Corel® PaintShop® Pro

Scripting Guide

Revision date:
August 10th, 2015

Contents

Introduction 1
Scripting resources 1
Anatomy of a script 1
Anatomy of a preset 4
The App Object 5
The Do method 6
GeneralSettings 6
The Documents collection 8
The Document object 9
The Constants object 10
The Commands collection 11
Tips 11
Common operations 19
Selecting a layer 19
Select multiple layers and objects 22
Iterating layers 24
Selecting a different document 26
Iterating documents 26
Saving/Restoring a selection 27
Checking for a selection 27
Testing Image/Layer Attributes 28
Requiring a Document 28
Promoting To True Color 29
Processing all the files in a directory 29
Getting information from PaintShop Pro 31
Commands for Scripting 32
Command: SelectNextLayer 32
Command: SelectPreviousLayer 34
Command: GetNextObject 34
Command: GetPrevObject 34
Command: HasVectorSelection 34
Command: GetVectorSelectionRect 35
Command: GetRasterSelectionRect 35
Command: SaveMaterialSwatch 36

Command: SetMaterial ... 36
Command: SetMaterialStyle ... 37
Command: ShowSwatchView ... 37
Command: EnableMaterialTexture ... 37
Command: GetMaterial ... 38
Command: GetVersionInfo ... 38
Command: ReturnImageInfo ... 39
Command: ReturnFileLocations ... 40
Command: ReturnLayerProperties ... 40
Command: ReturnVectorObjectProperties ... 42
Command: AddGroupsAndObjects ... 44
Command: NodeEditAddPath ... 45
Command: DeleteEXIF ... 46
Command: ReturnGeneralPreferences ... 46
Command: GetCommandInfo ... 46
Command: EventNotify ... 50
Command: StartForeignWindow ... 51
Command: MsgBox ... 52
Command: GetString ... 54
Command: GetNumber ... 55
Command: ScriptWndAutoShow ... 57
Command: ScriptWndClear ... 57
Command: DumpScriptOutputPalette ... 57
Command: ScrollScriptOutputPaletteToEnd ... 58
Command: SelectDocument ... 58
Command: SelectTool ... 59
Command: SelectPreviousTool ... 60
Command: SelectNextTool ... 61
Appendix A: Sample Scripts ... 62

Introduction

Most users of Corel® PaintShop® Pro never need to look at the source code of a script. The script recording features inside PaintShop Pro are such that many scripts can be created using nothing more than the script recorder.

However, in order to write scripts that make decisions, that use data from an open document, or that prompt the user for data, it is necessary to delve deeper into scripting.

PaintShop Pro scripts are written in the Python programming language. There are a number of books written about Python, but you can also find a number of Python tutorials at www.python.org.

Before writing or editing a script, you should have at least minimal familiarity with Python. It's hard to understand what's happening in a script if you are struggling to understand the syntax in which it is presented.

Scripting resources

In addition to this scripting guide, you can use the following resources:

- **Sample Scripts** – sample scripts are usually bundled with the Scripting Guide. For more information about sample scripts, see *Appendix A: Sample Scripts*.
- **Scripting API** – an online HTML-based reference guide to the commands and parameters that can be used in scripts. Click the following link: Corel PaintShop Pro Scripting API.

Anatomy of a script

All scripts created by the PaintShop Pro script recorder have an identical format. To access the script recorder in PaintShop Pro, click **View** > **Toolbars** > **Script**. For more information about using the script recorder, see "Working with scripting tools and features" in the Help.

A simple script is shown below. Don't be intimidated by the length, all it does is execute four commands:

- Float the selection
- Apply the Inner Bevel command
- Apply the Drop Shadow command
- Promote the selection to a layer

This is a simple script that was created entirely from the script recorder.

```
from PSPApp import *

def ScriptProperties():
      return {
            'Author': u'Corel Corporation',
            'Copyright': u'Copyright (C) 2015, Corel Corporation, All Rights Reserved.',
            'Description': u'Bevel and cast a shadow from a selection',
            'Host': u'PaintShop Pro',
            'Host Version': u'18.00'
            }

def Do(Environment):

      # FloatSelection
      App.Do( Environment, 'FloatSelection', {
            'ClearSource': None,
            'BackupSelection': None,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Default,
                  'AutoActionMode': App.Constants.AutoActionMode.Match,
                  'Version': ((18,0,0),1)
                  }
            })

      # Inner Bevel
      App.Do( Environment, 'InnerBevel', {
            'Ambience': 0,
            'Angle': 315,
            'Bevel': 0,
            'Color': (255,255,255),
            'Depth': 20,
            'Elevation': 30,
            'Intensity': 50,
            'Shininess': 0,
            'Smoothness': 0,
            'Width': 8,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Default,
                  'AutoActionMode': App.Constants.AutoActionMode.Match,
                  'Version': ((18,0,0),1)
                  }
            })
      # Drop Shadow
      App.Do( Environment, 'DropShadow', {
            'Blur': 5,
            'Color': (0,0,0),
            'Horizontal': 10,
            'NewLayer': False,
            'Opacity': 50,
            'Vertical': 10,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Default,
                  'AutoActionMode': App.Constants.AutoActionMode.Match,
                  'Version': ((18,0,0),1)
                  }
```

```
        })

    # SelectPromote
    App.Do( Environment, 'SelectPromote', {
            'KeepSelection': None,
            'LayerName': None,
            'GeneralSettings': {
                'ExecutionMode': App.Constants.ExecutionMode.Default,
                'AutoActionMode': App.Constants.AutoActionMode.Match,
                'Version': ((18,0,0),1)
                }
            })
```

The first line of the script says "from PSPApp import *". PaintShop Pro implements the PSPApp module, and all communication from the script to PaintShop Pro is by means of methods and objects exposed by the PSPApp module. Though it is possible to write a script that does not make use of PSPApp, as a practical matter all scripts will import PSPApp.

Next, there is the ScriptProperties method. This serves as documentation for the script (author/copyright/description), as well as fields that PaintShop Pro can use to ensure that the version of the script is compatible with the version of PaintShop Pro (Host/Host Version). The ScriptProperties method can be omitted and the script will still run, but it should be included so that the data in it can be extracted by PaintShop Pro.

Next comes the Do method. When PaintShop Pro runs a script, it does so by loading the script into memory and invoking the Do method. The Do method takes a single parameter, by convention called Environment. This parameter exists to help coordination between the script and PaintShop Pro, and needs to be passed back to PaintShop Pro on all calls to App.Do. Do not modify the environment variable.

For all scripts created by the PaintShop Pro script recorder, the Do method is nothing more than a series of calls to App.Do. App is a member of the PSPApp module, and could also be written as PSPApp.App

The App.Do method is used to call a PaintShop Pro command (i.e. open a file, run an effect, create a layer, put down a paint stroke). The App.Do command takes four parameters, though the last two are optional:

1. The first parameter is the Environment variable passed to the script's Do method.
2. The second parameter is the name of the command to run (e.g. FloatSelection).
3. The third parameter is a dictionary that contains all the parameters needed by the command. This is frequently referred to as the parameter repository.
4. The fourth parameter (always omitted by the script recorder) is the document on which the command should be run. When omitted, commands are always run on the target document, which is the document that was active when the script was started.

The script shown above is 68 lines long, including whitespace. However, virtually all of it is the specification of the parameter repositories. Without the repositories it would only be 16 lines long:

```
from PSPApp import *

def ScriptProperties():
      return {
            'Author': u'Corel Corporation',
            'Copyright': u'Copyright (C) 2015, Corel Corporation, All Rights Reserved.',
            'Description': u'Bevel and cast a shadow from a selection',
            'Host': u'PaintShop Pro',
            'Host Version': u'18.00'
            }

def Do(Environment):
      App.Do( Environment, 'FloatSelection', {})
      App.Do( Environment, 'InnerBevel', {})
      App.Do( Environment, 'DropShadow', {})
      App.Do( Environment, 'SelectPromote', {})
```

Since the parameter repositories specify precisely how each command should behave, it is proper that they consume the majority of the script. It is also worth noting that the shorter script above is perfectly valid and would run without error – any parameters not specified default to the last used value.

All scripts recorded by PaintShop Pro follow the canonical format shown in the initial example– the single import statement, a script properties method, and a Do method containing calls to App.Do.

Anatomy of a preset

PaintShop Pro supports presets for just about every command. As shown below, these are just a specialized form of a script:

```
from PSPApp import *

def ScriptProperties():
      return {
            'Author': u'Corel Corporation',
            'Copyright': u'Copyright (C) 2015 Corel Corporation, All Rights Reserved.',
            'Description': u'800x600 greyscale, transparent, raster',
            'Host': u'PaintShop Pro',
            'Host Version': u'18.00'
            }

def Preset_NewFile():
      return {
            'Width': 800,
            'Height': 600,
            'ColorDepth': App.Constants.Colordepth.Greyscale,
            'DimensionUnits': App.Constants.DimensionType.Pixels,
```

```
            'ResolutionUnits': App.Constants.ResolutionUnits.PixelsPerIn,
            'Resolution': 300,
            'FillMaterial': {
                  'Color': (0,0,0),
                  'Pattern': None,
                  'Gradient': None,
                  'Texture': None,
                  'Art': None
                  },
            'Transparent': True,
            'LayerType': App.Constants.NewLayerType.Raster,
            'ArtMediaTexture': {
                  'Category': u'Art Media',
                  'Name': u'Canvas coarse'.
                  'EnableFill': True,
                  'FillColor': (255,255,255)
                  },
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Default,
                  'AutoActionMode': App.Constants.AutoActionMode.Match,
                  'Version': ((18,0,0),1)
                  }
            }

def Do(Environment):
      # FileNew
      App.Do( Environment, 'NewFile', Preset_NewFile())
```

Looking at the sample shown above, you can see that the only difference between a script and a preset is that the preset has an additional method defined, in this case Preset_NewFile().

Preset method names are defined by the command that creates them, but as a rule take the form Preset_CommandName.

When the preset is loaded, the preset method is called instead of the Do method. The method returns the parameters for the command (in this case NewFile), which are then loaded without actually executing the command.

Note also that the preset includes a Do method as well. The Do method just calls App.Do with the associated command (NewFile), and for the third parameter, it passes the return value of the preset method. Since the third parameter to App.Do is the parameters of the command, the net result is that executing a preset as a script runs the proper command using the parameters specified in the preset.

The App Object

The App object is the core structure exposed by the PSPApp module. Although you will usually interact with it by calling commands with App.Do, other methods and members are useful to the script author.

The Do method

The Do method is the primary interface between a script and PaintShop Pro. The Do method can execute any of the more than 650 different commands that exist in PaintShop Pro.

The general form of the Do command is:

```
result = App.Do( Environment, command, parameter_repository, TargetDocument )
```

The Environment parameter is required, and should always be the Environment parameter that was provided to the script as a parameter to the Do method of the script. This is used by PaintShop Pro to keep track of which script is executing a command, and does not contain any information that is useful to the script.

The command parameter is just what it implies–the name of the command you want to execute, such as 'FileClose' or 'SelectNone'. This is a required parameter.

The parameter repository parameter is a Python dictionary containing all of the parameters required by the command. With the exception of the general settings subrepository described below, the contents of this dictionary are different for each command. This is an optional parameter – if not specified, the last used parameters of the command are used.

The target document parameter controls which document the command is executed against. The script recorder always omits this parameter, and unless changed by executing a command that selects a different document, the target document is the document that was active when the script was invoked.

GeneralSettings

All commands support a sub repository called GeneralSettings. These parameters are common to many different commands. Not every command uses every value, but all commands support ExecutionMode and AutoActionMode at a minimum.

An example of a GeneralSettings repository is shown below:

```
'GeneralSettings': {
      'ExecutionMode': App.Constants.ExecutionMode.Default,
      'DialogPlacement': {
            'ShowMaximized': False,
            'Rect': ((639,417),634,517)
            },
      'PreviewVisible': True,
      'AutoProof': True,
      'RandomSeed': 55042743,
      'AutoActionMode': App.Constants.AutoActionMode.Match,
      'Version': ((18,0,0),1)
      }
```

The execution mode value controls whether or not the command displays a dialog. You can specify three different values in a script:

- **Silent** – the command is run without showing a dialog. Any errors cause a message to be written to the output window.
- **Interactive** – the command loads the parameters from the script into its dialog, and allows the user to edit the parameters before continuing. Errors are shown in a message box. Specifying interactive mode on a command that has no dialog is not an error.
- **Default** – use the execution mode of the calling command, or if there is no calling command, use the execution mode of PaintShop Pro itself. When a script runs from PaintShop Pro, the user can specify that the execution mode should be either silent or interactive, and any command with a default execution mode will use that mode.

An execution mode of Default is indeterminate–at the time the command is run it uses the execution mode of the parent command to determine an effective execution mode, which is always either Silent or Interactive.

The Version parameter indicates the version of the program that the script was recorded under, (major, minor, micro), followed by the command version.

