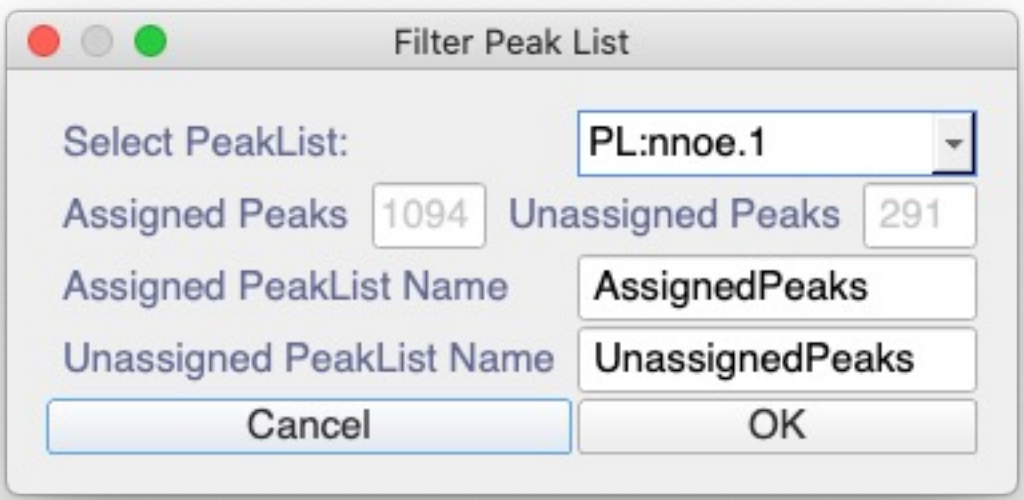
CCPN

CcpNmr Analysis Version 3

GUI Macro Writing Tutorial



Introduction

This tutorial is designed to introduce you to creating pop-ups when writing macros in CcpNmr Analysis 3.2. It begins by taking you through some basic concepts in graphical user interface (GUI) design and the CcpNmr Analysis data structure. This is followed by a series of worked examples and problems which gradually increase in complexity.

It is assumed that you have some basic familiarity with the CcpNmr Analysis program (e.g. from having completed our Beginners Tutorial) and some familiarity with writing macros in CcpNmr Analysis (e.g. from having completed our Macro Writing Tutorial). This also implies that you should have some basic knowledge of Python programming (e.g. indentations, conditionals and loops).

We recommend that you use the project provided in the **GUIMacroTutorial** data directory provided by CCPN while working through the examples and problems.
Please note that the images shown are only representative and you may encounter minor differences in your setup.
Please also note that over time we will make changes to the code base described here, but we will try to ensure that macros created with this code will remain functional.

Contents

Start CcpNmr Analysis V3

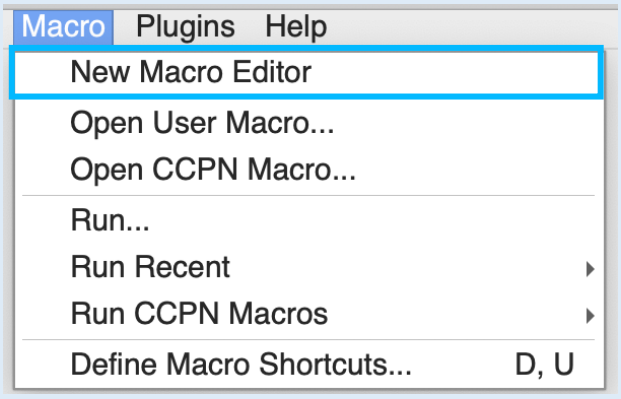
Apple users by double-clicking the *CcpNmrAnalysis* icon

Linux users by using the terminal command: *bin/assign*

Windows users by double-clicking on the *assign.bat* file

1 Introduction: the CCPN Macro Editor

When writing code, you can do so either in your own preferred code editor, or you can use the Macro Editor integrated within CcpNmr Analysis:

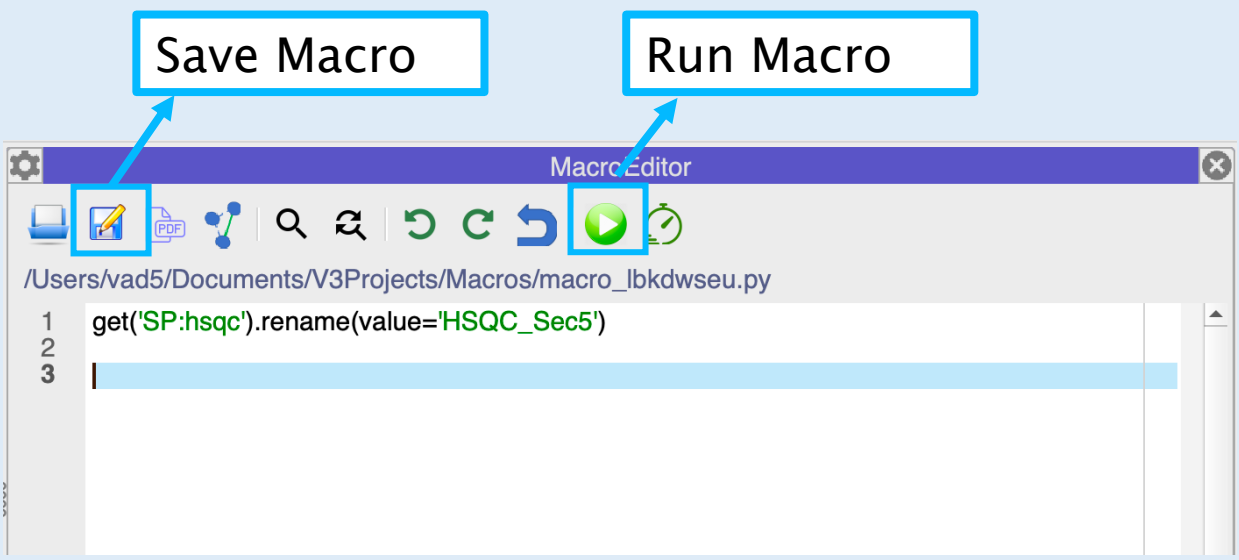


Open Macro editor:

- Go to **Main Menu → Macro → New Macro Editor**

OR

- use shortcut **NM**.

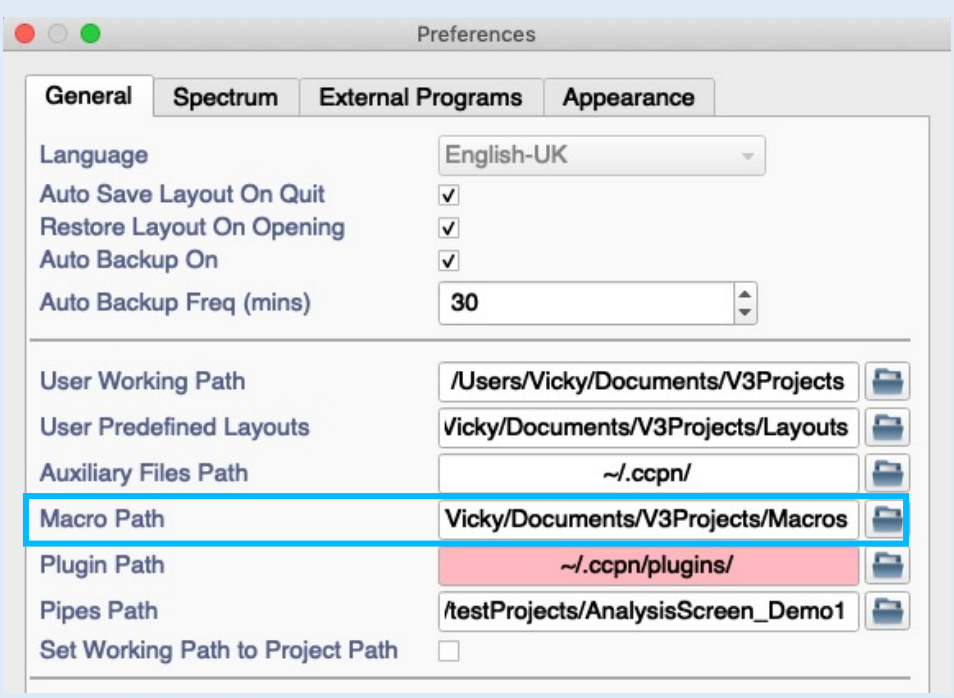


Set Macros Path:

- Go to **Main Menu → File → Preferences**

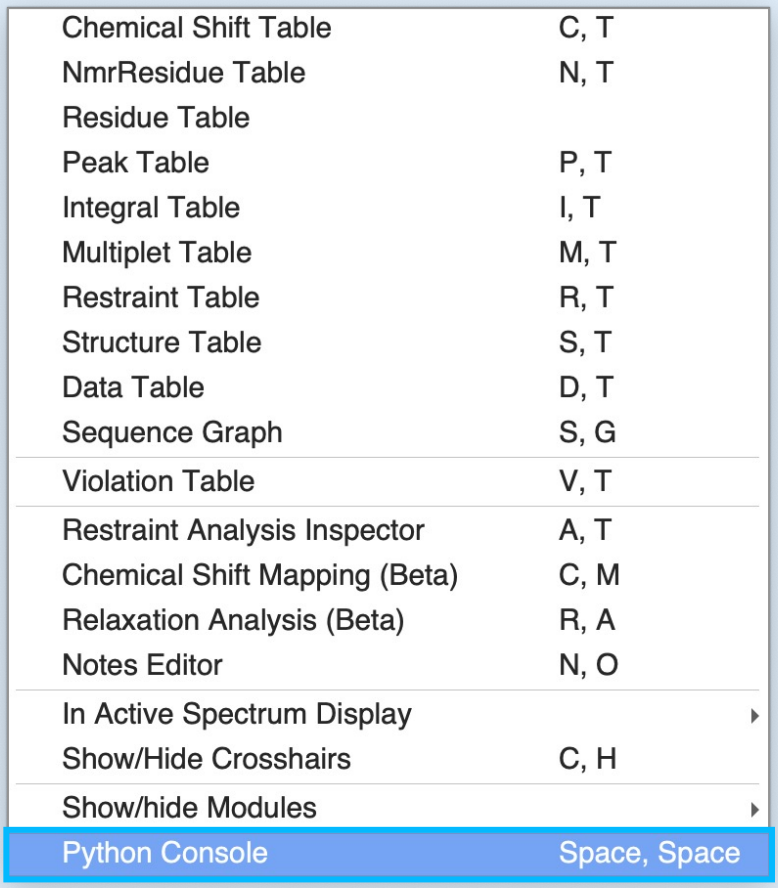
OR

- use shortcut **Ctrl/Cmd+,**



1 Introduction: the Python Console

The Python Console within CcpNmr Analysis can be useful for trying out bits of code, finding methods and properties or simply for running your macro:



Open Python Console:

- Go to **Main Menu → View → Python Console**

OR

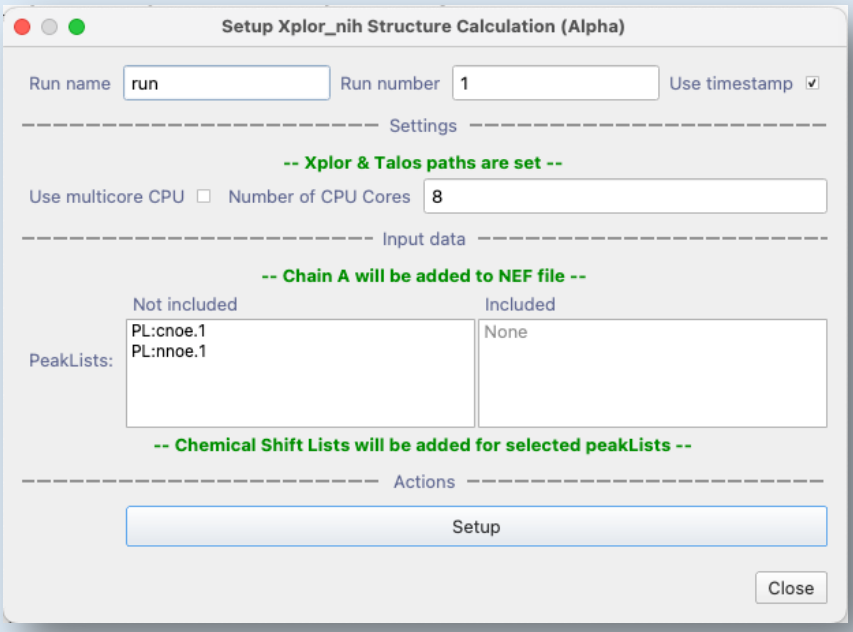
- use shortcut **Space, Space**.

