

Tutorial

**on 3D Slicer python scripting
and programming**

Agenda

Part 1

- Software architecture

Part 2

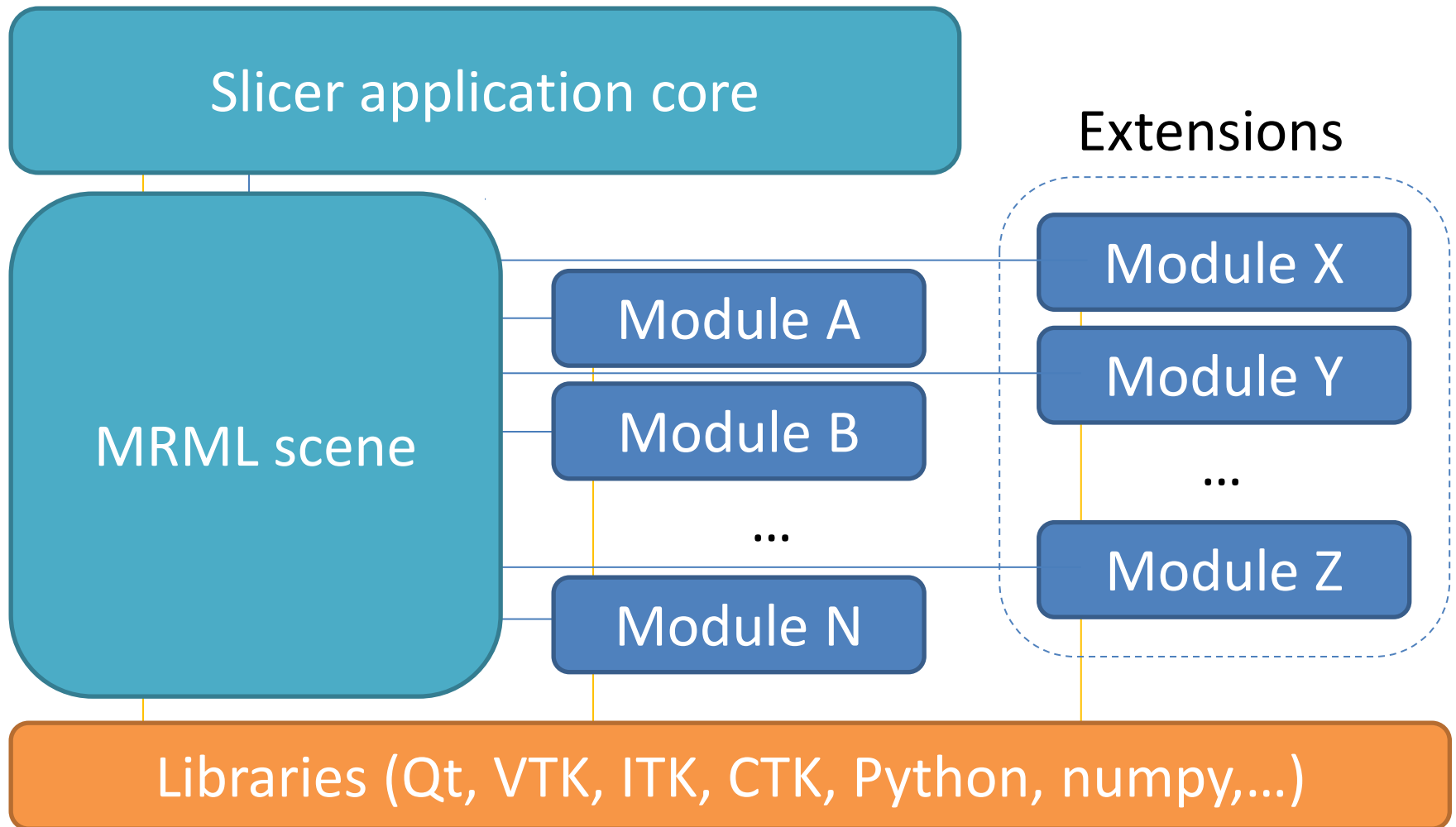
- Use python console in Slicer
- Simple scripted module example

Part 3

- Write simple scripted module individually



Slicer application architecture



Module type: C++ loadable

- Written in C++
- Full Slicer API is accessible
- Useful for implementing complex, performance-critical, interactive components, application infrastructure (e.g., reusable low-level GUI widgets)

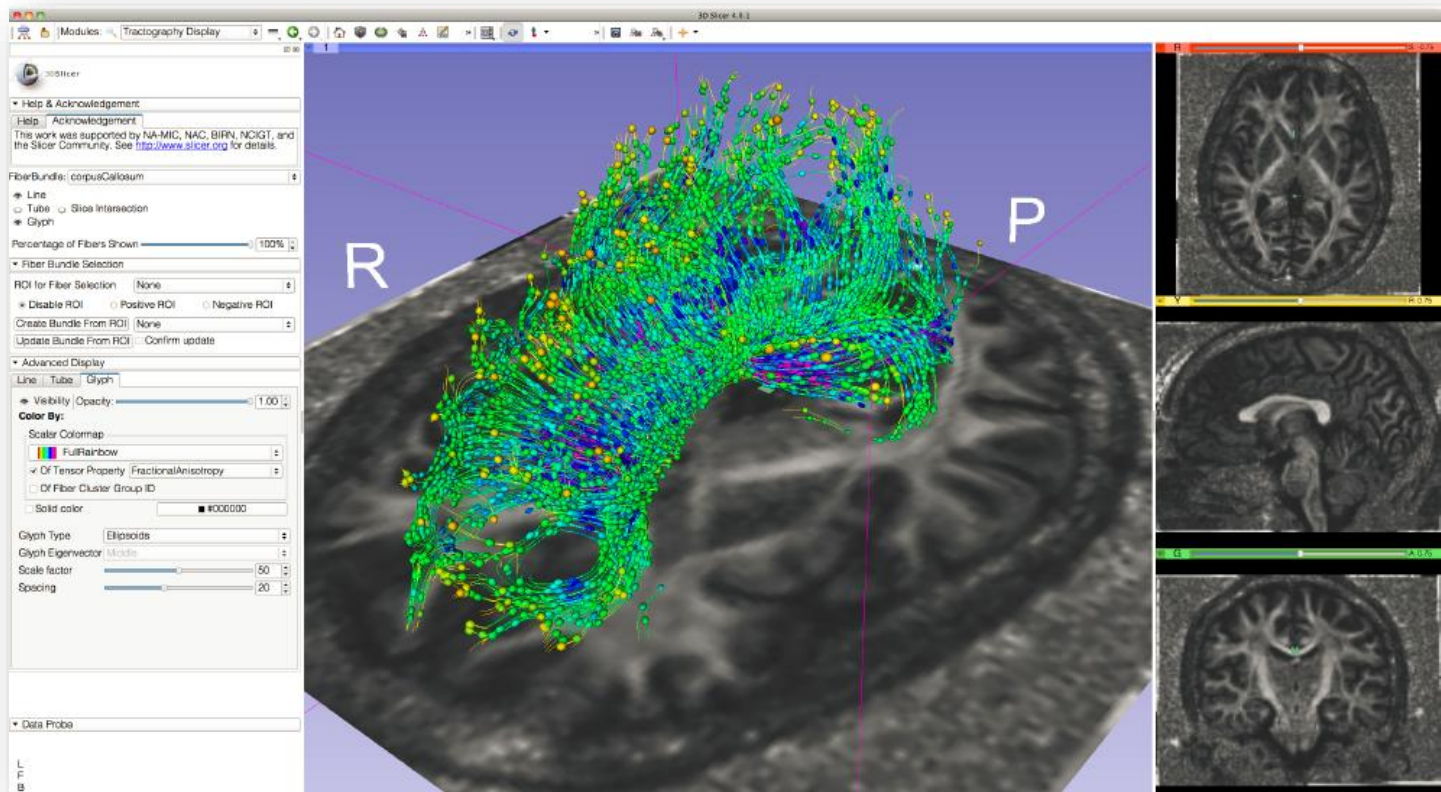
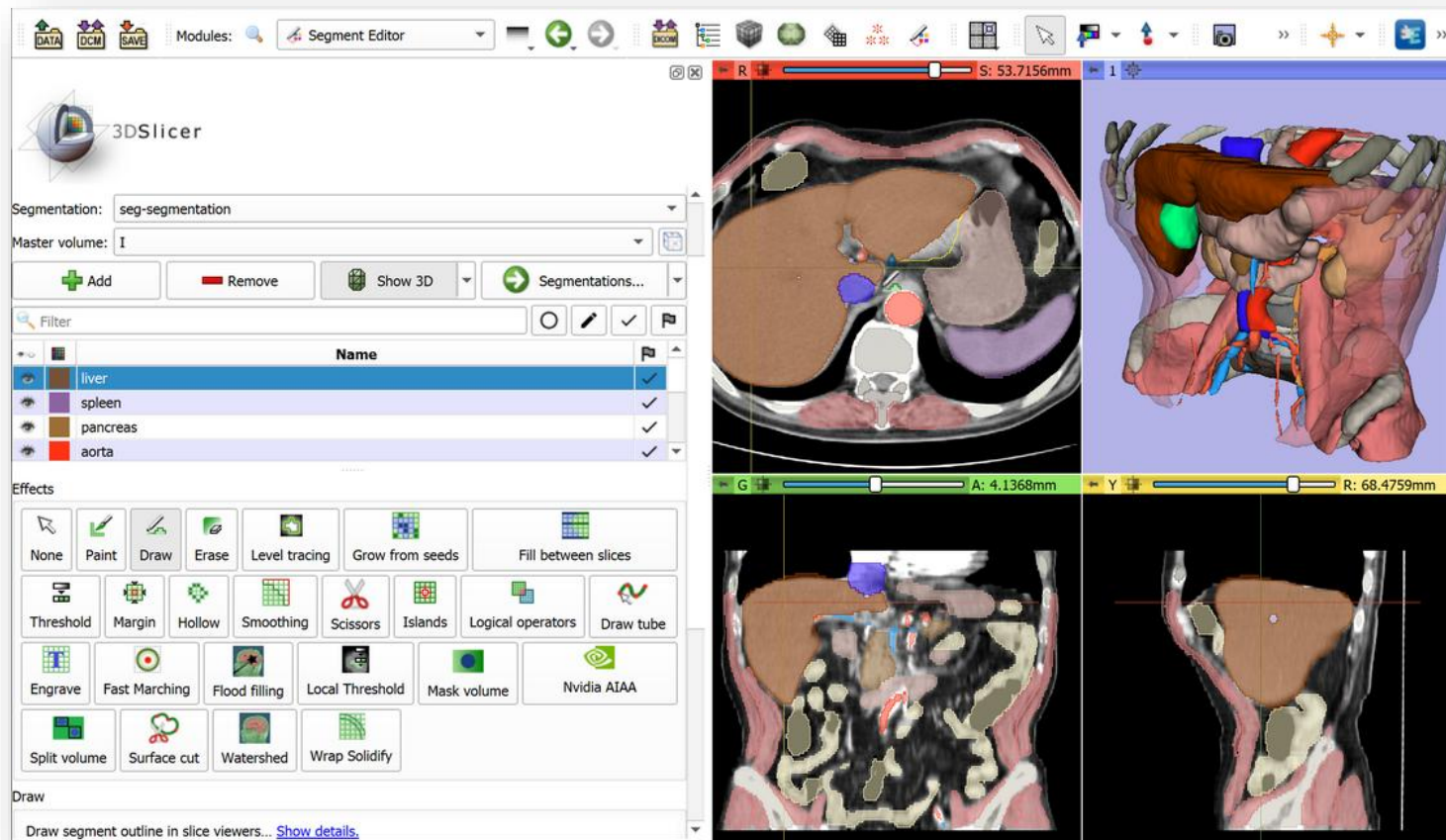


Image courtesy: Sonia Pujol

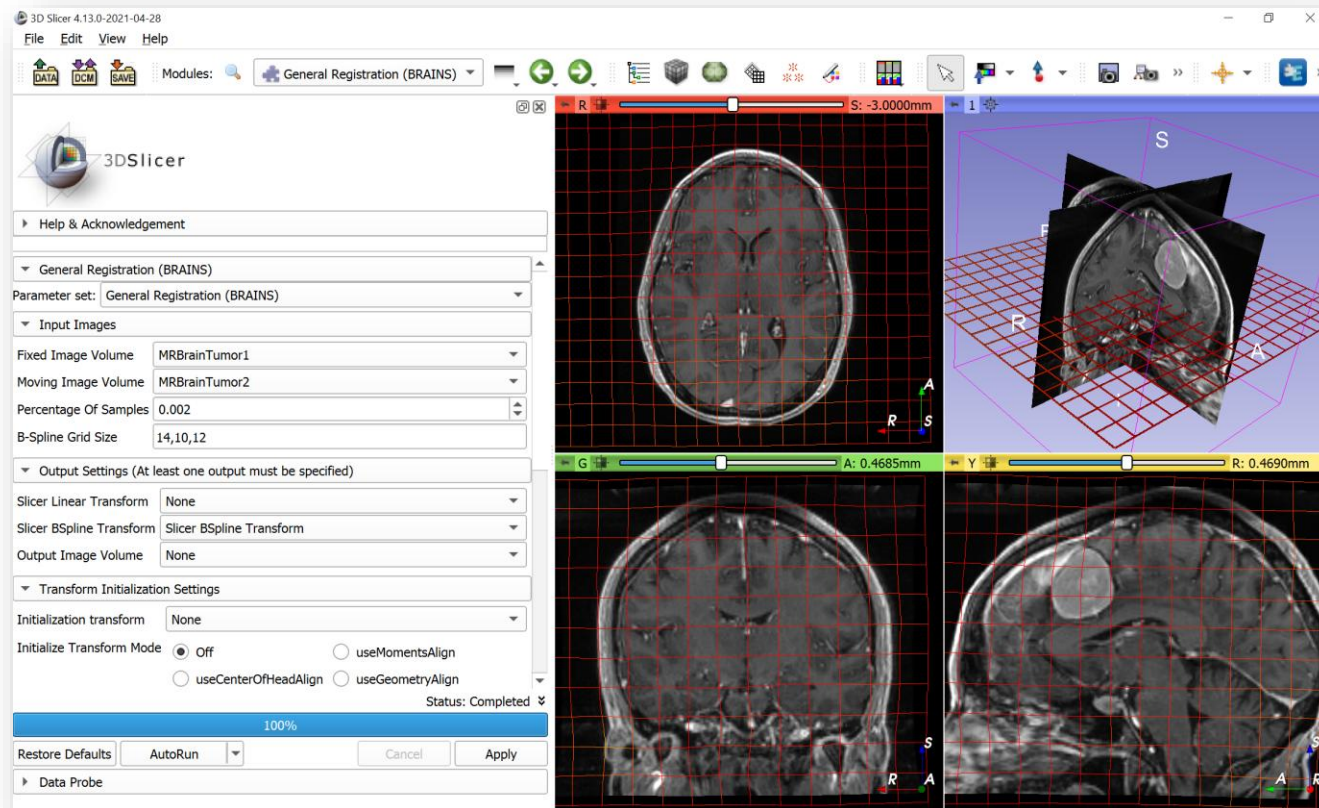
Modules type: Scripted loadable

- Written in Python
- Full Slicer API is accessible
- Simplest way to extend/customize Slicer