The AutoActionMode value controls the behavior of auto actions (also known as fixups). Auto actions are actions that PaintShop Pro takes to enable a command to run. The simplest example is attempting to run an effect on a paletted image when the effect requires a true-color image. When the command is run, PaintShop Pro executes the "Convert image to 16-million colors" auto action before running the effect.

Auto actions in turn have 3 different states:

- Always execute the action without prompting the user.
- Never execute the command (any command depending on this auto action is disabled).
- Ask the user for permission to execute the auto action.

Commands can override the default auto action behavior by setting the AutoActionMode value in the GeneralSettings.

There are six possible auto action modes:

1. **Match** – if the effective execution mode is silent, equivalent to **PromoteAsk**. If the effective execution mode is interactive, equivalent to **Default**.
2. **AllNever** – assume all auto actions are set to "Never"
3. **DemoteAsk** – all auto actions set to "Ask" are demoted to "Never"
4. **Default** – use the auto actions as specified in the preferences

5. **PromoteAsk** – all auto actions set to "Ask" are promoted to "Always"

6. **AllAlways** – all auto actions are set to "Always"

Many commands in PaintShop Pro execute other commands as subcommands, and frequently set the AutoActionMode to control fixup behavior. The parameter is available to a script, but you should err on the side of respecting the user's preferences.

The DialogPlacement dictionary (as this illustrates, dictionaries can be nested) contains information about the size and placement of the dialog on screen. It is written only to a recorded script if the "save dialog positions" box is checked when a script is recorded, and is present only for commands that use a dialog. The Rect tuple contains the size and position of the dialog on screen. PaintShop Pro always encodes rectangles as ((x,y), dx, dy).

The PreviewVisible and AutoProof flags are present only for effect dialogs, and capture whether the preview windows are shown or hidden, and the state of the auto-proof toggle. Like DialogPlacement, these members are only written if dialog positions are being saved.

The RandomSeed parameter is only present for commands that make use of random numbers, such as the Add Noise dialog or the Brush Strokes effect. If a random seed is specified, repeated runs of the command on the same input would produce exactly the same output. If the random seed is omitted, a new seed value is generated for each run of the command, leading to different results for each run.

The Documents collection

PaintShop Pro maintains a list of all the currently open documents. This order of the list is the order in which the documents were opened.

Note that the list returned by App.Documents is a snapshot of the current state of PaintShop Pro, not a pointer to the internal state of PaintShop Pro – operations done on the list have no effect on PaintShop Pro.

In addition to the Documents collections, PaintShop Pro exposes two other document objects:

- **App.ActiveDocument** is the document active in the GUI. This will have the active title bar, and be represented in the layer palette and overview window.
- **App.TargetDocument** is the document that commands will be applied to. In most cases, it is the document that was active when the script was launched, though opening or closing documents changes the target document, as does the SelectDocument command.

Be very careful in using App.ActiveDocument. This is completely dynamic, and can change while your script is running. For instance, the following code tries to blur the active image 5

times – if the user switches documents while the script is running, the script will follow the currently active document.

```
for x in range(5):
      App.Do( Environment, 'Blur', {}, App.ActiveDocument )
```

The simplest alternative is to use App.TargetDocument rather than App.ActiveDocument. However, if you want to use App.ActiveDocument, capture its current value before entering the loop:

```
DocToBlur = App.ActiveDcoument
for x in range(5):
      App.Do( Environment, 'Blur', {}, DocToBlur )
```

The Document object

The objects in the documents collection are document objects. Document objects expose the following members:

- Height – height of the image in pixels
- Width – width of the image in pixels
- Size – tuple of (width, height)
- Title – title of the document. For an unsaved image, this will be something like "Image1". For a saved document, it will be the filename and extension, without the path.
- Name – the complete pathname of the document. If the document is unsaved this will be an empty string.

```
def Do(Environment):
      print App.ActiveDocument.Width
      print App.ActiveDocument.Height
      print App.ActiveDocument.Size
      print App.ActiveDocument.Title
      print App.ActiveDocument.Name
```

The members of the Document object are simply present for convenience and are not intended to be a comprehensive list of all of the properties of an image. For more in- depth information you need to use the ReturnImageInfo and ReturnLayerProperties commands.

If no document is opened the Active and Target documents are "None". If you try to access an attribute on a "None" object you will receive an error. You can check for the existence of a document by checking for the "None" value. Example:

```
      if App.TargetDocument is None:
            print 'No document'
      else:
            print 'We have a document'
```

The Constants object

In the few snippets of sample code given, you have probably noticed such torturous constructs as App.Constants.ExecutionMode.Silent.

These are simply constants that are defined by the PSPApp module. Though to Python they are simply integers, using the symbolic names serves two purposes:

1. They make the script easier to read – App.Constants.Blendmode.Normal makes more sense for defining a blend mode than a value of 0.

2. They improve portability across versions. While we won't guarantee that App.Constants.Blendmode.Normal always has a value of zero, we can guarantee that so long as we have a normal blend mode, App.Constants.Blendmode.Normal will be a valid identifier for it.

The biggest problem with constants is to know which constants exist and what the symbolic names are. The simplest way to get information on a constant is to run the ExploreConstants script that is included with the script samples material. This tells you all of the names and values for any constant in the system. The blend mode is shown below.

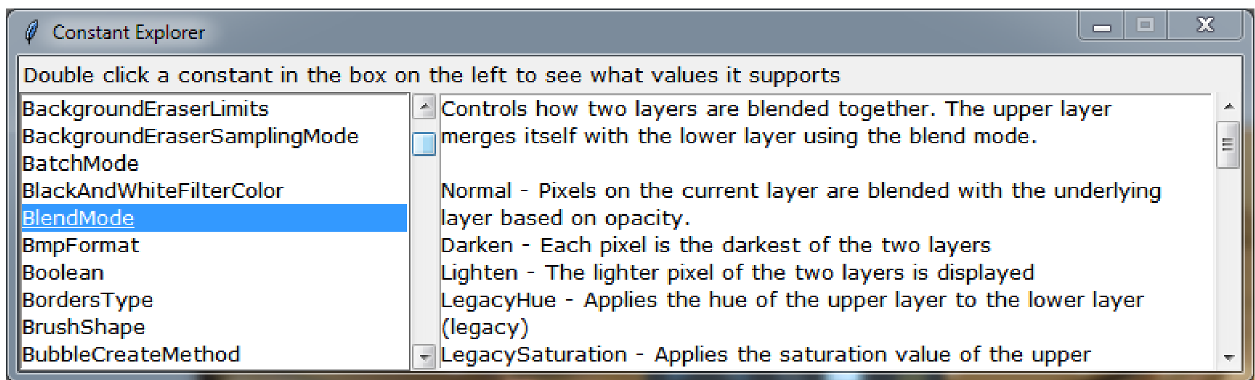

Figure 1

This script is possible because PaintShop Pro exposes information about constants through the App object. App.Constants is the complete pool of all constants, and it has a method called All(), which returns a list of all the constants in the system.

Each constant object has further methods for extracting the name, description, and values defined by that constant.

- Name() returns the name of the constant, such as BlendMode
- Description() returns the description string associated with the constant. This is purely for documentation purposes and has no meaning to PaintShop Pro.
- Values() returns a list of tuples describing the possible values of the constant. Each tuple has 3 elements: (name, integer_value, description).

The methods described above are what the ExploreConstants script uses to fill in the list box it shows the user. A simpler example is shown below:

```
# iterate over the constants collection
for EachConstant in App.Constants.All():
    # print the constant name and description
    print '%s - %s' % ( EachConstant.Name(), EachConstant.Description() )

    # now print the values of the constant
    # Values returns a list of tuples of the form (name, integer value, description)
    for Value in EachConstant.Values():
        print '     %s (%d) - %s' % Value

    # put a blank line at the end
    print
```

For Boolean values you can use either the Python values True and False or the constants App.Constants.Boolean.true and App.Constants.Boolean.false.

The Commands collection

App.Commands returns a list of all the command names that are known to PaintShop Pro.
This has little utility in and of itself, but when combined with the GetCommandInfo command makes it possible to write scripts that report information for all of the commands in PaintShop Pro itself. The fragment shown below prints the command name and associated description for all of the commands in PaintShop Pro:

```
def Do(Environment):
    for cmdName in App.Commands:
        try:
            Info = App.Do(Environment, 'GetCommandInfo', { 'TargetCmd': cmdName } )
            # print command name and descriptions
            print '%s: %s' % (cmdName, Info[ 'Description' ] )

        except:
            print >> sys.stderr, 'Error getting info on command ', cmdName
            pass
```

Tips

Writing a PaintShop Pro script is really nothing more than writing a program in Python, and everyone has their favorite tricks and methods. However, if you are new to the game, consider the following pearls of wisdom.

Start by recording

In almost every case, it is simplest to start by recording a script that does some or all of what you want. PaintShop Pro records a command using the proper syntax and specifies all members of the parameter repository.

Even when working on an existing script you should make things easier for yourself – as you need to add a command to an existing script, just record something close to what you want and cut and paste the App.Do calls into your script.

Think in terms of PaintShop Pro operations

If you are not a programmer then this tip is likely obvious; if you are a programmer it may require a different mindset.

Remember that scripting in PaintShop Pro is intended as a way to drive PaintShop Pro from a program. It is not intended as a graphics API or toolkit. When trying to script something, think in terms of how you would accomplish the same task by using PaintShop Pro interactively. The commands available in a script are all things that are available using the menus and tools present in PaintShop Pro.

For example, if you want to paint on a given area of the canvas, think in terms of making a selection and using the flood fill tool, not looking for a command that fills a rectangle.

Use a Python editor

Python source code can be edited with any text editor you like, but you will find it much easier to use a Python-aware editor. Python editors offer syntax coloring, smart indenting, access to Python manuals, and usually an interactive window so that you can try out code as you are writing it. Some even check your code for syntax errors. Python editors can be downloaded from www.python.org.

Use the output window for debugging

You can't use a debugger inside PaintShop Pro, so the output window is your friend. When working on scripts, use print statements to send text to the script output window in PaintShop Pro.

If you send text to the error stream it shows up in red in the output window – this is an easy way to draw attention to any errors. Unless you override it, PaintShop Pro automatically displays the script output window whenever any text is written to the error stream.

To write to the normal output stream just use a regular print statement:

```
print 'Hello world'
```

To write to the error stream use the redirect form (you need to import the sys module as well)

```
print >> sys.stderr, 'Hello world'
```

Just don't forget to clean up your print statements when you are done and everything is working properly.

Use stepping for debugging

PaintShop Pro has the added ability to single step scripts. This option can be found in the script menu under the File menu. When enabled, you are prompted just before the script executes its next PaintShop Pro command. You have the option to “Continue”, “Skip the Command”, or “Stop Scripts”.

Use exception handlers if a command might fail

If a command can’t run, it throws an exception. If not caught, the exception will abort the script. In many cases this is appropriate – if any of the commands in the script fail you may not be able to continue. However, in some cases you should be prepared for failures, and for that you need an exception handler. In Python, exceptions are done with try/except. You put the command that might fail in the try block, and on any error you end up in the exception handler.

A simple example is shown below:

```
def Do(Environment):
      try:
            for x in range( 10 ):     # zoom in 10 steps
                  App.Do( Environment, 'ZoomIn1', {
                        'GeneralSettings': {
                              'ExecutionMode': App.Constants.ExecutionMode.Silent,
                              'AutoActionMode': App.Constants.AutoActionMode.Match,
                              'Version': ((18,0,0),1)
                              }
                        })
      except:
            print 'Max zoom reached'
```

One thing you do have to be aware of is that execution mode affects error handling. If a command fails in interactive mode, PaintShop Pro displays a message box with the failure text, as well as writing the error information to the output window. If a command fails in silent mode, it sends the message to the output window only, without using a message box.

Omit parameters you don’t care about

All parameters of the parameter repository are optional. You can leave out any parameter you want and the last used value will be substituted. Because of this, it is generally not a good idea to omit parameters from the parameter repository, since you can’t predict what values will be used. However, unless you can control the system your script will run on, you should always omit the DialogPlacement section of the GeneralSettings repository. You can’t make any assumptions about the screen layout used by the person running the script, so it is better to leave DialogPlacement unspecified and let windows come up where they want them.

Use ExecutionMode sensibly

The execution mode parameter controls whether a command is done interactively (i.e. with a dialog) or silently (with no UI). There are three relevant values for execution mode:

- Default – use whatever mode was specified in the run script command
- Interactive – display a dialog even if the script is run silently.
- Silent – do not display a dialog, even if the script is run interactively.

PaintShop Pro records all commands as Default, and in many cases this is a good choice–it gives the user the ability to run it quickly and silently, or to run it interactively and fine-tune the settings.

However, in many cases you will find that a number of commands are needed just to get the script working, and the user shouldn't or has no interest in changing the parameters of these commands. Set these commands to silent so that the user is not bothered by dialogs that add nothing to your script.

Script save option – Save Dialog Positions

When a script is saved, one of the checkboxes on the save dialog is called "Save Dialog Positions". If you are writing a script that you intend to share with others, you should not check this option.