Run Macro from Console:

- type

```
%run –i ~/my_path/myMacro.py
```

1 Introduction: Graphical User Interfaces (GUIs)



When writing a macro you might have various bits of information or data which you want a user to specify or select before running the macro. Rather than making the user edit the macro code, you can use graphical elements (technically a graphical user interface or GUI for short) to collect this information, e.g. popup windows with messages, buttons, drop-down menus, checkboxes etc.

Adding GUIs will typically make your macro more user-friendly and intuitive for others to use, especially those who are not familiar with using command-line tools or writing scripts.

The good thing about using graphical elements within CcpNmr Analysis is that we and others have already written lots of code to create all these graphical elements and you simply need to collate and arrange them as you would like.

2 Messages and Warnings

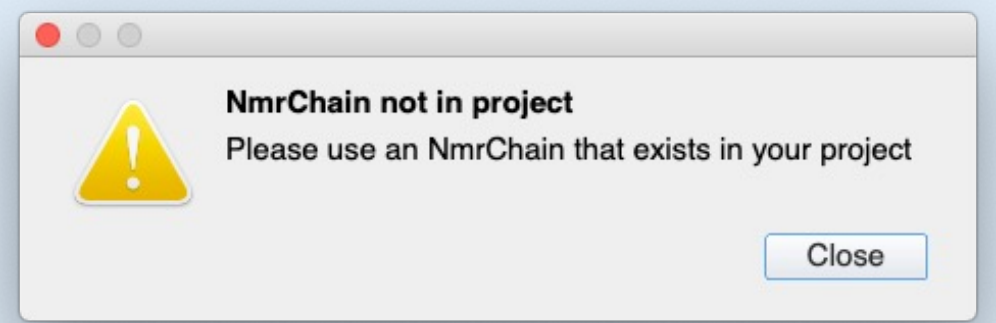
We will start this tutorial with a small basic macro that changes the name of an NmrChain (a little bit pointless, since you can do this in the NmrChain pop-up, but a nice easy example to get us thinking about GUI macros):

```
nc = get('NC:A')
newName = 'Z'
nc.rename(value= newName)
```

Over time, we will gradually add more and more graphical elements to this macro to make it more interactive.

This code will allow you to pop up an automatic warning message:

```
from ccpn.ui.gui.widgets.MessageDialog import Warning

showWarning('NmrChain not in project', 'Please use an NmrChain that exists in your project ')
```



2A Adding a Warning Message

- Open the **GUIMacros.ccpn** project provided in the tutorial data.
- Modify the basic macro provided above to pop up a warning message if the NmrChain that has been specified does not exist in the project. Remember that **project.** will allow you to access lots of different groups of objects in your project, such as all NmrChains.

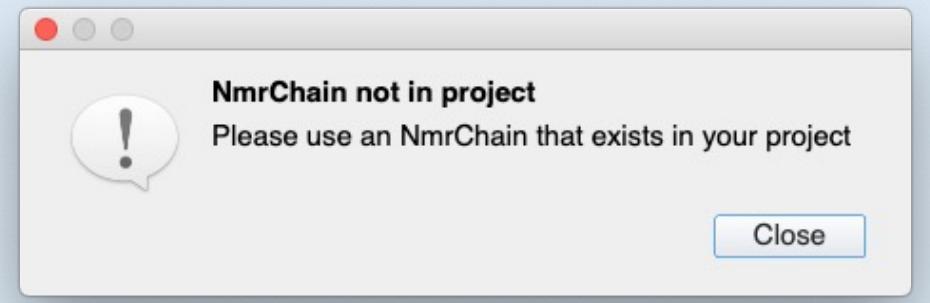
A possible version for this macro is provided in in the tutorial data as **2A_RenameNmrChainWithWarning.py**.

2 Messages and Warnings

Other message that you can show have different icons.

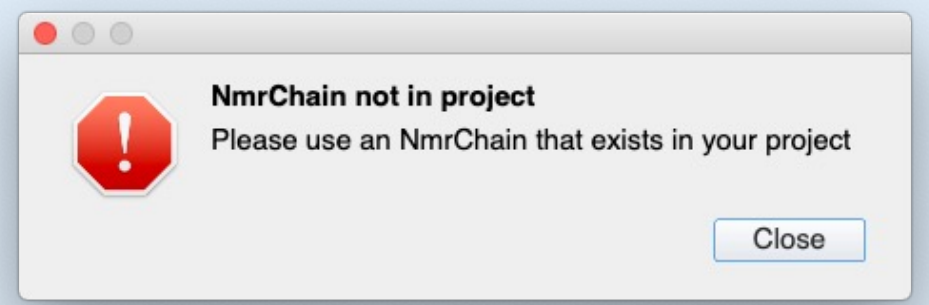
Here is a simple **Message Dialog**:

```
from ccpn.ui.gui.widgets.MessageDialog import showMessage

showMessage('NmrChain not in project', 'Please use an NmrChain that exists in your project ')
```



And here an **Error Dialog**:

```
from ccpn.ui.gui.widgets.MessageDialog import showError

showError('NmrChain not in project', 'Please use an NmrChain that exists in your project ')
```



2 Messages and Warnings

Other message dialogs will allow you to collect feedback from the user, for example you could provide some information and then ask whether they want to proceed:

```
from ccpn.ui.gui.widgets.MessageDialog import showYesNoWarning
yes = showYesNoWarning('Do you want to proceed?',
                       f'Are you sure you want to rename NmrChain {nc}?')
if yes:
   #DO SOMETHING
```

Other similar message dialogs include:

showYesNo

showOKCancel

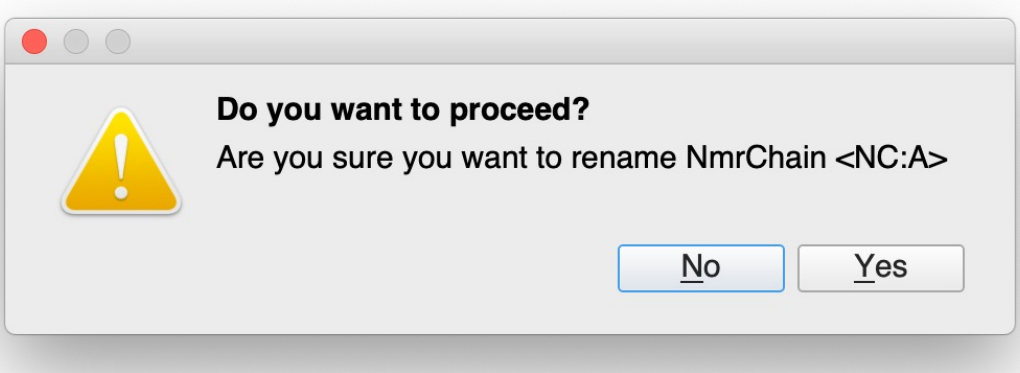
showOKCancelWarning

Remember that using too many popup dialogs and warnings can be annoying for users. So think about the best/most appropriate way to interact with users.

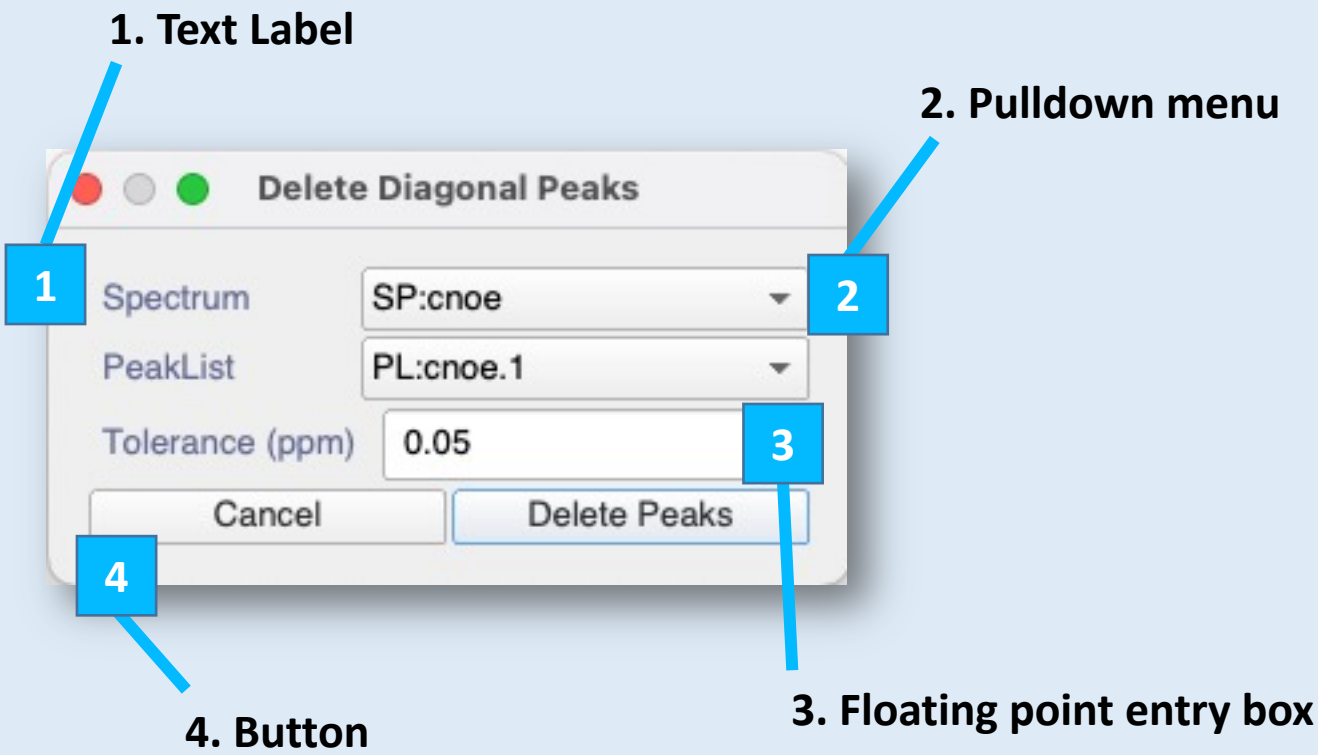
2B Asking for Confirmation

- Modify your macro from **Section 2A** to pop up a warning message asking if the user wants to proceed or not.

