Exscript Documentation

Release 0.3.4

Samuel Abels

Sep 11, 2017

Contents

Exscript is a Python module and a template processor for automating network connections over protocols such as Telnet or SSH. We attempt to create the best possible set of tools for working with Telnet and SSH.

Exscript also provides a set of tools and functions for sysadmins, that simplify **regular expression matching**, **reporting** by email, **logging**, or **syslog** handling, **CSV parsing**, **ip address handling**, **template processing**, and many more.

Exscript may be used to automate sessions with routers from Cisco, Juniper, OneAccess, Huawei, or any others. If you want to configures machines running Linux/Unix, IOS, IOS-XR, JunOS, VRP, or any other operating system that can be used with a terminal, Exscript is what you are looking for.

CHAPTER 1

Using Exscript with Python

Make sure to check out the *Python Tutorial*. You may also want to look at the Python examples.

CHAPTER 2

Using the Exscript command line tool

Have a look at our *CLI Tutorial*. You will also want to learn about *Exscript Templates*.

CHAPTER 3

Main design goals

- Exscript provides **high reliability** and **scalability**. Exscript is used by some of the world's largest ISPs to maintain hundreds of thousands of sessions every day.
- Exscript is **extremely well tested**. The Exscript public API has almost 100% test coverage.
- Exscript is **protocol agnostic**, so if you are migrating from Telnet to SSH later, you can switch easily by simply changing an import statement.

CHAPTER 4

Development

Exscript is on GitHub.

CHAPTER 5

License

Exscript is published under the MIT licence.

CHAPTER 6

Contents

Installation

Prerequisites

Exscript requires Python 2.7, and the following modules:

```
future
configparser
pycrypto
paramiko>=1.17
```

Installing with PIP

```
sudo pip install exscript
```

Installing from GitHub

```
git clone git://github.com/knipknap/exscript
cd exscript
sudo make install
```

Running the automated test suite

If you installed from GitHub, you can run the integrated testsuite:

```
make tests
```

There shouldn't be any errors, so if something comes up, please file a bug.

Python Tutorial

Introduction

This is a step by step introduction to using Exscript in Python.

We'll assume that Exscript is already installed. You can confirm that your installation works by typing `exscript --version` into a terminal; if this prints the version number, your installation is complete.

We will also assume that you have at least a little bit of programming experience, though most of the examples should be pretty simple.

Exscript also has extensive *API documentation*, which may be used as a reference throughout this tutorial.

Getting started

As a first simple test, let's try to connect to a network device via SSH2, and execute the `uname -a` command on it.

Create a file named `start.py` with the following content:

```
from Exscript.util.interact import read_login
from Exscript.protocols import SSH2

account = read_login()     # Prompt the user for his name and password
conn = SSH2()              # We choose to use SSH2
conn.connect('localhost') # Open the SSH connection
conn.login(account)        # Authenticate on the remote host
conn.execute('uname -a')   # Execute the "uname -a" command
conn.send('exit\r')        # Send the "exit" command
conn.close()               # Wait for the connection to close
```

Awesome fact: Just replace `SSH2` by `Telnet` and it should still work with Telnet devices.

As you can see, we prompt the user for a username and a password, and connect to `localhost` using the entered login details. Once logged in, we execute `uname -a`, log out, and make sure to wait until the remote host has closed the connection.

You can see one important difference: We used `conn.execute` to run `uname -a`, but we used `conn.send` to execute the `exit` command. The reason is that *''conn.execute'' waits until the server has acknowledged that the command has completed*, while `conn.send` does not. Since the server won't acknowledge the `exit` command (instead, it just closes the connection), using `conn.execute` would lead to an error.

Making it easier

While the above serves as a good introduction on how to use *`Exscript.protocols`*, it has a few drawbacks:

1. It only works for SSH2 or for Telnet, but not for both.
2. It contains a lot of unneeded code.
3. You can't use the script to connect to multiple hosts.

Let's solve drawbacks 1. and 2. first. Here is a shortened version of the above script:

```
from Exscript.util.start import quickstart

def do_something(job, host, conn):
    conn.execute('uname -a')
```

```
quickstart('ssh://localhost', do_something)
```

As you can see, we made two major changes:

1. We moved the code that executes `uname -a` into a function named `do_something`. (Note: We could have picked any other name for the function.)
2. We imported and used the *Exscript.util.start.quickstart()* function.

Exscript.util.start.quickstart() does the following:

1. It prompts the user for a username and a password.
2. It connects to the specified host, using the specified protocol.
3. It logs in using the given login details.
4. It calls our `do_something()` function.
5. When `do_something()` completes, it closes the connection.

Running a script on multiple hosts

In practice, you may want to run this script on multiple hosts, and optimally at the same time, in parallel. Using the *Exscript.util.start.quickstart()* function, this is now really easy:

```
from Exscript.util.start import quickstart

def do_something(job, host, conn):
    conn.execute('uname -a')

hosts = ['ssh://localhost', 'telnet://anotherhost']
quickstart(hosts, do_something, max_threads=2)
```

We only changed the last lines of the script:

1. We pass in two hosts, `localhost` and `anotherhost`. Note that `localhost` uses SSH, and `anotherhost` speaks Telnet.
2. We added the `max_threads=2` argument. This tells Exscript to open two network connections in parallel.

If you run this script, it will again ask for the login details, and run `do_something()` for both hosts in parallel.

Note that the login details are only asked once and used on both hosts - this may or may not be what you want. For instructions one using different login mechanisms please refer to the section on accounts later.

Loading hosts from a text file

If you do not wish to hard code the host names into the script, you may also list them in a text file and load it using *Exscript.util.file.get_hosts_from_file()* as follows:

```
from Exscript.util.start import start
from Exscript.util.file import get_hosts_from_file

def do_something(job, host, conn):
    conn.execute('uname -a')
```

```
hosts = get_hosts_from_file('myhosts.txt')
start(hosts, do_something, max_threads=2)
```

Reading login information

Depending on how you would like to provide the login information, there are a few options. The first is by hard coding it into the hostname:

```
hosts = ['ssh://localhost', 'telnet://myuser:mypassword@anotherhost']
quickstart(hosts, do_something, max_threads=2)
```

In this case, `quickstart` still prompts the user for his login details, but the entered information is only used on hosts that do not have a user/password combination included in the hostname.

If you do not wish to hard code the login details into the hostname, you can also use the Exscript.Host object as shown in the following script:

```
from Exscript import Host, Account
...
account1 = Account('myuser', 'mypassword')
host1 = Host('ssh://localhost')
host1.set_account(account1)

account2 = Account('myuser2', 'mypassword2')
host2 = Host('ssh://otherhost')
host2.set_account(account2)

quickstart([host1 , host2], do_something, max_threads=2)
```

This script still has the problem that it prompts the user for login details, even though the details are already known. By using *Exscript.util.start.start()* instead of *Exscript.util.start.quickstart()*, you can avoid the prompt, and optionally pass in a pre-loaded list of accounts as seen in the following code:

```
from Exscript.util.start import start
from Exscript.util.file import get_hosts_from_file

def do_something(job, host, conn):
    conn.execute('uname -a')

accounts = [] # No account needed.
hosts = get_hosts_from_file('myhosts.txt')
start(accounts, hosts, do_something, max_threads=2)
```

Instead of passing in no account at all, you may also create one in the script:

```
from Exscript import Account
...
accounts = [Account('myuser', 'mypassword')]
...
```

Or you may load it from an external file:

```
from Exscript.util.file import get_accounts_from_file
...
accounts = get_accounts_from_file('accounts.cfg')
...
```

Note that `accounts.cfg` is a config file with a defined syntax as seen in the API documentation for *Exscript.util.file.get_accounts_from_file()*.

Logging

Exscript has built-in support for logging. In a simple case, just pass the `stdout` and `stderr` parameters for log and errors to `start()` or `quickstart()` and you are done:

```
with open('log.txt','w+') as fp:
    start(accounts, hosts, do_something, stdout=fp)
```

Exscript creates one logfile per device. In the case that an error happened on the remote device, it creates an additional file that contains the error (including Python's traceback).

Interacting with a device

So far we only fired and forgot a command on a device, there was no true interaction. But Exscript does a lot to make interaction with a device easier. The first notable tool is *Exscript.util.match* - a module that builds on top of Python's regular expression support. Let's look at an example:

```
from Exscript.util.start import quickstart
from Exscript.util.match import first_match

def do_something(job, host, conn):
    conn.execute('uname -a')
    print "The response was", repr(conn.response)
    os, hostname = first_match(conn, r'^(\S+)\s+(\S+)')
    print "The hostname is:", hostname
    print "Operating system:", os

quickstart('ssh://localhost', do_something)
```

The experienced programmer will probably wonder what happens when *Exscript.util.match.first_match()* does not find a match. The answer is: It will return a tuple `None, None`. In other words, no matter what happens, the one liner can not fail, because *Exscript.util.match.first_match()* always returns a tuple containing the same number of elements as there are groups (bracket expressions) in the regular expression. This is more terse than the following typical regular idiom:

```
match = re.match(r'^(\S+)\s+(\S+)', conn.response)
if match:
    print match.group(1)
```

Similarly, the following use of *Exscript.util.match.any_match()* can never fail:

```
from Exscript.util.start import quickstart
from Exscript.util.match import any_match

def do_something(job, host, conn):
    conn.execute('ls -l')
    for permissions, filename in any_match(conn, r'^(\S+).*\s+(\S+)$'):
        print "The filename is:", filename
        print "The permissions are:", permissions

quickstart('ssh://localhost', do_something)
```

Exscript.util.match.any_match() is designed such that it always returns a list, where each item contains a tuple of the same size. So there is no need to worry about checking the return value first.

Advanced queueing and reporting

Exscript.Queue is a powerful, multi-threaded environment for automating more complex tasks. It comes with features such as logging, user account management, and error handling that make things a lot easier. The above functions *Exscript.util.start.start()* and *Exscript.util.start.quickstart()* are just convenience wrappers around this queue.

In some cases, you may want to use the *Exscript.Queue* directly. Here is a complete example that also implements reporting:

```
#!/usr/bin/env python
from Exscript import Queue, Logger
from Exscript.util.log import log_to
from Exscript.util.decorator import autologin
from Exscript.util.file import get_hosts_from_file, get_accounts_from_file
from Exscript.util.report import status, summarize

logger = Logger()  # Logs everything to memory.


@log_to(logger)
@autologin()
def do_something(job, host, conn):
    conn.execute('show ip int brie')

# Read input data.
accounts = get_accounts_from_file('accounts.cfg')
hosts = get_hosts_from_file('hostlist.txt')

# Run do_something on each of the hosts. The given accounts are used
# round-robin. "verbose=0" instructs the queue to not generate any
# output on stdout.
queue = Queue(verbose=5, max_threads=5)
queue.add_account(accounts)      # Adds one or more accounts.
queue.run(hosts, do_something)   # Asynchronously enqueues all hosts.
queue.shutdown()                 # Waits until all hosts are completed.

# Print a short report.
print status(logger)
print summarize(logger)
```

Emulating a remote device

Exscript also provides a dummy protocol adapter for testing purposes. It emulates a remote host and may be used in place of the Telnet and SSH adapters:

```
from Exscript.protocols import Dummy
conn = Dummy()
...
```

In order to define the behavior of the dummy, you may define it by providing a Python file that maps commands to responses. E.g.:

```
def echo(command):
    return command.split(' ', 1)[1]

commands = (
('ls -l', """
-rw-r--r-- 1 sab nmc 1906 Oct  5 11:18 Makefile
-rw-r--r-- 1 sab nmc 1906 Oct  5 11:18 myfile
"""),

(r'echo [\r\n]+', echo)
)
```

Note that the command name is a regular expression, and the response may be either a string or a function.

CLI Tutorial

Introduction

With the *exscript* command line tool, you can quickly automate a conversation with a device over Telnet or SSH.

This is a step by step introduction to using the Exscript command line tool.

We'll assume that Exscript is already installed. You can confirm that your installation works by typing `exscript --version` into a terminal; if this prints the version number, your installation is complete.

Getting started

As a first simple test, let's try to connect to a Linux/Unix machine via SSH2, and execute the `uname -a` command on it.

Create a file named `test.exscript` with the following content:

```
uname -a
```

To run this Exscript template, just start Exscript using the following command:

```
exscript test.exscript ssh://localhost
```

Awesome fact: Just replace `ssh://` by `telnet://` and it should still work with Telnet devices.

Hint: The example assumes that `localhost` is a Unix server where SSH is running. You may of course change this to either an ip address (such as `ssh://192.168.0.1`), or any other hostname.

Exscript will prompt you for a username and a password, and connect to `localhost` using the entered login details. Once logged in, it executes `uname -a`, waits for a prompt, and closes the connection.

Running a script on multiple hosts

In practice, you may want to run this script on multiple hosts, and optimally at the same time, in parallel. Using the `-c` option, you tell Exscript to open multiple connections at the same time:

```
exscript -c 2 test.exscript ssh://localhost ssh://otherhost
```

`-c 2` tells Exscript to open two connections in parallel. So if you run this script, Exscript will again ask for the login details, and run `uname -a` for both hosts in parallel.

Note that the login details are only asked once and used on both hosts - this may or may not be what you want. The following section explains some of the details of using different login accounts.

Reading login information

Depending on how you would like to provide the login information, there are a few options. The first is by including it in the hostname:

```
exscript -c 2 test.exscript ssh://localhost
ssh://user:password@otherhost
```

In this case, Exscript still prompts the user for his login details, but the entered information is only used on hosts that do not have a user/password combination included in the hostname.

If you do not want for Exscript to prompt for login details, the `-i` switch tells Exscript to not ask for a user and password. You need to make sure that all hosts have a user and password in the hostname if you use it.

Reading host names from a text file

If you do not wish to hard code the host names or login details into the command, you may also list the hosts in an external file and load it using the `--hosts` option as follows:

```
exscript -c 2 --hosts myhosts.txt test.exscript
```

Note that *hosts.txt* is a file containing a list of hostnames, e.g.:

```
host1
host2
...
host20
```

Reading host names from a CSV file

Similar to the `--hosts`, you may also use `--csv-hosts` option to pass a list of hosts to Exscript while at the same time providing a number of variables to the script. The CSV file has the following format:

```
address my_variable another_variable
telnet://myhost value another_value
ssh://yourhost hello world
```

Note that fields are separated using the tab character, and the first line **must** start with the string "address" and is followed by a list of column names.

In the Exscript template, you may then access the variables using those column names:

```
ls -l $my_variable
touch $another_variable
```

Using Account Pooling

You can also pass in a pre-loaded list of accounts from a separate file. The accounts from the file are used for hosts that do not have a user/password combination specified in their URL.

```
exscript -c 2 --hosts myhosts.txt --account-pool accounts.cfg test.exscript
```

Note that `accounts.cfg` is a config file with a defined syntax as seen in the API documentation for *Exscript.util.file.get_accounts_from_file()*.

It is assumed that you are aware of the security implications of saving login passwords in a text file.

Logging

Exscript has built-in support for logging - just pass the `--logdir` or `-l` option with a path to the directory in which logs are stored:

```
exscript -l /tmp/logs -c 2 --hosts myhosts.txt --account-pool accounts.cfg test.
↪exscript
```

Exscript creates one logfile per device. In the case that an error happened on the remote device, it creates an additional file that contains the error (including Python's traceback).

Interacting with a device

So far we only fired and forgot a command on a device, there was no true interaction. But Exscript does a lot to make interaction with a device easier. The first notable tool is the `extract` keyword. Let's look at an example:

```
uname -a{extract /^(\S+)\s+(\S+)/ as os, hostname}
```

The Exscript Template Language

The Exscript template language is in some ways comparable to Expect, but has unique features that make it a lot easier to use and understand for non-developers.

A first example:

```
{fail "not a Cisco router" if connection.guess_os() is not "ios"}

show ip interface brief {extract /^(\S+)\s/ as interfaces}
configure terminal
{loop interfaces as interface}
    interface $interface
    description This is an automatically configured interface description!
    cdp enable
    no shut
    exit
{end}
copy running-config startup-config
```

Exscript templates support many more commands. Here is another example, to automate a session with a Cisco router:

```
show version {extract /^(cisco)/ as vendor}
{if vendor is "cisco"}
  show ip interface brief {extract /^(\S+)\s/ as interfaces}
  {loop interfaces as interface}
    show running interface $interface
    configure terminal
    interface $interface
    no shut
    end
  {end}
  copy running-config startup-config
{end}
```

Advanced Templates

Exscript templates support many more commands. For a full overview over the template language, please check *Exscript Templates*.

Command Line Options

Overview

You can pass parameters (or lists of parameters) into the templates and use them to drive what happens on the remote host. Exscript easily supports logging, authentication mechanisms such as TACACS and takes care of synchronizing the login procedure between multiple running connections.

These features are enabled using simple command line options. The following options are currently provided:

```
Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --account-pool=FILE   Reads the user/password combination from the given
                        file instead of prompting on the command line. The
                        file may also contain more than one user/password
                        combination, in which case the accounts are used round
                        robin.
  -c NUM, --connections=NUM
                        Maximum number of concurrent connections. NUM is a
                        number between 1 and 20, default is 1.
  --csv-hosts=FILE      Loads a list of hostnames and definitions from the
                        given file. The first line of the file must have the
                        column headers in the following syntax: "hostname
                        [variable] [variable] ...", where the fields are
                        separated by tabs, "hostname" is the keyword
                        "hostname" and "variable" is a unique name under which
                        the column is accessed in the script. The following
                        lines contain the hostname in the first column, and
                        the values of the variables in the following columns.
  -d PAIR, --define=PAIR
                        Defines a variable that is passed to the script. PAIR
                        has the following syntax: STRING=STRING.
  --default-domain=STRING
                        The IP domain name that is used if a given hostname
                        has no domain appended.
```

```
--delete-logs         Delete logs of successful operations when done.
-e EXSCRIPT, --execute=EXSCRIPT
                      Interprets the given string as the script.
--hosts=FILE          Loads a list of hostnames from the given file (one
                      host per line).
-i, --non-interactive
                      Do not ask for a username or password.
-l DIR, --logdir=DIR  Logs any communication into the directory with the
                      given name. Each filename consists of the hostname
                      with "_log" appended. Errors are written to a separate
                      file, where the filename consists of the hostname with
                      ".log.error" appended.
--no-echo             Turns off the echo, such that the network activity is
                      no longer written to stdout. This is already the
                      default behavior if the -c option was given with a
                      number greater than 1.
-n, --no-authentication
                      When given, the authentication procedure is skipped.
                      Implies -i.
--no-auto-logout      Do not attempt to execute the exit or quit command at
                      the end of a script.
--no-prompt           Do not wait for a prompt anywhere. Note that this will
                      also cause Exscript to disable commands that require a
                      prompt, such as "extract".
--no-initial-prompt   Do not wait for a prompt after sending the password.
--no-strip            Do not strip the first line of each response.
--overwrite-logs      Instructs Exscript to overwrite existing logfiles. The
                      default is to append the output if a log already
                      exists.
-p STRING, --protocol=STRING
                      Specify which protocol to use to connect to the remote
                      host. Allowed values for STRING include: dummy,
                      pseudo, ssh, ssh1, ssh2, telnet. The default protocol
                      is telnet.
--retry=NUM           Defines the number of retries per host on failure.
                      Default is 0.
--retry-login=NUM     Defines the number of retries per host on login
                      failure. Default is 0.
--sleep=TIME          Waits for the specified time before running the
                      script. TIME is a timespec as specified by the 'sleep'
                      Unix command.
--ssh-auto-verify     Automatically confirms the 'Host key changed' SSH
                      error  message with 'yes'. Highly insecure and not
                      recommended.
--ssh-key=FILE        Specify a key file that is passed to the SSH client.
                      This is equivalent to using the "-i" parameter of the
                      openssh command line client.
-v NUM, --verbose=NUM
                      Print out debug information about the network
                      activity. NUM is a number between 0 (min) and 5 (max).
                      Default is 1.
-V NUM, --parser-verbose=NUM
                      Print out debug information about the Exscript
                      template parser. NUM is a number between 0 (min) and 5
                      (max). Default is 0.
```

Using Account Pooling

It is possible to provide an account pool from which Exscript takes a user account whenever it needs to log into a remote host. Depending on the authentification mechanism used in your network, you may significantly increase the speed of parallel connections by using more than one account in parallel. The following steps need to be taken to use the feature:

1. Create a file with the following format:

   ```
   [account-pool]
   user=password
   other_user=another_password
   somebody=yet_another_password
   ```

   Note that the password needs to be base64 encrypted, just putting plain passwords there will NOT work.

2. Save the file. It is assumed that you are aware of the security implications of saving your login passwords in a text file.

3. Start Exscript with the `--account-pool FILE` option. For example:

   ```
   exscript --account-pool /home/user/my_accounts my.exscript host4
   ```

Using a CSV file as input

By providing the –csv-hosts option you may pass a list of hosts to Exscript while at the same time providing a number of variables to the script. The CSV file should have the following format:

```
hostname my_variable another_variable
myhost value another_value
yourhost hello world
```

Note that fields are separated using the tab character, and the first line must start with the string “hostname” and is followed by a list of column names.

