FreeCAD Scripted Objects Modern API

META	VALUE
generated	2024-08-14 03:07:00.477830
author	Frank David Martínez Muñoz
copyright	(c) 2024 Frank David Martínez Muñoz.
license	LGPL 2.1
version	1.0.0-beta1
min_python	3.8
min_freecad	0.21

TABLE OF CONTENTS

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 no conflicts 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.

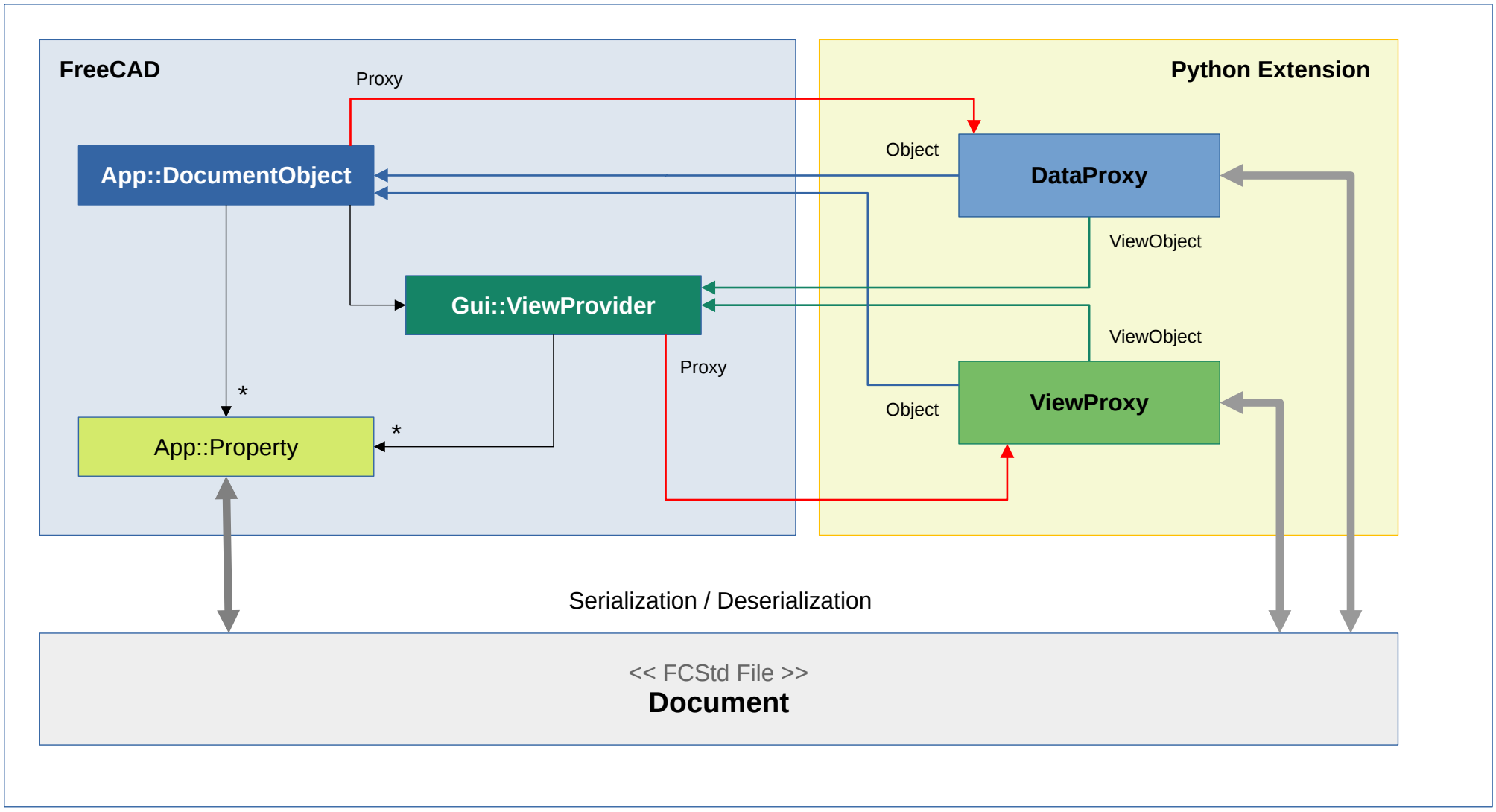
Features

- √ Declarative DataProxy (@proxy)
    - √ Observable explicit and well defined lifecycle
    - √ Serialization/Deserialization
    - √ Automatic Proxy-Object association
    - √ Declarative extensions
    - √ Extension lifecycle management
- √ Properties
    - √ Declarative creation (Property*)
    - √ Proxy based read/write
    - √ Observable
- √ Declarative ViewProxy (@view_proxy)
    - √ Display Modes
        - √ Declarative creation (DisplayMode)
        - √ Builder based creation of display modes
    - √ Drag and Drop support
- √ Migrations (@migrations)
    - √ Declarative support
    - √ Redirect to different class
    - √ Automatic version management
    - √ upgrade/downgrade/redirect
- √ Extensions
    - √ Declarative Extension support
- √ Preferences
    - √ Proxy based read/write
    - √ Declarative listeners (@Preference.subscribe)
    - √ Automatic type handling
- √ Documentation in markdown format.

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 `ScriptedObject`. Every `ScriptedObject` has two main components: The Data component and the View component. Each main component is also divided in two parts: the FreeCAD object and the python Proxy object. All these 4 pieces conforms the `ScriptedObject` 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`.

> **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

`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`	
`Path::FeatureAreaViewPython`	
`Path::FeatureCompoundPython`	

OBJECT TYPE	DESCRIPTION
`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

> **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.

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

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, obj):
        print_log("length=", self.length)
        ...

# -- usage
obj = 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.

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

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.

```python
from fpo import proxy, view_proxy

@view_proxy()
class MyCustomViewProxy:
    ...

@proxy(view_proxy=MyCustomViewProxy)  # <-- associate DataProxy with ViewProxy
class MyCustomObjectProxy:
    ...

# -----
obj = MyCustomObjectProxy.create(name="MyThing")
```

Object creation and binding

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

```python
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:

```python
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.