PaintShop Pro saves the on-screen position of virtually all dialogs, and displays them in that position on the next run. For standard effect dialogs this also includes whether the window was maximized, the setting of auto-proof, and whether previews were hidden.

All of these settings can be represented in a script, and will be if "Save Dialog Positions" is checked. While on your system you may not notice any difference because of this setting, anyone who runs your script gets the dialog positions from your system. In some cases there may be a reason to do that, but in the absence of a specific reason you should not include dialog positions in a script. Since anything not specified in a script defaults to the last used values, windows always appear in the correct position on the local system.

The dialog position data is stored in the GeneralSettings repository, as shown below:

```
'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Default,
            'DialogPlacement': {
                    'ShowMaximized': False,
                    'Rect': ((1154.000,108.000),402.000,534.000)
                    },
                    'PreviewVisible': True,
                    'AutoProof': True,
                    'AutoActionMode': App.Constants.AutoActionMode.Match,
                    'Version': ((18,0,0),1)
            }
```

Script save option – Save Materials

Another checkbox on the save script dialog is the "Save materials" setting. Materials refer to the settings on the materials palette.

This affects how tools (and only tools) are recorded. Consider a paintbrush. As a paint stroke is made, it uses either or both of the foreground and background materials set in the materials palette. Changes to the materials palette are not recorded; instead the tool records the materials that were used.

Materials have a special attribute however. If a material repository is None (which is a special object in Python), it is replaced with the material currently set on the materials palette.

If "Save materials" is checked when a script is saved, the tool commands include the materials used, and the playback of the script is independent of the materials palette.

If the box is not checked, the tool commands will be recorded with all material repositories set to None.

Consider the following sequence:

1. Select a blue foreground on the materials palette
2. Paint with the left mouse button
3. Select a red foreground on the materials palette
4. Paint with the left mouse button

When you record this operation, the output is 2 paint strokes, one red and one blue.

If recorded with materials saved, when the script is played back it draws one red and one blue stroke, just as it was recorded.

If recorded without materials, when the script is played back it draws two strokes, both in the current foreground material.

By not including materials in a script you can let the user control the behavior by setting the materials palette before the script is run, but you need to ensure that the same set of materials can be used everywhere.

As a script author you can mix the modes – use explicit materials for commands where the user has no need to select the precise material, but set the material to None for anything that you want the user to be able to override. By making judicious use of both foreground and background materials, you can effectively parameterize your script with two different materials.

Script save option – Remove Undone Commands

If you chose to undo any actions while recording a script, you can choose to retain (by not marking) or remove (by marking) the undone commands via the Remove Undone Commands check box located toward the bottom of the Save As dialog. Allowing the script recorder to remove the undone commands can greatly simplify the script. It also allows the script recorder to insert the EnableOptimizedScriptUndo command into the script, which can make the script run much more efficiently. For more info, see *Improving Script Performance.*

Scripts and Threads

Python supports creation of threads, but PaintShop Pro itself is not thread safe. Though a script can use multiple threads, only the first thread in a script is allowed to communicate with PaintShop Pro. This includes accessing anything exposed from PSPApp (certainly including the App.Do method), and also access to stdout and stderr. Since PaintShop Pro captures these streams and redirects them to its own windows, only the initial thread in a script is allowed to use a print statement.

Though threads are nominally supported, the restrictions are severe enough that Corel recommends that you not make use of threads in script files.

Scripts with Dialogs

One of the more powerful features of scripts is that they can contain their own GUI. This makes it possible to write scripts that gather complex data from the user. However, the interaction of a script dialog and PaintShop Pro is rather problematic.

1. Since the dialog is not properly a child of PaintShop Pro, PaintShop Pro is perfectly happy if the dialog ends up behind PaintShop Pro.

2. While the dialog is up the PaintShop Pro UI is not running, so menu items and toolbars don't update properly.

3. If PaintShop Pro attempts to start a new script while a dialog window is running, the Python interpreter will be in an undefined state and the program will crash.

To deal with these issues, PaintShop Pro implements a command called StartForeignWindow. This takes a parameter, which is the window handle of the dialog window. For a Tkinter dialog, the window handle is obtained by calling winfo_id() on the dialog window or one of its parents. In the sample below, assume that ExploreConstants is a class that implements a dialog—the definition for ExploreConstants has also been omitted (available in sample scripts, ExploreConstants.pspscript).

```
# just show the dialog
Wnd = ExploreConstants()
      App.Do( Environment, 'StartForeignWindow', { 'WindowHandle': int(Wnd.winfo_id())
} ) Wnd.mainloop()
App.Do( Environment, 'StartForeignWindow', { 'WindowHandle': 0 } )
```

So long as the window identified in the call to StartForeignWindow exists, PaintShop Pro will ensure that it remains on top of the application, and it will keep the PaintShop Pro UI updating. While a foreign window is running, PaintShop Pro will disable running scripts, loading presets, and loading brush tips.

PaintShop Pro will automatically terminate the foreign window state as soon as the identified window is destroyed. You can also manually terminate the state by calling StartForeignWindow again with a window handle of zero.

Scripts and Batch Processing

The batch processing command in PaintShop Pro gives the ability to apply a script to an arbitrary set of images.

For performance reasons, opening an image in batch processing is not the same as opening an image interactively. No window is created (so the image is never visible), and the script being run does not have access to any other currently open documents.

When run from batch processing, the App.Documents collection will consist only of the document currently being processed, and App.ActiveDocument and App.TargetDocument will be the same.

Most users will never be aware of these restrictions, but if there is a need to access multiple files during a batch operation, the logic to open and save the files will need to be moved from the batch process dialog to the script.

See the section on *Processing all the files in a directory* for more information.

Using Scriptlets

The command line for PaintShop Pro supports the ability to drop images onto a shortcut (referred to as a "Scriptlet") that calls PaintShop Pro and a script to be applied to the dropped images.

The procedure for making a PaintShop Pro Scriptlet is:

1. Create a desktop shortcut.
2. Right-click the shortcut icon, and choose Properties.
3. In the Shortcut tab’s Target field (shown below), enter the path for PaintShop Pro, followed by the switch /Script, and then enter the path for the desired script you want executed against the image(s) you drop onto the shortcut.

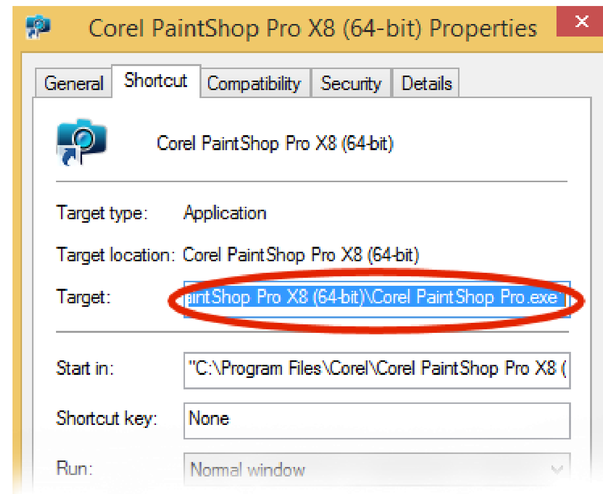

Figure 2

Here is an example of what you could enter into the Target field:

```
"C:\Program Files\Corel\Corel PaintShop Pro X8 (64-bit)\Corel PaintShop Pro.exe" /Script.
"C:\Users\<InsertYourNameHere>\My Documents\Corel PaintShop Pro\18.0\Scripts-
Restricted\<ScriptName>.PSPScript"
```

(where <InsertYourNameHere> represents your login name, and <ScriptName> represents the name of the script you want the shortcut to run.)

When you drop image files on the shortcut, PaintShop Pro opens and runs the specified script on the files.

Scriptlet Limitations

Be aware of the following limitations when using Scriptlets:

- Due to Microsoft Windows command line length limitations, there is a limit to the number of files you can drop on the Scriptlet. The limit is approximately 22 files if they have 8.3 format filenames, less if they have long file names.
- A workaround to this limitation is to drop a folder of files onto the Scriptlet, which will open and process all of the files in the folder. Be aware, however, that PaintShop Pro will open all the files at once, and then process them. Thus if you do not want a large number of files opened all at once in PaintShop Pro, don't drop a folder containing many images onto a Scriptlet.
- Folders are only processed one level deep. Folders within folders will not be processed.
- The script is responsible for saving and closing the files or you can do it manually after they've all been processed.

Improving Script Performance

From an undo perspective, PaintShop Pro treats a script as an atomic operation (all script commands are consolidated) – a single undo will undo all of the operations done by the script.

Internally however, a script is a sequence of commands, each of which manages its own undo data. For example, a blur command has to save the state of the data before it is blurred, so it saves a copy of the data it is about to modify. If a script does 10 operations, each of those operations might save a copy of the active layer for undo purposes.

But when the script is undone it is always undone as a whole, so only the initial state of the image is significant. Hence the undo data of all the intervening commands is irrelevant and needn't be saved.

But like just about everything else in PaintShop Pro, undo is a scriptable operation. If a script includes undo commands as part of its processing, all of the intermediate undo data must be preserved.

When a script is recorded, PaintShop Pro will check to see if the script contains any embedded

undo operations. If it does not, it will insert a call to 'EnableOptimizedScriptUndo' at the start of the script.

With optimized undo on, the amount of data that has to be saved during the execution of the script can be (possibly dramatically) reduced, improving the performance and decreasing the footprint of the script.

When hand writing a script, you can add the call to EnableOptimizedScriptUndo so long as your script does not perform any undo commands, (UndoLastCmd or SelectiveUndo).

```
# EnableOptimizedScriptUndo
App.Do( Environment, 'EnableOptimizedScriptUndo' )
```

Optimized script undo can be turned on at any point, but you should do it at the beginning of the script. Optimized script undo is automatically turned off at the end of the script.

Common operations

A number of operations come up over and over, and are worthy of dedicated descriptions. For any given task there are likely to be a number of possible solutions; if you don't care for the methods described here feel free to roll your own.

Selecting a layer

Many operations in PaintShop Pro will involve selecting a layer. When you do this task interactively it is simple – see the layer you want and click on it in the layer palette.

From a script this is a more complicated operation, since the script needs to define the layer to be selected in a logical way, one that is portable across different documents.

Most commands operate on the active layer so do not need to specify the target layer in any way. However, commands that operate on an arbitrary layer need a means of specifying the target. Layer selection is the most obvious example of this – to select a new layer we have to have a means of specifying which layer to select.

Layer names are both useless and undesirable – useless because they do not have to be unique, and undesirable since it should be possible to run a script that selects a layer without having a dependency on the layer names in the image.

The answer is a layer select tuple. A Python tuple is simply an ordered set of values. Syntactically it is represented by a set of values separated by commas and enclosed in parentheses.

The layer select tuple describes how to get from the current layer to any other layer in the document.

- The first element is how many levels to go out in the layer hierarchy. A value of 1 means to go to the parent's level, a value of 2 to the grandparent level, etc.
- The second element is the number of siblings to traverse. Positive numbers go up in the z-order, negative numbers go down in the z-order
- The third element is a list of children to traverse. A list is a variable length set of ordered elements separated by commas and enclosed in brackets.
- The fourth element is a boolean that specifies whether strict or loose interpretation of the tuple should be used. If true, the tuple must be specified exactly or an error will result. If false, the tuple will be followed as closely as possible, and an error will be returned only if a non-empty tuple resulted in navigating to the active layer.

Everything is numbered with the bottom of the z-order as 0, and counts up in the Layers palette. This means that selecting the first thing in a group actually selects the bottommost element.

The logic is rather torturous, so perhaps some examples will help. Consider the following layer arrangement:

Using this arrangement, how do we navigate from the active layer (Vector 1) to the layer named 'Top'?

Think of layer navigation in terms of out/in, and up/down. The current layer of 'Vector 1' is at

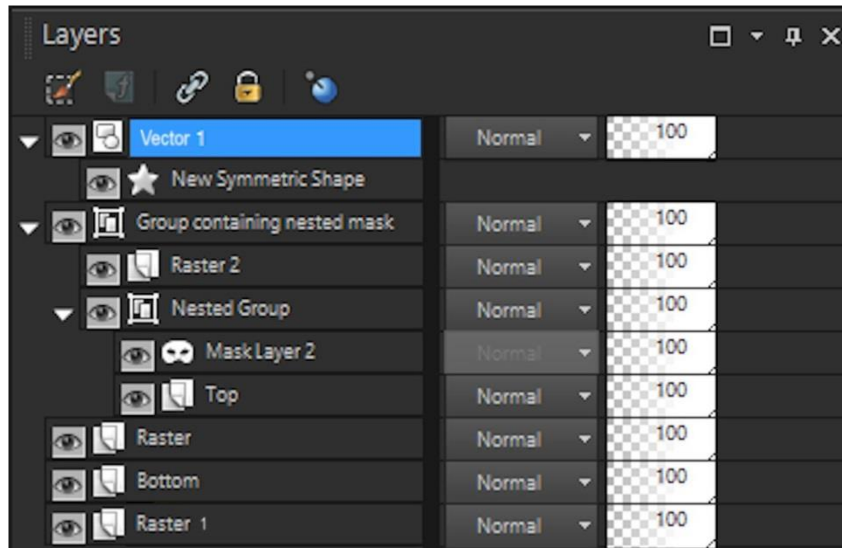

Figure 3

the same level as the layer group named 'Group containing nested mask', so the first element of the tuple is zero. Now we must go down to 'Group containing nested mask'. This is one layer below 'Vector 1', so the second parameter is -1.