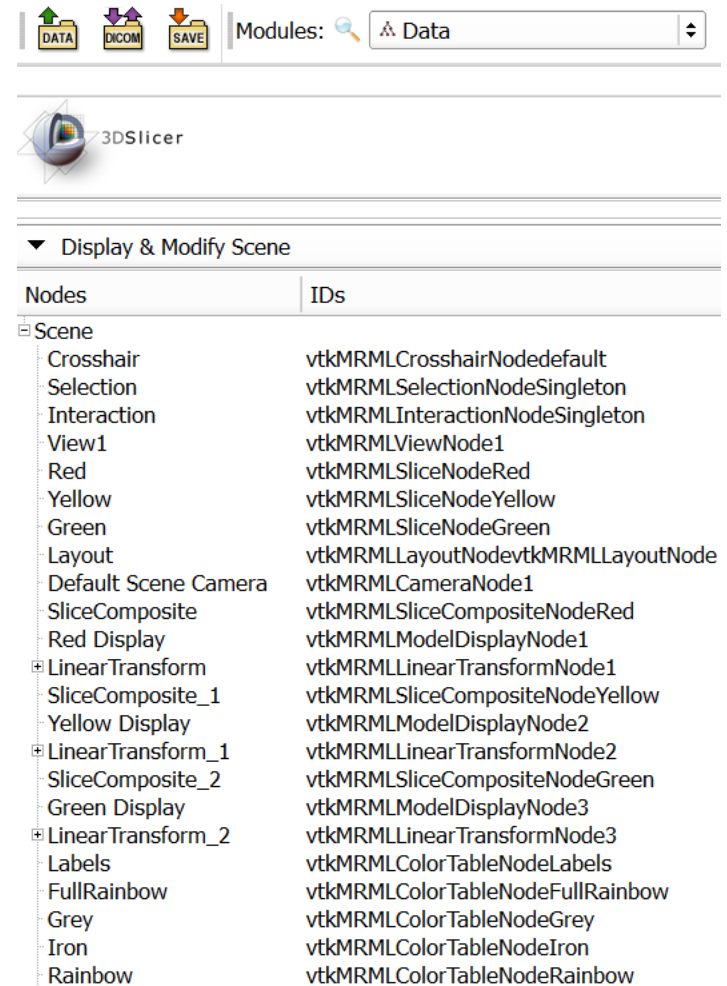
Module type: CLI

- CLI = Command-line interface
- Slicer can run any command-line application from the GUI
- Can be implemented in any language (C++, Python, ...)
- Inputs and outputs specified in XML file, GUI is generated automatically
- Good for implementing computational algorithms



Slicer data model

- **MRML scene:** shared storage of all data objects (MRML: Medical Reality Markup Language)
 - List of MRML nodes, each identified by a unique string ID
 - References, observations between nodes
- Modules communicate through reading/writing MRML nodes
 - Modules do not need to know about each other!



MRML node

- Responsibilities:
 - Store data
 - Serialization to/from XML for file storage
 - No display or processing *methods*
- Basic types:
 - Data node
 - Display node: visualization options for data node content; multiple display nodes allowed
 - Storage node: what format, file name to use for persistent storage of the data node content

Scripted module implementation

Module
(*MyFirst*)

Widget
(*MyFirstWidget*)

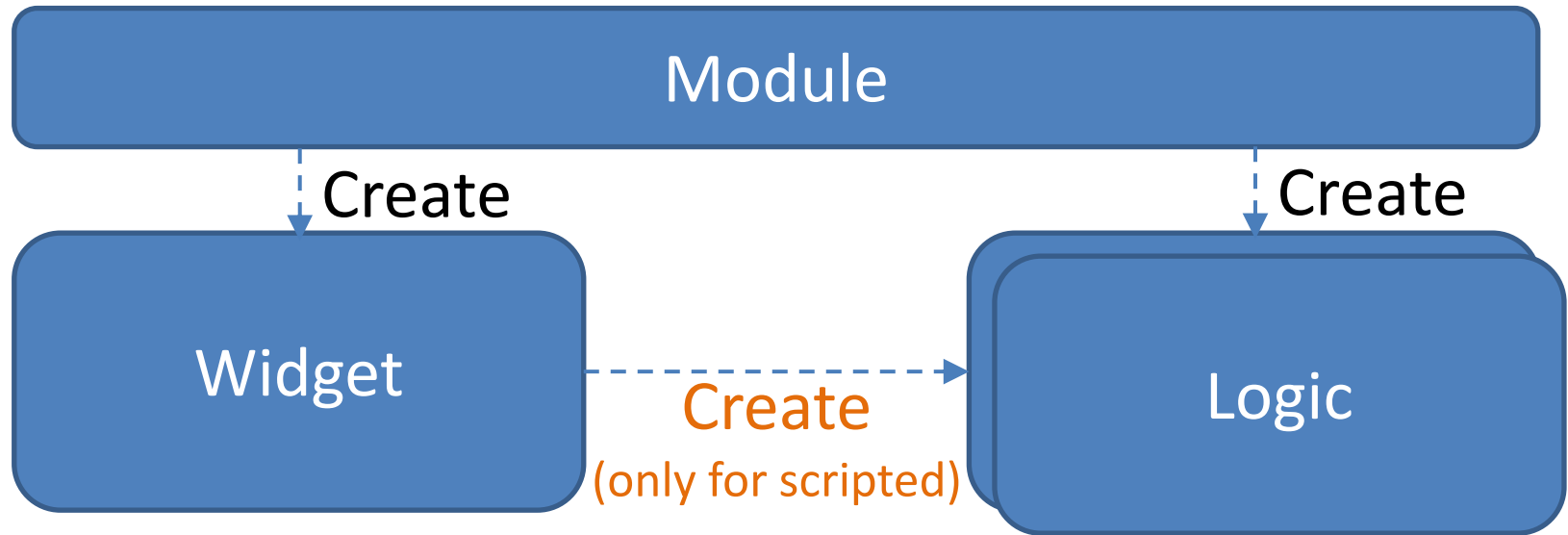
Logic
(*MyFirstLogic*)



Module class

- Required. Only one global instance exists:
`module = slicer.util.getModule('Volumes')`
- Stores module name, description, icon, etc.
- Creates and holds a reference to logic and widget:
`widget = slicer.util.getModuleWidget('Volumes')`
`logic = slicer.util.getModuleLogic('Volumes')`
 - In scripted loadable modules: `getModuleWidget()` requires the widget to have a `logic` member. This is the case for module with **passive logic**.
 - Widget and logic has not much use for CLI modules. Use them via MRML (edit CLI parameter node) and `slicer.cli.runSync()`.

Scripted module implementation

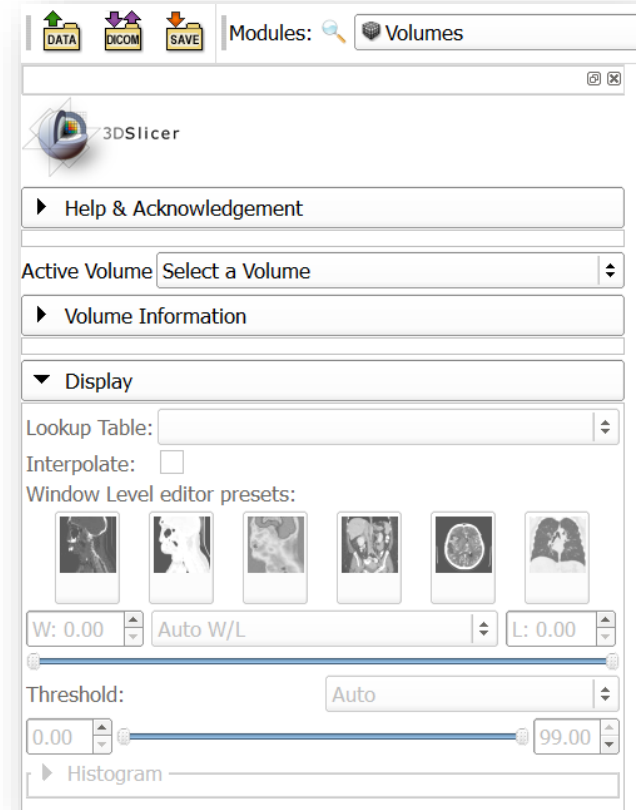


Scripted module logic is not created automatically (to facilitate dynamic reloading of the module without restarting the application). The logic has to be instantiated manually:

- Option A: Passive logic. Create logic object in the Widget class. This is the simplest and most commonly used (used in the scripted module template). If other modules need to access the logic, they may instantiate the other module's logic (see [example](#)).
- Option B: Active logic. Create logic object on *startupCompleted()* signal and store it in the module class. For example, this is the case when the logic observes the scene and reacts to changes, modifies nodes automatically. Only one such logic is allowed per application, because multiple logics could interfere with each other.

Widget class

- Needed if the module has a user interface
- Typically only one global instance exists
- Defines the module's user interface
- Keeps user interface and nodes in sync (observes MRML nodes to get change notifications)
- Launches processing methods implemented in the logic class

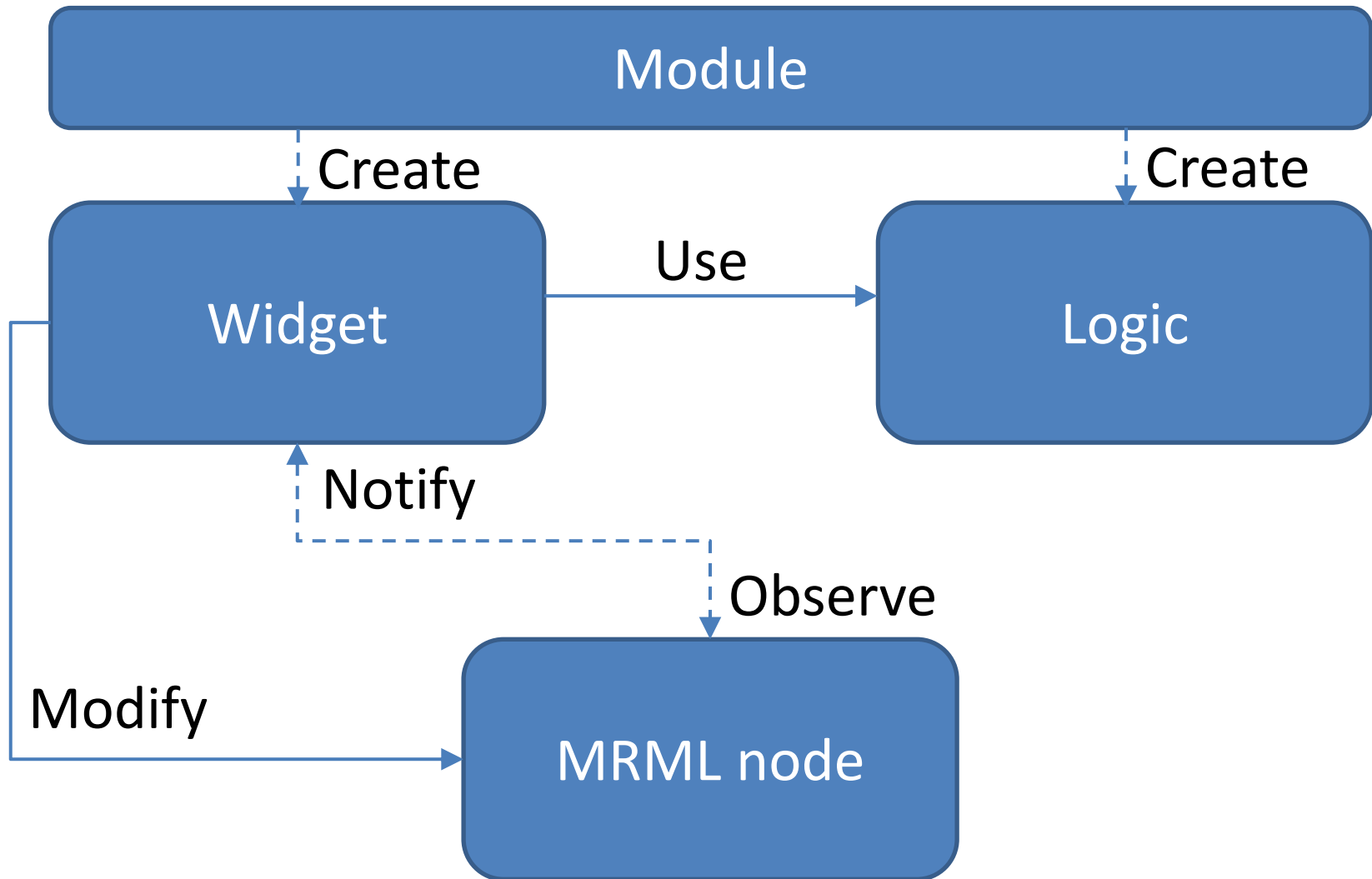


Widget class

- Include a parameter node selector at the top (or use a singleton parameter node)
- If a parameter node is selected then add an observer to its modified events; if modified then call widget's `updateGUIFromParameterNode()` method
- If the user changes the GUI then update MRML node by calling methods that update values in the parameter node. A shared `updateParameterNodeFromGUI()` method may be used, too, which is called on any GUI user interaction.



