# Tutorial 3: Where to Use Scripts in VRED with Typical Use Cases

The *Tutorial 3: Where to Use Scripts in VRED with Typical Use Cases* video covers the following:

- Script Editor
- Variant Set scripts
- Script preferences
- Command line parameters: `-prepython` and `-postpython`
- Import of Python files (.py)

**Download the example scripts here**: https://github.com/raemorris/VRED-Python-Tutorial-Examples-Scripts/blob/main/tutorial3.zip?raw=true

## Sample Python Code

There are the three example Python scripts for the *Tutorial 3: Where to Use Scripts in VRED with Typical Use Cases* video.

- To view the video, visit: https://www.youtube.com/watch?v=qqzeeaAaPgU
- To review the video caption, see *Video Captions*.

## command_line_parameter.txt

```
REM Example 1

REM -prepython "print('Executed before file has loaded')" -postpython
"print('Executed after file has loaded')"

REM Example 2

start "D:\Programme\Autodesk\VREDPro-13.3\bin\WIN64\VREDPro.exe"
"C:\ProgramData\Autodesk\VREDPro-13.3\examples\Automotive_Genesis.vpb" -
prepython "print('---');print('Executed before file has loaded')" -postpython
"print('---');print('Executed after file has loaded')"
```

## script_editor_scripts.py

```
# Example 1.0

# Global function library
```

```
import time
from datetime import datetime


def logInfo(message):
    now = datetime.now().time().strftime("%H:%M:%S.%f")
    print(now, '[INFO]', message)



def logWarn(message):
    now = datetime.now().time().strftime("%H:%M:%S.%f")
    print(now, '[WARN]', message)



def createViewpointFromCamera():
    now = datetime.now().time().strftime("%H:%M:%S.%f")
    vrCameraService.createViewpoint("vp_{}".format(now))



def renderViewpoints():
    viewpoints = vrCameraService.getAllViewpoints()

    renderDirectory = vrFileDialog.getExistingDirectory(
                                        "Select a render directory:",
                                        vrFileIO.getFileIOBaseDir()
                                    )

    if not renderDirectory:
        logWarn("No directory where to save the renderings!")
        return

    for viewpoint in viewpoints:
        name = viewpoint.getName()

        if name.startswith('vp_'):
            vrRenderSettings.setRenderFilename("{}.jpg".format(name))
            vrRenderSettings.startRenderToFile(False)
```

## variant_set_scripts.py

```
# Example 1
# Recreating material switches
print('--- Create Switch Materials ---')
allMaterials = getAllMaterials()
# get all materials which name starts with "switch"
switchMaterials = list(filter(lambda matPtr:
matPtr.getName().startswith('switch'), allMaterials))
# iterate all switch materials
for switchMaterial in switchMaterials:
switchPtr = switchMaterial
switchName = switchPtr.getName()
print('Found switch: ' + str(switchName))
# extract material prefix
materialPrefix = '_'.join(switchName.split('_')[1:])
# Get all materials which name starts with the material prefix
childMaterials = list(filter(lambda matPtr:
matPtr.getName().startswith(materialPrefix), allMaterials))
childMaterialNames = [x.getName() for x in childMaterials]
print('Found child materials: ' + ', '.join(childMaterialNames))
# Create new material switch
print('Create new material switch...')
newMaterialSwitch = createMaterial("SwitchMaterial")
newMaterialSwitch.setName(switchName)
# Add child materials to switch
for child in childMaterials:
```

```
newMaterialSwitch.addMaterial(child)

# swap old switch material with new switch material

print('Assign new material switch...')

nodes = switchPtr.getNodes()

for node in nodes:

node.setMaterial(newMaterialSwitch)

print('')

print('Finished!')

print('---')

# Example 2

# Calling global functions in the script editor

logInfo('This is an info message from a variant set.')

logWarn('This is a warning message from a variant set.')

createViewpointFromCamera()

renderViewpoints()
```

## Video Captions

Hello and welcome to our Python Scripting Tutorials for VRED Pro. I'm your host Christopher and today I will talk about all the different Python interfaces in VRED and how you can use them efficiently.