Now we need to go in to find layer 'Top'. First, we have to get to 'Nested Group', which is the first (bottommost) child of 'Group containing nested mask'. From there, we need to get to its first child, 'Top'. Therefore the third element of the tuple is [1,1].

In this particular case, the strict parameter doesn't matter because we have defined the precise path we want. When recording scripts, PaintShop Pro always sets the strict parameter to false, so we will assume that. The resulting layer select tuple is then **(0, -1, [1,1], False)**.

Now assume that the active layer is 'Top' and that we want to navigate to the layer named 'Bottom'.

To get from 'Top' to 'Bottom', we have to first find a parent of 'Top' that is a sibling of 'Bottom' (or a sibling to one of Bottom's ancestors). That is the graphic, so we go out 2 levels. Now we are at 'Group containing nested mask', which is two layers above 'Bottom', so the up/down parameter is -2. We don't need to go into any group, so the in parameter is empty. The resulting layer select tuple is then **(2, -2, [], False)**.

The strict parameter is useful in creating a select tuple that should always select a logical element, regardless of what the layer configuration of the document is. It also permits a script

to run without error even if the layer configuration is not an exact match of the configuration that existed when the script was recorded.

To select the bottommost layer	`(9999, -9999, [], False)`
To select the topmost layer	`(9999, 9999, [], False)`
To select the topmost element of the current group	`(0, 9999, [], False)`
To select the topmost child of the current layer	`(0, 0, [9999], False)`
To select the next layer up	`(0, 1, [], False)`
To select the next layer down	`(0, -1, [], False)`
To select the parent layer	`(1, 0, [], False)`

In non-strict mode, it will process as much of the tuple as it can. Attempting to go out 9999 levels in the layer hierarchy will reach the graphic (the container of all the layers). PaintShop Pro can't go any further in that direction and moves on to the next element.

Processing an over element of 9999 will reach either the top or the bottom of the group, so again it stops and moves on to the next element.

Select multiple layers and objects

As of PaintShop Pro X4, you can select and apply actions to multiple layers and objects. The existing command "VectorSelectionUpdate" has been extended to support this scenario. Here is the command syntax:

```
# Vector Selection Update
App.Do( Environment, 'VectorSelectionUpdate', {
      'Path': (0,1,[],False),
      'Type': App.Constants.ObjectSelection.AddToSelection,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Silent,
            'AutoActionMode': App.Constants.AutoActionMode.Default,
            'Version': ((18,0,0),1)
            }
      })
```

The "Path" field, which serves as a layer select tuple, has the same definition as it has in the "SelectLayer" command. The "Type" field has two kinds of values:

- "App.Constants.ObjectSelection.Select": discard all layer/object selection, and select the specified layer/object.

- "App.Constants.ObjectSelection.AddToSelection": add specified layer/object to current selection.

Take Figure 3 as an example. If the currently selected layer is "Vector 1", and we want to select multiple layers, "Bottom", "Mask Layer 2" and "Raster 2", the script would be:

```
# Vector Selection Update
App.Do( Environment, 'VectorSelectionUpdate', {
      'Path': (0,-3,[],False),
      'Type': App.Constants.ObjectSelection.Select,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Silent,
            'AutoActionMode': App.Constants.AutoActionMode.Default,
            'Version': ((18,0,0),1)
            }
      })

# Vector Selection Update
App.Do( Environment, 'VectorSelectionUpdate', {
      'Path': (0,-1,[2],False),
      'Type': App.Constants.ObjectSelection.AddToSelection,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Silent,
            'AutoActionMode': App.Constants.AutoActionMode.Default,
            'Version': ((18,0,0),1)
            }
      })


# Vector Selection Update
App.Do( Environment, 'VectorSelectionUpdate', {
      'Path': (0,2,[2],False),
      'Type': App.Constants.ObjectSelection.AddToSelection,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Silent,
            'AutoActionMode': App.Constants.AutoActionMode.Default,
            'Version': ((18,0,0),1)
            }
      })
```

The first command clears the selection and adds a new layer. The two commands that follow add two more layers to this selection.

And, if current layer is "Vector 1", and we want to select "New Symmetric Shape", "Raster", and "Raster 1", the script would be:

```
# Vector Selection Update
App.Do( Environment, 'VectorSelectionUpdate', {
      'Path': (0,0,[1],False),
      'Type': App.Constants.ObjectSelection.Select,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Silent,
            'AutoActionMode': App.Constants.AutoActionMode.Default,
```

```
                'Version': ((18,0,0),1)
                }
        })

    # Vector Selection Update
    App.Do( Environment, 'VectorSelectionUpdate', {
        'Path': (0,-2,[],False),
        'Type': App.Constants.ObjectSelection.AddToSelection,
        'GeneralSettings': {
                'ExecutionMode': App.Constants.ExecutionMode.Silent,
                'AutoActionMode': App.Constants.AutoActionMode.Default,
                'Version': ((18,0,0),1)
                }
        })

    # Vector Selection Update
    App.Do( Environment, 'VectorSelectionUpdate', {
        'Path': (0,-2,[],False),
        'Type': App.Constants.ObjectSelection.AddToSelection,
        'GeneralSettings': {
                'ExecutionMode': App.Constants.ExecutionMode.Silent,
                'AutoActionMode': App.Constants.AutoActionMode.Default,
                'Version': ((18,0,0),1)
                }
        })
```

The first command clears the selection and selects the only child of "Vector Layer 1". The other two commands select two more raster layers.

Note: Even in multiple selection scenarios, there is still an "active layer" concept. This is to allow for backward compatibility. When multiple layers/objects are selected, the last selected layer, or the parent layer containing the last selected object, is treated as the "active layer". With this definition, all legacy scripts depend on a "single selected layer" or they need an "anchor" layer to work.

Iterating layers

The layer select tuple provides a means of selecting any layer in an image, but it does not help much with the problem of doing something to every layer of a document.

For scripts, two dedicated commands are provided for iterating the layers:

- SelectNextLayer goes up in the z-order (towards the top of the layer palette).
- SelectPreviousLayer goes down in the z-order (towards the bottom of the layer palette).

These commands return a Boolean value–true if they successfully selected a new layer, false if there are no more layers in the indicated direction.

Layer groups are also layers, and are included in the traversal by SelectNextLayer and SelectPreviousLayer. When traversing groups, the group layer itself is visited first, then its

lowest child next. So for the layer arrangement shown below, the numbers to the right list the order visited (assuming SelectNextLayer).

Vector 1	10
Group 1	4
Raster 1	9
Group 2	6
Mask 1	8
Raster 2	7
Vector 2	5
Mask 1	3
Raster 3	2
Background	1

The following code fragment will save the current layer position, then iterate all layers from the bottom up, and finally restore the active layer.

```
def Do(Environment):
      # get the path from the bottom from the active layer so we can restore it when done
      Props = App.Do( Environment, 'ReturnLayerProperties' )
      PathBack = Props[ 'Path' ]

      # start by selecting the bottommost layer in the image.
      App.Do( Environment, 'SelectLayer',
            { 'Path': (9999,-9999, [], False ) } )

      FoundLayer = True
      while FoundLayer == True:

            # *** Insert useful code here

            # go to the next layer
            FoundLayer = App.Do( Environment, 'SelectNextLayer' )

      # now that the loop is done, select the bottom layer and then
      # use the pathback to restore the layer that was active when we started
      App.Do( Environment, 'SelectLayer',
            { 'Path': (9999,-9999, [], False ) } )
      App.Do( Environment, 'SelectLayer',
            { 'Path': PathBack } )
```

Selecting a different document

The select document command is used to switch between documents. It takes a parameter that selects a document relative to the target document.

Documents are referenced in the order in which they were opened. Suppose you have four opened documents, which were opened in the order A, B, C, D. Assume that the target document is C.

To get to document B we need to select the previously opened document. This is always document -1 (remember that document selection is relative):

```
App.Do( Environment, 'SelectDocument', {
      'SelectedImage': -1,
      'Strict': False,
      'GeneralSettings': {
            'ExecutionMode': App.Constants.ExecutionMode.Default,
            'AutoActionMode': App.Constants.AutoActionMode.Match,
            'Version': ((18,0,0),1)
            }
      })
```

Likewise document C to D would be 1, and C to A would be -2.

Like layer selection, document selection supports a strict parameter, and it is interpreted the same way – if strict is false the SelectedImage parameter can be out of range and the command will stop when it reaches the first or last document.

Scripts in PaintShop Pro have the notion of a target document (always available as App.TargetDocument). The target document is set when the script starts, and won't change unless a call to SelectDocument is used.

When recording a script, PaintShop Pro will include calls to SelectDocument with a 'SelectedImage' parameter of 0. This simply sets the target document to match the currently active document.

Iterating documents

PaintShop Pro maintains a collection of all the open documents, called App.Documents. This is a Python list object, and can be iterated like any list. To do something to all open documents is as simple as:

```
for Doc in App.Documents:
      App.Do( Environment, 'Flip', {}, Doc )
```

This makes use of the fact that the fourth parameter to App.Do is the target document to use for the command.

Saving/Restoring a selection

There is a class in PSPUtils.py that can be used to save a selection to disk and restore it later. Using it is quite simple:

```
import PSPUtils

# backup a selection if one exists
SelSaver = PSPUtils.SaveSelection( Environment, App.TargetDocument )

# do something

SelSaver.RestoreSelection()
```

PSPUtils can be used in restricted scripts.

The SaveSelection class just saves the current selection to disk and does a select none. When RestoreSelection is called, it just checks to see SaveSelection actually did a save operation, and if so restores it.

The save selection class uses a fixed filename (__$TempSavedSelection$__), so you can't have multiple selections saved in this manner. The document whose selection will be saved is specified in the call to SaveSelection.

If given a floating selection, the SaveSelection function will promote it to a layer before removing the selection.

Checking for a selection

Some scripts won't work properly with a selection, and may not be able to use the SaveSelection class to back it up (for instance doing a deform or a resize makes saving a selection rather pointless).

The GetRasterSelectionRect command will return information on the position type of selection. The following fragment tests for the presence of a raster selection and gives the user the opportunity to cancel the script if one is present.

```
    # We use a selection, meaning that any existing selection will be destroyed.
    # ask the user if this is OK to do.
    SelectionInfo = App.Do( Environment, 'GetRasterSelectionRect' )
    if SelectionInfo[ 'Type' ] != App.Constants.SelectionType.None:
        Result = App.Do(Environment, 'MsgBox', {
            'Buttons': App.Constants.MsgButtons.YesNo,
            'Icon': App.Constants.MsgIcons.Question,
            'Text': 'This script will remove any existing selection. OK?',
            })
        if Result == 0: # they pressed no in the message box, so abort
            return
        else:
            App.Do( Environment, 'SelectNone' ) # get the selection out of the way now
```

Testing Image/Layer Attributes

The module PSPUtils.py implements a number of utility functions for checking the type of layer or image. These are:

Utility function	Information returned
IsFlatImage(Environment, Doc)	Returns true if image consists of a single background layer.
IsPaletted(Environment, Doc)	Returns true if image is a paletted format. This function does not consider greyscale to be paletted.
IsTrueColor(Environment, Doc)	Returns true if the image is 16 million colors.
IsGreyScale(Environment, Doc)	Returns true if the image is greyscale.
LayerIsArtMedia(Environment, Doc)	Returns true if the active layer is an Art media layer.
LayerIsRaster(Environment, Doc)	Returns true if the active layer is a raster layer. This is a strict test. Even though adjustment layers, floating selections and mask layers contain raster data, they are not raster layers.
LayerIsVector(Environment, Doc)	Returns true if the active layer is a vector layer.
LayerIsBackground(Environment, Doc)	Returns true if the current layer is the background layer.
GetLayerCount(Environment, Doc)	Returns the number of layers in the image.
GetCurrentLayerName(Environment, Doc)	Returns the name of the current layer.

All of these functions are passed a document parameter. To use the target document simply pass App.TargetDocument as the second parameter.

Requiring a Document

Most scripts will require that a document be open when they are run. Since this is such a common requirement, a test for it is implemented in PSPUtils.

```
import PSPUtils

if PSPUtils.RequireADoc( Environment ) == False:
      return
```

If no document is open when RequireADoc is called, it will display a message box telling the user that a document is required, and will return false. Otherwise it will just return true.

Promoting To True Color

Many scripts are going to need to work with true color images. PSPUtils implements a method for testing the image type and automatically doing a promotion if necessary. True color and greyscale images are left alone, paletted images are promoted to true color. Calling it is simple:

```
import PSPUtils

PSPUtils.PromoteToTrueColor( Environment, App.TargetDocument )
```

Processing all the files in a directory

The short answer on this is to use the batch conversion functions on the file menu – that dialog lets you select an arbitrary collection of images, and run a script before saving them.