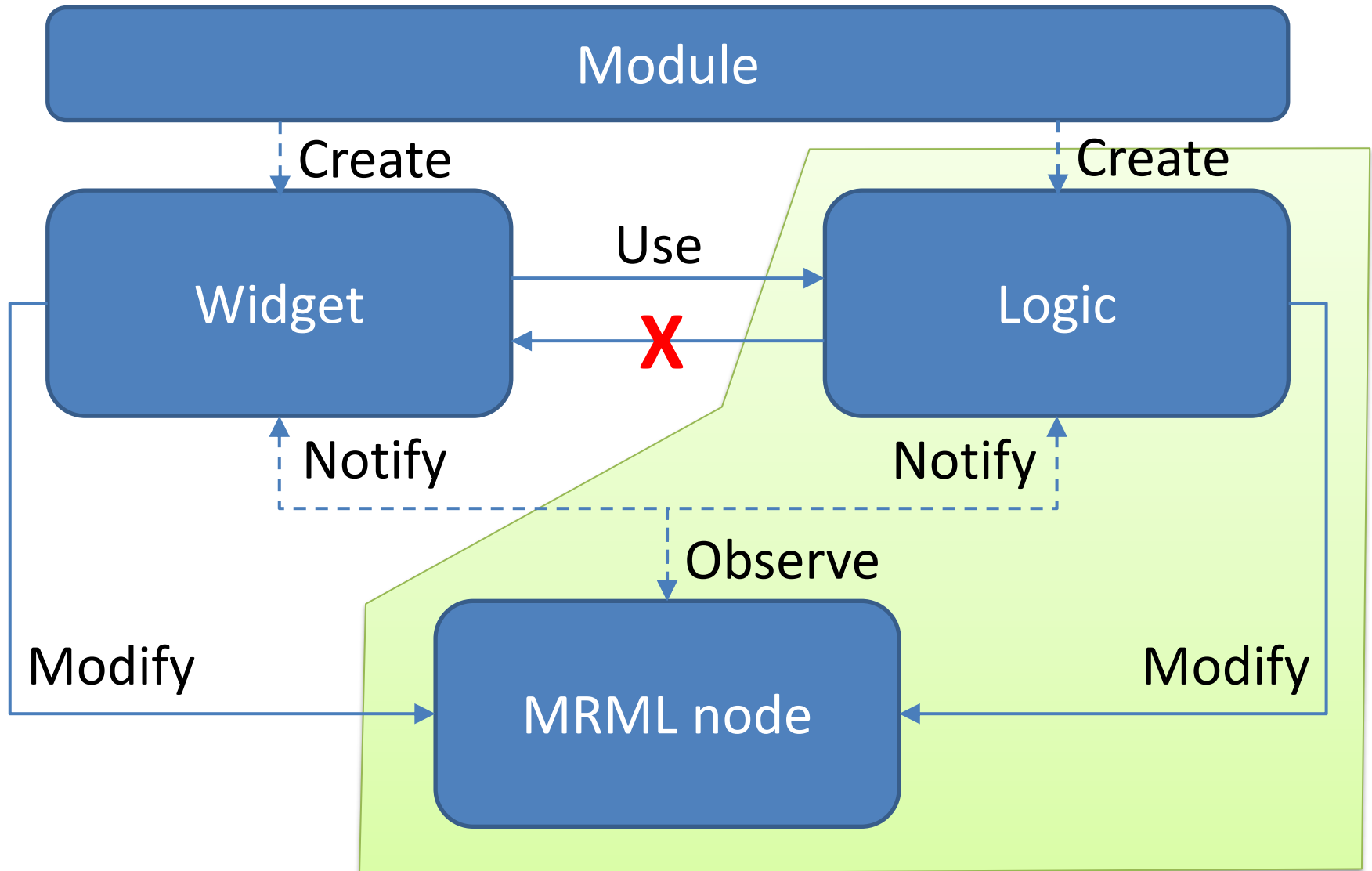
Scripted module implementation



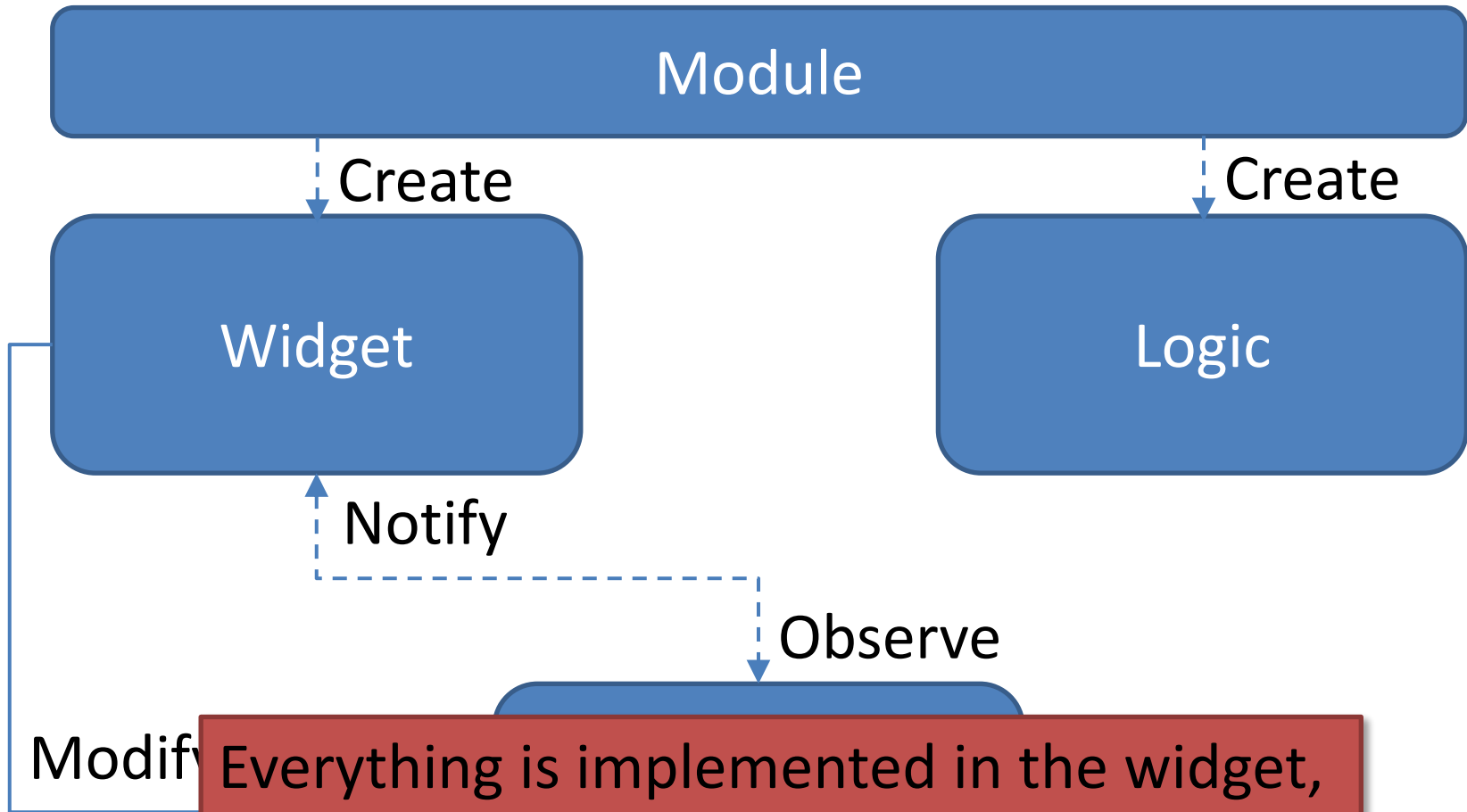
Logic class

- Needed if the module does any processing (always)
- The module must be usable from another module, just by calling logic methods
- Must not rely on the Widget class: the module must be usable without even having a GUI
- Logic may be instantiated many times (to access utility functions inside)
- Logic may observe nodes (active logic): only if real-time background processing is needed (e.g., we observe some input nodes and update other nodes if input nodes are changed)

Scripted module implementation

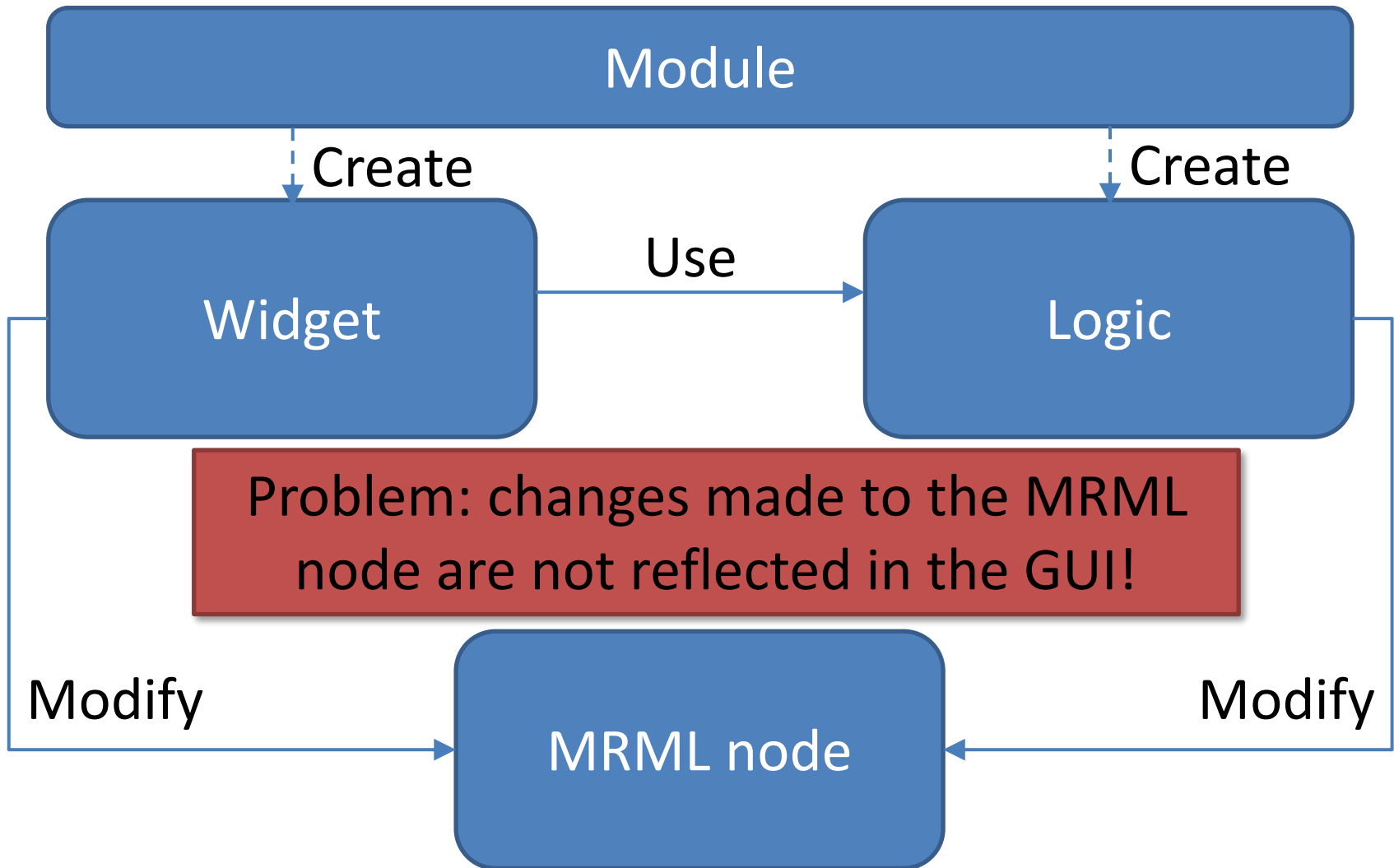


Common mistakes 1

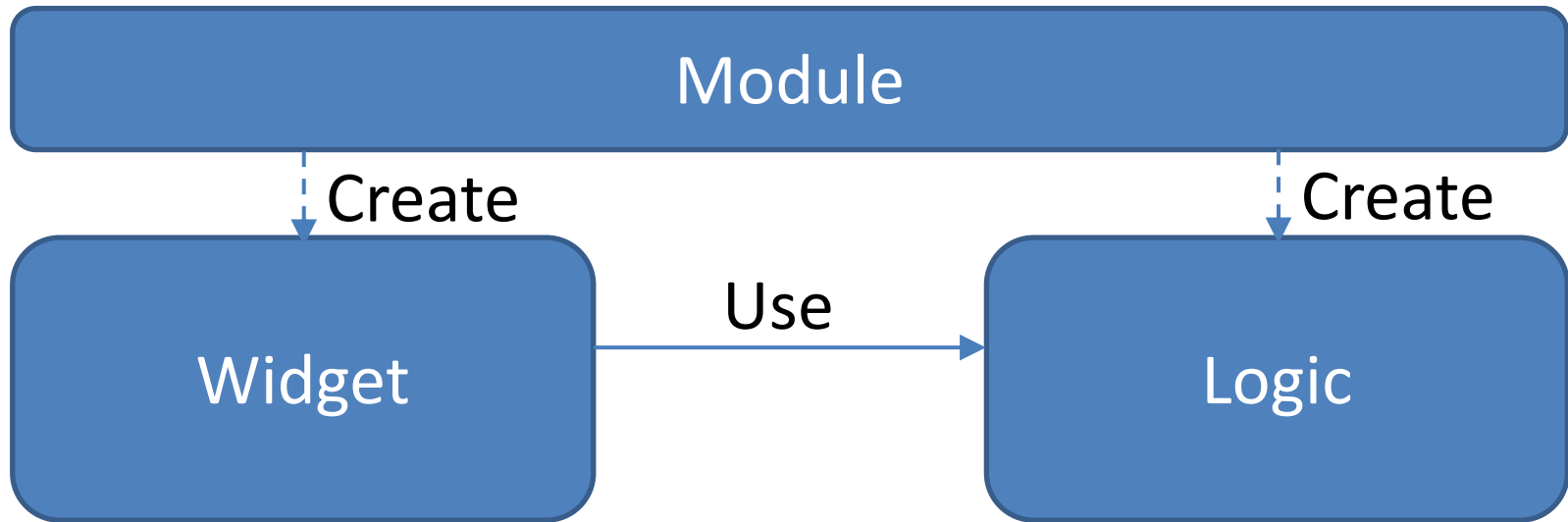


Everything is implemented in the widget, therefore the module is not usable from another module or with a custom GUI!

