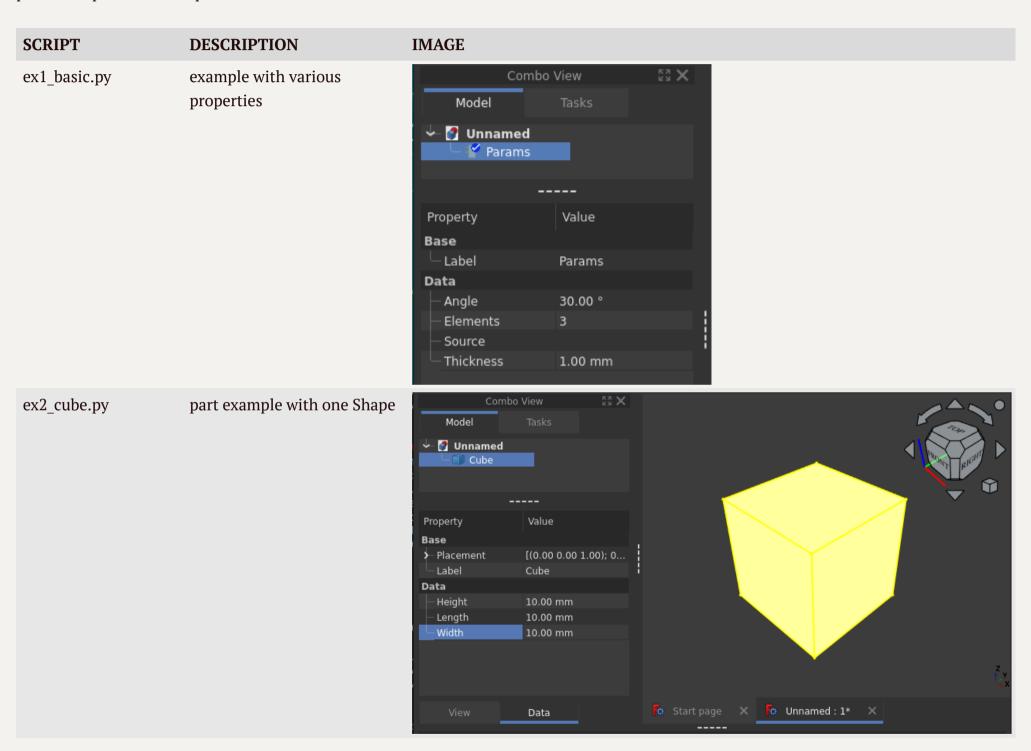
Official documentation sources:

- https://wiki.freecad.org/App FeaturePython
- https://wiki.freecad.org/Viewprovider
- https://wiki.freecad.org/FeaturePython_Custom_Properties
- https://wiki.freecad.org/Create_a_FeaturePython_object_part_I
- https://wiki.freecad.org/Create a FeaturePython object part II
- https://wiki.freecad.org/Scripted objects
- https://wiki.freecad.org/Scripted objects migration
- https://forum.freecad.org/viewforum.php?f=22

• https://wiki.freecad.org/File:FreeCAD core objects.svg

Code examples

There are some basic examples in the examples folder. The examples are numbered in order to imply increasing complexity, I do not repeat comments that were already present in previous examples.



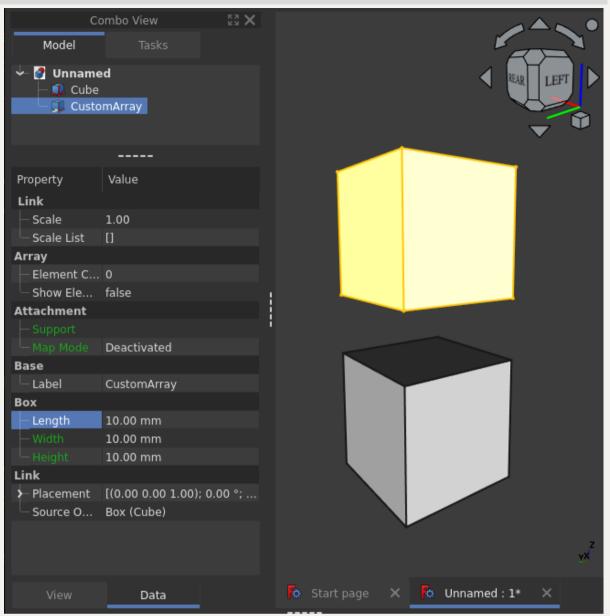
SCRIPT DESCRIPTION IMAGE part example with one Shape ex3_spring.py Model – 📝 Unnamed My Spring Property Value Base > Placement [(0.00 0.00 1.00); 0.00 °; ... Label My Spring Data Bar Diam... 3.00 mm 12.00 mm Height 30.00 mm Pitch 5.00 mm Unnamed: 1* Data part example with one ex4_attachable.py attachable Shape Model – 📝 Unnamed Cyl Cube Property Value Attachment Support Box (Cube) Map Mode ObjectXZ Map Reve... false ➤ Attachme... [(0.00 0.00 1.00); 0.00 °; ... Base ➤ Placement [(1.00 0.00 0.00); 90.00 °... Label Data Height 10.00 mm Radius 10.00 mm

