Pretty Good Terminal v7.0.5.0

Laszlo Frank

User's manual

3/10/2017

1 CONTENTS

1 Contents 1
2 WARNING 4
3 Versioning and Change Log 5
4 Quick Start 6
5 Concept 7
6 Editions 8
7 Requirements 8
8 Installation 9
9 Activation 10
9.1 Upgrading to new version 10
10 Compatibility with earlier versions 11
11 Knonw Issues 12
11.1 Terminal related issues 12
11.2 Cryptographic issue 12
11.3 Python related issues 12
12 Why Jump Servers 13
13 PGT Script file format 14
14 NetConf protocol support and limitations 15
15 Configuration 16
15.1 Scripting settings 16
15.2 General settings 20
15.3 Jump server settings 22
15.4 Custom Action Handlers 24
15.5 Custom Menu Handlers 25
15.6 Multiple Vendor Support 26
15.7 Visual Script Settings 27
15.8 Python settings 28
15.9 Recovering a profile on forgotten password 29
16 The User Interface 30
17 Generating a script 31
17.1 Using Netconf protocol 33
17.2 The None connection protocol 33

18 Custom Action ID 34
19 Opening and executing Scripts 35
19.1 Schedule script execution 36
20 Modifying the script 38
21 Creating dialogues in scripts 39
22 Saving the script and the results 40
23 Working with Python Scripts 42
23.1 Interactive Python scripts 42
23.2 Running Python scripts automatically 46
23.3 Editor shortcut keys 46
23.4 Advanced Python editor features 47
24 Working with Visual Scripts 49
24.1 What is a Visual Script 49
24.2 Concept 49
24.3 The Basics 50
24.3.1 Building a simple Script 50
24.3.2 About vScript Compilation errors 54
24.3.3 Using a vScript 55
24.3.4 Debugging a vScript 58
24.3.5 Watch Variables 61
24.3.6 Handling vScript Runtime Errors 62
24.3.7 Logging information from a vScript 62
24.4 Advanced Topics 64
24.4.1 The building blocks of a vScript 64
24.4.2 Command vs Simple Command element 65
24.4.3 The Simple Command element 65
24.4.4 The Command Element 66
24.4.5 Breakpoint Operation 67
24.4.6 About Script Variables 69
24.4.7 Built-in variables 71
24.4.8 Using variables to build CLI commands 71
24.4.9 The Connector Element 75
24.4.10 More About Variables 76
24.4.11 Advanced Options and Code View 77
24.4.12 CustomActionHandlers Priority 78
24.4.13 vScript Objects Lifecycle 79
24.4.14 vScript Limitations 80

24.5 Templates and Repositories 81
24.6 Using Python for vScripts 82
24.7 Debugging Python vScript code 83
24.7.1 Navigate in code 86
25 Cisco YDK-Py Support 87
25.1 YDK-Py based Visual Scripts 88
25.2 YDK-Py Python scripts 90
26 Logging of information 92
27 The terminal window 92
28 Command line options 94
29 Development support 95
30 LICENSE 96
31 LIMITATION OF LIABILITY 97
32 WARRANTY DISCLAIMER 98

2 WARNING

This software – Pretty Good Terminal – is a very powerful application designed to change the configuration of lots of networking devices. Be aware, that the software is not able to determine the validity and the effect of the commands it is executing. It is always the sole responsibility of the user using the application. Issuing the wrong commands may result in serious damage to the network and business.

Always use the software on your own risk !

3 VERSIONING AND CHANGE LOG

PGT version digits are representing the following logic :

- 1st digit : main version number. It is only changed when there is a major change touching the whole logic of the application
- 2nd digit : subversion number. Indicates the development interface version. It is changing only if there is a change in the PGTInterfaces.DLL. This DLL always has the main and subversion, like 4.4.0.0.
- 3rd digit : release version number. It changes frequently when a new feature is introduced but is reset on a newer main- or subversion
- 4th digit : fix version number. This gets zeroed on each new release, and only increased when an error fix is issued for the current release version.

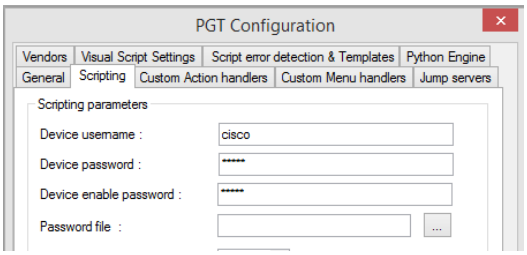
Besides the 4 digit version information a patch level can also be present. A patch is usually published to instantly fix certain errors without version update. Patches are available for download from the website. Please read instructions carefully how to apply a patch.

There is a Release Notes.txt file included with PGT, new features and changes are always summarized there.

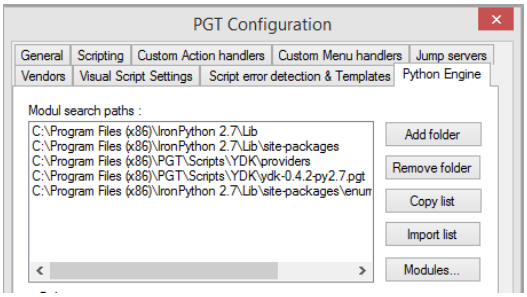
4 QUICK START

If you do not want to go into details for now running through a dozen of pages and just want to get things ready and start using the software, follow these simple steps :

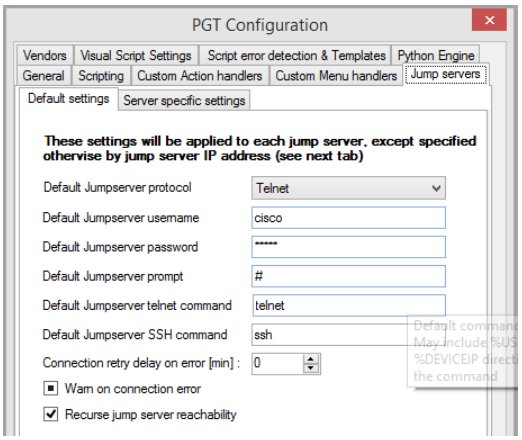
1. Set username and password:
   The first thing you need to set is your access details to devices. Go to Tools/Options and set the following fields :



2. If you want to use Python for vScripts, first ensure IronPython is installed on the system, then set Python search path in Tools/Options/Python Engine. The first time you go to this tab for a specific profile and hence the search path list is epty, PGT will try to discover IronPython installation directory and sets default search paths accordingly. Please also read chapter 7, Requirements with regard to IronPython interoperability.



3. If you need to use jump servers, you can configure general jump server settings (later you may want to customize it server-by-server):



After the above settings done, please select "Save and Close" then restart the application. Without restart, Python search path changes may not be effective.
That's it, you're finished and you can start creating and using scripts.

5 CONCEPT

PGT was originally written as a dual purpose application:

- Provide a powerful UI to run simple CLI-based scripts on several devices for those unfamiliar with scripting or programming languages. What is called a script, is a series of device CLI commands, and is basically stored as a simple CSV file which can also be edited in Excel, or any other text editor. Each and every line is a single, device specific command and also contains all information required to connect to the device and execute a command. Command results recorded, filtered with regex expressions and can be saved after the script terminated.

- Provide an extensible framework for developers to write simple VBS scripts or create and run custom code for complex tasks. Any programming language might be used to exploit this feature as long as it supports writing class libraries and can implement interfaces.
    - VBS scripting support provides a means of automation. One can call various function of PGT from a vbs script, for instance using Excel. Using Excel, the script can easily be debugged.

    - A more sophisticated approach of extensibility is creating a class library and implement a basic interface by a class. In this context, the base framework handles the CSV input file, connects to hosts as described and then transfers execution to the CustomActionHandler class in the class library. This CustomActionHandler object can then call all the built-in functions of the internal ScriptableTerminal class, such as WaitForPrompt(), ExecCommand() functions. The custom action handler can perform any addition tasks as required, such as building an inventory database.

Version 5.0 introduced Visual Scripts which represents a logical layer between the above two by providing an intuitive way of creating complex scripts with limited programming skills.

Starting with version 6.0, Python scripts can also be used. Both the Visual Scripts can be set to use Python and also standalone Python script files can be used as a custom action. The Python Interactive console provides an easy way to test and deploy scripts.

Version 7.0 was primarily targeting to support Cisco YDK-Py API. As PGT was built on IronPython, it has some inherent restrictions of module handling. As a result, the original Cisco YDK-Py code could not be used and PGT v7.x includes a slightly modified version of Cisco YDK library. PGT has its internal, .NET based SSH and NetConf support, therefore does not have to rely on Paramiko – Python SSH library – and NCCClient, the Python NetConf client solution. Instead, leveraging this built-in protocol support, PGT defines its own PGTNetconfServiceProvider Python class which is compatible with the original Cisco code included in YDK.

6 EDITIONS

Starting with version 7.0.0.1 PGT is available in two Editions and is completely free for private use. Licensed version must only be purchased by companies using the software for business goals :

Feature / Edition	Public	Enterprise
Number of activations	unlimited	According to license pack
Development support	full	full
Visual script support	yes	yes
Python support	yes	yes
Licensing	Single user Individual, private use	Multiple user Corporate, business use
Branding	no	possible

The Enterprise edition also includes the Scripting Project Manager plugin, which is an SQL based power tool allowing the management, history tracking and reporting of scripts ran on thousands of devices. The Enterprise edition is not available for download it is only provided upon purchase of license.

7 REQUIREMENTS

PGT is an x86 Windows application. It requires .Net Framework 4.0 which implies that Windows XP / Windows Server 2003 or later is required to run the application. However, PGT has only been tested on x86 builds of Windows version 6.1 and as a result it is the only officially supported platform. Deploying PGT on NTFS partitions are strongly recommended due to increased functionality within script files.

As PGT hosts Python scripts using IronPython, it is necessary that IronPython be present on the host computer. PGT will not install IronPython on its own, it can be downloaded from http://ironpython.net , current release is 2.7.7 which is supported by PGT v7.0

As basic IronPython modules are distributed with PGT, it is not absolutely necessary to install IronPython, as long as only the default, built-in modules are used from Python scripts, *such as Sys, math, Re.*, etc.

Visual Scripts, however, use the uuid Python module which is not present in the standard distribution. As a result, installing IronPython is required for Python based vScripts. For more information please visit IronPython website.

8 INSTALLATION

PGT is a portable application and does not have an installer. You will only have to extract the contents of the downloaded archive and PGT is ready to run. You can also safely copy the installation folder to another location, however, you may have to re-activate the software.

If you wish to use Python for scripts, you may need to install IronPython on the system separately. PGT includes IronPython binaries but not Python libraries, therefore they must be installed separately. Currently release of PGT comes with IronPython version 2.7.7rc2 binaries and only tested with this release.

The majority of program settings are stored in an encrypted XML configuration file named PGTConfiguration.xml. This file should be retained between installations to keep settings. Some other, version dependent settings are stored in folder %userprofile%\appdata\local \Laszlo_Frank\PGT\PrettyGoodTerminal.exe_Strngnamexxx. PGT will take care of these settings between PGT versions and will upgrade the settings as required.

9 ACTIVATION

Any editions of PGT requires a valid license and hence activation. Even though the program is free for personal use, you will have to activate the product on the first start.

The activation process of a purchased version will require your personal registration data (name and email address) that you used for registration at www.prettygoodterminal.com, as well as an Activation ID granted to you when you purchased the software. Your Activation ID is displayed on the http://www.prettygoodterminal.com/MyAccount.aspx page.

If you are not a corporate user and do not use PGT for business purposes you can omit these fields and simply press Request License. You will then be provided a License valid for the amount of time described on the product website.

The activation process of the software will generate a license which is valid only for the computer the software runs on. If you want to run PGT on several computers, you have to activate it on each machine. Should you ever need to re-install PGT on the same computer, the same license will be issued to you.

If you are connected to the Internet, activation will take place in around 10 seconds. If any error occurs, please check the Troubleshooting section of this User's Guide for possible solutions.

If the computer PGT was installed on does not have internet connectivity, you can use the offline activation method. For further information on offline product activation please visit http://www.prettygoodterminal.com/activation.aspx

If you want to learn more about licensing, please follow the link :
http://www.prettygoodterminal.com/licensing.aspx

9.1 UPGRADING TO NEW VERSION

If a new version is released and you want to upgrade to, follow the procedure below.

To activate a new version, you will need a new activation ID. To have the new ID, you need to purchase the new version. To do this, log in to the website and navigate to My Account page. If the new version is provided as a free upgrade, it will be priced zero. Click the "I want to buy" button. If the upgrade is free, the new activation ID will immediately be displayed, otherwise you need to pay the upgrade price before acquiring the new activation code.

At this point delete all files from the installation folder **except the PGTConfiguration.xml file** as it contains all of your settings. Then extract the downloaded archive of the new version and proceed with activation as described above.

10 COMPATIBILITY WITH EARLIER VERSIONS

PGT version 7.x has several enhancements, like supporting NetConf protocol and Cisco YDK-Py API, introducing vScript repository and templates to mention only the most significant changes.

These developments required the transformation of some core components resulting in minor compatibility issues with Python and Visual scripts developed for v6.x editions.

Although changes are subtle at the API level, exiting code must be modified to be made compatible with version 7.x.

When you open a script – either Python or vScript – developed for an earlier version, PGT will try to upgrade the code to match new syntax and notifies you about code conversion.

Conversion made is the following:

- For vScripts, PGT will rename any references to *Executor.STerminal* variable to *Executor.Session*
- For Python scripts any references to *Terminal.* are renamed to *Session.*

For more details about the nature of change please check the Developers reference.

11 KNONW ISSUES

11.1 TERMINAL RELATED ISSUES

- PGT has scripting issues with some escape sequence terminals not sending CR/LF characters at line endings but rather just cursor positioning sequences. In some cases changing the Newline character setting in Advanced Terminal settings (Tools/Options – General) this problem can be resolved.
  This behaviour is mainly observed when connection to hosts is made via jump servers, however, this has never occurred to OpenSSH server.

- Version 6.2 introduced Keyboard Interactive authentication support for jump servers, but this feature is not fully tested yet and has common issues.
- Default Authentication type for scripts is Password authentication, and currently this cannot be changed.

11.2 CRYPTOGRAPHIC ISSUE

When FIPS is enabled on a machine, PGT will not run and a cryptographic error is generated. For more information see https://support.microsoft.com/en-us/kb/811833

FIPS gets enabled for example when Cisco AnyConnect is installed, and the VPN concentrator policy requires FIPS on client machines.

As a quick workaround FIPS can be disabled from registry, changing the value of HKLM\System\CurrentControlSet\Control\Lsa\FIPSAlgorithmPolicy\Enabled to 0.

11.3 PYTHON RELATED ISSUES

PGT is using IronPython to host and manage Python scripts. Currently IronPython can't load modules that are written in C and distributed as a compiled binary image in the form of *.pyd files. Such modules are lxml (etree.pyd), parts of pycrypto and paramiko for example. Later release of IronPython, and hence PGT will probably support this feature. A workaround could be to use IronClad with IronPython, but there is no windows binary release is available at the time of writing.

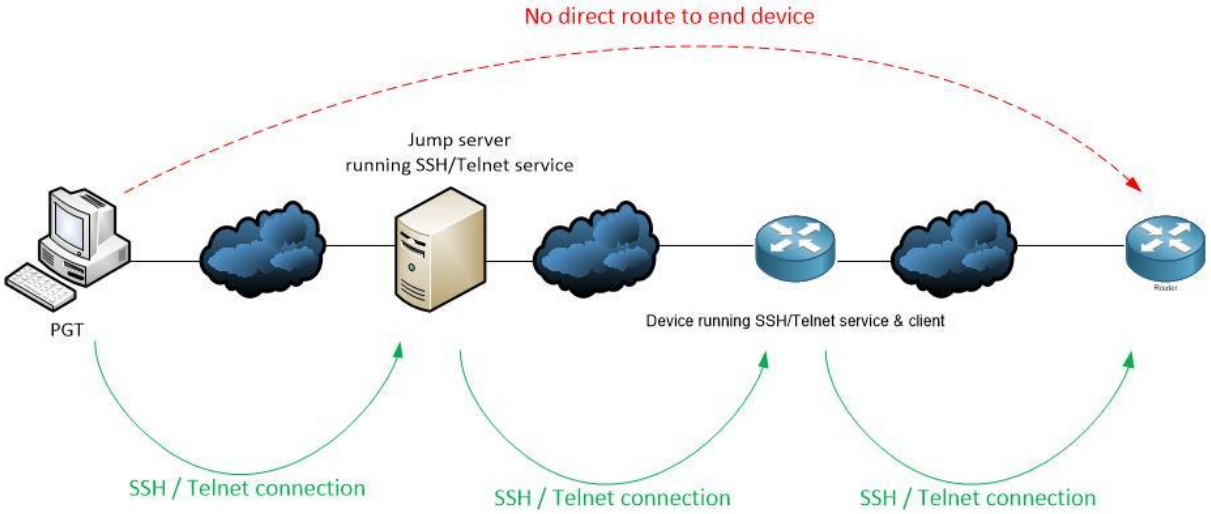
Current release of PGT was built using IronPython 2.7.7rc2 and it is the only supported version.

PGT version 7.x introduced Python Librarian feature which is basically a module explorer. Unfortunately, loading and parsing some modules causes the application to crash and currently PGT is unable to avoid this issue. As a precaution PGT will remember the item caused the crash and next time will try to avoid parsing it. This issue is also documented with IronPython here.

12 WHY JUMP SERVERS

PGT main advantage lays in its capability of using multiple jump servers for establising connection to end hosts. The need for using jumps servers stems from a situation when routed IP connectivity cannot be directly made between a client computer (running PGT) and the end host that needs to be scripted. This isolated network setup can be a result of security measure and is a common scenario in enterprise management networks.

The below drawing illustrates this type of access:



This case, instead of routed access, connection is made through a series of devices capable accepting and originating telnet and/or SSH connections to a reachable next hop. At each hop authentication might be required with different credentials.

PGT would then set up the connection to the first hop (jump server) then would script the rest of the connection through issuing telnet/ssh commands and parsing the result.

Please note that currently PGT does not support jump servers for NetConf protocol.

13 PGT SCRIPT FILE FORMAT

PGT supports different type of scripts like Python scripts or Visual Scripts, but all type of script is relevant to and executed on *one single device*. In PGTs terminology, a PGT script is basically a collection of devices that are the script targets along with connection specification and optionally an action reference that is to be carried out when connection was made.

A PGT Script input file is a CSV text file and must contain the following columns in the exact order as specified below. Additional columns may also be present following mandatory columns and will be read in or written out during load/save operations. When reading the input file, the first line is considered the column header. Mandatory columns are :

Selected, JumpServerList, Vendor, DeviceIP, HostName, ConnectionProtocol, Command, PrivilegedModeRequired, ReconnectRequired, RegExSearch, CustomActionID, RegexResult, CommandResult

Valid values and the meaning of these columns are :

Seleceted : mandatory, can be 0 or 1 and indicates whether the item will be selected for execution

JumpServerList : value is optional, can contain the ip addresses of the jump server(s) through which the connection to the end device is established. Multiple addresses must be separated by semicolons. (please consider the CSV separator character setting in Tools/Options – General as if it was set to semicolon, you cannot enter multiple jump server this way as it would conflict with CSV column separation)

The first – or primary - jump server is handled differently than the rest of the list, since communication channel is always established to the first jump server.

From this jump server on the connection to the rest of the jump servers – or end hosts - will always be controlled by sending text commands and parsing the answer of the terminal. *A single jump server ip address may refer to the last jump server in a chain if **jump server chains** are used. Refer to jump server settings for more information.*

If this field is empty PGT will try to connect to end hosts directly.

Vendor : only mandatory if a **DeviceIP or Hostname** is also specified. Valid values are those specified on the Vendors tab in Tools/Options. If something else specified it will be ignored and a warning dialog presented. It is case insensitive.

DeviceIP : optional, the ipv4 address of the destination device. Can either be empty or must contain a valid ipv4 address. If empty, but a jump server is set, then commands are executed directly on the jump server. Either a jump server or a device ip address is required.

HostName : optional**,** the hostname of the destination device. It serves as a validation point that commands are executed on the correct device.

Valid combinations of JumpServerList, DeviceIP and Hostname are :

- All columns have valid values
- JumpServerList has a value and Hostname is present
- JumpServerList has a value and DeviceIP is present
- Only DeviceIP is present

ConnectionProtocol: mandatory if the **DeviceIP** is specified. This is the protocol used to connect from the jump server to the end device. PGT currently support the following connection protocols:

1. Telnet
2. SSH (SSH1 and SSH2)
3. NetConf
4. None (PGT does not establish any connection to end host. Used for advanced scripts)

Depending on the specified protocol, the corresponding command will be executed on the jump server. See Tools/Options – Jump servers tab.

Command : the command to be executed on the device or the jump server. If, however, a Custom Action ID is set, and there is a valid handler for that action, the command will be ignored and passed directly to the handler. In this case this column is primarily used for passing parameters for the custom action handler.

PrivilegedModeRequired : controls whether privileged (enable) mode is required on the device to execute the command. Its value can be „yes" – as case insensitive, or anything else. Only „yes" will be interpreted as required. In order for this to function, the enable password must be set in Tools/Options.

ReconnectRequired : its value can be „yes" or anything else, only „yes" is interpreted as a valid requirement. As PGT executes the script, it will only establish a new connection to a device, when a new device ip is specified or a connection protocol changes. In some cases, it might be required to reconnect to the same device before the command is executed. In these scenarios, this column should contain „yes", otherwise can be left empty.

RegExSearch : a RegEx pattern can be supplied to search for match in command result string. If a pattern is given it will be evaluated and the result displayed in a separate column.

CustomActionID : this is described later in detail

RegexResult,CommandResult : these columns are optional, but are read in if present . This is to ensure compatibility between the saved script output format and the input format.

Any number of custom columns may be present after the CommandResult column. These columns are not processed by PGT in any way. Plug-ins may use these fields for their own purpose.

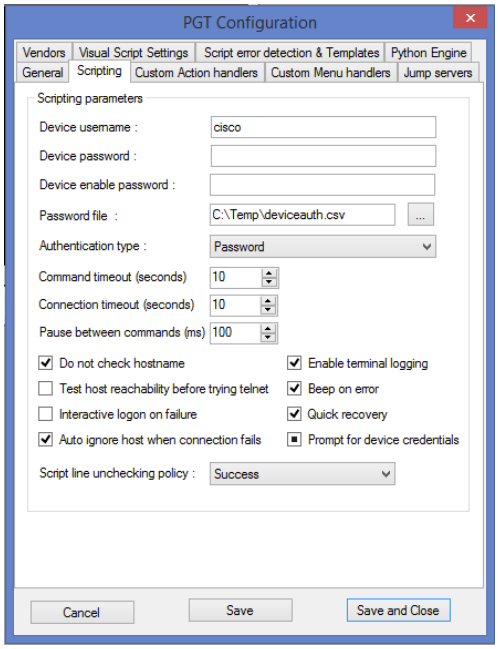
14 NETCONF PROTOCOL SUPPORT AND LIMITATIONS

Currently the NetConf protocol is only supported for direct connections without jump servers.

15 CONFIGURATION

Several parameters are required for PGT to operate correctly. These settings are stored in an XML configuration file, named PGTConfiguration.xml. These options can be set from Tools/Options menu. As this configuration file contains lots of settings, please take care not to delete or damage the file. It is advisable to make backup copies of this file.

15.1 SCRIPTING SETTINGS



On the scripting tab the username and password used for end devices must be set. Do not confuse this setting with jump server settings, the credentials used for jump servers must be specified along with the jump server on the jump servers tab.

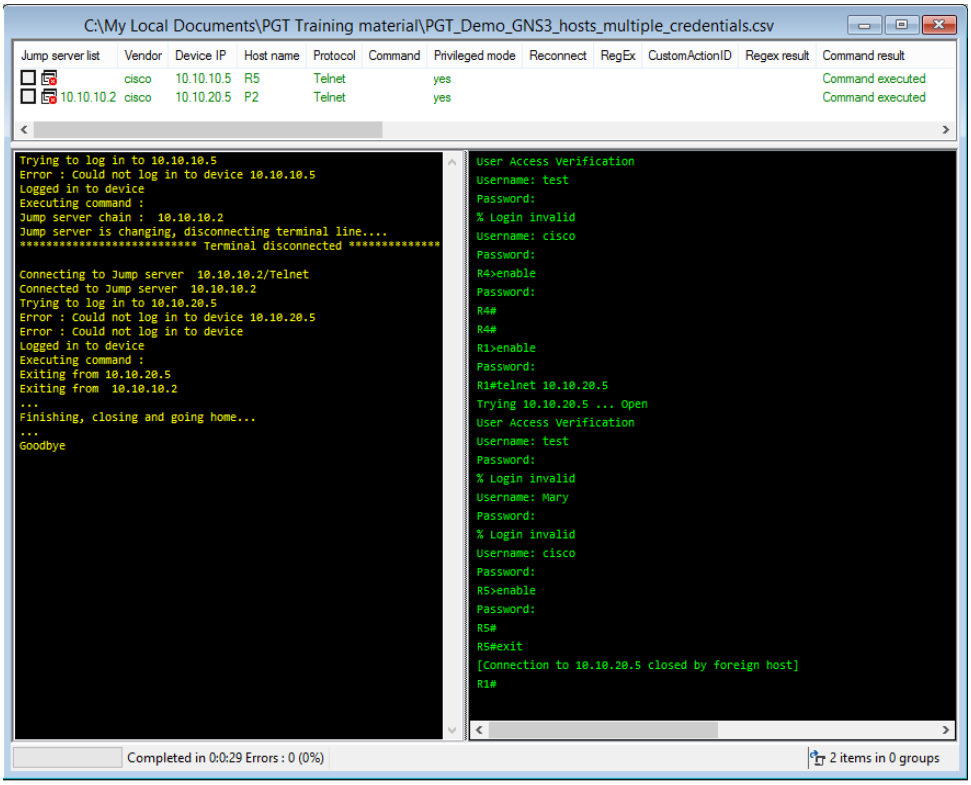
If passwords are different for some devices, a password input CSV file can be specified. This file must contain four columns in the following order: *Device_Ip_address; Logon_username; Logon_password; Enable password*. The file must contain a header line, but the name of the columns are not important. The separator character in this CSV file must match the one set on the general tab.

A sample password file in Excel looks like below example:

	A	B	C	D
1	DeviceIP	UserName	Password	EnablePassword
2	0.0.0.0/0	test	test	cisco
3	10.10.10.0/24	cisco	cisco	cisco
4	10.10.10.5	Joe	guess1	guess1
5	10.10.10.5	Mary	guess2	password
6	10.10.20.5	Mary	guess2	password
7	10.10.20.0/24	cisco	cisco	cisco

As you can see, multiple options can be specified for the same host. In this case, PGT will try each one until can connect successfully. In order to allow the usage of the same credentials for multiple hosts, it is possible to specify network address and mask length as ip address in column A, such as 10.10.10.0/24. The address of 0.0.0.0/0 is handled like a default route, that is, it matches all ip addresses. *PGT will process credentials in the order they are listed in file and will not consider the better prefix length match.*

The output below illustrates this feature in operation:



Although fixed credentials can be provided this way, if authentication process is using one time passwords –in case of RSA tokens for example– password will be different for each logon and it is therefore required to be entered manually. To use this approach, check the "Prompt for device credentials" option.

The default authentication type used to log in devices can also be specified for a profile. It is not possible to specify authentication type individually for each end host.

If multiple, subsequent commands refer to the same host in the scrip they can be automatically ignored should the connection at the first command fail. This is controlled by selecting the „Auto ignore host" checkbox.

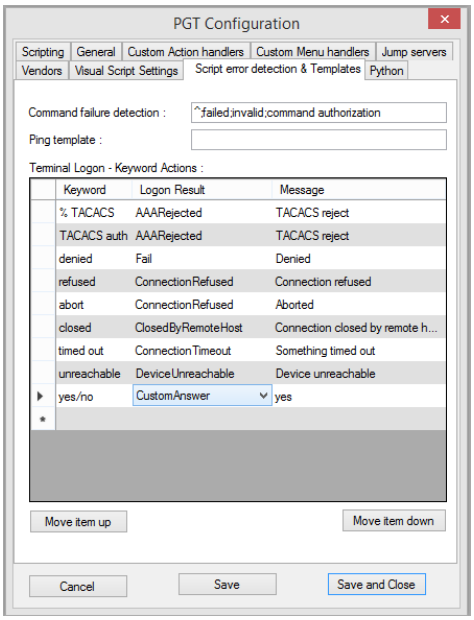
When the „Test host reachability" checkbox is checked, the program will try to ping the end hosts from the jump server (or directly if no jump server is specified) before trying to connect to them. This can save time as telnet/ssh timeout can be very long. On the other hand, this may also result in ignoring valid hosts when icmp echo requests are dropped by the end host while telnet/ssh would be allowed.

The Script line unchecking policy determines the way the lines of the PGT script are unchecked after executed. Possible choices ares:

- None : Not any lines will be unchecked automatically;
- Success : line is only uchecked if execution was successful;
- Error : line is only unchecked if execution failed;
- All : this is the default setting. Each line is always unchecked after execution.

Some basic logic is programmed into PGT to supervise command execution. If a command result contains any of the characters, words, or expressions listed in "Command failure detection" then a warning dialog is displayed whether to continue script execution. As a default, the ^,% characters and the following words (expressions) "failed", "invalid", "command authorization failure" set here. This mechanism may be ignored by using the IGNOREERROR directive as described in chapter 18.

Please note that the ^ character is usually present in banner configuration part of cisco devices, and therefore - with default settings - executing the command "show run" PGT will warn you that an execution error occurred. This can be avoided by removing the ^character from the "Command failure detection" strings.



When PGT is logging in to a device it can detect several logon conditions by identifying keywords and reacting a predefined way. The list of Terminal Logon – Keyword Actions is used to define the keywords. Keywords are matched in the list order. Should multiple keywords be matched, only the first matching is considered. The logon result returned can be chosen from a predefined list and the displayed message can be customized. Some keywords, like `"denied"`, `"refused"`, `"abort"`, `"closed"`, `"timed out"` and `"unreachable"` are set by default, but they can be changed or deleted. There is a special logon result, the "CustomAnswer". When this logon result is used, PGT will send the text defined in the Message column in response to the detected keyword. For example, handling the "yes/no" prompts works this way. This mechanism can be used to define custom dialogues.

If all keywords are removed, PGT will restore the default list of keywords automatically.

Pay special attention to timeout values. In a real environment one can expect quite long response times, especially over a congested management connection. In these cases it is wise to increase these values to prevent execution timeout issues.