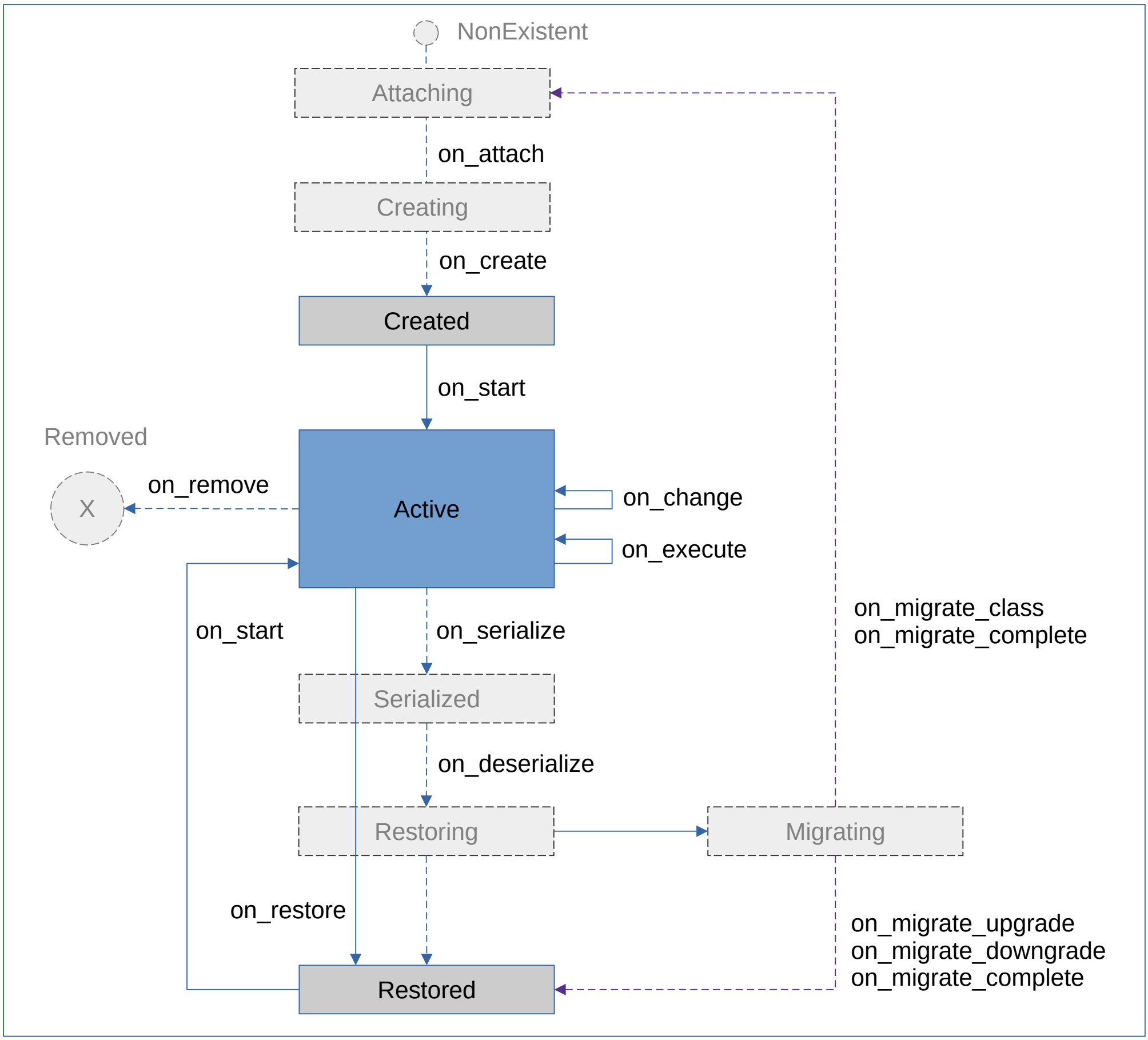
All possible states

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

> **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:

```python
from fpo import proxy

@proxy()
class MyCustomObjectProxy:

    def on_create(self, fpo):
        print("My Object was created")

    def on_attach(self, fpo):
        print("My Object was attached to the document")

    def on_start(self, fpo):
        print("My Object is ready")

    def on_remove(self, fpo):
        print("My Object was removed")

    def on_restore(self, fpo):
        print("My Object was loaded from the file")

    ...
```

on_attach

```python
def on_attach(self: Proxy, fpo: DocumentObject) -> None
```

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

on_create

```python
def on_create(self: Proxy, fpo: DocumentObject) -> None
```

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

on_start

```python
def on_start(self: Proxy, fpo: DocumentObject) -> 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.

on_remove

```
def on_remove(self: Proxy, fpo: DocumentObject) -> None
```

Called before the object is removed from the Document

on_restore

```
def on_restore(self: Proxy, fpo: DocumentObject) -> None
```

Called when the object is restored from the *FCStd* file

Persistence events listeners

FreeCAD is responsible for managing the persistence of the `DocumentObjects` and `ViewProviders` but your `Proxy` classes are responsible for persisting/loading its own internal state from/to the document.

on_serialize

```
def on_serialize(self: Proxy, state: Dict[str, Any]) -> 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.

```
from fpo import proxy

@proxy()
class MyCustomObjectProxy:
    var1: str
    var2: int

    def __init__(self, fp):
        self.var1 = 'Hello'
        self.var2 = 5

    def on_serialize(self, state: Dict[str. Any]):
        state['my_value_1'] = self.var1
        state['my_value_1'] = self.var2

    ...
```

on_deserialize

```
def on_deserialize(self: Proxy, state: Dict[str, Any]) -> 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.

```
from fpo import proxy

@proxy()
class MyCustomObjectProxy:
    var1: str
    var2: int

    def on_deserialize(self, state: Dict[str, Any]):
        self.var1 = state.get('my_value_1', '')
        self.var2 = 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.

on_execute

```
def on_execute(self: Proxy, fpo: DocumentObject) -> 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).

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

    ...

# -----
```

```
obj = CustomBoxProxy.create(name='box1')
```

on_change

```
def on_change(self: Proxy, fpo: DocumentObject,
              prop_name: str, new_value: Any, old_value: Any) -> None
```

Called after any property has changed.

on_before_change

```
def on_before_change(self: Proxy, fpo: DocumentObject,
              prop_name: str, old_value: Any) -> None
```

Called when a property change will be performed.