VRED offers many ways to script with Python. There are multiple interfaces where you can place Python code to affect the behavior of your scene or VRED in general. But not every interface is suited for the same tasks. Some problems are better solved in one place instead of the other. Here I want to present all different places to put your Python scripts, from simple to complex, with some examples to get you started.

Let's start with the simplest form of Python interface there is in VRED, the Terminal. The Terminal both acts as input for Python commands and output for print messages that are generated by VRED, other scripts, or your own code. When you open the Terminal after starting VRED, you will see a bunch of messages that are basically initialization messages from VRED that are telling you what resources have been loaded and whether there have been any errors. Every print message you, or any other script generates, does show up in the Terminal, so it is the only place where we can directly see whether there was an error, while running your script, or where you can look at your log messages.

The Terminal also acts as a basic input for Python commands, much like the Python interpreter in interactive mode. You enter a command and either receive a green feedback for a successful run, or red feedback when an error occurs. Because the Terminal is just a one line input it is mostly used to test out Python functions. Also, everything you write in the Terminal is lost when you restart VRED and is not saved with your file. The Terminal can access variables and functions that are defined globally in the

Script Editor, in the preferences, or in the variant sets, but all variables and functions you define in the Terminal are not accessible in other scripts.

Before we discuss the more prominent Python scripting interfaces, let's have a look at script preferences. Script preferences are Python scripts that are stored in your VRED application preferences and are always available, independent of the file you have open. Script preferences are great for defining custom key mappings. When you first open your script preferences, there are already some key mappings defined on the F keys that change render settings or other things.

Script preferences are the place for small Python scripts or variables you need all the time and that are independent from the file you have open. So, for example, you could define some custom log functions that automatically add a timestamp to your Terminal message. You also could define a function that takes a screenshot or does preview rendering and saves it on your hard drive. But most of the time, you probably want to customize your key mappings with extra functions. For example, switching between different render qualities. All functions you defined in the Script preferences are globally available to the Script Editor, the variant sets, and the Terminal. That means you can, for example, call a function from the Script preferences directly from a variant set.

Variants sets are also our next topic. You can write Python scripts directly in variants sets. They have a Script tab that can contain Python code that is executed when the variant set is triggered. You can define variables, functions, and even classes, but mostly you would write scripts that do one particular thing.

In this example, I implemented a data preparation tool that is triggered by activating the variant set. The variant set here acts as a Python tool that can be triggered by just double-clicking. The script itself matches materials and material switches that got lost when importing from another CAD software. It searches for materials with a particular name and generates material switches. And then, it replaces the broken materials with the new material switches.

Here you can see that the script is not too long and solves exactly one problem. So it is also quite maintainable. There are some comments in the code and print messages that help make the code more readable and understandable, and it also helps a user to see if the code actually did what it was supposed to do or if there have been any errors.

The Python script you write in a variant set is locally scoped. That means that variables, functions, and classes are only available in this particular variant set. You cannot call a function that is defined in variant set A from the script in variant set B. This also means, that if you want to use a script in more than one variant set, you should probably put it in the Script Editor, where it is globally available, and just call it from the variant set. This also prevents duplicate code.

Last but not least, you should also be aware of how much code you put in a variant set. The previous example is about 40 lines, which is ok. But when you have scripts that contains hundreds of lines of code, you basically have a program that is unmaintainable and hard to understand. You will need lots of resources to extend such code or eliminate errors and bugs. In this case, you are better off using a plugin or defining functions in the globally available Script Editor, which we will discuss next.

The Script Editor is available under the menu, Edit and Script Editor. The Script Editor is comparable to a Python file that is directly attached to your VRED scene. Here you can define globally available variables, functions, and also classes. All Python code you write here is also available in, for example, the variant sets, the Terminal, or the webengine. So, it's globally available.

At the bottom of the editor, you have the Run button, which executes all the code in the Script Editor. But there are also settings in the preferences under FileIO. VRED files where you can specify whether the script in the editor should automatically execute when the file is loaded. The default setting is on, so if you don't want to run a script automatically, you want to turn this option off.