When the connection to a host fails and the "Interactive logon on failure" checkbox is set, a terminal window will be opened to where it is possible to interactively log on to the device. This interactive terminal windows has two buttons : Ok and Cancel. The Ok button should only be chosen when logon was successful, otherwise logon session should be cancelled. This interactive logon procedure will only be triggered in the following authentication error scenarios :

- Logon Username/Password was not accepted
- Password rejected by the remote device
- Username is requested once again due to some error
- Enable access denied or failed
- TACACS authentication error detected (see TACACS error detection above)

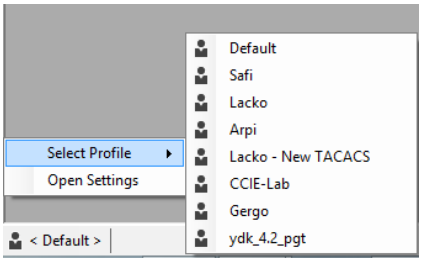
Depending on the "Quick recovery" state, when PGT is used with a jump server and does not receive response in the expected timeframe, it will drop current session and try to establish a new jump server session, rather than waiting for long timeouts (such as password failures). This mode of operation is only performed when :

- Privileged mode required but enable authentication fails
- Only password is requested upon connecting to the device, but password is not accepted

A given set of the above parameters is called a configuration profile, and can be given a name. PGT supports multiple profiles to be stored and re-used as needed. Almost all of the settings are profile specific, except just one settings:

- The working directory: it stores the file system path where the configuration file is searched for and terminal logs are created.

Profiles stored in the current configuration file can be switched quickly from the status bar by clicking the icon showing the currently selected profile:

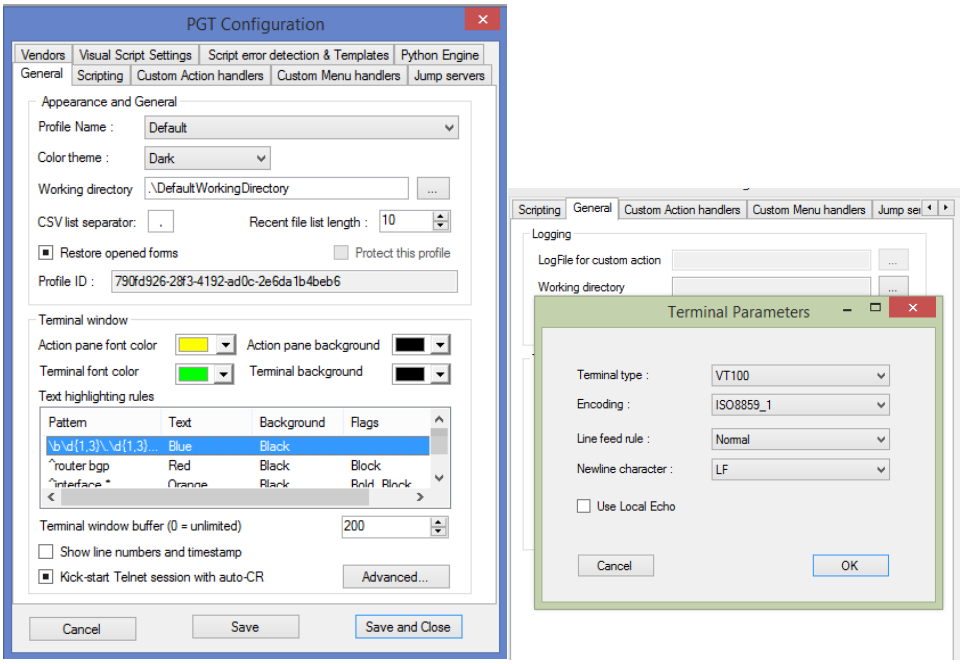


As profiles contain sensitive data, one can select to protect a profile. When a protected profile is accessed the first time, PGT will prompt the user to validate access right by entering a password stored in the profile. If no password is stored in the profile, access validation will not be carried out. It is also a restriction that the Default profile cannot be set as protected. As a result, it is advisable to create individual profiles to protect user credentials. *Please note that the cipher used by PGT cannot be considered strong enough to protect highly sensitive passwords, and it is strongly advised to apply some other type of protection in these cases.*

Sometimes it is vital to retain the terminal output log of sessions. If "Enable terminal logging" is checked, PGT will log all terminal output to a file, separately for each scripting window. The log file is created under the folder specified for Working directory (see general settings)

15.2 GENERAL SETTINGS

On the General tab it is possible to change the colours of the terminal and action windows, and set a working directory where PGTConfiguration.xml is searched for and terminal output log files are created. If the working directory is not set, the directory of the application is used by default.



The CSV list separator character can also be set here. It is used when importing or saving CSV script files. The Terminal windows buffer specifies the number of terminal lines displayed.

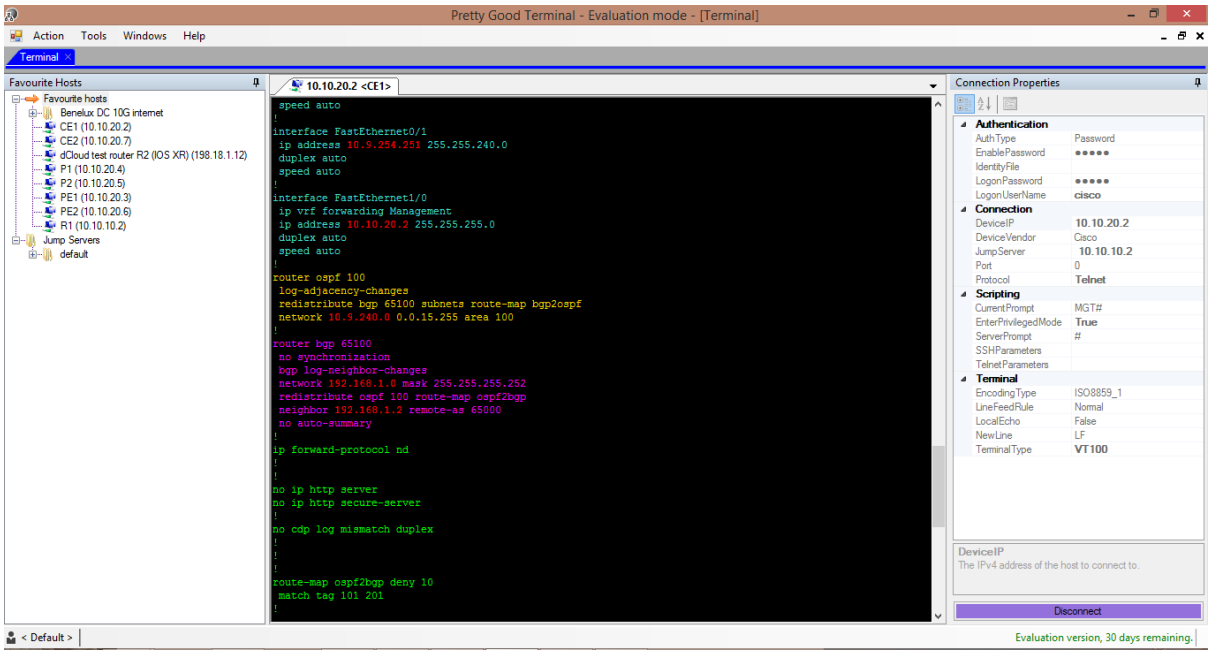
If the “Show line numbers” checkbox is ticked, terminal pane of the scripting form will display the line numbers and timestamp of the current line.

Some terminals will not send back any data when a connection is established and this would put extra delays – depending on timeout values - in scripting as PGT would wait for a response and it takes some time until PGT tries to send a newline character to initiate communication. Selecting the “Kick-start…” option will trigger an automatic sending of newline character to terminal connections upon session establishment. In most cases, however, this is rather just disturbing so it is switched off by default.

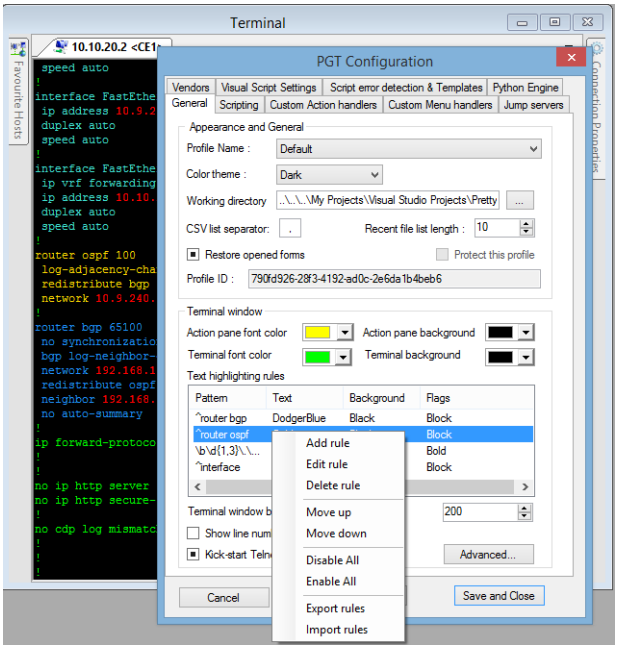
Clicking on the Advanced button, special terminal communication parameters can be selected.

The Profile ID may be used by plug-ins to uniquely identify profiles.

PGT supports terminal text highlighting rules based on regex pattern match. A rule can match a word, a complete line or an indent block and can define text colouring and format like bold or underlined characters. Rules are processed and applied in the order they are defined. Single line rules will always have precedence over block rules. A sample terminal output with the default – sample – rules, looks like below :



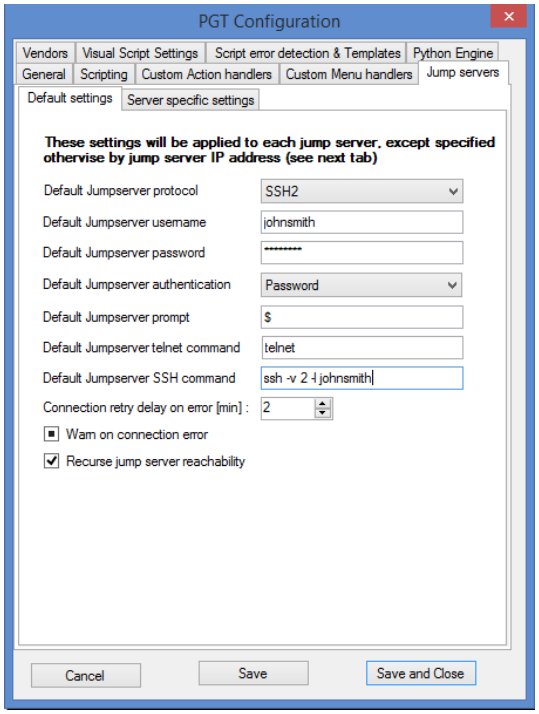
To define a new rule, export or import rules or change the order of rules use the context menu of the list showing the rules :

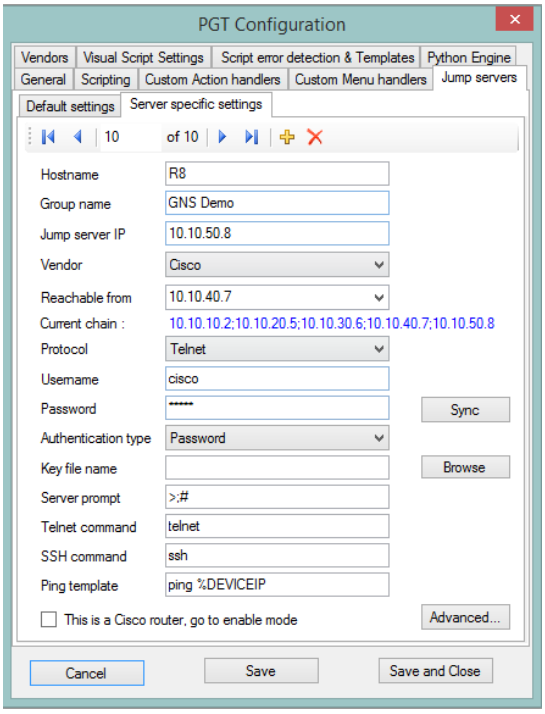


15.3 JUMP SERVER SETTINGS

Jump server properties can be set in two ways:

- Via default settings. These are valid for each jump servers, unless configured individually
- Individually for specific jump servers





Depending on the connection protocol specified in the input file, either the command given in „Jump server telnet command" or „Jump server SSH command" will be issued when connecting to the end host or the secondary jump server.

There are two main groups of jump server settings :

- Settings related to the way how the jump server is accessed. These are :
  - Protocol : defines the protocol to connect to the jump server
  - Vendor : vendor specific settings defined on the Vendors tab
  - Username : the username to use for logging on to the jump server
  - Password : the password to use for logging on to the jump server
  - Authentication type : Password or Public Key
  - Key file : the public key file path. Must be in ssh.com file format (see RFC 4716)
  - Prompt should contain only the terminating character in the jump server prompt, such as : #,>,$
- Settings controlling how to connect to the next hop from this jump server
  - Telnet command : this command is executed on this jump server if telnet is required to access the next hop
  - SSH command : this command is executed on this jump server if ssh is required to access the next hop

At the SSH or Telnet command four placeholders (variables) could be used to formulate the required connection command. These placeholders are substituted at runtime with actual values. Valid placeholders are :

- %USERNAME : substituted at runtime with the username
- %DEVICEIP : substituted at runtime with the device ip
- %CONNPORT : may be used to divert ssh connection from using the standard TCP port of 22. The Port parameter of a ConnectionParameters structure may be passed to LogonToHost command, ant the port value indicated there would be substituted to this placeholder. For more details please consult the developer reference.
- %PARAMETERS : the ConnectionParameters structure supplied to LogonToHost contains both TelnetParameters and SSHParameters fields. The value of this placeholder will be extracted from that structure by LogonToHost method. For more details please consult the developer reference.

For the variable substitution to work, it is required to have at least the %DEVICEIP configured. If this placeholder is not set in the command template then the connection command will be constructed as follows : *telnetcommand deviceip connport telnetparameters*

The purpose of %DEVICEIP – for instance - is to enable the construct of telnent/ssh command which uses a source interface or VRF for instance. In these examples, the ip address of the target device needs to be inserted in the middle of the command, such as :

- telnet %DEVICEIP /vrf purple

The above example does not require a CustomActionHandler but may be set as a telnet/ssh command template.

When specifying a jump server, one can *optionally* select a jump server already specified as the host from where the current jump server is reachable from. This will eventually create a chain of jump servers as displayed by the current chain label. The advantage of this functionality is that one can refer only to the last hop in scripts instead of having to remember the whole chain. In the background, PGT will connect to each jump server as specified to get to the last hop. This mechanism can be enabled by the selection of "recurs jump server reachability" check-box on the default settings tab.

As the example shown above, if the jump server 192.168.10.1 is only reachable from 10.10.10.12 (as shown above), first define 10.10.10.2, then select it as "reachable from" when defining 192.168.10.1. This creates the chain 10.10.10.2;192.168.10.1, and in a script it is enough to refer to 192.168.10.1 as the jump server to reach a particular end host from. PGT will then first connect to 10.10.10.2 then 192.168.10.1.

Even if this feature is turned on, one can still use multiple jump server definition in a line of a script. PGT will expand the jump server chain when it is necessary. However, the expanded jump server chain length may not exceed the number of jump servers supported in a row.

Ping template is used when the "Test host reachability" option is set. The actual ping command sent out will be constructed using this template while on the current jump server. For some extended ping commands, however, elevated mode is required on Cisco routers. For instance, to reduce the timeout value for ping, one could use the **ping ip %DEVICEIP timeout 1 repeat 3** command template. This, however, requires privileged mode. In order to make this possible, PGT must enter to privileged mode on the jump server. To indicate this, set the "This is a Cisco router, go to enable mode" option. As it suggests, this feature is only supported on Cisco platform.

Should a jump server connection fail, PGT will retry the connection at least twice, at most 5 times. At the first connection error PGT will immediately retry the connection. After the second failure PGT will wait the time specified by “Connection retry delay” field. If this value is set to zero, PGT will not wait anymore and give up retrying connection after the second failed connection.

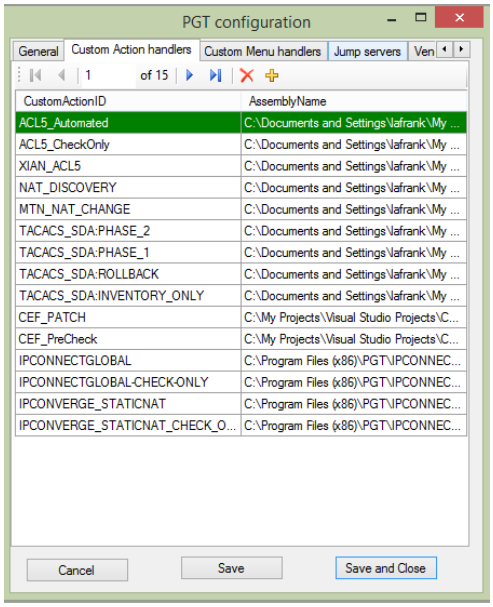
15.4 CUSTOM ACTION HANDLERS

Custom Action Handlers play the primary role in PGT’s extensibility.

Custom action handlers are basically scripts (or code) and are either external class libraries (dll files) containing code for scripting logic or Python (.py) files or Visual Script files (.vs). *Visual Scripts are explained in details in following chapters, for external class library (or plugin) development see the Developers Reference guide.*

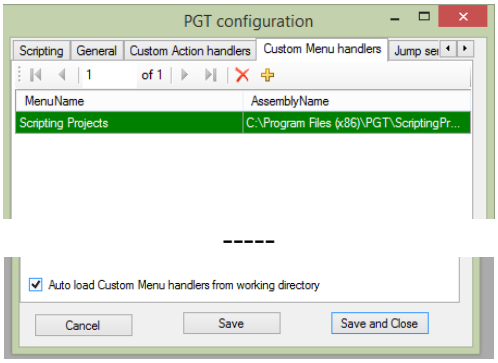
Custom action handlers can be configured by registering the relevant files as a Custom Action Handler in Tools/Options dialog. The Custom Action ID is the alias name – or script identifier if you like - for the code and is used by PGT to identify the code to be executed.

Registered Custom Action Handlers are displayed in Tools/Options on the Custom Action Handlers tab :

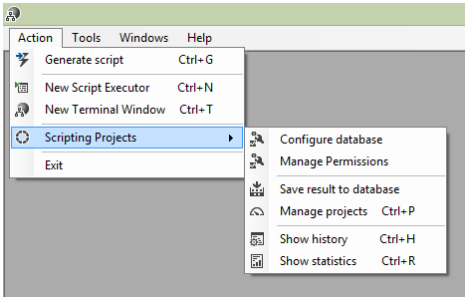


15.5 CUSTOM MENU HANDLERS

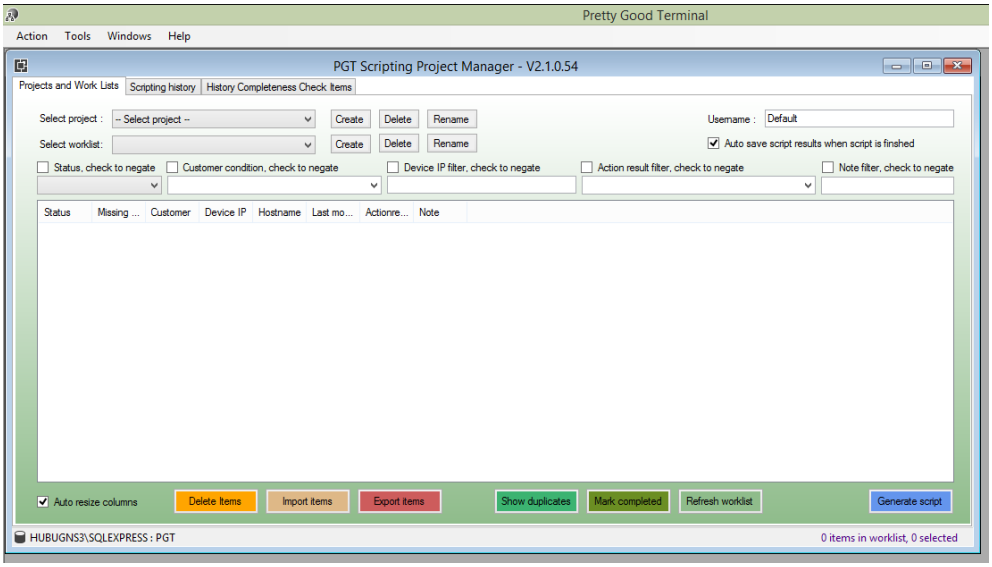
Another type of PGT extension is a called a CustomMenuHandler. They are always class libraries (dll files) and are developed much the same way as a CustomActionHandler library, but their purpose is to define a custom user interface for various tasks. An example is the Scripting Project Manager for PGT which maintains an SQL database with device list, history and status. This module has its own menu structure, forms and business logic to automate script creation and reporting. By default, PGT will automatically search for these kind of plugins in the working directory, but modules can also be registered manually by adding them to the list of CustomMenuHandlers. The following pictures illustrates this functionality:



The inserted menu structure appears in the Action menu :



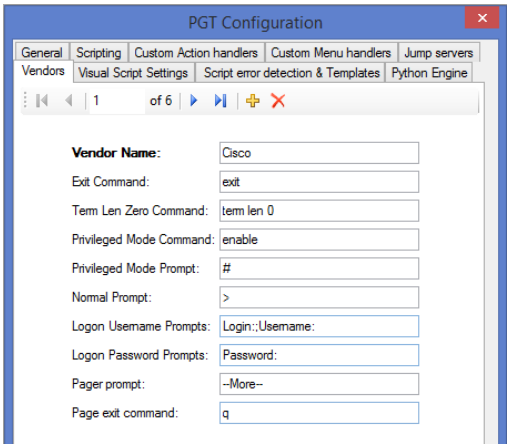
And the user interface of the module:



For more details read the developer's guide and see code examples on www.prettygoodterminal.com

15.6 MULTIPLE VENDOR SUPPORT

There are some vendor dependent settings Pretty Good Terminal is taking care of. These are required to enable correct handling of prompts and some basic commands. These settings can be found on the Vendors tab in Tools/Options:



Vendor name is the text as it should appear in scripts. This directs PGT to use the correct settings during login or command execution for a particular device. Most of these settings are quite self-explanatory. The “term len zero” command is used by PGT after a successful logon to instruct the device to not insert page breaks in its response and hence do not require user interaction.

When connecting to a console session of a virtual device it is sometimes unavoidable to receive the pager prompt, therefore PGT needs to recognize and handle it. The pager prompt setting defines the prompt text that is considered a page break, and the page exit command is the one that exits the current page. Examples for Cisco are “--More--“ and “q”. Using the proper combination of these settings can help the smooth scripting of console sessions.

The “Logon user prompts” are the prompts asking for user name during authentication. For Cisco devices it can be both Login: and Username:, depending the connection protocol used. The prompt asking for password is always “Password:” on the other hand.

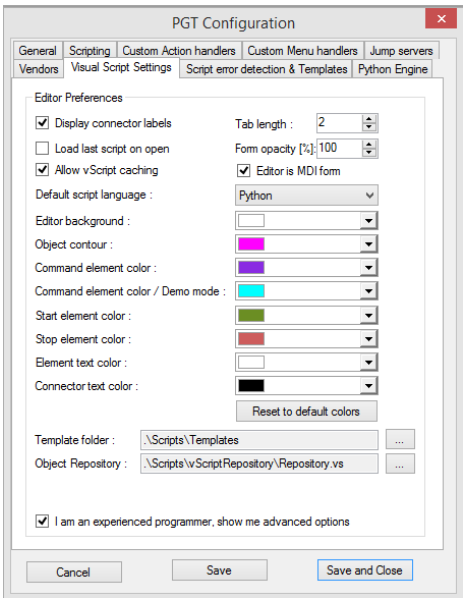
15.7 VISUAL SCRIPT SETTINGS

There are many configuration options applicable to vScripts. Besides code editor and flowchart visualization appearance, the important one is vScript caching settings.

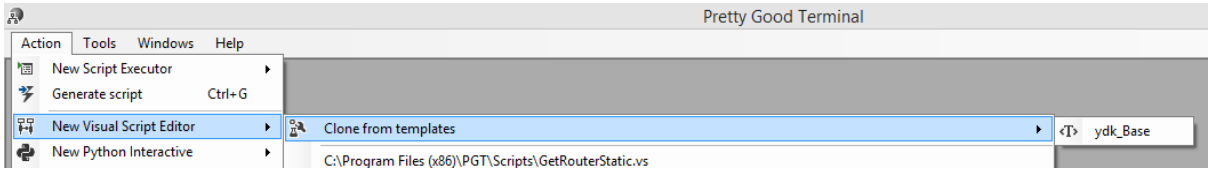
When caching is allowed, a vScript object will be created only once for a given CustomActionID referenced in a script running in a specific scripting form. When the same CustomActionID is referenced again, the same vScript object is provided.

In contrast, when caching is disabled, a new vScript object instance is created each time a CustomActionID is referenced.

Disregarding caching, separate scripting forms will always have a distinct vScript object assigned whenever a CustomActionID is referenced.



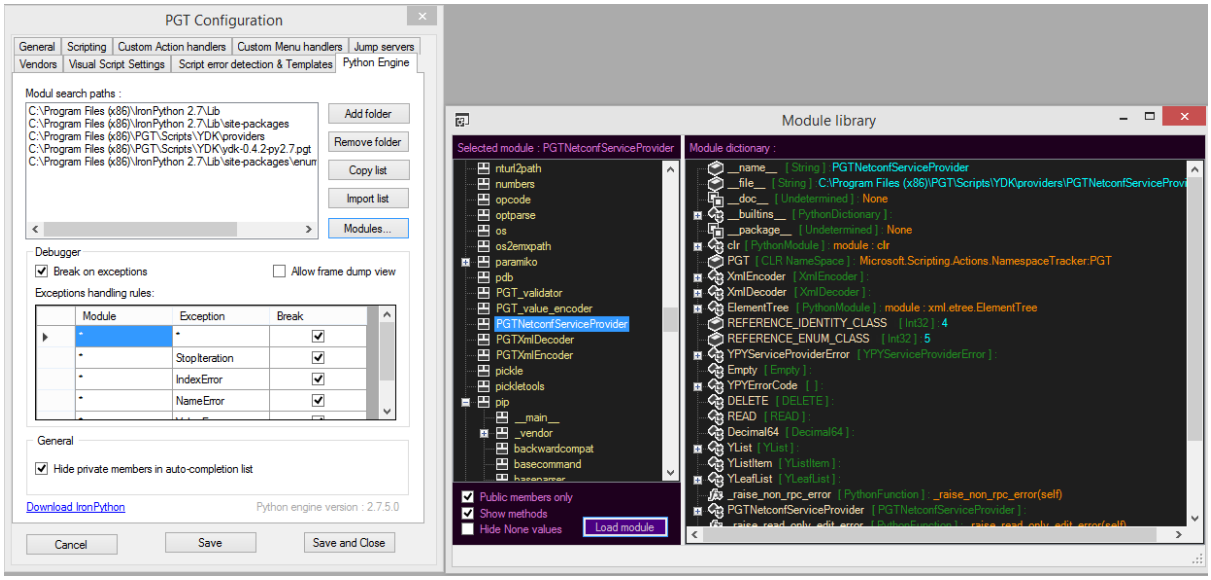
The template folder is where vScript templates are searched for. Any vScript template found in this directory (*.vst file) will be displayed under the “Clone from template” menu item in the main form :



Common visual script elements may be saved to or retrieved from a special visual script, a repository. You can defined multiple repositories but only one can be active at a time. This repository file can be defined in the Object Repository text box.

15.8 PYTHON SETTINGS

For the Python environment the module search paths needs to be configured. These settings are profile specific. When search paths are set and configuration is saved, you can use the Modules button to browse the list of discovered modules. As module discovery takes some time, please retry clicking the modules button should the output be empty.



Additionally to these common search paths, each visual script may have its own set of search paths defined from within the vScript Parameters dialog. However, to display the Module search paths tab, the advanced view options must be enabled on the Visual Script Settings tab.

With the settings in Exception Handling, it is possible to define at Module and Exception name level whether the debugger should stop when an exception is caught. The global switch “Break on exception“ controls whether rules are evaluated. With the default rule (*;*;break) one can control the default behaviour.

When the “allow frame dump” option is selected, the debugger will report each frame it captured. This is a powerful, but also a resource intensive tool.

15.9 RECOVERING A PROFILE ON FORGOTTEN PASSWORD

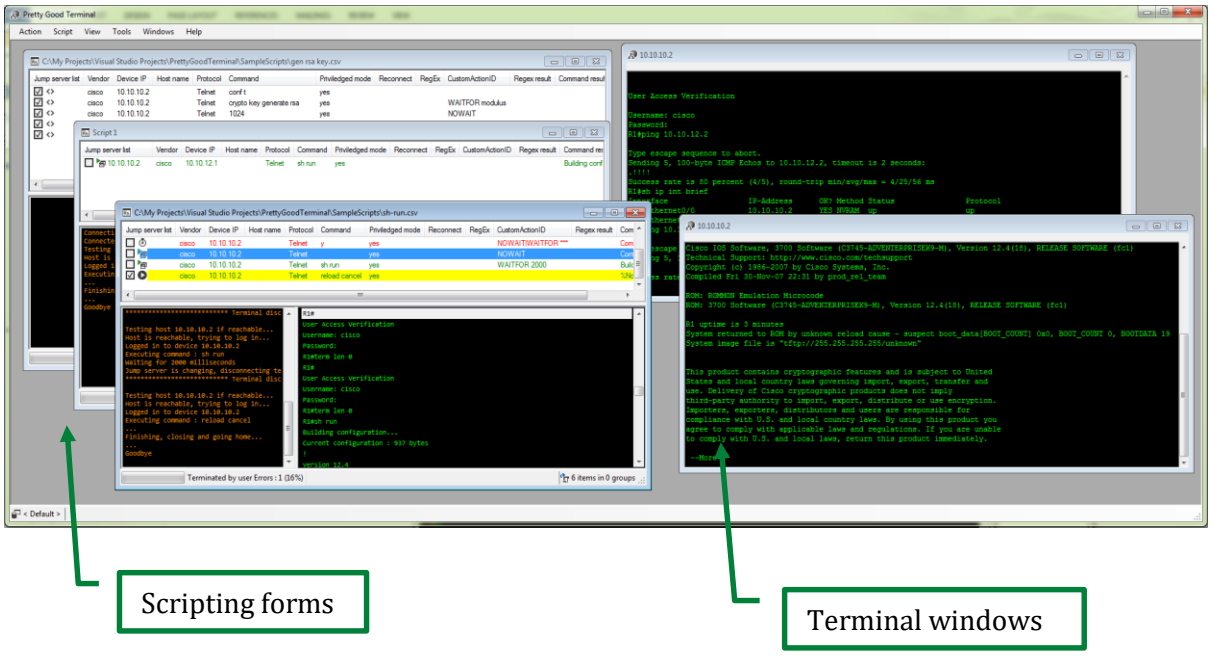
Protected profiles can only be accessed if a valid password is presented.

If the password is forgotten, it is possible to recover the profile as it still may contain important settings.

