

# "it" Custom Commands

Imagine that you want to give your artists some function in "it" that they are going to perform many times, over and over again, repeatedly, but they're not keen on using typed commands and can't remember them anyway, what with being so busy making their art. Luckily for you, "it" has a customizable **Commands** menu that can be used for just this situation.

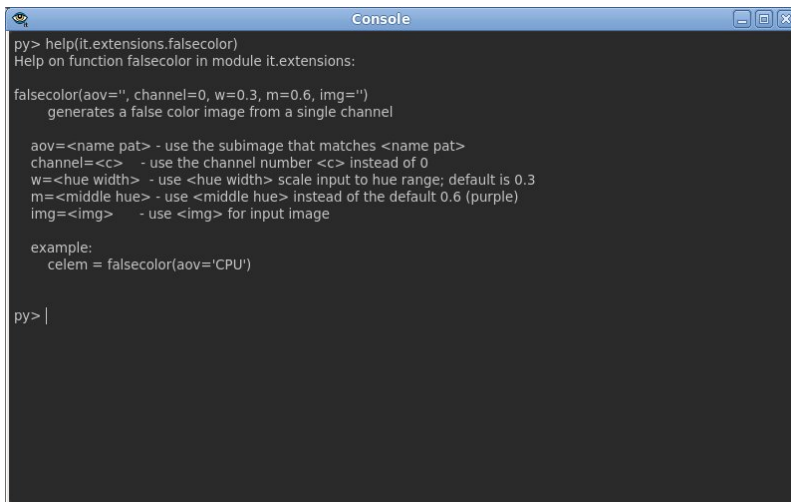
This tutorial is going to build on the `ice.extensions.falsecolor` command available in the "it" console, providing a simple UI that makes it easier to control.

## Background

If you are not familiar with existing "it" console command `ice.extensions.falsecolor`, bring up the "it" console window and print out its usage by typing:

```
help(it.extensions.falsecolor)
```

You should see something like this:



```
py> help(it.extensions.falsecolor)
Help on function falsecolor in module it.extensions:

falsecolor(aov="", channel=0, w=0.3, m=0.6, img="")
    generates a false color image from a single channel

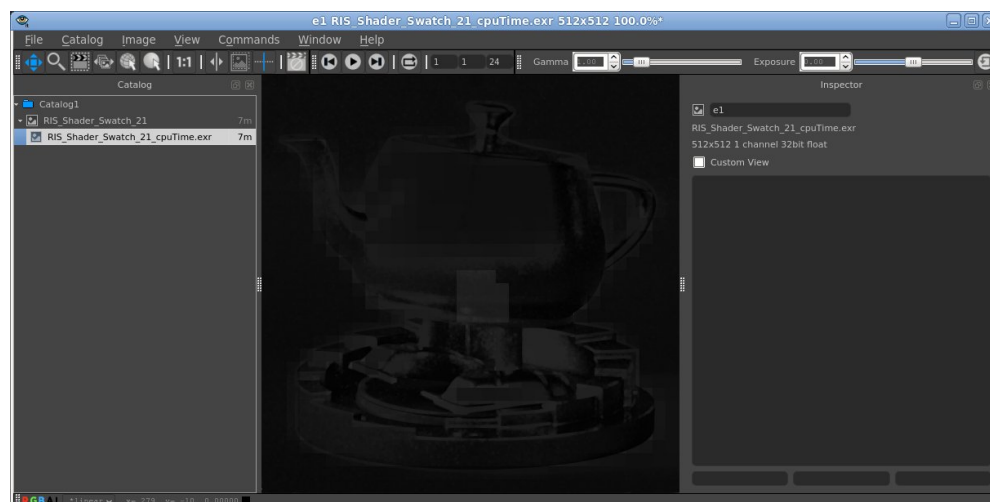
    aov=<name pat> - use the subimage that matches <name pat>
    channel=<c>    - use the channel number <c> instead of 0
    w=<hue width>  - use <hue width> scale input to hue range; default is 0.3
    m=<middle hue> - use <middle hue> instead of the default 0.6 (purple)
    img=<img>      - use <img> for input image

example:
    celem = falsecolor(aov='CPU')
```