Data

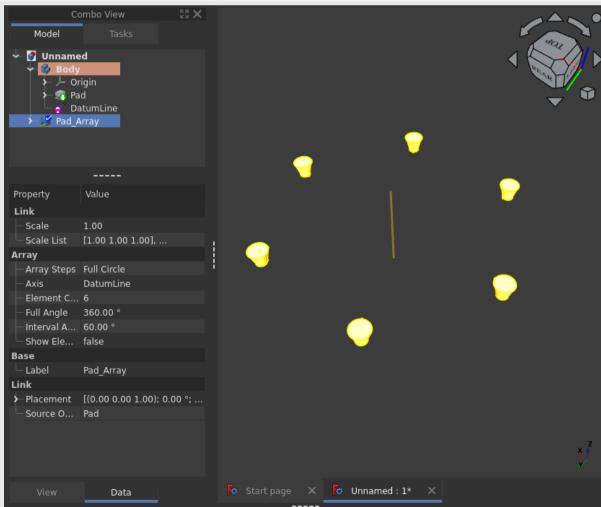
Unnamed: 1*

SCRIPT DESCRIPTION IMAGE

ex5_link.py Object with Link behavior

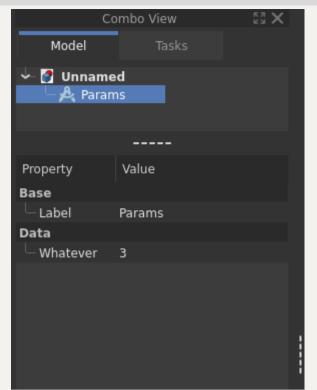


ex6_link_array.py Object with Link array behavior

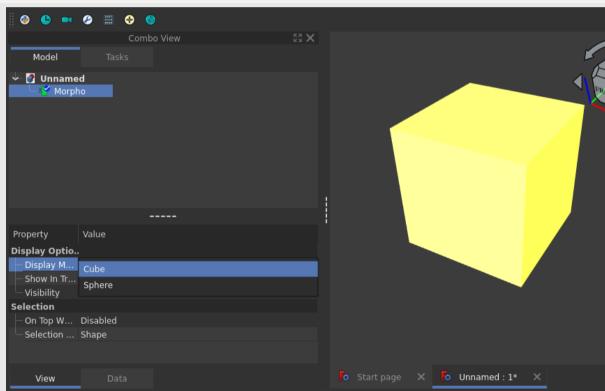


SCRIPT DESCRIPTION IMAGE

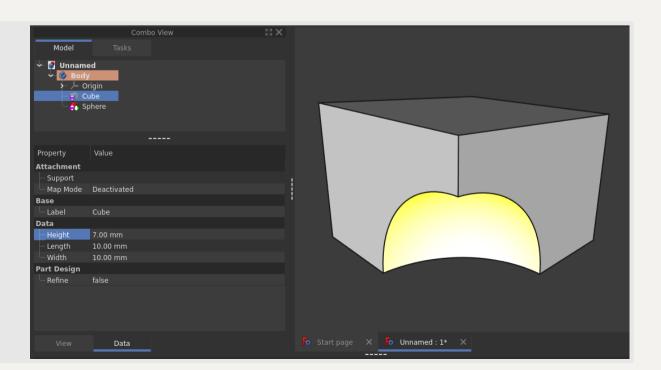
ex7_icon.py
Using a ViewProxy to setup
the icon