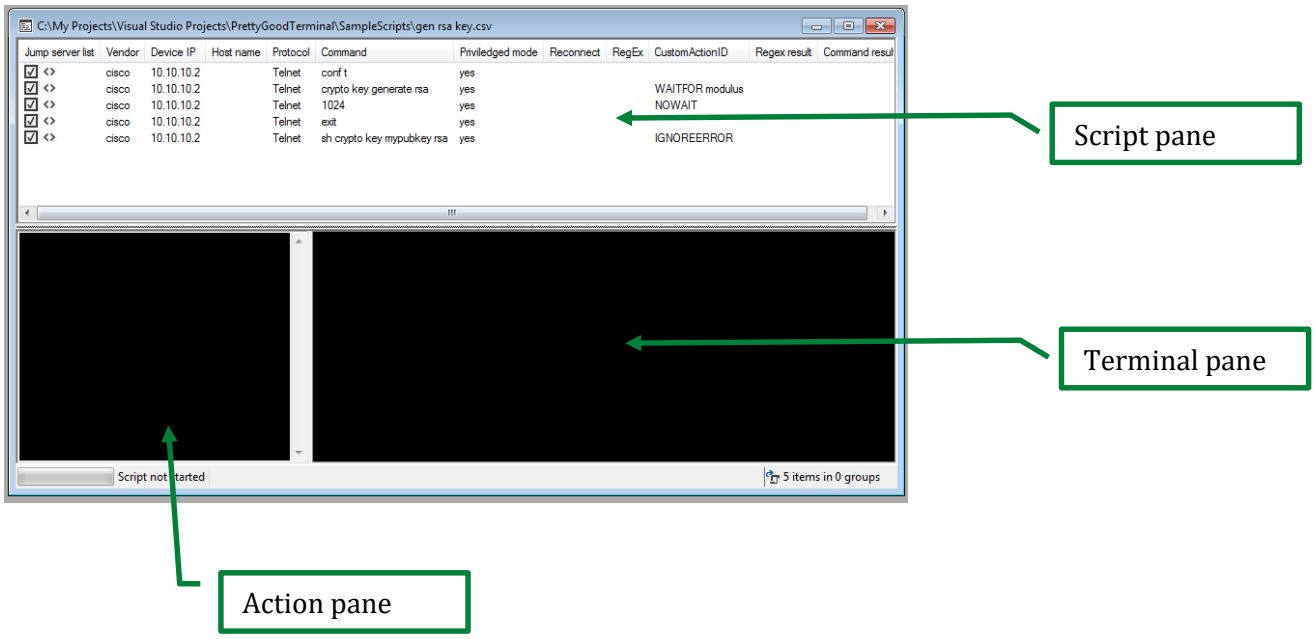
During profile recovery all passwords will be deleted from a profile which makes the profile accessible without a password. The profile recovery tool can be found in Tools menu.

16 THE USER INTERFACE

Pretty Good Terminal has an MDI user interface, which consist of the Main Window and several Child Windows. The following example shows several Scripting Forms and Terminal Windows opened :



A Scripting form has three main areas



In later chapters the description refers to these areas as the Scripting pane, Terminal pane and Action pane.

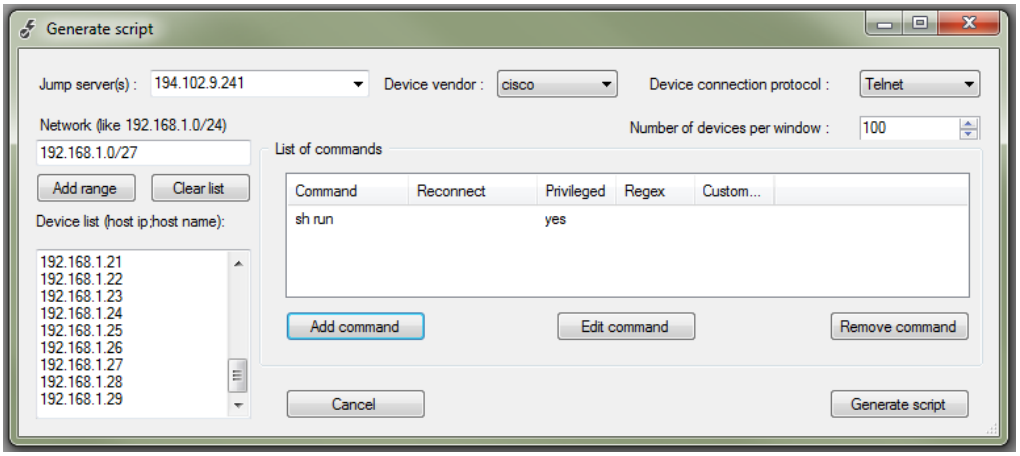
17 GENERATING A SCRIPT

In cases when the same commands are to be sent to several similar hosts, PGT includes a script generator interface. You have two options to generate a script :

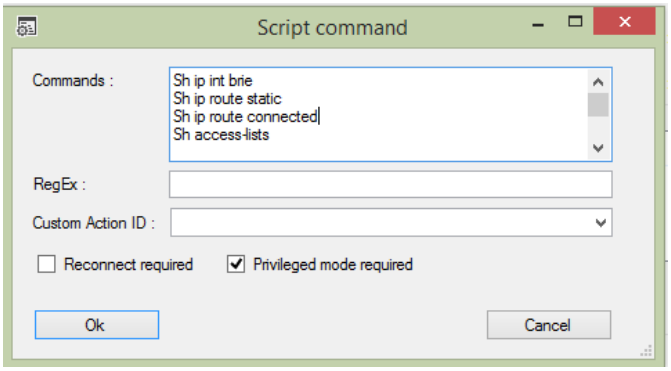
1. Once a script executor window was opened, script generator can be started from the Script menu;
2. Alternatively, you can start script generation from the Action menu before opening any script executor window

The difference is that generating a script from the Action menu will open as many script executor windows as required by the settings, whereas generating a script from the script menu will always populate the active script executor window only.

When you start script generation for the active window, the following dialog will appear:



The list of devices can be entered in a form like 192.168.10.1;switch_01, one host per line. Host name is optional, but when set, the separator character must be the same as set in Tools/Options – General Tab. All hosts must be the same vendor and support a common connection protocol. Commands can be added / modified / deleted individually. When adding commands, it is also possible to add multiple commands at a time, like it is shown below:



When a jump server is specified, specifying device ip address is not a requirement and hostnames are also optional. It is also possible to add a continues range of host ip addresses of a network.

The script will be generated directly into the opened Script executor's windows, from where it can be saved to a CSV file for later use. Once saved, it can easily be edited by Excel for instance.

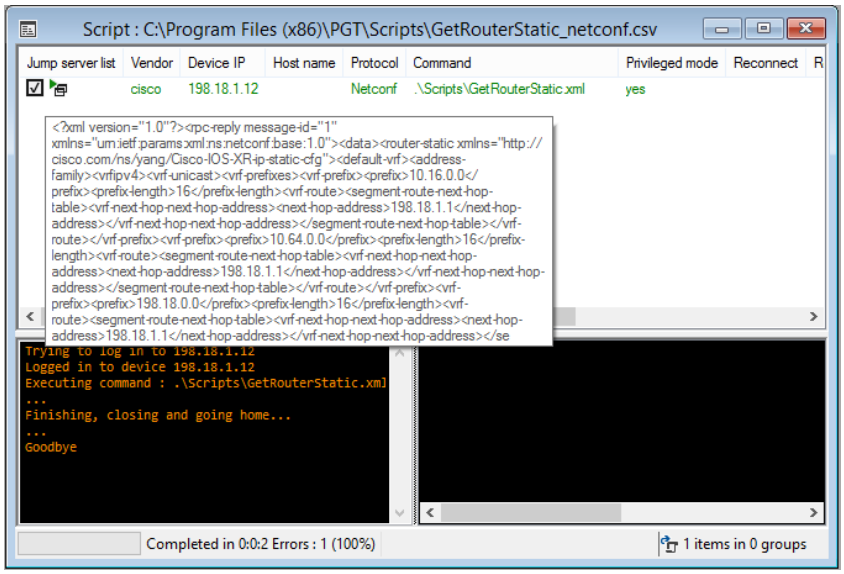
The generated script can also be edited from inside PGT, as long as it remains in the Script Executor window. To do this, select Edit script from the Script menu. Once the script was saved and reloaded, it cannot be edited by PGT anymore, but you can modify the script lines as described in chapter 20.

If you open the script generation form from the Action menu, you can control how many hosts should a single script executor contain. For instance, if you entered (copy-pasted) 1000 ip addresses and set 200 hosts per window, then 5 script executor windows will be opened.

17.1 USING NETCONF PROTOCOL

If Netconf is chosen as the connection protocol, script generation options are changed slightly. First of all, no jump server can be selected and only one command can be specified. The command must be a valid xml statement or an xml file name. Alternatively, when no command is specified, the script line must refer to a CustomActionID. If the above conditions are not met, the script line violating the conditions is ignored.

There is a sample script file named GetRouterStatic.csv in the Scripts folder which refers to GetRouterStatic.xml file. This script queries static interface configuration from an IOS XR router :



17.2 THE NONE CONNECTION PROTOCOL

A PGT script can specify “None” as connection protocol. In this case PGT will not attempt to connect to the specified device. The reason of including this protocol is to allow custom scripts to do fancy things and manage the connection on their own. The ConnectionInfo class supplied to scripts contain all information from the script line, like vendor name, device ip, jump server, etc. Other scripting parameters like passwords, usernames, wait times, prompts and many others can be retrieved programmatically by the script. For details please refer to developer reference documentation.

18 CUSTOM ACTION ID

This field is used for two purposes:

1. To specify a directive on how to process a command. More than one directive may be listed, separated by the pipe "|" character. Directives are always evaluated before command execution. Valid directives are :

   - **NOWAIT** : after issuing the specified command, the program will NOT wait for the device prompt to return before proceeding to the next script item
   - **NOTRIM** : normally commands are trimmed, that is, leading and trailing spaces are stripped off. There might be occasions (such as configuring banner text) when these spaces must remain untouched. Then use this directive.
   - **WAITFOR *number*** : execution will be paused for this much time expressed in milliseconds after the command was executed.
   - **WAITFOR *text*** : after executing the command, execution will be paused until the given text is found in the terminal output text, or until command timeout expires.
   - **IGNOREERROR** : directs the scripting engine to ignore any error condition detected by searching for a command failure pattern expression in the command response. See script settings for the details of command failure detection.

   WAITFOR *number* and WAITFOR *text* must not be present at the same time in a certain script line.

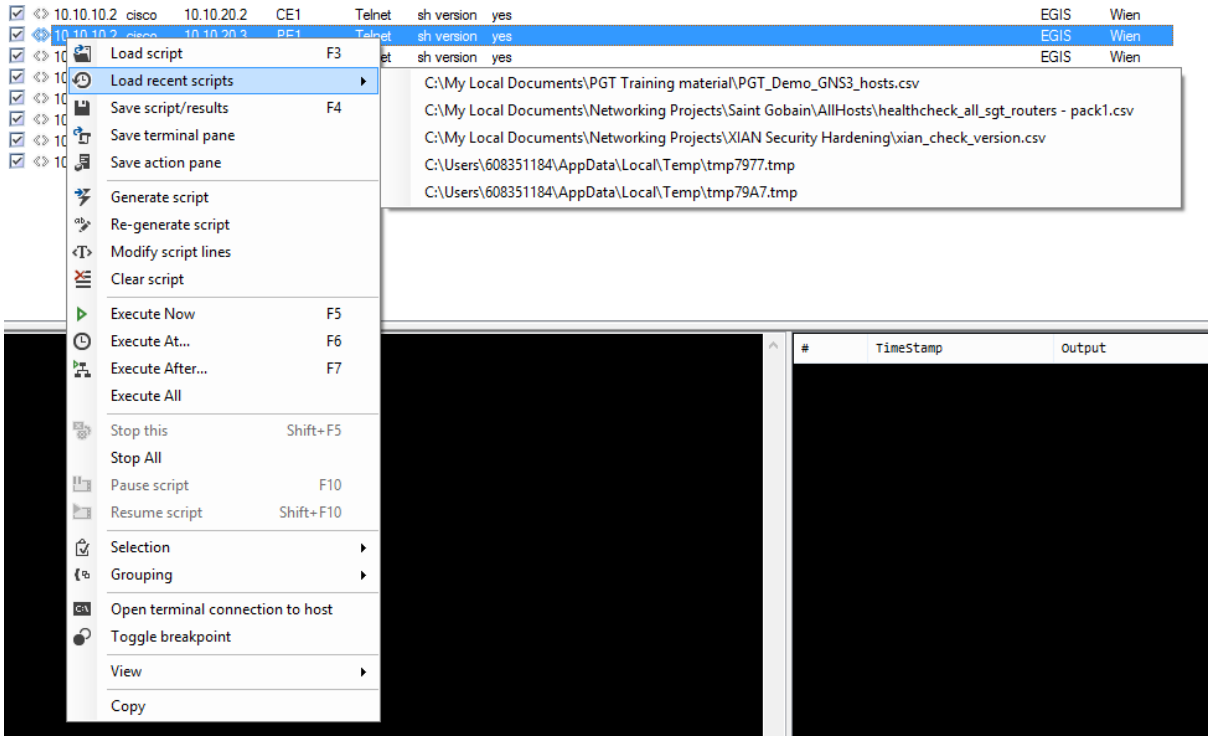
2. To specify a custom action identifier. This ID is nothing else than a string tag telling PGT what to do after the connection was successfully made to a specific device. A Custom Action ID can be mapped to a file using Tools/Options/Custom Action Handlers tab to:
   1. A class library (.dll file)
   2. A Visual Script (.vs file)
   3. A Python Script (.py file)

   This is used when a complicated logic must be implemented. In these cases PGT establish the connection to the end device and then calls the specified custom action handler.

   Visual Script development and Python script usage is detailed later in this guide. Developing an external library (a dll file) is also a simple task and can be done in any programming language which supports writing class libraries and implementing interfaces. For details see the Developers' Reference.

19 OPENING AND EXECUTING SCRIPTS

Script CSV files can be loaded from the context menu of the scripting form. The recently used ten script files can also be opened from the “Load recent scripts” sub menu :



Another way to open a script file is via drag and drop method. One file can be dropped onto the script are of the Script Executor windows, or it is possible to drop several script files onto the Main Form (not the ScripExecutor). Dropping multiple files onto the Main Form will open each file in a separate Scrip Executor and will also tile-arrange the opened windows.

Script execution can be started by pressing F5 or from the context menu. The scripting engine starts executing the script line by line, but only lines with a check mark will be executed. Each individual line can be enabled or disabled for execution by selecting or clearing the check box on it. The currently executing line is highlighted by a yellow background.

A script breakpoint can be set on any selected script line. When execution reaches a line with breakpoint, it will display a dialog and asks permission to continue the script.

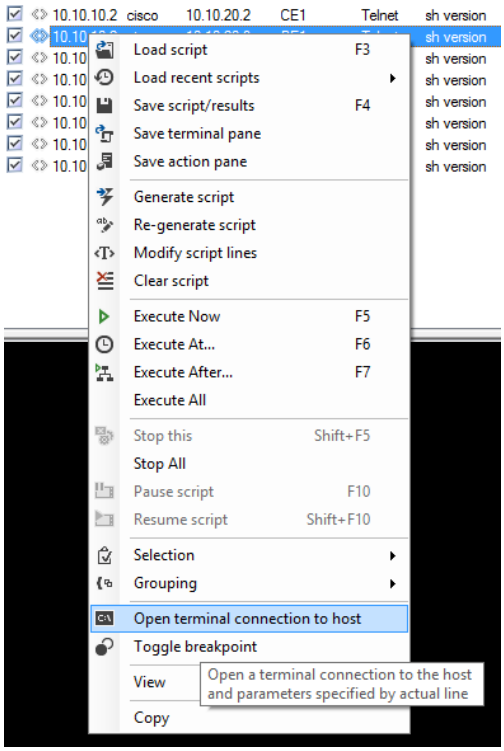
As the script executes, the executed lines are unchecked. This mechanism provides a simple way to continue a script from the location where it was stopped. Of course, selection may have to be modified when restarting a script to ensure correct sequence of commands (eg : entering into config mode is required to in order to execute a command)

Some basic logic is programmed into PGT to supervise command execution. If a command result contains the characters or expressions specified in Tools/Optins on the Script error detection tab, then a warning dialog is displayed whether to continue script execution.

As the script executes, the scripting engine is basically waits for specific responses before proceeding to the next command. The currently expected and awaited prompts are shown in the status area of the Script Executor. This may help in certain situations to find out why the script execution is paused and if a settings (such as server prompt) is incorrect.

During the execution the status bar will display the elapsed time, the estimated remaining time and the number of errors encountered so far.

It is also possible to open a terminal connection to a host with just one click, even when it is reachable through a series of jump servers. Right click on the required line and select Open terminal connection to host :



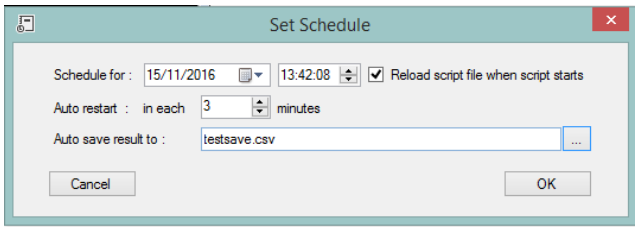
PGT will automatically connect to each jump server specified in the script line.

19.1 SCHEDULE SCRIPT EXECUTION

A script can be started at a specified date/time or after another scripts finished.

If a script is loaded from a file, the file can optionally be reloaded at the script start. This makes possible to have a script that creates a script file that is executed in another script window when the first script finished creating the script file. Of course a sample – even empty – script file must exist first that can be opened as the scheduled script.

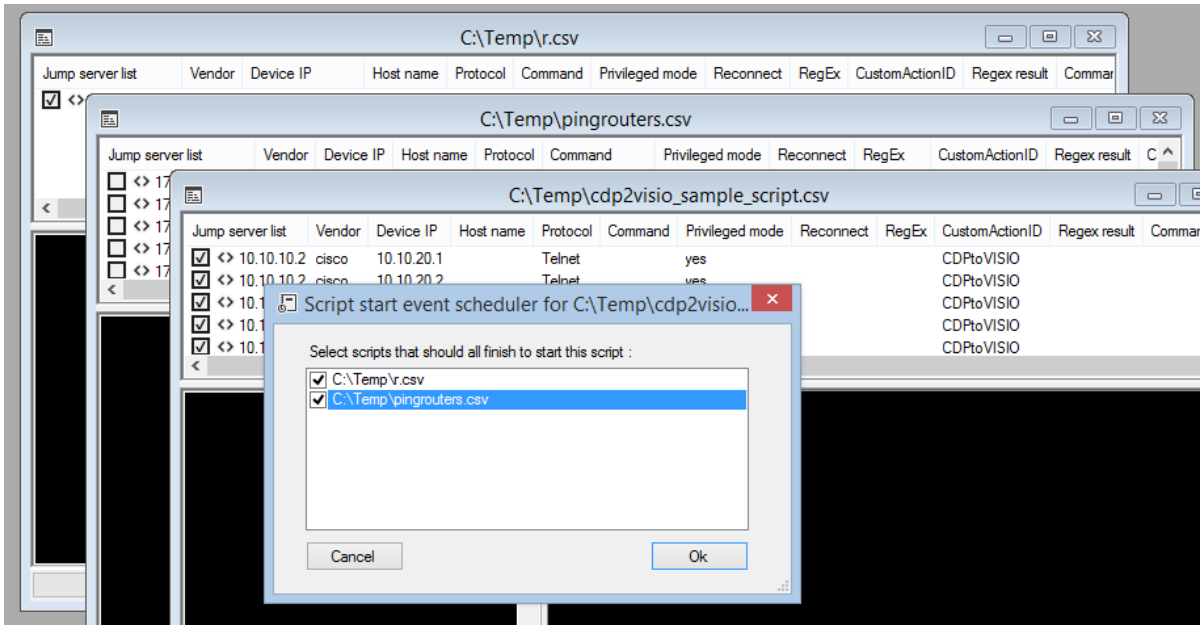
To schedule a script to start at a specified time select “Execute at…” menu item from the Script main menu. The following dialog box presented:



The “Reload script file” in only displayed if the current script is loaded from a file.

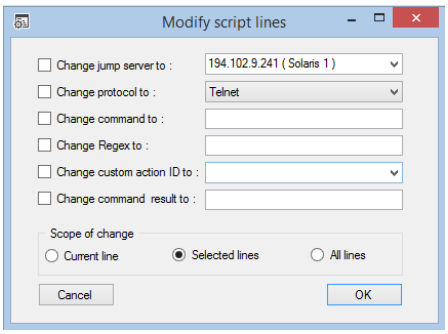
The “Auto restart” option along with the “Script line unchecking policy” in Tools/Options makes it possible to have a script run several times, until each line of the script completes successfully. For instance if a script needs to run on several devices which are not always reachable, then you can set “Script line unchecking” to “Success” and schedule the script to run repeatedly in every two hours. The script must be constructed to return Success only when finished successfully. As the script runs, all lines will be unchecked when completed and remains checked if not. For the next run, only checked items will be scripted. This repeats until all lines are unchecked, then there remains nothing to run the script against.

If you want a script to be started after another scripts finished executing, select “Execute after…” menu item and select the scripts whose termination is awaited:



20 MODIFYING THE SCRIPT

Some columns in a script may be modified from within PGT by right clicking the script pane and selecting “Modify script lines” menu item. The following dialog box is displayed :



A column’s value can be changed by entering the required value and selecting the check box for the corresponding value. The scope of the change can be set to the current line only, the checked (selected) lines in the script or the whole script.

Script can be sorted by clicking the column header. Even if not visible, script lines are always grouped by the hostname/ip address. The sorting algorithm will take care to not change the order of commands for a specific host. As a result, the sequence of script lines will always remain the same inside a host group, whatever the sorting criterion is.

Clicking on a column header which column contains the same items (vendor for instance can be like that) will restore the original order of lines.

21 CREATING DIALOGUES IN SCRIPTS

Some commands require basic interactivity from the user, such as the “reload in nn” or “crypto key generate rsa”. Using the combination of directives in the CustomActionID filed can help in these situations. Here is an example for a script to generate an RSA key :

```
1,,cisco,10.10.10.2,,Telnet,conf t,yes,,,,,
1,,cisco,10.10.10.2,,Telnet,crypto key generate rsa,yes,,,WAITFOR modulus,,
1,,cisco,10.10.10.2,,Telnet,1024,yes,,,NOWAIT,,
1,,cisco,10.10.10.2,,Telnet,exit,yes,,,,,
1,,cisco,10.10.10.2,,Telnet,sh crypto key mypubkey rsa,yes,,,IGNOREERROR,,
```

Normally the scripting engine would wait for the device prompt before proceeding to the next command. This can be overridden by the WAITFOR directive. In line #2, the scripting engine will wait for the word “modulus” before continuing to the next command. It is also a general principle to wait for the device prompt *before sending* a command. In this case it would also fail, so using the NOWAIT directive is necessary in line #3.

Line #5 demonstrates the purpose of the IGNOREERROR directive. As the command “sh crypto key...” displays a % character in the results, the scripting engine with the default settings would detect it as a failure and displayed a warning dialog box. Using the IGNOREEROR here prevents the display of the warning. Although this situation could also be avoided by removing the % character from the command failure detection expressions in script settings, it might not be the best choice in general.

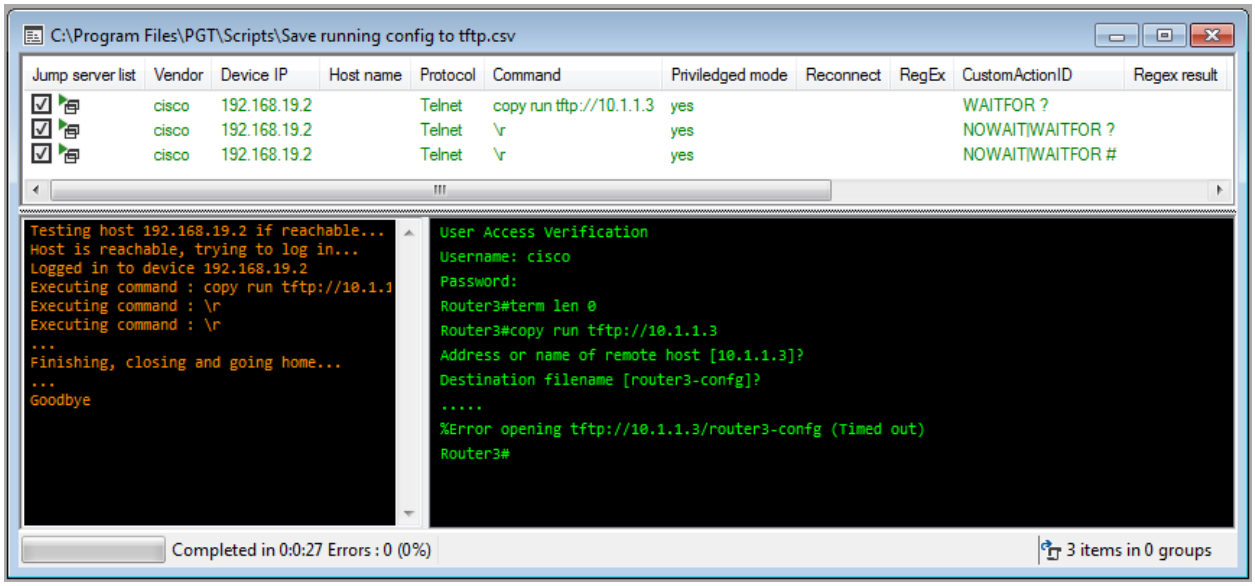
Another example might be the scheduling of a reload operation :

```
1,,cisco,10.10.10.2,,Telnet,reload in 30,yes,,,WAITFOR [confirm],,
1,,cisco,10.10.10.2,,Telnet,y,yes,,,NOWAIT|WAITFOR ***,,
1,,cisco,10.10.10.2,,Telnet,,yes,,,NOWAIT,,
```

In line#2, both the NOWAIT and WAITFOR directives must be used, because we won’t have standard router prompt to wait for before sending “yes”, and we also have to wait for the text “***” before proceeding further. At this stage the router still won’t display the standard prompt and requires an ENTER to go further. As a result, NOWAIT has to be specified along with an empty line to send an ENTER to the router.

Another common task is to send ENTER to the device for answering a question. For this purpose, one can both use an empty command, or the “\r” text as the command. PGT will send an ENTER to the terminal line in both cases.

For instance, copying the running configuration of a Cisco router to a tftp destination requires this type of interaction. The below screenshot shows a working example of this dialogue script :

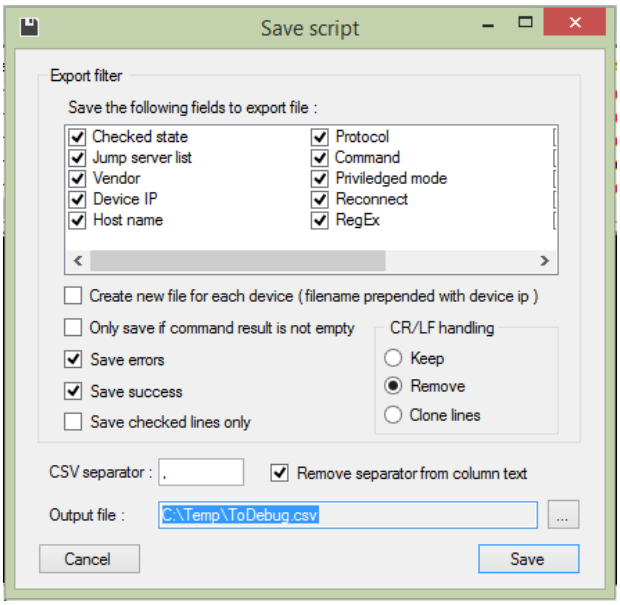


22 SAVING THE SCRIPT AND THE RESULTS

When a script finished, one probably wants to save the results. PGT supports saving the following items from the context menu :

- *The contents of the action pane* : this can be saved a simple text file
- *The contents of the terminal pane* : this can be saved a simple text file. Please note, that this output is remembered up to the number of terminal lines set in the general page.
- *The script and its results* : it is possible to filter what to save, and how to save it

When clicking on save script/results, the following dialog box is displayed:



In the first section one can select what to save, below is how to save it. It is possible to select the fields which are saved to the output file. The format of the file is a csv text file.

When the “create new file for each device” checkbox is selected, PGT is going to create a new output file for each different host. The file will be named after the filename selected as output file, but the ip address of the host will be prepended to the filename. For instance, if config.txt was selected as output then the following output file would be created for host 192.168.1.1 : 192.168.1.1_config.txt

When saving script results having multiple lines separated by CR/LF characters in the command result column (such as the output of show run command), one have to decide how to handle them in export. PGT support three possible way :

- **Keep** : in this case CR/LF characters will be retained in the output. This is not compatible with the CSV file format, and is a good choice only if the Command Result column is saved solely. For instance, one could use this feature combined with “create new file for each device” setting to collect the running configuration of multiple devices and save the configurations in separate files.
- **Remove** : CR/LF characters will be removed, this results in a standard CSV file format and the saved script can be reloaded into PGT.
- **Clone lines** : this is the best way if the output is to be filtered in Excel for instance. For each line in the multiline column text (typically Command Result) a complete line will be written with all other column data repeated. This option is only a valid choice, if only one column contains multiline text.

When a script runs, each line of the script is marked internally as success/undefined/error. This information is needed for selective operations on lines, such as save/export/modify lines/ change selection. This state information along with the colours of the lines is also saved into the output file in a special way. This information is only retained, however, as long as no external text editor is used to manipulate the script output. Keeping this state information might only be in the point of interest when one needs to stop running the script, but also wants to continue running it later from exactly the same state as it was when stopped.

23 WORKING WITH PYTHON SCRIPTS

Starting with version 6.0, PGT can host and run Python scripts. Python scripts can be run in interactive mode, or fully automatic. You will need a list of devices to connect to – so to say a PGT script – and you have to specify a Custom Action ID referring to the Python script to use.

As PGT executes the PGT script it will connect to specified devices, and upon a successful connection the execution is transferred to the referenced Python script. This way, the Python script does not have to take care of telnet/ssh connection setup and tear down, it is all handled by PGT. In fact, PGT does not support many libraries used by Python SSH library called Paramiko, but it is also not required.

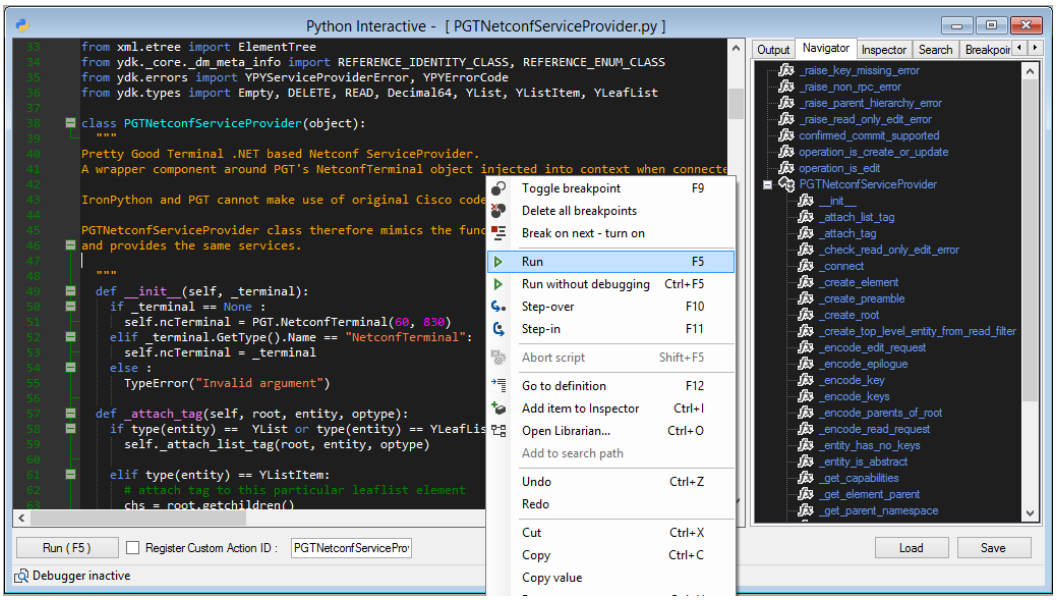
The following chapters will detail the usage of Interactive and Automatic mode of running Python scripts.