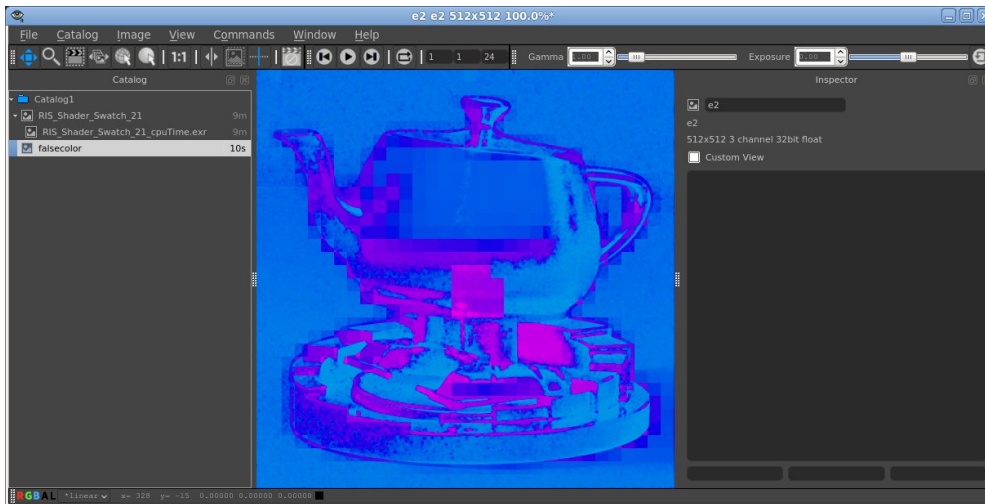
`falsecolor` can operate on any image; open an image and try running it by typing in the console:

```
it.extensions.falsecolor()
```

This might not be terribly useful, but there was a very specific situation that `falsecolor` was created for: to make sense of the `cpuTime` AOV. If you view the AOV in "it", you'll see something like this:



Applying the `falsecolor` command will give us something much more visual:



blue is lower CPU time values and red is higher.

## Create a Command Extension File

The "it" commands are stored in files that are loaded by the usual LoadExtension command from an .ini file. As always, we discourage editing the factory shipped .ini files and, instead, recommend setting the environment variable RMS\_SCRIPT\_PATHS to point to directories where you put your own, site-specific .ini files.

So, quit "it", make sure the aforementioned variable is set, then add the following line to your local it.ini file:

```
LoadExtension python falsecolor.py
```

This tells it to load a file containing an "it" custom command from that file. And in that file, put the following:

```
#
# A "it" command example
#
import it
import ice
class FalseColorCommand(It3Command):

    def __init__(self):
        self.m_menuPath = 'Commands/CustomCmds/False Color'

    def Invoke(self):
        elem = it.GetCurrentElement()
        img = elem.GetImage()
        it.app.Info('Applying false color.')
        it.extensions.falsecolor(aov='CPU', img=img)

it.commands.append(FalseColorCommand)
```

If you open "it" now you should see the logging messages we added to your it.ini file, and in the **Commands/CustomCmds** menu you'll see a False Color option. Open or render an image with a CPUTime AOV and you can now select the new command.



If you do not specify a path to your extension file on the LoadExtension line, "it" will automatically look in the same directory where the it.ini file exists for your extension file.

Let's take a closer look at falsecolor.py to see what it is doing. If you have experience in programming with Python, this should all look familiar to you.

```
import it
import ice
```

This imports both the 'it' and 'ice' modules, which gives us access to both "it" and IceMan scripting functions.

```
class FalseColorCommand(It3Command)
```

This declares a new class 'FalseColorCommand' for our custom command. Note that all "it" custom commands must be a subclass of 'It3Command'.

```
def __init__(self):
    self.m_menuPath = 'Commands/CustomCmds/False Color'
```

This is the \_\_init\_\_ constructor method for our new class. The m\_menuPath defines where to place our custom command in the "it" menu.

```
def Invoke(self):
    elem = it.GetCurrentElement()
    img = elem.GetImage()
    it.extensions.falsecolor(aov='CPU', img=img)
```