In the Exscript, you may then access the variables using those column names:

```
ls -l $my_variable
touch $another_variable
```

Exscript Templates

Simple example

The simplest possible template is one that contains only the commands that are sent to the remote host. For example, the following Exscript template can be used to retrieve the response of the ls -l and df commands from a unix host:

```
ls -l
df
```

Comments

Lines starting with a hash (“#”) are interpreted as comments and ignored. For example:

```
1. # This line is ignored...
2. {if __hostname__ is "test"}
3.   # ...and so is this one.
4. {end}
```

Using Variables

The following template uses a variable to execute the ls command with a filename as an argument:

```
ls -l $filename
```

When executing it from the command line, use:

```
exscript -d filename=.profile my.exscript localhost
```

Note that the -d switch allows passing variables into the template. The example executes the command ls -l .profile. You can also assign a value to a variable within a template:

```
{filename = ".profile"}
ls -l $filename
```

You may also use variables in strings by prefixing them with the "$" character:

```
1. {test = "my test"}
2. {if "my test one" is "$test one"}
3.   # This matches!
4. {end}
```

In the above template line 3 is reached. If you don't want the "$" character to be interpreted as a variable, you may prefix it with a backslash:

```
1. {test = "my test"}
2. {if "my test one" is "\$test one"}
3.   # This does not match
4. {end}
```

Adding Variables To A List

In Exscript, every variable is a list. You can also merge two lists by using the "append" keyword:

```
1. {
2.   test1 = "one"
3.   test2 = "two"
4.   append test2 to test1
5. }
```

This results in the "test1" variable containing two items, "one" and "two".

Using Built-in Variables

The following variables are available in any Exscript template, even if they were not explicitly passed in:

1. `__hostname__` contains the hostname that was used to open the current connection.

2. `__response__` contains the response of the remote host that was received after the execution of the last command.

Built-in variables are used just like any other variable. You can also assign a new value to a built-in variable in the same way.

Using Expressions

An expression is a combination of values, variables, operators, and functions that are interpreted (evaluated) according to particular rules and that produce a return value. For example, the following code is an expression:

```
name is "samuel" and 4 * 3 is not 11
```

In this expression, *name* is a variable, *is*, *is not*, and * are operators, and *"samuel"*, *4*, *3*, and *11* are values. The return value of this particular expression is *true*.

In Exscript, expressions are used in many places, such as if-conditions or variable assignments. The following operators may be used in an expression.

Priority 1 Operators

1. `*` multiplies the operators (numerically).
2. `/` divides the operators (numerically).

Priority 2 Operators

1. `+` adds the operators (numerically).
2. `-` subtracts the operators (numerically).

Priority 3 Operators

1. `.` concatenates two strings.

Priority 4 Operators

1. `is` tests for equality. If both operators are lists, only the first item in the list is compared.
2. `is not` produces the opposite result from is.
3. `in` tests whether the left string equals any of the items in the list given as the right operator.
4. `not in` produces the opposite result from in.
5. `matches` tests whether the left operator matches the regular expression that is given as the right operator.
6. `ge` tests whether the left operator is (numerically) greater than or equal to the right operator.
7. `gt` tests whether the left operator is (numerically) greater than the right operator.
8. `le` tests whether the left operator is (numerically) less than or equal to the right operator.
9. `lt` tests whether the left operator is (numerically) less than the right operator.

Priority 5 Operators

1. `not` inverts the result of a comparison.

Priority 6 Operators

1. `and` combines two tests such that a logical AND comparison is made. If the left operator returns FALSE, the right operator is not evaluated.
2. `or` combines two tests such that a logical OR comparison is made. If the left operator returns TRUE, the right operator is not evaluated.

Using Hexadecimal Or Octal Numbers

Exscript also supports hexadecimal and octal numbers using the following syntax:

```
{
  if 0x0a is 012
    sys.message("Yes")
  else
    sys.message("No")
  end
}
```

Using Regular Expressions

At some places Exscript uses Regular Expressions. These are NOT the same as the expressions documented above, and if you do not know what regular expressions are it is recommended that you read a tutorial on regular expressions first.

Exscript regular expressions are similar to Perl and you may also append regular expression modifiers to them. For example, the following is a valid regular expression in Exscript:

```
/^cisco \d+\s+\w/i
```

Where the appended "i" is a modifier (meaning case-insensitive). A full explanation of regular expressions is not given here, because plenty of introductions have been written already and may be found with the internet search engine of your choice.

Built-in Commands

By default, any content of an Exscript template is sent to the remote host. However, you can also add instructions with special meanings. Such instructions are enclosed by curly brackets (`{` and `}`). The following commands all use this syntax.

Extracting Data From A Response

Exscript lets you parse the response of a remote host using regular expressions. If you do not know what regular expressions are, please read a tutorial on regular expressions first.

extract ... into ...

If you already know what regular expressions are, consider the following template:

```
ls -l {extract /^(d.*)/ into directories}
```

The extract command matches each line of the response of "ls -l" against the regular expression `/(d.*)/` and then appends the result of the first match group (a match group is a part of a regular expression that is enclosed by brackets) to the list variable named directories.

You can also extract the value of multiple match groups using the following syntax:

```
ls -l {extract /^(d\S+)\s.*\s(\S+)$/ into modes, directories}
```

This extracts the mode and the directory name from each line and appends them to the modes and directories lists respectively. You can also apply multiple matches to the same response using the following syntax:

```
ls -l {
  extract /^[^d].*\s(\S+)$/ into files
  extract /^d.*\s(\S+)$/    into directories
}
```

There is no limit to the number of extract statements.

extract ... into ... from ...

When used without the "from" keyword, "extract" gets the values from the last command that was executed. You may however also instruct Exscript to extract the values from a variable. The following example shows how this may be done.

```
ls -l {
  extract /^(.*)/  into lines
  extract /^(d.*)/ into directories from lines
}
```

extract ... as ...

The "as" keyword is similar to "into", the difference being that with as, the destination variable is cleared before new values are appended.

```
ls -l {extract /^(d.*)/ as directories}
```

"as" may be used anywhere where "into" is used.

If-Conditions

You can execute commands depending on the runtime value of a variable or expression.

if ... end

The following Exscript template executes the `ls` command only if `ls -l .profile` did not produce a result:

```
ls -l .profile {extract /(\.profile)$/ as found}
{if found is not ".profile"}
  ls
{end}
```

if ... else ... end

You can also add an else condition:

```
ls -l .profile {extract /(\.profile)$/ as found}
{if found is not ".profile"}
  ls
{else}
  touch .profile
{end}
```

if ... else if ...

You can perform multiple matches using else if:

```
ls -l .profile {extract /(.*profile)$/ as found}
{if found is ".profile"}
  ls
{else if found matches /my_profile/}
  ls -l p*
{else}
  touch .profile
{end}
```

Loops

Loops with counters

You can execute commands multiple times using the "loop" statement. The following Exscript template executes the "ls" command three times:

```
{number = 0}
{loop until number is 3}
  {number = number + 1}
  ls $directory
{end}
```

Similarly, the while statement may be used. The following script is equivalent:

```
{number = 0}
{loop while number is not 3}
  {number = number + 1}
  ls $directory
{end}
```

Another alternative is using the "loop from ... to ..." syntax, which allows you to specify a range of integers:

```
# Implicit "counter" variable.
{loop from 1 to 3}
  ls $directory$counter
{end}

# Explicit variable name.
{loop from 1 to 3 as number}
  ls $directory$number
{end}
```

Loops over lists

The following Exscript template uses the ls command to show the content of a list of subdirectories:

```
ls -l {extract /^d.*\s(\S+)$/ as directories}
{loop directories as directory}
  ls $directory
{end}
```

You can also walk through multiple lists at once, as long as they have the same number of items in it:

```
ls -l {extract /^(d\S+)\s.*\s(\S+)$/ as modes, directories}
{loop modes, directories as mode, directory}
  echo Directory has the mode $mode
  ls $directory
{end}
```

List loops can also be combined with the until or while statement seen in the previous section:

```
ls -l {extract /^d.*\s(\S+)$/ as directories}
{loop directories as directory until directory is "my_subdir"}
  ls $directory
{end}
```

Functions

Exscript provides builtin functions with the following syntax:

```
type.function(EXPRESSION, [EXPRESSION, ...])
```

For example, the following function instructs Exscript to wait for 10 seconds:

```
{sys.wait(10)}
```

For a list of supported functions please check here:

Exiting A Script

fail “message”

The “fail” keyword may be used where a script should terminate immediately.

```
show something
{fail "Error: Failed!"}
show something else
```

In this script, the "show something else" line is never reached.

fail "message" if ...

It is also possible to fail only if a specific condition is met. The following snippet terminates only if a Cisco router does not have a POS interface:

```
show ip int brie {
  extract /^(POS)\S+/ as pos_interfaces
  fail "No POS interface found!" if "POS" not in pos_interfaces
}
```

Error Handling

Exscript attempts to detect errors, such as commands that are not understood by the remote host. By default, Exscript considers any response that includes one of the following strings to be an error:

```
invalid
incomplete
unrecognized
unknown command
[^\r\n]+ not found
```

If this default configuration does not suit your needs, you can override the default, setting it to any regular expression of your choice using the following function:

```
{connection.set_error(/[Ff]ailed/)}
```

Whenever such an error is detected, the currently running Exscript template is cancelled on the current host. For example, when the following script is executed on a Cisco router, it will fail because there is no ls command:

```
ls -l
show ip int brief
```

The "show ip int brief" command is not executed, because an error is detected at "ls -l" at runtime.

If you want to execute the command regardless, you can wrap the "ls" command in a "try" block:

```
{try}ls -l{end}
show ip int brief
```

You can add as many commands as you like in between a try block. For example, the following will also work:

```
{try}
  ls -l
  df
  show running-config
{end}
show ip int brief
```

Trouble Shooting

Common Pitfalls

Generally, the following kinds of errors that may happen at runtime:

1. **A script deadlocks.** In other words, Exscript sends no further commands even though the remote host is already waiting for a command. This generally happens when a prompt is not recognized.
2. **A script executes a command before the remote host is ready.** This happens when a prompt was detected where none was really included in the response.
3. **A script terminates before executing all commands.** This happens when two (or more) prompts were detected where only one was expected.

The following sections explain when these problems may happen and how to fix them.

Deadlocks

Exscript tries to automatically detect a prompt, so generally you should not have to worry about prompt recognition. The following prompt types are supported:

```
[sam123@home ~]$
sam@knip:~/Code/exscript$
sam@MyHost-X123$
MyHost-ABC-CDE123$
MyHost-A1$
MyHost-A1(config)$
FA/0/1/2/3$
FA/0/1/2/3(config)$
admin@s-x-a6.a.bc.de.fg:/$
```

Note: The trailing "$" may also be any of the following characters: "$#>%"

However, in some rare cases, a remote host may have a prompt that Exscript can not recognize. Similarly, in some scripts you might want to execute a special command that triggers a response that does not include a prompt Exscript can recognize.

In both cases, the solution includes defining the prompt manually, such that Exscript knows when the remote host is ready. For example, consider the following script:

```
1. show ip int brief
2. write memory
3. {enter}
4. show configuration
```

Say that after executing line 2 of this script, the remote host asks for a confirmation, saying something like this:

```
Are you sure you want to overwrite the configuration? [confirm]
```

Because this answer does not contain a standard prompt, Exscript can not recognize it. We have a deadlock. To fix this, we must tell Exscript that a non-standard prompt should be expected. The following change fixes the script:

```
1. show ip int brief
2. {connection.set_prompt(/\[confirm\]/)}
3. write memory
4. {connection.set_prompt()}
```

```
5. {enter}
6. show configuration
```

The second line tells Exscript to wait for "[confirm]" after executing the following commands. Because of that, when the write memory command was executed in line 3, the script does not deadlock (because the remote host's response includes "[confirm]"). In line 4, the prompt is reset to it's original value. This must be done, because otherwise the script would wait for another "[confirm]" after executing line 5 and line 6.

A Command Is Sent Too Soon

This happens when a prompt was incorrectly detected in the response of a remote host. For example, consider using the following script:

```
show interface descriptions{extract /^(\S+\d)/ as interfaces}
show diag summary
```

Using this script, the following conversation may take place:

```
1. router> show interface descriptions
2. Interface              Status         Protocol Description
3. Lo0                    up             up       Description of my router>
4. PO0/0                  admin down     down
5. Serial1/0              up             up       My WAN link
6. router>
```

Note that line 3 happens to contain the string "Router>", which looks like a prompt when it really is just a description. So after receiving the ">" character in line 3, Exscript believes that the router is asking for the next command to be sent. So it immediately sends the next command ("show diag summary") to the router, even that the next prompt was not yet received.

Note that this type of error may not immediately show, because the router may actually accept the command even though it was sent before a prompt was sent. It will lead to an offset however, and may lead to errors when trying to capture the response. It may also lead to the script terminating too early.

To fix this, make sure that the conversation with the remote host does not include any strings that are incorrectly recognized as prompts. You can do this by using the "connection.set_prompt(...)" function as explained in the sections above.

The Connection Is Closed Too Soon

This is essentially the same problem as explained under "A Command Is Sent Too Soon". Whenever a prompt is (correctly or incorrectly) detected, the next command is send to the remote host. If all commands were already executed and the next prompt is received (i.e. the end of the script was reached), the connection is closed.

To fix this, make sure that the conversation with the remote host does not include any strings that are incorrectly recognized as prompts. You can do this by using the "connection.set_prompt(...)" function as explained in the sections above.

Exscript

Exscript package

The core module.

class Exscript.**FileLogger**(*logdir*, *mode=u'a'*, *delete=False*, *clearmem=True*)
Bases: *Exscript.logger.Logger*

A Logger that stores logs into files.

__init__(*logdir*, *mode=u'a'*, *delete=False*, *clearmem=True*)
The logdir argument specifies the location where the logs are stored. The mode specifies whether to append the existing logs (if any). If delete is True, the logs are deleted after they are completed, unless they have an error in them. If clearmem is True, the logger does not store a reference to the log in it. If you want to use the functions from *Exscript.util.report* with the logger, clearmem must be False.

add_log(*job_id*, *name*, *attempt*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

class Exscript.**Logger**
Bases: future.types.newobject.newobject

A QueueListener that implements logging for the queue. Logs are kept in memory, and not written to the disk.

__init__()
Creates a new logger instance. Use the *Exscript.util.log.log_to* decorator to send messages to the logger.

add_log(*job_id*, *name*, *attempt*)

get_aborted_actions()
Returns the number of jobs that were aborted.

get_aborted_logs()

get_logs()

get_succeeded_actions()
Returns the number of jobs that were completed successfully.

get_succeeded_logs()

log(*job_id*, *message*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

class Exscript.**Host**(*uri*, *default_protocol='telnet'*)
Bases: future.types.newobject.newobject

Represents a device on which to open a connection.

__init__(*uri*, *default_protocol='telnet'*)
Constructor. The given uri is passed to Host.set_uri(). The default_protocol argument defines the protocol that is used in case the given uri does not specify it.

Parameters

- **uri** (string) – A hostname; see set_uri() for more info.
- **default_protocol** (string) – The protocol name.

account

address

append(*name*, *value*)
Appends the given value to the list variable with the given name.

Parameters

- **name** (*string*) – The name of the variable.
- **value** (*object*) – The appended value.

get(*name*, *default=None*)
Returns the value of the given variable, or the given default value if the variable is not defined.

Parameters

- **name** (*string*) – The name of the variable.
- **default** (*object*) – The default value.

Return type object

Returns The value of the variable.

get_account()
Returns the account that is used to log in.

Return type *Account*

Returns The account.

get_address()
Returns the address that is used to open the connection.

Return type *string*

Returns The address that is used to open the connection.

get_all()
Returns a dictionary containing all variables.

Return type dict

Returns The dictionary with the variables.

get_dict()
Returns a dict containing the host's attributes. The following keys are contained:

•hostname

•address

•protocol

•port

Return type dict

Returns The resulting dictionary.

get_name()
Returns the name.

Return type *string*

Returns The hostname excluding the name.

get_option(*name*, *default=None*)
Returns the value of the given option if it is defined, returns the given default value otherwise.

Parameters

- **name** (*str*) – The option name.

- **default** (*object*) – A default value.

get_options()
Return a dictionary containing all defined options.

Return type dict

Returns The options.

get_protocol()
Returns the name of the protocol.

Return type *string*

Returns The protocol name.

get_tcp_port()
Returns the TCP port number.

Return type *string*

Returns The TCP port number.

get_uri()
Returns a URI formatted representation of the host, including all of it's attributes except for the name. Uses the address, not the name of the host to build the URI.

Return type str

Returns A URI.

has_key(*name*)
Returns True if the variable with the given name is defined, False otherwise.

Parameters **name** (*string*) – The name of the variable.

Return type bool

Returns Whether the variable is defined.

name

options

protocol

set(*name*, *value*)
Stores the given variable/value in the object for later retrieval.

Parameters

- **name** (*string*) – The name of the variable.
- **value** (*object*) – The value of the variable.

set_account(*account*)
Defines the account that is used to log in.

Parameters **account** (*Exscript.Account*) – The account.

set_address(*address*)
Set the address of the remote host the is contacted, without changing hostname, username, password, protocol, and TCP port number. This is the actual address that is used to open the connection.

Parameters **address** (*string*) – A hostname or IP name.

set_all (*variables*)
Like set(), but replaces all variables by using the given dictionary. In other words, passing an empty dictionary results in all variables being removed.

Parameters **variables** (*dict*) – The dictionary with the variables.

set_default (*name*, *value*)
Like set(), but only sets the value if the variable is not already defined.

Parameters

- **name** (string) – The name of the variable.
- **value** (*object*) – The value of the variable.

set_name (*name*)
Set the hostname of the remote host without changing username, password, protocol, and TCP port number.

Parameters **name** (string) – A hostname or IP address.

set_option (*name*, *value*)
Defines a (possibly protocol-specific) option for the host. Possible options include:

verify_fingerprint: bool

Parameters

- **name** (*str*) – The option name.
- **value** (*object*) – The option value.

set_protocol (*protocol*)
Defines the protocol. The following protocols are currently supported:

•telnet: Telnet

•ssh1: SSH version 1

•ssh2: SSH version 2

•ssh: Automatically selects the best supported SSH version

•dummy: A virtual device that accepts any command, but that does not respond anything useful.

•pseudo: A virtual device that loads a file with the given "hostname". The given file is a Python file containing information on how the virtual device shall respond to commands. For more information please refer to the documentation of protocols.Dummy.load_command_handler_from_file().

Parameters **protocol** (string) – The protocol name.

set_tcp_port (*tcp_port*)
Defines the TCP port number.

Parameters **tcp_port** (*int*) – The TCP port number.

set_uri (*uri*)
Defines the protocol, hostname/address, TCP port number, username, and password from the given URL. The hostname may be URL formatted, so the following formats are all valid:

•myhostname

•myhostname.domain

•ssh:hostname

•ssh:hostname.domain

•ssh://hostname

•ssh://user@hostname

•ssh://user:password@hostname

•ssh://user:password@hostname:21

For a list of supported protocols please see set_protocol().

Parameters **uri** (string) – An URL formatted hostname.

tcp_port

vars

class Exscript.**Account**(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Bases: future.types.newobject.newobject

This class represents a user account.

__init__(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Constructor.

The authorization password is only required on hosts that separate the authentication from the authorization procedure. If an authorization password is not given, it defaults to the same value as the authentication password.

If the *needs_lock* argument is set to True, we ensure that no two threads can use the same account at the same time. You will want to use this setting if you are using a central authentication server that allows for only one login to happen at a time. Note that you will still be able to open multiple sessions at the time. It is only the authentication procedure that will not happen in parallel; once the login is complete, other threads can use the account again. In other words, the account is only locked during calls to *protocols.Protocol.login()* and the **authenticate** methods.

Warning: Setting *lock* to True drastically degrades performance!

Parameters

- **name** (*str*) – A username.
- **password** (*str*) – The authentication password.
- **password2** (*str*) – The authorization password, if required.
- **key** (PrivateKey) – A private key, if required.
- **needs_lock** (*bool*) – True if the account will be locked during login.

acquire(*signal=True*)
Locks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the acquired_event signal.

context()
When you need a 'with' context for an already-acquired account.

get_authorization_password()
Returns the authorization password of the account.

Return type *string*

Returns The account password.

get_key()
Returns the key of the account, if any.

Return type PrivateKey|None

Returns A key object.

get_name()
Returns the name of the account.

Return type *string*

Returns The account name.

get_password()
Returns the password of the account.

Return type *string*

Returns The account password.

release(*signal=True*)
Unlocks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the released_event signal.

set_authorization_password(*password*)
Changes the authorization password of the account.

Parameters **password** (string) – The new authorization password.

set_name(*name*)
Changes the name of the account.

Parameters **name** (string) – The account name.

set_password(*password*)
Changes the password of the account.

Parameters **password** (string) – The account password.

class Exscript.**AccountPool**(*accounts=None*)
Bases: future.types.newobject.newobject

This class manages a collection of available accounts.

__init__(*accounts=None*)
Constructor.

Parameters **accounts** (*Account* | *list* [Account]) – Passed to add_account()

acquire_account(*account=None*, *owner=None*)
Waits until an account becomes available, then locks and returns it. If an account is not passed, the next available account is returned.

Parameters

- **account** (Account) – The account to be acquired, or None.
- **owner** (*object*) – An optional descriptor for the owner.

Return type *Account*

Returns The account that was acquired.

add_account(*accounts*)
Adds one or more account instances to the pool.

Parameters **accounts** (*Account | list[Account]*) – The account to be added.

get_account_from_hash(*account_hash*)
Returns the account with the given hash, or None if no such account is included in the account pool.

get_account_from_name(*name*)
Returns the account with the given name.

Parameters **name** (*string*) – The name of the account.

has_account(*account*)
Returns True if the given account exists in the pool, returns False otherwise.

Parameters **account** (*Account*) – The account object.

n_accounts()
Returns the number of accounts that are currently in the pool.

release_accounts(*owner*)
Releases all accounts that were acquired by the given owner.