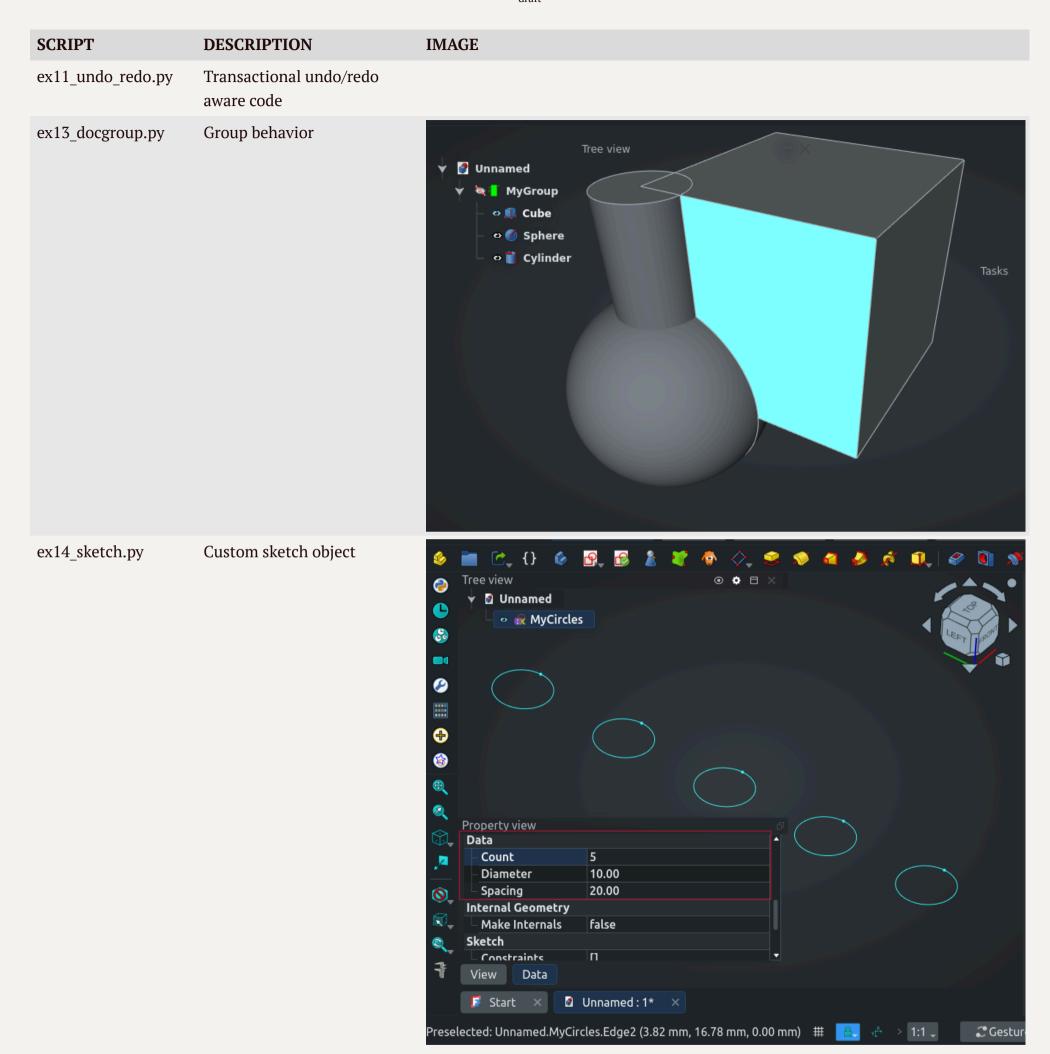


ex8_display_modes.py Using a ViewProxy to setup display modes



ex9_migrations.py Basic migration
ex10_part_design.py Object compatible with
PartDesign





Quick setup

as it is not a FreeCAD core thing by now.

Usage from a Workbench

Put fpo.py in your *Workbench* folder and use it. It is supposed that this API is used for *Workbench* developers so no other special configuration is required for that case.

As fpo.py is not part of the core FreeCAD distribution, it is possible that other *Workbenches* already include the file. So it is a good precaution to rename your copy or put it in your internal module.

Remember that the recommended layout for workbenches is to put the code into a module of the freecad package.

```
1
    └─ FreeCAD Config Dir/
       └─ Mod/
3
            └─ YourWorkbench/
               └─ freecad/
                    └─ your_module/
6
                        ├─ __init__.py
8
                        ├─ init_gui.py
9
                        ├─ fpo.py
                        your_amazing_thing.py
10
11
                        └─ ...
```

Usage from Macros

It is better to not define your proxy classes directly in **Macros** because FreeCAD will have have a hard time finding them when reloading the objects from saved documents.

What you can do in this case is putting the fpo.py file directly in the FreeCAD's Macros directory, then create your proxy classes in its own file in Macros dir, then import them from your macros. That way FreeCAD will find the Proxies next time you open your Documents.

Quick and dirty setup

Another easy way if you don't want to develop a *Workbench* is to fake one, To do that simply create a folder inside FreeCAD's Mod directory and put fpo.py and your other python files there. This will make fpo and your modules visible and importable from FreeCAD.

Examples setup

Copy fpo.py and examples/* (the files, not the directory) into FreeCAD's *Macro* dir. then you can run the examples from the FreeCAD's Python console:

```
import ex3_spring as ex3
ex3.create_spring()

import ex10_part_design as ex10
ex10.create_cube_pd()

...
```

☐ Important

Copy fpo.py in **only one place** to avoid name conflicts.