Common mistakes 2

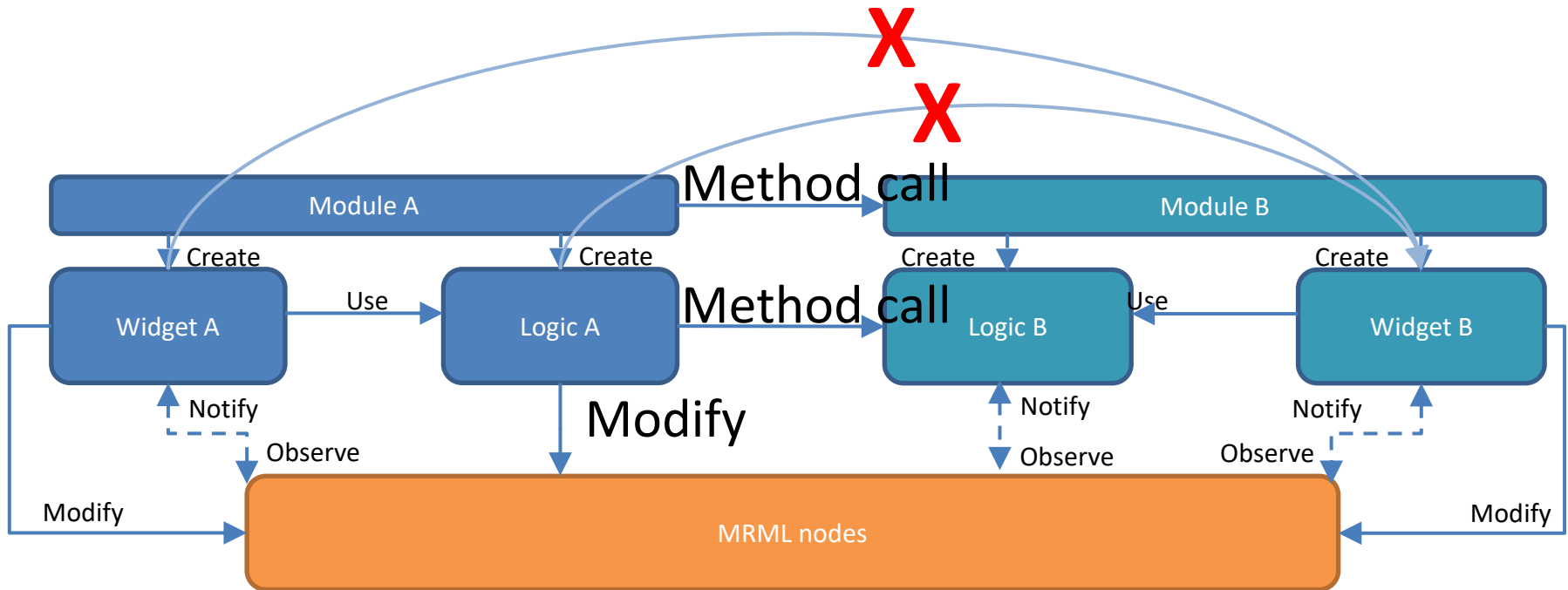


Common mistakes 3



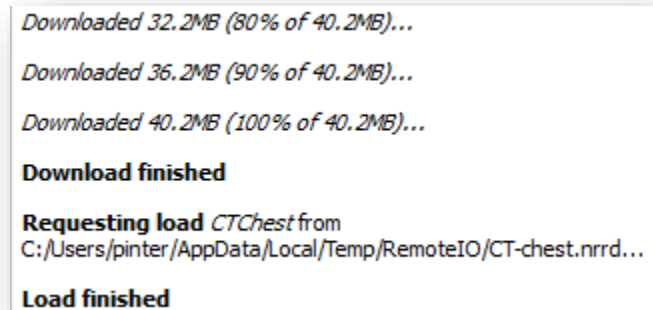
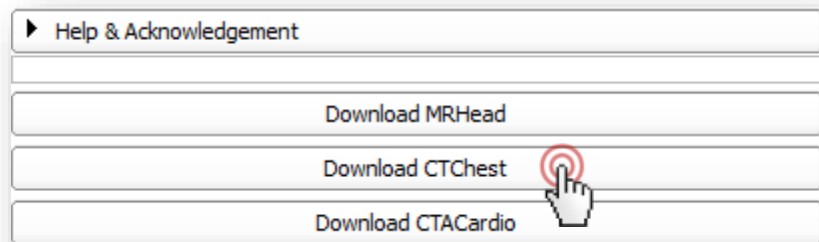
No parameter node is used. When the scene is saved and reloaded all the settings in the module user interface are lost!

How module A can use module B?

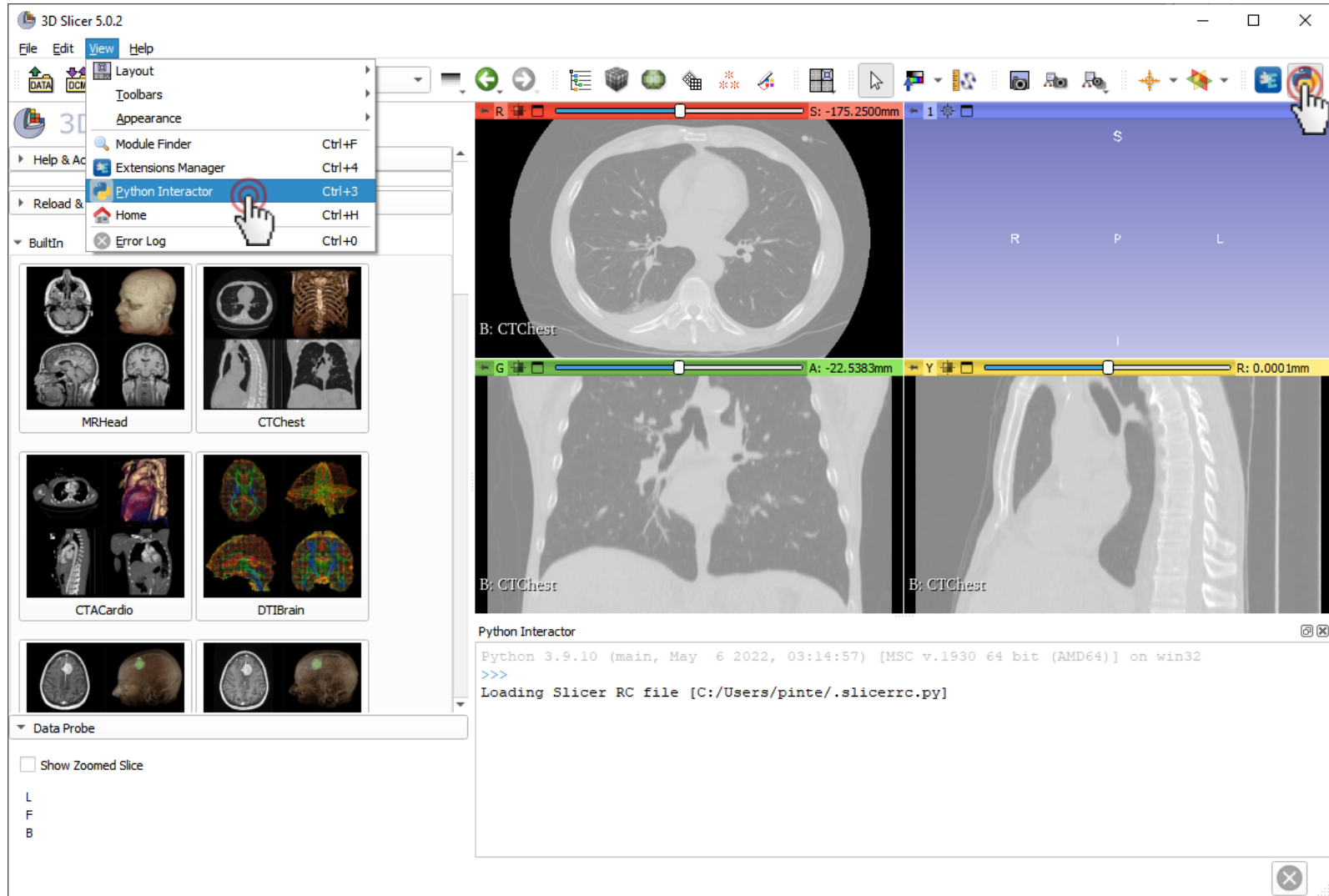


- Module logic may modify any MRML nodes – most common form of communication
- Module logic class may use another module's logic class
- Module class may use another module class, but this is rarely needed (e.g., to access features that require Qt or other GUI classes)

Launch Slicer and load data

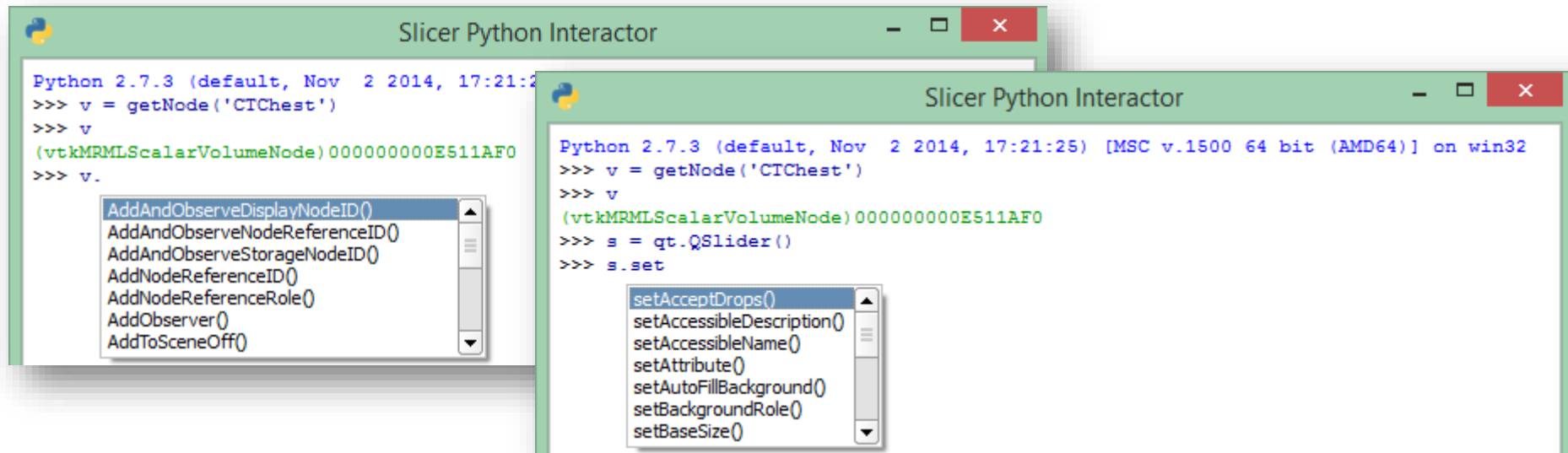


Introducing the python console



Auto-completion feature

- Essential tool that provides API information
- Press TAB to bring up auto-complete window to
 - Explore available functions of a certain object
 - Save typing



Accessing the MRML scene and nodes

- Using utility functions – in slicer.util

```
v = getNode( 'CTChest' )
```

OR

```
v = getNode( 'CT*' )
```

getNode: somewhat ambiguous, recommended for testing & debugging only

- Accessing MRML scene directly

```
v=slicer.mrmlScene.GetFirstNodeByName( 'CTChest' )
```

OR

```
v=slicer.mrmlScene.GetFirstNodeByClass( 'vtkMRMLSCalarVolumeNode' )
```



Information about variables

- Get variable type and pointer: enter the variable name

v

```
(vtkMRMLScalarVolumeNode)0000008FF76243B8
```

Note: It's always good to check the variable after you create it

- Show node content: all members and attributes inheritance tree

```
print(v)
```

- Show node API: description of all methods

```
help(v)
```



Manipulating Volumes

- Setting window/level values programmatically

```
vd = v.GetDisplayNode()
```

```
vd.SetAutoWindowLevel(0)
```

```
vd.SetWindowLevel(350,40)
```

- What methods/parameters are available:

```
help(vd)
```

OR

```
vd
```

```
(vtkCommonCorePython.vtkMRMLScalarVolumeDisplayNode)0000021AD5436588
```

<http://www.slicer.org/doc/html/classes.html>



Manipulating Volumes

- Accessing, changing voxels – using numpy
- Get voxel value at (100,200,30) position

```
va = slicer.util.arrayFromVolume(v)  <= get voxels as a numpy array  
va[100,200,30]
```

-986 <= voxel value

- Thresholding

```
vaOriginal=va.copy()  <= save the original voxel values
```

```
va[va<200] = -3000
```

```
va[va>200] = 2000
```

```
slicer.util.arrayFromVolumeModified(v)  <= indicate to Slicer that  
updates are completed
```

- Process with arbitrary function

```
va[:] = vaOriginal[:] * 2.5 - 500; v.Modified()
```



Load model

<http://perk-software.cs.queensu.ca/plus/doc/nightly/modelcatalog/>

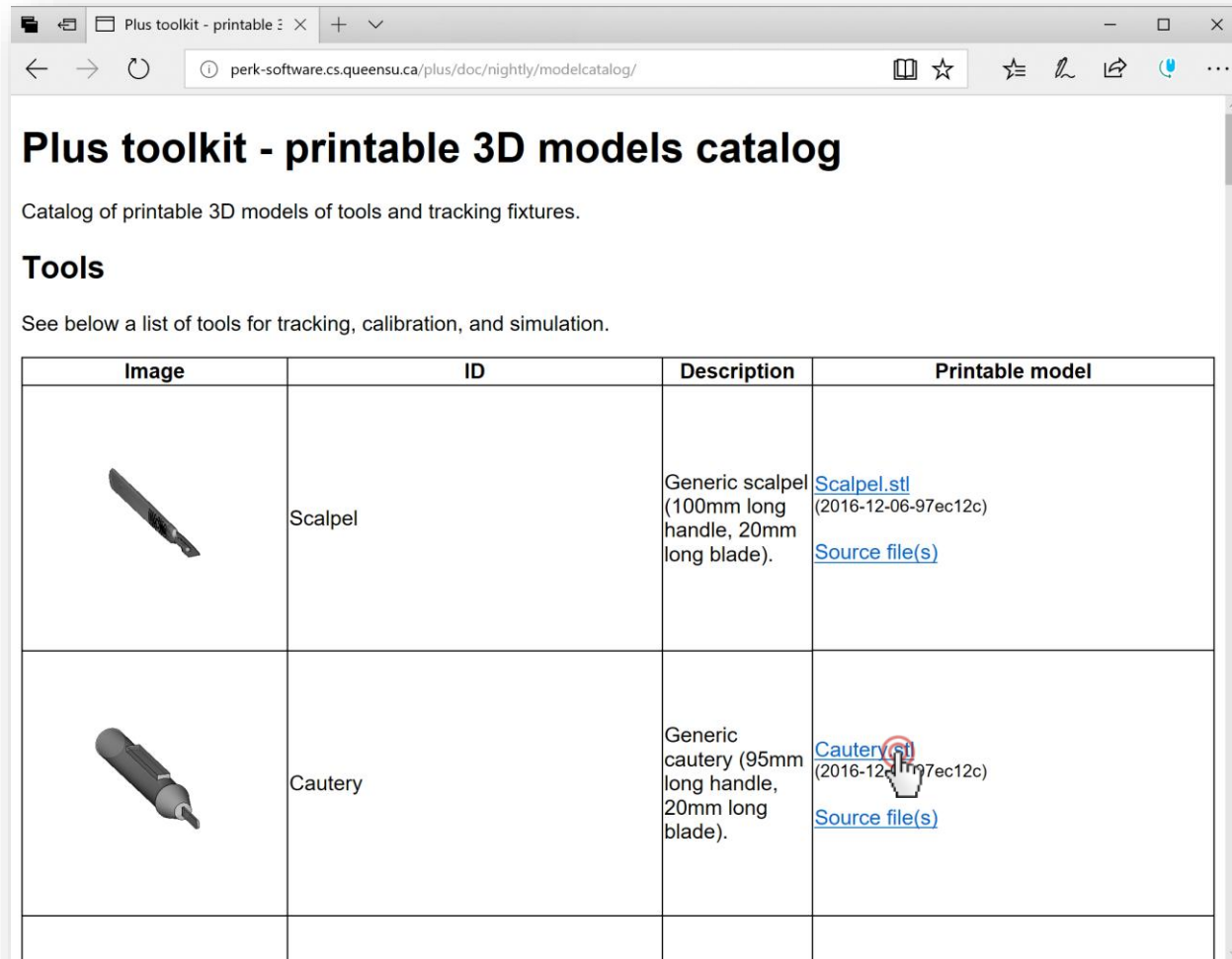




Image	ID	Description	Printable model
	Scalpel	Generic scalpel (100mm long handle, 20mm long blade).	Scalpel.stl (2016-12-06-97ec12c) Source file(s)
	Cautery	Generic cautery (95mm long handle, 20mm long blade).	Cautery.stl (2016-12-06-97ec12c) Source file(s)