Parameters **owner** (*object*) – The owner descriptor as passed to acquire_account().

reset()
Removes all accounts.

class Exscript.**Queue**(*domain=''*, *verbose=1*, *mode='threading'*, *max_threads=1*, *host_driver=None*, *exc_cb=None*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *stderr=<open file '<stderr>'*, *mode 'w'>*)
Bases: future.types.newobject.newobject

Manages hosts/tasks, accounts, connections, and threads.

__init__(*domain=''*, *verbose=1*, *mode='threading'*, *max_threads=1*, *host_driver=None*, *exc_cb=None*, *stdout=<open file '<stdout>'*, *mode 'w'>*, *stderr=<open file '<stderr>'*, *mode 'w'>*)
Constructor. All arguments should be passed as keyword arguments. Depending on the verbosity level, the following types of output are written to stdout/stderr (or to whatever else is passed in the stdout/stderr arguments):

- S = status bar
- L = live conversation
- D = debug messages
- E = errors
- ! = errors with tracebacks
- F = fatal errors with tracebacks

The output types are mapped depending on the verbosity as follows:

- verbose = -1: stdout = None, stderr = F
- verbose = 0: stdout = None, stderr = EF
- verbose = 1, max_threads = 1: stdout = L, stderr = EF
- verbose = 1, max_threads = n: stdout = S, stderr = EF
- verbose >= 2, max_threads = 1: stdout = DL, stderr = !F

•verbose >= 2, max_threads = n: stdout = DS, stderr = !F

Parameters

- **domain** (*str*) – The default domain of the contacted hosts.
- **verbose** (*int*) – The verbosity level.
- **mode** (*str*) – 'multiprocessing' or 'threading'
- **max_threads** (*int*) – The maximum number of concurrent threads.
- **host_driver** (*str*) – driver name like "ios" for manual override
- **exc_cb** (*func(jobname, exc_info)*) – callback function to call on exceptions
- **stdout** (file) – The output channel, defaults to sys.stdout.
- **stderr** (file) – The error channel, defaults to sys.stderr.

add_account(*account*)

Adds the given account to the default account pool that Exscript uses to log into all hosts that have no specific *Account* attached.

Parameters **account** (Account) – The account that is added.

add_account_pool(*pool*, *match=None*)

Adds a new account pool. If the given match argument is None, the pool the default pool. Otherwise, the match argument is a callback function that is invoked to decide whether or not the given pool should be used for a host.

When Exscript logs into a host, the account is chosen in the following order:

Exscript checks whether an account was attached to the *Host* object using *Host.set_account()*), and uses that.

If the *Host* has no account attached, Exscript walks through all pools that were passed to *Queue.add_account_pool()*. For each pool, it passes the *Host* to the function in the given match argument. If the return value is True, the account pool is used to acquire an account. (Accounts within each pool are taken in a round-robin fashion.)

If no matching account pool is found, an account is taken from the default account pool.

Finally, if all that fails and the default account pool contains no accounts, an error is raised.

Example usage:

```
def do_nothing(conn):
    conn.autoinit()

def use_this_pool(host):
    return host.get_name().startswith('foo')

default_pool = AccountPool()
default_pool.add_account(Account('default-user', 'password'))

other_pool = AccountPool()
other_pool.add_account(Account('user', 'password'))

queue = Queue()
queue.add_account_pool(default_pool)
queue.add_account_pool(other_pool, use_this_pool)
```

```
host = Host('localhost')
queue.run(host, do_nothing)
```

In the example code, the host has no account attached. As a result, the queue checks whether use_this_pool() returns True. Because the hostname does not start with 'foo', the function returns False, and Exscript takes the 'default-user' account from the default pool.

Parameters

- **pool** (AccountPool) – The account pool that is added.
- **match** (*callable*) – A callback to check if the pool should be used.

destroy (*force=False*)
Like shutdown(), but also removes all accounts, hosts, etc., and does not restart the queue. In other words, the queue can no longer be used after calling this method.

Parameters **force** (*bool*) – Whether to wait until all jobs were processed.

enqueue (*function*, *name=None*, *attempts=1*)
Places the given function in the queue and calls it as soon as a thread is available. To pass additional arguments to the callback, use Python's functools.partial().

Parameters

- **function** (*function*) – The function to execute.
- **name** (string) – A name for the task.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

force_run (*hosts*, *function*, *attempts=1*)
Like priority_run(), but starts the task immediately even if that max_threads is exceeded.

Parameters

- **hosts** (*string*|*list(string)*|*Host*|*list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

get_max_threads ()
Returns the maximum number of concurrent threads.

Return type int

Returns The maximum number of connections.

get_progress ()
Returns the progress in percent.

Return type float

Returns The progress in percent.

is_completed()
Returns True if the task is completed, False otherwise. In other words, this methods returns True if the queue is empty.

Return type bool

Returns Whether all tasks are completed.

join()
Waits until all jobs are completed.

priority_run(*hosts*, *function*, *attempts=1*)
Like run(), but adds the task to the front of the queue.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

priority_run_or_raise(*hosts*, *function*, *attempts=1*)
Like priority_run(), but if a host is already in the queue, the existing host is moved to the top of the queue instead of enqueuing the new one.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns A task object, or None if all hosts were duplicates.

reset()
Remove all accounts, hosts, etc.

run(*hosts*, *function*, *attempts=1*)
Add the given function to a queue, and call it once for each host according to the threading options. Use decorators.bind() if you also want to pass additional arguments to the callback function.

Returns an object that represents the queued task, and that may be passed to is_completed() to check the status.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

run_or_ignore(*hosts*, *function*, *attempts=1*)
Like run(), but only appends hosts that are not already in the queue.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns A task object, or None if all hosts were duplicates.

set_max_threads(*n_connections*)
Sets the maximum number of concurrent connections.

Parameters **n_connections** (*int*) – The maximum number of connections.

shutdown(*force=False*)
Stop executing any further jobs. If the force argument is True, the function does not wait until any queued jobs are completed but stops immediately.

After emptying the queue it is restarted, so you may still call run() after using this method.

Parameters **force** (*bool*) – Whether to wait until all jobs were processed.

class Exscript.**PrivateKey**(*keytype='rsa'*)
Bases: future.types.newobject.newobject

Represents a cryptographic key, and may be used to authenticate useing *Exscript.protocols*.

__init__(*keytype='rsa'*)
Constructor. Supported key types are provided by their respective protocol adapters and can be retrieved from the PrivateKey.keytypes class attribute.

Parameters **keytype** (string) – The key type.

static from_file(*filename*, *password=''*, *keytype=None*)
Returns a new PrivateKey instance with the given attributes. If keytype is None, we attempt to automatically detect the type.

Parameters

- **filename** (string) – The key file name.
- **password** (string) – The key password.
- **keytype** (string) – The key type.

Return type *PrivateKey*

Returns The new key.

get_filename()
Returns the name of the key file.

Return type *string*

Returns The key password.

get_password()
Returns the password for the key.

Return type *string*

Returns The key password.

get_type()
Returns the type of the key, e.g. RSA or DSA.

Return type *string*

Returns The key type

keytypes = set([u'rsa', u'dss'])

set_filename(*filename*)
Sets the name of the key file to use.

Parameters **filename** (string) – The key filename.

set_password(*password*)
Defines the password used for decrypting the key.

Parameters **password** (string) – The key password.

Subpackages

Exscript.emulators package

Emulating a device for testing your scripts.

class Exscript.emulators.**VirtualDevice**(*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)
Bases: future.types.newobject.newobject

An object that emulates a remote device.

LOGIN_TYPE_BOTH = 3

LOGIN_TYPE_NONE = 4

LOGIN_TYPE_PASSWORDONLY = 1

LOGIN_TYPE_USERONLY = 2

PROMPT_STAGE_CUSTOM = 3

PROMPT_STAGE_PASSWORD = 2

PROMPT_STAGE_USERNAME = 1

__init__(*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)

Parameters

- **hostname** (*str*) – The hostname, used for the prompt.
- **echo** – bool
- **echo** – whether to echo the command in a response.
- **login_type** – int
- **login_type** – integer constant, one of LOGIN_TYPE_PASSWORDONLY, LOGIN_TYPE_USERONLY, LOGIN_TYPE_BOTH, LOGIN_TYPE_NONE.
- **strict** – bool
- **strict** – Whether to raise when a given command has no handler.
- **banner** – str

- **banner** – A string to show as soon as the connection is opened.

add_command(*command*, *handler*, *prompt=True*)
Registers a command.

The command may be either a string (which is then automatically compiled into a regular expression), or a pre-compiled regular expression object.

If the given response handler is a string, it is sent as the response to any command that matches the given regular expression. If the given response handler is a function, it is called with the command passed as an argument.

Parameters

- **command** (*str|regex*) – A string or a compiled regular expression.
- **handler** (*function|str*) – A string, or a response handler.
- **prompt** (*bool*) – Whether to show a prompt after completing the command.

add_commands_from_file(*filename*, *autoprompt=True*)
Wrapper around add_command_handler that reads the handlers from the file with the given name. The file is a Python script containing a list named 'commands' of tuples that map command names to handlers.

Parameters

- **filename** (*str*) – The name of the file containing the tuples.
- **autoprompt** (*bool*) – Whether to append a prompt to each response.

do(*command*)
"Executes" the given command on the virtual device, and returns the response.

Parameters **command** (*str*) – The command to be executed.

Return type str

Returns The response of the virtual device.

get_prompt()
Returns the prompt of the device.

Return type str

Returns The current command line prompt.

init()
Init or reset the virtual device.

Return type str

Returns The initial response of the virtual device.

set_prompt(*prompt*)
Change the prompt of the device.

Parameters **prompt** (*str*) – The new command line prompt.

class Exscript.emulators.**IOSEmulator**(*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)
Bases: *Exscript.emulators.vdevice.VirtualDevice*

__init__(*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)

class Exscript.emulators.**CommandSet**(*strict=True*)
Bases: future.types.newobject.newobject

A set of commands to be used by the Dummy adapter.

__init__(*strict=True*)
Constructor.

add(*command*, *response*)
Register a command/response pair.

The command may be either a string (which is then automatically compiled into a regular expression), or a pre-compiled regular expression object.

If the given response handler is a string, it is sent as the response to any command that matches the given regular expression. If the given response handler is a function, it is called with the command passed as an argument.

Parameters

- **command** (*str | regex*) – A string or a compiled regular expression.
- **response** (*function | str*) – A reponse, or a response handler.

add_from_file(*filename*, *handler_decorator=None*)
Wrapper around add() that reads the handlers from the file with the given name. The file is a Python script containing a list named 'commands' of tuples that map command names to handlers.

Parameters

- **filename** (*str*) – The name of the file containing the tuples.
- **handler_decorator** (*function*) – A function that is used to decorate each of the handlers in the file.

eval(*command*)
Evaluate the given string against all registered commands and return the defined response.

Parameters **command** (*str*) – The command that is evaluated.

Return type str or None

Returns The response, if one was defined.

Submodules

Exscript.emulators.command module

Defines the behavior of commands by mapping commands to functions.

class Exscript.emulators.command.**CommandSet**(*strict=True*)
Bases: future.types.newobject.newobject

A set of commands to be used by the Dummy adapter.

__init__(*strict=True*)
Constructor.

add(*command*, *response*)
Register a command/response pair.

The command may be either a string (which is then automatically compiled into a regular expression), or a pre-compiled regular expression object.

If the given response handler is a string, it is sent as the response to any command that matches the given regular expression. If the given response handler is a function, it is called with the command passed as an argument.

Parameters

- **command** (*str | regex*) – A string or a compiled regular expression.
- **response** (*function | str*) – A reponse, or a response handler.

add_from_file (*filename*, *handler_decorator=None*)
Wrapper around add() that reads the handlers from the file with the given name. The file is a Python script containing a list named 'commands' of tuples that map command names to handlers.

Parameters

- **filename** (*str*) – The name of the file containing the tuples.
- **handler_decorator** (*function*) – A function that is used to decorate each of the handlers in the file.

eval (*command*)
Evaluate the given string against all registered commands and return the defined response.

Parameters **command** (*str*) – The command that is evaluated.

Return type str or None

Returns The response, if one was defined.

Exscript.emulators.iosemu module

Cisco IOS emulator.

class Exscript.emulators.iosemu.**IOSEmulator** (*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)
Bases: *Exscript.emulators.vdevice.VirtualDevice*

__init__ (*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)

Exscript.emulators.iosemu.**show_diag** (*data*)

Exscript.emulators.vdevice module

Defines the behavior of a device, as needed by *Exscript.servers*.

class Exscript.emulators.vdevice.**VirtualDevice** (*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)
Bases: future.types.newobject.newobject

An object that emulates a remote device.

LOGIN_TYPE_BOTH = 3

LOGIN_TYPE_NONE = 4

LOGIN_TYPE_PASSWORDONLY = 1

LOGIN_TYPE_USERONLY = 2

PROMPT_STAGE_CUSTOM = 3

PROMPT_STAGE_PASSWORD = 2

PROMPT_STAGE_USERNAME = 1

__init__ (*hostname*, *echo=True*, *login_type=3*, *strict=True*, *banner=None*)

Parameters

- **hostname** (*str*) – The hostname, used for the prompt.
- **echo** – bool
- **echo** – whether to echo the command in a response.
- **login_type** – int
- **login_type** – integer constant, one of LOGIN_TYPE_PASSWORDONLY, LOGIN_TYPE_USERONLY, LOGIN_TYPE_BOTH, LOGIN_TYPE_NONE.
- **strict** – bool
- **strict** – Whether to raise when a given command has no handler.
- **banner** – str
- **banner** – A string to show as soon as the connection is opened.

add_command(*command*, *handler*, *prompt=True*)
Registers a command.

The command may be either a string (which is then automatically compiled into a regular expression), or a pre-compiled regular expression object.

If the given response handler is a string, it is sent as the response to any command that matches the given regular expression. If the given response handler is a function, it is called with the command passed as an argument.

Parameters

- **command** (*str|regex*) – A string or a compiled regular expression.
- **handler** (*function|str*) – A string, or a response handler.
- **prompt** (*bool*) – Whether to show a prompt after completing the command.

add_commands_from_file(*filename*, *autoprompt=True*)
Wrapper around add_command_handler that reads the handlers from the file with the given name. The file is a Python script containing a list named 'commands' of tuples that map command names to handlers.

Parameters

- **filename** (*str*) – The name of the file containing the tuples.
- **autoprompt** (*bool*) – Whether to append a prompt to each response.

do(*command*)
"Executes" the given command on the virtual device, and returns the response.

Parameters **command** (*str*) – The command to be executed.

Return type str

Returns The response of the virtual device.

get_prompt()
Returns the prompt of the device.

Return type str

Returns The current command line prompt.

init()
Init or reset the virtual device.

Return type str

Returns The initial response of the virtual device.

set_prompt(*prompt*)
Change the prompt of the device.

Parameters **prompt** (*str*) – The new command line prompt.

Exscript.protocols package

Exscript.protocols.**connect**(*host*, *default_protocol='telnet'*, ***kwargs*)
Like *prepare()*, but also connects to the host by calling *Protocol.connect()*. If the URL or host contain any login info, this function also logs into the host using *Protocol.login()*.

Parameters

- **host** (*str or Host*) – A URL-formatted hostname or a *Exscript.Host* object.
- **default_protocol** (*str*) – Protocol that is used if the URL specifies none.
- **kwargs** (*dict*) – Passed to the protocol constructor.

Return type *Protocol*

Returns An instance of the protocol.

Exscript.protocols.**prepare**(*host*, *default_protocol='telnet'*, ***kwargs*)
Creates an instance of the protocol by either parsing the given URL-formatted hostname using *Exscript.util.url*, or according to the options of the given *Exscript.Host*.

Parameters

- **host** (*str or Host*) – A URL-formatted hostname or a *Exscript.Host* instance.
- **default_protocol** (*str*) – Protocol that is used if the URL specifies none.
- **kwargs** (*dict*) – Passed to the protocol constructor.

Return type *Protocol*

Returns An instance of the protocol.

Exscript.protocols.**create_protocol**(*name*, ***kwargs*)
Returns an instance of the protocol with the given name.

Parameters **name** (*str*) – The name of the protocol.

Return type *Protocol*

Returns An instance of the protocol.

class Exscript.protocols.**Telnet**(***kwargs*)
Bases: *Exscript.protocols.protocol.Protocol*

The Telnet protocol adapter.

__init__(***kwargs*)

cancel_expect()

close(*force=False*)

interact(*key_handlers=None*, *handle_window_size=True*)

send(*data*)

class Exscript.protocols.**Dummy**(*device=None*, ***kwargs*)
Bases: *Exscript.protocols.protocol.Protocol*

This protocol adapter does not open a network connection, but talks to a *Exscript.emulators.VirtualDevice* internally.

__init__(*device=None*, ***kwargs*)

Hint: Also supports all keyword arguments that *Protocol* supports.

Parameters **device** – The *Exscript.emulators.VirtualDevice* with which to communicate.

cancel_expect()

close(*force=False*)

is_dummy()

send(*data*)

class Exscript.protocols.**Account**(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Bases: future.types.newobject.newobject

This class represents a user account.

__init__(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Constructor.

The authorization password is only required on hosts that separate the authentication from the authorization procedure. If an authorization password is not given, it defaults to the same value as the authentication password.

If the *needs_lock* argument is set to True, we ensure that no two threads can use the same account at the same time. You will want to use this setting if you are using a central authentication server that allows for only one login to happen at a time. Note that you will still be able to open multiple sessions at the time. It is only the authentication procedure that will not happen in parallel; once the login is complete, other threads can use the account again. In other words, the account is only locked during calls to protocols.Protocol.login() and the **authenticate** methods.

Warning: Setting *lock* to True drastically degrades performance!

Parameters

- **name** (*str*) – A username.
- **password** (*str*) – The authentication password.
- **password2** (*str*) – The authorization password, if required.
- **key** (PrivateKey) – A private key, if required.
- **needs_lock** (*bool*) – True if the account will be locked during login.

acquire(*signal=True*)
Locks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the acquired_event signal.

context()
When you need a 'with' context for an already-acquired account.

get_authorization_password()
Returns the authorization password of the account.

Return type *string*

Returns The account password.

get_key()
Returns the key of the account, if any.

Return type PrivateKey|None

Returns A key object.

get_name()
Returns the name of the account.

Return type *string*

Returns The account name.

get_password()
Returns the password of the account.

Return type *string*

Returns The account password.

release(*signal=True*)
Unlocks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the released_event signal.

set_authorization_password(*password*)
Changes the authorization password of the account.

Parameters **password** (string) – The new authorization password.

set_name(*name*)
Changes the name of the account.

Parameters **name** (string) – The account name.

set_password(*password*)
Changes the password of the account.

Parameters **password** (string) – The account password.

class Exscript.protocols.**Url**
Bases: future.types.newobject.newobject

Represents a URL.

__init__()

static from_string(*url*, *default_protocol=u'telnet'*)
Parses the given URL and returns an URL object. There are some differences to Python's built-in URL parser:

- It is less strict, many more inputs are accepted. This is necessary to allow for passing a simple hostname as a URL.

•You may specify a default protocol that is used when the http:// portion is missing.

•The port number defaults to the well-known port of the given protocol.

•The query variables are parsed into a dictionary (Url.vars).

Parameters

- **url** (*str*) – A URL.
- **default_protocol** (*string*) – A protocol name.

Return type *Url*

Returns The Url object contructed from the given URL.

to_string()
Returns the URL, including all attributes, as a string.

Return type str

Returns A URL.

class Exscript.protocols.**SSH2**(***kwargs*)
Bases: *Exscript.protocols.protocol.Protocol*

The secure shell protocol version 2 adapter, based on Paramiko.

KEEPALIVE_INTERVAL = 150.0

__init__(***kwargs*)

cancel_expect()

close(*force=False*)

interact(*key_handlers=None*, *handle_window_size=True*)

send(*data*)

Exscript.protocols.**get_protocol_from_name**(*name*)
Returns the protocol class for the protocol with the given name.

Parameters **name** (*str*) – The name of the protocol.

Return type *Protocol*

Returns The protocol class.

Exscript.protocols.**to_host**(*host*, *default_protocol='telnet'*, *default_domain=''*)
Given a string or a Host object, this function returns a Host object.

Parameters

- **host** (*string|Host*) – A hostname (may be URL formatted) or a Host object.
- **default_protocol** (*str*) – Passed to the Host constructor.
- **default_domain** (*str*) – Appended to each hostname that has no domain.

Return type *Host*

Returns The Host object.

class Exscript.protocols.**Protocol**(*driver=None*, *stdout=None*, *stderr=None*, *debug=0*, *connect_timeout=30*, *timeout=30*, *logfile=None*, *termtype=u'dumb'*, *verify_fingerprint=True*, *account_factory=None*, *banner_timeout=20*, *encoding=u'latin-1'*)

Bases: future.types.newobject.newobject

This is the base class for all protocols; it defines the common portions of the API.

The goal of all protocol classes is to provide an interface that is unified across protocols, such that the adapters may be used interchangeably without changing any other code.