23.1 INTERACTIVE PYTHON SCRIPTS

In interactive mode scripts can be viewed, changed and debugged directly from the Python Interactive window. Although this way of using Python scripts called herein as interactive, please note that the script should not use the default input/output console in an interactive way as it is not supported by PGT.

To start a new script, open a new Python Interactive window. You can start typing a new Python script or load an existing one from a file.

When ready, you can start the script by pressing F5 / Ctrl-F5 / F10 / F11 or from the local menu of the interactive console:

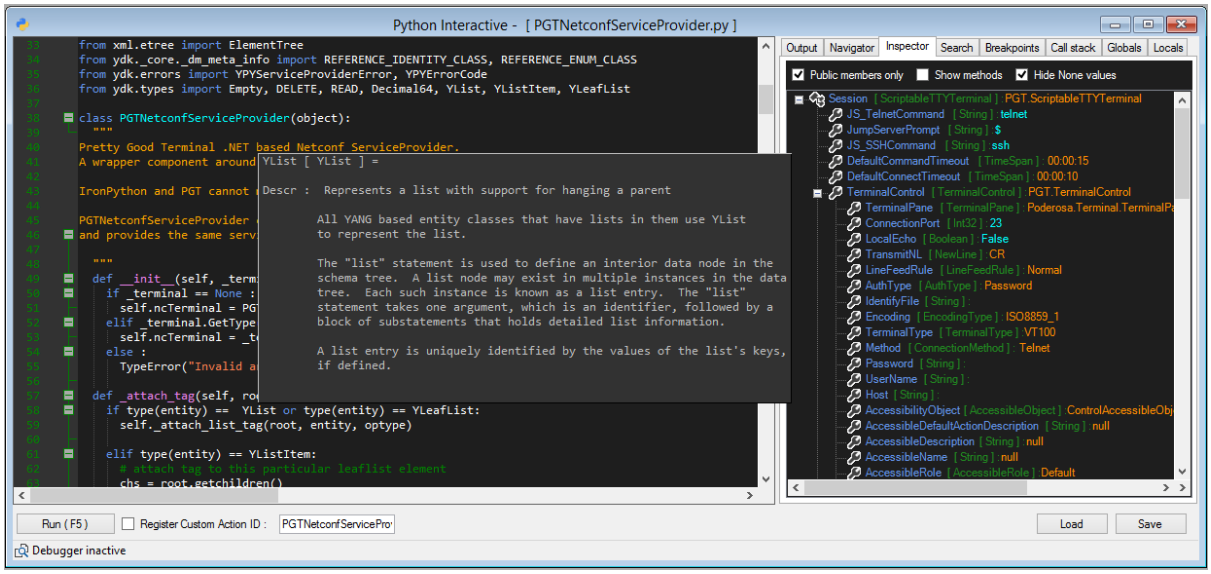


While F5 starts the script in debugging mode, Ctrl-F5 disables debugger and runs the script faster.

To debug the script you can add breakpoints and execute commands step-by-step. PGT will remember breakpoints set in a given file when the Editor is closed and next time the same file is loaded breakpoints and the last cursor position will be restored.

While in debugging, you can choose between Step-Over (F10) and Step-In(F11) operations to proceed to the next statement. When you choose Step-Over, the debugger will proceed and break at the next statement in the *current module.* Using Step-In, if next statement is located in another module and the module filename can be resolved the debugger will open the module file in a new window and pause at the next statement.

To check variable values add them to the inspector view or simply hover over a variable to check its value. With inspector you can dig deep into complex objects:



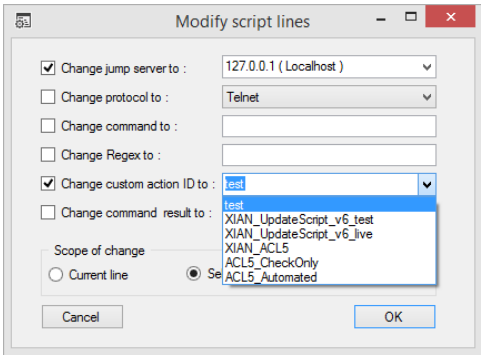
You can easily navigate in code using the Navigator or pressing F12– Go to definition – while caret is over a function / class or module name to jump to the source code of the item if possible.

In order to refer to the script loaded in the Python Interactive window from a PGT script, a Custom Action ID tag must be assigned to it. Let's name it "test" and then select the "Register Custom Action ID" checkbox.

Load an existing PGT script to a new Script Executor window and select Modify Script lines from the local context menu.

In the Custom Action ID drop-down list you will find the "test" ID assigned previously to the Python script.

Select it, and apply to all lines in the loaded PGT script.



Write some very basic Python script, like the one below. This simple test script want to introduce some return variables PGT uses to communicate with Python scripts. Although using these return variables are optional in scripts, setting them from the Python script provides an easy and elegant integration support. These variables are:

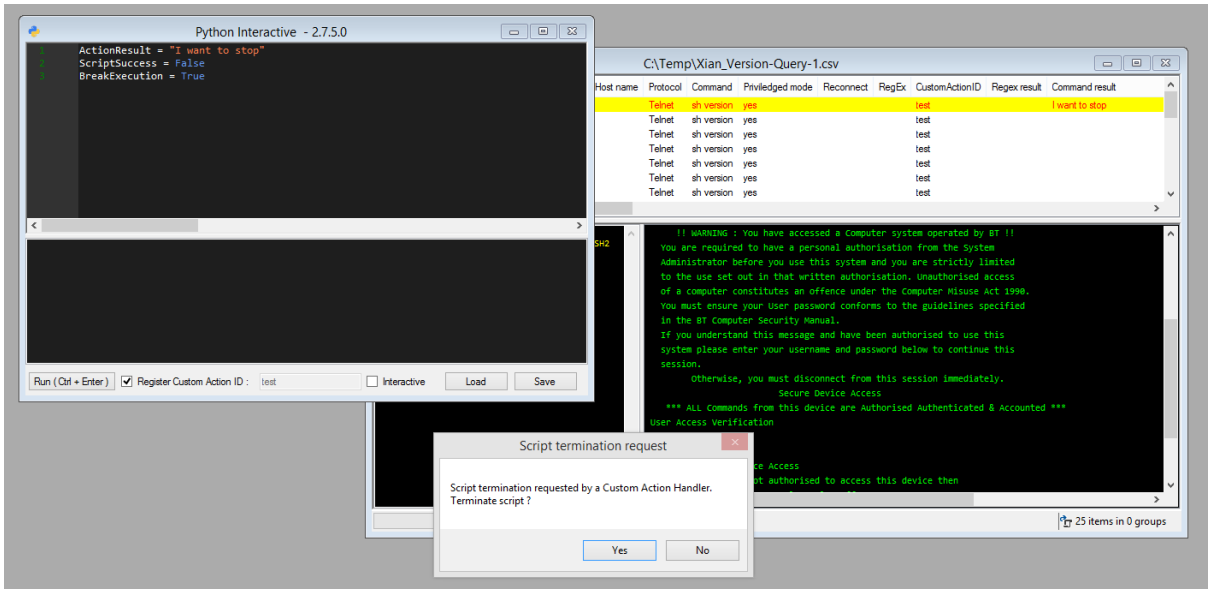
- ***ActionResult*** : string -> this is the text reported back to PGT and displayed in the Command Result column of the PGT script;

- ***ScriptSuccess***: boolean -> reports PGT whether the script was successfully executed. Depending on the returned value the line of the script will be coloured red or green;
- ***BreakExecution***: boolean -> can be used to signal the script execution engine to pause before proceeding to the next line in the PGT script. User can then decide whether to continue script execution.
- ***ConnectionDropped***: boolean -> can be used to notify PGT that some script operation might have terminated the existing connection and as result PGT should not check for device prompt after the Python script executed.

So a very basic could look like the below one:

```
ActionResult = "I want to stop"
ScriptSuccess = False
BreakExecution = True
```

If you want the Python script to be executed automatically, leave the "Interactive" checkbox cleared at the bottom, then start the PGT script. Once the above Python script executed, you will notice the returned text value "I want to stop" on the Command result column, line is coloured red as ScriptSuccess was set to False, and PGT will display a dialog whether the script execution shall be stopped:

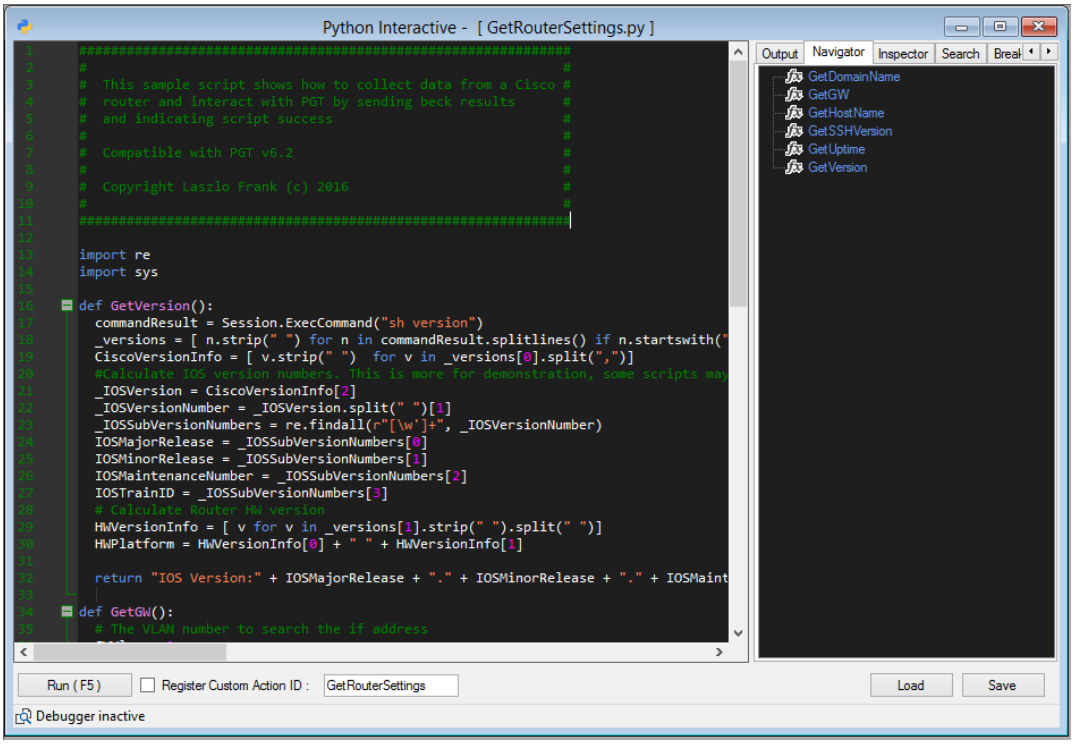


If the "Interactive" checkbox is set, PGT will not automatically execute the present Python script when execution is transferred to the Python Interactive console but the user must click the "Run" button to start the script. Running the script can be repeated several times until desired results achieved. The click the Continue button to signal the PGT scripting engine to process the results and progress to next item. Please note that the script must be finished in order to continue. The Continue button remains disabled while the script is being debugged.

Obviously, the purpose of a Python script is to interact with the connected device some way. For this purpose, the **Session variable** is made available in the Python scope. The Session object represents either the terminal line of the connected device in case of SSH/Telnet connection, or the active Netconf session and provides the functionality necessary to send command and receive the answer.

Most of the time, the **`Session.ExecCommand(Commands: Array[str], waitPrompt: str) -> (str, bool)`** function can be used for this purpose. It will send the specified command, wait for the passed prompt, and returns the result. The returned value is a (string, bool) Tuple, representing the command result text, and a boolean value indicating if a timeout has occurred. Please note the parameters of ExecCommand may vary depending on the protocol used to connect to the device.

For example, the following simple script gets the IOS version and many other parameters from a Cisco router. (You can find this script in the Scripts subfolder of PGT standard installation named as GetRouterSettings.py)



The Session object normally represents the end host, even if the connection was made through multiple jump servers. When, however, an unexpected error happens after a successful connection, or device times out, *Session object may actually represent the jumps server vty line* the same way as any terminal application would do. This can be dangerous from script perspective and it is therefore strongly recommended to make some checks in the script to ensure the script is operating on the desired host.

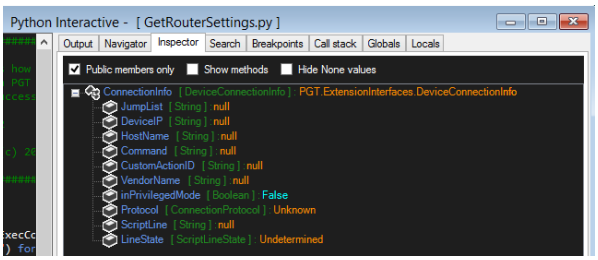
For instance, the **`Session.GetHostName(Prompt:str)`** function might be used to determine the actual host name. This function will return the host name of the connected device, assumed the given prompt string is matched.

For each functions of the Session class where Prompt argument is required, multiple valid prompts may be enlisted separated by semicolon, like the following. The argument named prompt is not actually the full prompt, but rather the prompt delimited character:

```
Session.GetFullPrompt("#;$")
```

Calling GetFullPrompt like the way above can ensure the full prompt is returned if any of the given prompt delimiter character is detected.

There is another variable pushed to Python script scope by PGT, it is named **`ConnectionInfo`**. This object represents the current connection parameters used to conenct to the device and has the following members:



1. Command: the command instruction set in PGT script.
2. CustomActionID: the ID the script was called with
3. DeviceIP : the ip address of the connected device
4. HostName : the hostname *as specified* in PGT script
5. inPrivilegedMode : Boolean to indicate whether PGT entered into elevated mode on the device when connecting
6. JumpList : the jump server list used to reach the end host
7. Protocol : the protocol used for connection
8. VendorName : the name of the vendor as specified in PGT script

23.2 RUNNING PYTHON SCRIPTS AUTOMATICALLY

When a Python script is tested and proved to work correctly in all conditions, it can be deployed in automatic mode. Automatic mode only means that a Python script is permanently assigned a Custom Action ID which can be referenced from PGT scripts.

To do so, register the Python script in Tools/Options/Custom Action Handlers tab by selecting the required .py file and assign the desired ID for it.

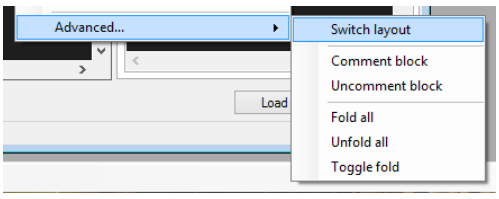
When the script is loaded the first time, the Python engine and Script Scope is initialized, then the script is executed. Depending on the object caching settings in Tools/Options/Visual Script Settings, the script scope – and hence variable values - can be retained among distinct invocations of the script. Please take care of this settings and develop / deploy scripts accordingly.

23.3 EDITOR SHORTCUT KEYS

There are a couple of useful shortcut keys handled by the Python Editor, the below table gives a quick overview on them:

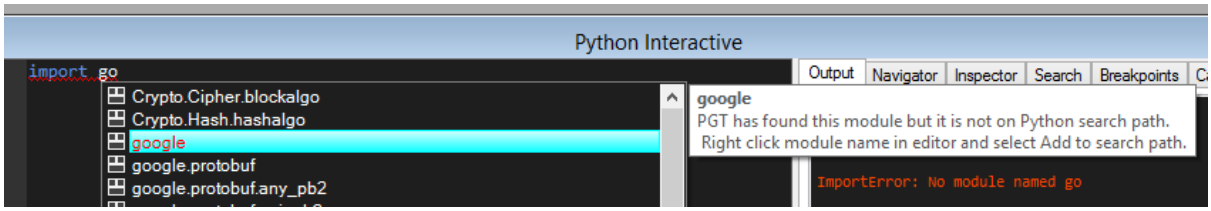
Key	**Function**
Ctrl + L	Cuts the current line to clipboard
Alt + Arrow Up/Down	Moves the selected block of text – or the current line – up or down
Ctrl + S	Saves the current script file
Ctrl + F	Switches to find window

From the editor context menu, some advanced function are also available, like switching between horizontal and vertical split layout, commenting block or folding lines.



23.4 ADVANCED PYTHON EDITOR FEATURES

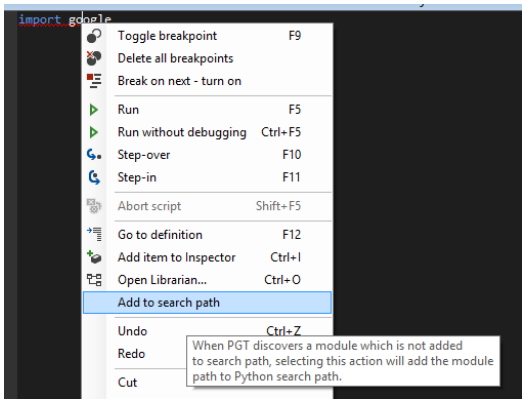
When you specify an import statement, PGT will show the list of discovered modules in the auto completion menu. Depending on settings, you may notice that some module names are displayed with red text, like the below example shows:



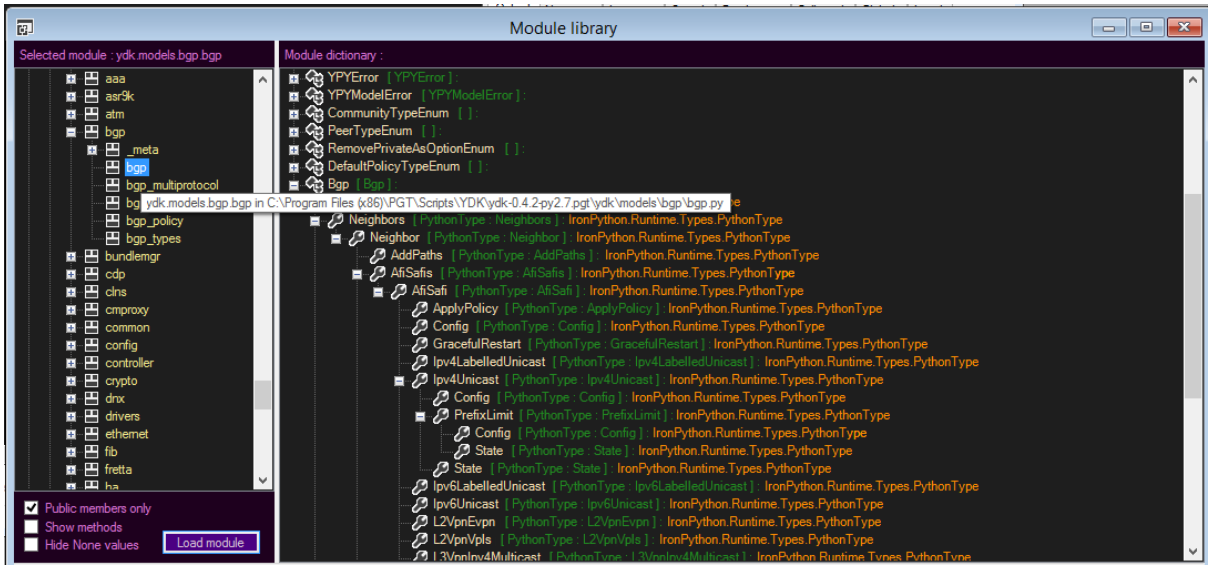
Positioning the mouse over the item you will see the explanation. The reason is, that PGT has discovered a module that is not explicitly defined on the search path.

When you finish entering the module name and right click on it, PGT will offer to include the module on the search path.

Once done, the module will later be discovered automatically.



From the editor context menu you can open Python Librarian. Librarian is basically a general purpose module explorer that will give an overview of discovered modules ant theirs contents. Although this is a general tool, it has been developed specifically to help discovering the Cisco YDK namespace and the overwhelming depths of object inside it, for instance BGP:



To help debugging, the Python Interactive editor offers four special views:

- Global / Local variables:

Global / Local variables views are hat they name suggests: they will show you the Global and Local variables of the current scope for the *current stack frame*.

- Call Stack :
  - Call stack displays the sequence of calls that led to the execution of the current statement.

- Frame dump:
  - Displays all the stack frames captured during the whole debug session. Using this feature is very resource intensive and slows down the debug process, therefore it is disabled by default. To enable, go to Tools/Options/Python Engine tab.

Both the “Call Stack” and “Frame dump” views have a context menu that allows loading the actual variables of the selected frame to Global / Local variables.

24 WORKING WITH VISUAL SCRIPTS

24.1 WHAT IS A VISUAL SCRIPT

Visual Script Editor provides an intuitive GUI for creating and debugging complex scripts with limited programming skills.

First of all, *Visual Scripts are not a replacement of PGT Scripts as described in chapter 13*. Visual Scripts are basically flowcharts compiled into Custom Action Handlers by PGT internally.

It is worth clarification that there are two distinct scripts: the script in the PGT Scripting window is named a PGT Script (or just Script), and the Visual Script, which is referred to as vScript.

Said that, vScript files can be registered with PGT and used much the same way as any Custom Action Handler described earlier.

This means that PGT will connect to a device according to a script, and once the connection is established it will hand over the execution of commands on the device to the relevant vScript based on the CustomActionID in the script. In other words, a vScript starts when a connection is established and when the vScript is finished, PGT will proceed to the next item in the script.

vScripts support both C# and Python as the internal script language. Although below examples show mainly C# code, there is no big difference if writing the script in Python or C# and basic concepts remain the same. Python vScripts have the advantage that they can be interactively debugged from PGT, while C# code debugging is only supported at block level.

24.2 CONCEPT

At the heart of the operation of a Visual Script it is nothing else than sending CLI commands to a device and processing the response to determine the next action to be taken. It is eventually a flowchart with a set of well-defined conditions to drive the execution sequence.

During the execution of the vScript the following steps are repeated until a stop condition occurs:

- A command is sent to a device
- Response is received and stored
- The response is evaluated to determine the next action

24.3 THE BASICS

This chapter is dedicated to a basic explanation of how a vScript operates. For this description C# language is used where needed, however, basic concepts remain the same whatever script language was used for a vScript. For the details of using Python for vScripts see following chapter.

Read this chapter whether you are an experienced programmer or not.

24.3.1 BUILDING A SIMPLE SCRIPT

The best way to understand what a vScript is and how it operates is to discuss a simple example. For this reason let us assume we have a list of routers and we need to update the dialer interface only if :

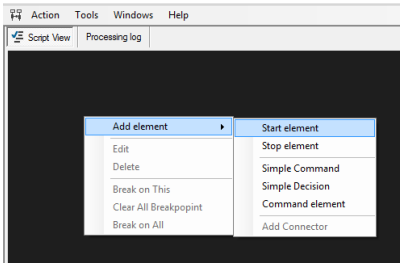
- It is a Cisco router
- Belongs to a specific BGP AS
- The dialer if bandwidth equals to 128

Without vScript, using only the conventional, CLI commands driven simple scripts this would be a challenging task. However, with vScript it is very simple, straightforward and does not even involve any programming.

Now let's see how it works, how to build and use the script.

Just open a new Visual Script Editor from the Actions menu of PGT. If there is any default script presented, select all elements by pressing Ctrl-A - or select with the mouse - and press delete to clear the workspace.

First of all, we need a Start Element. Right click the workspace and from the Add elements menu select the Start element:

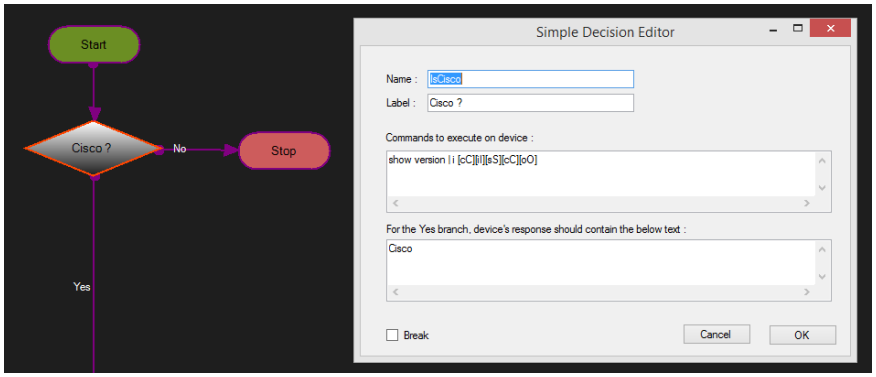


In its simplest form, there is nothing to configure on the Start element, so we can continue with adding a Simple Decision element for checking if we are on a Cisco router.

For this purpose, select Simple Decision element from the context menu shown above.

When the Simple Decision Editor appears, enter the following data to the editor :

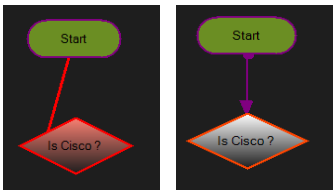
- Name : IsCisco
- Label: Is Cisco ?
- Command : show version | i [cC][iI][sS][cC][oO]
- Text to check in answer : Cisco



This will work exactly as one would expect: sends the command "sh version" to the connected device and checks whether the received answer *contains* the word "Cisco". **This is a simple decision**: the answer can be "yes" or "no". Please note, that this is a case-sensitive operation both at the router operating system (in case of Cisco IOS and also when parsing the response text. In other words, "cisco" is not equal to "Cisco" when evaluating the response. For the command, we can use the syntax *show version | i [cC][iI][sS][cC][oO]* to match any letter case but the decision element will still evaluate the response in a case sensitive way.

We also need to connect the Start element to this Simple Decision. To do so, go to Start, and select Add Connector from the context menu. The Visual Script designer gets into connection mode and as the mouse is moved over a possible connection target, the element will be highlighted:

Click on the highlighted Simple Decision element to have it connected.

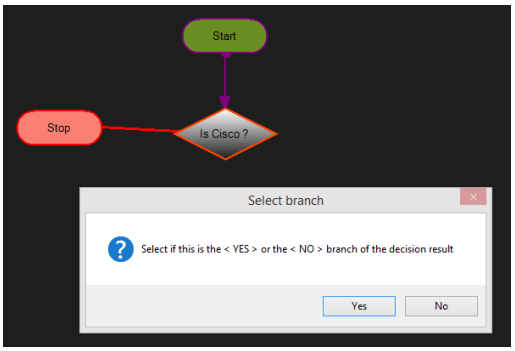


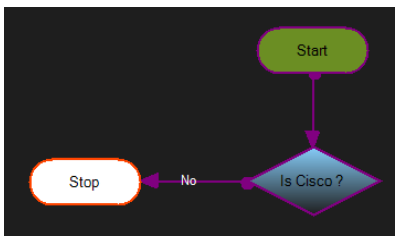
As the Start element may have only a single connection, there is nothing to configure on this connector object, so you can safely click OK I the appearing vScript Connection Editor dialog box. Now the connection is established between the elements.

From the IsCisco decision element, we have two options: in case the answer was "yes", continue to the next check, if "no", stop the script and report back the results.

Let's start with the "no" branch: first add a Stop element to the script from the context menu. Then go back to the simple decision element, right click on it, and select Add Connector. The same way as above connect it to the newly added Stop element.

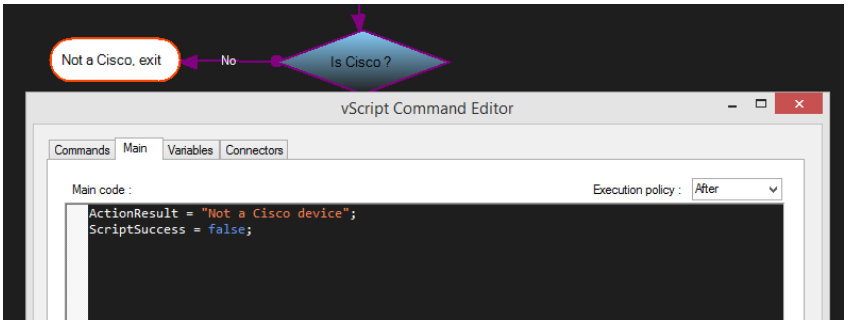
Now it must be decided whether the added connector will represent the "No" or the "Yes" branch of the decision. For this example select "No" as the goal is to stop the script if we are not on a Cisco device.





You will see the "No" branch is now connected to the Stop element.

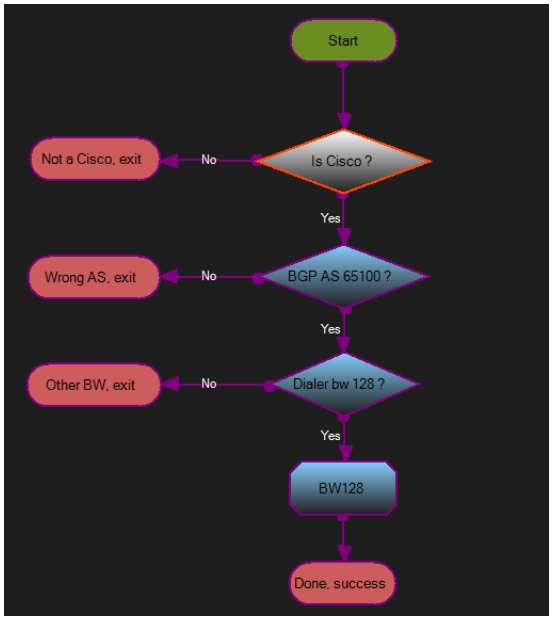
At this point we may want to report the result with text and also with a logical expression if the configuration was successful or not. To do so, go to the added Stop element and open its editor by double clicking on it. You may want to assign a display label to this Stop element to better visualize its function. Then go to the Main tab, and enter the following :



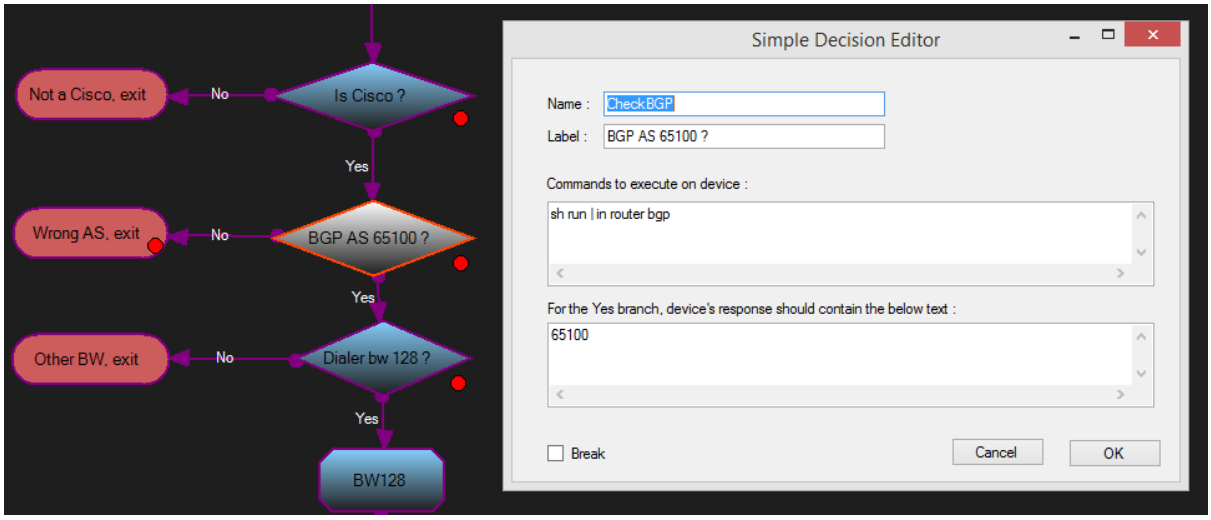
As you start typing you will notice that the editor will bring up a popup window for auto completion of the entered text. This way syntax errors can be avoided and you also do not need to remember the correct wording of variables.