Manipulating Models

- Setting model color programmatically

```
c = getNode('Cautery')  
cd = c.GetDisplayNode()  
cd.SetColor(1,0,0)
```

- Change model to a sphere

```
s = vtk.vtkSphereSource()  
s.SetRadius(30)  
s.SetCenter(30,40,60)  
s.Update()  
c.SetAndObservePolyData(s.GetOutput())
```



Manipulating Markups

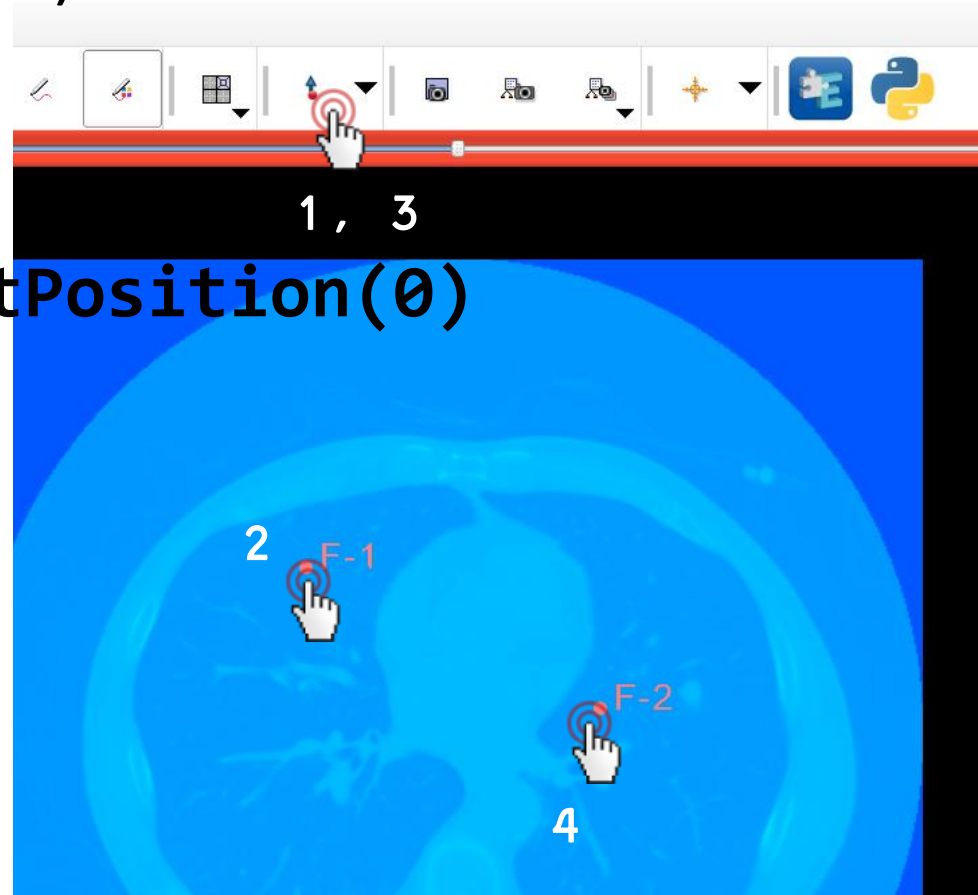
- Create 2 markup points (this creates a markup fiducial list node named 'F')

- Get markup position

```
f = getNode('F')
```

```
f.GetNthControlPointPosition(0)
```

```
(58.93727622783058,  
45.58082600473318,  
-170.2500000000001)
```



Observing MRML objects

```
def printPos(caller=None, event=None):
```

```
    f = getNode('F')
```

```
    pos = f.GetNthControlPointPosition(0)
```

```
    print(pos)
```

} These lines start with 2 spaces!

```
printPos()
```

```
[58.93727622783058, 45.58082600473318, -170.2500000000001]
```

```
obsTag=f.AddObserver(slicer.vtkMRMLMarkupsNode.PointModifiedEvent,  
printPos)
```

=> Drag-and-drop first fiducial

```
[20.267904116373643, 4.977985287703447, -170.2500000000001]
```

```
[21.92516292115039, 1.6634676781499849, -170.2500000000001]
```

```
[22.477582522742637, 1.1110480765577364, -170.2500000000001]
```

```
[24.687260929111574, 0.5586284749655164, -170.2500000000001]
```

```
[26.34451973388832, 0.00620887337326792, -170.2500000000001]
```

```
f.RemoveObserver(obsTag)
```



Share your tools

- In the spirit of the open-source paradigm, it is encouraged to share your tools
- The shared extensions
 - appear in the Extension Manager
 - are nightly tested on the Slicer Factory platforms
- How to share?
 - Fork ExtensionIndex from GitHub and upload your extension description (.s4ext) file
<https://github.com/Slicer/ExtensionsIndex>
 - Ask the core team to integrate (send a “pull request”)



Part 2

- Use python console in Slicer
- Simple scripted module
example

Python in general

- **Blocks defined by indentation: 2 spaces**

- Case sensitive

- Comments

```
# This whole row is a comment
```

```
"""This is a potentially multi-line comment"""
```

- Blocks defined by indentation
- An object refers to itself as `self` (in C++: `this`)
- Namespaces: `slicer`, `ctk`, `vtk`, `qt`
- Blocks ... indentation ... spaces!

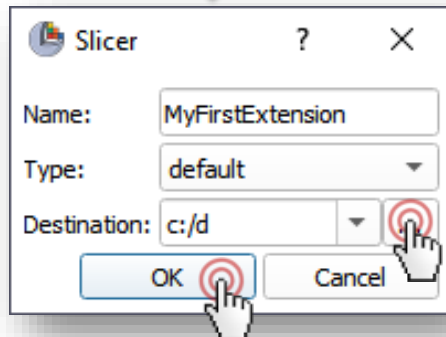
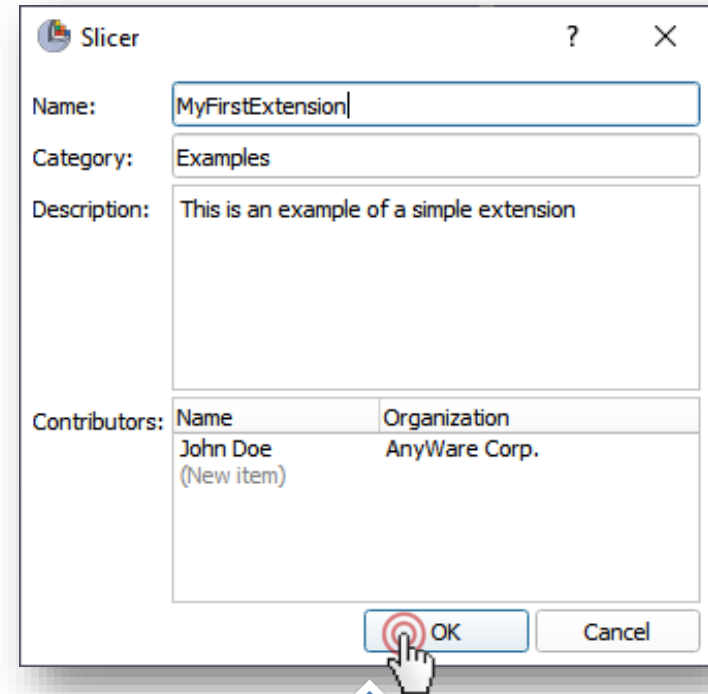
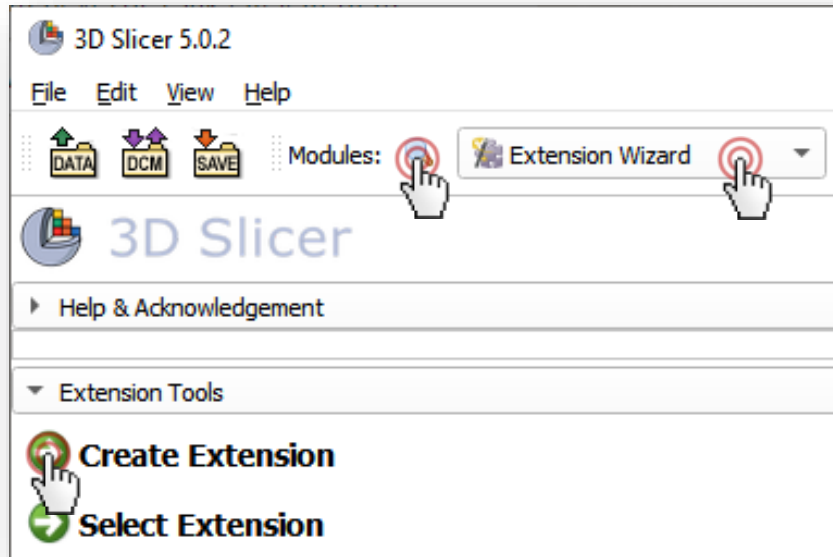


Text editor / IDE

- Using a proper text editor is essential
 - Replace all, Easy comment/uncomment, indent, ...
 - Syntax highlighting
 - Keyboard shortcuts
 - Recommended:
Cross-platform: [Visual Studio Code](#), [Sublime Text](#)
 - Windows-only: [Notepad++](#), Mac-only: Xcode
- Integrated development environment is even better:
 - Text editor + debugger, code browser, ...
 - Recommended: [Visual Studio Code](#), [PyCharm](#)

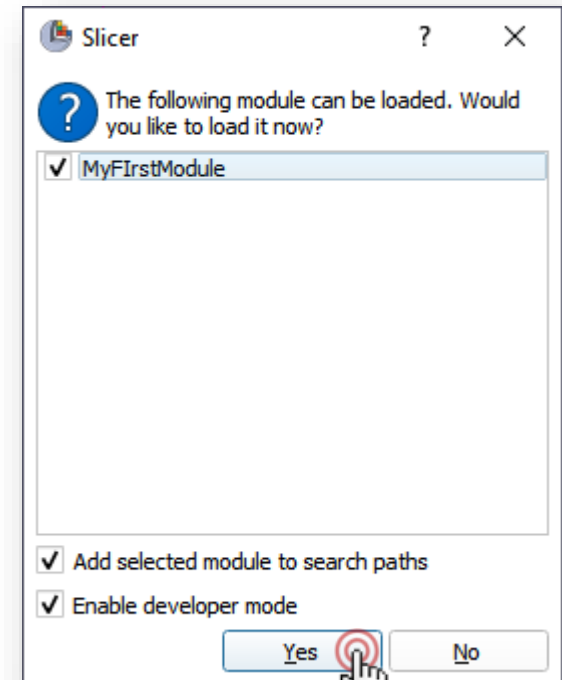
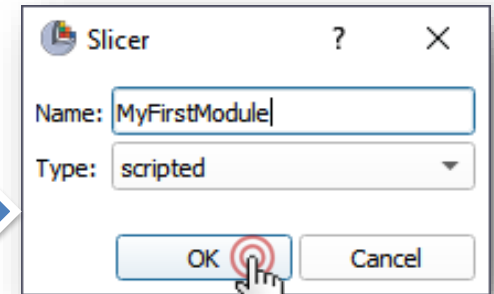
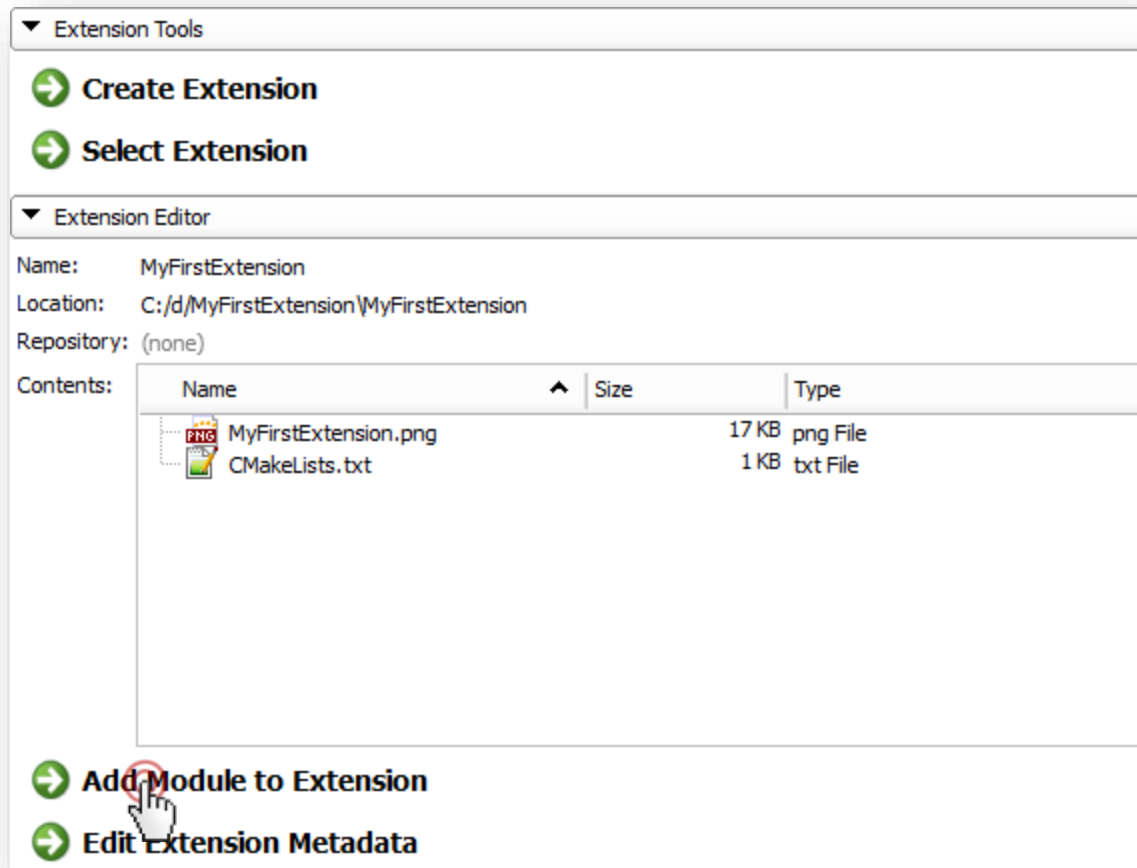