However, there may be cases where you want to do operations on multiple files inside of a single script. Beware that this will not work from a restricted path, and since PaintShop Pro has to actually open each document and create a window for it this will likely run slower than the batch conversion method.

Iterating a directory is just straight Python code, but it comes up often enough that it is worth a quick sample. The following script iterates all files matching a particular pattern, opens each one, and then calls a function that you can change to do whatever processing you like. The following example is available in the collection of sample scripts:

```
from PSPApp import *
import os.path
import fnmatch
import sys
import glob


# Template script to process files in a given directory tree. You can specify
# the tree to search, the files to open, and whether they should be saved or not.
# The ProcessFile method is called for each file opened, and then the file is
# optionally saved
#
# Areas you are expected to modify are marked with ***


# *** Set this to the directory tree you want to process
DirectoryToUse = 'C:\\WorkFiles'

# *** Include all of the extensions or patterns you care about - or just make it *.*
DirectorySearchString = [ '*.jpg', '*.pspimage', '*.png' ]

# *** Set this to true to search subdirectories of DirectoryToUse
```

```
# set it to false to search only in the specified directory
#SearchSubDirectories = False
SearchSubDirectories = True

# *** Set this value to true to save the file before closing it. If false, the file
# is not saved, meaning that the ProcessFile method is responsible for doing something
# with the files
#SaveFileAfterProcessing = False
SaveFilesAfterProcessing = True

def ScriptProperties():
      return {
            'Author': 'Corel Corporation',
            'Copyright': 'Copyright 2015 Corel Corporation, all rights reserved.',
            'Description': 'Open and process an entire directory of files.',
            'Host': 'PaintShop Pro'
            'Host Version': '18.00'
            }

def ProcessFile( Environment, FileName ):
      ' *** Process a single file - put your code here'
      print 'Processing File %s' % FileName
      App.Do( Environment, 'Flip' )


def SearchDir( OutList, CurrentDir, FilesInDir ):
      ''' Called by os.path.walk for each directory we encounter. Gets passed the
            list of files we are building up (OutList), the directory we are visiting
            (CurrentDir), and a list of all the filenames (without path) in the current
            directory. We need to strip out anything that is a directory or doesn't
            match our search string.
      '''
      for File in FilesInDir:
            for SearchPattern in DirectorySearchString:
                  if fnmatch.fnmatch( File, SearchPattern ):
                        FullPath = os.path.join( CurrentDir, File )
                        # make sure what we have is a file and not a subdirectory
                        if not os.path.isdir( FullPath ):
                              OutList.append( FullPath )


def Do(Environment):
      # iterate through the search strings and glob all the results into one big list
      CandidateFiles = []     # this will be a list of all the files we find

      if SearchSubDirectories == True:
            # walk the directory tree, calling SearchDir on each directory visited.
            os.path.walk( DirectoryToUse, SearchDir, CandidateFiles )
      else:
            # just glob the path provided rather than walk a tree
            for Search in DirectorySearchString:
                  # concatenate the dir and file spec together
                  SearchPath = os.path.join( DirectoryToUse, Search )
                  GlobbedFiles = glob.glob( SearchPath ) # glob will return a list of files
                  for File in GlobbedFiles:  # now iterate the list
```

```
                        CandidateFiles.append( File )       # and add each one to candidate
list

      # going to have a problem if there aren't any files.
      if len(CandidateFiles) == 0:
            print >>sys.stderr, "No files found to process"
            return

      # now process the list of files
      for File in CandidateFiles:
            print 'Opening file ', File
            # open the file
            try:
                  App.Do( Environment, 'FileOpen', {
                        'FileList': [ File ],
                        'GeneralSettings': {
                              'ExecutionMode': App.Constants.ExecutionMode.Silent,
                              'AutoActionMode': App.Constants.AutoActionMode.Match,
                              'Version': ((18,0,0),1)
                              }
                        })

                  # set the newly opened doc as the TargetDocument
                  App.Do( Environment, 'SelectDocument', {
                        'SelectedImage': 0,
                        'Strict': True,
                        'GeneralSettings': {
                              'ExecutionMode': App.Constants.ExecutionMode.Default,
                              'AutoActionMode': App.Constants.AutoActionMode.Match
                              }
                        })
            except:
                  print >>sys.stderr, 'Error opening file %s - skipping' % File

            # for simplicity, place your code in the ProcessFile method
            ProcessFile( Environment, File )

            # save the file
            try:
                  if SaveFilesAfterProcessing == True:
                        App.Do( Environment, 'FileSave' )
                  # close the file before going on to the next
                  App.Do( Environment, 'FileClose' )
            except:
                  print >>sys.stderr, 'Error on save/close of file %s - aborting' % File
```

Getting information from PaintShop Pro

Scripts that do nothing but execute an unconditional sequence of commands can be done with the script recorder. However, many operations are going to require that a script have access to the current data in a document – color depth, presence of a selection, number of layers, type of layers, etc.

The members of the Python document object such as Width and Height only scratch the surface of this information, but there are dedicated commands for returning data from the application or a document.

The following commands can be used to return data about an image to a script. This allows scripts to make decisions based on data found in the document. These commands are all described in the "Commands for Scripting" section:

- ReturnImageInfo
- ReturnLayerProperties
- GetRasterSelectionRect
- ReturnGeneralPreferences
- ReturnFileLocations
- ReturnVectorObjectProperties

Commands for Scripting

The script recorder captures most everything done interactively to PaintShop Pro, and the simplest method to create scripts is to record something close to what you want and then modify it as needed.

In addition to the commands that are accessible through the script recorder, there are a number of commands that are only available via a script, and exist to make scripts easier to write. These are detailed below.

Command: SelectNextLayer

Parameters: None

This command returns a Boolean value. If a new layer is selected, it returns true. If there is no next layer it returns false.

Sample Script

```
# iterate all layers of the current document and print out their layer
# name and opacity

def Do(Environment):
      # start by selecting the bottommost layer in the image.      This needs to be
      # in a try block because if we are already on the bottomost layer we will
      # throw an error
      try:
            App.Do( Environment, 'SelectLayer',
                  { 'Path': (9999,-9999, [], False ) } )
      except:
```

```
        pass  # error is OK - just ignore and keep going

    LayerNum = 1

    # now iterate all of the layers in the document with SelectNextLAyer
    FoundLayer = True
    while FoundLayer == True:
        Props = App.Do( Environment, 'ReturnLayerProperties' )
        print '%3d: %s (%d%% Opacity)' % (LayerNum,
              Props[ 'General' ][ 'Name' ],
              Props[ 'General' ][ 'Opacity' ])
        LayerNum += 1

        FoundLayer = App.Do( Environment, 'SelectNextLayer' )
```

Description

Select the next layer up in the z-order. This command treats the layer hierarchy as if it were flat, so that all layers can be traversed in a single loop.

When traversing groups, the group layer itself is visited first, then its lowest child next. So for the layer arrangement shown below, the numbers to the right list the order visited

Vector Layer 1	10
Group Layer 1	4
Raster Layer 1	9
Group Layer 2	6
Mask Layer 1	8
Raster Layer 2	7
Vector Layer 2	5
Mask Layer 1	3
Raster Layer 3	2
Background	1

Command: SelectPreviousLayer

Parameters: None

This command returns a Boolean value. If a new layer is selected, it returns true. If there is no previous layer, it returns false.

Sample Script

```
FoundLayer = App.Do( Environment, 'SelectPreviousLayer' )
```

Description

This command is identical to SelectNextLayer except that it hunts down in the z-order instead of up.

Command: GetNextObject

Parameters: None

Sample Script

```
Result = App.Do( Environment, 'GetNextObject' )
if Result['HaveObject'] == 0 :
      print 'We do not have a next object'
```

Description

Used to get the next selection object on a vector layer. If there is no object selected on the vector layer it will get the first object created on the layer.

Command: GetPrevObject

Parameters: None

Sample Script

```
Result = App.Do( Environment, 'GetPrevObject' )
if Result['HaveObject'] == 1 :
      print 'We have a prev object'
```

Description

Gets the previous object on the vector layer. If there is no object selected on the vector layer it will get the most recently added object.

Command: HasVectorSelection

Parameters: None

The return value is a Boolean – true if a vector selection exists, false otherwise.

Sample Script

```
Result = App.Do( Environment, 'HasVectorSelection' )
if Result == True:
     print 'Found a vector selection'
```

Description

Returns whether or not a vector selection exists.

Command: GetVectorSelectionRect

Parameters: None

Sample Script

```
Result = App.Do( Environment, 'GetVectorSelectionRect' )
if Result['HasVectorRect'] :
     print 'Selection rect is ', Result[ 'VectorRect' ]
```

Description

Gets the current vector selection rectangle if one exists.

Command: GetRasterSelectionRect

Parameters: None

The return value of the command is a dictionary with two elements:

- Type – the type of the selection
- Rect – if a selection exists, the bounding box of the selection in document coordinates

Sample Script

```
Result = App.Do( Environment, 'GetRasterSelectionRect', {} )
if Result['Type'] == App.Constants.SelectionType.Floating:
      print 'Document has a floating selection'
      print 'Selection rect is ', Result['Rect']
elif Result['Type'] == App.Constants.SelectionType.NonFloating:
      print 'Document has a non-floating selection'
      print 'Selection rect is ', Result['Rect']
else: # Result == App.Constants.SelectionType.None:
      print 'Document has no selection'
```

Description

This command retrieves information about the type of raster selection (floating, non- floating or none), and if a selection exists it returns the bounding box of the selection.

If there is no selection an empty rectangle is returned.

The rectangle is defined as a tuple consisting of ((top, left), width, height).

Command: SaveMaterialSwatch

Parameters:

- SwatchMaterial - the material to use for the swatch
- SwatchName – a string to use for the swatch name

Sample Script

```
App.Do( Environment, 'SaveMaterialSwatch', {
      'SwatchMaterial': {
            'Color': (0,0,0),
            'Pattern': None,
            'Gradient': None,
            'Texture': None
            },
      'SwatchName': 'foo'
      })
```

Description

This command saves a material to a swatch file. If no material is supplied, the current foreground material is used. If executed interactively, the user is prompted for a swatch name. The above script will create a swatch file named "foo" that contains a black solid color.

Command: SetMaterial

Parameters:

- IsPrimary - Boolean that determines if we are setting the foreground/stroke material, or the background/fill material.

- NewMaterial - Material to be used

Sample Script

```
App.Do( Environment, 'SetMaterial', {
      'IsPrimary': True,
      'NewMaterial': {
            'Color': (0,0,0),
            'Pattern': None,
            'Gradient': None,
            'Texture': None
            }
      })
```

Description

Set the specified material into the material palette. In the above example, the foreground/stroke material would get set to solid black.

Command: SetMaterialStyle

Parameters:

- Style - an enumeration that specifies the material style to use, can currently be Color, Pattern or Gradient
- IsPrimary - a boolean value that determines if the command is to be applied to the foreground/stroke material or the background/fill material

Sample Script

```
App.Do( Environment, 'SetMaterialStyle', {
      'Style': App.Constants.MaterialStyle.Color,
      'IsPrimary': True
      })
```

Description

Set the style type of the specified material in the material palette. The material palette uses the values that were previously used for that particular style.

Command: ShowSwatchView

Parameters:

- Show - Boolean value, if true, the swatch view is shown on the material palette, otherwise the rainbow tab is shown

Sample Script

```
App.Do( Environment, 'ShowSwatchView', {
        'Show': True
        })
```

Description

Simple command that toggles the visibility of the swatch tab on the material palette.

Command: EnableMaterialTexture

Parameters:

- IsEnabled - Boolean that determines if the texture is on or off
- IsPrimary - Boolean that determines if this command is to operate on the primary (foreground/stroke) material or the secondary (background/fill)

Sample Script

```
App.Do( Environment, 'EnableMaterialTexture', {
      'IsEnabled': True,
      'IsPrimary': True
      })
```

Description

Simple command that turns the texture on or off for the primary or secondary material in the material palette.

Command: GetMaterial

Parameters:

- IsPrimary - Boolean that determines which material is to be retrieved; the primary (foreground/stroke) or the secondary (background/fill).

Sample Script

```
Material = App.Do( Environment, 'GetMaterial', {
      'IsPrimary': True
      })
```

Description

This command retrieves the current primary or secondary material from the material palette.

If the execution mode is silent, the current material is retrieved. If the execution mode is interactive then the material picker is brought up, and the user can select a new material to use.

Command: GetVersionInfo

Parameters: None

Sample Script

```
def Do(Environment):
      Version = App.Do(Environment, 'GetVersionInfo')
      for key in Version:
            print '%s - %s' % (key, Version[key])
```

Description

This command returns a dictionary containing version information from the program.

The version is given both as a string that contains the version, build type, and build number, as well as individual components. The version number is given in 3 components, the major, minor and micro version. For a version of 18.01, the major number is 18, the minor version is 0, and the micro version is 1.