A possible solution is provided in **2B_RenameNmrChainWithYesNoWarning.py**.



3 Widgets



Most likely, you will want to start designing and building your own pop-ups rather than relying on the automated ones we supply. You can put these together from individual graphical building blocks (buttons, text boxes, checkboxes, drop-down menus, sliders etc. – see above) which are referred to as **widgets**.

Many of the CcpNmr Analysis widgets (and all of the ones we will be discussing in this tutorial) are based on the PyQt library. PyQt provides the basic components (such as buttons, menus and dialogs) and we then extend these to create our own CcpNmr Analysis specific widgets that can for instance show compound information such as structures, menus for Chemical Shift Lists, Chains, Spectra etc. In this tutorial we will show you how you can access these and incorporate them into your macros.

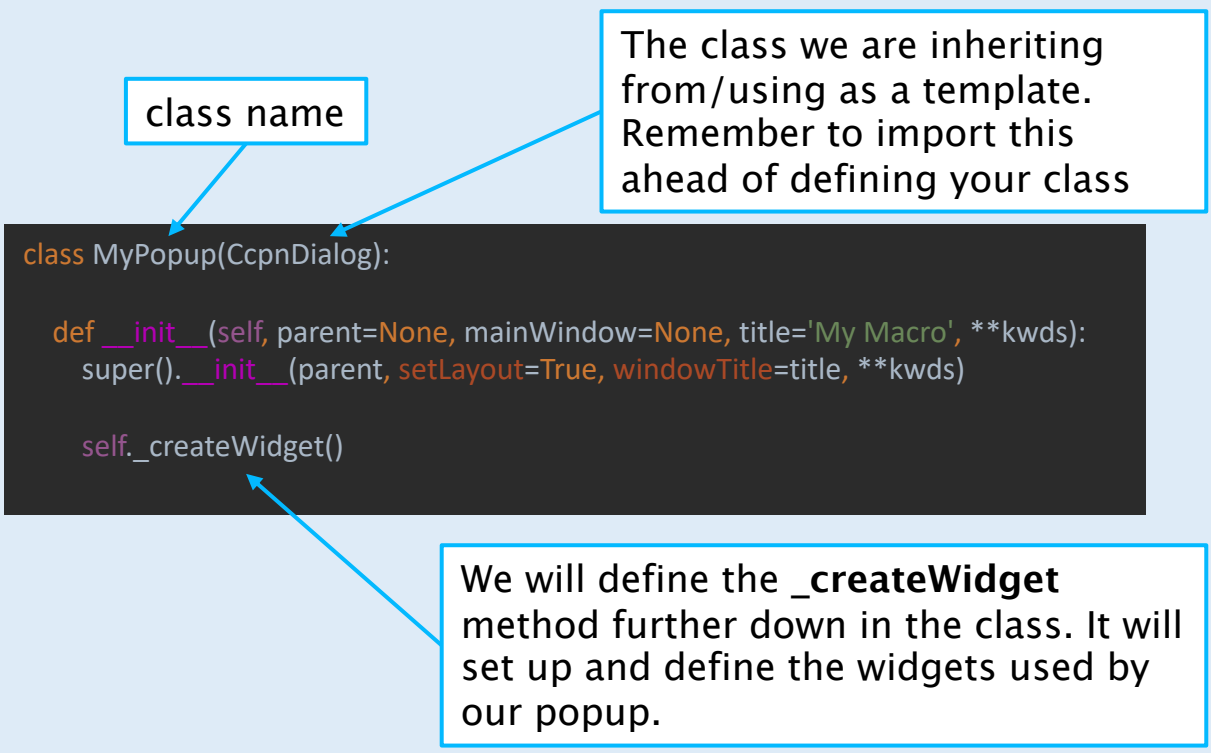
We recommend that you always use our widgets rather than working with PyQt directly, as we will always ensure that our widgets work with the latest version of PyQt, thus future proofing your macros.

3 Classes

Our first macro tutorial introduced you to some basic elements of object-oriented programming: **classes**, **objects**, **properties** and **methods**. In that context, we thought of classes as defining a particular type of data.

When using GUI elements in our code it will be useful to think about and use classes in a slightly different way. We will also start defining/writing our own classes. A class will now not define a particular data type, but be a means of organising our data and code. It will keep all the code related to our pop-up in a single container. It will also be a way to inherit functionality from other classes which effectively function as templates.

Here is an example of the basics of a class for a popup:



The first thing the class must do, is define the **__init__** function which initialises the class and should list all the necessary conditions for the class to be used.

4 Basic pop-ups

We are now going to change our NmrChain renaming macro to bring up a pop-up in which the user can select which NmrChain they want to rename. (This means our previous warning will no longer be needed!) The basic class from just now is extended with the following sections:

```
from ccpn.ui.gui.popups.Dialog import CcpnDialog
from ccpn.ui.gui.widgets.Button import Button
from ccpn.ui.gui.widgets.PulldownListsForObjects import NmrChainPulldown
from ccpn.core.lib.ContextManagers import undoBlock
from ccpn.ui.gui.widgets.MessageDialog import showWarning


class RenameNmrChain(CcpnDialog):
    title = 'Rename NmrChain'

    def __init__(self, parent=None, mainWindow=None, title=title,  **kwds):
        CcpnDialog.__init__(self, parent, setLayout=True, windowTitle=title, **kwds)

        self._createWidgets()

    def _createWidgets(self):
        self.NCPulldown = NmrChainPulldown(self,
                                   labelText='Select NmrChain:', grid = (0,0))
        Button(self, text='OK', grid=(1,0), callback=self._okCallback)

    def _okCallback(self, *args):
        if self.NCPulldown.getSelectedObject().name != '@-':
            with undoBlock():
                self._renameNmrChain(self.NCPulldown.getSelectedObject())
            return self.accept()
        else:
            showWarning('NmrChain @- cannot be renamed',
                                   'Please choose a different NmrChain to rename.')

    def _renameNmrChain(self, nc):
        newName = 'Z'
        nc.rename(value=newName)


if __name__ == "__main__":
    popup = RenameNmrChain(mainWindow=mainWindow)
    popup.show()
    popup.raise_()
```

Import all the classes used in the macro

Add an NmrChain Pulldown and button widget

Do something after the button was pressed

Method which exceutes the renaming of the NmrChain

Run the macro

4 Basic pop-ups

A few things to note:

```
def _createWidgets(self):
    self.NCPulldown = NmrChainPulldown(self,
                                       labelText='Select NmrChain:', grid = (0,0))
    Button(self, text='OK', grid=(1,0), callback=self._okCallback)
```

- The button is added directly with the **Button()** command because we do not need to use it later on in the macro.
- For widgets, such as the NmrChain pulldown menu, where we want to access a value from it at a later point, we need to create a new instance of that object with a *unique name*. In this case we add the NmrChain pulldown as **self.NCPulldown = NmrChainPulldown**. We can then access the value of this pulldown later in the macro using **self.NCPulldown.getSelectedObject()**.
- The **grid=(0,0)** and **grid=(1,0)** parameters simply place the widgets one below the other. You will find out more about the arrangement of widgets later on in **Section 7**.
- The **callback** parameter is used if you want to do something when the widget is activated. Here the **_okCallback** method will be executed. Note that the function does not include the brackets () at this point, otherwise it would be called immediately.

```
def _okCallback(self, *args):
    if self.NCPulldown.getSelectedObject().name != '@-':
        with undoBlock():
            self._renameNmrChain(self.NCPulldown.getSelectedObject())
        return self.accept()
    else:
        showWarning('NmrChain @- cannot be renamed',
                                       'Please choose a different NmrChain to rename.')
```

- The if/else statements prevent errors if someone tries to rename **NC:@-**.
- The actual content of the macro could be located within this **_okCallback** method, but for clarity it has been placed in a separate method called **_renameNmrChain**.
- By placing the **self._renameNmrChain()** method in an **undoBlock**, the macro can easily be undone.
- The **return self.accept()** command will close the pop-up once everything has been executed.

- All the method names (**_createWidgets** etc.) are preceded by a _ . This makes them local to this class.
- You will note the frequent use of **self**. This refers to things that are local to this class. It may feel confusing at first, but you will soon get the hang of it.

4 Basic pop-ups

Here are a few other widgets that you can use.

To add a Label or some text:

```
from ccpn.ui.gui.widgets.Label import Label
Label(self, text='A Text Label', grid=(0, 0), textColour='Blue', bold=False, italic=False)
```

To add a Text Entry box:

```
from ccpn.ui.gui.widgets import Entry
self.EntryBox = Entry.Entry(self, grid=(1, 0), editable=True, text='Input')
inputtedText = self.EntryBox.get()
```

4A Add a Text Entry box to enter the new NmrChain name

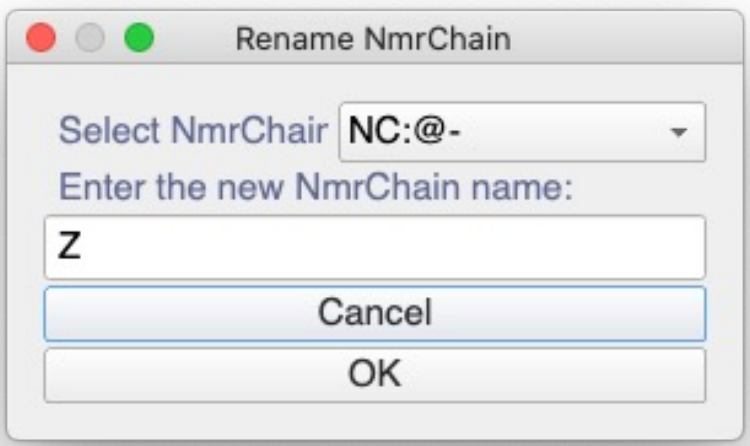
- Modify our basic pop-up macro (**4_RenameNmrChainBasicPopup.py** in the tutorial data) to include a Text Entry box where the user can enter the new NmrChain name.
- Try to add a Label which will ask the user to do this.
- Try experimenting with changing some of the Label or Entry Box parameters if you like.

Remember your imports and to pass the information entered by the user to the correct part of the macro to be used in the renaming.
A possible solution is shown in **4A_ RenameNmrChainEntryBox.py**

4B Add a Cancel button

- Now add a Cancel button above the OK button in case someone changes their mind about renaming the NmrChain. User **self.reject()** rather than **self.accept()** to close the popup.