This is the Invoke method for our class, which is the meat of our custom command. This method will be called when our custom command is executed. The first line of our method retrieves the current image element in our "it" catalog. The second line retrieves the ice.Image instance for the image elements. Finally, the last line calls the falsecolor command passing in the name of the AOV to search for (CPU) and the ice.Image instance.

```
it.commands.append(FalseColorCommand)
```

This tells the "it" application about the new command. The python list *it.commands* sets the contents and the order of the Commands menu. Note that you add the python class, in this case *FalseColorCommand*, not an instance of the class to the list.

## Add a Dialog to the Command

ice.extensions.falsecolor takes a couple of options - w and m - which control what range of hue to scale the input to. Since a hue value isn't very meaningful or easy to remember, we'll make a dialog for the user to pick from a few color schemes where we have pre-selected the values for the hue range.

As "it" is now built with the Qt framework, users can also use Qt to create their own dialogs.

Update falsecolor.py with the example below and restart "it".

```
import it
import ice
from it.It3Command import It3Command

from PyQt5.QtGui import QDialogButtonBox
from PyQt5.QtGui import QHBoxLayout
from PyQt5.QtGui import QLabel
from PyQt5.QtGui import QPushButton
from PyQt5.QtGui import QComboBox
from PyQt5.QtGui import QWidget

class FalseColorCommand(It3Command):

    def __init__(self):
        self.m_menuPath = 'Commands/CustomCmds/False Color'
        self.m_dlg = None
        self.m_stdButtons = QDialogButtonBox.Close |
        QDialogButtonBox.Apply
        self.m_colorSelected = 0
        self.m_wList = [0.2, 0.3, 0.3]
        self.m_mList = [0.6, 0.4, 0.1]

    def Invoke(self):
        if self.m_dlg == None:
            # since we're going to run modeless need to hang onto the
```

```

        # dialog object or it'll get deleted when Invoke exits.
        # 'it' has a hold of 'self' so we won't go away.
        self.m_dlg = self.makeUI()
    self.m_dlg.show()
    self.m_dlg.raise_()
    self.m_dlg.activateWindow()

def apply(self):

    elem = it.GetCurrentElement()
    img = elem.GetImage()
    w = self.m_wList[ self.m_colorSelected ]
    m = self.m_mList[ self.m_colorSelected ]
    it.extensions.falsecolor(aov='CPU', img=img, w=w, m=m)

def currentIndexChanged(self, index):
    self.m_colorSelected = index

def makeUI(self):
    dlg = self.CreateDialog('False Color...')
    contents = dlg.findChild(QVBoxLayout, 'contents')

    layout = QHBoxLayout()
    contents.addLayout(layout)

    layout = QHBoxLayout()
    contents.addLayout(layout)
    label = QLabel("Color: ")
    layout.addWidget(label)

    colorComboBox = QComboBox()
    colorComboBox.addItem("Bluish")
    colorComboBox.addItem("Greenish")
    colorComboBox.addItem("Yellowish")
    colorComboBox.connect('currentIndexChanged(int)', self.currentIndexChanged)
    layout.addWidget(colorComboBox)
    layout.addStretch()

    layout = QHBoxLayout()
    contents.addLayout(layout)
    doItButton = QPushButton("Ok")
    layout.addWidget(doItButton)
    doItButton.connect('clicked()', self.apply)

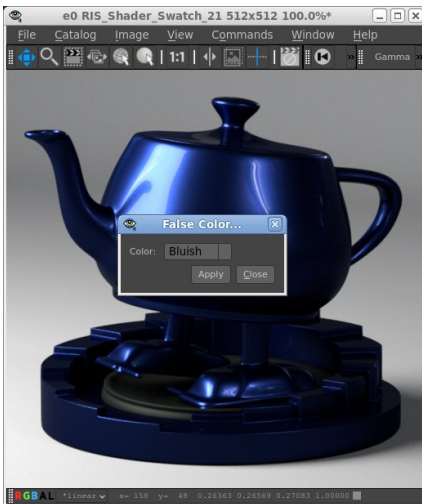
    bbox = dlg.findChild(QDialogButtonBox, 'bbox')
    doItButton = bbox.button(QDialogButtonBox.Apply)
    doItButton.connect('clicked()', self.apply)

    return dlg

it.commands.append(FalseColorCommand)