The Script Editor is great for globally defined functions and variables. For example, you could have a function that creates a viewpoint from the current camera position. This function can be called from any variant set or even be triggered remotely. Then you could have a function that batch renders all these viewpoints and stores them in a directory you can define. But of course, you also can write complex scripts in the Script Editor.

In Python, you can put multiple modules, classes, and functions in one file. So in theory, you could implement global functions, multiple tools, and also user interfaces in this single Script Editor. But in my opinion, this would be really messy, overly complex, and not very maintainable. If your script is longer than a few hundred lines of code, you probably want to separate the functionality in script plugins or a modules. We have a tutorial video about extending VRED's VR Menu, where we implement additional tools. For this task, the Script Editor is the right place.

When you have a repository with Python tools, it can be useful to directly import script with the Import function. This is as easy as opening a file and the imported script is just appended to the content in the Script Editor and executed. But here you have to be careful. If there is already a script in your editor that manipulates the scene in some way, then this script is executed again every time you import another script.

The Import function can be used to quickly add a custom user interface or other Python tools you have lying around as a file. In our Extending the VR Menu tutorial, we use this method to import a custom VR tool in our scenes.

In VRED, it's also possible to build up fully automated render pipelines. To load scenes and automatically render images or movies, the next two command line options come in handy: `-prepython` and `-postpython`. You can use `-prepython` to execute scripts before the scene is loaded, and `-postpython` to run script after the scene is loaded. That means with `-prepython`, you can, for example, run scripts that are independent from your scene, like from modules or even the script preferences. And with `-postpython`, you could run scripts that are defined in your Script Editor or start a rendering job that is stored as a preset. Basically, you can enter all Python commands you can also enter in your VRED Terminal, separated with a semicolon.

A common use case would be loading a scene, importing other CAD models, and then render all the viewpoints or movies. After this, you can terminate VRED and start with another file. For automated rendering, it's recommended to have some render presets, so you can select the correct render settings with the command line.

Another way of adding often used Python tools and functions is to use custom Python modules. They are added in the VRED application directory under `lib` directory, and `python`. They are structured like any other module and can contain classes, functions, and basically anything a Python module can hold.

What you have to be careful about is that not all VRED modules are automatically import, as it’s the case when you script in the Script Editor. That means you have to explicitly import all API v1 modules before you can use them. The API v2 modules, however, are injected by VRED and don’t have to be imported.

But, nonetheless, modules are a great way to provide multiple tools, Python functions, and classes all at once. You are able to install these modules for any VRED designer or developer, and they all share the same set of Python scripts. This is great because these scripts can be managed in a repository and it's much easier to maintain a good code quality. For Python scripts that are stored in a VRED file or in variant sets, this is not so easy because all scripts are scattered across files.

Script plugins are probably the most advanced way of adding functionality to a VRED instance. They are basically VRED tools that come with a user interface and are available in the VRED menu bar. Plugins can be just a few lines of code that do one small task, but they also can be hundreds of lines of code and change every aspect on how you interact and work with VRED. Of course, you could implement the same functionality in the Script Editor. But plugins are independent from VRED files and are usually written with third-party code editors.

The advantage of developing your own plugins is that plugins are integrated in the VRED menu bar, plugins can be easily shared with other designers and developers, and they are separate from the other Python code you have in your scene. So, a plugin cannot accidently change the state of other Python scripts you are using.

I do not want to tell you too much about script plugins right now, because we have a whole tutorial about it. In this tutorial, I will show you how you can implement your own script plugin, along with a user interface. Be sure to check it out!

As you see, there are multiple different ways to change your scene or the way you work with VRED with some simple, or some more complex Python scripts. At this point, you should have a pretty good idea about all the possibilities on how to use Python scripting in VRED. For each problem, there is more than one solution to it, but some Python interfaces are more suited than others to achieve your goals.

I hope I could give you some guidelines in this videos, how you can approach these challenges. That’s it for today. I hope we see each other in the upcoming tutorials. Thanks for joining me and see you next time!