Create extension



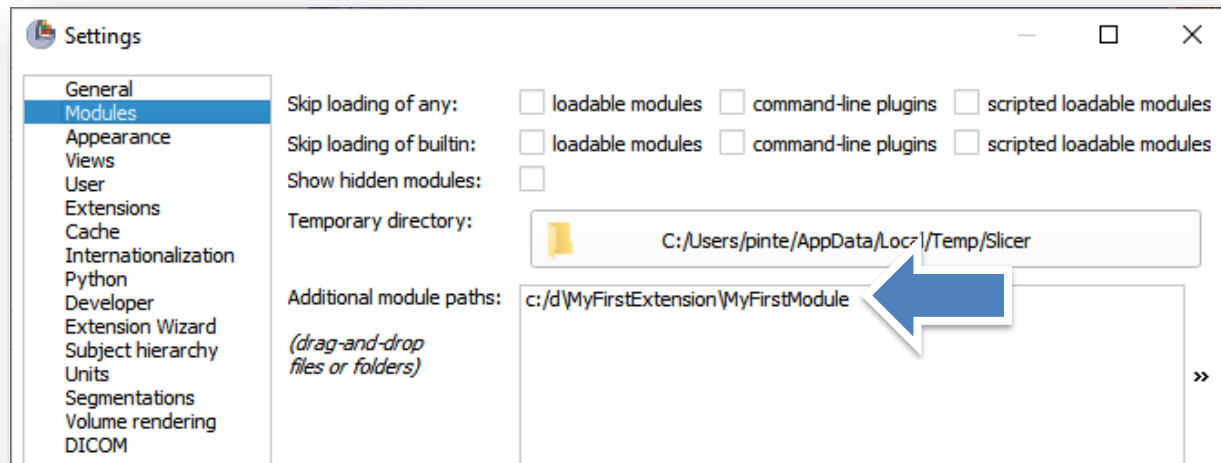
1. Enter name: *MyFirstExtension*
2. Choose destination: In your Git repo!

Create module



Module paths

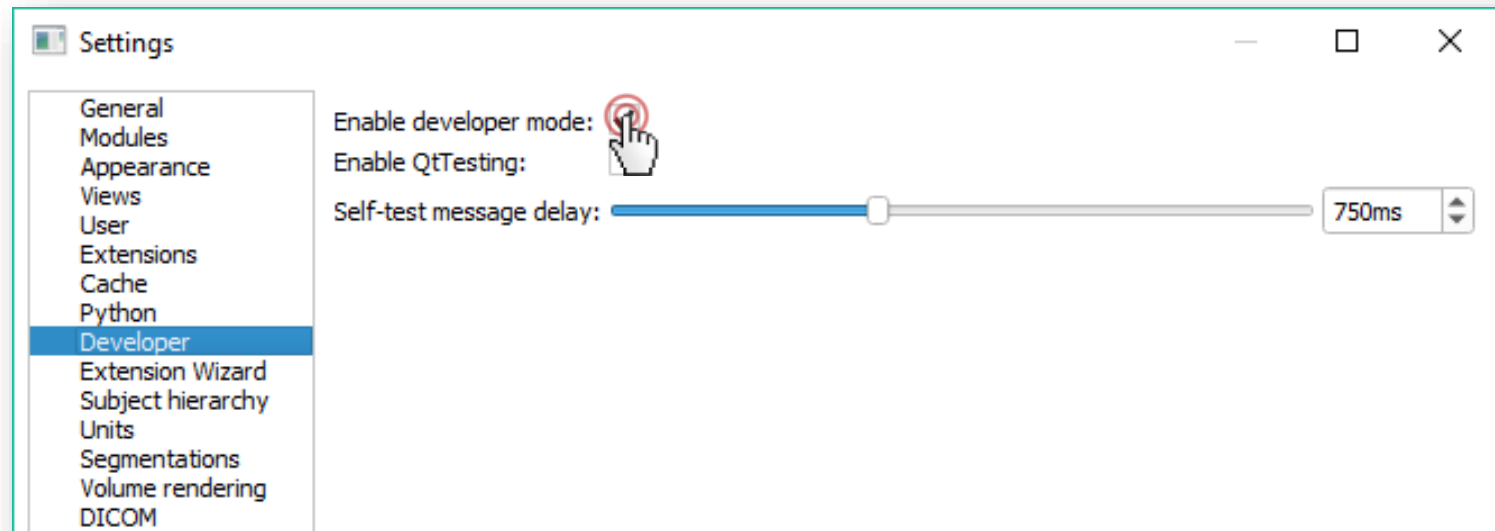
- To make Slicer load a module, add its folder (where the .py file is located) to “Additional module paths” in Application Settings



- You can drag&drop the .py file or folder
 - Checking the “Add selected module to search paths” checkbox added the listed modules to this path list already

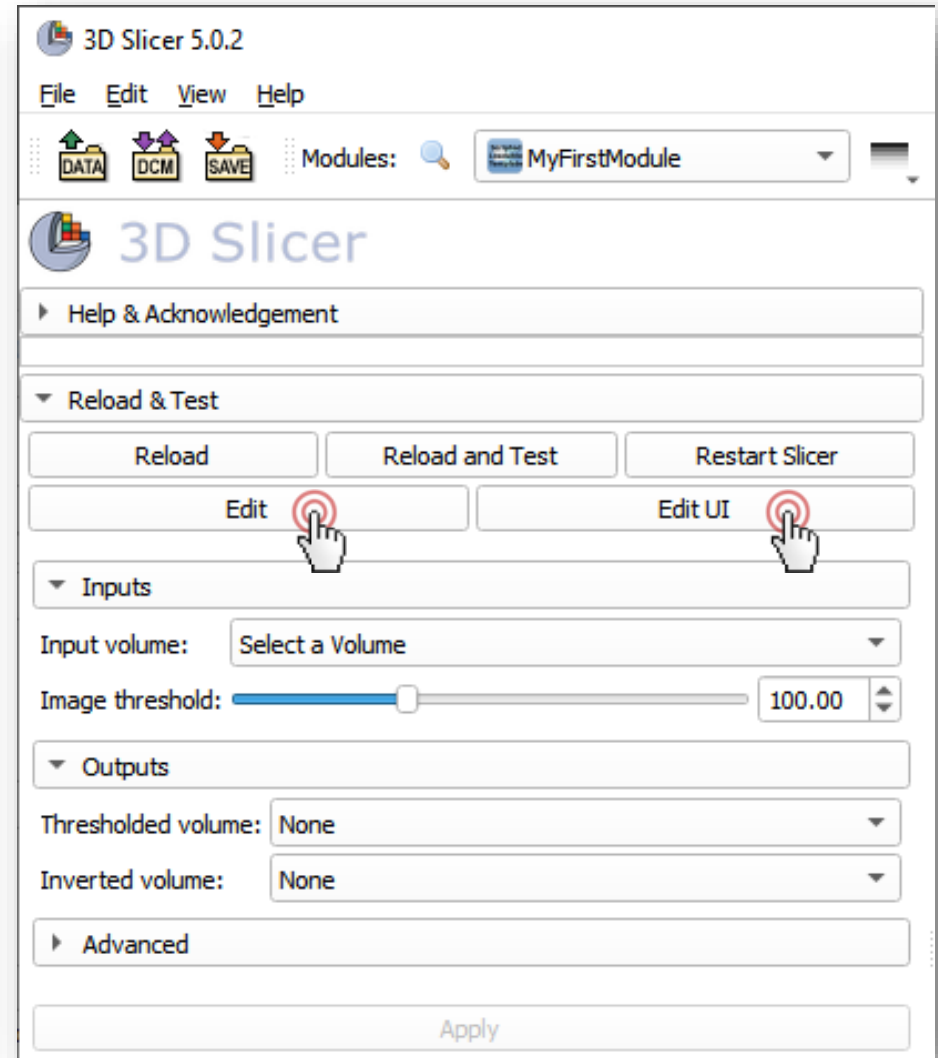
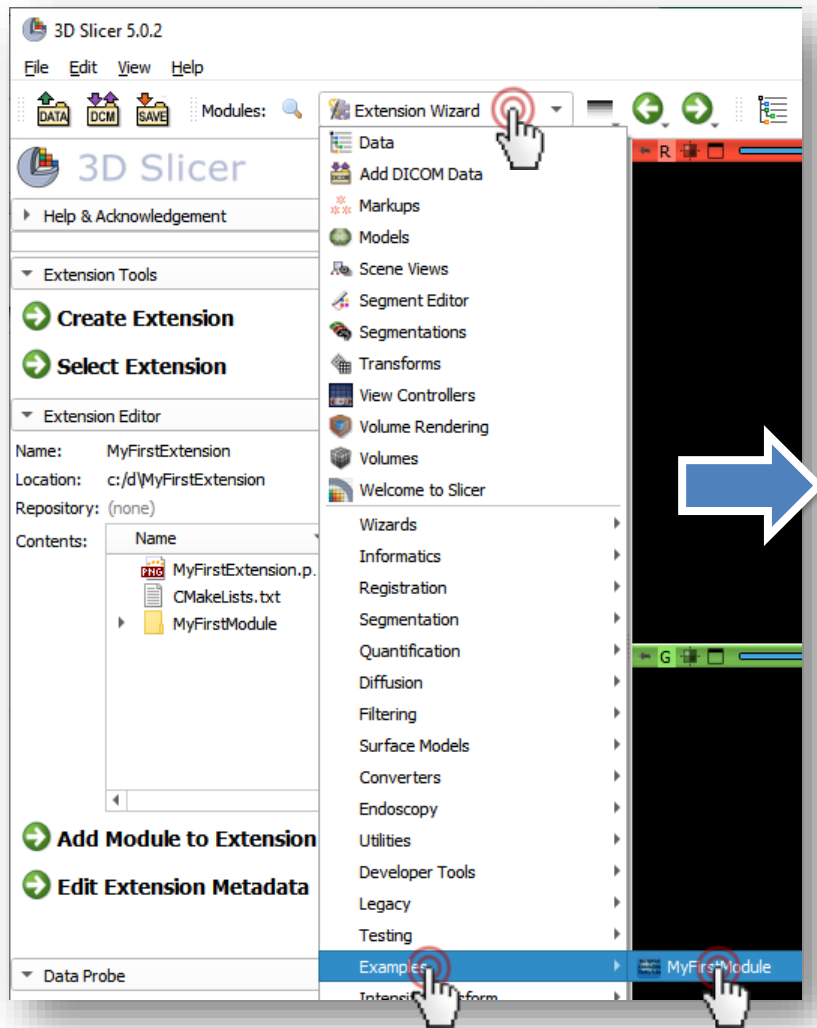
Developer mode

- Enable features useful for development
 - Allow dynamic reload of source code etc.
- In Application Settings



- Restart Slicer

Find the new module in Slicer



Commit your changes regularly

- Commit your changes
- New files need to be added explicitly
- Commit message should look like this:
"Re #7: Description of why I did what I did"
(do not describe what you did, it's obvious from the diff)
- When you think you're done
"Test #7: Description of why I did what I did"
- When everybody agrees you're done
"Fixed #7: Description of why I did what I did"
- Pull (Fetch+Rebase) before each commit!

Write our scripted module #1

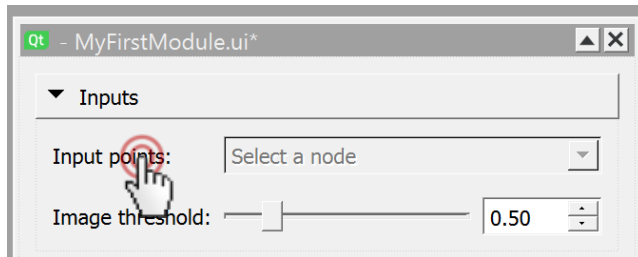
- Rename the module in the .py file

```
def __init__(self, parent):  
    ScriptedLoadableModule.__init__(self, parent)  
    self.parent.title = _("Center of Mass")
```

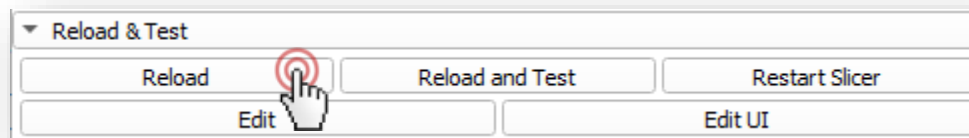
Note: the `_()` function is used for internationalization, i.e., to be able to translate the user interface of modules to various languages. Slicer's translation infrastructure collects all the strings used as arguments in `_()` functions and offer it for translation. You can learn more about translation of 3D Slicer in the [LanguagePacks extension's documentation](#).

Write our scripted module #2

- Customize widgets in Qt Designer: change the “Input volume” label to “Input points”, because our module will use a point list as input
 - Double-click “Input Volume” label and enter “Input **points**:”



- Don't forget to save the .ui file in Qt Designer
- Click “Reload” button in 3D Slicer, the module GUI will show “Input points” in the label



Write our scripted module #3

Change the source code so that the user can select point list markups node instead of an image volume as input. Class name corresponding to a point list markup is “vtkMRMLMarkupsFiducialNode”.