A possible solution is shown in **4B_ RenameNmrChainCancelButton.py**

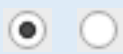


5 More Widgets

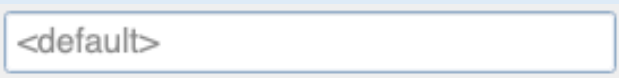
Here are some relatively generic widgets you might most commonly want to use.

Checkbox 

```
from ccpn.ui.gui.widgets import CheckBox
self.checkBox1 = CheckBox.CheckBox(self, grid=(1,0), checked=True)
if self.checkBox1.isChecked():
                    #DO SOMETHING
```

Radio Buttons 

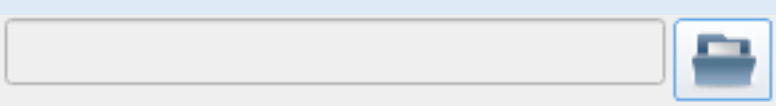
```
from ccpn.ui.gui.widgets.RadioButtons import RadioButtons
self.options = RadioButtons(self,  text=['button 1 txt', 'button 2 txt', 'button 3 txt'],
                                                direction='v',  grid=(2, 0))
self. option1, self.option2, self.option3 = self.options.radioButtons
if self.option1.isChecked():
                    #DO SOMETHING
```

Entry Boxes 

You have already encountered the generic entry box, **Entry.Entry**. This doesn't validate any of the data you provide.

There are additional specific entry boxes for integers (**Entry.IntEntry**), floating point numbers (**Entry.FloatEntry**) and arrays (**Entry.ArrayEntry**) which do validate the data.

```
from ccpn.ui.gui.widgets.Entry import IntEntry
self.EntryBox1 = IntEntry(self, grid=(1, 0), editable=True, text='Input')
inputtedNum = self.EntryBox1.get()
```

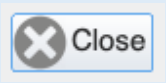
PathEdit 

This allows users to specify directories and files if you need to work with these.

```
from ccpn.ui.gui.lib.GuiPath import PathEdit
self.pathData1 = PathEdit(self, grid=(1, 0), vAlign='t', editable=True)
inputtedPath = self.pathData1.get()
```

5 More Widgets

Buttons



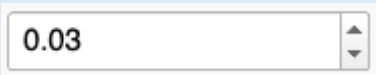
As well as having text-based buttons you can also have buttons with an icon:

```
from ccpn.ui.gui.widgets.Button import Button
button1 = Button(self, grid=(1, 0), callback=self._myCallback, text='Close',
                                    icon='.icons/close-icon.png', hPolicy='fixed')
```

You can either use your own icon, or one of the many we have in the /ccpnmr3.x/src/python/ccpn/ui/gui/widgets/icons folder.

The **hPolicy='fixed'** option will fit the button to your text.

Spin Box



These are an alternative way to allow users to enter numbers. You can specify the min, max and step size.

```
from ccpn.ui.gui.widgets.Spinbox import Spinbox
spinBox1 = Spinbox(self, grid=(1, 0), step=0.01, min=0.0, max=1.0)
inputtedNum = self.spinBox1.get()
```

Spacers

These will provide some extra space in your pop-up.

```
from ccpn.ui.gui.widgets.Spacer import Spacer
Spacer(self, width=0, height=10, grid=(1, 0))
```

You can also set the options **hPolicy** and **vPolicy** to **'minimum'**, **'minimumExpanding'** or **'fixed'**.

Horizontal Line

This will add a dividing line between sections of your pop-up.

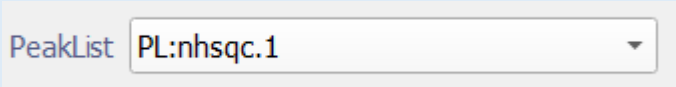
```
from ccpn.ui.gui.widgets.HLine import HLine
HLine(self, colour=getColours()[DIVIDER], height=20, grid=(3, 2))
```

The default **style** is **SolidLine**, but you can also set it to **DashLine**, **DashDotLine** or **DashDotDotLine**.

5 More Widgets

Now let's have a look at same useful CcpNmr Analysis specific widgets which enable you to work with objects like NmrChains, Spectra, PeakLists, Residues etc.

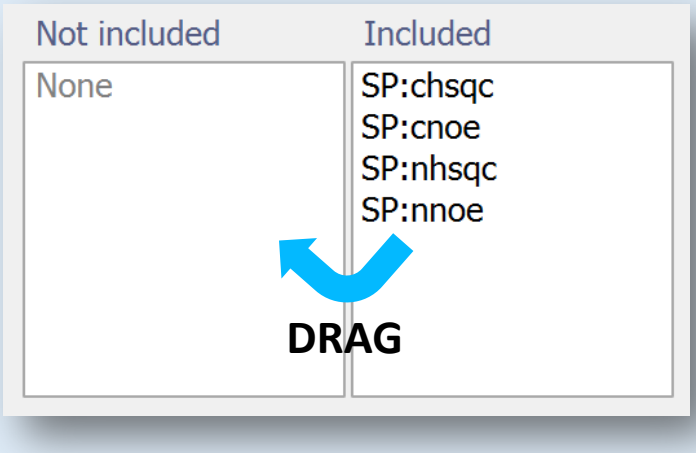
PullDownLists for Objects




```
from ccpn.ui.gui.widgets.PulldownListsForObjects import PeakListPulldown
self.peakListPullDown1 = PeakListPulldown(self, labelText='Target PeakList',
                                          filterFunction=self._filterPeakLists, grid=(0, 0))
selectedPeakList = self.peakListPullDown1.getSelectedObject()
```

You have encountered this for an NmrChain pulldown already. These pulldowns autofill with objects of a specific type and are available for all CCPN object types such as (Nmr)Chains, ChemicalShiftLists, PeakLists, NmrAtoms, Residues, PeakLists, Restraints etc. The filterFunction option lets you additionally filter your pulldown list, e.g. only to contain 3D peakLists or something like that.

ListWidgetPair




```
from ccpn.ui.gui.widgets.ListWidget import ListWidgetPair
self.spectraList = ListWidgetPair(self, grid=(0, 1), gridSpan=(1, 1))
self._populateSpectrumListFromProjectInfo()

def _populateSpectrumListFromProjectInfo(self):
    if self.project:
        self.spectraList._populate(self.spectraList.rightList, self.project.spectra)

listOfPids = self.spectraList.getRightList()
for pid in listOfPids:
    spectrum = self.project.getByPid(pid)
```

This creates a widget with two columns which we can be filled from a list of objects (in the example above from project.spectra). The objects or items can then be dragged between the columns to generate, for example, inclusion/exclusion lists. Unlike the PullDownLists for Objects this must be explicitly populated with the data of interest.

5 More Widgets

We have many more widgets available which more advanced programmers are welcome to explore.

For full details of widgets in CcpNmr Analysis see

https://ccpn.ac.uk/api-documentation/v3/html/ccpn/ccpn.ui.gui.widgets.html

Action
Application
Arrow
BalloonMetrics
BarGraph
BarGraphWidget
Base
BasePopup
Blank
Button
ButtonList
CalibrateXSpectrum1DWidget
CalibrateXSpectrumNDWidget
CalibrateYSpectrum1DWidget
CalibrateYSpectrumNDWidget
CallbackTimer
CcpnGridItem
CcpnArea
CcpnWebView
CheckBox
CheckBoxes
ColourDialog
Column
ColumnViewSettings
CompoundBaseWidget
CompoundView
CompoundWidgets
ConcentrationsWidget
Console
CustomExportDialog
DataFrameTableExample
DateTime
DialogButtonBox
Dock
DoubleSlider
DoubleSpinbox
DropBase
Entry
FileDialog
FillBetweenRegions
FilteringPulldownList
Font
Frame
GLAxis
GLLinearRegionsPlot
GLWidgets
GroupBox
GuiTable
HLine
HighlightBox
Icon
InputDialog
IpythonConsole
Label
LegendItem
LineEdit
LinearRegionsPlot
ListView
ListWidget
MainWindow
MathSymbol
Menu
MessageDialog
MoreLessFrame
NmrAtomsSelections
PhasingFrame
PipelineWidgets
PlaneToolbar
PlotWidget
PlotterWidget
PlotterWidgetUtils
ProjectTreeCheckBoxes
PulldownList
PulldownListsForObjects
PythonEditor
RadioButton
RadioButtons
RowExpander
ScatterPlotWidget
ScrollArea
ScrollBarVisibilityWatcher
SearchWidget
SequenceWidget
SettingsWidgets
SideBar
Slider
Spacer
SpectraSelectionWidget
SpectrumGroupToolBar
SpectrumToolBar
SpectrumWidget
SpeechBalloon
Spinbox
SplashScreen
Splitter
Stack1DWidget
Table
TableFilter
TableModel
TableSearch
TableSorting
Tabs
Test
TextEditor
TipOfTheDay
ToolBar
ToolButton
VLine
VerticalLabel
ViewBox
WebBrowser
WebView
Widget
resources_rc
testWidget

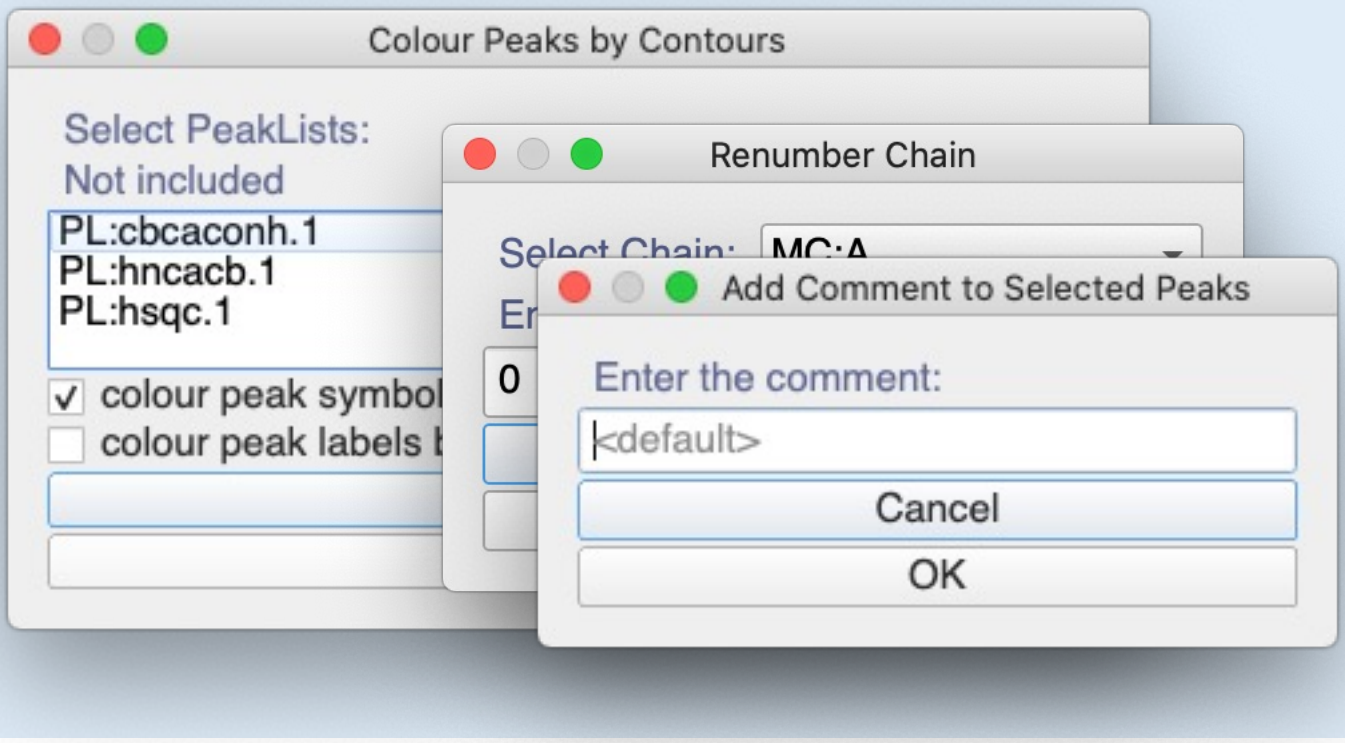
Please note that this list includes some internal widgets of little/no interest to others. In future versions we will separate these out.