In order to achieve this, the main challenge are the differences arising from the authentication methods that are used. The reason is that many devices may support the following variety authentication/authorization methods:

1. Protocol level authentication, such as SSH's built-in authentication.
   - p1: password only
   - p2: username
   - p3: username + password
   - p4: username + key
   - p5: username + key + password
2. App level authentication, such that the authentication may happen long after a connection is already accepted. This type of authentication is normally used in combination with Telnet, but some SSH hosts also do this (users have reported devices from Enterasys). These devices may also combine protocol-level authentication with app-level authentication. The following types of app-level authentication exist:
   - a1: password only
   - a2: username
   - a3: username + password
3. App level authorization: In order to implement the AAA protocol, some devices ask for two separate app-level logins, whereas the first serves to authenticate the user, and the second serves to authorize him. App-level authorization may support the same methods as app-level authentication:
   - A1: password only
   - A2: username
   - A3: username + password

We are assuming that the following methods are used:

- Telnet:
  - p1 - p5: never
  - a1 - a3: optional
  - A1 - A3: optional
- SSH:
  - p1 - p5: optional
  - a1 - a3: optional
  - A1 - A3: optional

To achieve authentication method compatibility across different protocols, we must hide all this complexity behind one single API call, and figure out which ones are supported.

As a use-case, our goal is that the following code will always work, regardless of which combination of authentication methods a device supports:

```
key = PrivateKey.from_file('~/.ssh/id_rsa', 'my_key_password')

# The user account to use for protocol level authentication.
# The key defaults to None, in which case key authentication is
# not attempted.
account = Account(name     = 'myuser',
                  password = 'mypassword',
                  key      = key)

# The account to use for app-level authentication.
# password2 defaults to password.
app_account = Account(name      = 'myuser',
                      password  = 'my_app_password',
                      password2 = 'my_app_password2')

# app_account defaults to account.
conn.login(account, app_account = None, flush = True)
```

Another important consideration is that once the login is complete, the device must be in a clearly defined state, i.e. we need to have processed the data that was retrieved from the connected host.

More precisely, the buffer that contains the incoming data must be in a state such that the following call to expect_prompt() will either always work, or always fail.

We hide the following methods behind the login() call:

```
# Protocol level authentication.
conn.protocol_authenticate(...)
# App-level authentication.
conn.app_authenticate(...)
# App-level authorization.
conn.app_authorize(...)
```

The code produces the following result:

```
Telnet:
    conn.protocol_authenticate -> NOP
    conn.app_authenticate
        -> waits for username or password prompt, authenticates,
           returns after a CLI prompt was seen.
    conn.app_authorize
        -> calls driver.enable(), waits for username or password
           prompt, authorizes, returns after a CLI prompt was seen.

SSH:
    conn.protocol_authenticate -> authenticates using user/key/password
    conn.app_authenticate -> like Telnet
    conn.app_authorize -> like Telnet
```

We can see the following:

- protocol_authenticate() must not wait for a prompt, because else app_authenticate() has no way of knowing whether an app-level login is even necessary.

- app_authenticate() must check the buffer first, to see if authentication has already succeeded. In the case that app_authenticate() is not necessary (i.e. the buffer contains a CLI prompt), it just returns.

  app_authenticate() must NOT eat the prompt from the buffer, because else the result may be inconsistent with devices that do not do any authentication; i.e., when app_authenticate() is not called.

- Since the prompt must still be contained in the buffer, conn.driver.app_authorize() needs to eat it before it sends the command for starting the authorization procedure.

  This has a drawback - if a user attempts to call app_authorize() at a time where there is no prompt in the buffer, it would fail. So we need to eat the prompt only in cases where we know that auto_app_authorize() will attempt to execute a command. Hence the driver requires the Driver.supports_auto_authorize() method.

  However, app_authorize() must not eat the CLI prompt that follows.

- Once all logins are processed, it makes sense to eat the prompt depending on the wait parameter. Wait should default to True, because it's better that the connection stalls waiting forever, than to risk that an error is not immediately discovered due to timing issues (this is a race condition that I'm not going to detail here).

__init__(*driver=None*, *stdout=None*, *stderr=None*, *debug=0*, *connect_timeout=30*, *timeout=30*, *logfile=None*, *termtype=u'dumb'*, *verify_fingerprint=True*, *account_factory=None*, *banner_timeout=20*, *encoding=u'latin-1'*)

Constructor. The following events are provided:

- data_received_event: A packet was received from the connected host.
- otp_requested_event: The connected host requested a

one-time-password to be entered.

Parameters

- **driver** – Driver()|str
- **stdout** – Where to write the device response. Defaults to an in-memory buffer.
- **stderr** – Where to write debug info. Defaults to stderr.
- **debug** – An integer between 0 (no debugging) and 5 (very verbose debugging) that specifies the amount of debug info sent to the terminal. The default value is 0.
- **connect_timeout** – Timeout for the initial TCP connection attempt
- **timeout** – See set_timeout(). The default value is 30.
- **logfile** – A file into which a log of the conversation with the device is dumped.
- **termtype** – The terminal type to request from the remote host, e.g. 'vt100'.
- **verify_fingerprint** – Whether to verify the host's fingerprint.
- **account_factory** – A function that produces a new *Account*.
- **banner_timeout** (*bool*) – The time to wait for the banner.
- **encoding** (*str*) – The encoding of data received from the remote host.

add_monitor(*pattern*, *callback*, *limit=80*)

Calls the given function whenever the given pattern matches the incoming data.

Hint: If you want to catch all incoming data regardless of a pattern, use the Protocol.data_received_event event instead.

Arguments passed to the callback are the protocol instance, the index of the match, and the match object of the regular expression.

Parameters

- **pattern** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.
- **callback** (*callable*) – The function that is called.
- **limit** (*int*) – The maximum size of the tail of the buffer that is searched, in number of bytes.

app_authenticate(*account=None*, *flush=True*, *bailout=False*)

Attempt to perform application-level authentication. Application level authentication is needed on devices where the username and password are requested from the user after the connection was already accepted by the remote device.

The difference between app-level authentication and protocol-level authentication is that in the latter case, the prompting is handled by the client, whereas app-level authentication is handled by the remote device.

App-level authentication comes in a large variety of forms, and while this method tries hard to support them all, there is no guarantee that it will always work.

We attempt to smartly recognize the user and password prompts; for a list of supported operating systems please check the Exscript.protocols.drivers module.

Returns upon finding the first command line prompt. Depending on whether the flush argument is True, it also removes the prompt from the incoming buffer.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

app_authorize(*account=None*, *flush=True*, *bailout=False*)

Like app_authenticate(), but uses the authorization password of the account.

For the difference between authentication and authorization please google for AAA.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

authenticate(*account=None*, *app_account=None*, *flush=True*)

Like login(), but skips the authorization procedure.

Hint: If you are unsure whether to use *authenticate()* or *login()*, stick with *login*.

Parameters

- **account** (Account) – The account for protocol level authentication.
- **app_account** (Account) – The account for app level authentication.
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.

auto_app_authorize(*account=None*, *flush=True*, *bailout=False*)
Like authorize(), but instead of just waiting for a user or password prompt, it automatically initiates the authorization procedure by sending a driver-specific command.

In the case of devices that understand AAA, that means sending a command to the device. For example, on routers running Cisco IOS, this command executes the 'enable' command before expecting the password.

In the case of a device that is not recognized to support AAA, this method does nothing.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

autoinit()
Make the remote host more script-friendly by automatically executing one or more commands on it. The commands executed depend on the currently used driver. For example, the driver for Cisco IOS would execute the following commands:

```
term len 0
term width 0
```

cancel_expect()
Cancel the current call to *expect()* as soon as control returns to the protocol adapter. This method may be used in callbacks to the events emitted by this class, e.g. Protocol.data_received_event.

close(*force=False*)
Closes the connection with the remote host.

connect(*hostname=None*, *port=None*)
Opens the connection to the remote host or IP address.

Parameters

- **hostname** (string) – The remote host or IP address.
- **port** (*int*) – The remote TCP port number.

execute(*command*, *consume=True*)
Sends the given data to the remote host (with a newline appended) and waits for a prompt in the response. The prompt attempts to use a sane default that works with many devices running Unix, IOS, IOS-XR, or Junos and others. If that fails, a custom prompt may also be defined using the set_prompt() method. This method also modifies the value of the response (self.response) attribute, for details please see the documentation of the expect() method.

Parameters

- **command** (string) – The data that is sent to the remote host.
- **consume** (*boolean (Default: True)*) – Whether to consume the prompt from the buffer or not.

Return type int, re.MatchObject

Returns The index of the prompt regular expression that matched, and the match object.

expect(*prompt*)
Like waitfor(), but also removes the matched string from the buffer containing the incoming data. In other words, the following may not alway complete:

```
conn.expect('myprompt>')
conn.expect('myprompt>') # timeout
```

Returns the index of the regular expression that matched.

Hint: May raise the same exceptions as *waitfor*.

Parameters **prompt** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.

Return type int, re.MatchObject

Returns The index of the regular expression that matched, and the match object.

expect_prompt(*consume=True*)
Monitors the data received from the remote host and waits for a prompt in the response. The prompt attempts to use a sane default that works with many devices running Unix, IOS, IOS-XR, or Junos and others. If that fails, a custom prompt may also be defined using the set_prompt() method. This method also stores the received data in the response attribute (self.response).

Parameters **consume** (*boolean (Default: True)*) – Whether to consume the prompt from the buffer or not.

Return type int, re.MatchObject

Returns The index of the prompt regular expression that matched, and the match object.

get_connect_timeout()
Returns the current connect_timeout in seconds.

Return type int

Returns The connect_timeout in seconds.

get_driver()
Returns the currently used driver.

Return type *Driver*

Returns A regular expression.

get_error_prompt()
Returns the regular expression that is used to monitor the response of the connected host for errors.

Return type regex

Returns A regular expression.

get_host()
Returns the name or address of the currently connected host.

Return type *string*

Returns A name or an address.

get_login_error_prompt()
Returns the regular expression that is used to monitor the response of the connected host for login errors; this is only used during the login procedure, i.e. app_authenticate() or app_authorize().

Return type regex

Returns A regular expression.

get_password_prompt()
Returns the regular expression that is used to monitor the response of the connected host for a username prompt.

Return type regex

Returns A regular expression.

get_prompt()
Returns the regular expressions that is matched against the host response when calling the expect_prompt() method.

Return type *list*(re.RegexObject)

Returns A list of regular expression objects.

get_timeout()
Returns the current timeout in seconds.

Return type int

Returns The timeout in seconds.

get_username_prompt()
Returns the regular expression that is used to monitor the response of the connected host for a username prompt.

Return type regex

Returns A regular expression.

guess_os()
Returns an identifier that specifies the operating system that is running on the remote host. This OS is obtained by watching the response of the remote host, such as any messages retrieved during the login procedure.

The OS is also a wild guess that often depends on volatile information, so there is no guarantee that this will always work.

Return type *string*

Returns A string to help identify the remote operating system.

interact(*key_handlers=None*, *handle_window_size=True*)
Opens a simple interactive shell. Returns when the remote host sends EOF. The optional key handlers are functions that are called whenever the user presses a specific key. For example, to catch CTRL+y:

```
conn.interact({'': mycallback})
```

Parameters

- **key_handlers** (*dict(str: callable)*) – A dictionary mapping chars to a functions.
- **handle_window_size** (*bool*) – Whether the connected host is notified when the terminal size changes.

is_app_authenticated()
Returns True if the application-level authentication procedure was completed, False otherwise.

Return type bool

Returns Whether the authentication was completed.

is_app_authorized()
Returns True if the application-level authorization procedure was completed, False otherwise.

Return type bool

Returns Whether the authorization was completed.

is_dummy()
Returns True if the adapter implements a virtual device, i.e. it isn't an actual network connection.

Return type Boolean

Returns True for dummy adapters, False for network adapters.

is_protocol_authenticated()
Returns True if the protocol-level authentication procedure was completed, False otherwise.

Return type bool

Returns Whether the authentication was completed.

login(*account=None*, *app_account=None*, *flush=True*)
Log into the connected host using the best method available. If an account is not given, default to the account that was used during the last call to login(). If a previous call was not made, use the account that was passed to the constructor. If that also fails, raise a TypeError.

The app_account is passed to *app_authenticate()* and *app_authorize()*. If app_account is not given, default to the value of the account argument.

Parameters

- **account** (Account) – The account for protocol level authentication.
- **app_account** (Account) – The account for app level authentication.
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.

protocol_authenticate(*account=None*)
Low-level API to perform protocol-level authentication on protocols that support it.

Hint: In most cases, you want to use the login() method instead, as it automatically chooses the best login method for each protocol.

Parameters **account** (Account) – An account object, like login().

send(*data*)
Sends the given data to the remote host. Returns without waiting for a response.

Parameters **data** (string) – The data that is sent to the remote host.

Return type Boolean

Returns True on success, False otherwise.

set_connect_timeout(*timeout*)
Defines the maximum time that the adapter waits for initial connection.

Parameters **timeout** (*int*) – The maximum time in seconds.

set_driver(*driver=None*)
Defines the driver that is used to recognize prompts and implement behavior depending on the remote system. The driver argument may be an instance of a protocols.drivers.Driver subclass, a known driver name (string), or None. If the driver argument is None, the adapter automatically chooses a driver using the guess_os() function.

Parameters **driver** (*Driver() | str*) – The pattern that, when matched, causes an error.

set_error_prompt(*error=None*)
Defines a pattern that is used to monitor the response of the connected host. If the pattern matches (any time the expect() or expect_prompt() methods are used), an error is raised.

Parameters **error** (*RegEx*) – The pattern that, when matched, causes an error.

set_login_error_prompt(*error=None*)
Defines a pattern that is used to monitor the response of the connected host during the authentication procedure. If the pattern matches an error is raised.

Parameters **error** (*RegEx*) – The pattern that, when matched, causes an error.

set_password_prompt(*regex=None*)
Defines a pattern that is used to monitor the response of the connected host for a password prompt.

Parameters **regex** (*RegEx*) – The pattern that, when matched, causes an error.

set_prompt(*prompt=None*)
Defines a pattern that is waited for when calling the expect_prompt() method. If the set_prompt() method is not called, or if it is called with the prompt argument set to None, a default prompt is used that should work with many devices running Unix, IOS, IOS-XR, or Junos and others.

Parameters **prompt** (*RegEx*) – The pattern that matches the prompt of the remote host.

set_timeout(*timeout*)
Defines the maximum time that the adapter waits before a call to *expect()* or *expect_prompt()* fails.

Parameters **timeout** (*int*) – The maximum time in seconds.

set_username_prompt(*regex=None*)
Defines a pattern that is used to monitor the response of the connected host for a username prompt.

Parameters **regex** (*RegEx*) – The pattern that, when matched, causes an error.

waitfor(*prompt*)
Monitors the data received from the remote host and waits until the response matches the given prompt. Once a match has been found, the buffer containing incoming data is NOT changed. In other words, consecutive calls to this function will always work, e.g.:

```
conn.waitfor('myprompt>')
conn.waitfor('myprompt>')
conn.waitfor('myprompt>')
```

will always work. Hence in most cases, you probably want to use expect() instead.

This method also stores the received data in the response attribute (self.response).

Returns the index of the regular expression that matched.

Parameters **prompt** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.

Return type int, re.MatchObject

Returns The index of the regular expression that matched, and the match object.

@raise TimeoutException: raised if the timeout was reached. @raise ExpectCancelledException: raised when cancel_expect() was

called in a callback.

@raise ProtocolException: on other internal errors. @raise Exception: May raise other exceptions that are caused

within the underlying protocol implementations.

Subpackages

Exscript.protocols.drivers package

Exscript.protocols.drivers.**add_driver**(*cls*)

Exscript.protocols.drivers.**disable_driver**(*name*)

Exscript.protocols.drivers.**isdriver**(*o*)

Submodules

Exscript.protocols.drivers.ace module

A driver for Cisco Application Control Engine (ACE)

class Exscript.protocols.drivers.ace.**ACEDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.aironet module

A driver for Cisco Aironet Wireless Controllers

class Exscript.protocols.drivers.aironet.**AironetDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.aix module

A driver for AIX.

class Exscript.protocols.drivers.aix.**AIXDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

Exscript.protocols.drivers.arbor_peakflow module

A driver for Peakflow SP by Arbor Networks.

class Exscript.protocols.drivers.arbor_peakflow.**ArborPeakflowDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

Exscript.protocols.drivers.aruba module

A driver for Aruba controllers

class Exscript.protocols.drivers.aruba.**ArubaDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.bigip module

A driver for F5 Big-IP system (TMSH SHELL)

class Exscript.protocols.drivers.bigip.**BigIPDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.brocade module

A driver for Brocade XMR/MLX devices.

class Exscript.protocols.drivers.brocade.**BrocadeDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.cienasaos module

A driver for Ciena SAOS carrier ethernet devices

class Exscript.protocols.drivers.cienasaos.**CienaSAOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.driver module

Base class for all drivers.

class Exscript.protocols.drivers.driver.**Driver**(*name*)
Bases: future.types.newobject.newobject

__init__(*name*)

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

check_response_for_os(*string*)

clean_response_for_re_match(*response*)

init_terminal(*conn*)

supports_auto_authorize()

supports_os_guesser()

Exscript.protocols.drivers.enterasys module

A driver for Enterasys devices.

class Exscript.protocols.drivers.enterasys.**EnterasysDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

Exscript.protocols.drivers.enterasys_wc module

A driver for Enterasys/Extreme (HiPath) Wireless Controller devices.

Created using a C5110 device.

class Exscript.protocols.drivers.enterasys_wc.**EnterasysWCDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.eos module

A driver for Arista EOS.

class Exscript.protocols.drivers.eos.**EOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

init_terminal(*conn*)

Exscript.protocols.drivers.ericsson_ban module

A driver for devices running Ericsson's Broadband Access Nodes (BAN) OS

class Exscript.protocols.drivers.ericsson_ban.**EricssonBanDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

check_response_for_os(*string*)

Exscript.protocols.drivers.fortios module

A driver for FortiOS devices.

Created using a Fortigate device and FortiOS 5.0.

class Exscript.protocols.drivers.fortios.**FortiOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.generic module

The default driver that is used when the OS is not recognized.

class Exscript.protocols.drivers.generic.**GenericDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

Exscript.protocols.drivers.hp_pro_curve module

A driver for HP ProCurve switches.

class Exscript.protocols.drivers.hp_pro_curve.**HPProCurveDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

clean_response_for_re_match(*response*)

init_terminal(*conn*)

Exscript.protocols.drivers.ios module

A driver for Cisco IOS (not IOS XR).

class Exscript.protocols.drivers.ios.**IOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.ios_xr module

A driver for Cisco IOS XR.

class Exscript.protocols.drivers.ios_xr.**IOSXRDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_response_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.isam module

A driver for devices running ISAM (runs on Alcatel ISAM).

class Exscript.protocols.drivers.isam.**IsamDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

check_response_for_os(*string*)

Exscript.protocols.drivers.junos module

A driver for devices running JunOS (by Juniper).

class Exscript.protocols.drivers.junos.**JunOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.junos_erx module

A driver for devices running Juniper ERX OS.

class Exscript.protocols.drivers.junos_erx.**JunOSERXDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.mrv module

A driver for MRV Linux.

class Exscript.protocols.drivers.mrv.**MRVDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.nxos module

A driver for Cisco Nexus OS (NXOS)

class Exscript.protocols.drivers.nxos.**NXOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.one_os module

A driver for OneOS (OneAccess).

class Exscript.protocols.drivers.one_os.**OneOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.rios module

A driver for Riverbed RIOS

class Exscript.protocols.drivers.rios.**RIOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

init_terminal(*conn*)

Exscript.protocols.drivers.shell module

A generic shell driver that handles unknown unix shells.

class Exscript.protocols.drivers.shell.**ShellDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

Exscript.protocols.drivers.smart_edge_os module

A driver for Redback Smart Edge OS.

class Exscript.protocols.drivers.smart_edge_os.**SmartEdgeOSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.sros module

A driver for Alcatel SROS.

class Exscript.protocols.drivers.sros.**SROSDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.vrp module

A driver for devices running VRP (by Huawei).

class Exscript.protocols.drivers.vrp.**VRPDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

check_head_for_os(*string*)

init_terminal(*conn*)

Exscript.protocols.drivers.vxworks module

A driver for devices running Vxworks, which can be found on huawei5600T

class Exscript.protocols.drivers.vxworks.**VxworksDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

check_response_for_os(*string*)

Exscript.protocols.drivers.zte module

A driver for devices running Zte operating system.

class Exscript.protocols.drivers.zte.**ZteDriver**
Bases: *Exscript.protocols.drivers.driver.Driver*

__init__()

auto_authorize(*conn*, *account*, *flush*, *bailout*)

check_head_for_os(*string*)

Submodules

Exscript.protocols.dummy module

A client that talks to a *Exscript.emulators.VirtualDevice*.

class Exscript.protocols.dummy.**Dummy**(*device=None*, ***kwargs*)
Bases: *Exscript.protocols.protocol.Protocol*

This protocol adapter does not open a network connection, but talks to a *Exscript.emulators.VirtualDevice* internally.

__init__(*device=None*, ***kwargs*)

Hint: Also supports all keyword arguments that `Protocol` supports.

Parameters **device** – The *Exscript.emulators.VirtualDevice* with which to communicate.

cancel_expect()

close(*force=False*)

is_dummy()

send(*data*)

Exscript.protocols.exception module

Network related error types.

exception Exscript.protocols.exception.**DriverReplacedException**
Bases: *Exscript.protocols.exception.ProtocolException*

An exception that is thrown if the protocol driver was switched during a call to expect().

exception Exscript.protocols.exception.**ExpectCancelledException**
Bases: *Exscript.protocols.exception.ProtocolException*

An exception that is thrown if Protocol.cancel_expect() was called.

exception Exscript.protocols.exception.**InvalidCommandException**
Bases: *Exscript.protocols.exception.ProtocolException*

An exception that is thrown if the response of a connected host contained a string that looked like an error.

exception Exscript.protocols.exception.**LoginFailure**
Bases: *Exscript.protocols.exception.ProtocolException*

An exception that is thrown if the response of a connected host looked like it was trying to signal a login error during the authentication procedure.

exception Exscript.protocols.exception.**ProtocolException**
Bases: exceptions.Exception

Default exception that is thrown on most protocol related errors.

exception Exscript.protocols.exception.**TimeoutException**
Bases: *Exscript.protocols.exception.ProtocolException*

An exception that is thrown if the connected host did not respond for too long.