Direct property change listeners

You can listen to changes on specific properties using the @{prop}.observer 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, fpo):
        fpo.Shape = Part.makeBox(self.length, self.width, self.height)

    @length.observer
    def length_changed(self, fp, new_value, old_value):
        print(f"Hey! length has changed from {old_Value} to {new_value}")

```

Other listeners

on_extension

```
def on_extension(self: Proxy, fpo: DocumentObject, extension: str) -> None
```

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

Hooks

More optional methods called by FreeCAD to get some info from the Proxy

can_link_properties

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

Return true to cause PropertyView to show linked object's property

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:

```
@proxy()
class MyMagicProxy:
    my_property = PropertyInteger(section="Basic", default=5)

    # Optional listener
    @my_property.observer
    def my_property_obs(self, fp, new_value, old_value):
        print(f"my_property has changed from {old_value} to {new_value}")

```

Property constructors

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

ARGUMENT	DESCRIPTION
name	Name of the property, deduced from the attribute if missing

ARGUMENT	DESCRIPTION
section	Sub section 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.

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

MODE	DESCRIPTION
`PropertyEditorMode.ReadOnly`	Property is read-only in the editor
`PropertyEditorMode.Hidden`	Property won't appear in the editor

Property change listener

Any function can be subscribed to the property to listen for change events. The arguments of the property listener are all optional

```
@proxy()
class MyMagicProxy
    my_prop1 = PropertyInteger(section="Basic", default=5)
    my_prop2 = PropertyInteger(section="Basic", default=5)
    my_prop3 = PropertyInteger(section="Basic", default=5)

    @my_prop1.observer
    def listener1(self, fp, new_value, old_value):
        print(f"my_property1 has changed from {old_value} to {new_value}")

    @my_prop2.observer
    def listener2(self, fp, new_value):
        print(f"my_property2 has changed {new_value}")

    @my_prop3.observer
    def listener3(self, fp):
        print(f"my_property3 has changed")

```

> **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`.

```
@proxy()
class MyMagicProxy
    my_property1 = PropertyInteger(section="Basic", default=5)

    def on_execute(self, fp):

        # Transparently access the property from the remote object
        x = self.my_property1

        # Transparently update the property from the remote object
        self.my_property1 = 10

```

> **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 direct API, but in that case you manage them directly from the object.

```
@proxy()
class MyMagicProxy

    def on_start(self, fp):
        if not hasattr(fp, 'Length'):
            fp.addProperty(
                "App::PropertyLength",
                "Length", "Box", "Length of the box").Length = 1.0

    def on_execute(self, fp):
        # read
        x = fp.Length
        # write
        fp.Length = 10

```

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, vp: ViewObject) -> None
```

Called when the `ViewObject` is attached to the Document. Usually init logic is here.

on_start

```
def on_start(self, vp: ViewObject) -> None
```

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

on_edit_start

```
def on_edit_start(self, vp: ViewObject, mode: EditMode = EditMode.Default) -> None
```

Called when the user request edit. See edit modes

on_edit_end

```
def on_edit_end(self, vp: ViewObject, mode: EditMode = EditMode.Default) -> None
```

Called when the user terminates editing. See edit modes

on_dbl_click

```
def on_dbl_click(self, vp: ViewObject) -> bool
```

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

on_context_menu

```
def on_context_menu(self, vp: ViewObject, menu: QMenu) -> None
```

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

```

def on_context_menu(self, vp, menu):
    menu.addAction(...)

```

on_delete

```
def on_delete(self, vp: ViewObject, sub_elements) -> None
```

Called when the ViewObject is deleted. Usually to re-expose the child nodes.

on_claim_children

```
def on_claim_children(self) -> List[DocumentObject]
```

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

on_drag_object

```
def on_drag_object(self, vp: ViewObject, obj: DocumentObject) -> None
```

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

on_drop_object

```
def on_drop_object(self, vp: ViewObject, obj: DocumentObject) -> None
```

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

on_object_change

```
def on_object_change(self, fp: DocumentObject, prop_name: str) -> None
```

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

ViewProxy hooks

---

can_drag_objects

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

Returns True if this VP accepts dragging of sub-elements

can_drop_objects

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

Returns True if this VP accepts dropping of sub-elements

can_drag_object

```
def can_drag_object(self, obj: DocumentObject) -> bool
```

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

can_drop_object

```
def can_drop_object(self, obj: DocumentObject) -> bool
```

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

icon

```
def icon(self) -> str
```

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

```
@proxy(
    object_type: str = 'App::FeaturePython',
    subtype: str = None,
    view_proxy: ViewProxy = None,
    extensions: Iterable[str] = None,
    view_provider_name_override: str = None,
    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...

```
@view_proxy(
    view_provider_name_override: str = None,
    extensions: Iterable[str] = None,
    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

```
def DisplayMode(name: str = None, is_default: bool = False, builder: Callable = None)
```

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

ARGUMENT	TYPE	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 if required

Example:

```
@view_proxy()
class MyViewProxy:
    wireframe_plus = DisplayMode(name="WireframePlus", is_default=True)
    shaded = DisplayMode(name="Shaded")

    @wireframe_plus.builder
    def wireframe_plus_builder(self, vp):
        return SoSeparator()

```

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
6      def on_migrate_complete(self, version, obj):
7          # Called after all migrations are applied
8
9      def on_migrate_upgrade(self, version, fp):
10         # Called if version is less than current version
11         # Do any required migration code here
12
13     def on_migrate_downgrade(self, version, fp):
14         # Called if version is greater than current version
15         # Do any required migration code here
16
17     def on_migrate_error(self, version, fp):
18         # Called if migration fails
19
```

```
    ...

```

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.

```
# file: original.py

from fpo import migrations, proxy
from better import BetterFpo, BetterFpoViewProvider

@migrations(current=BetterFpo)
@proxy()
class OriginalFpo:

    def on_migrate_class(self, version, fp):
        # Perform any migration logic here ....