5 More Widgets

You will have seen some of the options for our widgets. All of our widgets include the following options which you can explore and use if you wish:

tipText: add tiptext to widget
grid: insert widget at (row,col) of parent layout (if available)
hidden: hide widget upon creation
gridSpan: extend widget over (rows,cols); default (1,1)
stretch: stretch factor (row,col) of widget; default (0, 0)
hAlign: horizontal alignment: (l, left, r, right, c, center, centre)
vAlign: vertical alignment: (t, top, b, bottom, c, center, centre)
hPolicy: horizontal policy of widget: (fixed, minimum, maximum, preferred, expanding, minimumExpanding, ignored)
vPolicy: vertical policy of widget: (fixed, minimum, maximum, preferred, expanding, minimumExpanding, ignored)
minimumHeight: set minimum height
maximumHeight: set maximum height
fixedHeight: set fixed height
minimumWidth: set minimum width
maximumWidth: set maximum width
fixedWidth: set fixed width
bgColor: background RGB colour tuple; depreciated: use styleSheet routines instead
fgColor: foreground RGB colour tuple; depreciated: use styleSheet routines instead
isFloatWidget: indicates widget to be floating

5 More Widgets



5A Add comment to selected peaks macro

- Create a macro in which the user can write a comment that will be applied to all currently selected peaks.

A possible solution is shown in **5A_ addCommentToSelectedPeaks.py**

5B Renumber Chain macro

- Create a macro in which the user can choose a Chain which they wish to renumber as well as specifying the offset by which they wish to renumber it (remember: it only makes sense to set the offset to an integer number!).

A possible solution is shown in **5B_ renumberChain.py**

5C Renumber Chain and NmrChain macro

- Modify your macro in 5B above, so that if there is a matching NmrChain, the user is asked if they want to renumber that, too.

A possible solution is shown in **5C_ renumberChainNmrChain.py**

5D Colour peaks by contours macro

- Create a macro in which the user can choose a number of peakLists for which they want to colour the peak symbols and/or labels in the positive contour colour of the spectrum.

A possible solution is shown in **5D_ colourPeaksByContours.py**

6 Setting widget data

You've seen how we can get a value back from a widget using get() or getSelectedObject(). All widgets have similar in-built functions to set their specified values (e.g. to initialise the GUI for the user). For most widgets setting the value is as simple as calling set() of the instance.

Checkbox.set() will return a Boolean True/False depending on whether it is checked or not

```
self.checkBox1.set(True)
```

Entry.set() will set the value of the input box, e.g. if you use a FloatEntry (the floating point number entry box) then you can set the value to an appropriate float.

```
self.entryBox1.set('Please enter some text')
```

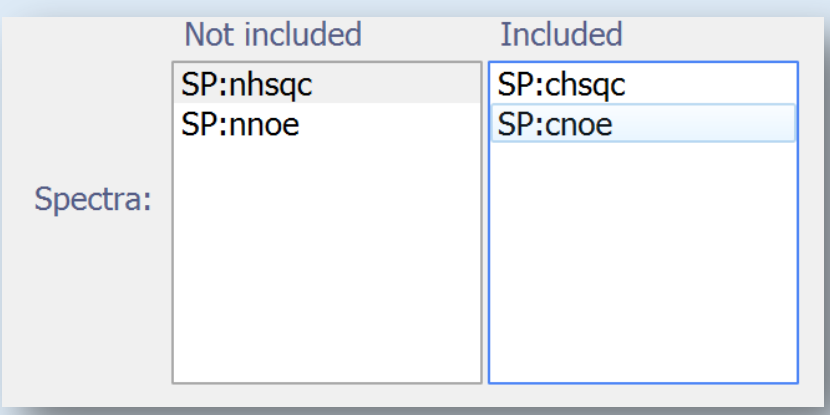
PullDownLists.set() will select the desired item.

```
object = self.project.spectra[0]
self.objectPulldown.set (object)
```

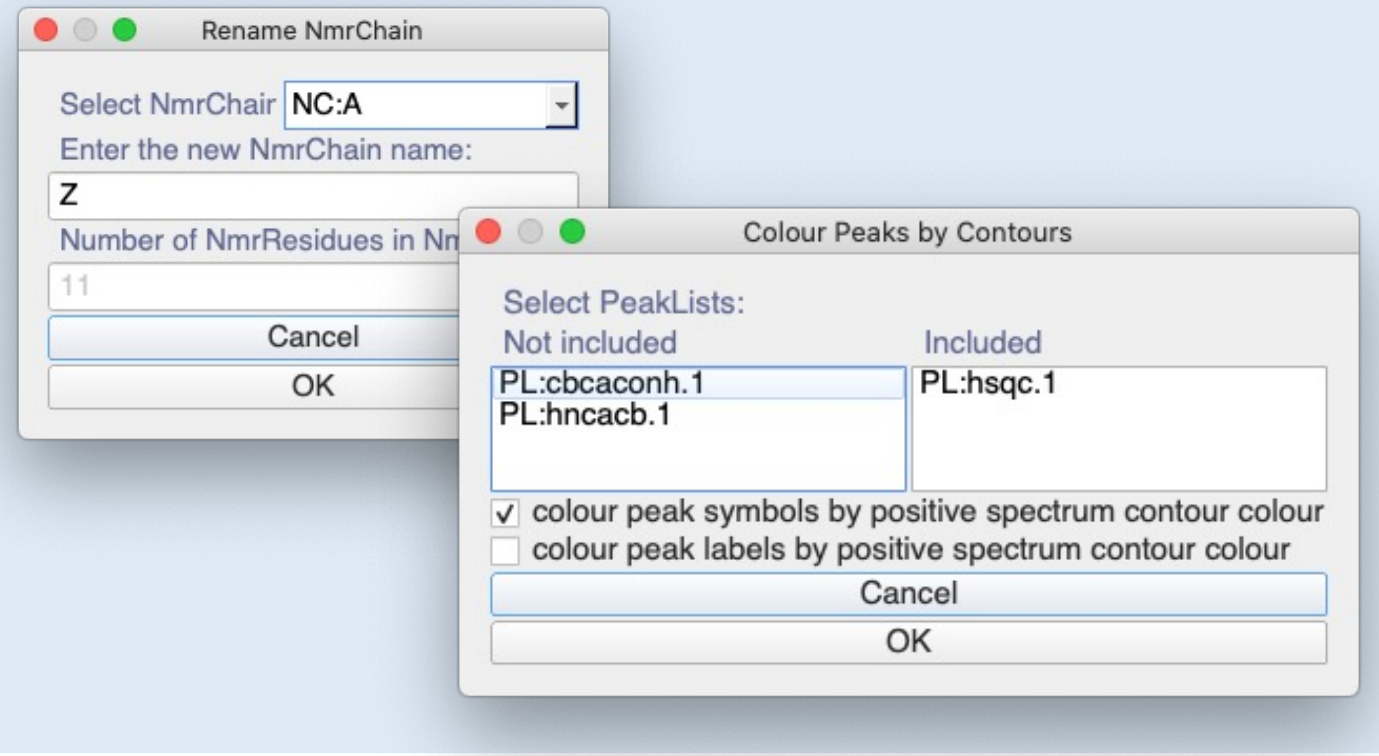
ListPair widgets

A special case is the ListPair widgets where you can select whether to set the values into the left list (not included) or the right list (included) separately. You can do this using the _populate function which is specifically for CCPN Objects with a valid PID.

```
self.spectraList._populate(self.spectraWidget.rightList, list1)
self.spectraList._populate(self.spectraWidget.leftList, list2)
```



6 Setting widget data



6A Add a Callback to say how many NmrResidues are in the NmrChain

- Modify the latest version of the NmrChain renaming macro (**4B_ RenameNmrChainCancelButton.py** in the tutorial data) to include a Callback to tell you how many NmrResidues are in the NmrChain that has been selected in the drop-down menu.

A possible solution is shown in **6A_ RenameNmrChainCallback.py**

6B Pre-select 2D peakLists

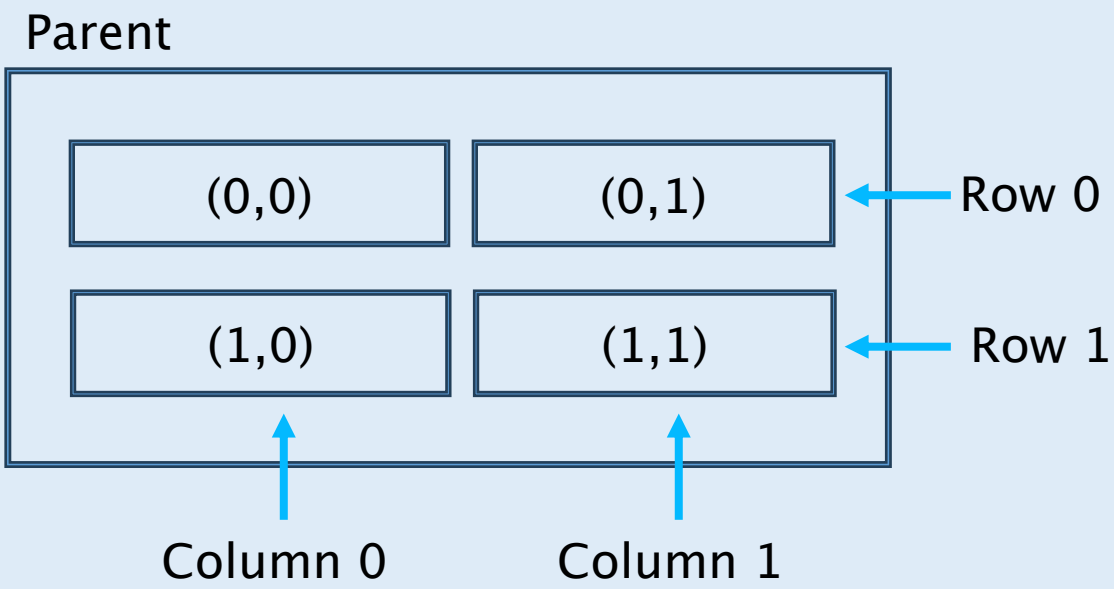
- Modify the Peak Colouring macro (**5A_ colourPeaksByContours.py** in the tutorial data) to pre-select only peak lists of 2D spectra. (Hint: you can get the dimensionality of a spectrum with spectrum.dimensionCount.)