Exscript.protocols.osguesser module

class Exscript.protocols.osguesser.**OsGuesser**
Bases: future.types.newobject.newobject

The OsGuesser monitors everything that happens on a Protocol, and attempts to collect data out of the network activity. It watches for specific patterns in the network traffic to decide what operating system a connected host is running. It is completely passive, and attempts no changes on the protocol adapter. However, the protocol adapter may request information from the OsGuesser, and perform changes based on the information provided.

__init__()

data_received(*data*, *app_authentication_done*)

get(*key*, *confidence=0*)
Returns the info with the given key, if it has at least the given confidence. Returns None otherwise.

reset(*auth_buffer=''*)

set(*key*, *value*, *confidence=100*)
Defines the given value with the given confidence, unless the same value is already defined with a higher confidence level.

set_from_match(*key*, *regex_list*, *string*)

Given a list of functions or three-tuples (regex, value, confidence), this function walks through them and checks whether any of the items in the list matches the given string. If the list item is a function, it must have the following signature:

```
func(string) : (string, int)
```

Where the return value specifies the resulting value and the confidence of the match. If a match is found, and the confidence level is higher than the currently defined one, the given value is defined with the given confidence.

Exscript.protocols.protocol module

An abstract base class for all protocols.

class Exscript.protocols.protocol.**Protocol**(*driver=None*, *stdout=None*, *stderr=None*, *debug=0*, *connect_timeout=30*, *timeout=30*, *logfile=None*, *termtype=u'dumb'*, *verify_fingerprint=True*, *account_factory=None*, *banner_timeout=20*, *encoding=u'latin-1'*)

Bases: future.types.newobject.newobject

This is the base class for all protocols; it defines the common portions of the API.

The goal of all protocol classes is to provide an interface that is unified across protocols, such that the adapters may be used interchangeably without changing any other code.

In order to achieve this, the main challenge are the differences arising from the authentication methods that are used. The reason is that many devices may support the following variety authentication/authorization methods:

1. Protocol level authentication, such as SSH's built-in authentication.
   - p1: password only
   - p2: username
   - p3: username + password
   - p4: username + key
   - p5: username + key + password
2. App level authentication, such that the authentication may happen long after a connection is already accepted. This type of authentication is normally used in combination with Telnet, but some SSH hosts also do this (users have reported devices from Enterasys). These devices may also combine protocol-level authentication with app-level authentication. The following types of app-level authentication exist:
   - a1: password only
   - a2: username
   - a3: username + password
3. App level authorization: In order to implement the AAA protocol, some devices ask for two separate app-level logins, whereas the first serves to authenticate the user, and the second serves to authorize him. App-level authorization may support the same methods as app-level authentication:
   - A1: password only
   - A2: username
   - A3: username + password

We are assuming that the following methods are used:

- Telnet:
  - –p1 - p5: never
  - –a1 - a3: optional
  - –A1 - A3: optional
- SSH:
  - –p1 - p5: optional
  - –a1 - a3: optional
  - –A1 - A3: optional

To achieve authentication method compatibility across different protocols, we must hide all this complexity behind one single API call, and figure out which ones are supported.

As a use-case, our goal is that the following code will always work, regardless of which combination of authentication methods a device supports:

```
key = PrivateKey.from_file('~/.ssh/id_rsa', 'my_key_password')

# The user account to use for protocol level authentication.
# The key defaults to None, in which case key authentication is
# not attempted.
account = Account(name     = 'myuser',
                  password = 'mypassword',
                  key      = key)

# The account to use for app-level authentication.
# password2 defaults to password.
app_account = Account(name      = 'myuser',
                      password  = 'my_app_password',
                      password2 = 'my_app_password2')

# app_account defaults to account.
conn.login(account, app_account = None, flush = True)
```

Another important consideration is that once the login is complete, the device must be in a clearly defined state, i.e. we need to have processed the data that was retrieved from the connected host.

More precisely, the buffer that contains the incoming data must be in a state such that the following call to expect_prompt() will either always work, or always fail.

We hide the following methods behind the login() call:

```
# Protocol level authentication.
conn.protocol_authenticate(...)
# App-level authentication.
conn.app_authenticate(...)
# App-level authorization.
conn.app_authorize(...)
```

The code produces the following result:

```
Telnet:
    conn.protocol_authenticate -> NOP
    conn.app_authenticate
        -> waits for username or password prompt, authenticates,
```

```
            returns after a CLI prompt was seen.
    conn.app_authorize
        -> calls driver.enable(), waits for username or password
           prompt, authorizes, returns after a CLI prompt was seen.

SSH:
    conn.protocol_authenticate -> authenticates using user/key/password
    conn.app_authenticate -> like Telnet
    conn.app_authorize -> like Telnet
```

We can see the following:

- protocol_authenticate() must not wait for a prompt, because else app_authenticate() has no way of knowing whether an app-level login is even necessary.
- app_authenticate() must check the buffer first, to see if authentication has already succeeded. In the case that app_authenticate() is not necessary (i.e. the buffer contains a CLI prompt), it just returns.

  app_authenticate() must NOT eat the prompt from the buffer, because else the result may be inconsistent with devices that do not do any authentication; i.e., when app_authenticate() is not called.
- Since the prompt must still be contained in the buffer, conn.driver.app_authorize() needs to eat it before it sends the command for starting the authorization procedure.

  This has a drawback - if a user attempts to call app_authorize() at a time where there is no prompt in the buffer, it would fail. So we need to eat the prompt only in cases where we know that auto_app_authorize() will attempt to execute a command. Hence the driver requires the Driver.supports_auto_authorize() method.

  However, app_authorize() must not eat the CLI prompt that follows.
- Once all logins are processed, it makes sense to eat the prompt depending on the wait parameter. Wait should default to True, because it's better that the connection stalls waiting forever, than to risk that an error is not immediately discovered due to timing issues (this is a race condition that I'm not going to detail here).

__init__(*driver=None*, *stdout=None*, *stderr=None*, *debug=0*, *connect_timeout=30*, *timeout=30*, *logfile=None*, *termtype=u'dumb'*, *verify_fingerprint=True*, *account_factory=None*, *banner_timeout=20*, *encoding=u'latin-1'*)

Constructor. The following events are provided:

- data_received_event: A packet was received from the connected host.
- otp_requested_event: The connected host requested a

one-time-password to be entered.

Parameters

- **driver** – Driver()|str
- **stdout** – Where to write the device response. Defaults to an in-memory buffer.
- **stderr** – Where to write debug info. Defaults to stderr.
- **debug** – An integer between 0 (no debugging) and 5 (very verbose debugging) that specifies the amount of debug info sent to the terminal. The default value is 0.
- **connect_timeout** – Timeout for the initial TCP connection attempt
- **timeout** – See set_timeout(). The default value is 30.
- **logfile** – A file into which a log of the conversation with the device is dumped.

- **termtype** – The terminal type to request from the remote host, e.g. 'vt100'.
- **verify_fingerprint** – Whether to verify the host's fingerprint.
- **account_factory** – A function that produces a new Account.
- **banner_timeout** (*bool*) – The time to wait for the banner.
- **encoding** (*str*) – The encoding of data received from the remote host.

add_monitor(*pattern*, *callback*, *limit=80*)
Calls the given function whenever the given pattern matches the incoming data.

Hint: If you want to catch all incoming data regardless of a pattern, use the Protocol.data_received_event event instead.

Arguments passed to the callback are the protocol instance, the index of the match, and the match object of the regular expression.

Parameters

- **pattern** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.
- **callback** (*callable*) – The function that is called.
- **limit** (*int*) – The maximum size of the tail of the buffer that is searched, in number of bytes.

app_authenticate(*account=None*, *flush=True*, *bailout=False*)
Attempt to perform application-level authentication. Application level authentication is needed on devices where the username and password are requested from the user after the connection was already accepted by the remote device.

The difference between app-level authentication and protocol-level authentication is that in the latter case, the prompting is handled by the client, whereas app-level authentication is handled by the remote device.

App-level authentication comes in a large variety of forms, and while this method tries hard to support them all, there is no guarantee that it will always work.

We attempt to smartly recognize the user and password prompts; for a list of supported operating systems please check the Exscript.protocols.drivers module.

Returns upon finding the first command line prompt. Depending on whether the flush argument is True, it also removes the prompt from the incoming buffer.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

app_authorize(*account=None*, *flush=True*, *bailout=False*)
Like app_authenticate(), but uses the authorization password of the account.

For the difference between authentication and authorization please google for AAA.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.

- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

authenticate(*account=None*, *app_account=None*, *flush=True*)
Like login(), but skips the authorization procedure.

Hint: If you are unsure whether to use *authenticate()* or *login()*, stick with *login*.

Parameters

- **account** (Account) – The account for protocol level authentication.
- **app_account** (Account) – The account for app level authentication.
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.

auto_app_authorize(*account=None*, *flush=True*, *bailout=False*)
Like authorize(), but instead of just waiting for a user or password prompt, it automatically initiates the authorization procedure by sending a driver-specific command.

In the case of devices that understand AAA, that means sending a command to the device. For example, on routers running Cisco IOS, this command executes the 'enable' command before expecting the password.

In the case of a device that is not recognized to support AAA, this method does nothing.

Parameters

- **account** (Account) – An account object, like login().
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **bailout** (*bool*) – Whether to wait for a prompt after sending the password.

autoinit()
Make the remote host more script-friendly by automatically executing one or more commands on it. The commands executed depend on the currently used driver. For example, the driver for Cisco IOS would execute the following commands:

```
term len 0
term width 0
```

cancel_expect()
Cancel the current call to *expect()* as soon as control returns to the protocol adapter. This method may be used in callbacks to the events emitted by this class, e.g. Protocol.data_received_event.

close(*force=False*)
Closes the connection with the remote host.

connect(*hostname=None*, *port=None*)
Opens the connection to the remote host or IP address.

Parameters

- **hostname** (string) – The remote host or IP address.
- **port** (*int*) – The remote TCP port number.

execute(*command*, *consume=True*)
Sends the given data to the remote host (with a newline appended) and waits for a prompt in the response. The prompt attempts to use a sane default that works with many devices running Unix, IOS, IOS-XR, or Junos and others. If that fails, a custom prompt may also be defined using the set_prompt() method.

This method also modifies the value of the response (self.response) attribute, for details please see the documentation of the expect() method.

Parameters

- **command** (*string*) – The data that is sent to the remote host.
- **consume** (*boolean (Default: True)*) – Whether to consume the prompt from the buffer or not.

Return type int, re.MatchObject

Returns The index of the prompt regular expression that matched, and the match object.

expect(*prompt*)

Like waitfor(), but also removes the matched string from the buffer containing the incoming data. In other words, the following may not alway complete:

```
conn.expect('myprompt>')
conn.expect('myprompt>') # timeout
```

Returns the index of the regular expression that matched.

Hint: May raise the same exceptions as *waitfor*.

Parameters **prompt** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.

Return type int, re.MatchObject

Returns The index of the regular expression that matched, and the match object.

expect_prompt(*consume=True*)

Monitors the data received from the remote host and waits for a prompt in the response. The prompt attempts to use a sane default that works with many devices running Unix, IOS, IOS-XR, or Junos and others. If that fails, a custom prompt may also be defined using the set_prompt() method. This method also stores the received data in the response attribute (self.response).

Parameters **consume** (*boolean (Default: True)*) – Whether to consume the prompt from the buffer or not.

Return type int, re.MatchObject

Returns The index of the prompt regular expression that matched, and the match object.

get_connect_timeout()

Returns the current connect_timeout in seconds.

Return type int

Returns The connect_timeout in seconds.

get_driver()

Returns the currently used driver.

Return type *Driver*

Returns A regular expression.

get_error_prompt()

Returns the regular expression that is used to monitor the response of the connected host for errors.

Return type regex

Returns A regular expression.

get_host()
Returns the name or address of the currently connected host.

Return type *string*

Returns A name or an address.

get_login_error_prompt()
Returns the regular expression that is used to monitor the response of the connected host for login errors; this is only used during the login procedure, i.e. app_authenticate() or app_authorize().

Return type regex

Returns A regular expression.

get_password_prompt()
Returns the regular expression that is used to monitor the response of the connected host for a username prompt.

Return type regex

Returns A regular expression.

get_prompt()
Returns the regular expressions that is matched against the host response when calling the expect_prompt() method.

Return type *list*(re.RegexObject)

Returns A list of regular expression objects.

get_timeout()
Returns the current timeout in seconds.

Return type int

Returns The timeout in seconds.

get_username_prompt()
Returns the regular expression that is used to monitor the response of the connected host for a username prompt.

Return type regex

Returns A regular expression.

guess_os()
Returns an identifier that specifies the operating system that is running on the remote host. This OS is obtained by watching the response of the remote host, such as any messages retrieved during the login procedure.

The OS is also a wild guess that often depends on volatile information, so there is no guarantee that this will always work.

Return type *string*

Returns A string to help identify the remote operating system.

interact(*key_handlers=None*, *handle_window_size=True*)
Opens a simple interactive shell. Returns when the remote host sends EOF. The optional key handlers are functions that are called whenever the user presses a specific key. For example, to catch CTRL+y:

```
conn.interact({'': mycallback})
```

Parameters

- **key_handlers** (*dict(str: callable)*) – A dictionary mapping chars to a functions.
- **handle_window_size** (*bool*) – Whether the connected host is notified when the terminal size changes.

is_app_authenticated()
Returns True if the application-level authentication procedure was completed, False otherwise.

Return type bool

Returns Whether the authentication was completed.

is_app_authorized()
Returns True if the application-level authorization procedure was completed, False otherwise.

Return type bool

Returns Whether the authorization was completed.

is_dummy()
Returns True if the adapter implements a virtual device, i.e. it isn't an actual network connection.

Return type Boolean

Returns True for dummy adapters, False for network adapters.

is_protocol_authenticated()
Returns True if the protocol-level authentication procedure was completed, False otherwise.

Return type bool

Returns Whether the authentication was completed.

login(*account=None*, *app_account=None*, *flush=True*)
Log into the connected host using the best method available. If an account is not given, default to the account that was used during the last call to login(). If a previous call was not made, use the account that was passed to the constructor. If that also fails, raise a TypeError.

The app_account is passed to *app_authenticate()* and *app_authorize()*. If app_account is not given, default to the value of the account argument.

Parameters

- **account** (Account) – The account for protocol level authentication.
- **app_account** (Account) – The account for app level authentication.
- **flush** (*bool*) – Whether to flush the last prompt from the buffer.

protocol_authenticate(*account=None*)
Low-level API to perform protocol-level authentication on protocols that support it.

Hint: In most cases, you want to use the login() method instead, as it automatically chooses the best login method for each protocol.

Parameters **account** (Account) – An account object, like login().

send(*data*)
Sends the given data to the remote host. Returns without waiting for a response.

Parameters **data** (*string*) – The data that is sent to the remote host.

Return type Boolean

Returns True on success, False otherwise.

set_connect_timeout(*timeout*)
Defines the maximum time that the adapter waits for initial connection.

Parameters **timeout** (*int*) – The maximum time in seconds.

set_driver(*driver=None*)
Defines the driver that is used to recognize prompts and implement behavior depending on the remote system. The driver argument may be an instance of a protocols.drivers.Driver subclass, a known driver name (string), or None. If the driver argument is None, the adapter automatically chooses a driver using the guess_os() function.

Parameters **driver** (*Driver() | str*) – The pattern that, when matched, causes an error.

set_error_prompt(*error=None*)
Defines a pattern that is used to monitor the response of the connected host. If the pattern matches (any time the expect() or expect_prompt() methods are used), an error is raised.

Parameters **error** (*RegEx*) – The pattern that, when matched, causes an error.

set_login_error_prompt(*error=None*)
Defines a pattern that is used to monitor the response of the connected host during the authentication procedure. If the pattern matches an error is raised.

Parameters **error** (*RegEx*) – The pattern that, when matched, causes an error.

set_password_prompt(*regex=None*)
Defines a pattern that is used to monitor the response of the connected host for a password prompt.

Parameters **regex** (*RegEx*) – The pattern that, when matched, causes an error.

set_prompt(*prompt=None*)
Defines a pattern that is waited for when calling the expect_prompt() method. If the set_prompt() method is not called, or if it is called with the prompt argument set to None, a default prompt is used that should work with many devices running Unix, IOS, IOS-XR, or Junos and others.

Parameters **prompt** (*RegEx*) – The pattern that matches the prompt of the remote host.

set_timeout(*timeout*)
Defines the maximum time that the adapter waits before a call to *expect()* or *expect_prompt()* fails.

Parameters **timeout** (*int*) – The maximum time in seconds.

set_username_prompt(*regex=None*)
Defines a pattern that is used to monitor the response of the connected host for a username prompt.

Parameters **regex** (*RegEx*) – The pattern that, when matched, causes an error.

waitfor(*prompt*)
Monitors the data received from the remote host and waits until the response matches the given prompt. Once a match has been found, the buffer containing incoming data is NOT changed. In other words, consecutive calls to this function will always work, e.g.:

```
conn.waitfor('myprompt>')
conn.waitfor('myprompt>')
conn.waitfor('myprompt>')
```

will always work. Hence in most cases, you probably want to use expect() instead.

This method also stores the received data in the response attribute (self.response).

Returns the index of the regular expression that matched.

Parameters **prompt** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.

Return type int, re.MatchObject

Returns The index of the regular expression that matched, and the match object.

@raise TimeoutException: raised if the timeout was reached. @raise ExpectCancelledException: raised when cancel_expect() was

called in a callback.

@raise ProtocolException: on other internal errors. @raise Exception: May raise other exceptions that are caused

within the underlying protocol implementations.

Exscript.protocols.ssh2 module

SSH version 2 support, based on paramiko.

class Exscript.protocols.ssh2.**SSH2**(***kwargs*)

Bases: *Exscript.protocols.protocol.Protocol*

The secure shell protocol version 2 adapter, based on Paramiko.

KEEPALIVE_INTERVAL = 150.0

__init__(***kwargs*)

cancel_expect()

close(*force=False*)

interact(*key_handlers=None*, *handle_window_size=True*)

send(*data*)

Exscript.protocols.telnet module

The Telnet protocol.

class Exscript.protocols.telnet.**Telnet**(***kwargs*)

Bases: *Exscript.protocols.protocol.Protocol*

The Telnet protocol adapter.

__init__(***kwargs*)

cancel_expect()

close(*force=False*)

interact(*key_handlers=None*, *handle_window_size=True*)

send(*data*)

Exscript.protocols.telnetlib module

TELNET client class.

Based on RFC 854: TELNET Protocol Specification, by J. Postel and J. Reynolds

Example:

```
>>> from telnetlib import Telnet
>>> tn = Telnet('www.python.org', 79)   # connect to finger port
>>> tn.write('guido
')
>>> print tn.read_all()
Login       Name               TTY         Idle    When    Where
guido    Guido van Rossum      pts/2        <Dec  2 11:10> snag.cnri.reston..
```

```
>>>
```

Note that read_all() won't read until eof – it just reads some data – but it guarantees to read at least one byte unless EOF is hit.

It is possible to pass a Telnet object to select.select() in order to wait until more data is available. Note that in this case, read_eager() may return '' even if there was data on the socket, because the protocol negotiation may have eaten the data. This is why EOFError is needed in some cases to distinguish between "no data" and "connection closed" (since the socket also appears ready for reading when it is closed).

Bugs: - may hang when connection is slow in the middle of an IAC sequence

To do: - option negotiation - timeout should be intrinsic to the connection object instead of an

option on one of the read calls only

class Exscript.protocols.telnetlib.**Telnet**(*host=None*, *port=0*, *encoding='latin1'*, ***kwargs*)

Bases: future.types.newobject.newobject

Telnet interface class.

An instance of this class represents a connection to a telnet server. The instance is initially not connected; the open() method must be used to establish a connection. Alternatively, the host name and optional port number can be passed to the constructor, too.

Don't try to reopen an already connected instance.

This class has many read_*() methods. Note that some of them raise EOFError when the end of the connection is read, because they can return an empty string for other reasons. See the individual doc strings.

read_all() Read all data until EOF; may block.

read_some() Read at least one byte or EOF; may block.

read_very_eager() Read all data available already queued or on the socket, without blocking.

read_eager() Read either data already queued or some data available on the socket, without blocking.

read_lazy() Read all data in the raw queue (processing it first), without doing any socket I/O.

read_very_lazy() Reads all data in the cooked queue, without doing any socket I/O.

__init__(*host=None*, *port=0*, *encoding='latin1'*, ***kwargs*)
Constructor.

When called without arguments, create an unconnected instance. With a hostname argument, it connects the instance; a port number is optional.

close()
Close the connection.

expect(*relist*, *timeout=None*, *cleanup=None*)
Like waitfor(), but removes the matched data from the incoming buffer.

fileno()
Return the fileno() of the socket object used internally.

fill_rawq()
Fill raw queue from exactly one recv() system call.

Block if no data is immediately available. Set self.eof when connection is closed.

get_socket()
Return the socket object used internally.

interact()
Interaction function, emulates a very dumb telnet client.

listener()
Helper for mt_interact() – this executes in the other thread.

msg(*msg*, **args*)
Print a debug message, when the debug level is > 0.