Command: ReturnImageInfo

Parameters: None

Sample Script

```
def Do(Environment):
      ImageInfo = App.Do( Environment, 'ReturnImageInfo',)
      for key in ImageInfo:
            print '%-30s:     %s' % (key, ImageInfo[key])
```

Description

This command will return the same information about the current image that is displayed in the "Current Image Information" dialog inside the application. This includes both basic image information such as file name, width, height, and EXIF data. The following keys/data will be returned:

```
'FileName', 'Width', 'Height', 'PixelsPerUnit', 'Unit', 'BitDepth', 'Modified',
'Selection', 'LayerNum', 'AlphaNum', 'UndoMem', 'UndoDisk', 'MemoryUsage', 'Title',
'ArtistName', 'CopyrightHolder', 'Description', 'CreatedDate', 'ModifedDate', 'ExifMake',
'ExifModel', 'ExifSoftware', 'ExifVersion', 'ExifFlashVersion', 'ExifCameraNotes',
'ExifDateTime', 'ExifSubSecond', 'ExifOriginalDate', 'ExifOriginalSubSecond',
'ExifDigitizedDate', 'ExifDigitizedSubSecond', 'ExifExposureProgram', 'ExifSceneType',
'ExifExposureIndex', 'ExifExposureBias', 'ExifBrightness', 'ExifExposureTime',
'ExifFNumber', 'ExifLensAperture', 'ExifMaxAperture', 'ExifShutterSpeed',
'ExifFocalLength', 'ExifFlash', 'ExifFlashEnergy', 'ExifFocalPlaneWidth',
'ExifFocalPlaneHeight', 'ExifISOSpeed', 'ExifOECF', 'ExifSpectralFrequencyTable',
'ExifLightSource', 'ExifMeterMode', 'ExifSensorType', 'ExifImageSource',
'ExifCFAPattern', 'ExifArtistName', 'ExifCopyright', 'ExifComments', 'ExifTitle',
'ExifWidth', 'ExifHeight', 'ExifSamplesPerPixel', 'ExifPlanerConfiguration',
'ExifOrientation', 'ExifPhotometricInterpretation', 'ExifXResolution', 'ExifYResolution',
'ExifResolutionUnit', 'ExifPixelWidth', 'ExifPixelHeight', 'ExifComponentConfig',
'ExifCompressedBPP', 'ExifColorSpace', 'ExifReferenceBlackWhite', 'ExifWhitePoint',
'ExifPrimaryChromaticities', 'ExifYCbCrProperties', 'ExifSoundFile',
'ExifSpectralSensitivity', 'ExifJPEGBytesSize', 'ExifJPEGOffset', 'ExifCompression',
'ExifFocalPlaneResUnit', 'ExifGPSVersion', 'ExifLatitude', 'ExifLatitudeRef',
'ExifLongitude', 'ExifLongitudeRef', 'ExifAltitude', 'ExifAltitudeRef', 'ExifGPSTime',
'ExifGPSSatellites', 'ExifGPSStatus', 'ExifGPSMeasureMode', 'ExifGPSDegreeOfPrecision',
'ExifGPSSpeed', 'ExifGPSSpeedRef', 'ExifGPSDirectionOfGPS', 'ExifGPSDirectionOfGPSRef',
'ExifGPSImageDirection', 'ExifGPSImageDirectionRef', 'ExifGPSGeodeticSurveyData',
'ExifGPSDestinationLatitude', 'ExifGPSDestinationLatitudeRef',
'ExifGPSDestinationLongitude', 'ExifGPSDestinationLongitudeRef', 'ExifGPSDestinationPt',
'ExifGPSDestinationPtRef', 'ExifGPSDestinationDistance', 'ExifGPSDestinationDistanceRef'
```

In addition to the data described above, which is all data that duplicates the contents of the Image Info dialog, the following data is returned:

- 'PaletteColorList' – a list of color values that describe the palette of the image. For true color images this list will be empty.
- 'AlphaChannelList' – list of strings containing the name of each alpha channel in the image

- 'PixelFormat' – an enumeration describing the canvas pixel format. Possible values are: Grey, GreyA, BGR, BGRA, Index1, Index4, and Index8.

Command: ReturnFileLocations

Parameters: None

Sample Script

```
ListOfDirectories = ['Temp', 'Tubes', 'Patterns', 'Textures', 'Gradients', 'Brushes',
            'Frame', 'Shape', 'StyledLines', 'Plugins', 'RestrictedScripts',
            'TrustedScripts', 'Presets', 'Swatches', 'Cache',
            'Masks', 'Selections', 'CMYKProfiles', 'PrintTemplates',
            'Workspaces', 'Palettes', 'DeformationMaps',
            'EnvironmentMaps', 'BumpMaps', 'PythonSourceEditor',
            'DisplacementMaps', 'CategoryDatabase', 'MixerPages' ]

DirInfo = App.Do( Environment, 'ReturnFileLocations',)

# Print out all the directories for each location
for DirName in ListOfDirectories:
      DirCntName = DirName + 'Num'
      DirCnt = DirInfo[ DirCntName ]
      print "\nNumber of directories defined for '", DirName, "' is ", DirCnt
      Cnt = 0
      while Cnt < DirCnt:
            print "     ", DirInfo[ DirName ][ Cnt ]
            Cnt = Cnt + 1
```

Description

This command will return all defined file location information. The file locations are defined in the application with the file location dialog under preferences. This command simply allows scripts to access that information in a read only fashion. The number and full paths of each type of directory is returned (0 based). For example: If there are 3 plug-in directories defined then 'PluginsNum' will equal 3, and 'Plugins' 0 to 2 will be valid. Referring to the sample code above the following code would print the remaining plug-in directories:

```
print 'Plugins Dir 1: ', ImageInfo[ 'Plugins' ][1]
print 'Plugins Dir 2: ', ImageInfo[ 'Plugins' ][2]
```

Note: The first directory returned for each type is the default write directory for that type.

Command: ReturnLayerProperties

Parameters: None

It does have a large return value. It returns a dictionary, which is all of the repositories from the LayerProperties command, plus three additional values:

- **Path** – a layer select tuple to select this layer starting from the bottommost layer of the application
- **LayerType** – an enumerated value that gives the type of layer
- **LayerRect** – the bounding box of the visible data on the layer. This is the actual size, regardless of whether or not the layer is expanded.

Sample Script

```
def Do(Environment):
      # start by selecting the bottommost layer in the image.      This needs to be
      # in a try block because if we are already on the bottomost layer we will
      # throw an error
      try:
            # get the path from the bottom from the active layer
            # so we can restore it when done
            Props = App.Do( Environment, 'ReturnLayerProperties' )
            PathBack = Props[ 'Path' ]
            App.Do( Environment, 'SelectLayer',
                  { 'Path': (9999,-9999, [], False ) } )
      except:
            pass # error is OK - just ignore and keep going

      FoundLayer = True
      LayerNum = 1
      while FoundLayer == True:
            Props = App.Do( Environment, 'ReturnLayerProperties' )
            print Props[ 'Path' ]
            print Props[‘LayerType’]
            print Props[‘LayerRect’]
            print '%3d: %s (%d%% Opacity)' % (LayerNum,
                  Props[ 'General' ][ 'Name' ],
                  Props[ 'General' ][ 'Opacity' ])
            LayerNum += 1

            FoundLayer = App.Do( Environment, 'SelectNextLayer' )

      # now that the loop is done, select the bottom layer and then
      # use the pathback to restore the layer that was active when we started
      App.Do( Environment, 'SelectLayer',
            { 'Path': (9999,-9999, [], False ) } )
      App.Do( Environment, 'SelectLayer', { 'Path': PathBack } )
```

Description

This command permits programmatic access to everything in the layer properties dialog. The return value is the same set of repositories that is the input to the LayerProperties command.

In comparison to simple command recording, this command makes it possible to write scripts that make decisions based on the properties of a layer. For example, this command would make it easy to write a script that increases the opacity of all layers by 10%.

The layer properties command doesn't explicitly have a value for the layer type, so an enumerated value is included in the return type to identify the layer. The possible values (assuming App.Constants.Layertype) are:

- Raster
- Vector
- Group
- Mask
- ColorBalance
- ChannelMixer
- Curves
- HueSatLum
- Posterize
- Threshold
- Invert
- Levels
- BrightnessContrast
- Selection
- FloatingSelection
- ArtMedia

The path in the return value is the path to select this layer starting from the bottommost layer in the image. This makes it possible to write scripts that save the active layer, perform a set of operations and then restore the active layer.

The layer rectangle is the bounding box of visible data on the layer. In the case of vector and group layers it is the bounding box of the children on the layer. Masks and adjustment layers may well contract to an empty rectangle even though they affect the whole canvas. In those cases this will return the actual rectangle, not the effective rectangle.

Command: ReturnVectorObjectProperties

Parameters: None

The return value is a list of all the selected vector objects.

Sample Script

The sample shown below will dump all the properties of an object to the output window:

```
from PSPApp import *
import pprint

def Do(Environment):
      # get the vector object properties
      Result = App.Do( Environment, 'ReturnVectorObjectProperties' )

      # use the pretty printer to send it to the output window
            pp = pprint.PrettyPrinter()
      pp.pprint( Result )
      return
```

Description

This command returns the properties of all the selected vector objects. If a group is selected, the properties of both the group and the members of the group are returned.

Vector objects can be nested many levels deep, but the returned list is always flat. So to properly encode groups, the list entry for a group includes the number of children in the group. The children will immediately follow the group. If a group is a child of another group, then only the group object itself is included in the child count. The nested group will have its own children, and those children will not count against the parent group's child count.

For example, assume the following arrangement:

Group 1
Circle
Triangle
Group 2
Square
Pentagon
Hexagon
Octagon
Dodecagon

The order of objects in the returned list would be:

[(Group 1, 4 children), (Circle), (Triangle), (Group 2, 3 children), (Square), (Pentagon), (Hexagon), (Octagon), (Dodecagon)]

Each element in the list is a dictionary containing eight elements.

- Child Count – number of children of this object. For a group, this is the number of members of the group. For all other objects this is 1.

- **ObjectType** – one of ‘Path, ‘Group’, ‘TextEx’, ‘Rectangle’, ‘Ellipse’ or ‘SymmetricShape’
- **GroupName** – Name of the Group, or None if ObjectType is not Group
- **ObjectData** – properties of a Path object, or None if ObjectType is not Path
- **TextExData** – properties of a Text object, or None if ObjectType is not TextEx
- **RectangleData** – properties of a Rectangle object, or None if ObjectType is not Rectangle
- **EllipseData** - properties of an Ellipse object, or None if ObjectType is not Ellipse
- **SymmetricShapeData** - properties of a SymmetricShape object, or None if ObjectType is not SymmetricShape

The properties returned for Path, Rectangle, Ellipse and Symmetric Shape objects are the same set of properties that are displayed in the vector properties dialog. For Path and Symmetric Shape objects, a list of nodes that make up the path is also returned. Each node is defined by a tuple containing a type and zero or more points. The makeup of the tuple varies with the type:

- **App.Constants.PathEntryType.MoveTo** – the second element of the tuple is the point to move to
- **App.Constants.PathEntryType.LineTo** – the second element of the tuple is the end point of the line
- **App.Constants.PathEntryType.CurveTo** – the second and third elements of the tuple are the bezier control points, and the fourth member of the tuple is the end point
- **App.Constants.PathEntryType.ClosePath** – this is the only member of the tuple

The properties returned for each type of object can be used as input to the AddGroupsAndObjects command, as well as the specific command that is normally used to create that type of object. For example, EllipseData properties as returned by the ReturnVectorObjectProperties command can be used as input to the CreateEllipseObject command. The Path property of a Path object can also be used as input for the NodeEditAddPath command.

Command: AddGroupsAndObjects

Parameters:

- **ListOfObjects** – a list of objects to create using the same format as the data returned by ReturnVectorObjectProperties. See that command for a description of the parameters.

Sample Script

The script shown below uses the AddGroupsAndObjects command to replicate all of the selected objects onto a new document.

```
from PSPApp import *

def Do(Environment):
      # get all the selected objects
      Result = App.Do( Environment, 'ReturnVectorObjectProperties' )

      # make a new document
      App.Do( Environment, 'NewFile', {
            'Width': App.TargetDocument.Width,
            'Height': App.TargetDocument.Height,
            'ColorDepth': App.Constants.Colordepth.SixteenMillionColor,
            'DimensionUnits': App.Constants.DimensionType.Pixels,
            'ResolutionUnits': App.Constants.ResolutionUnits.PixelsPerIn,
            'Resolution': 100,
            'Transparent': True,
            'VectorBackground': True,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Silent,
                  'AutoActionMode': App.Constants.AutoActionMode.Match
                  }
            })

      # SelectDocument
      App.Do( Environment, 'SelectDocument', {
                  'SelectedImage': 0,
                  'Strict': False,
                  'GeneralSettings': {
                        'ExecutionMode': App.Constants.ExecutionMode.Default,
                        'AutoActionMode': App.Constants.AutoActionMode.Match
                        }
                  })

      # create all those objects on the new document?
      App.Do( Environment, 'AddGroupsAndObjects', {
            'ListOfObjects': Result['ListOfObjects']
            })
```

Description

The AddGroupsAndObjects command is the inverse of the ReturnVectorObjectProperties command. The list of objects that it takes is in exactly the same format as that returned by ReturnVectorObjectProperties.