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

```

@migrations decorator

function: migrations

```
def migrations(current=None)
```

Install migrations management into the class.

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

Example:

```
@migrations()
@proxy(version=5)
class MyScriptedObjectClass:
    ...
```

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:

```
.
└── User parameter:BaseApp/
    └── MyExtension/
        └── My Group/
            ├── My Param X = 0.5
            ├── My Param Y = 10
            └── My Param Z = 100
```

- 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:

```

#------
# file: preferences.py
#  Declare preferences wherever you want,
#  but usually in some `preferences.py` module
#  so you can reuse from everywhere.
from fpo import Preference
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)
config_z = Preference(group="MyExtension/My Group", name="My Param Z", default=10)

#------
# file: whatever.py
import preferences as pref

# read values
print(f"X = {pref.config_x()}")
print(f"Y = {pref.config_y()}")
print(f"Z = {pref.config_z()}")

# 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

```
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

```
@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

```python
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 included them here for quick reference because they are used in the examples.

Global functions in fpo module

function: **get_selection**

```python
def get_selection(*args) -> Tuple
```

Returns current selection in specific order and matching specific types.

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

```python
sel = get_selection()
```

case 2: args are the required selection types:

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

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

```python
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.

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	TYPE	DESCRIPTION
*args	*[str|re.Pattern]	List of patterns

function: **set_immutable_prop**

```
1 def set_immutable_prop(obj: Union[DocumentObject, ViewProviderDocumentObject], name: str, value: Any) ->
  None
```

Force update a property with Immutable status. It temporarily removes the immutable flag, sets the value and restore the flag if required.

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

function: **message_box**

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

Shows a basic message dialog (modal) if `App.GuiUp` is True, else prints to the console.

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

function: **confirm_box**

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

Ask for a confirmation with a basic dialog. Requires App.GuiUp == True

RETURN TYPE	DESCRIPTION
bool	True if user accepts

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

function: **print_log**

```
1  def print_log(*args)
```

Print into the FreeCAD console with info level.

function: **print_err**

```
1  def print_err(*args)
```

Print into the FreeCAD console with error level.

function: **get_pd_active_body**

```
1  def get_pd_active_body() -> PartDesign_Body
```

Retrieve the active PartDesign Body if any

RETURN TYPE	DESCRIPTION
PartDesign_Body	Active Body

function: **set_pd_shape**

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

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:

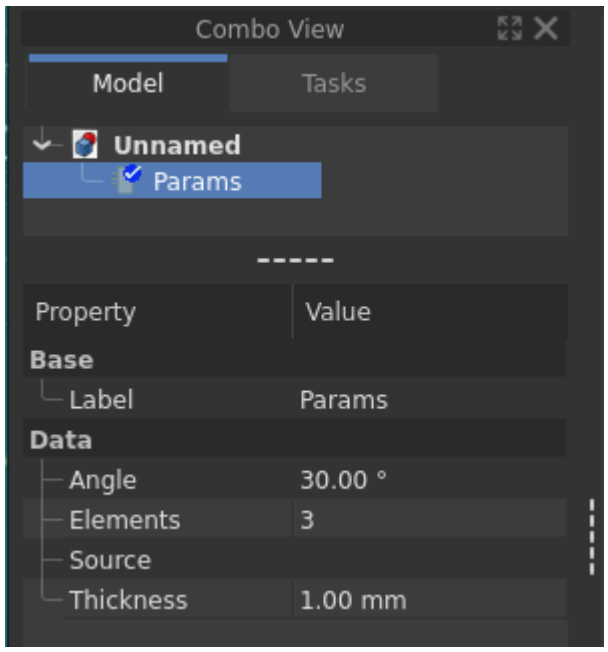
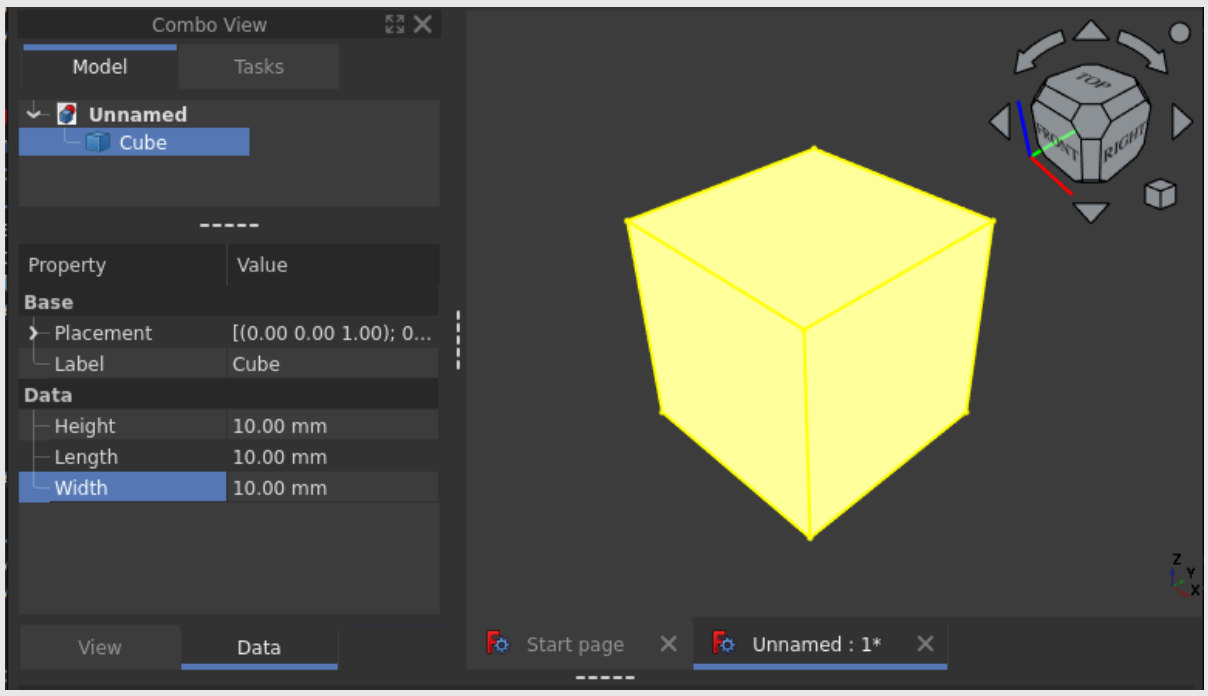
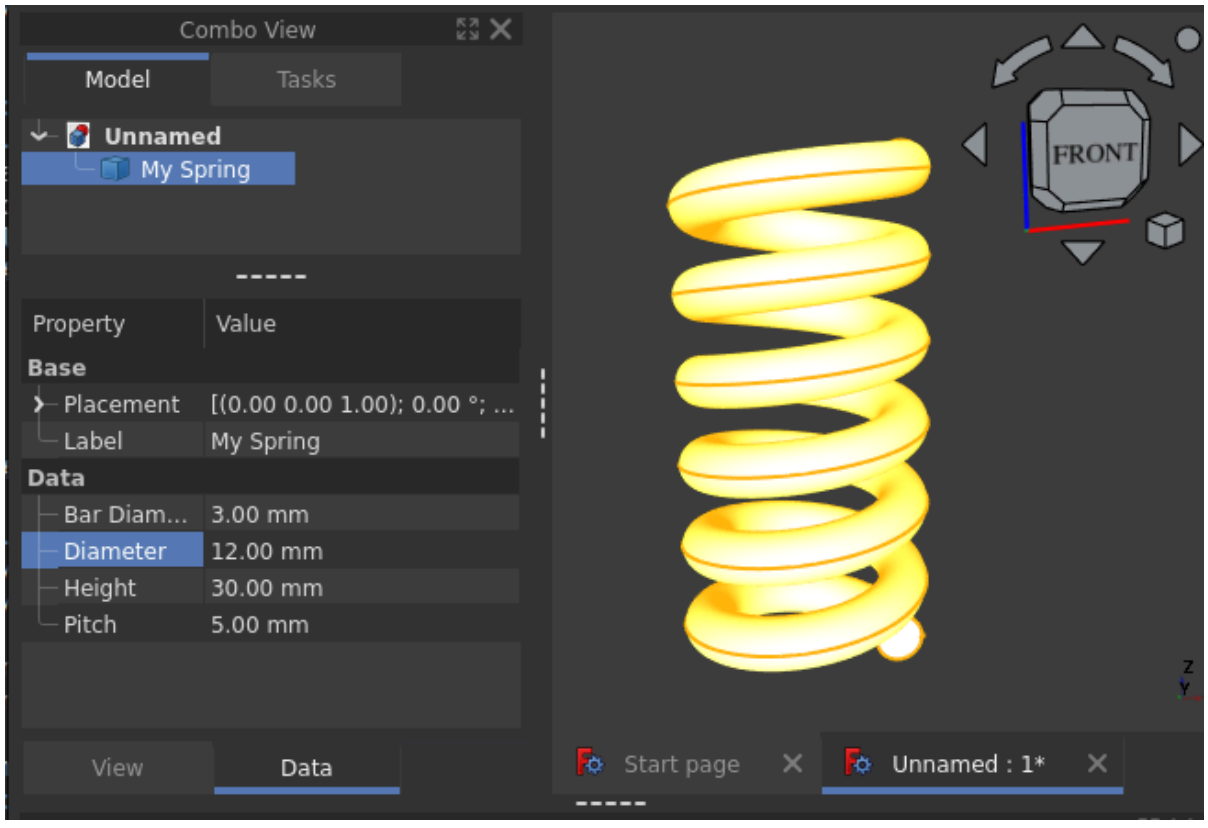
- https://github.com/FreeCAD/FreeCAD/pull/12243
- https://github.com/FreeCAD/FreeCAD/pull/10769