If extra arguments are present, they are substituted in the message using the standard string formatting operator.

mt_interact()
Multithreaded version of interact().

open(*host*, *port=0*)
Connect to a host.

The optional second argument is the port number, which defaults to the standard telnet port (23).

Don't try to reopen an already connected instance.

process_rawq()
Transfer from raw queue to cooked queue.

Set self.eof when connection is closed. Don't block unless in the midst of an IAC sequence.

rawq_getchar()
Get next char from raw queue.

Block if no data is immediately available. Raise EOFError when connection is closed.

read_all()
Read all data until EOF; block until connection closed.

read_eager()
Read readily available data.

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_lazy()
Process and return data that's already in the queues (lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_some()
Read at least one byte of cooked data unless EOF is hit.

Return '' if EOF is hit. Block if no data is immediately available.

read_very_eager()
Read everything that's possible without blocking in I/O (eager).

Raise EOFError if connection closed and no cooked data available. Return '' if no cooked data available otherwise. Don't block unless in the midst of an IAC sequence.

read_very_lazy()
Return any data available in the cooked queue (very lazy).

Raise EOFError if connection closed and no data available. Return '' if no cooked data available otherwise. Don't block.

set_debuglevel(*debuglevel*)
Set the debug level.

The higher it is, the more debug output you get (on stdout).

set_receive_callback(*callback*, **args*, ***kwargs*)
The callback function called after each receipt of any data.

set_window_size(*rows*, *cols*)
Change the size of the terminal window, if the remote end supports NAWS. If it doesn't, the method returns silently.

sock_avail()
Test whether data is available on the socket.

waitfor(*relist*, *timeout=None*, *cleanup=None*)
Read until one from a list of a regular expressions matches.

The first argument is a list of regular expressions, either compiled (re.RegexObject instances) or uncompiled (strings). The optional second argument is a timeout, in seconds; default is no timeout.

Return a tuple of three items: the index in the list of the first regular expression that matches; the match object returned; and the text read up till and including the match.

If EOF is read and no text was read, raise EOFError. Otherwise, when nothing matches, return (-1, None, text) where text is the text received so far (may be the empty string if a timeout happened).

If a regular expression ends with a greedy match (e.g. '.*') or if more than one expression can match the same input, the results are undeterministic, and may depend on the I/O timing.

write(*buffer*)
Write a string to the socket, doubling any IAC characters.

Can block if the connection is blocked. May raise socket.error if the connection is closed.

Exscript.servers package

Very simple servers, useful for emulating a device for testing.

class Exscript.servers.**HTTPd**(*addr*, *handler_cls*, *user_data=None*)
Bases: SocketServer.ThreadingMixIn, BaseHTTPServer.HTTPServer

An HTTP server, derived from Python's HTTPServer but with added support for HTTP/Digest. Usage:

```
from Exscript.servers import HTTPd, RequestHandler
class MyHandler(RequestHandler):
    def handle_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write('You opened ' + self.path)

server = HTTPd(('', 8080), MyHandler)
server.add_account('testuser', 'testpassword')
print('started httpserver...')
server.serve_forever()
```

__init__(*addr*, *handler_cls*, *user_data=None*)
Constructor.

Parameters

- **address** (*(str, int)*) – The address and port number on which to bind.
- **handler_cls** (RequestHandler) – The RequestHandler to use.
- **user_data** (*object*) – Optional data that, stored in self.user_data.

add_account(*username*, *password*)
Adds a username/password pair that HTTP clients may use to log in.

Parameters

- **username** (*str*) – The name of the user.
- **password** (*str*) – The user's password.

daemon_threads = True

get_password(*username*)
Returns the password of the user with the given name.

Parameters **username** (*str*) – The name of the user.

class Exscript.servers.**SSHd**(*host*, *port*, *device*, *key=None*)
Bases: *Exscript.servers.server.Server*

A SSH2 server. Usage:

```
device = VirtualDevice('myhost')
daemon = SSHd('localhost', 1234, device)
device.add_command('ls', 'ok', prompt = True)
device.add_command('exit', daemon.exit_command)
daemon.start() # Start the server.
daemon.exit()  # Stop the server.
daemon.join()  # Wait until it terminates.
```

Parameters **key** – An Exscript.PrivateKey object.

__init__(*host*, *port*, *device*, *key=None*)

class Exscript.servers.**Telnetd**(*host*, *port*, *device*, *encoding='utf8'*)
Bases: *Exscript.servers.server.Server*

A Telnet server. Usage:

```
device = VirtualDevice('myhost')
daemon = Telnetd('localhost', 1234, device)
device.add_command('ls', 'ok', prompt = True)
device.add_command('exit', daemon.exit_command)
daemon.start() # Start the server.
daemon.exit()  # Stop the server.
daemon.join()  # Wait until it terminates.
```

Submodules

Exscript.servers.httpd module

A threaded HTTP server with support for HTTP/Digest authentication.

class Exscript.servers.httpd.**HTTPd**(*addr*, *handler_cls*, *user_data=None*)
Bases: SocketServer.ThreadingMixIn, BaseHTTPServer.HTTPServer

An HTTP server, derived from Python's HTTPServer but with added support for HTTP/Digest. Usage:

```
from Exscript.servers import HTTPd, RequestHandler
class MyHandler(RequestHandler):
    def handle_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write('You opened ' + self.path)

server = HTTPd(('', 8080), MyHandler)
server.add_account('testuser', 'testpassword')
print('started httpserver...')
server.serve_forever()
```

__init__(*addr*, *handler_cls*, *user_data=None*)
Constructor.

Parameters

- **address** (*(str, int)*) – The address and port number on which to bind.
- **handler_cls** (*RequestHandler*) – The RequestHandler to use.
- **user_data** (*object*) – Optional data that, stored in self.user_data.

add_account(*username*, *password*)
Adds a username/password pair that HTTP clients may use to log in.

Parameters

- **username** (*str*) – The name of the user.
- **password** (*str*) – The user's password.

daemon_threads = True

get_password(*username*)
Returns the password of the user with the given name.

Parameters **username** (*str*) – The name of the user.

class Exscript.servers.httpd.**RequestHandler**(*request*, *client_address*, *server*)
Bases: BaseHTTPServer.BaseHTTPRequestHandler

A drop-in replacement for Python's BaseHTTPRequestHandler that handles HTTP/Digest.

do_GET()

do_POST()

handle_GET()
Overwrite this method to handle a GET request. The default action is to respond with "error 404 (not found)".

handle_POST()
Overwrite this method to handle a POST request. The default action is to respond with "error 404 (not found)".

send_response(*code*)
See Python's BaseHTTPRequestHandler.send_response().

Exscript.servers.httpd.**md5hex**(*x*)

Exscript.servers.server module

Base class for all servers.

class Exscript.servers.server.**Server**(*host*, *port*, *device*, *encoding='utf8'*)
Bases: multiprocessing.process.Process

Base class of the Telnet and SSH servers. Servers are intended to be used for tests and attempt to emulate a device using the behavior of the associated *Exscript.emulators.VirtualDevice*. Sample usage:

```
device = VirtualDevice('myhost')
daemon = Telnetd('localhost', 1234, device)
device.add_command('ls', 'ok', prompt = True)
device.add_command('exit', daemon.exit_command)
daemon.start() # Start the server.
daemon.exit()  # Stop the server.
daemon.join()  # Wait until it terminates.
```

__init__(*host*, *port*, *device*, *encoding='utf8'*)
Constructor.

Parameters

- **host** (*str*) – The address against which the daemon binds.
- **port** (*str*) – The TCP port on which to listen.
- **device** (VirtualDevice) – A virtual device instance.
- **encoding** (*str*) – The encoding of data between client and server.

exit()
Stop the daemon without waiting for the thread to terminate.

exit_command(*cmd*)
Like exit(), but may be used as a handler in add_command.

Parameters **cmd** (*str*) – The command that causes the server to exit.

Exscript.servers.sshd module

An SSH2 server.

class Exscript.servers.sshd.**SSHd**(*host*, *port*, *device*, *key=None*)
Bases: *Exscript.servers.server.Server*

A SSH2 server. Usage:

```
device = VirtualDevice('myhost')
daemon = SSHd('localhost', 1234, device)
device.add_command('ls', 'ok', prompt = True)
device.add_command('exit', daemon.exit_command)
daemon.start() # Start the server.
daemon.exit()  # Stop the server.
daemon.join()  # Wait until it terminates.
```

Parameters **key** – An Exscript.PrivateKey object.

__init__(*host*, *port*, *device*, *key=None*)

Exscript.servers.telnetd module

A Telnet server.

class Exscript.servers.telnetd.**Telnetd**(*host*, *port*, *device*, *encoding='utf8'*)
Bases: *Exscript.servers.server.Server*

A Telnet server. Usage:

```
device = VirtualDevice('myhost')
daemon = Telnetd('localhost', 1234, device)
device.add_command('ls', 'ok', prompt = True)
device.add_command('exit', daemon.exit_command)
daemon.start() # Start the server.
daemon.exit()  # Stop the server.
daemon.join()  # Wait until it terminates.
```

Exscript.stdlib package

Submodules

Exscript.stdlib.connection module

Exscript.stdlib.connection.**authenticate**(**args*, ***kwargs*)
Looks for any username/password prompts on the current connection and logs in using the login information that was passed to Exscript.

Exscript.stdlib.connection.**authenticate_user**(**args*, ***kwargs*)
Like authenticate(), but logs in using the given user and password. If a user and password are not given, the function uses the same user and password that were used at the last login attempt; it is an error if no such attempt was made before.

Parameters

- **user** (string) – A username.
- **password** (string) – A password.

Exscript.stdlib.connection.**authorize**(**args*, ***kwargs*)
Looks for a password prompt on the current connection and enters the given password. If a password is not given, the function uses the same password that was used at the last login attempt; it is an error if no such attempt was made before.

Parameters **password** (string) – A password.

Exscript.stdlib.connection.**auto_authorize**(**args*, ***kwargs*)
Executes a command on the remote host that causes an authorization procedure to be started, then authorizes using the given password in the same way in which authorize() works. Depending on the detected operating system of the remote host the following commands are started:

•on IOS, the "enable" command is executed.

•nothing on other operating systems yet.

Parameters **password** (string) – A password.

Exscript.stdlib.connection.**autoinit**(**args*, ***kwargs*)
Make the remote host more script-friendly by automatically executing one or more commands on it. The commands executed depend on the currently used driver. For example, the driver for Cisco IOS would execute the following commands:

```
term len 0
term width 0
```

Exscript.stdlib.connection.**close**(**args*, ***kwargs*)
Closes the existing connection with the remote host. This function is rarely used, as normally Exscript closes the connection automatically when the script has completed.

Exscript.stdlib.connection.**exec_**(**args*, ***kwargs*)
Sends the given data to the remote host and waits until the host has responded with a prompt. If the given data is a list of strings, each item is sent, and after each item a prompt is expected.

This function also causes the response of the command to be stored in the built-in __response__ variable.

Parameters **data** (string) – The data that is sent.

Exscript.stdlib.connection.**execline**(**args*, ***kwargs*)
Like exec(), but appends a newline to the command in data before sending it.

Parameters **data** (string) – The data that is sent.

Exscript.stdlib.connection.**guess_os**(**args*, ***kwargs*)
Guesses the operating system of the connected host.

The recognition is based on the past conversation that has happened on the host; Exscript looks for known patterns and maps them to specific operating systems.

Return type *string*

Returns The operating system.

Exscript.stdlib.connection.**send**(**args*, ***kwargs*)
Like exec(), but does not wait for a response of the remote host after sending the command.

Parameters **data** (string) – The data that is sent.

Exscript.stdlib.connection.**sendline**(**args*, ***kwargs*)
Like execline(), but does not wait for a response of the remote host after sending the command.

Parameters **data** (*string*) – The data that is sent.

Exscript.stdlib.connection.**set_error**(**args*, ***kwargs*)
Defines a pattern that, whenever detected in the response of the remote host, causes an error to be raised.

In other words, whenever Exscript waits for a prompt, it searches the response of the host for the given pattern and raises an error if the pattern is found.

Parameters **error_re** (*regex*) – The error pattern.

Exscript.stdlib.connection.**set_prompt**(**args*, ***kwargs*)
Defines the pattern that is recognized at any future time when Exscript needs to wait for a prompt. In other words, whenever Exscript waits for a prompt, it searches the response of the host for the given pattern and continues as soon as the pattern is found.

Exscript waits for a prompt whenever it sends a command (unless the send() method was used). set_prompt() redefines as to what is recognized as a prompt.

Parameters **prompt** (*regex*) – The prompt pattern.

Exscript.stdlib.connection.**set_timeout**(**args*, ***kwargs*)
Defines the time after which Exscript fails if it does not receive a prompt from the remote host.

Parameters **timeout** (*int*) – The timeout in seconds.

Exscript.stdlib.connection.**wait_for**(**args*, ***kwargs*)
Waits until the response of the remote host contains the given pattern.

Parameters **prompt** (*regex*) – The prompt pattern.

Exscript.stdlib.crypt module

Exscript.stdlib.crypt.**otp**(**args*, ***kwargs*)
Calculates a one-time password hash using the given password, seed, and sequence number and returns it. Uses the md4/sixword algorithm as supported by TACACS+ servers.

Parameters

- **password** (*string*) – A password.
- **seed** (*string*) – A username.
- **seqs** (*int*) – A sequence number, or a list of sequence numbers.

Return type *string*

Returns A hash, or a list of hashes.

Exscript.stdlib.file module

Exscript.stdlib.file.**chmod**(*scope*, *filename*, *mode*)
Changes the permissions of the given file (or list of files) to the given mode. You probably want to use an octal representation for the integer, e.g. "chmod(myfile, 0644)".

Parameters

- **filename** (*string*) – A filename.
- **mode** (*int*) – The access permissions.

Exscript.stdlib.file.**clear**(*scope*, *filename*)
Clear the contents of the given file. The file is created if it does not exist.

Parameters **filename** (string) – A filename.

Exscript.stdlib.file.**exists**(**args*, ***kwargs*)
Returns True if the file with the given name exists, False otherwise. If a list of files is given, the function returns True only if ALL of the files exist.

Parameters **filename** (string) – A filename.

Return type bool

Returns The operating system of the remote device.

Exscript.stdlib.file.**mkdir**(*scope*, *dirname*, *mode=None*)
Creates the given directory (or directories). The optional access permissions are set to the given mode, and default to whatever is the umask on your system defined.

Parameters

- **dirname** (string) – A filename, or a list of dirnames.
- **mode** (*int*) – The access permissions.

Exscript.stdlib.file.**read**(*scope*, *filename*)
Reads the given file and returns the result. The result is also stored in the built-in __response__ variable.

Parameters **filename** (string) – A filename.

Return type *string*

Returns The content of the file.

Exscript.stdlib.file.**rm**(*scope*, *filename*)
Deletes the given file (or files) from the file system.

Parameters **filename** (string) – A filename, or a list of filenames.

Exscript.stdlib.file.**write**(*scope, filename, lines, mode=['a']*)
Writes the given string into the given file. The following modes are supported:

•'a': Append to the file if it already exists.

•'w': Replace the file if it already exists.

Parameters

- **filename** (string) – A filename.
- **lines** (string) – The data that is written into the file.
- **mode** (string) – Any of the above listed modes.

Exscript.stdlib.ipv4 module

Exscript.stdlib.ipv4.**broadcast**(**args*, ***kwargs*)
Given a prefix, this function returns the corresponding broadcast address.

Parameters **prefixes** (string) – An IP prefix.

Return type *string*

Returns The broadcast address(es) of the prefix length(s).

Exscript.stdlib.ipv4.**in_network**(**args*, ***kwargs*)
Returns True if the given destination is in the network range that is defined by the given prefix (e.g. 10.0.0.1/22). If the given prefix does not have a prefix length specified, the given default prefix length is applied. If no such prefix length is given, the default length is /24.

If a list of prefixes is passed, this function returns True only if the given destination is in ANY of the given prefixes.

Parameters

- **prefixes** (string) – A prefix, or a list of IP prefixes.
- **destination** (string) – An IP address.
- **default_pfxlen** (*int*) – The default prefix length.

Return type True

Returns Whether the given destination is in the given network.

Exscript.stdlib.ipv4.**mask**(**args*, ***kwargs*)
Applies the given IP mask (e.g. 255.255.255.0) to the given IP address (or list of IP addresses) and returns it.

Parameters

- **ips** (string) – A prefix, or a list of IP prefixes.
- **mask** (string) – An IP mask.

Return type *string*

Returns The network(s) that result(s) from applying the mask.

Exscript.stdlib.ipv4.**mask2pfxlen**(**args*, ***kwargs*)
Converts the given IP mask(s) (e.g. 255.255.255.0) to prefix length(s).

Parameters **masks** (string) – An IP mask, or a list of masks.

Return type *string*

Returns The prefix length(s) that result(s) from converting the mask.

Exscript.stdlib.ipv4.**network**(**args*, ***kwargs*)
Given a prefix, this function returns the corresponding network address.

Parameters **prefixes** (string) – An IP prefix.

Return type *string*

Returns The network address(es) of the prefix length(s).

Exscript.stdlib.ipv4.**pfxlen2mask**(**args*, ***kwargs*)
Converts the given prefix length(s) (e.g. 30) to IP mask(s).

Parameters **pfxlen** (*int*) – An IP prefix length.

Return type *string*

Returns The mask(s) that result(s) from converting the prefix length.

Exscript.stdlib.ipv4.**pfxmask**(**args*, ***kwargs*)
Applies the given prefix length to the given ips, resulting in a (list of) IP network addresses.

Parameters

- **ips** (string) – An IP address, or a list of IP addresses.
- **pfxlen** (*int*) – An IP prefix length.

Return type *string*

Returns The mask(s) that result(s) from converting the prefix length.

`Exscript.stdlib.ipv4.`**`remote_ip`**(**args*, ***kwargs*)
Given an IP address, this function calculates the remaining available IP address under the assumption that it is a /30 network. In other words, given one link net address, this function returns the other link net address.

Parameters **`local_ips`** (`string`) – An IP address, or a list of IP addresses.

Return type *string*

Returns The other IP address of the link address pair.

Exscript.stdlib.list module

`Exscript.stdlib.list.`**`get`**(**args*, ***kwargs*)
Returns a copy of the list item with the given index. It is an error if an item with teh given index does not exist.

Parameters

- **`source`** (`string`) – A list of strings.
- **`index`** (`string`) – A list of strings.

Return type *string*

Returns The cleaned up list of strings.

`Exscript.stdlib.list.`**`length`**(**args*, ***kwargs*)
Returns the number of items in the list.

Return type *string*

Returns The model of the remote device.

`Exscript.stdlib.list.`**`new`**(**args*, ***kwargs*)
Returns a new, empty list.

Return type *string*

Returns The model of the remote device.

`Exscript.stdlib.list.`**`unique`**(**args*, ***kwargs*)
Returns a copy of the given list in which all duplicates are removed such that one of each item remains in the list.

Parameters **`source`** (`string`) – A list of strings.

Return type *string*

Returns The cleaned up list of strings.

Exscript.stdlib.mysys module

`Exscript.stdlib.mysys.`**`env`**(*scope*, *varname*)
Returns the value of the environment variable with the given name.

Parameters **`varnames`** (`string`) – A variable name.

`Exscript.stdlib.mysys.`**`execute`**(*scope*, *command*)
Executes the given command locally.

Parameters **command** (string) – A shell command.

Exscript.stdlib.mysys.**message**(**args*, ***kwargs*)
Writes the given string to stdout.

Parameters **string** (string) – A string, or a list of strings.

Exscript.stdlib.mysys.**wait**(**args*, ***kwargs*)
Waits for the given number of seconds.

Parameters **seconds** (*int*) – The wait time in seconds.

Exscript.stdlib.string module

Exscript.stdlib.string.**replace**(**args*, ***kwargs*)
Returns a copy of the given string (or list of strings) in which all occurrences of the given source are replaced by the given dest.

Parameters

- **strings** (string) – A string, or a list of strings.
- **source** (string) – What to replace.
- **dest** (string) – What to replace it with.

Return type *string*

Returns The resulting string, or list of strings.

Exscript.stdlib.string.**tolower**(**args*, ***kwargs*)
Returns the given string in lower case.

Parameters **strings** (string) – A string, or a list of strings.

Return type *string*

Returns The resulting string, or list of strings in lower case.

Exscript.stdlib.string.**toupper**(**args*, ***kwargs*)
Returns the given string in upper case.

Parameters **strings** (string) – A string, or a list of strings.

Return type *string*

Returns The resulting string, or list of strings in upper case.

Exscript.stdlib.util module

Exscript.stdlib.util.**secure_function**(*function*)

Exscript.util package

Submodules

Exscript.util.buffer module

A buffer object.

class Exscript.util.buffer.**MonitoredBuffer**(*io=None*)
Bases: future.types.newobject.newobject

A specialized string buffer that allows for monitoring the content using regular expression-triggered callbacks.

__init__(*io=None*)
Constructor. The data is stored in the given file-like object. If no object is given, or the io argument is None, a new StringIO is used.

Parameters **io** (*file-like object*) – A file-like object that is used for storing the data.

add_monitor(*pattern*, *callback*, *limit=80*)
Calls the given function whenever the given pattern matches the buffer.

Arguments passed to the callback are the index of the match, and the match object of the regular expression.