A possible solution is shown in **6B_ colour2DPeaksByContours.py**

7 Widget layout

It is finally time to think a bit more about the layout of our popup.

Widgets are aligned with respect to one another in a grid consisting of a parent frame, rows and columns.

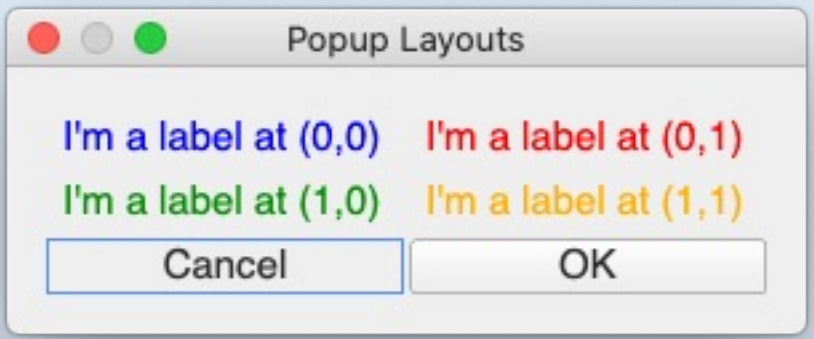


The default parent is **self**, the window of the popup.

When we define a widget, we define its parent and then its grid position (row, column) within the parent.
NOTE: Grid positions follow the Pythonic standard and start at 0!

```
# This create a 2x2 grid of elements
Label(parent=self, text="I'm a label at (0,0)", grid=(0, 0), textColour='Blue')
Label(parent=self, text="I'm a label at (0,1)", grid=(0, 1), textColour='Red')
Label(parent=self, text="I'm a label at (1,0)", grid=(1, 0), textColour='Green')
Label(parent=self, text="I'm a label at (1,1)", grid=(1, 1), textColour='Orange')
```

As you will have noticed by comparing this code the previous code, **parent=self** can also be replaced simply with **self**.



7 Widget layout

The number of columns in a parent and the number of rows is defined as the maximum of the grid positions of the elements within the grid. The layout automatically adjusts to this structure.

For example, the following code defines a 4 row by 3 column grid:

```
# This create a 4x3 grid of elements
Label(parent=self, text="I'm a label at (0,0)", grid=(0, 0), textColour='Blue')
Label(parent=self, text="I'm a label at (1,2)", grid=(1, 2), textColour='Red')
Label(parent=self, text="I'm a label at (2,1)", grid=(2, 1), textColour='Green')
Label(parent=self, text="I'm a label at (3,2)", grid=(3, 2), textColour='Orange')
```

The Maximum row value is **3**, and the maximum column value is **2**. Resulting in the **4x3 layout** below (remember we start from 0).

Note: The width and height of the columns adjust to the dimensions of the contents.

Warning: It is possible to give elements the same grid position which will result in confusing, overlapped layouts.

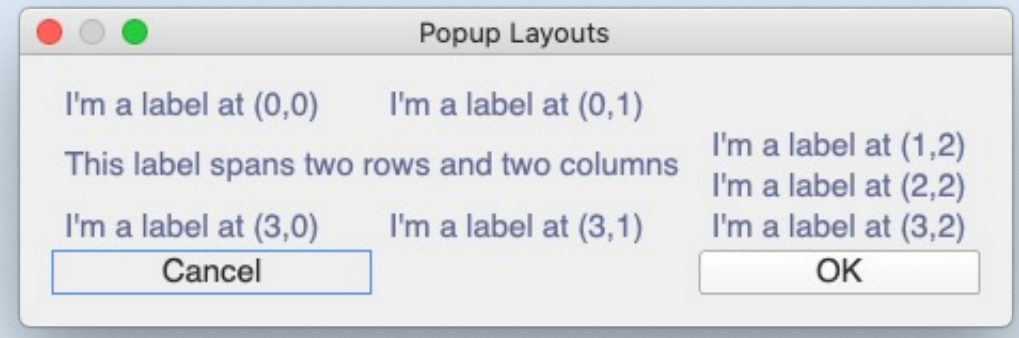
Widget layout

A useful way to ensure your next set of widgets are added to the next row in a grid is to use a variable for the row number which you increment:

```
row = 0
Label(parent=self, text="I'm a label at (0,0)", grid=(row, 0))
Label(parent=self, text="I'm a label at (0,1)", grid=(row, 1))

row += 1
Label(parent=self, text="I'm a label at (1,0)", grid=(row, 0))
Label(parent=self, text="I'm a label at (1,1)", grid=(row, 1))

row += 1
Label(parent=self, text="I'm a label at (2,0)", grid=(row, 0))
Label(parent=self, text="I'm a label at (2,1)", grid=(row, 1))
```

Sometimes you may want a widget to span several rows or columns. You can do this using the **gridSpan = (x, y)** parameter where x is the number rows and y the number of column across which to spread the widget:

```
row = 0
Label(parent=self, text="I'm a label at (0,0)", grid=(row, 0))
Label(parent=self, text="I'm a label at (0,1)", grid=(row, 1))

row += 1
Label(parent=self, text="This label spans two rows and two columns", grid=(row, 0), gridSpan=(2, 2))
Label(parent=self, text="I'm a label at (1,2)", grid=(row, 2))

row += 1
Label(parent=self, text="I'm a label at (2,2)", grid=(row, 2))

row += 1
Label(parent=self, text="I'm a label at (3,0)", grid=(row, 0))
Label(parent=self, text="I'm a label at (3,1)", grid=(row, 1))
Label(parent=self, text="I'm a label at (3,2)", grid=(row, 2))
```



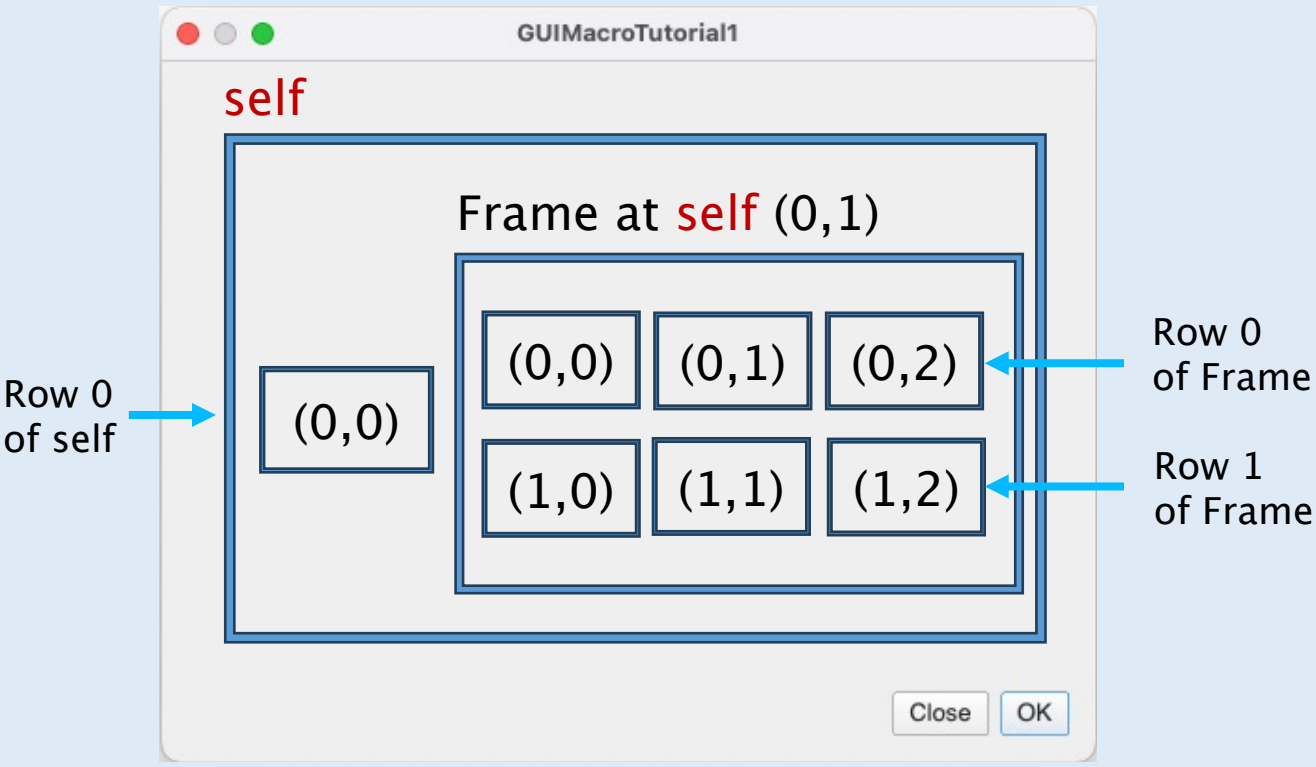
7 Widget layout

As you proceed in your coding, you may wish to have even greater control over the placement of widgets. To enable this, pyQT provides us with the concept of a Frame.

A Frame is a container which we can use as a parent in our layouts.

The Frame is itself an widget like a Checkbox or an Entrybox. As such, its position is defined with respect to a parent (normally **self**).

Elements within the Frame are defined by a Grid which operates inside the Frame.

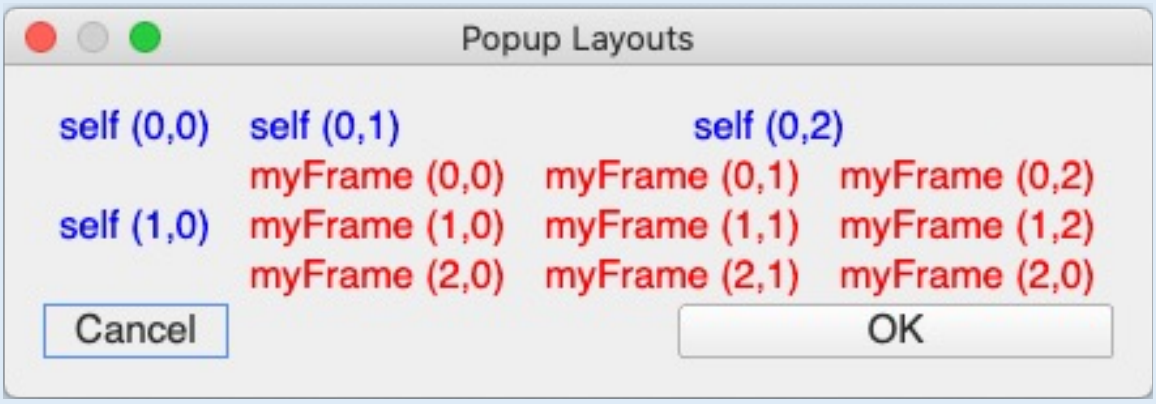


In more complicated layouts you may wish the Frame to span across multiple rows and columns of the **Parent**. To do this, you can use the **gridSpan** parameter when you define the Frame.