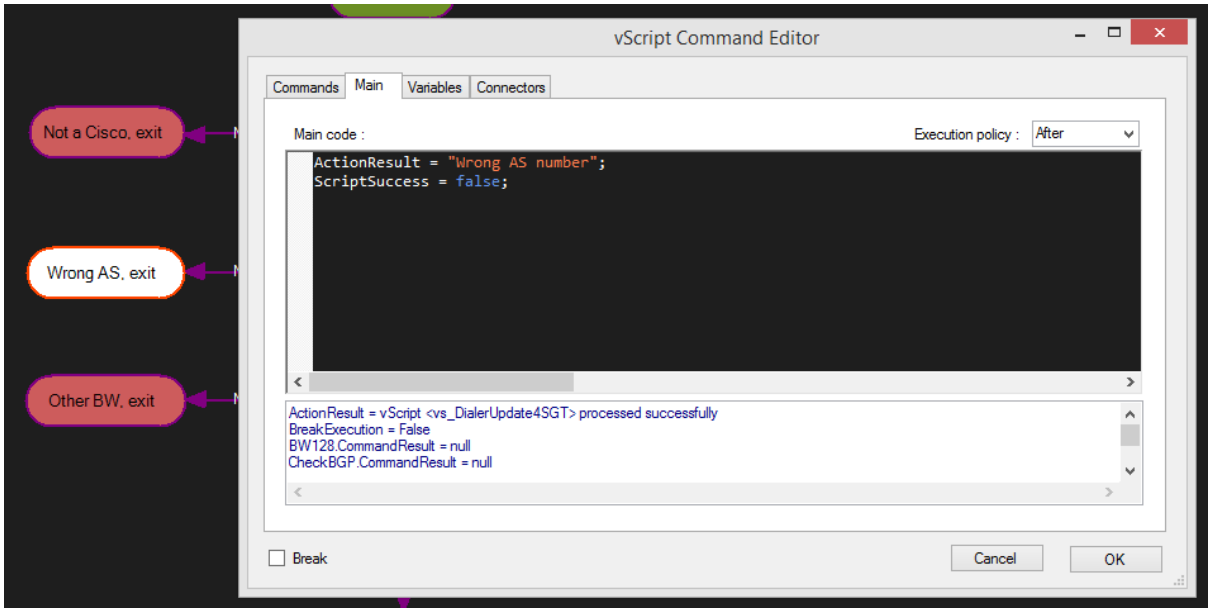
The text assigned to the ActionResult variable will be the one appearing in the PGT script window as the Command result. Depending the value of ScriptSuccess, the line in the PGT script window will be coloured green or red.

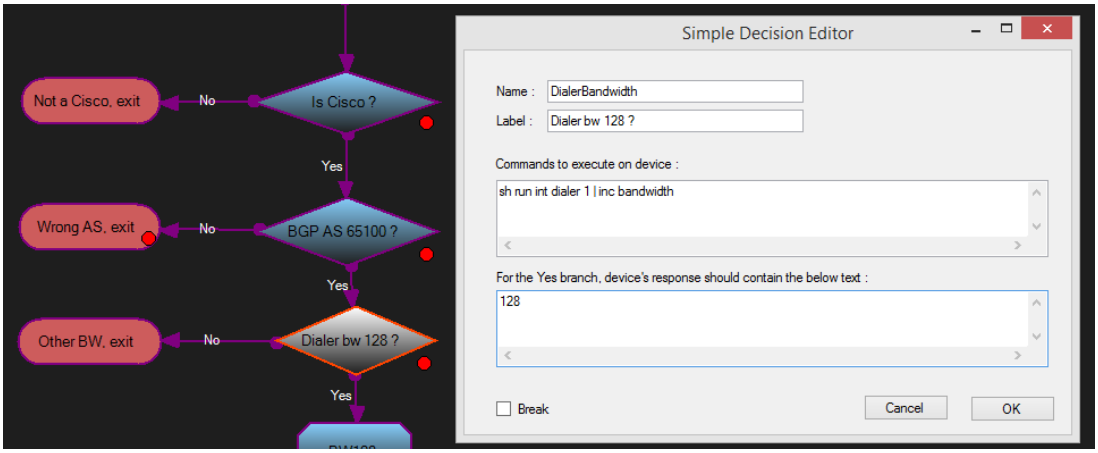
Following the steps described above, you can build the complete script to get the final one as :

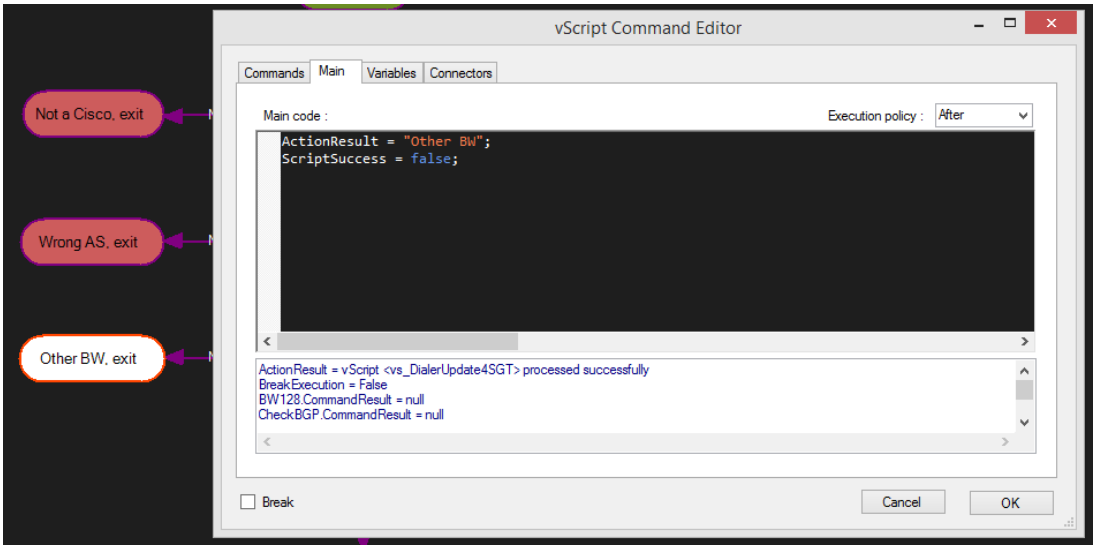


Here is what needs to be entered for each element :

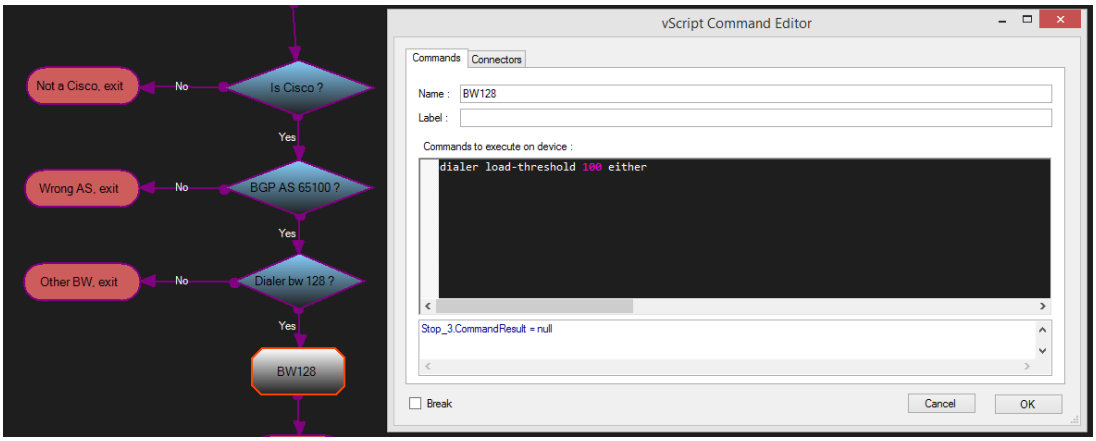

Not a Cisco, exit
No
Is Cisco ?
Yes
Wrong AS, exit
No
BGP AS 65100 ?
Yes
Other BW, exit
No
Dialer bw 128 ?
Yes
BW128
Simple Decision Editor
Name :
CheckBGP
Label :
BGP AS 65100 ?
Commands to execute on device :
sh run | in router bgp
For the Yes branch, device's response should contain the below text :
65100
Break
Cancel
OK


Not a Cisco, exit
Wrong AS, exit
Other BW, exit
vScript Command Editor
Commands
Main
Variables
Connectors
Main code :
Execution policy :
After
ActionResult = "Wrong AS number";
ScriptSuccess = false;
ActionResult = vScript <vs_DialerUpdate4SGT> processed successfully
BreakExecution = False
BW128.CommandResult = null
CheckBGP.CommandResult = null
Break
Cancel
OK


Not a Cisco, exit
No
Is Cisco ?
Yes
Wrong AS, exit
No
BGP AS 65100 ?
Yes
Other BW, exit
No
Dialer bw 128 ?
Yes
Simple Decision Editor
Name :
DialerBandwidth
Label :
Dialer bw 128 ?
Commands to execute on device :
sh run int dialer 1 | inc bandwidth
For the Yes branch, device's response should contain the below text :
128
Break
Cancel
OK



Following the decision elements, for the final configuration, a Simple Command was added to the script as :



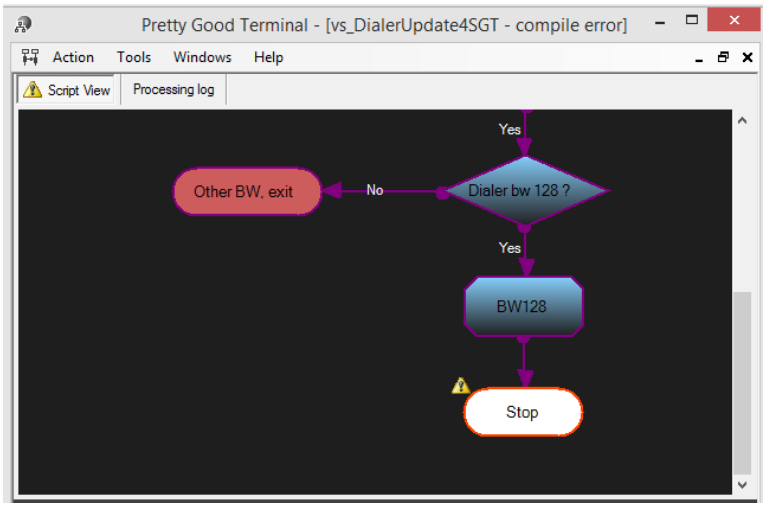
This Simple Command will then send the given, statically configured command to the device. The last Stop element will report success by setting variables as ActionResult = "Dialer IF updated successfully"; ScriptSuccess = true;

Now do not forget to save the script to a file. Visual Script files *must have the .vs extension*.

Before discussing how to make use of a vScript, one needs to understand errors in scripts.

24.3.2 ABOUT VSCRIPT COMPILATION ERRORS

How do you know if a script is error-free? Don't worry, the Visual Script editor will tell you in several ways if there was some error in the scrip. The most striking sign – if you don't like reading the error message boxes displayed ☺ - will be the appearance of yellow warning icons on the script tab, as well as on the element containing an error:



To identify what the problem exactly is, open the element with the warning sign and check the error display at the bottom. It will show the compiler error messages like this:

```
CS1002 : ; expected
CS1585 : Member modifier 'public' must precede the member type and name
```

Only from the message from unexperienced eye it could be hard to tell what the problem really is. For this reason, PGT continuously tries to compile the script as you type to text boxes. As soon as there is an error, the error message will be displayed. Right at that time, it is easier to identify the root cause of the error.

The Visual Script Editor also allows advanced error display, but this will be covered in a later chapter for Advanced Usage.

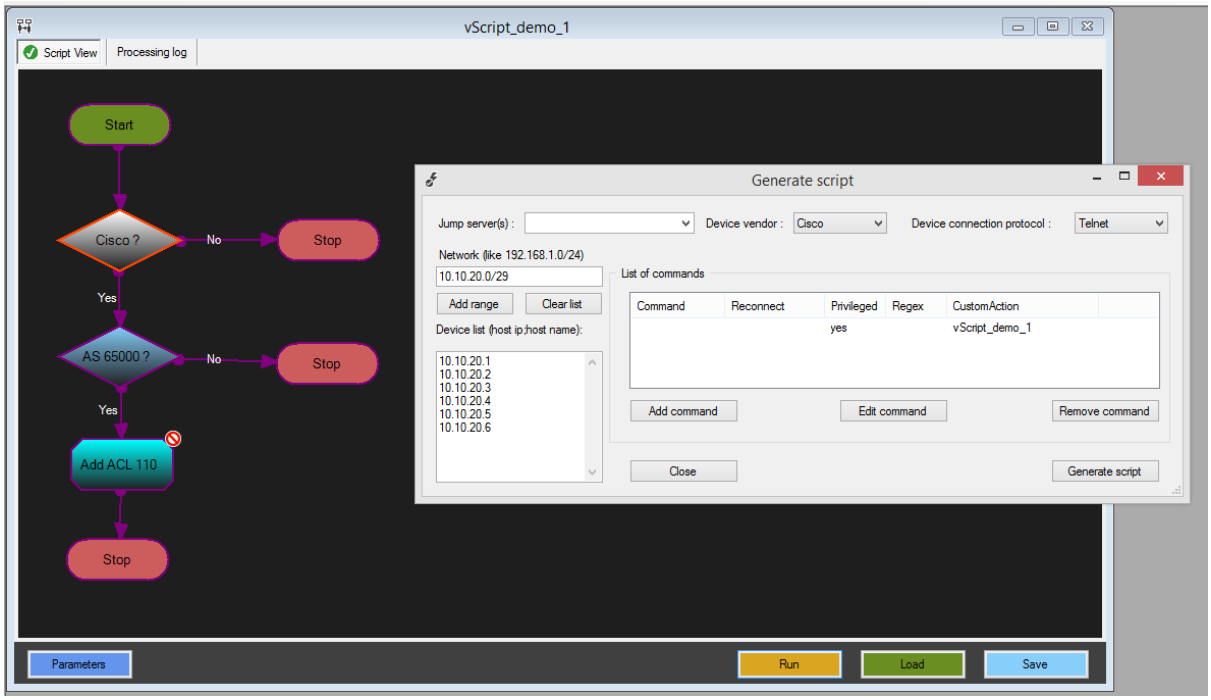
Let's see now how we can make use of the built script.

24.3.3 USING A VSCRIPT

To start using a vScript there are essentially three possibilities :

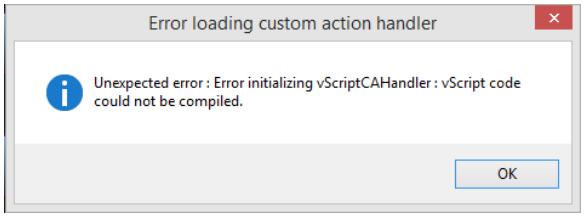
1. Press the Run button in the editor. For this option vScript debugging mode must be enabled among the vScript Parameters
2. Register the .vs script file with PGT as a Custom Action Handler
3. Keep open the Visual Script Editor window and open a new Scripting Form

For the first option the Script Generator Form will open prepopulated with the vScript name as the CustomAcionID. You then need to specify the list of hosts to run the vScript on and press Generate Script. From this point you can run the generated script by pressing F5 or click "Execute now" in the context menu of the scripting form.



To take the second approach, open the Tools/Options dialog box, go to the Custom Action Handlers tab, click the yellow add button and simply select the script file. Afterwards open an existing script file containing the required host list and set the CustomActionID to the one just been registered.

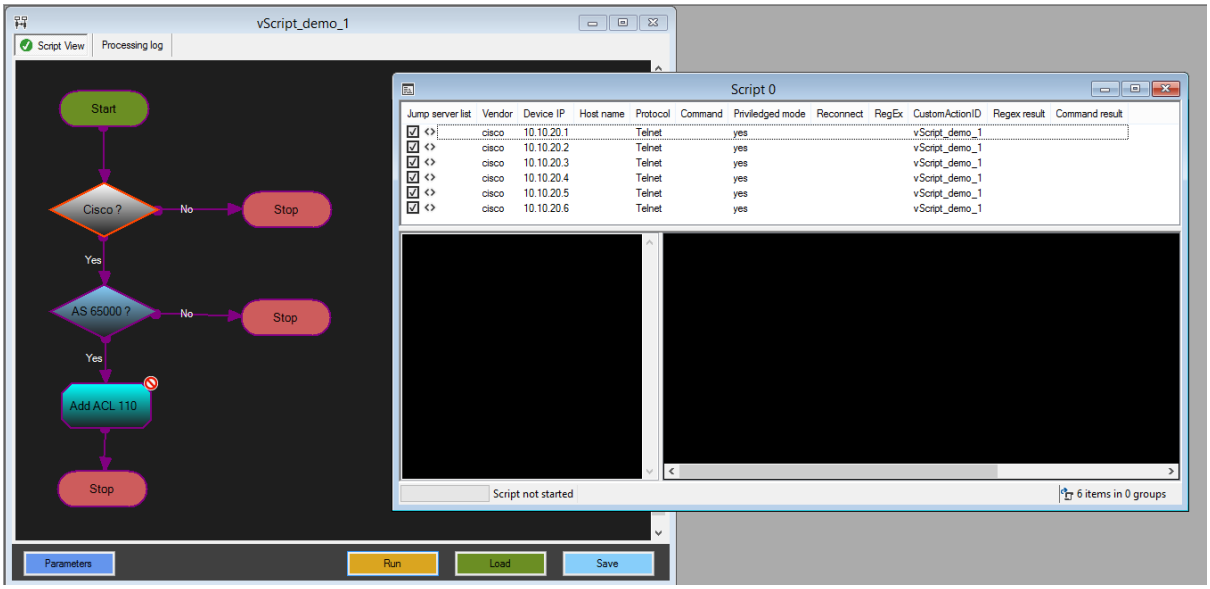
As Visual Scripts are compiled at runtime to real cod a script must be in error-free state for the Scripting Engine to load it. Therefore you will only be able to register an error-free script with PGT. If you tried to register a bad vScript, the following dialog will warn you bout the error:



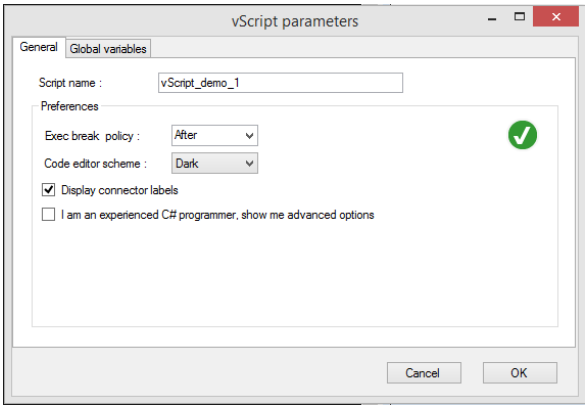
The third option is essentially identical to the first one, with the exception that it assumes you already have a script referencing the current vScript name as the CustomActionID.

Either way, a vScript is always presented to the PGT Scripting Engine by a tag, named the Custom Action ID. This ID is basically a string, and is uniquely identifying an action, more precisely, a Custom Action Handler.

You can use the Script Generator at any time to create a PGT script utilizing a vScript. The known Custom Action IDs are listed in Script Generator. In the Script Generator, instead of entering real CLI commands, select the desired Custom Action ID. Add the list of hosts and press Generate script. The following script would then be generated :



The only question is, where this ID comes from in case of a vScript ? This is actually the name of the vScript editable in the vScript Parameters dialog box :



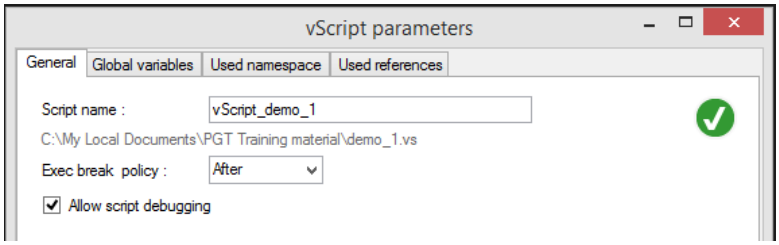
When a vScript is constructed, it is definitely a good idea to check it before deploying in a production environment. In other words, script debugging is necessary.

Let's discuss vScript debugging options provided by the Visual Script Editor.

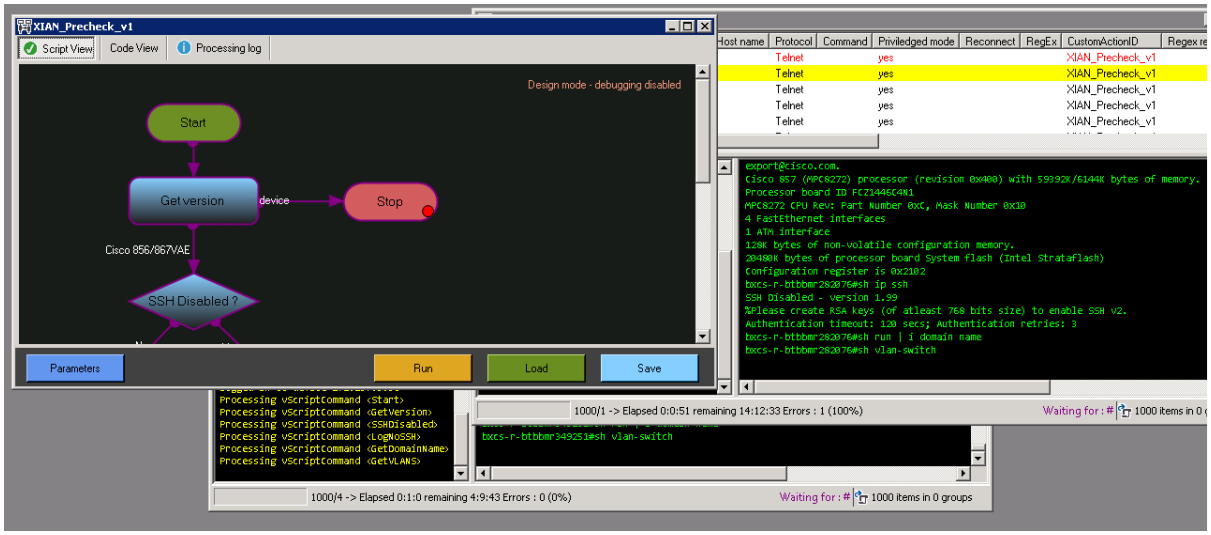
24.3.4 DEBUGGING A VSCRIPT

For C# based vScripts only block level, visual debugging is supported, while Python vScripts *also* support full featured code debugging using the Python Interactive window. For details see chapter 24.7 Debugging Python vScript code. Current chapter describes visual debugging.

A vScript can be visually debugged when the Visual Script Editor is open and the script is in error-free state. Then debugging mode can be enabled in scrip parameters:



Until debugging mode is not enabled, the script remains in design mode. This mode is also displayed at the top-right corner of the Visual Script Editor's canvas. Design mode is necessary when you want to edit the same vScript that is currently running in another script windows but you do not want to those scripts to switch the editor into debugging mode. The below screenshot illustrates the situation:

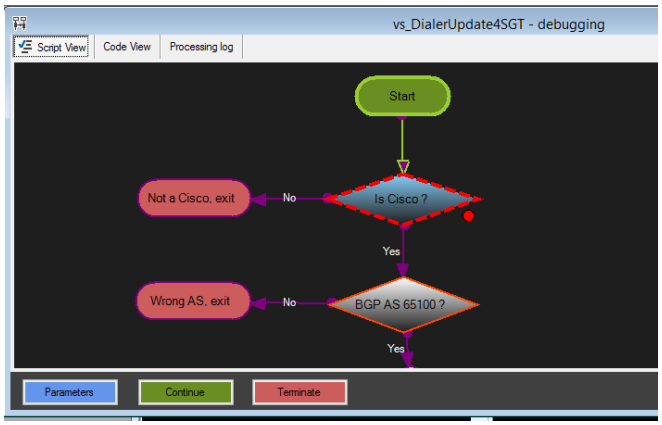


Two scripts are running using the vScript named XIAN_Precheck_v1 and this vScript is also opened for editing. In this situation you might want to prevent those scripts to switch the editor into debug mode.

To debug a vScript, click run to generate a new PGT script, or open an existing script referring to the Custom Action ID of the opened script.

Once the PGT script is started and a successful connection is established to a device and the Scripting Engine detects the Custom Action ID of the vScript in the PGT script, the Visual Script Editor will switch to debugging mode. While in debugging mode, neither the vScript can be modified, nor another vScript can be loaded.

To instruct the vScript execution engine to stop at a certain element, a breakpoint must be set on the element. To set a breakpoint, right click the element and in the context menu select “Break on this” item. A red circle will appear near the element indicating that a breakpoint is set on it.

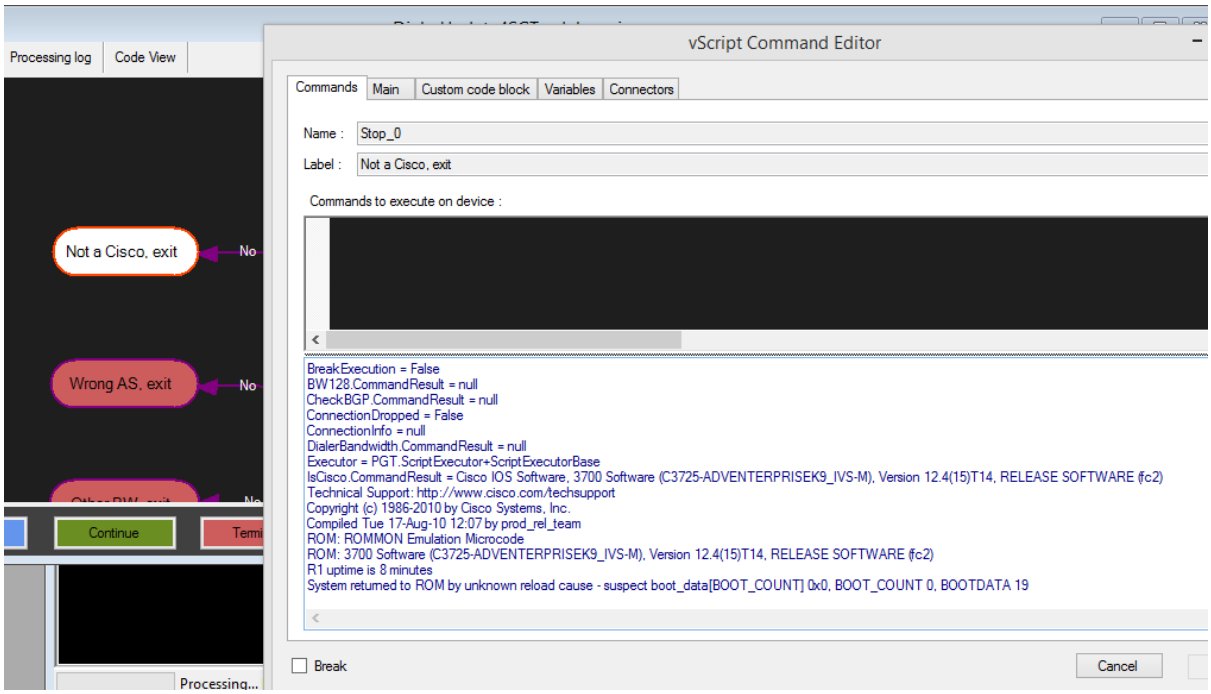


When an element with a breakpoint set is hit during the script processing, the execution will be paused, the Visual Script Editor window gets focused and the item the execution was paused on is highlighted.

The execution path up until the current element will also be highlighted to show how execution got there and also to indicate which elements were already executed.

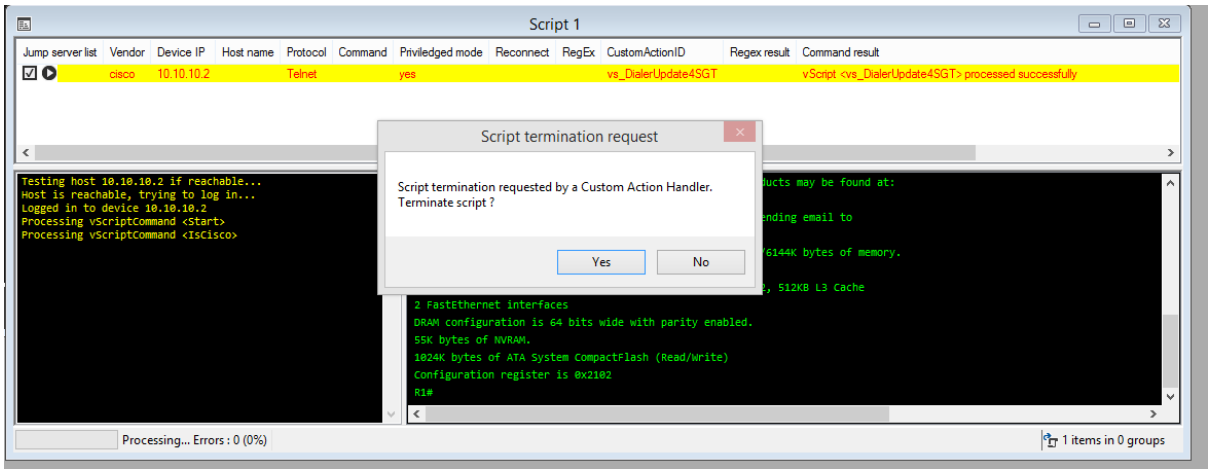
Execution can stop on an element **before or after** the element’s code was run and the respective commands were executed on the connected device. This is called the **execution break policy** which can be set in Script Parameters.