AddGroupsAndObjects always creates new objects. To add path data to an existing Path object, use NodeEditAddPath instead.

Command: NodeEditAddPath

Parameters:

- Path – an object path using the same form as ReturnVectorObjectProperties
- WindPath – Boolean, if True, nested objects are wound to become cutouts

- **ReplacePath** – Boolean, if True, replaces the existing data of the active shape rather than adding to it

Sample Script

The sample VectorMerge scripts that are included with Paint Shop Pro X8 are good examples of the use of the NodeEditAddPath command.

Description

NodeEditAddPath adds the contours of an object to the currently active shape. It is used to add path data to an existing Path object - not to Rectangle, Ellipse, SymmetricShape, or Text objects.

Command: DeleteEXIF

Parameters: None

Sample Script

```
def Do(Environment):
      # Delete all EXIF information
      App.Do( Environment, 'DeleteEXIF', { })
```

Description

The DeleteEXIF command strips all EXIF data from the target image. Since EXIF data can be as much as 65K bytes, stripping it from images can result in a significant size savings. This is especially true for small images such as thumbnails or anything intended for a web page.

Command: ReturnGeneralPreferences

Parameters: None

Sample Script

```
def Do(Environment):
      GeneralPrefs = App.Do( Environment, 'ReturnGeneralPreferences', )
      for key in GeneralPrefs:
            print '%-30s:     %s' % (key, GeneralPrefs[key])
```

Description

Returns all information from the “General Preferences” dialog.

Command: GetCommandInfo

Parameters:

- **TargetCmd** – the name of the command to retrieve
- **ParamInfo** – enumerated value that controls retrieval of either last used or default parameter repository. Choices are:
    - **App.Constants.ParamInfo.None** – no repository is returned
    - **App.Constants.ParamInfo.Default** – factory default settings are returned
    - **App.Constants.ParamInfo.LastUsed** – last used settings from the registry are returned

If default or last used is specified a copy of the command will be instantiated. For many commands this will fail if no document is open at the time.

Sample Script

```
from PSPApp import *
import sys

def DumpAttributes( AttribList ):
      ' print out the list of attribs, one per line.'
      for attrib in AttribList:
            if attrib[1] == 1:
                  print '\t\t', attrib[0], ': true'
            else:
                  print '\t\t', attrib[0], ': ', attrib[1]

def FormatCommandInfo( cmdName, Info ):
      ''' Take the output of GetCmdInfo and send it to the output with
            minimal formatting.      Unlike DumpCommandInfo, this command has
            knowledge of all the expected keys in the repository, so will
            not react well to the implementation of GetCommandInfo changing'''
      print 'Command Info for:\t', cmdName
      print 'Name:\t\t', Info['Name' ]
      print 'Local Name:\t', Info[ 'LocalName' ]
      print 'Library:\t\t', Info[ 'Library' ]
      print 'Description:\t', Info[ 'Description' ]
      print 'Status bar text:\t', Info[ 'StatusBarText' ]
      print 'Tool tip text:\t', Info[ 'ToolTipText' ]
      print 'Undo text:\t', Info[ 'UndoText' ]
      print 'Menu path:\t', Info[ 'MenuPath' ]
      print 'Menu text:\t', Info[ 'MenuText' ]
      print 'Command Attributes:'
      DumpAttributes( Info[ 'CommandAttributes' ] )
      print 'Resource Attributes:'
      DumpAttributes( Info[ 'ResourceAttributes' ] )
      print 'Parameter Repository:'
      paramDict = Info[ 'ParameterList' ]
      for key in paramDict:
            print '\t\t', key[0], '   -      ', key[1]
      print

def Do(Environment):
```

```
for cmdName in App.Commands:
      try:
            Info = App.Do(Environment, 'GetCommandInfo',
                  { 'TargetCmd': cmdName,
                  'ParamInfo': App.Constants.ParamInfo.Default } )
            FormatCommandInfo( cmdName, Info )
      except:
            print >> sys.stderr, 'Error getting info on command ', cmdName
            pass
```

Description

This command is used to get information about any command in the system.

The return value is in 3 parts. The first part is the basics of the command and resource data class, including:

- Name – the script name of the command. This is not localized
- LocalName – the localized version of the command name. This is used in the UI, but not in the script.
- Library – the command DLL that contains the command (e.g. CmdNonGraphic.dll)
- Description – the description of the command that was entered when it was created
- Status bar text – the help text shown on the status bar for the command
- Tool tip text – the tooltip text shown when the mouse hovers over the toolbar button for the command
- Undo text – the text shown in the edit menu and command history when the command is on the undo stack.
- Menu path – Nominally, the path to the command in the standard menu tree. Submenus are separated by ~ characters. For example, the gaussian blur command has a menu path of 'Effec&ts~&Blur'. Since the effect browser is the only thing that looks at menu path it may well be incorrect for anything that is not on the effects menu.
- Menu text – Nominally, the text shown on the menu for this item. In PaintShop Pro this value is not used since menus are defined in the application rather than being built dynamically.
- CommandAttributes – the complete set of command attributes. This is a list of tuples, where each tuple consists of ('attribute name', value). The set of command attributes changes as new ones are defined, but the currently defined set is listed below.
- ResourceAttributes – the complete set of resource data attributes. This is identical in format to the command attributes.

The defined command attributes are:

- IsUndoable

- IsExternal
- HasSideEffects
- NoScriptRedo
- NoCommandHistory
- NoDocumentNeeded
- DisallowNullMaterial
- ScriptCommand
- AvailableScanning
- Has RandomSeed
- FinalizeTransient
- ProcessChanges
- ProcessChangesOnSubCommand
- NoRepeat
- keCANoAnnotationErase
- NoScriptRecord
- NoIsValidCache
- ShowInEffectsBrowser
- SaveSettings
- NoProtectedMode
- EditRequiresGraphic
- Obsolete Command
- UsesCachedMaterials
- NoModeToggle

The defined resource data attributes are:

- HasWinPlacement
- IsPreviewable
- IsProofable
- Has Dialog
- HelpID
- ExclusionMask

- DynamicType
- DlgHasPresets
- DlgHasRandomize
- IsEditable

The second part is a copy of the parameter repository used by the command. The output will be the last used values for the command. This is a dictionary object.

The third part is a dictionary that contains information about each parameter in the repository. For each parameter it includes:

- Localized name
- Description
- Type identifier
- Default value
- Minimum and maximum values (numeric types only)
- Constant object pointer (enumerated types only)
- Parameter path. This is used to resolve names that are possibly duplicated in subrepositories. It describes the complete path to a parameter, as in
- 'GeneralSettings.ExecutionMode'
- Flags – is none, supports none, available in preset, etc.
- Name

Command: EventNotify

Parameters:

- EventType – an enumeration describing the event that occurred

Sample Script

```
def Do(Environment):
      # tell PaintShop Pro we just invalidated the pattern cache
      App.Do( Environment, 'EventNotify', {
            'EventType': App.Constants.Event.PatternCache
      } )
```

Description

To speed access to commonly used resource files such as patterns and gradients, PaintShop Pro builds cache files. When it checks for the existence of the resource file, it usually just checks if the file exists in the cache.

Interactively, PaintShop Pro will refresh the cache file whenever the user goes to choose a resource. But when running from a script, the refresh code will not be reached because the picker dialog is not displayed.

The solution is to use the EventNotify command to tell PaintShop Pro that a cache has been invalidated. This will force PaintShop Pro to reload the affected cache file.

The supported events are:

- PictureTubeCache
- PatternCache
- TextureCache
- GradientCache
- BrushCache
- PictureFrameCache
- StyledLineCache
- PluginCache
- ScriptCache
- PresetCache
- SwatchesCache
- MaskCache
- SelectionCache
- CMYKCache
- PrintTemplateCache
- WorkspaceCache
- PaletteCache
- DeformationMapCache
- EnvironmentMapCache
- BumpMapCache
- DisplacementMapCache
- PresetShapeCache

Command: StartForeignWindow

Parameters:

- **WindowHandle** – an integer containing the Win32 window handle of a dialog created by a script

Sample Script

```
# just show the dialog
Wnd = ExploreConstants()
App.Do( Environment, 'StartForeignWindow', {
      'WindowHandle': int(Wnd.winfo_id())
      } )
Wnd.mainloop()
App.Do( Environment, 'StartForeignWindow', { 'WindowHandle': 0 } )
```

Description

One of the more powerful features of scripts is that they can contain their own GUI. This makes it possible to write scripts that gather complex data from the user. However, the interaction of a script dialog and PaintShop Pro is rather problematic. Since the dialog is not properly a child of PaintShop Pro, PaintShop Pro is perfectly happy if the dialog ends up behind PaintShop Pro. Also, while the dialog is up the PaintShop Pro UI is not running, so menu items and toolbars don't update properly.

Finally, if PaintShop Pro attempts to start a new script while a dialog window is running, the Python interpreter will be in an undefined state and the program will crash.

This command exists to deal with these issues. The parameter is the Win32 window handle of the dialog being displayed. For a Tkinter dialog, this is returned by the winfo_id() method. For a wxWindow, use the GetHandle() method.

So long as the window identified in the call to StartForeignWindow exists, PaintShop Pro will ensure that it remains on top of the application, and it will keep the PaintShop Pro UI updating. While a foreign window is running, PaintShop Pro will disable running scripts, loading presets, and loading brush tips.

PaintShop Pro will automatically terminate the foreign window state as soon as the identified window is destroyed. You can also manually terminate the state by calling StartForeignWindow again with a window handle of zero.

If dialogs are not bracketed by calls to StartForeignWindow, PaintShop Pro will allow a second script to be launched, which will usually crash the application.

Command: MsgBox

Parameters:

- **Buttons** – a constant (App.Constants.MsgButtons) with the values OK, OKCancel or YesNo
- **Icon** – a constant (App.Constants.MsgIcons) with the values Info, Question, or Stop

- **Text** – the text displayed in the box.

The return value is the button pressed. OK/Yes = 1, No/Cancel = 0

Sample Script

```
from PSPApp import *

def Do(Environment):
      ButtonTypes = ('Cancel/No','OK/Yes' )
      result = App.Do(Environment,   'MsgBox', {
            'Buttons': App.Constants.MsgButtons.OK,
            'Icon': App.Constants.MsgIcons.Info,
            'Text': 'This is an OK button with an informational icon.'
            })
      print 'You pressed ', ButtonTypes[ result ]
      print

      result = App.Do(Environment,   'MsgBox', {
            'Buttons': App.Constants.MsgButtons.OKCancel,
            'Icon': App.Constants.MsgIcons.Question,
            'Text': 'These are OK and Cancel buttons with a question mark icon.'
            })
      print 'You pressed ', ButtonTypes[ result ]
      print

      result = App.Do(Environment,   'MsgBox', {
            'Buttons': App.Constants.MsgButtons.YesNo,
            'Icon': App.Constants.MsgIcons.Stop,
            'Text': 'These are Yes and No buttons with a stop sign icon.'
            })
      print 'You pressed ', ButtonTypes[ result ]
      print
```

Description

This command just provides a simple way of displaying a message box. The user can control the text, the caption, the icon and the buttons displayed. The return value is the button that was pressed. See Figure 4 for an example showing the OK button.

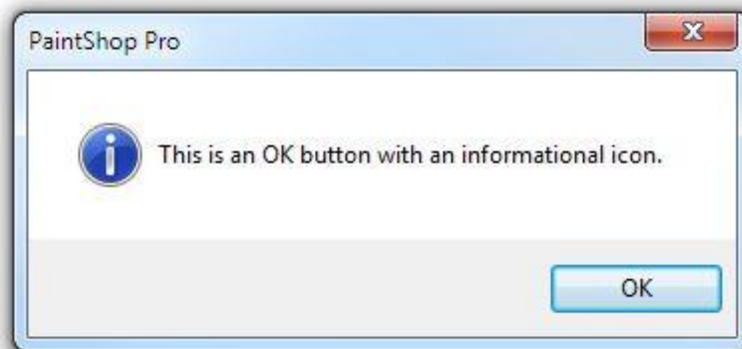

Figure 4

Command: GetString

Parameters:

- DefaultText – the text that is initially shown in the dialog
- DialogTitle – text shown in the title bar of the dialog
- Prompt – a string shown above the edit box
- MaxLength – the maximum length of the string allowed. If this is shorter than the DefaultText then it is automatically increased to be as long as the default text

Sample Script

```
from PSPApp import *

def Do(Environment):
      Result = App.Do( Environment, 'GetString', {
            'DefaultText': 'Hello World',
            'DialogTitle': 'This should be the title',
            'Prompt': 'This is a fairly long prompt string that will probably wrap around to
a second line',
            'MaxLength': 25,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Interactive
                  }

            })
      print Result[ 'OKButton' ]
      print Result[ 'EnteredText' ]

      Result = App.Do( Environment, 'GetString', {
            'DefaultText': 'Silent Mode',
            'DialogTitle': 'Foobar',
            'Prompt': 'Short entry',
            'MaxLength': 25,
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Silent
                  }
            })
      print Result[ 'OKButton' ]
      print Result[ 'EnteredText' ]
```

Description

This command brings up a simple dialog box that contains an edit box and prompt. The user types in a string and presses OK or Cancel.