```

Once you've restarted "it", load your image with a CPUtime AOV and run the command again. This time you should see our new dialog:



Going through the changes one at a time:

```
from PythonQt.QtGui import QDialogButtonBox
from PythonQt.QtGui import QHBoxLayout
from PythonQt.QtGui import QLabel
from PythonQt.QtGui import QPushButton
from PythonQt.QtGui import QComboBox
from PythonQt.QtGui import QWidget
```

Here, we are importing a couple more modules, necessary to create our dialog.

```
def __init__(self):
    self.m_menuPath = 'Commands/CustomCmds/False Color'
    self.m_dlg = None
    self.m_stdButtons = QDialogButtonBox.Close | QDialogButtonBox.Apply
    self.m_colorSelected = 0
    self.m_wList = [0.2, 0.3, 0.3]
    self.m_mList = [0.6, 0.4, 0.1]
```

We've modified our `__init__` constructor to add a couple of more member variables. Adding these will make it easier to share data between methods that need it. `m_dlg` holds our dialog instance that will be created in the `makeUI` method (see below). The third line allows us to add standard Close and Apply buttons to our dialog. `m_colorSelected` is the current selected color. `m_wList` and `mList` are lists of available values for the hues.

```

def Invoke(self):
    if self.m_dlg == None:
        # since we're going to run modeless need to hang onto the
        # dialog object or it'll get deleted when Invoke exits.
        # 'it' has a hold of 'self' so we won't go away.
        self.m_dlg = self.makeUI()
    self.m_dlg.show()
    self.m_dlg.raise_()
    self.m_dlg.activateWindow()

def apply(self):

    elem = it.GetCurrentElement()
    img = elem.GetImage()
    w = self.m_wList[ self.m_colorSelected ]
    m = self.m_mList[ self.m_colorSelected ]
    it.extensions.falsecolor(aov='CPU', img=img, w=w, m=m)

```

Here, we've moved the original code in the Invoke method to a new apply method. The Invoke method has been changed to call makeUI, which will create the Dialog instance. The instance is then saved to m\_dlg. Once we have an instance we display the dialog to the screen. We use the m\_colorSelected variable to select which values we want from the m\_wList and m\_mList.

```

def currentIndexChanged(self, index):
    self.m_colorSelected = index

def makeUI(self):
    dlg = self.CreateDialog('False Color...')
    contents = dlg.findChild(QVBoxLayout, 'contents')

    layout = QHBoxLayout()
    contents.addLayout(layout)

    layout = QHBoxLayout()
    contents.addLayout(layout)
    label = QLabel("Color: ")
    layout.addWidget(label)

    colorComboBox = QComboBox()
    colorComboBox.addItem("Bluish")
    colorComboBox.addItem("Greenish")
    colorComboBox.addItem("Yellowish")
    colorComboBox.connect('currentIndexChanged(int)', self.currentIndexChanged)
    layout.addWidget(colorComboBox)
    layout.addStretch()

    bbox = dlg.findChild(QDialogButtonBox, 'bbox')
    doItButton = bbox.button(QDialogButtonBox.Apply)
    doItButton.connect('clicked()', self.apply)

    return dlg

```

The makeUI method is where our dialog window is created. Programming with Qt to create a UI is beyond the scope of this documentation, but, essentially, a dialog is created with the CreateDialog method. We then add layouts to the dialog, followed by a label describing the option, and a QComboBox that will present a dropdown menu for the user to select the color option they would like to use. The QComboBox is connected to our currentIndexChanged method so that whenever the current selection is changed, the new index is passed to the currentIndexChanged method where we can update our m\_colorSelected member variable. Finally, we retrieve the standard Apply button from our dialog and connect our apply method, so that it is called whenever the button is pressed.