At this point, script variable values can be checked whether they contain the required value or not. To do so, open an element other than a Simple Decision – as it does not support code view at the tie of writing – to check the variables. For instance, click on the connected Stop element to open the editor:



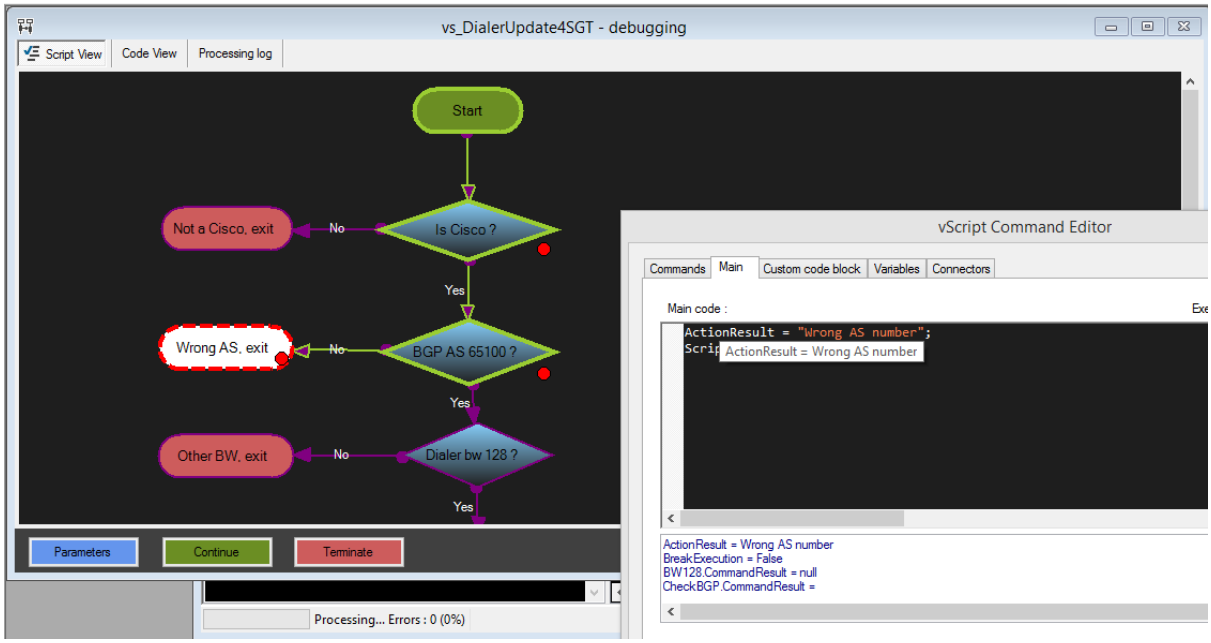
At the bottom, all known variables and their values are listed in the blue text box. In the current example we can see the value of IsCisco.CommandResult, that is, the response received from the device to the sent command – currently the “show version”.

To continue script execution click on Continue or terminate the script with the Terminate button. If you chose Terminate, the PGT Script will warn you that a script termination was requested and whether to stop execution of the *PGT Script*.

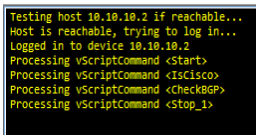


If you clicked Continue, script execution will go forward and pause when the next breakpoint is hit. The highlighted execution path will clearly show the script flow.

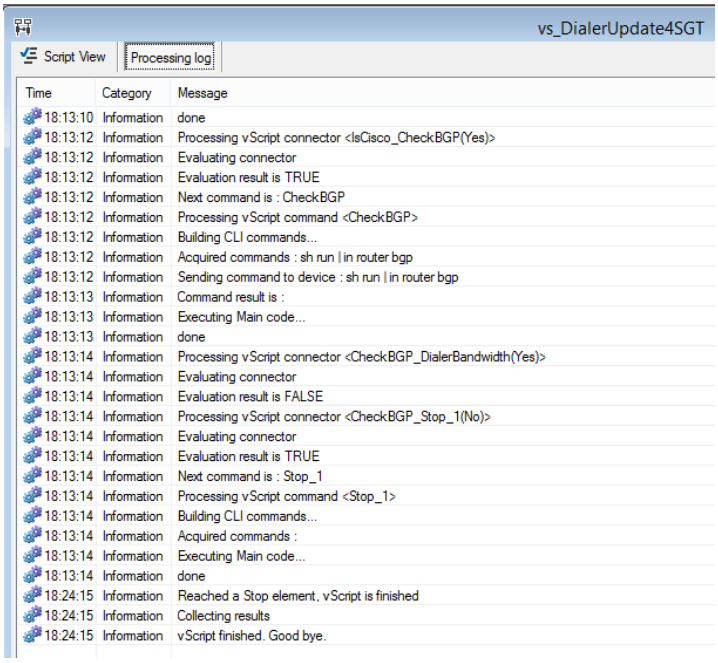
In a code editor box a variable's value can also be checked by moving the mouse cursor above the variable text, a small tooltip windows will appear to display the vale.



If you switch to the PGT script window, the Action pane will also display which vScript elements were processed in a sequence.



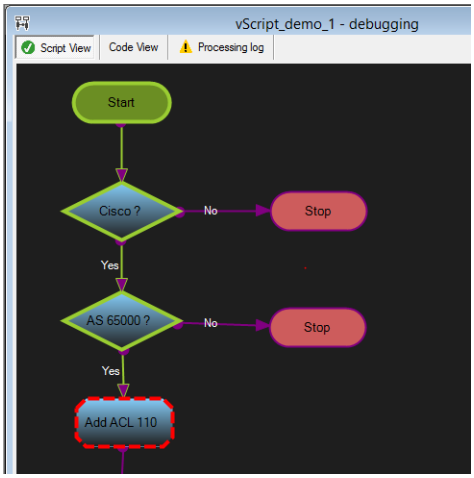
To help finding issues, a more detailed process log is also available on the Visual Script Editor 's Process log tab:



You may experience that script execution gets paused as if a breakpoint was set on an element, even if there is no breakpoint. This situation is illustrated on the right.

At the same time either a warning or an error icon is displayed on the Processing Log tab page. This can happen if a there was some command execution error or a runtime error occurred during the code execution. Look into the Processing log details to find out what happened.

Depending on the problem you can continue or terminate the script as required.



24.3.5 WATCH VARIABLES

When editing a vScript or while in debugging mode, script variable values can be added to the watch window in the Command Editor or in Connector Editor dialog boxes.

To add a variable, either highlight the text in the code editor, or just simply right click on a word and select Add to watch menu item.

To remove a variable from the watch window, select it and press the Del button.

The list of watch variables selected for each element will be saved with the script, so next time the script is loaded the same variable list will be presented.

There are two levels of watch variables in a vScript :

- Watch variables added to specific vScript elements – this is a local watch list
- Watch variables added to the vScript in the Parameters editor – a global watch list

When displaying watch variables in editor Dialogs, both the element's watch list and the global watch list will be shown.

24.3.6 HANDLING VSCRIPT RUNTIME ERRORS

Even if a vScript is in an error-free state and therefore successfully compiled, it is still possible that during the script execution an error occurs. This could happen as a result of a faulty code in the Commands code block or the Main code block.

If the vScript is being debugged, an error dialog is displayed, script execution gets paused and the Processing log will also contain the error details. However, when the script is not being debugged then no warning will be displayed and no other action will be taken, as the vScript execution engine just takes a safe approach and ignore these errors.

But the script should be able to react itself to runtime errors conditions somehow. The way a runtime errors is handled is really dependent on the actual situation. To detect errors, each script element has its own ExecError boolean variable which can be checked against errors at runtime and appropriate action can be taken.

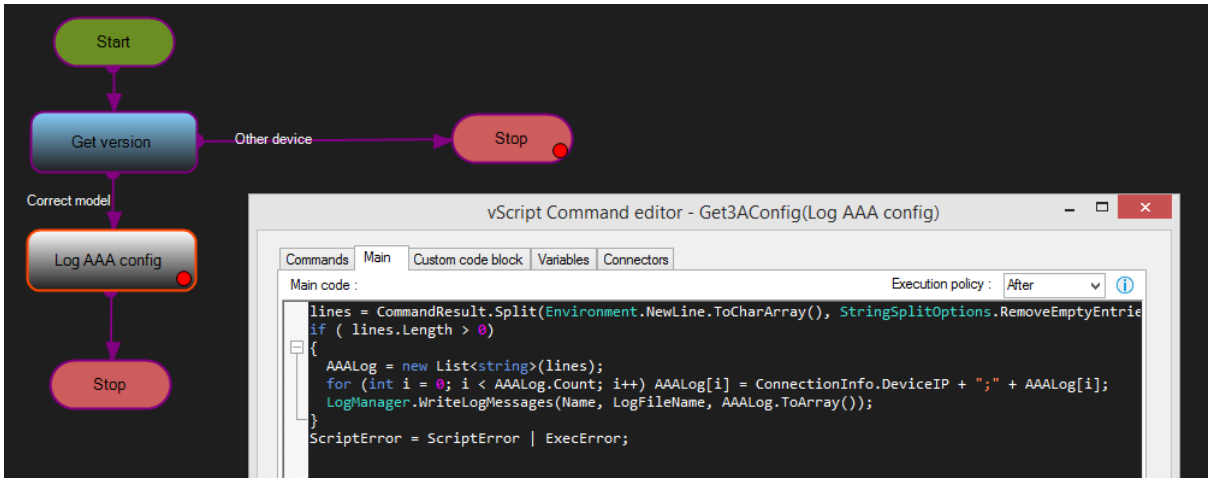
The ExecError variable is also set to true in the following two conditions:

- A timeout error occurs while the CLI commands executed. That is, the prompt is not returned by a device within the time span specified by the Command Timeout value in script settings.
- When the CommandResult of the executed CLI command contains any of the items listed in Command Failure Detection (see 15.2 General settings)

24.3.7 LOGGING INFORMATION FROM A VSCRIPT

PGT vScript execution engine provides a simple logging facility what you can use to log information from within a vScript to write required information to a log file.

For instance, consider the below vScript command element that extracts aaa configuration from a Cisco router and logs this information to a text file :

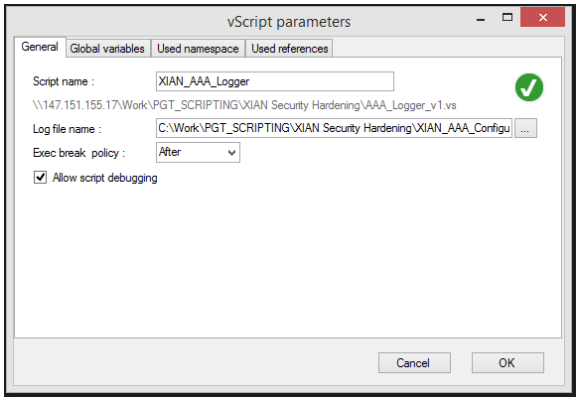


The command "sh run | s aaa" is sent to a router and the result is stored in `CommandResult` variable. As the result is a multi-line text, it is first split to create a string array. Then each element of the array is prepended with the IP address of the device and is written to a log file using the

```
LogManager.WriteLogMessages(string LogSource, string LogFileName, string[] Messages)
```

method of `LogManager`, which is a built-in static class in PGT. (*Static class means that it does not need to be instantiated*).

The main advantage of using `LogManager` is that it can handle a situation when multiple, parallely running scripts write to the same log file. You do not have even care with File creation, opening or closing, this is automatically managed by `LogManager`. All you need to do is call `WriteLogMessage` and reference the name of the log file.



But where does the name of the log file come from? You can simply specify a log file name in script parameters of a vScript, which is stored in the `LogfileName` script global variable what can be used in the code as shown above.

Each running vScript instance will log to the file specified. Keep in mind, that if multiple script write to the same log file, theirs' log messages will be mixed. In order to keep connected lines together, use LogMessages instead of LogMessage. The former method will ensure that the passed string array elements will be written to the file in one sequence and other vScripts logging into the same file will not inject messages among these lines.

When the vScript does not need the log file anymore, it can optionally signal that the file may be closed by calling the `LogManager.CloseFile(string LogFileName)` method. This will momentarily close the log file but should other, simultaneously running vScripts need to access the same file, it will be reopened. When all vScript using the same log file finished its operation, the file will be eventually closed.

Calling `LogManager.CloseFile()` is optional, and in case it is missing from the code, PGT will automatically close the log file after 30 seconds of inactivity.

24.4 ADVANCED TOPICS

Up until now, I tried to refrain from using programming "terminal technicus" to describe the operation of a vScript. And actually it is not necessary to be a skilled programmer to build simple Visual Scripts.

Avoiding programming was the main goal and a strong intention during the development and introduction of Visual Scripts. Mainly because Network Engineers – the main audience of PGT - tend to focus on their task and do not want to get involved with programming.

But under the hood, the Visual Script Editor is nothing else than a simplified c# compiler that builds the source code of the vScript at runtime and keeps compiling it to ensure it is in an error free and executable state.

If you want to develop more complex scripts, some programming exercise is inevitable, at a minimum you must understand how to deal with variables and understand basic string manipulation.

This chapter will slowly introduce the more advanced features of vScripts. At the end you will recognize that *virtually anything can be programmed into a vScript*.

24.4.1 THE BUILDING BLOCKS OF A VSCRIPT

The former simple script revealed some vScript elements. To summarize, the currently supported building blocks of a vScript are:

- Start element
- Connector
- Simple decision
- Simple command
- Command
- Stop element

Both the Start and the Stop elements are actually Command elements serving the special purpose of defining the script entry and exit points.

A vScript must have a single starting point where execution starts and may have many stop elements where the script terminates.

Except a connector, all elements of the script have three main purposes:

1. Construct one or more CLI commands to be sent to a device
2. Execute the constructed command(s)
3. Store the result

On the other hand, Connectors have defined conditions to determine the execution path. Outgoing connectors of a Command (also Start, Stop and Simple Decision) are evaluated one by one in the defined order until the first evaluates to TRUE. Execution will then proceed to the connected element. *The rest of the connectors will NOT be evaluated afterwards. If a connector has no condition it will always evaluate to TRUE.*

24.4.2 COMMAND VS SIMPLE COMMAND ELEMENT

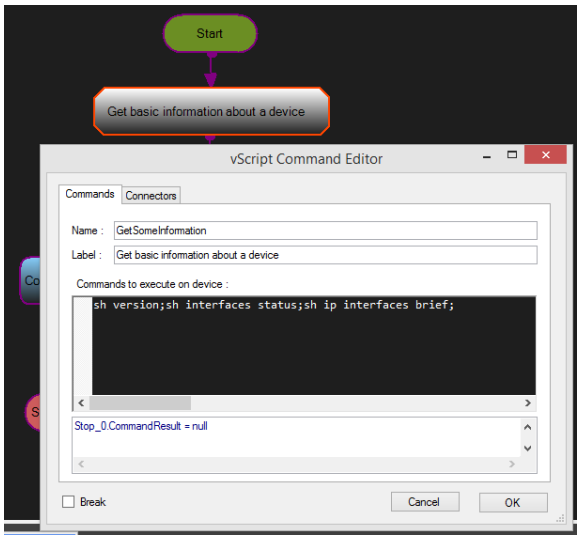
One advantage of a vScript is that someone with limited or no programming skills can still build a sophisticated, decision based script. The vScript editor tries to hide the more complex functionality unless it is necessary to show the details. This is the drive behind the introduction of Simple Command (and also Simple Decision discussed later)

The most versatile element in a script is the Command element. To simply put, a Simple Command is the restricted view of the underlying Command element. So why is the restriction? Well, a Command element might get overwhelmingly difficult and when one has a simple purpose they may not want getting involved with the deeper details of a Command element.

24.4.3 THE SIMPLE COMMAND ELEMENT

The Simple Command represents static CLI commands. Use this element when you need to send out CLI commands independently of the script current state. The below example shows a case where one wants to collect some information about a device by sending three commands :

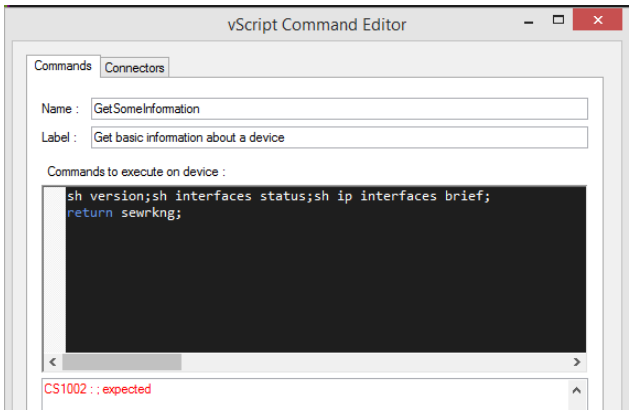
- sh version;
- sh interfaces status;
- sh ip interfaces brief;



The vScript execution engine will send out these commands and the returned answer will be stored in a variable named CommandResult. It is important to understand, that each element which is capable of executing commands (Start, Stop, Simple Decision, Simple Command) have its **own** CommandResult variable. Throughout the script, at any point, this variable can be checked against its value. To reference an element's CommandResult, the element's name must be prepended to it. For instance, considering the above example, the full name of the element's command result would be GetSomeInformation.CommandResult. *Do not confuse the name of the element with its display label* ! While the name must be unique in a script, the display label can hold any string to be displayed on the view.

Of course, if multiple commands were executed by an element, its CommandResult will contain the whole, concatenated answer consisting of multiple lines. Processing such an answer may require some tricks. Later chapters will show some possibilities.

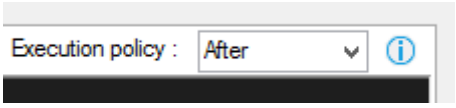
The important part is, that literally any text can be entered to a simple command's command section. There is an exception, however. The word return is a c# reserved word and if used, PGT assumes that you are writing some c# code to construct a command. Later chapters will explain it in more details, but using the return statement in a Simple Command should be avoided. If used incorrectly, the editor will display a compile error message with red below the entered text:



24.4.4 THE COMMAND ELEMENT

A command element is the most versatile part in a vScript as it can execute any code to achieve its functionality.

In its simplest operation it can build CLI commands to be sent to a device and then make checks on the returned answer by executing its Main code. In this mode the Main code should be executed "*After*" the CLI commands were sent and the answer received. For this mode set the Code execution policy to "*After*".



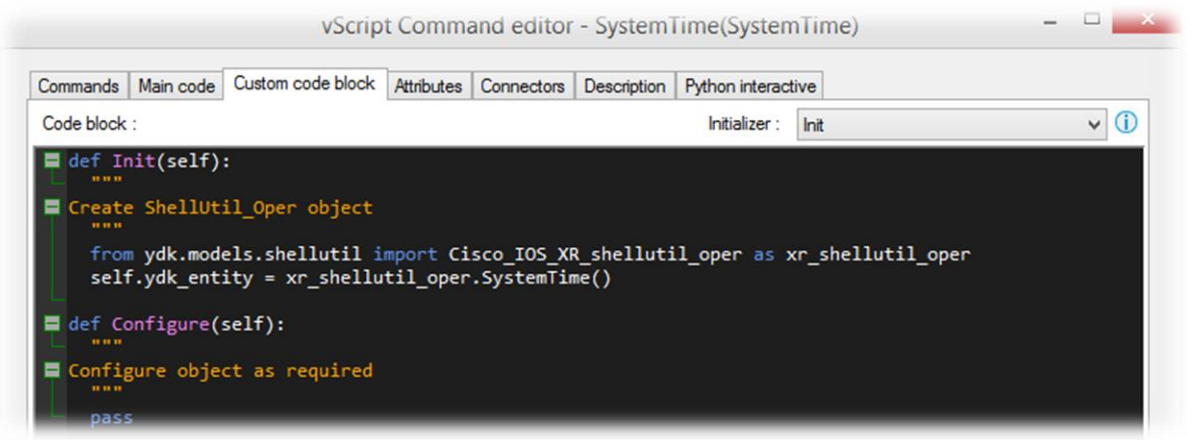
On the other hand, it is also an option to run the Main code first to make calculations to generate the appropriate CLI command. This can be achieved by setting local variables and using these variables in the Command code part to construct the CLI command to be sent out. For this operation, the Main code execution policy should be set to "*Before*".

When the Advanced options view is selected in general properties, a Custom code block can also be defined for the command element. This code block is normally a *method of the class embodied by the Command element.* This method will not be automatically invoked, by the vScript execution engine but either the Main code or the Commands code can refer to it.

When constructing a new vScript and testing in a production environment, it can be a wise decision to NOT send out some dynamically constructed CLI commands but rather just test what was built. For this reason, the Command element can be run in ***Demo mode***.

In this mode, the CLI commands constructed by the element **will not be sent to the device**, but the processing log view of the vScript will show what was assembled. An element set to Demo mode will display a red "No" icon and will have a modified colour.

Some Command elements may require initialization before they can be used. The custom code block part ensures this functionality by allowing the selection of custom a method to be designated as the Initializer code like it is illustrated below:



The designated initializer method will be called from the constructor of the object:

```
class TSystemTime(object):
    def __init__(self):
        self.ID = uuid.UUID("dc307008-94d4-4134-a6bb-f772b7ff3bfe")
        self.CommandResult = ""
        self.aCommandResult = []
        self.ExecError = False
        self.ydk_entity = None
        self.Init()
```

24.4.5 BREAKPOINT OPERATION

When a breakpoint is set on an element, the code execution will be paused and it must be manually resumed or terminated.

There are two factors influencing where the script code is paused on a command element:

1. The break execution policy : Before or After– set in script parameters
2. The command execution policy : Before or After – set individually in a command element

To discuss where the code is paused some clarification is needed on the exact operation of a Command element. This element has three main part:

1. The Main code block
2. The CLI Command construction block
3. Execution of the CLI commands

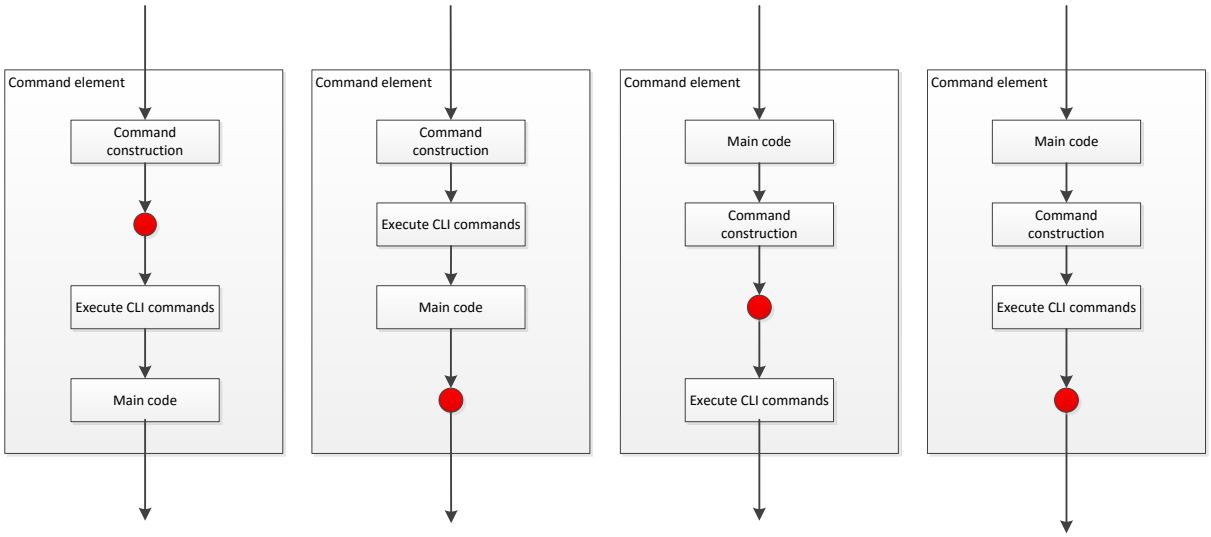
The Command execution policy represents two distinct purpose of the Command element:

1. When you want to dynamically construct the CLI commands that is sent to the device you probably want to run the Main code block before command construction. This is achieved by setting the execution policy to “Before”

2. The other option is to have statically configured CLI commands, and you want to use the Main code block to process the response. In this case, set the execution policy to “After”

When break policy is set to “Before” PGT assumes you probably want to check the CLI commands before they sent out. If you set the break policy to “After”, execution engine assumes you want to check the final result of the whole Command element’s operation.

Depending on the combination of the Break execution policy and Command execution policy the breakpoint is set in the execution path as illustrated below:



Execution policy : After
Break policy : Before

Execution policy : After
Break policy : After

Execution policy : Before
Break policy : Before

Execution policy : Before
Break policy : After

24.4.6 ABOUT SCRIPT VARIABLES

To explain the need and usage of script variables, let's consider the Command Element.

The Command element is the underlying element type of Start, Stop and Simple Command. As explained, the Simple Command is designed to execute statically entered CLI commands.

But a Command element was designed to *dynamically construct* CLI commands based on other script variables instead of using statically entered commands. To do so, one needs to understand the concept of variables in a script.

For those being familiar with the c# language there is nothing to really explain as a vScript is using c# variables, types and syntax.

First of all, variables have types. This means that a variable is only capable of storing information conforming to its type requirement. For instance, text and numbers are different types and a variable designed to hold a number cannot store text.

To use a variable, it must be declared. A variable can be declared in a vScript at two different locations:

- in an element
- globally for the script in script parameters

Most of the time you would be working with text and numbers, so you can use the following base types:

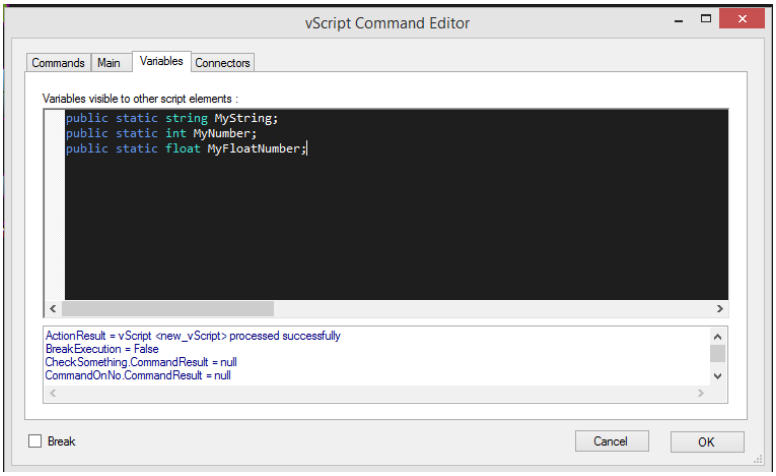
- For text, use type : string, or an array of strings string[]
- For whole numbers use type : int
- For decimal numbers use : float
- For logical use : bool, its value can be either True or False

To declare a variable, define its type and name using the syntax: string MyString; *Do not forget the semicolon at the end, it is obligatory.*

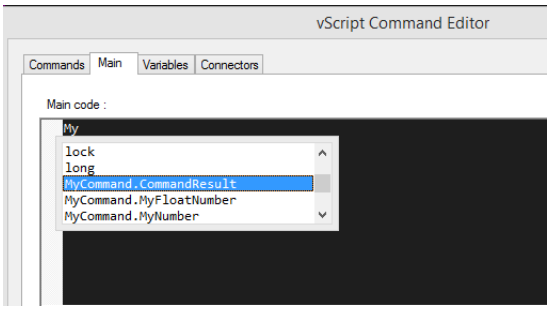
There is one more thing needs mentioning, the variable visibility. Not going into details for now, all variables should be declared in a vScript using the public static visibility modifiers:

public static string MyString;

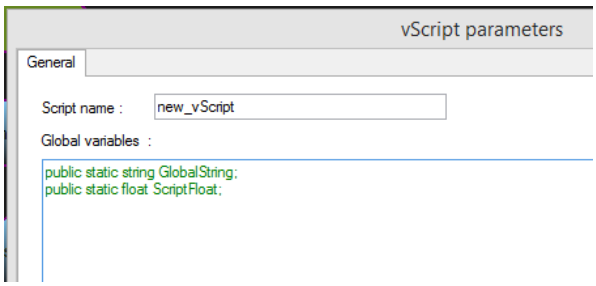
So here it is how to define variables in a Command element:



Assuming that the name of the element was MyCommand, you can reference these variables at any other element as MyCommand.MyNumber. As you start typing in a code editor box somewhere in the script, you will notice that the built-in code completion engine will discover these variables, so you do not even have to remember the correct naming, just select from the drop-down where you need it :



As already mentioned, you can also declare variables globally at script level. You will find the place for it in the Parameters window of the script:

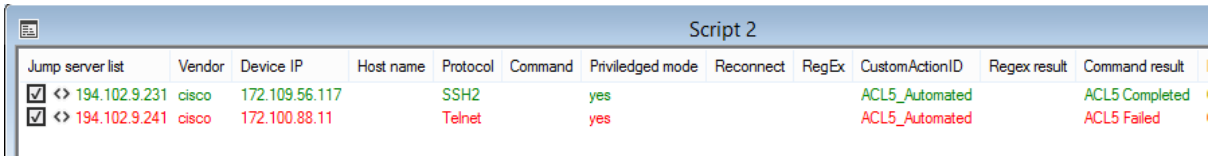


24.4.7 BUILT-IN VARIABLES

There are some built-in variables in the script required for the normal operation of a Custom Action Handler. These, with its default values are:

```
public static bool BreakExecution = false;
public static string ActionResult;
public static bool ConnectionDropped = false;
public static bool ScriptSuccess = true;
public static IScriptExecutorBase Executor;
public static IScriptableSession Session;
public static DeviceConnectionInfo ConnectionInfo;
```

The value of ActionResult will be displayed in the script output in the Scripting Form as the Command Result. The colour of the corresponding line depends on the ScriptSuccess : will be green if true, red if false. Example script output is :



In the first case the Custom Action Handler, named ACL5_Automated reported ScriptSuccess as true and set ActionResult to “ACL5 Completed”. For the second one, the operation failed.

If BreakExecution was set to true, the scripting engine would ask the user whether to stop the script execution. ConnectionDropped indicates the device connection could be lost as a result of some operation (reload for instance).

For a more detailed explanation of Executor, Session and ConnectionInfo, please consult the PGT Developers Guide documentation. Normally you won’t need to use them in a vScript but a more complex script might require their usage.

24.4.8 USING VARIABLES TO BUILD CLI COMMANDS

A sophisticated vScript would check the configuration of a device, and based upon the current configuration would update it as required. This functionality requires that CLI commands sent to the device is built at runtime and are not programmed statically into the vScript.

For example consider the following simple task: you need to add a static route to a 10.1.1.1/32. It sounds very easy, but when it came to realization you would recognize that the next-hop address of the static route to be added could be different on each devices and needs to be calculated individually on all routers.

There could be several ways to find out the gateway address, but for the time being assume we do so by checking an existing – well known route – to get the next-hop, like using the command:

```
"sh run | i ip route 192.168.10.0 255.255.255.0"
```

The answer received to this command will contain the next-hop address we need to use.

```
R1#sh run | i ip route 192.168.10.0
ip route 192.168.10.0 255.255.255.0 192.168.1.2
```

The question is, how to get it from the response string.

At this point some c# knowledge comes handy, or one need to search for help online. Fortunately, this is a very easy stuff. For example we can do it by splitting the response text to words and take the 4th word. This operation looks like the following in c# :

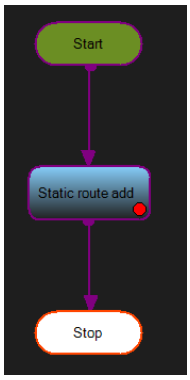
```
string[] fields = CommandResult.Split(" ".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
string next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
```

The variable *fields* of type string array will now store the split CommandResult. The StringSplitOptions.RemoveEmptyEntries is necessary because there could be multiple spaces between the words and as a result we might get empty array elements in the result. This would be bad, as in this case the 4th element might be something else.

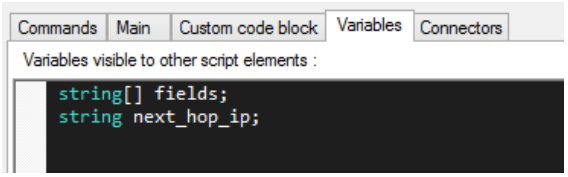
It is also a thing, that the response could contain CR LF charters which must be removed from our final next-hop address.