1. Near line 15: import vtkMRMLMarkupsFiducialNode

```
from slicer import vtkMRMLScalarVolumeNode
```

```
from slicer import vtkMRMLMarkupsFiducialNode
```



Add this line

2. Near line 114: replace vtkMRMLScalarVolumeNode by
vtkMRMLMarkupsFiducialNode

```
MyFirstModuleParameterNode
```

```
...
```

```
inputVolume: vtkMRMLScalarVolumeNode vtkMRMLMarkupsFiducialNode
```



Remove



Add



Write our scripted module #4

- Look for class `MyFirstModuleLogic`
- Insert this code in the logic class (for example, above the `process` function)

```
def getCenterOfMass(self, markupsNode):
    centerOfMass = [0,0,0]

    import numpy as np
    sumPos = np.zeros(3)
    for i in range(markupsNode.GetNumberOfControlPoints()):
        pos = markupsNode.GetNthControlPointPosition(i)
        sumPos += pos

    centerOfMass = sumPos / markupsNode.GetNumberOfControlPoints()

    logging.info(f'Center of mass for {markupsNode.GetName()}: {centerOfMass}')

    return centerOfMass
```

- Remember to save the file changes
- Click the Reload button in Slicer and check that there are no errors printed into the Python console.



Write our scripted module #5

- Change the `process` function to use the center of mass computation function and store the result in the logic object:

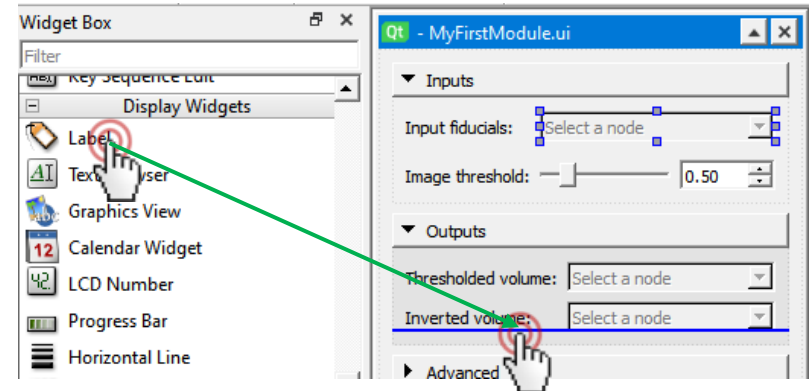
```
def process(self, inputMarkups, outputVolume, imageThreshold, enableScreenshots=0):  
    """  
    Compute center of mass of input markup points  
    """  
    self.centerOfMass = self.getCenterOfMass(inputMarkups)  
    return True
```



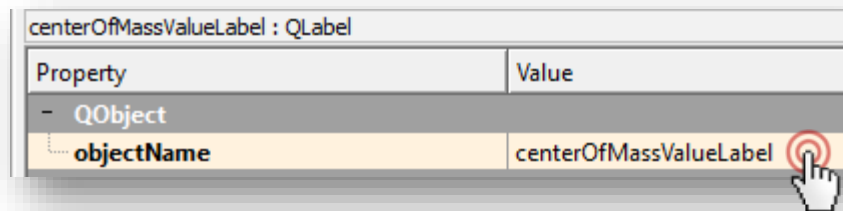
Write our scripted module #6

Add a new label to display the computed position.

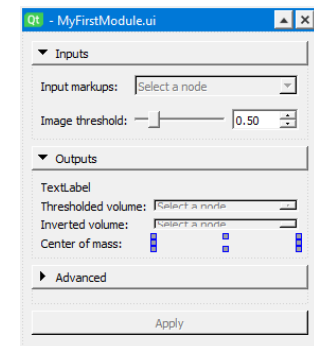
- Drag&drop a “Label” widget in the module widget at the bottom of the outputs section.



- Double-click the label to edit the text and enter “Center of mass:”
- Drag&drop a second one in the right column, make it empty
- Rename the object to **centerOfMassValueLabel**



The end result should look like this:



Write our scripted module #7

Update the widget class to show the result after computation.

- Set the computation result in the “centerOfMassValueLabel” in the “MyFirstModuleWidget” class:

```
def onApplyButton(self) -> None:
    """Run processing when user clicks "Apply" button."""
    with slicer.util.tryWithErrorDisplay(_("Failed to compute results."), waitCursor=True):
        # Compute output
        self.logic.process(self.ui.inputSelector.currentNode(),
                           self.ui.outputSelector.currentNode(),
                           self.ui.imageThresholdSliderWidget.value,
                           self.ui.invertOutputCheckBox.checked)
        self.ui.centerOfMassValueLabel.text = str(self.logic.centerOfMass)
```

- Update the code that checks if all the inputs are specified. We only need one input node

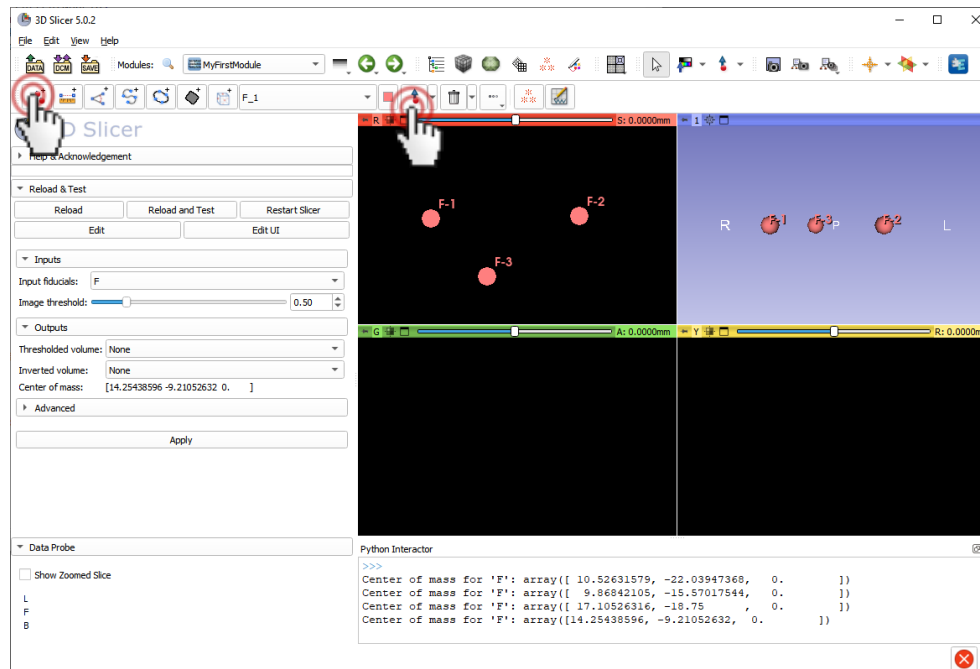
```
def _checkCanApply(self, caller=None, event=None) -> None:
    if self._parameterNode and self._parameterNode.inputVolume:
        self.ui.applyButton.setToolTip(_("Compute center of mass"))
        self.ui.applyButton.enabled = True
    else:
        self.ui.applyButton.setToolTip(_("Select input point list"))
        self.ui.applyButton.enabled = False
```

Pay attention to correct indentation!



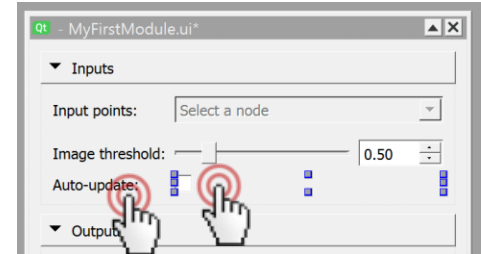
Try our scripted module

- Go to our module in *Examples / Center of Mass*
- Add a few markups
- Press *Apply*
 - 1. Display displacement center of mass position → works!
 - 2. Displays nothing → error can be seen in the Python interactor



Add auto-update checkbox GUI

- Add a new Label
- Double-click to rename to “Auto-update:”
- Add a checkbox
- Double click to remove the text from the checkbox label to follow Slicer user interface conventions
- Change “objectName” of the checkbox to: “autoUpdateCheckBox”
- Save the .ui file



Add auto-update checkbox widget #1

Implement an “observer” (a method that is called each time a node is modified) that is activated/deactivated with the auto-update checkbox.

1. Add variable that stores the currently observed markup node and the observer’s tag (an identifier that we need for removing the observer) to the `__init__` method.

```
def __init__(self, parent=None) -> None:
    """Called when the user opens the module the first time and the widget is initialized."""
    ScriptedLoadableModuleWidget.__init__(self, parent)
    VTKObservationMixin.__init__(self) # needed for parameter node observation
    self.logic = None
    self._parameterNode = None
    self._parameterNodeGuiTag = None
    self.observedMarkupNode = None
    self._markupsObserverTag = None
```



Add these lines

Add auto-update checkbox widget #2

2. Add a method to the “MyFirstModuleWidget” class that adds/removes an observer to the current markup node:

```
def onEnableAutoUpdate(self, autoUpdate):  
    if self._markupsObserverTag:  
        self.observedMarkupNode.RemoveObserver(self._markupsObserverTag)  
        self.observedMarkupNode = None  
        self._markupsObserverTag = None  
    if autoUpdate and self.ui.inputSelector.currentNode:  
        self.observedMarkupNode = self.ui.inputSelector.currentNode()  
        self._markupsObserverTag = self.observedMarkupNode.AddObserver(  
            slicer.vtkMRMLMarkupsNode.PointModifiedEvent, self.onMarkupsUpdated)
```

3. Add a callback method to the “MyFirstModuleWidget” class that is called when the observed markup node is changed:

```
def onMarkupsUpdated(self, caller=None, event=None):  
    self.onApplyButton()
```

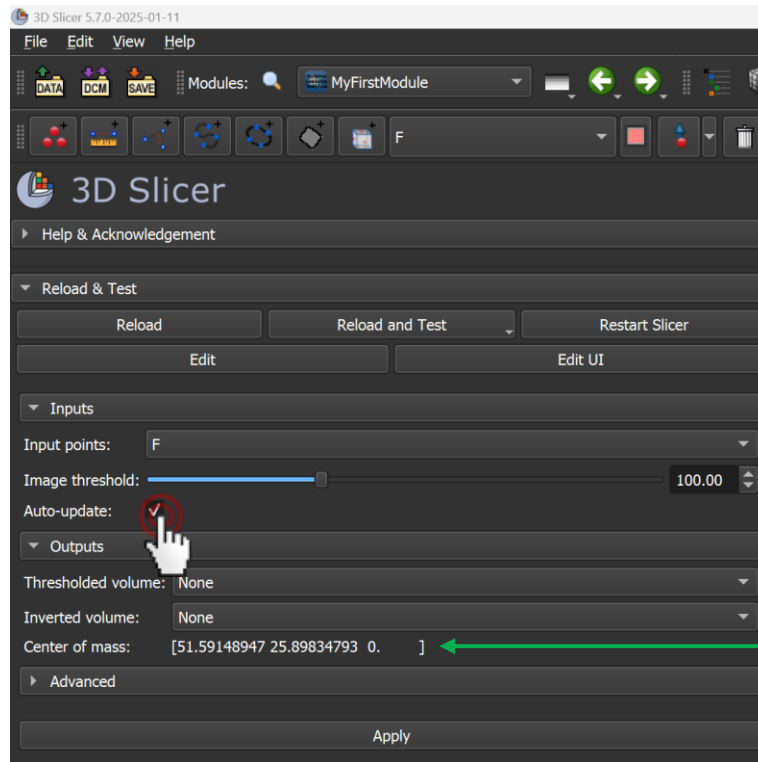
4. Make the auto-update checkbox enable/disable auto-update by adding a connection in “MyFirstModuleWidget” class “setup” method:

```
self.ui.autoUpdateCheckBox.connect("toggled(bool)", self.onEnableAutoUpdate)
```



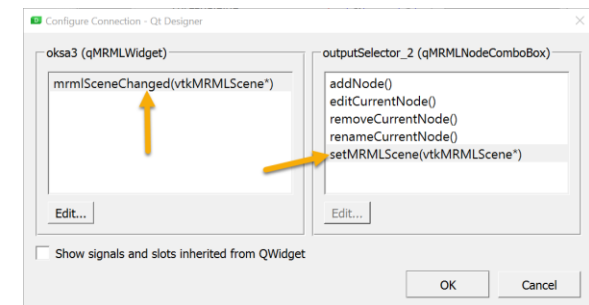
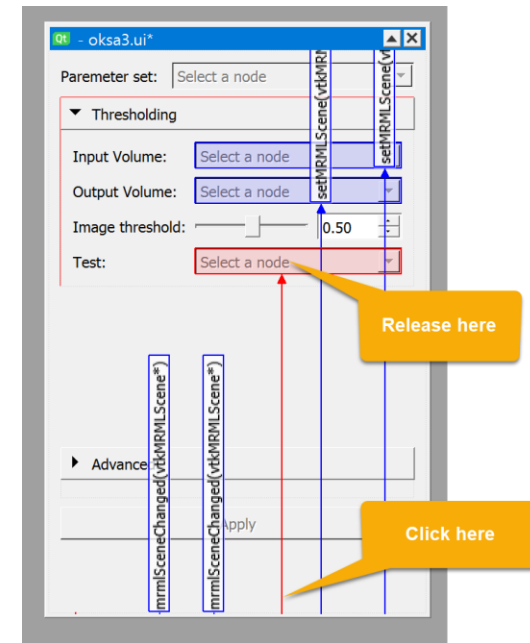
Try auto-update

- Reload the module
- Check auto-update checkbox
- Move markup points, the Center of mass value will update automatically



Tip #1: Adding MRML widgets

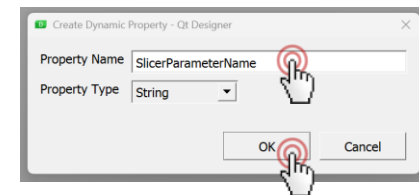
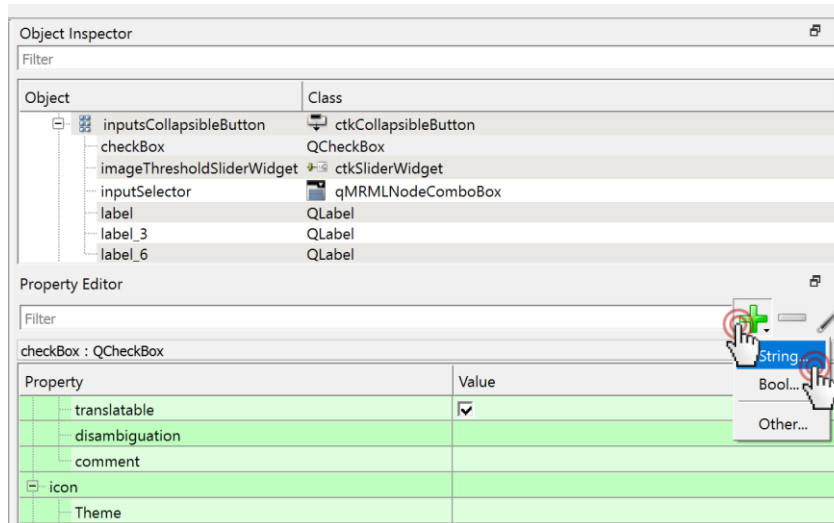
- MRML widgets, such as node selector (qMRMLNodeComboBox) require setting of the MRML scene
- Scene can be set in Designer by connecting the ***mrmlSceneChanged(vtkMRMLScene*)*** signal of the top-level widget to the ***setMRMLScene(vtkMRMLScene*)*** slot of the widget
 - Option A: Use *Signal/Slot Editor* (by default in the lower-right corner)
 - Option B: Switch to *Edit Signals/Slots* mode (menu: Edit -> Edit Signals/Slots or F4 key) and press mouse button in an empty area in the top-level widget, move the mouse over the MRML widget, and release the mouse button; then select the signal and slot