Parameters

- **pattern** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regular expressions.
- **callback** (*callable*) – The function that is called.
- **limit** (*int*) – The maximum size of the tail of the buffer that is searched, in number of bytes.

append(*data*)
Appends the given data to the buffer, and triggers all connected monitors, if any of them match the buffer content.

Parameters **data** (*str*) – The data that is appended.

clear()
Removes all data from the buffer.

head(*bytes*)
Returns the number of given bytes from the head of the buffer. The buffer remains unchanged.

Parameters **bytes** (*int*) – The number of bytes to return.

pop(*bytes*)
Like *head()*, but also removes the head from the buffer.

Parameters **bytes** (*int*) – The number of bytes to return and remove.

size()
Returns the size of the buffer.

Return type int

Returns The size of the buffer in bytes.

tail(*bytes*)
Returns the number of given bytes from the tail of the buffer. The buffer remains unchanged.

Parameters **bytes** (*int*) – The number of bytes to return.

Exscript.util.cast module

Handy shortcuts for converting types.

Exscript.util.cast.**to_host**(*host*, *default_protocol='telnet'*, *default_domain=''*)
Given a string or a Host object, this function returns a Host object.

Parameters

- **host** (*string|Host*) – A hostname (may be URL formatted) or a Host object.
- **default_protocol** (*str*) – Passed to the Host constructor.
- **default_domain** (*str*) – Appended to each hostname that has no domain.

Return type *Host*

Returns The Host object.

Exscript.util.cast.**to_hosts**(*hosts*, *default_protocol='telnet'*, *default_domain=''*)

Given a string or a Host object, or a list of strings or Host objects, this function returns a list of Host objects.

Parameters

- **hosts** (*string|Host|list(string)|list(Host)*) – One or more hosts or hostnames.
- **default_protocol** (*str*) – Passed to the Host constructor.
- **default_domain** (*str*) – Appended to each hostname that has no domain.

Return type *list*[*Host*]

Returns A list of Host objects.

Exscript.util.cast.**to_list**(*item*)

If the given item is iterable, this function returns the given item. If the item is not iterable, this function returns a list with only the item in it.

Parameters **item** (*object*) – Any object.

Return type *list*

Returns A list with the item in it.

Exscript.util.cast.**to_regex**(*regex*, *flags=0*)

Given a string, this function returns a new re.RegexObject. Given a re.RegexObject, this function just returns the same object.

Parameters

- **regex** (*string|re.RegexObject*) – A regex or a re.RegexObject
- **flags** (*int*) – See Python's re.compile().

Return type re.RegexObject

Returns The Python regex object.

Exscript.util.cast.**to_regexs**(*regexs*)

Given a string or a re.RegexObject, or a list of strings or re.RegexObjects, this function returns a list of re.RegexObjects.

Parameters **regexs** (*str|re.RegexObject|list(str|re.RegexObject)*) – One or more regexs or re.RegexObjects.

Return type *list*(re.RegexObject)

Returns A list of re.RegexObjects.

Exscript.util.collections module

class Exscript.util.collections.**OrderedDefaultDict**(*default_factory=None*, **a*, ***kw*)
Bases: collections.OrderedDict

A fusion of Python's defaultdict and Python's OrderedDict.

__init__(*default_factory=None*, **a*, ***kw*)

copy()

Exscript.util.crypt module

Encryption related utilities.

Exscript.util.crypt.**otp**(*password*, *seed*, *sequence*)
Calculates a one-time password hash using the given password, seed, and sequence number and returns it. Uses the MD4/sixword algorithm as supported by TACACS+ servers.

Parameters

- **password** (*str*) – A password.
- **seed** (*str*) – A cryptographic seed.
- **sequence** (*int*) – A sequence number.

Return type *string*

Returns A hash.

Exscript.util.daemonize module

Daemonizing a process.

Exscript.util.daemonize.**daemonize**()
Forks and daemonizes the current process. Does not automatically track the process id; to do this, use *Exscript.util.pidutil*.

Exscript.util.decorator module

Decorators for callbacks passed to Queue.run().

Exscript.util.decorator.**autoauthenticate**(*flush=True*, *attempts=1*)
Wraps the given function such that conn.authenticate() is executed before calling it. Example:

```
@autoauthenticate(attempts = 2)
def my_func(job, host, conn):
    pass # Do something.
Exscript.util.start.quickrun('myhost', my_func)
```

Parameters

- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **attempts** (*int*) – The number of login attempts if login fails.

Return type function

Returns The wrapped function.

Exscript.util.decorator.**autologin**(*flush=True*, *attempts=1*)
Wraps the given function such that conn.login() is executed before calling it. Example:

```
@autologin(attempts = 2)
def my_func(job, host, conn):
    pass # Do something.
Exscript.util.start.quickrun('myhost', my_func)
```

Parameters

- **flush** (*bool*) – Whether to flush the last prompt from the buffer.
- **attempts** (*int*) – The number of login attempts if login fails.

Return type function

Returns The wrapped function.

Exscript.util.decorator.**bind**(*function*, **args*, ***kwargs*)
Wraps the given function such that when it is called, the given arguments are passed in addition to the connection argument.

Parameters

- **function** (*function*) – The function that's ought to be wrapped.
- **args** (list) – Passed on to the called function.
- **kwargs** (*dict*) – Passed on to the called function.

Return type function

Returns The wrapped function.

Exscript.util.decorator.**os_function_mapper**(*map*)
When called with an open connection, this function uses the conn.guess_os() function to guess the operating system of the connected host. It then uses the given map to look up a function name that corresponds to the operating system, and calls it. Example:

```
def ios_xr(job, host, conn):
    pass # Do something.

def junos(job, host, conn):
    pass # Do something else.

def shell(job, host, conn):
    pass # Do something else.

Exscript.util.start.quickrun('myhost', os_function_mapper(globals()))
```

An exception is raised if a matching function is not found in the map.

Parameters

- **conn** (Exscript.protocols.Protocol) – The open connection.
- **map** (*dict*) – A dictionary mapping operating system name to a function.
- **args** (list) – Passed on to the called function.
- **kwargs** (*dict*) – Passed on to the called function.

Return type object

Returns The return value of the called function.

Exscript.util.event module

A simple signal/event mechanism.

class Exscript.util.event.**Event**
Bases: future.types.newobject.newobject

A simple signal/event mechanism, to be used like this:

```
def mycallback(arg, **kwargs):
    print(arg, kwargs['foo'])

myevent = Event()
myevent.connect(mycallback)
myevent.emit('test', foo = 'bar')
# Or just: myevent('test', foo = 'bar')
```

__init__()
Constructor.

connect(*callback*, **args*, ***kwargs*)
Connects the event with the given callback. When the signal is emitted, the callback is invoked.

Hint: The signal handler is stored with a hard reference, so you need to make sure to call *disconnect()* if you want the handler to be garbage collected.

Parameters

- **callback** (*object*) – The callback function.
- **args** (*tuple*) – Optional arguments passed to the callback.
- **kwargs** (*dict*) – Optional keyword arguments passed to the callback.

disconnect(*callback*)
Disconnects the signal from the given function.

Parameters **callback** (*object*) – The callback function.

disconnect_all()
Disconnects all connected functions from all signals.

emit(**args*, ***kwargs*)
Emits the signal, passing the given arguments to the callbacks. If one of the callbacks returns a value other than None, no further callbacks are invoked and the return value of the callback is returned to the caller of emit().

Parameters

- **args** (*tuple*) – Optional arguments passed to the callbacks.
- **kwargs** (*dict*) – Optional keyword arguments passed to the callbacks.

Return type object

Returns Returns None if all callbacks returned None. Returns the return value of the last invoked callback otherwise.

is_connected(*callback*)
Returns True if the event is connected to the given function.

Parameters **callback** (*object*) – The callback function.

Return type bool

Returns Whether the signal is connected to the given function.

listen(*callback*, **args*, ***kwargs*)
Like *connect()*, but uses a weak reference instead of a normal reference. The signal is automatically disconnected as soon as the handler is garbage collected.

Hint: Storing signal handlers as weak references means that if your handler is a local function, it may be garbage collected. To prevent this, use *connect()* instead.

Parameters

- **callback** (*object*) – The callback function.
- **args** (*tuple*) – Optional arguments passed to the callback.
- **kwargs** (*dict*) – Optional keyword arguments passed to the callback.

Return type *Exscript.util.weakmethod.WeakMethod*

Returns The newly created weak reference to the callback.

n_subscribers()
Returns the number of connected subscribers.

Return type int

Returns The number of subscribers.

Exscript.util.file module

Utilities for reading data from files.

Exscript.util.file.**get_accounts_from_file**(*filename*)
Reads a list of user/password combinations from the given file and returns a list of Account instances. The file content has the following format:

```
[account-pool]
user1 = cGFzc3dvcmQ=
user2 = cGFzc3dvcmQ=
```

Note that "cGFzc3dvcmQ=" is a base64 encoded password. If the input file contains extra config sections other than "account-pool", they are ignored. Each password needs to be base64 encrypted. To encrypt a password, you may use the following command:

```
python -c 'import base64; print(base64.b64encode("thepassword"))'
```

Parameters **filename** (*string*) – The name of the file containing the list of accounts.

Return type *list*[*Account*]

Returns The newly created account instances.

Exscript.util.file.**get_hosts_from_csv**(*filename*, *default_protocol='telnet'*, *default_domain=''*, *encoding='utf-8'*)

Reads a list of hostnames and variables from the tab-separated .csv file with the given name. The first line of the file must contain the column names, e.g.:

```
address testvar1        testvar2
10.0.0.1        value1  othervalue
10.0.0.1        value2  othervalue2
10.0.0.2        foo     bar
```

For the above example, the function returns *two* host objects, where the 'testvar1' variable of the first host holds a list containing two entries ('value1' and 'value2'), and the 'testvar1' variable of the second host contains a list with a single entry ('foo').

Both, the address and the hostname of each host are set to the address given in the first column. If you want the hostname set to another value, you may add a second column containing the hostname:

```
address hostname        testvar
10.0.0.1        myhost  value
10.0.0.2        otherhost       othervalue
```

Parameters

- **filename** (string) – A full filename.
- **default_protocol** (*str*) – Passed to the Host constructor.
- **default_domain** (*str*) – Appended to each hostname that has no domain.
- **encoding** (*str*) – The encoding of the file.

Return type *list*[*Host*]

Returns The newly created host instances.

Exscript.util.file.**get_hosts_from_file**(*filename*, *default_protocol='telnet'*, *default_domain=''*, *remove_duplicates=False*, *encoding='utf-8'*)

Reads a list of hostnames from the file with the given name.

Parameters

- **filename** (string) – A full filename.
- **default_protocol** (*str*) – Passed to the Host constructor.
- **default_domain** (*str*) – Appended to each hostname that has no domain.
- **remove_duplicates** (*bool*) – Whether duplicates are removed.
- **encoding** (*str*) – The encoding of the file.

Return type *list*[*Host*]

Returns The newly created host instances.

Exscript.util.file.**load_lib**(*filename*)

Loads a Python file containing functions, and returns the content of the __lib__ variable. The __lib__ variable must contain a dictionary mapping function names to callables.

Returns a dictionary mapping the namespaced function names to callables. The namespace is the basename of the file, without file extension.

The result of this function can later be passed to run_template:

```
functions = load_lib('my_library.py')
run_template(conn, 'foo.exscript', **functions)
```

Parameters **filename** (string) – A full filename.

Return type dict[string->object]

Returns The loaded functions.

Exscript.util.impl module

Development tools.

class Exscript.util.impl.**Context**(*obj*)
Bases: Exscript.util.impl._Context

context()

class Exscript.util.impl.**Decorator**(*obj*)
Bases: future.types.newobject.newobject

__init__(*obj*)

Exscript.util.impl.**add_label**(*obj*, *name*, ***kwargs*)
Labels an object such that it can later be checked with *get_label()*.

Parameters

- **obj** (*object*) – The object that is labeled.
- **name** (*str*) – A label.
- **kwargs** (*dict*) – Optional values to store with the label.

Return type object

Returns The labeled function.

Exscript.util.impl.**copy_labels**(*src*, *dst*)
Copies all labels of one object to another object.

Parameters

- **src** (*object*) – The object to check read the labels from.
- **dst** (*object*) – The object into which the labels are copied.

Exscript.util.impl.**debug**(*func*)
Decorator that prints a message whenever a function is entered or left.

Exscript.util.impl.**deprecated**(*func*)
A decorator for marking functions as deprecated. Results in a printed warning message when the function is used.

Exscript.util.impl.**deprecation**(*msg*)
Prints a deprecation warning.

Exscript.util.impl.**format_exception**(*thetype*, *ex*, *tb*)
This function is a drop-in replacement for Python's traceback.format_exception().

Since traceback objects can not be pickled, Exscript is forced to manipulate them before they are passed accross process boundaries. This leads to the fact the Python's traceback.format_exception() no longer works for those objects.

This function works with any traceback object, regardless of whether or not Exscript manipulated it.

Exscript.util.impl.**get_label**(*obj*, *name*)
Checks whether an object has the given label attached (see *add_label()*) and returns the associated options.

Parameters

- **obj** (*object*) – The object to check for the label.
- **name** (*str*) – A label.

Return type dict or None

Returns The optional values if the label is attached, None otherwise.

Exscript.util.impl.**serializeable_exc_info**(*thetype*, *ex*, *tb*)
Since traceback objects can not be pickled, this function manipulates exception info tuples before they are passed accross process boundaries.

Exscript.util.impl.**serializeable_sys_exc_info**()
Convenience wrapper around serializeable_exc_info, equivalent to serializeable_exc_info(sys.exc_info()).

Exscript.util.impl.**synchronized**(*func*)
Decorator for synchronizing method access.

Exscript.util.interact module

Tools for interacting with the user on the command line.

class Exscript.util.interact.**InputHistory**(*filename='~/.exscript_history'*, *section='sphinx-build'*)
Bases: future.types.newobject.newobject

When prompting a user for input it is often useful to record his input in a file, and use previous input as a default value. This class allows for recording user input in a config file to allow for such functionality.

__init__(*filename='~/.exscript_history'*, *section='sphinx-build'*)
Constructor. The filename argument allows for listing on or more config files, and is passed to Python's RawConfigParser; please consult the documentation of RawConfigParser.read() if you require more information. The optional section argument allows to specify a section under which the input is stored in the config file. The section defaults to the name of the running script.

Silently creates a tempfile if the given file can not be opened, such that the object behavior does not change, but the history is not remembered across instances.

Parameters

- **filename** (*str|list(str)*) – The config file.
- **section** (*str*) – The section in the configfile.

get(*key*, *default=None*)
Returns the input with the given key from the section that was passed to the constructor. If either the section or the key are not found, the default value is returned.

Parameters

- **key** (*str*) – The key for which to return a value.
- **default** (*str|object*) – The default value that is returned.

Return type str|object

Returns The value from the config file, or the default.

set(*key*, *value*)
Saves the input with the given key in the section that was passed to the constructor. If either the section or the key are not found, they are created.

Does nothing if the given value is None.

Parameters

- **key** (*str*) – The key for which to define a value.
- **value** (*str|None*) – The value that is defined, or None.

Return type str|None

Returns The given value.

Exscript.util.interact.**get_filename**(*key*, *message*, *default=None*, *history=None*)
Like *prompt()*, but only accepts the name of an existing file as an input.

Parameters

- **key** (*str*) – The key under which to store the input in the *InputHistory*.
- **message** (*str*) – The user prompt.
- **default** (*str|None*) – The offered default if none was found in the history.
- **history** (:class:‘InputHistory‘|None) – The history used for recording default values, or None.

Exscript.util.interact.**get_login**()
Prompts the user for the login name using get_user(), and also asks for the password. Returns a tuple containing the username and the password. May throw an exception if EOF is given by the user.

Return type (*string*, *string*)

Returns A tuple containing the username and the password.

Exscript.util.interact.**get_user**(*prompt=None*)
Prompts the user for his login name, defaulting to the USER environment variable. Returns a string containing the username. May throw an exception if EOF is given by the user.

Parameters **prompt** (*str|None*) – The user prompt or the default one if None.

Return type *string*

Returns A username.

Exscript.util.interact.**prompt**(*key*, *message*, *default=None*, *doverh=True*, *strip=True*, *check=None*, *history=None*)
Prompt the user for input. This function is similar to Python’s built in raw_input, with the following differences:

- You may specify a default value that is returned if the user presses “enter” without entering anything.
- The user’s input is recorded in a config file, and offered as the default value the next time this function is used (based on the key argument).

The config file is based around the *InputHistory*. If a history object is not passed in the history argument, a new one will be created.

The key argument specifies under which name the input is saved in the config file.

The given default value is only used if a default was not found in the history.

The strip argument specifies that the returned value should be stripped of whitespace (default).

The check argument allows for validating the input; if the validation fails, the user is prompted again before the value is stored in the InputHistory. Example usage:

```
def validate(input):
    if len(input) < 4:
        return 'Please enter at least 4 characters!'
value = prompt('test', 'Enter a value', 'My Default', check = validate)
print('You entered:', value)
```

This leads to the following output:

```
Please enter a value [My Default]: abc
Please enter at least 4 characters!
Please enter a value [My Default]: Foobar
You entered: Foobar
```

The next time the same code is started, the input 'Foobar' is remembered:

```
Please enter a value [Foobar]:        (enters nothing)
You entered: Foobar
```

Parameters

- **key** (*str*) – The key under which to store the input in the *InputHistory*.
- **message** (*str*) – The user prompt.
- **default** (*str|None*) – The offered default if none was found in the history.
- **doverh** (*bool*) – Whether to prefer default value over history value.
- **strip** (*bool*) – Whether to remove whitespace from the input.
- **check** (*callable*) – A function that is called for validating the input.
- **history** (:class:'InputHistory'|None) – The history used for recording default values, or None.

Exscript.util.interact.**read_login**()
Like get_login(), but returns an Account object.

Return type *Account*

Returns A new account.

Exscript.util.ip module

Wrapper around the ipv4 and ipv6 modules to handle both, ipv4 and ipv6.

Exscript.util.ip.**clean_ip**(*ip*)
Cleans the ip address up, useful for removing leading zeros, e.g.:

```
192.168.010.001 -> 192.168.10.1
1234:0000:0000:0000:0000:0000:0000:000A -> 1234::a
```

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The cleaned up IP.

Exscript.util.ip.**is_ip**(*string*)
Returns True if the given string is an IPv4 or IPv6 address, False otherwise.

Parameters **string** (string) – Any string.

Return type bool

Returns True if the string is an IP address, False otherwise.

Exscript.util.ip.**normalize_ip**(*ip*)
Transform the address into a fixed-length form, such as:

192.168.0.1 -> 192.168.000.001 1234::A -> 1234:0000:0000:0000:0000:0000:0000:000a

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The normalized IP.

Exscript.util.ipv4 module

IPv4 address calculation and conversion.

Exscript.util.ipv4.**broadcast**(*prefix*, *default_length=24*)
Given a prefix, this function returns the corresponding broadcast address.

Parameters

- **prefix** (string) – An IP prefix.
- **default_length** (*long*) – The default ip prefix length.

Return type *string*

Returns The IP broadcast address.

Exscript.util.ipv4.**clean_ip**(*ip*)
Cleans the ip address up, useful for removing leading zeros, e.g.:

```
192.168.010.001 -> 192.168.10.1
```

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The cleaned up IP.

Exscript.util.ipv4.**int2ip**(*number*)
Converts the given integer value to an IP address.

Parameters **number** (*long*) – An IP as a number.

Return type *string*

Returns The IP address.

Exscript.util.ipv4.**ip2int**(*ip*)
Converts the given IP address to a 4 byte integer value.

Parameters **ip** (string) – An IP address.

Return type long

Returns The IP, converted to a number.

Exscript.util.ipv4.**is_ip**(*string*)
Returns True if the given string is an IPv4 address, False otherwise.

Parameters **string** (string) – Any string.

Return type bool

Returns True if the string is an IP address, False otherwise.

Exscript.util.ipv4.**is_private**(*ip*)
Returns True if the given IP address is private, returns False otherwise.

Parameters **ip** (string) – An IP address.

Return type bool

Returns True if the IP is private, False otherwise.

Exscript.util.ipv4.**mask2pfxlen**(*mask*)
Converts the given IP mask to a prefix length.

Parameters **mask** (string) – An IP mask.

Return type long

Returns The mask, as a long value.

Exscript.util.ipv4.**matches_prefix**(*ip*, *prefix*)
Returns True if the given IP address is part of the given network, returns False otherwise.

Parameters

- **ip** (string) – An IP address.
- **prefix** (string) – An IP prefix.

Return type bool

Returns True if the IP is in the prefix, False otherwise.

Exscript.util.ipv4.**network**(*prefix*, *default_length=24*)
Given a prefix, this function returns the corresponding network address.

Parameters

- **prefix** (string) – An IP prefix.
- **default_length** (*long*) – The default ip prefix length.

Return type *string*

Returns The IP network address.

Exscript.util.ipv4.**normalize_ip**(*ip*)
Transform the address into a fixed-length form, such as:

```
192.168.0.1 -> 192.168.000.001
```

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The normalized IP.

Exscript.util.ipv4.**parse_prefix**(*prefix*, *default_length=24*)
Splits the given IP prefix into a network address and a prefix length. If the prefix does not have a length (i.e., it is a simple IP address), it is presumed to have the given default length.

Parameters

- **prefix** (string) – An IP mask.
- **default_length** (*long*) – The default ip prefix length.

Return type *string*, int

Returns A tuple containing the IP address and prefix length.

Exscript.util.ipv4.**pfxlen2mask**(*pfxlen*)
Converts the given prefix length to an IP mask.

Parameters **pfxlen** (*int*) – A prefix length.