Let's see this in operation.

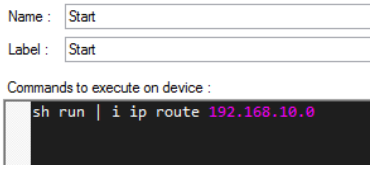
Build a very simple script:



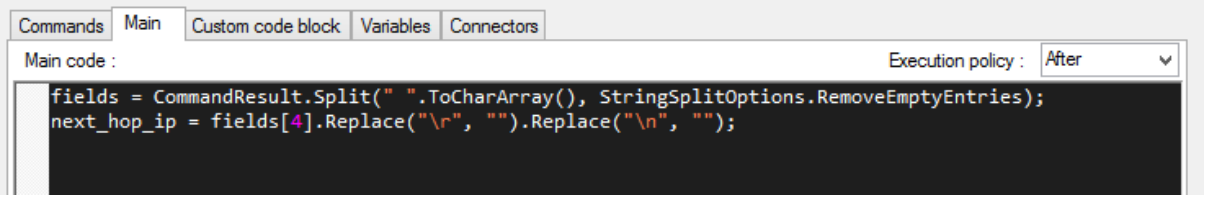
We will need two variables, declare them in the Command element



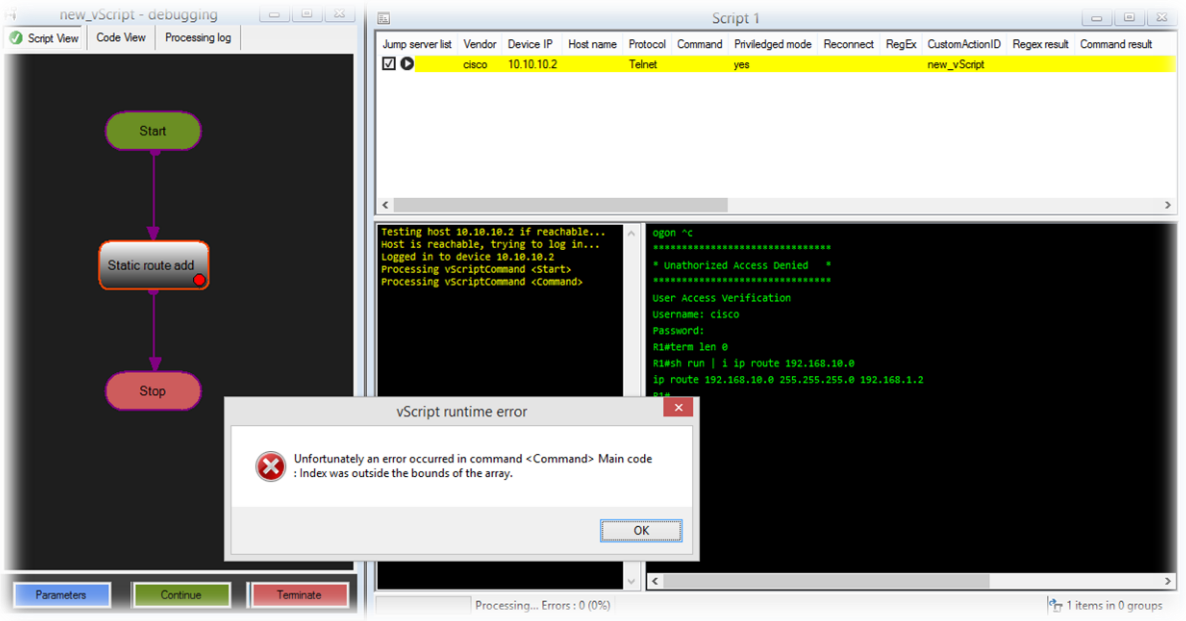
Why not check the existing routing in Start:



Then let's get the next hop address as discussed, write our code to the Main code block. Why there? Because the purpose of this code block is to perform any calculation before we send the command to a device. This is therefore the place to check the next hop address:



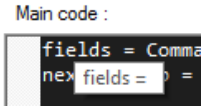
Let's see what we get, run the script.



Well, it's not really what we wanted to achieve. What could go wrong? The error says the index was outside the bounds of the array. What array? Well, we have just one array declared, the *fields*.

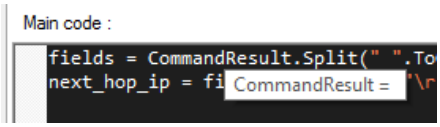
Open the Command Editor and check its value.

Well, this is really empty.

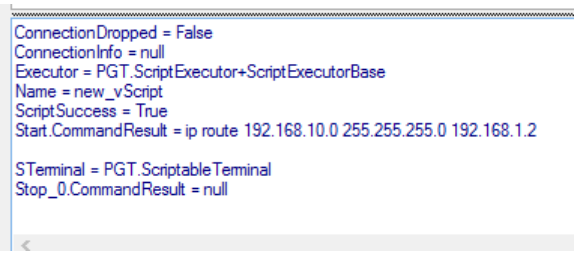


Now check the CommandResult.

It is also empty. But why ?



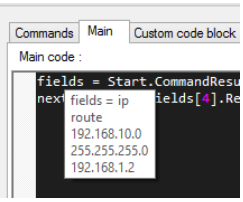
Because the CommandResult refers to the current element's CommandResult but we need the Start element's CommandResult ! Check its current value by opening the editor again, focus on the text box at the bottom:



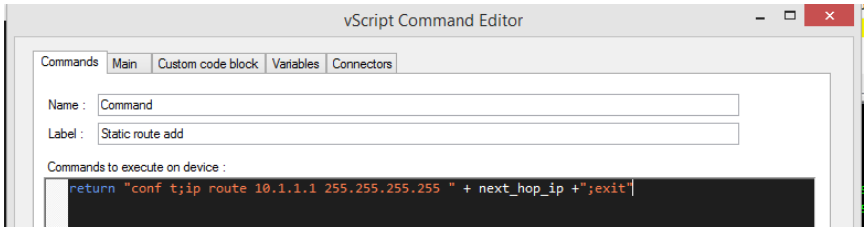
Here it is ! This is what we need.

So correct the code accordingly, instead of CommandResult use Start.CommandResult.

When we run the script again and check our variables' value, it seems fine, we are in a better shape:



Now we can construct the route command from it in the Command element Command code block:



We need to return 3 commands:

- Go to config mode
- Add the route
- And exit at the end

Multiple commands will always have to be separated by semicolon.

Unfortunately, as we run the vScript again, we notice that something is still wrong. Although the next_hop_ip now contains the correct ip address, the constructed and issued CLI command is incomplete, the next-hop is missing from it:

```
R1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#ip route 10.1.1.1 255.255.255.255
% Incomplete command.
R1(config)#exit
R1#
```

The problem is the execution sequence of code blocks inside the Command element. When you check the code editor of the Command element, switch to the Main code tab you can see the Code execution policy was set to “After”. This means, that the Main code block is only executed after sending the configured CLI command to the device. This is not good for us now, as we first need to run the Main code to construct the CLI command. Let’s change it to “Before” and run the script again.

You can relax for now, it works as expected:

```
R1#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
R1(config)#ip route 10.1.1.1 255.255.255.255 192.168.1.2
R1(config)#exit
R1#
```

To wrap up the lessons learnt, there are two important point:

1. Watch for using the correct CommandResult variable, do not forget the class qualifier (the element name) if checking another element’s variables
2. Take care of the Main code execution policy : Before or After sending the CLI commands

As a best practice, when working with arrays, you’d better to check the array’s length before referencing its element. For example, we could change the code to:

```
if (fields.Length >=5) next_hop_ip = fields[4].Replace("\r", "").Replace("\n", "");
else next_hop_ip = "";
```

With this change we can avoid runtime errors, but this might not always be a good thing. It could be better to get notified if something works differently as we expected.

24.4.9 THE CONNECTOR ELEMENT

A Connector is essentially a simple stuff. To decide whether to continue the vScript to a next element, the connecting object, the Connector needs to be evaluated. Evaluation is logical operation, the result can be True or False.

For this reason, a Connector's Condition code block:

1. Could be a c# function that must return a bool value
2. or be a logical expression that evaluates to True or False

PGT will check the entered text, and if it contains the return keyword, it assumes this is a code block. Otherwise PGT will take the entered text as a logical expression.

There is not much guidance to be given to this by this manual, except that the *order of Connectors is important.*

A Command element can have multiple connectors and these are checked in sequence until the first one evaluates to true. Therefore care must be taken when conditions are defined and connector order is chosen. By opening the Command element editor, the order of connectors can be changed.

The destination of an existing connector can be changed by selecting the connector, pressing Alt while clicking on the connector and then moving the connector's end to the required target.

24.4.10 MORE ABOUT VARIABLES

Script variables can be declared at several locations which influences their visibility. Above, the guide suggested to declare variables with public static qualifiers. In fact, this is not a requirement just a precaution to not confuse beginners with variable visibility.

If you understand the standard c# variable visibility modifiers, use them as you need it to make sensitive variables invisible to other elements.

To better understand the vScript object model read the chapter Advanced Options and Code View

24.4.11 ADVANCED OPTIONS AND CODE VIEW

This chapter explains the internals of vScript in a nutshell and is intended only to experienced .NET developers.

All of the elements defined in the Visual Script Editor translates to a runtime counterpart. All runtime elements are nested classes inside a class named ScriptProxy. The ScriptProxy is the class responsible for interaction between the runtime compiled code and PGT script execution engine. The visual elements are able to emit theirs own c# code representing the runtime object. PGT will interact with them via ScriptProxy object.

To enable the advanced options, go to the Visual Script Editor Parameters, and check the "I am an experienced developer…" checkbox. This will enable the following views and options :

1. Code view: the complete runtime code of the vScript.
2. Custom Assembly references and Namespaces in the vScript Parameters editor
3. Custom Code block in the Command elements' editor.

Command elements can have a custom code block. This is a code block that is inserted into the runtime class as is. This Custom Code block makes possible for each runtime script element to have user defined methods, which in turn can be called from the Main code of the element. There is no restriction what can be entered into the Custom Code block, as long as the text can be interpreted as part of a class, mainly like class methods.

For example, the below is a completely valid code to be inserted into any Command element's Custom Code block:

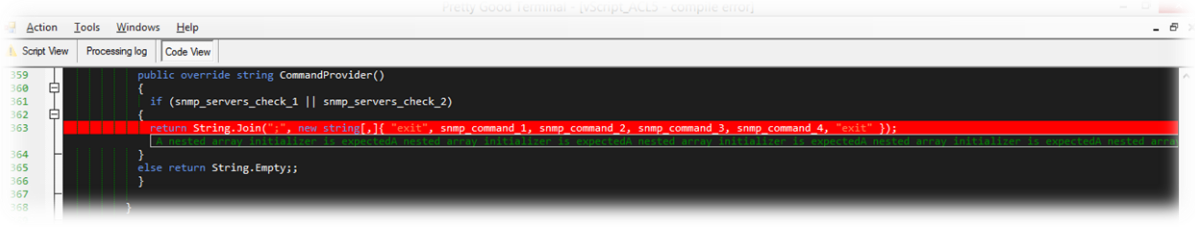
```
public string Dummy()
{
  return "Hello World";
}

public bool AreWelcome()
{
  string s = Dummy();
  return s == "Hello World";
}
```

Then, from the element's Main code block, both of them could be referenced.

Besides of the above, custom namespaces and assembly references can be added to a vScript. It is pointless to explain that this ultimately unleashes the code of a vScript.

The Code View can be very useful for troubleshooting code errors, as it shows the error lines in the code annotated with the respective error.



24.4.12 CUSTOMACTIONHANDLERS PRIORITY

It must be clear by now, that vScripts and external CustomActionHandler libraries registered via Tools/Options dialog box stem from the same root. Both of them are identified by a string, called the CustomAtionID. This ID should be unique for PGT to be able to call the intended handler: a vScript or the external dll. When you try to open a vScript twice in two editor windows, you will notice that PGT will warn you about conflicting CustomActionID names. This is by design.

There are some situations, however, when PGT allows two CustomActionHandler being registered with the same name.

In fact, PGT stamps CustomActionHandlers with priority and the restriction is that no two CustomActionHandlers can be responsible for the same CustomActionID at the same priority level.

This priority mechanism is implemented to allow debugging of a vScript which is also registered via Tools/Options. Without the priority ordering a naming conflict would occur: the CustomActionID (the vScript name) registered in Tools/Options would conflict with the one opened in the editor, and first the vScript would need to be removed from the list of the registered CustomActionHandlers. This would be inconvenient.

In the end, priority is assigned to CustomActionHandlers in the following way:

- The vScript opened in the editor will have priority 0
- The vScript registered in Tools/Options will have priority 1
- The external assembly registered in Tools/Options will have priority 2

And the rule is: always the lower the priority handler would be called when referenced.

This means that you could have three handlers registered, all responsible for the same CustomActionID. If you registered a vScript named “Demo1”, you can run scripts referencing the CustomActionID “Demo1”. As soon as you open the “Demo1” vScript in the Visual Script Editor and re-run the script, the opened vScript in editor will switch to debug mode and will be called.

A side effect of this mechanism is that vScripts will always have priority over external CustomActionHandler dlls if they share the same CustomActionID.

24.4.13 VSCRIPT OBJECTS LIFECYCLE

When you work with vScripts and run multiple script windows in parallel using the same vScript, it worth understanding how vScript objects created and ceased.

PGT vScript execution engine can operate in two different modes, depending on the "Allow vScript caching" setting in Tools/Options/Visual Script Settings tab.

- If caching is allowed – this is the default mode

Each Scripting Form referring to a vScript will initialize a new, private instance of the vScript by loading the .vs file. As long as the Scripting Form runs the PGT script, the same vScript object will be used for each device in the PGT script. *This model has an implication that must be considered carefully*: *class variables will retain their values between multiple calls to the vScript as PGT connects to devices one after the other. As a result* ***take care of consistently setting the value of a class variable and do not rely on default values.***

Only when the script is finished or terminated will the vScript objects be finalized and released and upon re-starting the PGT script a new vScript object will be created from the vScript file.

If a given vScript is used in a Scripting Form, and the same vScript is being edited in a Visual Script Editor window (in design mode) in the same instance of PGT, changes made to the vScript will be reflected in the running instances when the vScript is saved. In this case, when the Scripting Form needs to process the vScript next time, a new instance will be created from the .vs file instead of using the existing vScript instance. This behaviour only occurs when the vScript file was modified by the same PGT process. If the vScript file is modified outside of PGT, the changes made will only be reflected the next time the file is loaded.

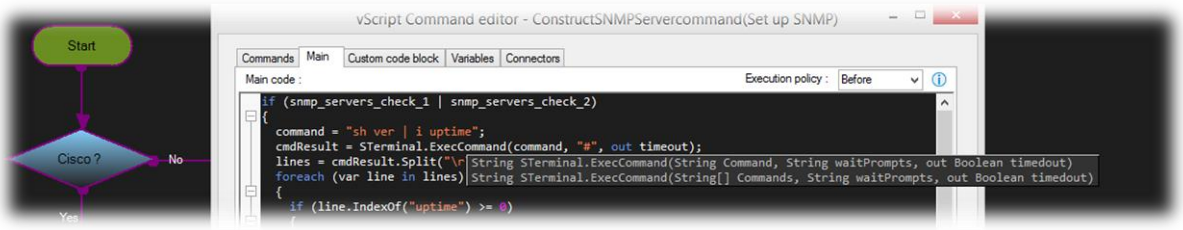
- If caching is disabled

In this mode, a new vScript object will always be created by loading the .vs file whenever a script needs it. As a result, any script variable value or dynamically created objects will not be retained between two runs of the vScript.

This mode also implies, that changes made and saved to a vScript file will be reflected immediately.

24.4.14 VSCRIPT LIMITATIONS

- The current version of Visual Script Editor cannot inspect variables declared inside a code block, such as Main code or Command code block, however, code completion works for these variables, too. For this reason, it may worth declaring variables in the Variables block as a class variable.
- The Visual Script Editor is not a real debugger. When it is switched to debug mode, it can pause script execution at vScript element block level and then it is possible to examine the values of variables. But the c# code inside the element cannot be executed line-by-line.
- When the vScript code contains errors and the code cannot be compiled – and has never been compiled since the editor was opened – code completion and variable value inspection operations will be degraded or entirely switched off, depending on the nature of error in the script code. As soon as the code can be compiled, both code completion and variable inspection will come back to live.
- It is also an important limitation of the current version, that the vScript execution engine can only check standard device prompts when sending CLI commands to devices. Standard prompts mean the prompts listed in the Vendor definition in Tools/Options. In case a special prompt needs to be checked – like described in chapter *Creating dialogues in scripts* – a workaround is to call Session.ExecCommand() directly from the script code, like the below example shows:



24.5 TEMPLATES AND REPOSITORIES

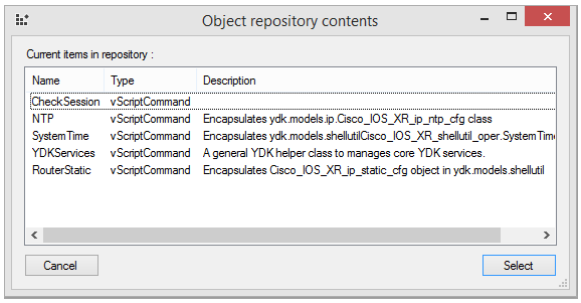
Both vScript templates and vScript repositories are tools for reusing existing code. Any vScript can be saved as a template and later new vScripts can be cloned from the selected template.

PGT searches for templates (*.vst files) in the templates folder as specified in Tools/Options/Visual Scripts and presents them on the Action/Visual Script Editor/Templates submenu ready to clone a new vScript from them.

The only limitation for templates that they cannot be executed or debugged but can be edited the same way as a normal vScript. When a template is edited, other vScripts cloned from that template *will not inherit the changes*.

A repository is also a special vScript but its purpose is to store individual vScript elements to be reused later. As a repository stores independent elements, it cannot be compiled or debugged nor can it be executed.

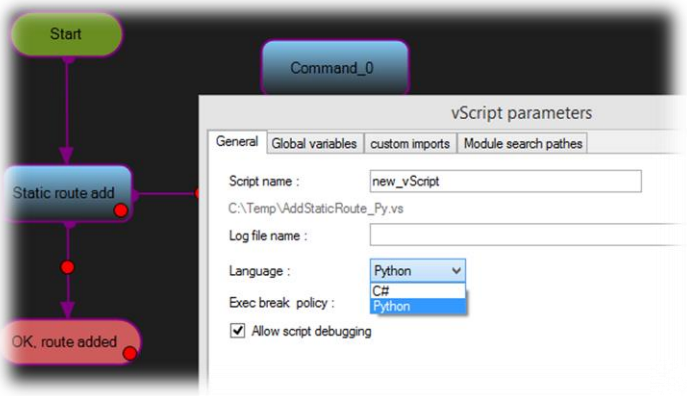
The standard repository shipped with PGT contains elements for YDK-Py samples:



The supported way of changing items in repository is to overwrite existing elements. If you open the repository with editor you can delete items but you should not change the code of an element this way.

24.6 USING PYTHON FOR VSCRIPTS

On the vScript parameters tab one can chose what language to use for the vScript, C# or Python:



Although PGT allows changing the script language for an existing vScript, it will NOT translate any existing code and as a result most probably a compilation error will occur.

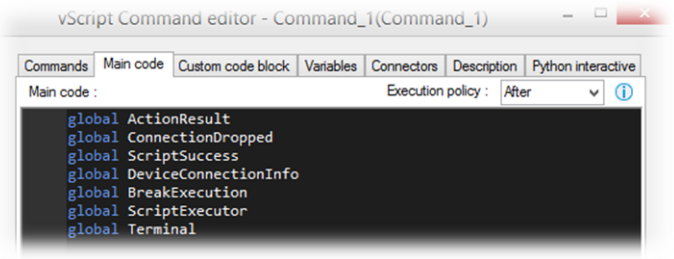
For the most part there is no difference whether C# or Python is used as a vScript language, however, there are some important points to note when Python is chosen.

C# and Python are inherently different in many aspects and their purpose. Python is rather a script language while C# is more about application development.

What is important regarding vScripts now is Python variable handling and scoping. The object model introduced for vScripts using C# can remain almost the same with the exception of not using the enclosing class (ScriptProxy) for Python classes representing the vScript elements.

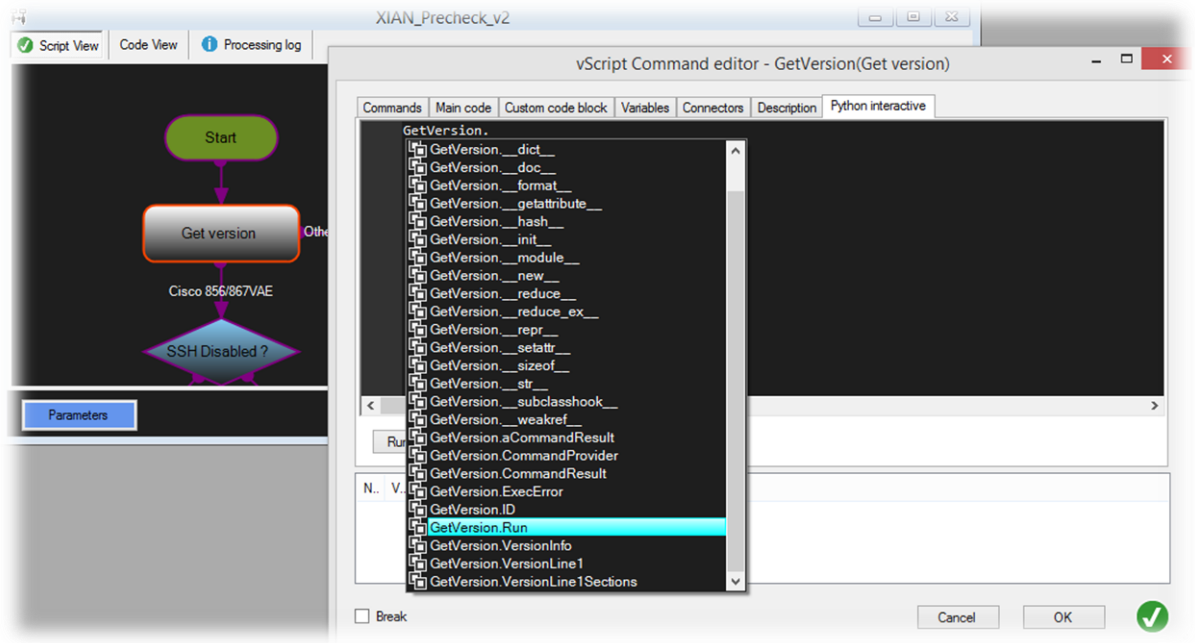
As a result, the variables used to communicate script result gets *global variables* in the Python module constructed by PGT at runtime. These variables are :

- ActionResult
- ConnectionDropped
- ScriptSuccess
- DeviceConnectionInfo
- BreakExecution
- ScriptExecutor
- Terminal
- LogManager



For convenience, when a new Command Element is created for the vScript, the Main code will include them.

Undoubtedly, an advantage of Python is its interpreted nature. That is, the interpreter can evaluate a code instantly. To exploit this feature, the Command editor has an extra tab with a Python interactive console to help Python code script code development. In this console, all the vScript elements can be referenced as objects:



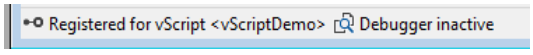
Additionally, when a vScript is written in Python and is successfully compiled, the Python Interactive console can be displayed from the Visual Script Editor.

The Python Interactive console is also the debugger for the vScript code. Please note, that the vScript Python code cannot be modified from the Python Interactive console, only through changing the code in the relevant command element.

24.7 DEBUGGING PYTHON VSCRIPT CODE

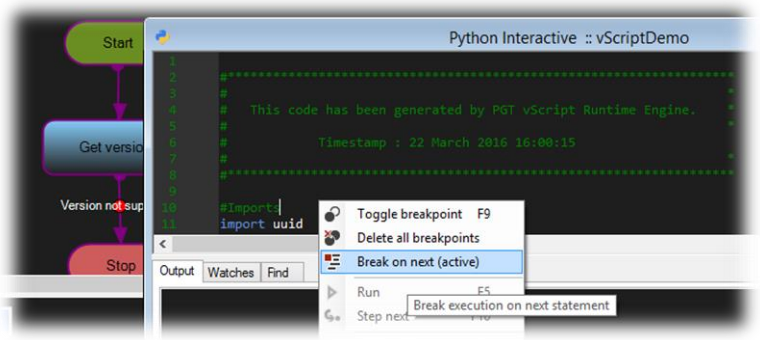
A big advantage of using Python as the script language is that it can be debugged from PGT directly.

When a Python vScript is in error-free state, the “Debug script” button is activated below. By clicking the button, the Python Interactive window opens and loads the vScript code. At the same time, the status bar will show that the script is registered as a vScript handler and the debugger is currently inactive :



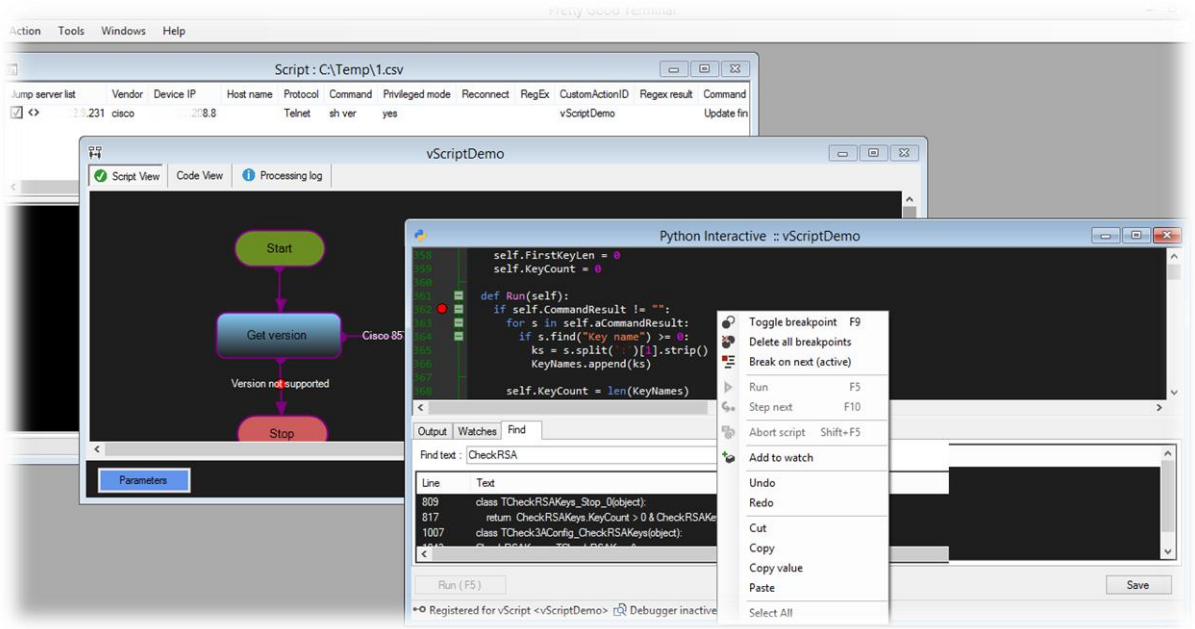
The loaded code, however, cannot be directly started or stopped, it must be called by the vScript engine. Also, the code is read-only as it cannot be directly modified from within the Python Interactive window but any modifications must be done via the vScript editor.

When the Python Interactive is opened, it is set by default to break at the first possible occasion as the context menu displays. This can be turned off by clicking the menu item and specific breakpoints can be set on interesting lines.



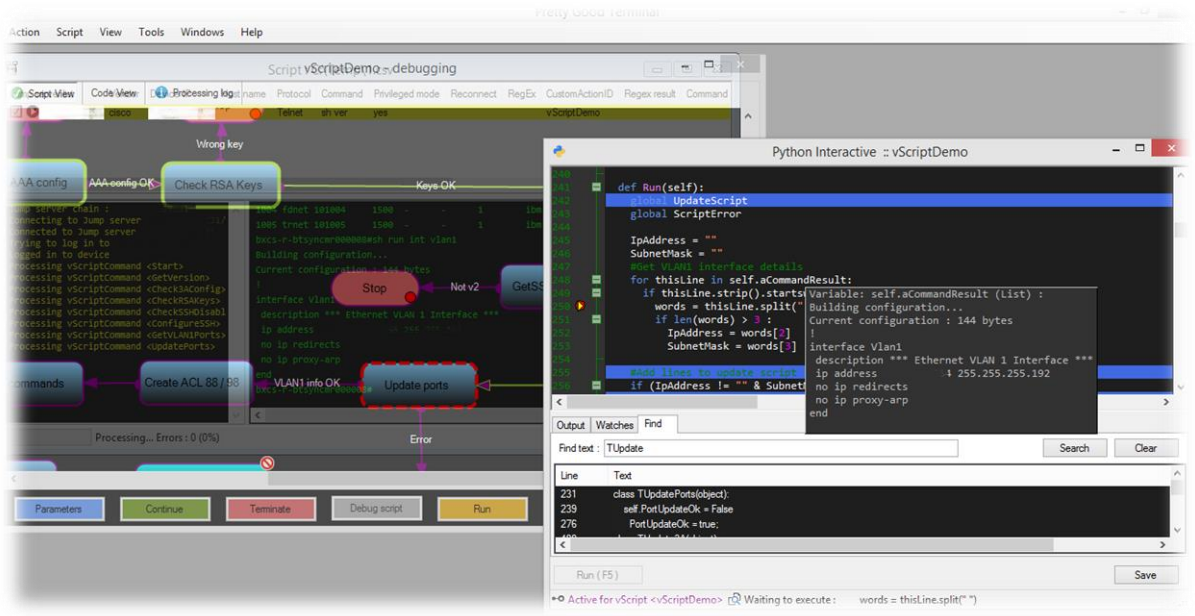
For a full debug session, 3 windows should be opened in total :

- A PGT script containing the device list
- The Visual Script Editor with the specific vScript loaded
- The Python Interactive opened *from* the Visual Script Editor

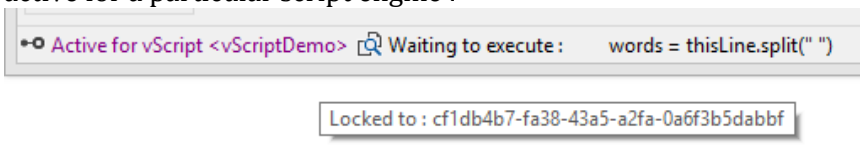


To start debugging, go to the PGT script and press F5 or select “Run” from the Script menu.