7 Widget layout

The following code demonstrates how **frames** can be used when laying out widgets (in this case simple text Labels).

```
# Using Frames
from ccpn.ui.gui.widgets.Frame import Frame

# The parent of these Labels is self
Label(parent=self, text="self (0,0)", grid=(0, 0), textColour='Blue')
Label(parent=self, text="self (0,1)", grid=(0, 1), textColour='Blue')
Label(parent=self, text="self (0,2)", grid=(0, 2), textColour='Blue')
Label(parent=self, text="self (1,0)", grid=(1, 0), textColour='Blue')

# Define the Frame (myFrame). This frame starts at (1,1) and spans 1 row
# and 2 columns relative to self
myFrame = Frame(self, grid=(1, 1), gridSpan=(1, 2), setLayout=True)

# Place the elements in the grid inside the Frame (myFrame)
Label(parent=myFrame, text="myFrame (0,0)", grid=(0, 0), textColour='Red')
Label(parent=myFrame, text="myFrame (0,1)", grid=(0, 1), textColour='Red')
Label(parent=myFrame, text="myFrame (0,2)", grid=(0, 2), textColour='Red')
Label(parent=myFrame, text="myFrame (1,0)", grid=(1, 0), textColour='Red')
Label(parent=myFrame, text="myFrame (1,1)", grid=(1, 1), textColour='Red')
Label(parent=myFrame, text="myFrame (1,2)", grid=(1, 2), textColour='Red')
Label(parent=myFrame, text="myFrame (2,0)", grid=(2, 0), textColour='Red')
Label(parent=myFrame, text="myFrame (2,1)", grid=(2, 1), textColour='Red')
Label(parent=myFrame, text="myFrame (2,0)", grid=(2, 2), textColour='Red')
```

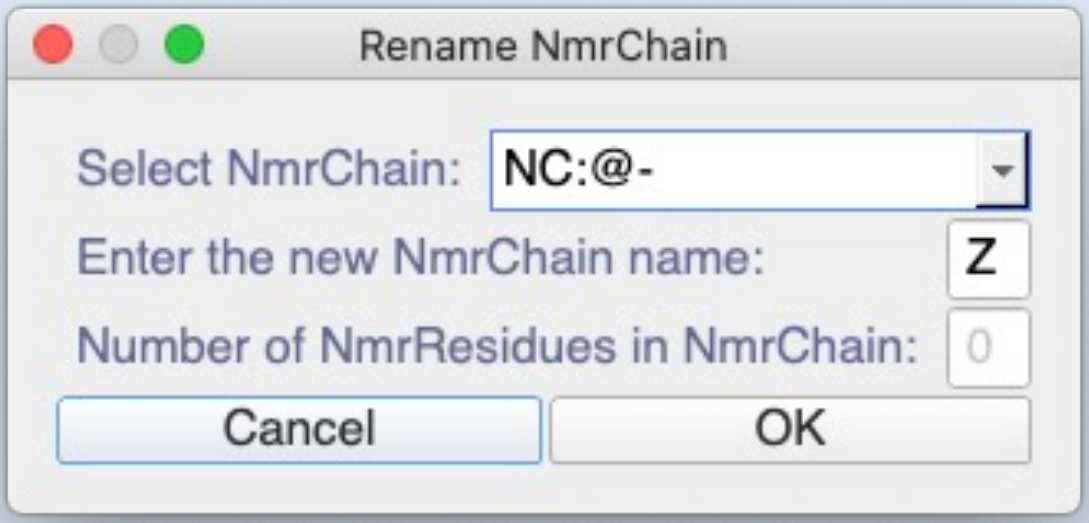


Note that you need an additional import statementat at the top of your code!

When defining the Frame, the **setLayout=True** parameter allows you to create an independent layout within the frame. The **gridSpan=(1,2)** parameter specifies that the 3 rows and 3 columns of **myFrame** should span across 1 row and 2 columns of **self**.

You can define multiple frames if you like. Some developers like to use frames to group related widgets rather than simply as a layout tool.

7 Widget layout

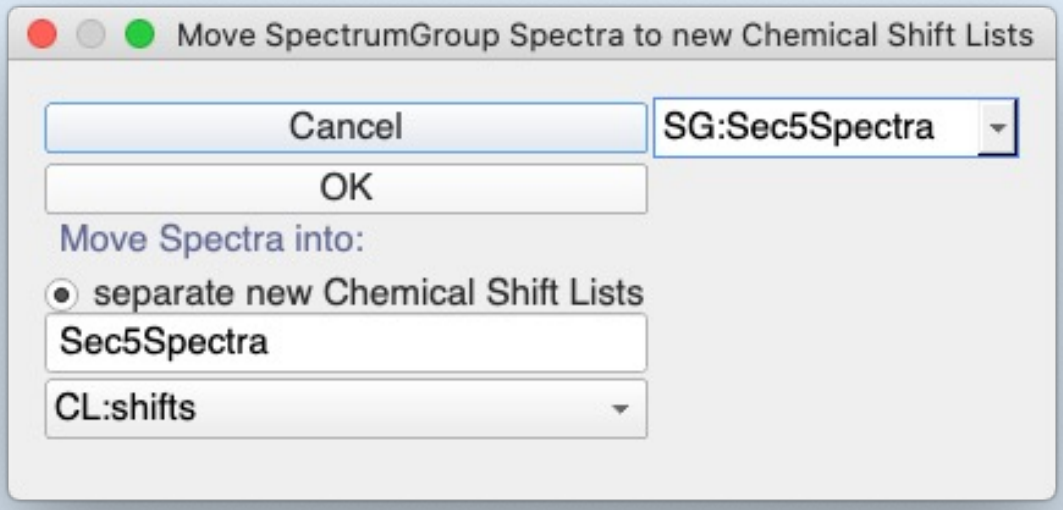


7A **Improve the RenameNmrChain macro layout**

- Modify your latest NmrChain renaming macro (**6A_RenameNmrChainCallback.py** in the tutorial data) to have the layout shown above.
- Try an alternative layout if you wish.

A possible solution is shown in **7A_ RenameNmrChainNewLayout.py**

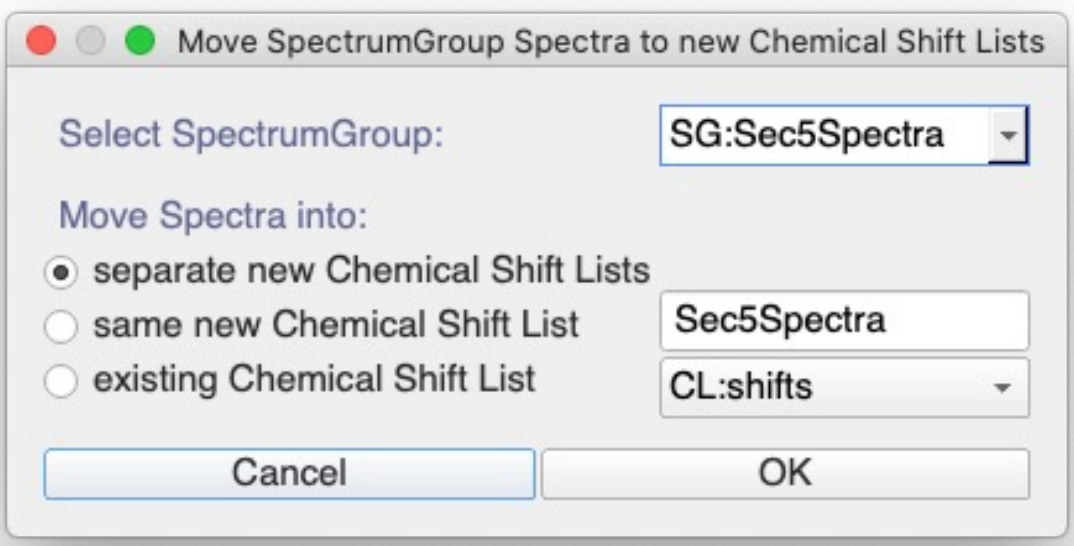
Widget layout



7B Fix a layout

- Run the **7B_moveSGSpectraIntoNewCLs.py** macro. You will see that the layout has got all muddled up.
- Try fixing the layout so that it looks like the one below.

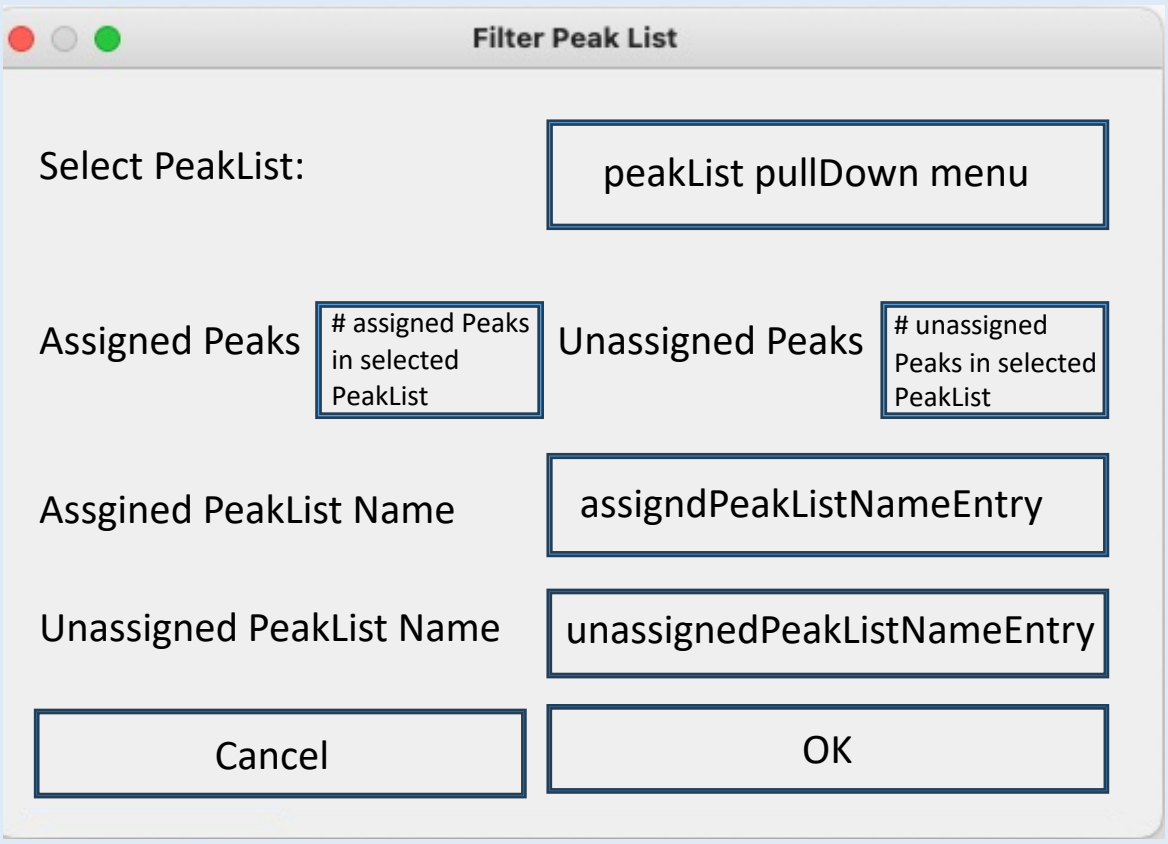
A possible solution is shown in **7B_ moveSGSpectraIntoNewCLs_fixed.py**