Tip #2: Binding GUI widgets to parameters

“Parameter node wrapper” infrastructure can be used to conveniently synchronize GUI widget (checkbox, slider, line edit, etc.) and a value in the parameter node. Values in the parameter node are saved into the scene and are easily accessible in the widget and logic classes. To bind a widget to a parameter:

1. Add a new “String” property named “SlicerParameterName”

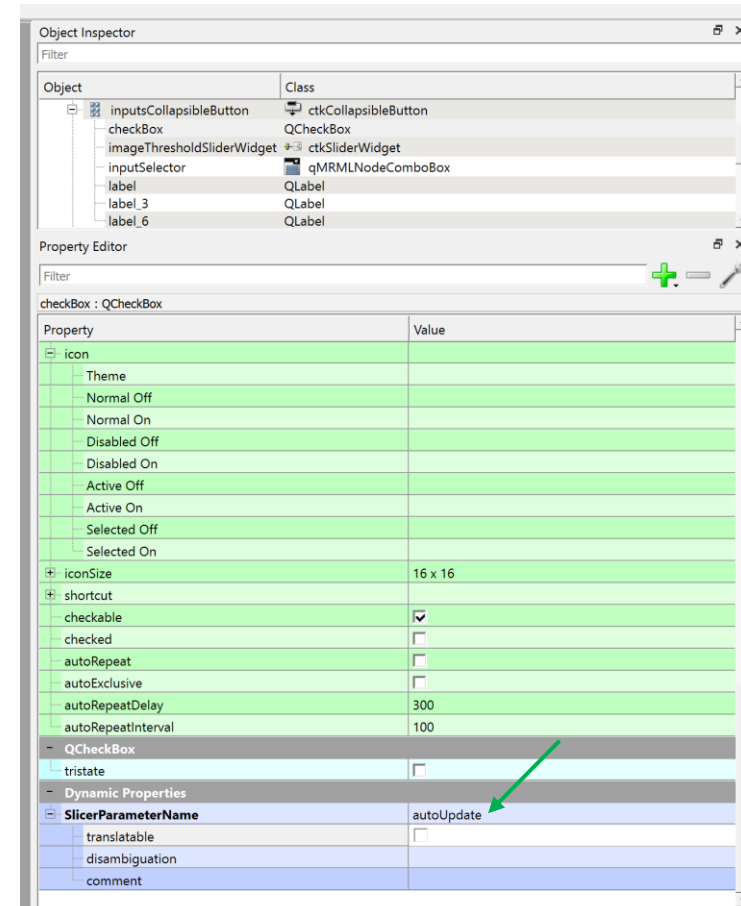


Tip #2: Binding GUI widgets to parameters

2. Disable “translatable” property (this string is the same in all languages, must not be translated)
3. Set the “SlicerParameterName” value to the parameter name used in the .py file:

```
@parameterNodeWrapper
class MyFirstModuleParameterNode:
    inputVolume: vtkMRMLMarkupsFiducialNode
    ...
    invertThreshold: bool = False
    thresholdedVolume: vtkMRMLScalarVolumeNode
    invertedVolume: vtkMRMLScalarVolumeNode
    autoUpdate: bool = False
```

Note that after this change, when the module is reloaded, the state of the button is now preserved.

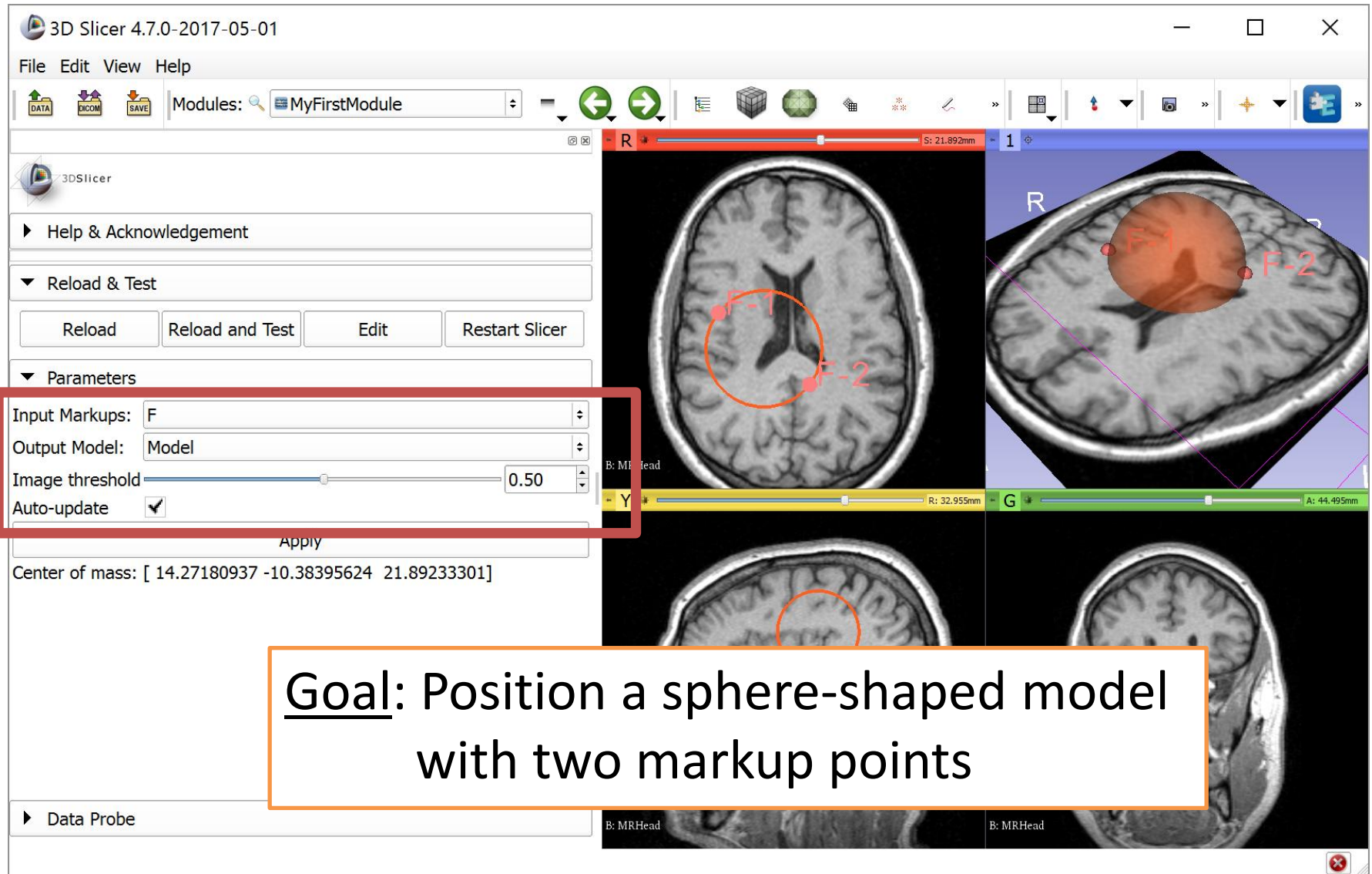


Part 3

Write simple scripted module
(somewhat) independently



Task description



Goal: Position a sphere-shaped model with two markup points

API documentation

Generic computing and GUI libraries

Toolkit	API documentation
Python	https://docs.python.org/es/3.9/index.html
Numpy	http://docs.scipy.org/doc/numpy/reference
VTK	https://vtk.org/doc/nightly/html
SimpleITK	http://www.itk.org/SimpleITKDoxygen/html/classes.html
Qt	https://doc.qt.io/qt-5.15/classes.html

Slicer-specific libraries

Toolkit	API documentation
Slicer core	C++: http://www.slicer.org/doc/html/classes.html Python: http://mwoehlke-kitware.github.io/Slicer/Base/slicer.html For up-to-date docs, type this into Python console: <code>help(slicer.util)</code>
CTK	http://www.commonstk.org/docs/html/classes.html



Where to find examples

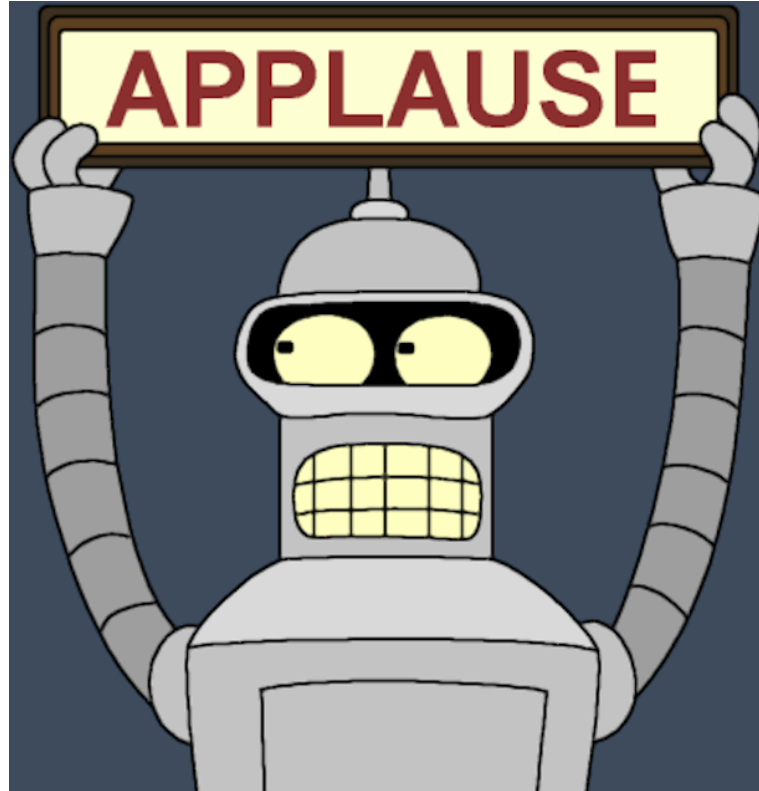
- Slicer core: <https://github.com/Slicer/Slicer>
- Extensions:
 - Index of extensions (see repository in *.s4ext):
<https://github.com/Slicer/ExtensionsIndex>
 - SlicerIGT:
[https://github.com/SlicerIGT/SlicerIGT/](https://github.com/SlicerIGT/SlicerIGT)
 - SlicerRT:
<https://github.com/SlicerRt/SlicerRT>



Hints

- To be able to show a model node, create display node:
`outputModel.CreateDefaultDisplayNodes()`
- To show model intersections with a 2D slice viewer:
`outputModel.GetDisplayNode().SetSliceIntersectionVisibility(True)`
- Remember that all the logic is implemented in the `run` function in the logic class, which is called in the `onApplyButton` function

Congratulations!



Thanks for attending!

Appendix

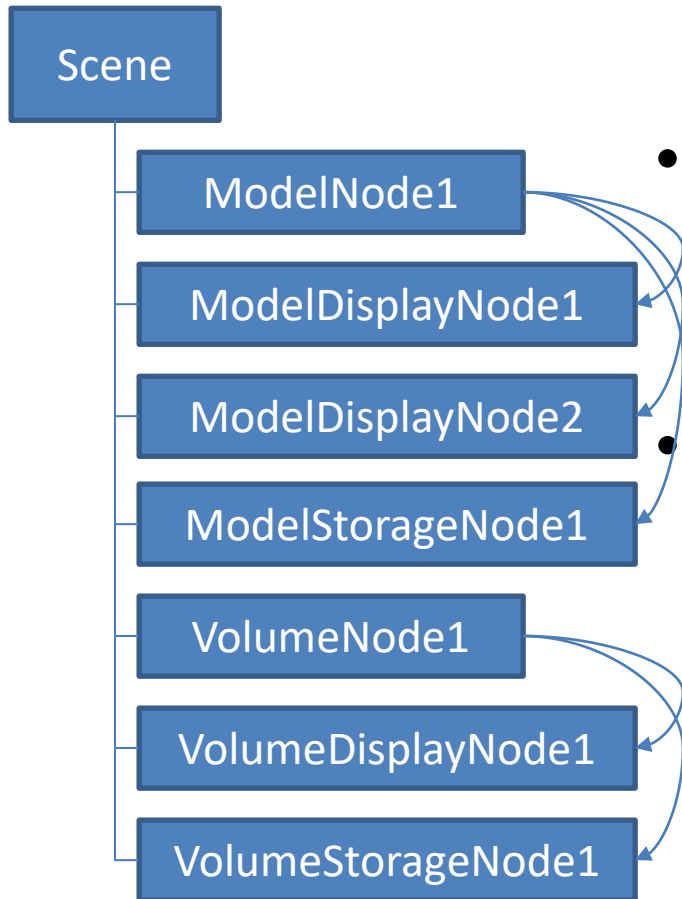


Transforms

- Slicer world coordinate system:
RAS (right-anterior-superior)
- Get linear transform from the node to RAS:

```
nodeToRas = vtk.vtkMatrix4x4()  
if node.GetTransformNodeID():  
    nodeToRasNode =  
        slicer.mrmlScene.GetNodeByID(node.GetTransformNodeID())  
    nodeToRasNode.GetMatrixTransformToWorld(nodeToRas)
```
- Transform may be non-linear
- At least log an error if transform is present but it is ignored

Node references



- Always use this whenever a node relies on data stored in other nodes
- Specified by role name, referenced node ID, index (multiple references with the same role is allowed)

- **Saved/restored with the scene**

Not trivial: When importing a scene and a node ID is already found in the current scene, the imported node ID is automatically renamed and all references are updated

Overview of API's accessible from python