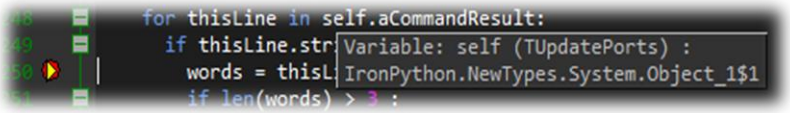
Script execution starts, and once the code reaches the breakpoint the debugger window will automatically brought to foreground. It is also an option to make the vScript Editor transparent and this way it can be viewed as an overlay through which the terminal window and CLI commands can easily observed :



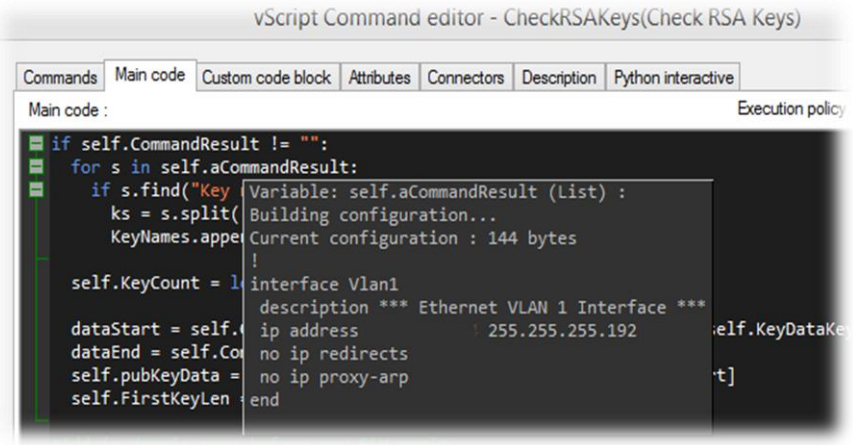
When the debugger attached to the code, the status line will change to purple and shows it is active for a particular script engine :



Hovering the mouse cursor over a variable will show its current value. It is worth remembering that the *self.* pointer will represent the current object where the debugger is paused. It is important as it can be confusing in certain situations. For instance, consider the above example, move the cursor over *self* :

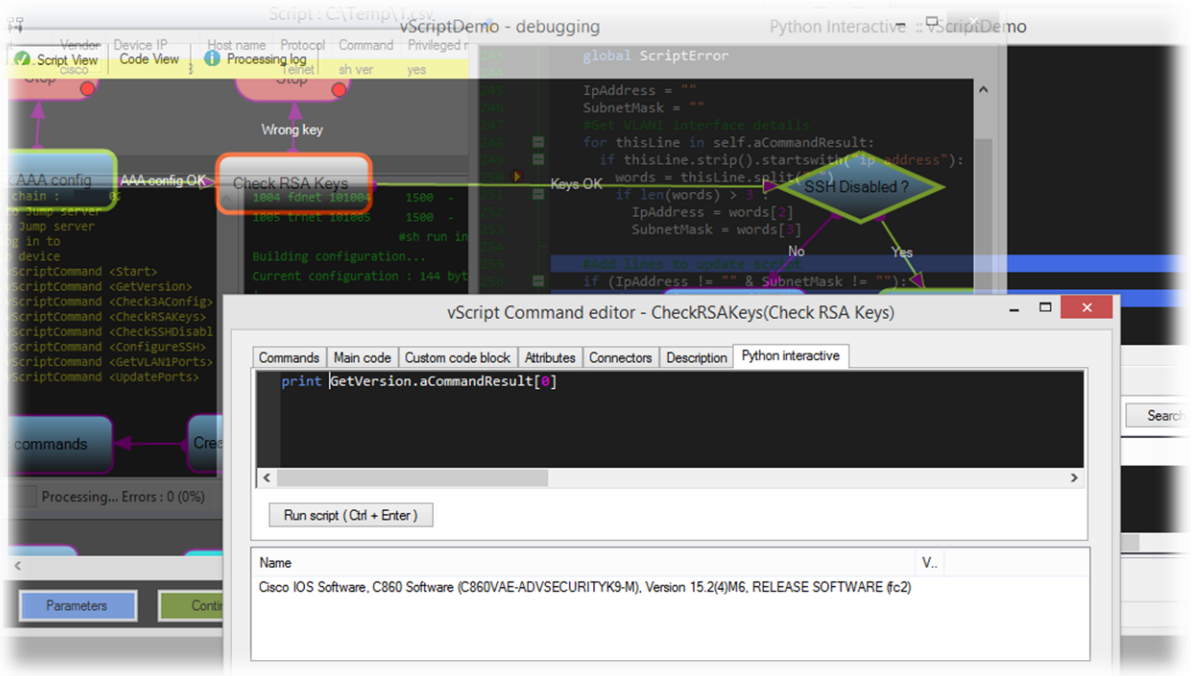


It shows that it is referring to a specific instance of the TUpdatePorts class, and as a result, *self.aCommandResult* is the variable of this object. Now, if one went back to the vScript editor and opened another element to the code inside, would find the following :



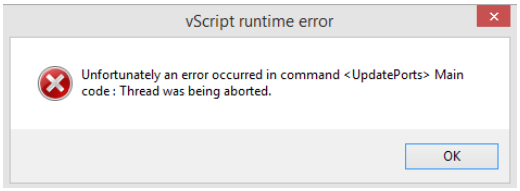
This is because the self pointer is not pointing to the CheckRSAKeys object but UpdatePorts, hence the *self.aCommandResult* will represent UpdatePorts.aCommandResult as long as the debugger is paused in UpdatesPorts object.

To check the value of other objects, theirs full name should be used. The command elements include a Python Interactive tab, which can be used to evaluate any expression regardless of the debugger state :



If the Python Interactive window is closed while the debugger while it was waiting for user action, script will be automatically continued.

To stop a vScript debug session gracefully, go back to the PGT script window, and press Shift-F5 or click on "Stop this" menu item in script menu. An error dialog showing that the script was aborted will be presented, but this is normal.



24.7.1 NAVIGATE IN CODE

You can easily navigate in vScript Python code to the class related to a specific vScript element by clicking on the element graphic. It will move the cursor in code editor to the class representing the element.

Also, in Python Interactive the navigator view helps discovering the required element by name.

Another possibility for navigation is searching for a text. Press Ctrl-F in the code editor anywhere, it will take you to the search window. The entered text is then searched throughout the whole code and any partial match is highlighted. When you click on a match, it will move the cursor in code window to that specific line.

25 CISCO YDK-PY SUPPORT

One of the main drives behind the development of PGT v7.0 was the intention to provide support for Cisco YDK-Py API. The reason is that the object oriented approach of YDK API as it represents network entities fits very well to the framework of PGT, especially in the case of vScripts.

The YDK API as published by Cisco unfortunately cannot be directly supported by PGT due to some limitations of IronPython v2.7.5. These limitations are:

- Python relative module import syntax used in YDK-Py is not supported
- The LibXml, Paramiko, PyCrypto and NCClient modules are not supported by IronPython

To overcome of these issues PGT exposes its own, internal NetConf client to the Python runtime and builds a wrapper class around it to provide support for YDK by introducing the PGTNetconfServiceProvider Python class. As a result, PGT YDK solution is independent from all of Paramiko, PyCrypto and NCClient modules and is transparently integrated to PGT connection services. At the same time, the usage of lXml module was replaced by the standard Python Xml module.

The result is a mixture of .NET and Python environments. While core connection services are provided by PGT in form of SSH and NetConf clients, some Python classes inherited from the original Cisco YDK API were slightly modified to interface with PGT connection services. The modified YDK-Py API is part of the PGT distribution. PGT v7.0.0.0 supports Cisco YDK-Py version 0.4.2. Later versions of Cisco YDK-Py may be supported by rewriting the newly published API the same way.

PGT contains the modified YDK API under the .\Scripts\YDK\ydk-0.4.2-py2.7.pgt and .\Scripts\YDK\providers folders. These folders are initially appended to the Python search paths settings by PGT when going to Tools/Options/Python Engine settings. The modification made to the API is subtle on one hand, but also quite complex when it comes to replacing NetconfServiceProvider class and lxml functionality:

- Updated all *.Modulename* import statements by removing the leading dot character. PGT (IronPython) can't understand relative references this way at the moment.
- NetconfServiceProvider class in original Cisco implementation builds on NCClient which uses Paramiko which references PyCrypto in turn, and so on and so forth. Unfortunately these libraries are not supported by IronPython and as a result a replacement class had to be developed. This class is called PGTNetconfServiceProvider, it is written in Python and combines the Netconf connection services provided by PGT and the functionality in original NetconfSergviceProvider in order to provide the same services for YDK API.
- All lxml imports were removed and its functionality is replaced by the standard Python xml library.

Cisco YDK-Py requires still a couple of Python packages to be installed on the system. These packages are the following:

- **Install Pip**

  Installing pip is done with a backport of the ensurepip module:

```
C:\Program Files (x86)\IronPython27\ipy -X:Frames -m ensurepip
```

- **Install Enum package**

  Download Enum34 from https://pypi.python.org/pypi/enum34 . Unzip it, go to the directory with setup.py then install it with
  `"C:\Program Files (x86)\IronPython 2.7\ipy.exe" setup.py install`

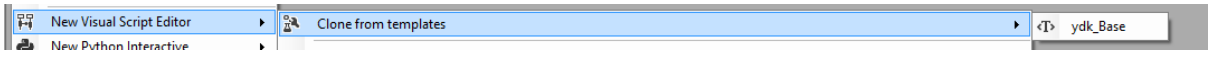
- **Install Six package**
  Also, the "six" package need to be installed. Download wheel file, then :
  `"C:\Program Files (x86)\IronPython 2.7\Scripts\pip.exe" install six-1.10.0-py2.py3-none-any.whl`

Current version of PGT only supports Cisco YDK-Py object model from Python scripts. Later version will focus of bringing the Python object model and services under the hood of C# scripts.

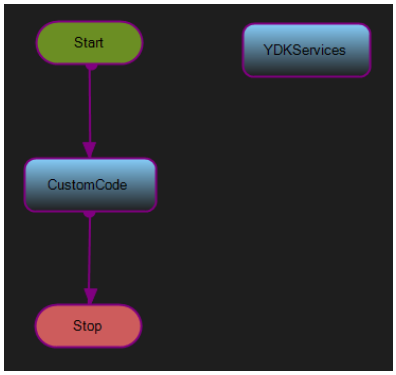
25.1 YDK-PY BASED VISUAL SCRIPTS

The most object-oriented network programming approach is to use vScripts to build Cisco YDK-Py based programs. The only requirement to run such a vScript is to use NetConf connection protocol in PGT scripts.

A YDK vScript can be built from scratch, but the easiest way is to make use of the built-in vScript template named ydk-Base and clone a new script from it:

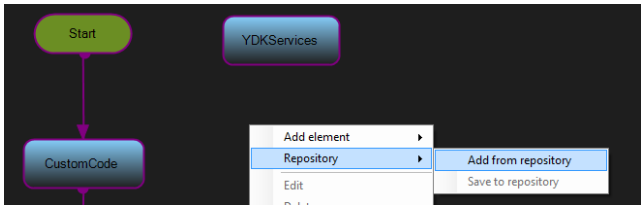


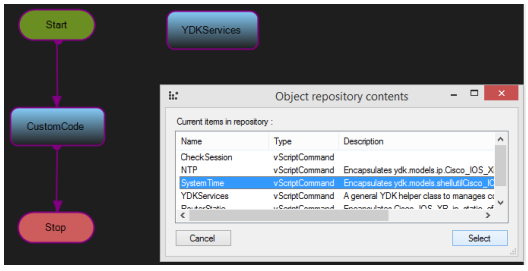
A newly cloned script is very simple and looks like the example here.



The YDKServices element ensures core YDK services while the CustomCode element can be customized to achieve the required configuration task.

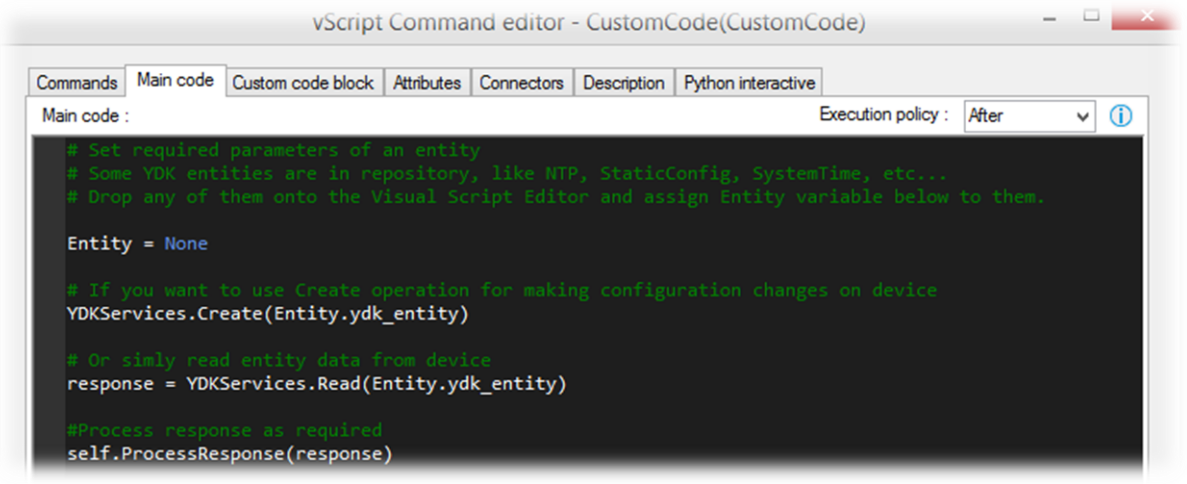
The next step will be to add a network configuration entity from the repository, like the example shows below. In this example we are inserting a SystemTime entity to the script and will customize the template as required.



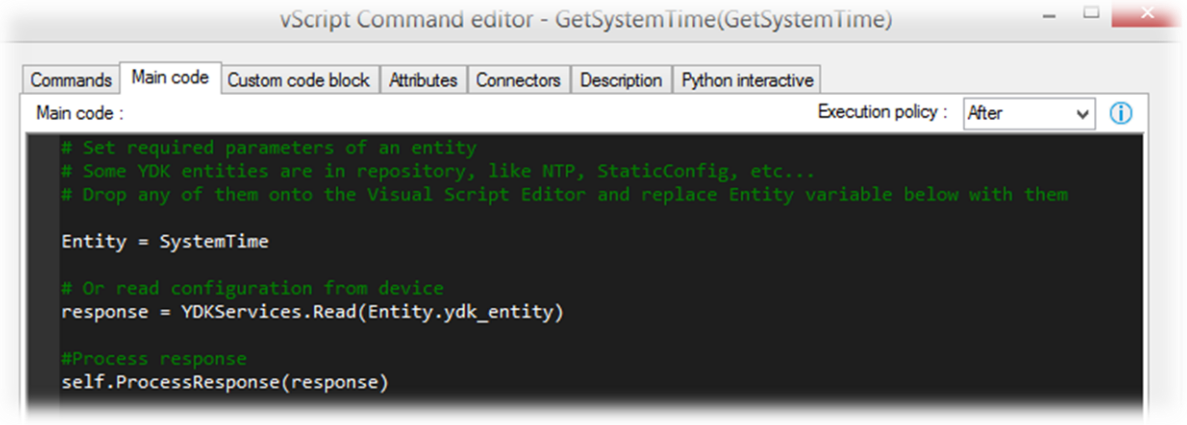


Let's rename the "Copy_of_SystemTime" object to "SystemTime" and the "CustomCode" to "GetSystemTime".

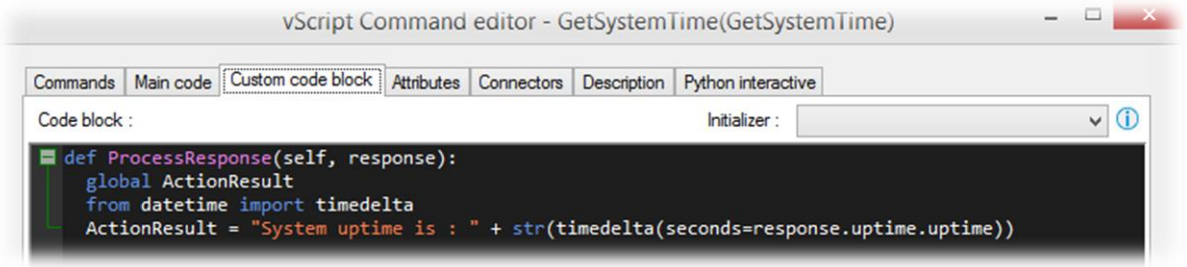
The template for CustomCode object contains the below code to be customized:



In our case we want to use SystemTime entity object we just have placed into the script, so let's modify the code as below. As you can see, this modification is very straightforward and easy to understand:

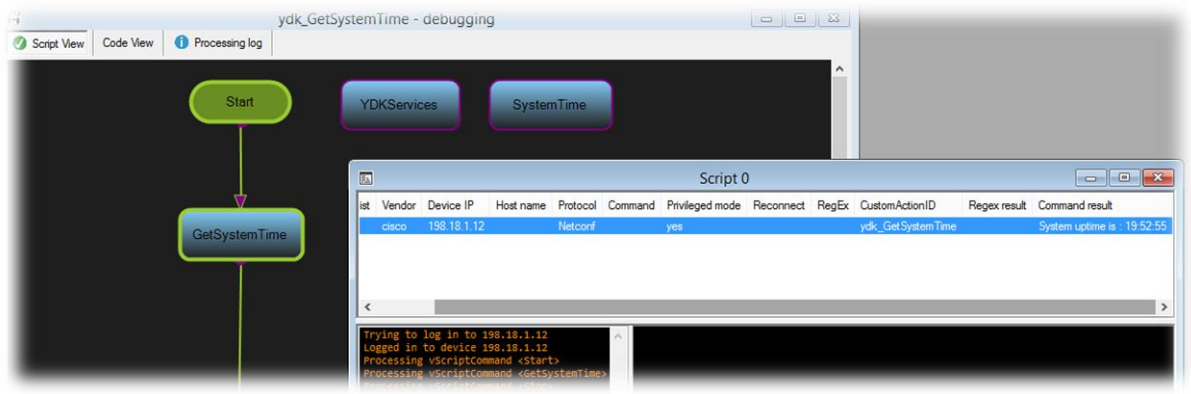


We also want to modify the empty ProcessResponse() function to return the queried SystemTime object and present it as a human legible string:



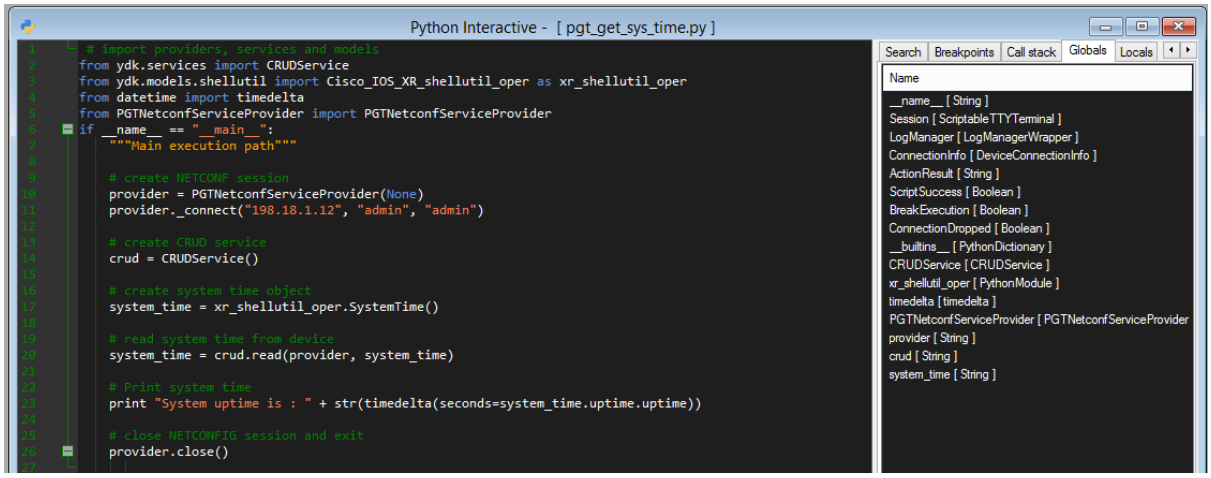
The template ensures that YDKServices.InitService() and YDKServices.Shutdown() methods are called by the Start and Stop elements which are required for successful operation.

Now we are ready to test and run the script. We are in a good shape, got system uptime:



25.2 YDK-PY PYTHON SCRIPTS

Besides Visual Scripts, PGT also supports using Cisco YDK-Py API through simple Python scripts. The above example of getting system uptime looks like the below when written as a normal Python script:



The deviance from original Cisco sample code is only the usage of PGTNetconfServiceProvider class instead of NetconfServiceProvider. The NetconfServiceProvider class is included in the YDK-Py API but can't be used by PGT for the reasons described earlier.

The script above has a static address configured and therefore can't be directly used from a PGT script where it would need to connect to different devices. But a simple modification allows us to use the script that way. Simply change the initialization of PGTNetconfServiceProvider class to use the Session object present in scope and also remove the call to connect method:

```
# create NETCONF session
provider = PGTNetconfServiceProvider(Session)
#provider._connect("198.18.1.12", "admin", "admin")
```

The Session object represents the Netconf session established by PGT internally and PGTNetconfServiceProvider will use it to exchange Netconf messages with the connected device.

Pretty Good Terminal v7.0.5.0

March 10, 2017

26 LOGGING OF INFORMATION

The text in the action pane and the terminal pane can be manually saved to text files from the context menu of the scripting form. As the script executes it updates the lines of the script. Regex result, command results and any related information are stored here. This can be saved from the by using the Save Script / results menu item.

The terminal output can also be manually saved from the script menu, or it is possible to configure automatic saving of terminal output in the Scripting settings.

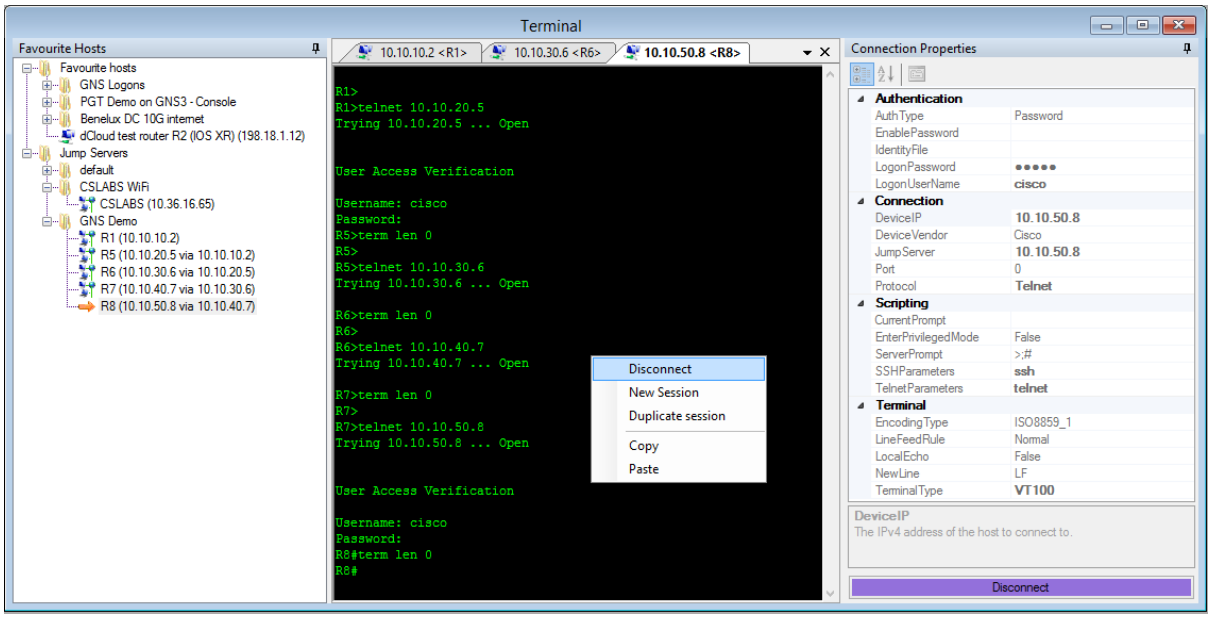
Further log file can be set for each Script window individually from the Tools/Option menu. The content of this log file is dependent on the CustomActionHandler class handling the specified custom action. There is no default content for this log file. Should the custom action handler require logging but it is not set here, PGT will display a notification window.

There is also a **debug logging** feature included starting with v4.4.36.4. When turned on, a verbose output file will be generated to a specified file. This file may be used to find out some connection related issues.
To activate this feature, PGT must be started with the following command line parameter : /d:filename.log (there must be no space between the file name and /d: prefix)

Handle the generated text file with caution, as passwords will appear as clear text in the output.

27 THE TERMINAL WINDOW

Beyond scripting capabilities, PGT also has a built-in terminal, so you do not have to start external tools for checking device configuration manually. The terminal window organizes favourite hosts and jump servers into a folder structure:



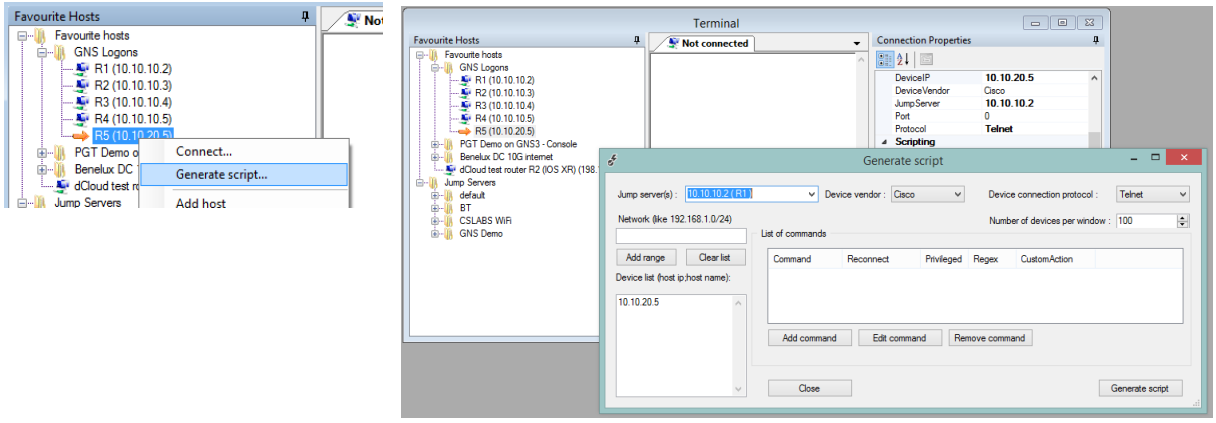
An extra feature of the built-in terminal emulator is that **it can automatically connect to hosts via jump servers** based on the configuration information in Tools/Options/Jump servers. It is also taking care of defined jump server chains, so you only have to select the last jump server and PGT will connect to all jump serves necessary to reach the specified host.

Favourite hosts can be saved and organized to a folder structure. While hosts can be saved and edited from the terminal windows, jump servers must be edited through Tools/Options. Hosts can be exported into and imported from CSV files. There are two main limitation with export/import functionality:

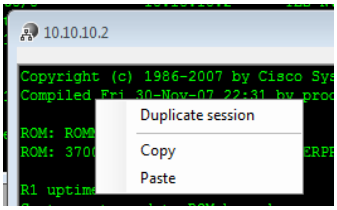
- As a safety precaution, no credentials can be imported or exported. Once hosts get imported, it is possible, however, to mass-update credentials on all hosts in a given folder hierarchy.
- Only flat structure is supported on import. In other words, hosts can be imported to a single designated folder at a time and the folder must be created manually beforehand.

The import CSV file format is not strict, except that the first line of the CSV file must be the header. On import, it is possible to map existing CSV columns to required import fields. It is also possible to set up a filter that determines the records to be imported. This way a single CSV file can be used to import hosts selectively to different folders.

Using the context menu of favourite hosts also helps generating a new script by automatically filling in the device ip addres, jump server, protocol and vendor information:



An opened terminal connection can also be duplicated using the local context menu of the terminal window :



Should you experience strange terminal behaviour, you may need to change the *Newline character* settings for various hosts/jump servers, as automatic detection does not always work with terminals.

28 COMMAND LINE OPTIONS

The following table summarize supported command line options and switches. The order of command line arguments are irrelevant to operation.

Argument	Explanation	Note
***.csv file name**	Opens scripting form and loads the script from the specified CSV file	For each files specified a new scripting form will be opened
***.py file name**	Opens Python Interactive window and loads the specified file	For each files specified a new Python Interactive window will be opened
***.vs file**	Opens Visual Script Editor and loads the file specified	For each files specified a new Visual Script Editor will be opened
/d:filename	Write debug output into the specified file.	Debug messages from the program will be written to the specified file. May be used together with /enabledebugview option
/enabledebugview	Makes the Debug View menu item visible in Help menu.	Debug messages from the program will be displayed in the Debug Trace Viewer form and can be saved to file.
/startscript	Scripts loaded from CSV files will be started automatically	Also required for /startat parameter
/startat:"22:00"	Script can be scheduled to start at a specific date/time	The specified Date / Time must be in a format matching system settings. The given start time will be applied to each csv scripts specified. ***The /startscript parameter must also be specified***.
/recurrence:minutes	Script will be restarted repeatedly at specified interval.	Minutes value must be greater than zero. *The next start time will be calculated by adding the given number of minutes as many times as required to the start time of the script to recon the next time in future.* ***The /startscript and /startat parameters must also be specified***.
/savescript:filename	Script output will be written to the specified file.	The output file will be created or truncated if already exists.

The below example shows how to repeatedly start a given csv script at a specific time and automatically save the results :

```
"c:\Program Files (x86)\PGT\PrettyGoodTerminal.exe" "C:\Temp\PGT_Demo_GNS3_basicscript.csv"
/startscript /recurrence:3 /savescript:"testsave.csv" /startat:11:52
```

29 DEVELOPMENT SUPPORT

PGT supports different ways of adding custom logic to scripting. For the details of development support please refer to the development documentation.

30 LICENSE

Portions of the software are licensed under Apache License v2.0

An individual license might be required to run specific Editions of the software. This license is generated and provided online through the product support website.

Before using the product you must read and accept licensing terms described on
http://www.prettygoodterminal.com/Licensing.aspx

31 LIMITATION OF LIABILITY

UNDER NO LEGAL THEORY, INCLUDING, BUT NOT LIMITED TO, NEGLIGENCE, TORT, CONTRACT, STRICT LIABILITY, OR OTHERWISE, SHALL THE AUTHOR OF THE PROGRAM CODE BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, EXEMPLARY, RELIANCE OR CONSEQUENTIAL DAMAGES INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, ACCURACY OF RESULTS, COMPUTER FAILURE OR MALFUNCTION, OR DAMAGES RESULTING FROM USE. THE AUTHOR OF THE PROGRAM CODE LIABILITY FOR DAMAGES OF ANY KIND WHATSOEVER ARISING OUT OF THIS AGREEMENT SHALL BE LIMITED TO THE FEES PAID BY LICENSEE FOR THE SOFTWARE.

32 WARRANTY DISCLAIMER

THE AUTHOR OF THE PROGRAM CODE PROVIDES THE SOFTWARE "AS IS" AND WITHOUT WARRANTY OF ANY KIND, AND HEREBY DISCLAIMS ALL EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, PERFORMANCE, ACCURACY, RELIABILITY, QUIET ENJOYMENT, INTEGRATION, TITLE, NON-INTERFERENCE AND NON-INFRINGEMENT. FURTHER, THE AUTHOR OF THE PROGRAM CODE DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS THAT THE SOFTWARE WILL BE FREE FROM BUGS OR THAT ITS USE WILL BE UNINTERRUPTED OR THAT THE SOFTWARE OR WRITTEN MATERIALS WILL BE CORRECT, ACCURATE, OR RELIABLE.

THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS AGREEMENT..