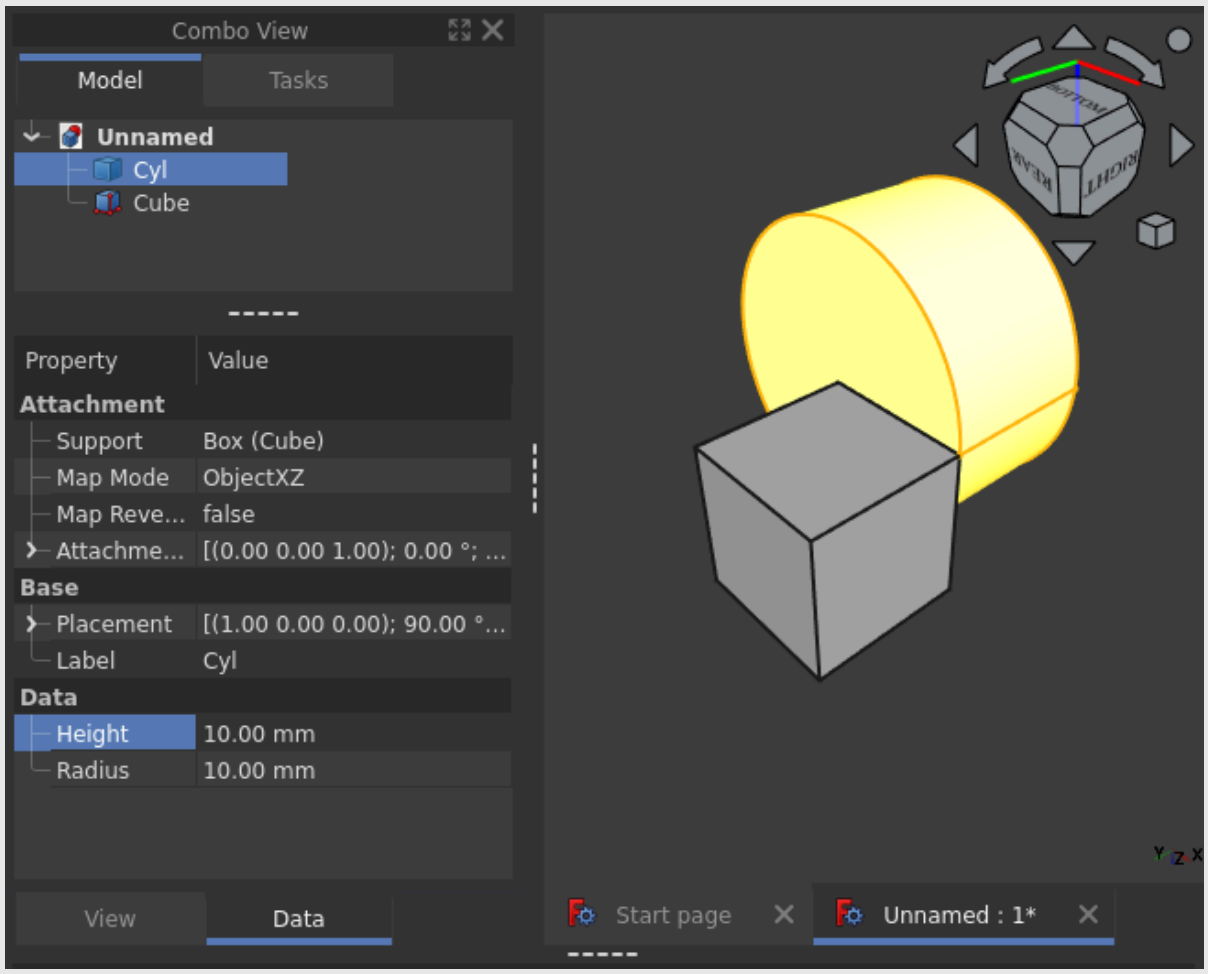
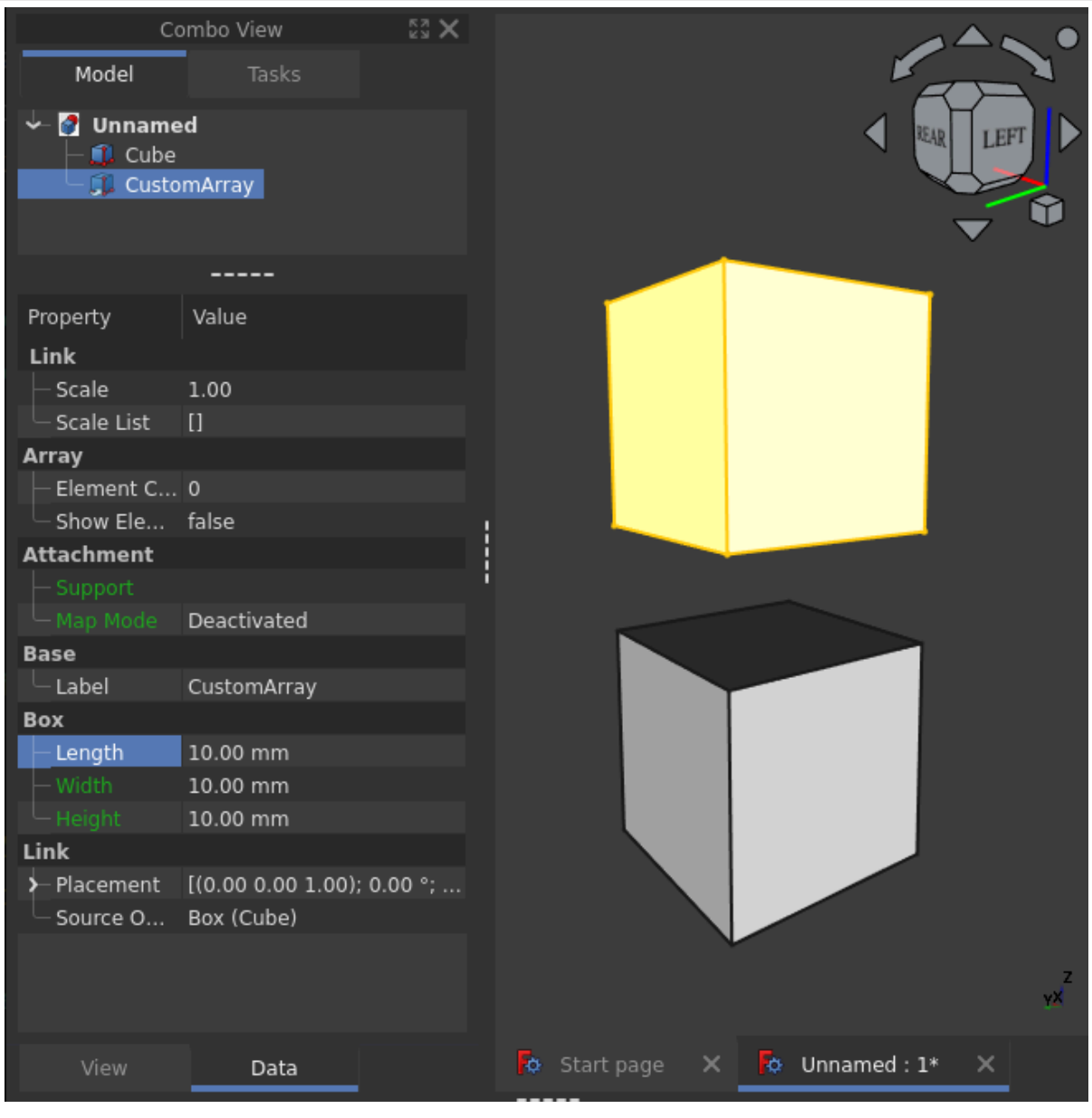
Official documentation sources:

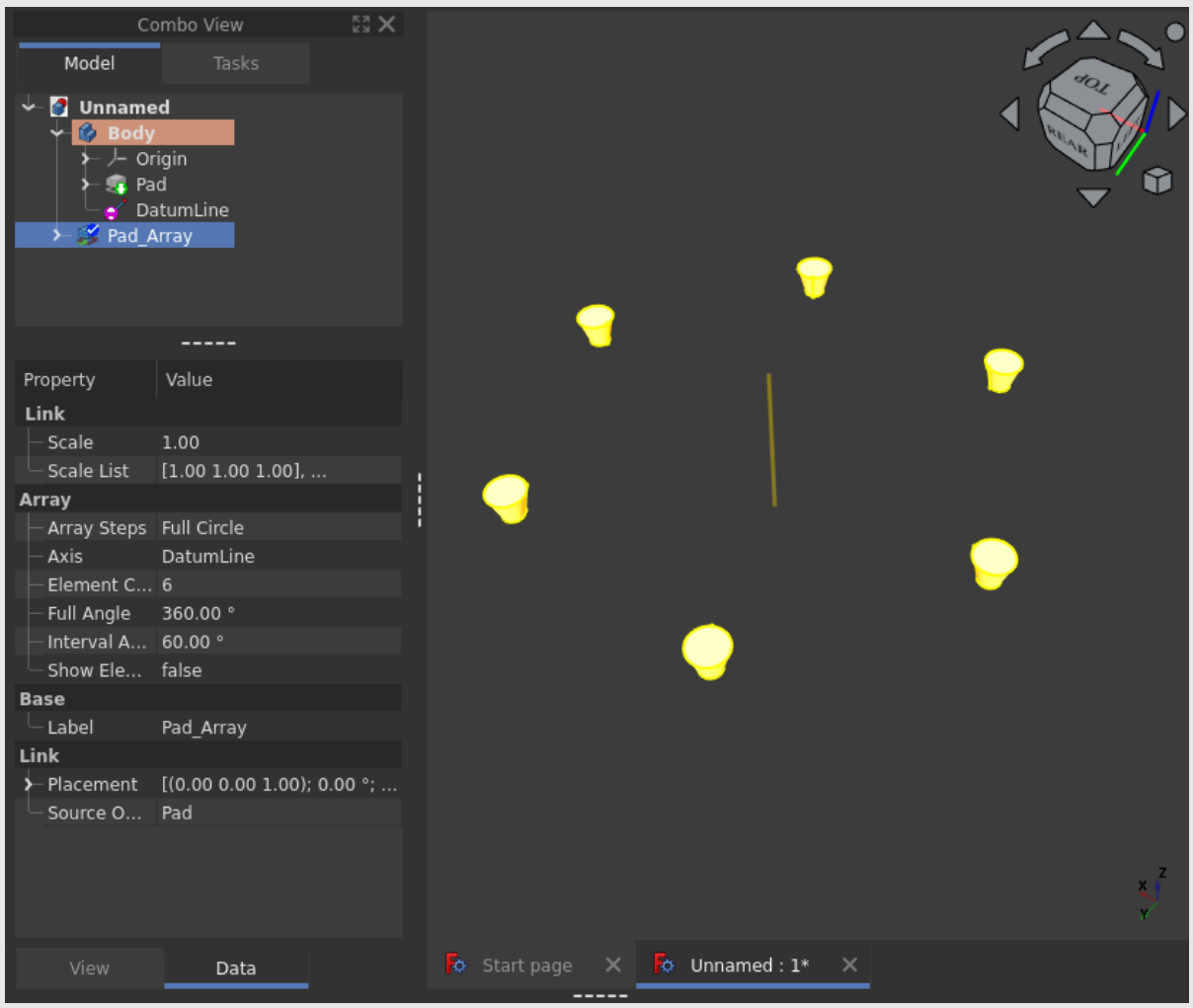
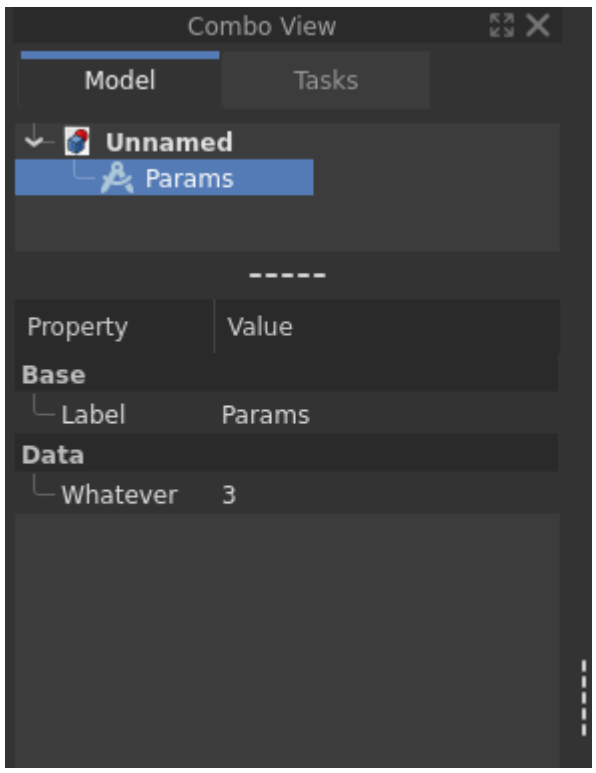
- https://wiki.freecad.org/App_FeaturePython
- https://wiki.freecad.org/Viewprovider
- https://wiki.freecad.org/FeaturePython_Custom_Properties
- https://wiki.freecad.org/Create_a_FeaturePython_object_part_I
- https://wiki.freecad.org/Create_a_FeaturePython_object_part_II
- https://wiki.freecad.org/Scripted_objects
- https://wiki.freecad.org/Scripted_objects_migration
- https://forum.freecad.org/viewforum.php?f=22
- https://wiki.freecad.org/File:FreeCAD_core_objects.svg