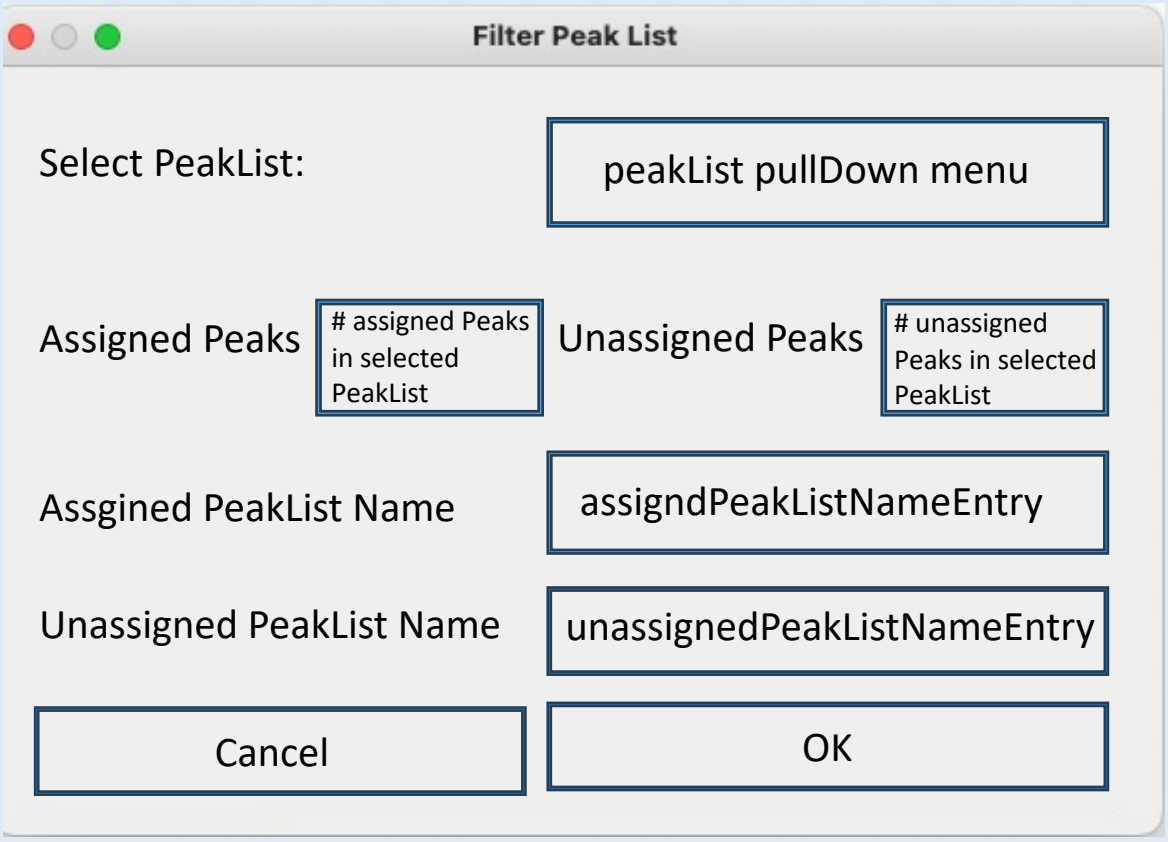
8 Advanced GUI macro writing

In this part of the tutorial we are going to build a slightly more advanced macro. Its function is to take a (NOESY) peak list and split it into two separate new peak lists: one for all the assigned peaks and another for all the unassigned peaks.

Here is a rough layout of what we want our **FilterPeakList** popup to look like:



8 Advanced GUI macro writing



8A Define the class and __init__

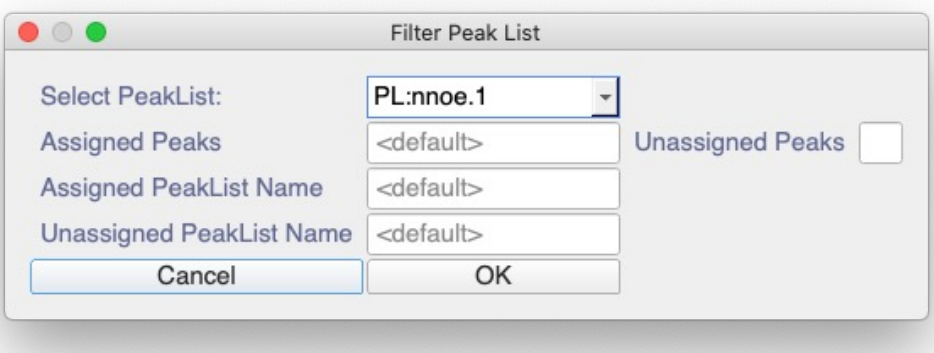
- As with your previous macros, start by defining your class and the **__init__** function.
  Remember any imports needed.
- Add the section to run the macro, so that you will be able to try the macro out easily as you start to put it together.

A possible solution is shown in **8A_ FilterPeakList.py**

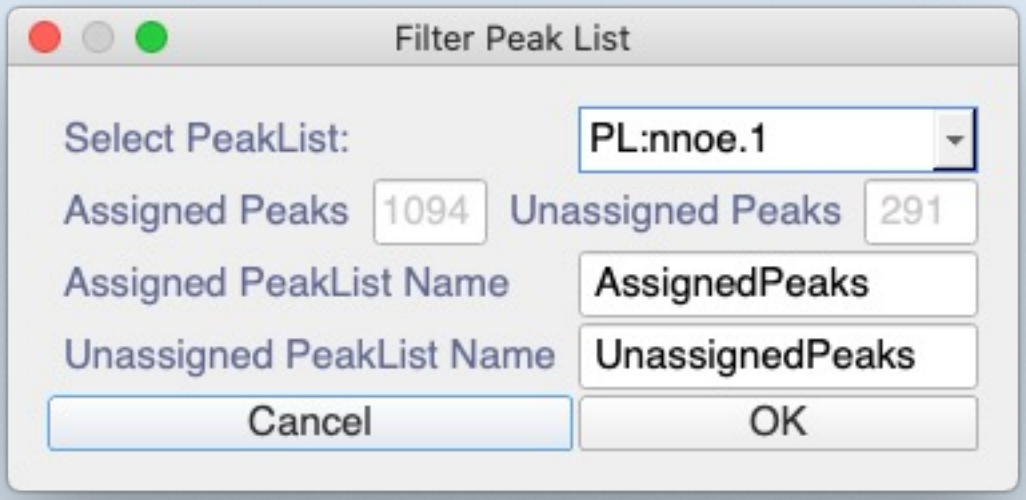
8B Create the widgets

- Starting with your work from 8A (or the **8A_FilterPeakList.py** file provided), add all the widgets required. Use the layout above to work out what widgets you will need.
- Add a Callback to your Cancel button to enable easier testing.

A possible solution is shown in **8B_ FilterPeakList.py**



8 Advanced GUI macro writing



8C Improve layout

- Starting with your work from 8B (or the **8B_ FilterPeakList.py** file provided), improve the layout, so that the widgets on the second row are sized independently of the other rows

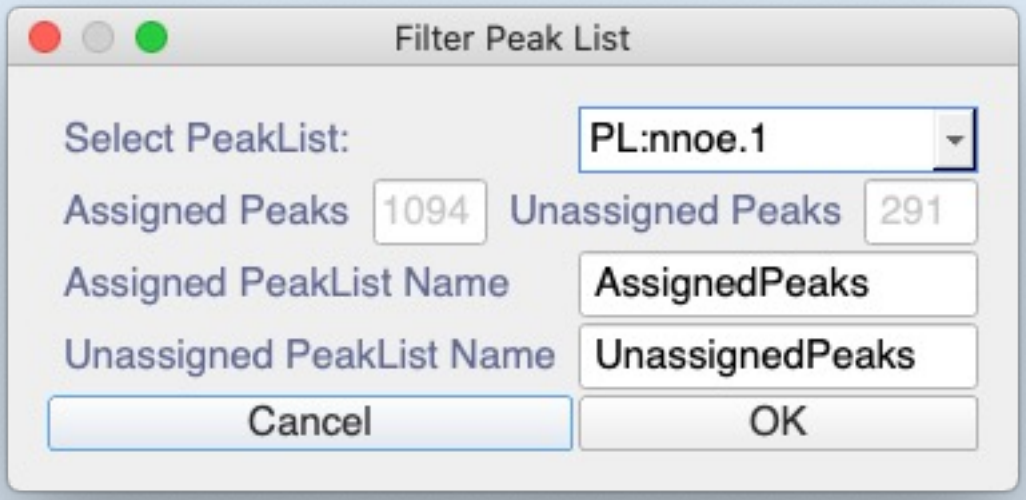
A possible solution is shown in **8C_ FilterPeakList.py**

8D Add the peak counting callback

- Starting with your work from 8C (or the **8C_ FilterPeakList.py** file provided), add a callback which will enter the number of assigned or unassigned peaks into the entry boxes in the second row.

A possible solution is shown in **8D_ FilterPeakList.py**

Advanced GUI macro writing



8E Create the PeakList filtering routine

- Starting with your work from 8D (or the **8D_ FilterPeakList.py** file provided), add the main routine to create the two new PeakLists and link this to a callback from the OK button.

  Note that since peakLists are generally identified by their index number rather than a name, you will need to place the PeakList Name provided by the user in the PeakList **comment**. This will then be visible when double-clicking on a PeakList in the sidebar.
- If you wish, add default names for the two peakLists.

A possible solution is shown in **8E_ FilterPeakList.py**

9 Best Practice

When designing a GUI popup try to keep things are simple as possible:

1. Minimise User Input: Focus on essential functionality and avoid overwhelming users with too many options or features. Reduce the amount of user input required whenever possible. Pre-fill or auto-populate fields with default values if appropriate. Avoid unnecessary confirmation prompts or complex input requirements.

2. Clear and Concise Layout: Keep the layout of your GUI clean and uncluttered. Avoid overcrowding by leaving enough white space between elements. Group related elements together logically.

3. Intuitive Navigation: Design a navigation structure that is intuitive and easy to follow. Provide clear visual cues, such as buttons or menus, to guide users through different sections or tasks in the application.

4. Clear Feedback and Error Handling: Provide clear and immediate feedback to users after their interactions. Display informative messages or visual indicators to confirm actions or inform users about any errors that occur.

Remember, simplicity in GUI design promotes ease of use, reduces cognitive load, and enhances the overall user experience. By focusing on clarity, intuitive navigation, and limited complexity, you can create a GUI that is user-friendly and enjoyable to interact with.

Contact Us

Website:
www.ccpn.ac.uk

Suggestions and comments:
support@ccpn.ac.uk

Issues and bug reports:
https://forum.ccpn.ac.uk/

Cite Us

Skinner, S. P. et al. CcpNmr AnalysisAssign: a flexible platform for integrated NMR analysis. J. Biomol. NMR 66, (2016)