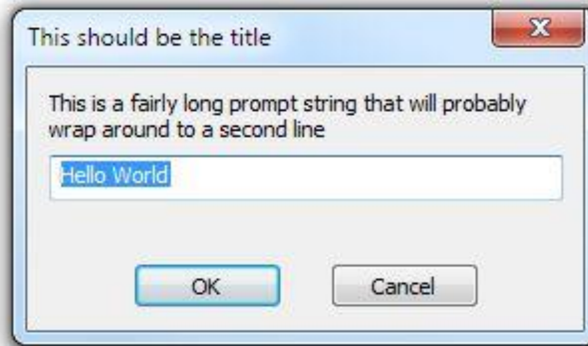

Figure 5

The return is a dictionary containing two keys:

- OKButton – True if the OK button was pressed
- EnteredText – the text entered by the user. If the user pressed cancel then the input text will be returned unchanged.

If the execution mode is set to silent then the dialog is suppressed and the default string is returned unchanged.

Command: GetNumber

Parameters:

- DefaultValue – the initial value in the numeric edit control (between –1,000,000 and 1,000,000, default 1)
- MinValue – the minimum value allowed (between –1,000,000 and 1,000,000, default 0)
- MaxValue - the maximum value allowed (between –1,000,000 and 1,000,000, default 100)
- DialogTitle – text shown in the title bar of the dialog (default is “Enter a Number”)
- Prompt – text shown above the numeric edit control (default is “Number:”)
- GetInteger – boolean, true if input is limited to integers. (defaults to true)
- LogarithmicSlider – boolean, true if the popout slider should use a logarithmic scale. (defaults to false)

Sample Script

```
from PSPApp import *

def Do(Environment):
    Result = App.Do( Environment, 'GetNumber', {
```

```
            'DefaultValue': 25.3,
            'MinValue': 1,
            'MaxValue': 202,
            'DialogTitle': 'GetNumber - foobar',
            'Prompt': 'Prompt from script',
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Interactive
                  }
            })
      print Result[ 'OKButton' ]
      print Result[ 'EnteredNumber' ]

      Result = App.Do( Environment, 'GetNumber', {
            'DefaultValue': 2318,
            'MinValue': 100,
            'MaxValue': 125000,
            'DialogTitle': 'Get a large number',
            'GetInteger': True,
            'LogarithmicSlider': True,
            'Prompt': 'Enter a floating point number that has no bearing on absolutely
anything',
            'GeneralSettings': {
                  'ExecutionMode': App.Constants.ExecutionMode.Interactive
                  }
            })
      print Result[ 'OKButton' ]
      print Result[ 'EnteredNumber' ]
```

Description

This displays a dialog box containing a numeric edit control and a user defined prompt:

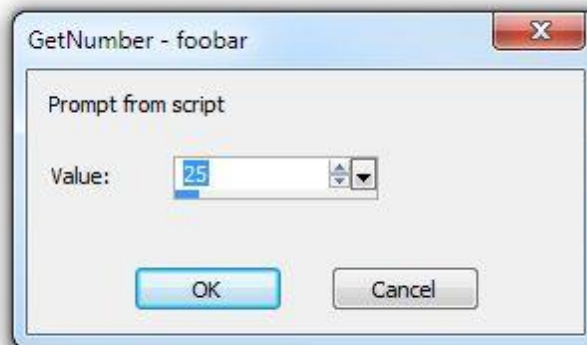

Figure 6

Like the GetString command, it returns a dictionary that contains the returned number and which button was pressed. This does some sanity checking on parameters – if min is greater than max the two parameters are inverted. Logarithmic sliders are automatically turned off if either min or max is less than zero.

Command: ScriptWndAutoShow

Parameters:

- Enable – a boolean value. True to enable auto show, false to disable it.

Sample Script

```
Result = App.Do( Environment, 'ScriptWndAutoShow',
      { 'Enable': True } )
print 'Script window auto show enabled, previous value was ', Result
```

Description

By default the script output window turns on its visibility whenever any data is written to sys.stderr. The assumption is that script errors are something that the user needs to know about, so displaying the window is a good thing. However, there are cases where errors are expected and handled normally. In these cases it is desirable to disable the auto show capability.

The parameter is a boolean that controls whether auto show is on or off. The return of the command is the previous state of the flag.

Command: ScriptWndClear

Parameters: None

Sample Script

```
App.Do( Environment, 'ScriptWndClear' )
```

Description

This command simply clears the script output window. There are no parameters.

Command: DumpScriptOutputPalette

Parameters:

- OutputFile – the file where the contents of the script output window is written
- OverwriteFile – if true and the output file exists, the output file will be overwritten; if false and the file exists, the output file will be appended to

Sample Script

```
App.Do( Environment, 'DumpScriptOutputPalette', {
      'OutputFile': 'c:\\SOPDump.txt',
      'OverwriteFile': False
      })
```

Description

This command saves the current contents of the Script Output palette to a text file. There are no parameters.

Command: ScrollScriptOutputPaletteToEnd

Parameters: None

Sample Script

```
App.Do( Environment, 'ScrollScriptOutputPaletteToEnd')
```

Description

This command can be used when you want to ensure that the last line of the Script Output palette is visible. There are no parameters.

Command: SelectDocument

Parameters:

- SelectedImage – The relative index of the document to select, or zero to select the currently active document.

The index of the document is relative to the current target document in opening sequence. For example: Say you opened four documents in the following order: "Doc A", "Doc B", "Doc C", and "Doc D". Now let's say "Doc C" was active when you executed a script. "Doc C" would be the current target document for the script. To set "Doc A" to be the target you would set the parameter to –2. Setting the parameter to 1 would set "Doc D" as the target. Note: this index is relative to the scripts current target document. The following flow shows this using the documents opened in the example above:

> Script started with "Doc C" active. So "Doc C" is the script's target document.
> "SelectDocument" called with a parameter of -2. "Doc A" becomes the target.
> "SelectDocument" called with a parameter of 3. "Doc D" becomes the target.
> "SelectDocument" called with a parameter of -2. "Doc B" becomes the target.
> "SelectDocument" called with a parameter of 1. "Doc C" becomes the target.

- Strict – Boolean value of True / False. If set to true and the requested document can't be found an error will be returned stopping the script. If set to false and the requested document cannot be found then the script's target document will remain unchanged, no error will reported, and the script will continue to execute.

Sample Script

```
# Set the currently active document to be the script new target document
App.Do( Environment, 'SelectDocument', {
      'SelectedImage': 0,
```

```
      'Strict': False } )

# Set the document opened after the current target document to be the
# new target document
App.Do( Environment, 'SelectDocument', {
      'SelectedImage': 1,
      'Strict': False } )
```

Description

This command will select a new document for the script to operate on. Whatever document was active (had focus) when a script is executed, becomes the "target" document for the script. Any commands that are executed in the script are applied to the script's target document. A script can change its target document by using the "SelectDocument" command.

Active document vs. Target document

The active document is the document that is active or currently has input focus. The target document is the same as the active document at the moment the script is executed. Once the script is running the active and target documents can be different. For example: If the user has two documents opened and executes a script on the 1st document, and then clicks on the 2nd document, the script will continue to operate on the 1st document. "SelectDocument" can also be used to select the currently active document as the script's target document by passing a 0 for the "SelectedImage" parameter.

When a script is used in a batch processing, only the image being processed is accessible, so this command will not succeed in selecting any other document.

Command: SelectTool

Parameters:

- Tool – the name of the tool to select. The return value is the name of the tool that was previously selected.

Sample Script

```
def Do(Environment):
      ListOfTools = ['Airbrush','ArtEraserTool','AutoSelection','BackgroundEraser','Burn',
            'ChalkTool','ChangeToTarget','CloneBrush', 'ColorChanger','ColoredPencilTool',
            'ColorReplacer','CrayonTool','Crop','Dodge','EllipseTool',
            'EmbossTool', 'Eraser','Eyedropper','Fill','FreehandSelection',
            'Hue','LightenDarken','MagicWand','Makeover','MarkerTool',
            'MeshWarping','Mover','OilBrushTool','PaintBrush','PaletteKnifeTool',
            'Pan','PatternFill','PastelTool','PenTool','PerspectiveTransform',
            'Pick','PictureTube','PresetShapes','PushBrush', 'RectangleTool',
            'Redeye','Saturation','ScratchRemover','Selection', 'SharpenBrush',
            'SmartSelection','SmearTool','SmudgeBrush','SoftenBrush','Straighten',
            'SymmetricShapeTool','TextEx', 'WarpingBrush', 'WatercolorBrushTool','Zoom']
```

```
for Tool in ListOfTools:
try:
      LastTool = App.Do( Environment, 'SelectTool', { 'Tool': Tool } )
      print 'Tool was %s, just selected %s' % ( LastTool, Tool )
except:
      print 'Error selecting tool %s, presumably IsValid failed' % ( Tool )

CurrentTool = App.Do( Environment, 'SelectTool', { 'Tool': '' } )
print 'The current tool is ', CurrentTool
print '----------------------------'
```

Description

This command allows a script to change the active tool. Tool changes are not recorded, but for blueprints, GUI automation, or tutorial purposes, it is desirable to allow setting a tool via script.

The return value is the name of the tool active before the command was executed. This name can be used to restore the active tool via another call to SelectTool. (Though calling SelectPreviousTool does the same thing without having to track it manually).

If the tool name passed in is blank, the current tool is returned without selecting a new one.

Command: SelectPreviousTool

Parameters: None

Sample Script

```
NewTool = App.Do( Environment, 'SelectPreviousTool' )
print 'After previous tool, tool is %s' % (NewTool)
```

Description

This command selects the previous tool in the history list. It is valid only when a document is open and there is a previous tool to select. It returns the name of the tool that was active before the command was executed.

Next tool/previous tool act like undo redo on tool selects. PaintShop Pro stores the ID of the last 100 tools used in a FIFO list. You can move forward and backwards through the list until such time as you select a new tool by means of something other than previous/next. At that point any tools forward in history (i.e. redo) are purged.

As an example, suppose you select tools A, B, C, D, E, F through the GUI. You then press previous tool three times, stepping backwards through E and D, to C. Your history list is still ABCDEF, but the current location is C. Stepping forward takes you to D. If you manually select tool G, everything forward of D is discarded from the history list, so the history is now ABCDG. Selecting previous tool will take you to D, but E and F have been removed.

This command is not normally present on the menus but can be customized in. If on a menu it will display the name of the tool to be selected as part of its menu text when enabled.

Command: SelectNextTool

Parameters: None

Sample Script

```
NewTool = App.Do( Environment, 'SelectNextTool' )
print 'After Next tool, tool is %s' % (NewTool)
```

Description

This command selects the next tool in the history list. It is valid only after a call to SelectPreviousTool and before any other select tool operations. It returns the name of the new tool that is selected.

See the description of SelectPreviousTool for more details.

Appendix A: Sample Scripts

Sample scripts accompany the Scripting Guide.

The following table describes the sample scripts that are available. The scripts range in complexity—from simple scripts that exercise a particular command to complex scripts that are very specialized. Some sample scripts can be used as they are; other sample scripts use hardcoded directory strings that need to be customized for your system.

Sample script	Description
AddBordersAndSelect	This script executes an AddBorders command and then selects the borders. It does this by using GetCommandInfo to retrieve the last used parameters of the AddBorders command, and uses those as input to the selection command.
DumpEXIFInfo	This script uses the ReturnImageInfo command to send all of the EXIF data to the script output window.
ExploreConstants	This is a Tk dialog that permits you to browse the collection of constants defined by PaintShop Pro. It is not a sample so much as a utility to assist with writing scripts.
NameThoseLayers	This script iterates the layers in the open image and prompts for a new name for each one. It illustrates how to iterate layers, and access layer property data.
PrepForStitching	This script arranges the currently open images edge to edge on a new document. It is useful as the first step in stitching multiple images together. As a sample it shows how to work on multiple documents.
ProcessDirectory	This is a utility script that processes all images in a directory tree you define. The directory name and search pattern are defined in the script, and must be modified to fit your local system. It then calls a ProcessFile function for each file it opens; this needs to be overridden to do something useful.
TestGetMaterial	Sample script that illustrates how to use the GetMaterial command to access the foreground/background colors set on the material palette.
TestGetRasterSelectionRect	Sample script that illustrates use of the GetRasterselectionRect command to query for information about the selection

TestGetStringAndGetNumber	Sample script that illustrates use of the GetString and GetNumber commands. These commands make it simple for a script to prompt for input without going to the effort of writing a Tk based dialog.
TestMsgBox	Sample script that illustrates use of the MsgBox command for displaying a message box.
TestScriptWindowCommands	Sample script that illustrates using commands that control the script output window.