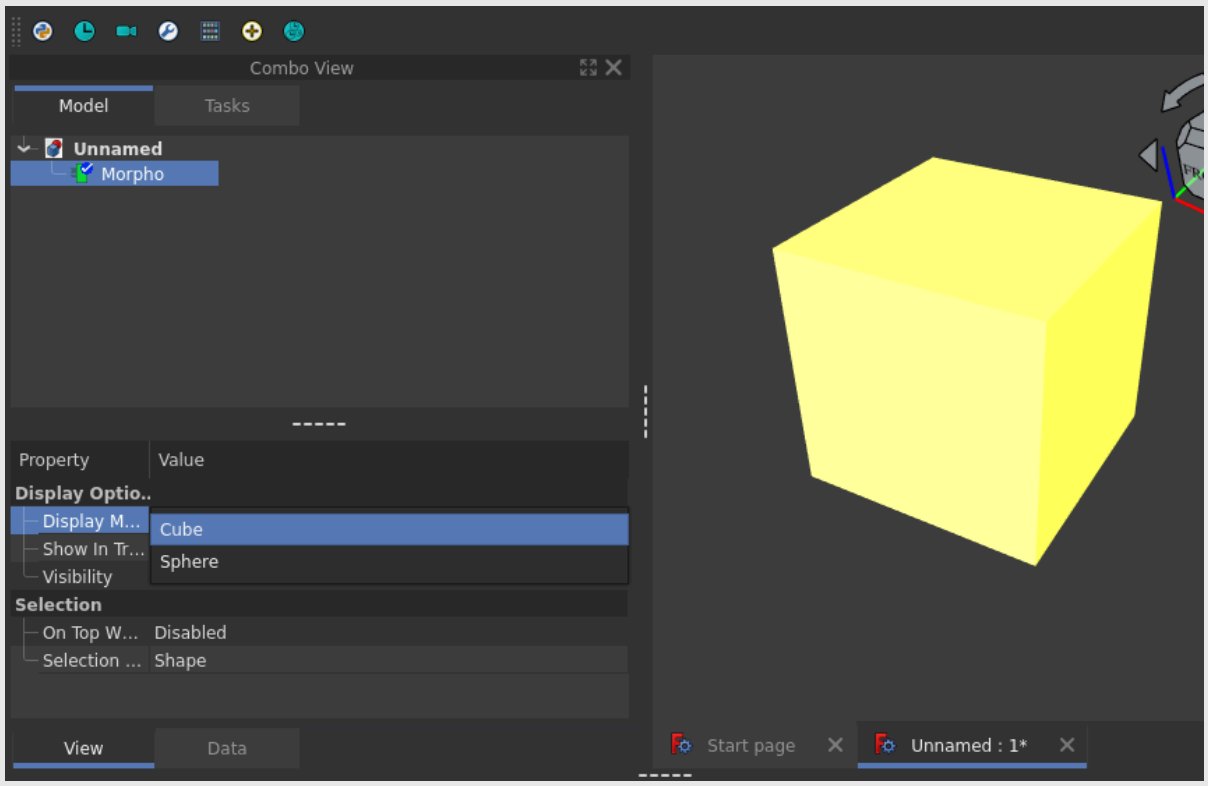
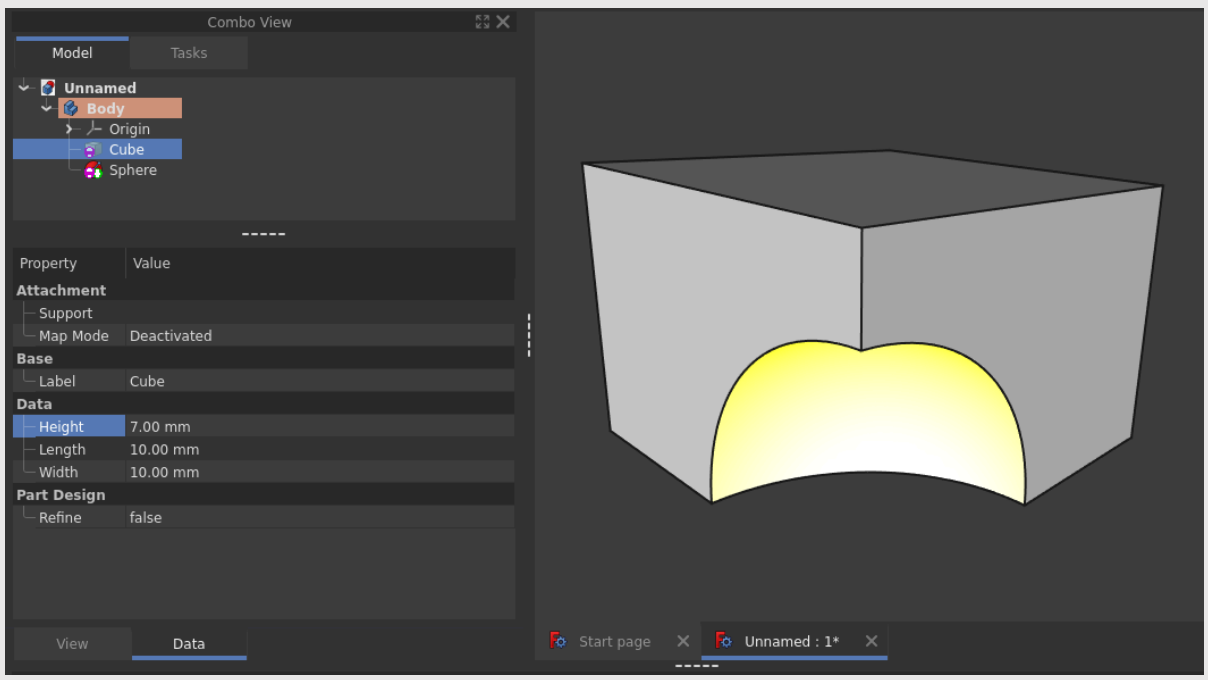
Code examples

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
ex1_basic.py	example with one property	
ex2_cube.py	part example with one Shape	
ex3_spring.py	part example with one Shape	

SCRIPT	DESCRIPTION	IMAGE
ex4_attachable.py	part example with one attachable Shape	
ex5_link.py	Object with Link behavior	

SCRIPT	DESCRIPTION	IMAGE
ex6_link_array.py	Object with Link array behavior	
ex7_icon.py	Using a ViewProxy to setup the icon	

SCRIPT	DESCRIPTION	IMAGE
ex8_display_modes.py	Using a ViewProxy to setup display modes	
ex9_migrations.py	Basic migration	
ex10_part_design.py	Object compatible with PartDesign	
ex11_undo_redo.py	Transactional undo/redo aware code	

Quick setup

The only required file to use this API is `fpo.py`, the key point is where to put it 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.

```
.
└── FreeCAD Config Dir/
    └── Mod/
        └── YourWorkbench/
            └── freecad/
                └── your_module/
                    ├── __init__.py
                    ├── init_gui.py
                    ├── fpo.py
                    ├── your_amazing_thing.py
                    └── ...
```

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 to 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 a name conflicts.