Return type *string*

Returns The mask.

Exscript.util.ipv4.**pfxlen2mask_int**(*pfxlen*)
Converts the given prefix length to an IP mask value.

Parameters **pfxlen** (*int*) – A prefix length.

Return type long

Returns The mask, as a long value.

Exscript.util.ipv4.**remote_ip**(*local_ip*)
Given an IP address, this function calculates the remaining available IP address under the assumption that it is a /30 network. In other words, given one link net address, this function returns the other link net address.

Parameters **local_ip** (string) – An IP address.

Return type *string*

Returns The other IP address of the link address pair.

Exscript.util.ipv4.**sort**(*iterable*)
Given an IP address list, this function sorts the list.

Parameters **iterable** (*Iterator*) – An IP address list.

Return type *list*

Returns The sorted IP address list.

Exscript.util.ipv6 module

IPv6 address calculation and conversion.

Exscript.util.ipv6.**clean_ip**(*ip*)
Cleans the ip address up, useful for removing leading zeros, e.g.:

```
1234:0:01:02:: -> 1234:0:1:2::
1234:0000:0000:0000:0000:0000:0000:000A -> 1234::a
1234:0000:0000:0000:0001:0000:0000:0000 -> 1234:0:0:0:1::
0000:0000:0000:0000:0001:0000:0000:0000 -> ::1:0:0:0
```

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The cleaned up IP.

Exscript.util.ipv6.**is_ip**(*string*)
Returns True if the given string is an IPv6 address, False otherwise.

Parameters **string** (string) – Any string.

Return type bool

Returns True if the string is an IP address, False otherwise.

Exscript.util.ipv6.**normalize_ip**(*ip*)
Transform the address into a standard, fixed-length form, such as:

1234:0:01:02:: -> 1234:0000:0001:0002:0000:0000:0000:0000 1234::A -> 1234:0000:0000:0000:0000:0000:0000:000a

Parameters **ip** (string) – An IP address.

Return type *string*

Returns The normalized IP.

Exscript.util.ipv6.**parse_prefix**(*prefix*, *default_length=128*)
Splits the given IP prefix into a network address and a prefix length. If the prefix does not have a length (i.e., it is a simple IP address), it is presumed to have the given default length.

Parameters

- **prefix** (string) – An IP mask.
- **default_length** (*long*) – The default ip prefix length.

Return type *string*, int

Returns A tuple containing the IP address and prefix length.

Exscript.util.log module

Logging utilities.

Exscript.util.log.**log_to**(*logger*)
Wraps a function that has a connection passed such that everything that happens on the connection is logged using the given logger.

Parameters **logger** (Logger) – The logger that handles the logging.

Exscript.util.log.**log_to_file**(*logdir*, *mode='a'*, *delete=False*, *clearmem=True*)
Like *log_to()*, but automatically creates a new FileLogger instead of having one passed. Note that the logger stays alive (in memory) forever. If you need to control the lifetime of a logger, use *log_to()* instead.

Exscript.util.mail module

Sending and formatting emails.

class Exscript.util.mail.**Mail**(*sender=None*, *to=''*, *cc=''*, *bcc=''*, *subject=''*, *body=''*)
Bases: future.types.newobject.newobject

Represents an email.

__init__(*sender=None*, *to=''*, *cc=''*, *bcc=''*, *subject=''*, *body=''*)
Creates a new email with the given values. If the given sender is None, one will be automatically chosen using getpass.getuser().

Parameters

- **sender** (string) – The email address of the sender.
- **to** (*string|list(string)*) – A list of email addresses, passed to set_to().
- **cc** (*string|list(string)*) – A list of email addresses, passed to set_cc().
- **bcc** (*string|list(string)*) – A list of email addresses, passed to set_bcc().
- **subject** (string) – A subject line, passed to set_subject().
- **body** (string) – The email body, passed to set_body().

add_attachment(*filename*)
Adds the file with the given name as an attachment.

Parameters **filename** (string) – A filename.

add_bcc(*bcc*)
Like add_to(), but for the 'bcc' field.

Parameters **bcc** (*string|list(string)*) – The list of email addresses.

add_cc(*cc*)
Like add_to(), but for the 'cc' field.

Parameters **cc** (*string|list(string)*) – The list of email addresses.

add_to(*to*)
Adds the given list of receipients to the 'to' field. Accepts the same argument types as set_to().

Parameters **to** (*string|list(string)*) – The list of email addresses.

get_attachments()
Returns a list of attached files.

Return type *list*[*string*]

Returns The list of filenames.

get_bcc()
Returns the value of the "bcc" field.

Return type *list*(*string*)

Returns The email addresses in the 'bcc' field.

get_body()
Returns the body of the mail.

Return type *string*

Returns The body of the mail.

get_cc()
Returns the value of the "cc" field.

Return type *list*(*string*)

Returns The email addresses in the 'cc' field.

get_receipients()
Returns a list of all receipients (to, cc, and bcc).

Return type *list*(*string*)

Returns The email addresses of all receipients.

get_sender()
Returns the value of the "From:" field.

Return type *string*

Returns The email address of the sender.

get_smtp_header()
Returns the SMTP formatted header of the line.

Return type *string*

Returns The SMTP header.

get_smtp_mail()
Returns the SMTP formatted email, as it may be passed to sendmail.

Return type *string*

Returns The SMTP formatted mail.

get_subject()
Returns the subject line.

Return type *string*

Returns The subject line.

get_to()
Returns the value of the "to" field.

Return type *list*(*string*)

Returns The email addresses in the 'to' field.

set_bcc(*bcc*)
Like set_to(), but for the 'bcc' field.

Parameters **bcc** (*string|list(string)*) – The email addresses for the 'bcc' field.

set_body(*body*)
Defines the body of the mail.

Parameters **body** (string) – The new email body.

set_cc(*cc*)
Like set_to(), but for the 'cc' field.

Parameters **cc** (*string|list(string)*) – The email addresses for the 'cc' field.

set_from_template_string(*string*)
Reads the given template (SMTP formatted) and sets all fields accordingly.

Parameters **string** (string) – The template.

set_sender(*sender*)
Defines the value of the "From:" field.

Parameters **sender** (string) – The email address of the sender.

set_subject(*subject*)
Defines the subject line.

Parameters **subject** (string) – The new subject line.

set_to(*to*)
Replaces the current list of receipients in the 'to' field by the given value. The value may be one of the following:

•A list of strings (email addresses).

•A comma separated string containing one or more email addresses.

Parameters **to** (*string|list(string)*) – The email addresses for the 'to' field.

Exscript.util.mail.**from_template**(*filename*, ***kwargs*)
Like from_template_string(), but reads the template from the file with the given name instead.

Parameters

- **filename** (string) – The name of the template file.
- **kwargs** (*str*) – Variables to replace in the template.

Return type *Mail*

Returns The resulting mail.

Exscript.util.mail.**from_template_string**(*string*, ***kwargs*)
Reads the given SMTP formatted template, and creates a new Mail object using the information.

Parameters

- **string** (*str*) – The SMTP formatted template.
- **kwargs** (*str*) – Variables to replace in the template.

Return type *Mail*

Returns The resulting mail.

Exscript.util.mail.**send**(*mail*, *server='localhost'*)
Sends the given mail.

Parameters

- **mail** (Mail) – The mail object.
- **server** (string) – The address of the mailserver.

Exscript.util.match module

Shorthands for regular expression matching.

Exscript.util.match.**any_match**(*string*, *regex*, *flags=8*)
Matches the given string against the given regex.

•If no match is found, this function returns an empty list.

•If a match is found and the regular expression has no groups,

a list of matching lines returned.

•If a match is found and the regular expression has one group,

a list of matching strings is returned.

•If a match is found and the regular expression has multiple groups,

a list containing tuples of matching strings is returned.

This behavior ensures that the following can never fail:

```
foo = '1 uno\n2 due'
for m in any_match(foo, r'aaa'):          # Returns []
    print(m)

for m in any_match(foo, r'\S+'):          # Returns ['1 uno', '2 due']
    print(m)

for m in any_match(foo, r'(aaa)'):        # Returns []
    print(m)

for m in any_match(foo, r'(\S+)'):        # Returns ['1', '2']
    print(m)

for one, two in any_match(foo, r'(aaa) (\S+)'): # Returns []
    print(m)

for one, two in any_match(foo, r'(\S+) (\S+)'): # Returns [('1', 'uno'), ('2',
↪'due')]
    print(m)
```

Parameters

- **string** (*string|Exscript.protocols.Protocol*) – The string that is matched, or a Protocol object.
- **regex** (string) – A regular expression.
- **flags** (*int*) – The flags for compiling the regex; e.g. re.I

Return type *list*[string|tuple]

Returns A list of strings, or a list of tuples.

Exscript.util.match.**first_match**(*string*, *regex*, *flags=8*)

Matches the given string against the given regex.

•If no match is found and the regular expression has zero or one

groups, this function returns None.

•If no match is found and the regular expression has more than one

group, this function returns a tuple of None. The number of elements in the tuple equals the number of groups in the regular expression.

•If a match is found and the regular expression has no groups,

the entire string is returned.

•If a match is found and the regular expression has one group,

the matching string from the group is returned.

•If a match is found and the regular expression has multiple groups,

a tuple containing the matching strings from the groups is returned.

This behavior ensures that the following assignments can never fail:

```
foo   = 'my test'
match = first_match(foo, r'aaa')          # Returns None
match = first_match(foo, r'\S+')          # Returns 'my test'
match = first_match(foo, r'(aaa)')        # Returns None
match = first_match(foo, r'(\S+)')        # Returns 'my'
match = first_match(foo, r'(aaa) (\S+)') # Returns (None, None)
match = first_match(foo, r'(\S+) (\S+)') # Returns ('my', 'foo')
```

Parameters

- **string** (*string|Exscript.protocols.Protocol*) – The string that is matched, or a Protocol object.
- **regex** (string) – A regular expression.
- **flags** (*int*) – The flags for compiling the regex; e.g. re.I

Return type stringltuple

Returns A match, or a tuple of matches.

Exscript.util.pidutil module

Handling PID (process id) files.

Exscript.util.pidutil.**isalive**(*path*)
Returns True if the file with the given name contains a process id that is still alive. Returns False otherwise.

Parameters **path** (*str*) – The name of the pidfile.

Return type bool

Returns Whether the process is alive.

Exscript.util.pidutil.**kill**(*path*)
Kills the process, if it still exists.

Parameters **path** (*str*) – The name of the pidfile.

Exscript.util.pidutil.**read**(*path*)
Returns the process id from the given file if it exists, or None otherwise. Raises an exception for all other types of OSError while trying to access the file.

Parameters **path** (*str*) – The name of the pidfile.

Return type int or None

Returns The PID, or none if the file was not found.

Exscript.util.pidutil.**remove**(*path*)
Deletes the pidfile if it exists.

Parameters **path** (*str*) – The name of the pidfile.

Exscript.util.pidutil.**write**(*path*)
Writes the current process id to the given pidfile.

Parameters **path** (*str*) – The name of the pidfile.

Exscript.util.report module

Formatting logs into human readable reports.

Exscript.util.report.**format**(*logger*, *show_successful=True*, *show_errors=True*, *show_traceback=True*)
Prints a report of the actions that were logged by the given Logger. The report contains a list of successful actions, as well as the full error message on failed actions.

Parameters **logger** (Logger) – The logger that recorded what happened in the queue.

Return type *string*

Returns A string summarizing the status of every performed task.

Exscript.util.report.**status**(*logger*)
Creates a one-line summary on the actions that were logged by the given Logger.

Parameters **logger** (Logger) – The logger that recorded what happened in the queue.

Return type *string*

Returns A string summarizing the status.

Exscript.util.report.**summarize**(*logger*)
Creates a short summary on the actions that were logged by the given Logger.

Parameters **logger** (Logger) – The logger that recorded what happened in the queue.

Return type *string*

Returns A string summarizing the status of every performed task.

Exscript.util.sigint module

A class for catching SIGINT, such that CTRL+c works.

class Exscript.util.sigint.**SigIntWatcher**
Bases: future.types.newobject.newobject

This class solves two problems with multithreaded programs in Python:

- A signal might be delivered to any thread and
- if the thread that gets the signal is waiting, the signal is ignored (which is a bug).

This class forks and catches sigint for Exscript.

The watcher is a concurrent process (not thread) that waits for a signal and the process that contains the threads. Works on Linux, Solaris, MacOS, and AIX. Known not to work on Windows.

__init__()
Creates a child process, which returns. The parent process waits for a KeyboardInterrupt and then kills the child process.

kill()

watch()

Exscript.util.sigintcatcher module

Exscript.util.start module

Quickstart methods for the Exscript queue.

Exscript.util.start.**quickrun**(*hosts*, *func*, ***kwargs*)
A wrapper around run() that creates the account by asking the user for entering his login information.

Parameters

- **hosts** (*Host|list[Host]*) – A list of Host objects.
- **func** (*function*) – The callback function.
- **kwargs** (*dict*) – Passed to the Exscript.Queue constructor.

Exscript.util.start.**quickstart**(*hosts*, *func*, *only_authenticate=False*, ***kwargs*)
Like quickrun(), but automatically logs into the host before passing the connection to the callback function.

Parameters

- **hosts** (*Host|list[Host]*) – A list of Host objects.
- **func** (*function*) – The callback function.
- **only_authenticate** (*bool*) – don't authorize, just authenticate?
- **kwargs** (*dict*) – Passed to the Exscript.Queue constructor.

Exscript.util.start.**run**(*users*, *hosts*, *func*, ***kwargs*)
Convenience function that creates an Exscript.Queue instance, adds the given accounts, and calls Queue.run() with the given hosts and function as an argument.

If you also want to pass arguments to the given function, you may use util.decorator.bind() like this:

```
def my_callback(job, host, conn, my_arg, **kwargs):
    print(my_arg, kwargs.get('foo'))

run(account,
    host,
    bind(my_callback, 'hello', foo = 'world'),
    max_threads = 10)
```

Parameters

- **users** (*Account|list[Account]*) – The account(s) to use for logging in.
- **hosts** (*Host|list[Host]*) – A list of Host objects.
- **func** (*function*) – The callback function.
- **kwargs** (*dict*) – Passed to the Exscript.Queue constructor.

Exscript.util.start.**start**(*users*, *hosts*, *func*, *only_authenticate=False*, ***kwargs*)
Like run(), but automatically logs into the host before passing the host to the callback function.

Parameters

- **users** (*Account|list[Account]*) – The account(s) to use for logging in.
- **hosts** (*Host|list[Host]*) – A list of Host objects.
- **func** (*function*) – The callback function.

- **only_authenticate** (*bool*) – don't authorize, just authenticate?
- **kwargs** (*dict*) – Passed to the Exscript.Queue constructor.

Exscript.util.syslog module

Send messages to a syslog server.

Exscript.util.syslog.**netlog**(*message*, *source=None*, *host='localhost'*, *port=514*, *priority=7*, *facility=8*)

Python's built in syslog module does not support networking, so this is the alternative. The source argument specifies the message source that is documented on the receiving server. It defaults to "scriptname[pid]", where "scriptname" is sys.argv[0], and pid is the current process id. The priority and facility arguments are equivalent to those of Python's built in syslog module.

Parameters

- **source** (*str*) – The source address.
- **host** (*str*) – The IP address or hostname of the receiving server.
- **port** (*str*) – The TCP port number of the receiving server.
- **priority** (*int*) – The message priority.
- **facility** (*int*) – The message facility.

Exscript.util.template module

Executing Exscript templates on a connection.

Exscript.util.template.**eval**(*conn*, *string*, *strip_command=True*, ***kwargs*)

Compiles the given template and executes it on the given connection. Raises an exception if the compilation fails.

if strip_command is True, the first line of each response that is received after any command sent by the template is stripped. For example, consider the following template:

```
ls -1{extract /(\S+)/ as filenames}
{loop filenames as filename}
    touch $filename
{end}
```

If strip_command is False, the response, (and hence, the 'filenames' variable) contains the following:

```
ls -1
myfile
myfile2
[...]
```

By setting strip_command to True, the first line is ommitted.

Parameters

- **conn** (Exscript.protocols.Protocol) – The connection on which to run the template.
- **string** (string) – The template to compile.
- **strip_command** (*bool*) – Whether to strip the command echo from the response.

- **kwargs** (*dict*) – Variables to define in the template.

Return type dict

Returns The variables that are defined after execution of the script.

Exscript.util.template.**eval_file**(*conn*, *filename*, *strip_command=True*, ***kwargs*)
Convenience wrapper around eval() that reads the template from a file instead.

Parameters

- **conn** (Exscript.protocols.Protocol) – The connection on which to run the template.
- **filename** (string) – The name of the template file.
- **strip_command** (*bool*) – Whether to strip the command echo from the response.
- **kwargs** (*dict*) – Variables to define in the template.

Exscript.util.template.**paste**(*conn*, *string*, ***kwargs*)
Compiles the given template and executes it on the given connection. This function differs from eval() such that it does not wait for a prompt after sending each command to the connected host. That means that the script can no longer read the response of the host, making commands such as 'extract' or 'set_prompt' useless.

The function raises an exception if the compilation fails, or if the template contains a command that requires a response from the host.

Parameters

- **conn** (Exscript.protocols.Protocol) – The connection on which to run the template.
- **string** (string) – The template to compile.
- **kwargs** (*dict*) – Variables to define in the template.

Return type dict

Returns The variables that are defined after execution of the script.

Exscript.util.template.**paste_file**(*conn*, *filename*, ***kwargs*)
Convenience wrapper around paste() that reads the template from a file instead.

Parameters

- **conn** (Exscript.protocols.Protocol) – The connection on which to run the template.
- **filename** (string) – The name of the template file.
- **kwargs** (*dict*) – Variables to define in the template.

Exscript.util.template.**test**(*string*, ***kwargs*)
Compiles the given template, and raises an exception if that failed. Does nothing otherwise.

Parameters

- **string** (string) – The template to compile.
- **kwargs** (*dict*) – Variables to define in the template.

Exscript.util.template.**test_file**(*filename*, ***kwargs*)
Convenience wrapper around test() that reads the template from a file instead.

Parameters

- **filename** (string) – The name of the template file.

- **kwargs** (*dict*) – Variables to define in the template.

Exscript.util.template.**test_secure**(*string*, ***kwargs*)

Like test(), but makes sure that each function that is used in the template has the Exscript.stdlib.util.safe_function decorator. Raises Exscript.interpreter.Exception.PermissionError if any function lacks the decorator.

Parameters

- **string** (*string*) – The template to compile.
- **kwargs** (*dict*) – Variables to define in the template.

Exscript.util.tty module

TTY utilities.

Exscript.util.tty.**get_terminal_size**(*default_rows=25*, *default_cols=80*)

Returns the number of lines and columns of the current terminal. It attempts several strategies to determine the size and if all fail, it returns (80, 25).

Return type int, int

Returns The rows and columns of the terminal.

Exscript.util.url module

Working with URLs (as used in URL formatted hostnames).

class Exscript.util.url.**Url**

Bases: future.types.newobject.newobject

Represents a URL.

__init__()

static from_string(*url*, *default_protocol=u'telnet'*)

Parses the given URL and returns an URL object. There are some differences to Python's built-in URL parser:

•It is less strict, many more inputs are accepted. This is necessary to allow for passing a simple hostname as a URL.

•You may specify a default protocol that is used when the http:// portion is missing.

•The port number defaults to the well-known port of the given protocol.

•The query variables are parsed into a dictionary (Url.vars).

Parameters

- **url** (*str*) – A URL.
- **default_protocol** (string) – A protocol name.

Return type *Url*

Returns The Url object contructed from the given URL.

to_string()

Returns the URL, including all attributes, as a string.

Return type str

Returns A URL.

Exscript.util.weakmethod module

Weak references to bound and unbound methods.

exception Exscript.util.weakmethod.**DeadMethodCalled**
Bases: exceptions.Exception

Raised by *WeakMethod* if it is called when the referenced object is already dead.

class Exscript.util.weakmethod.**WeakMethod**(*name*, *callback*)
Bases: future.types.newobject.newobject

Do not create this class directly; use *ref()* instead.

__init__(*name*, *callback*)
Constructor. Do not use directly, use *ref()* instead.

callback

get_function()
Returns the referenced method/function if it is still alive. Returns None otherwise.

Return type callable|None

Returns The referenced function if it is still alive.

isalive()
Returns True if the referenced function is still alive, False otherwise.

Return type bool

Returns Whether the referenced function is still alive.

name

Exscript.util.weakmethod.**ref**(*function*, *callback=None*)
Returns a weak reference to the given method or function. If the callback argument is not None, it is called as soon as the referenced function is garbage deleted.

Parameters

- **function** (*callable*) – The function to reference.
- **callback** (*callable*) – Called when the function dies.

Submodules

Exscript.account module

Manages user accounts.

class Exscript.account.**Account**(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Bases: future.types.newobject.newobject

This class represents a user account.

__init__(*name*, *password=''*, *password2=None*, *key=None*, *needs_lock=True*)
Constructor.

The authorization password is only required on hosts that separate the authentication from the authorization procedure. If an authorization password is not given, it defaults to the same value as the authentication password.

If the *needs_lock* argument is set to True, we ensure that no two threads can use the same account at the same time. You will want to use this setting if you are using a central authentication server that allows for only one login to happen at a time. Note that you will still be able to open multiple sessions at the time. It is only the authentication procedure that will not happen in parallel; once the login is complete, other threads can use the account again. In other words, the account is only locked during calls to `protocols.Protocol.login()` and the **authenticate** methods.

Warning: Setting *lock* to True drastically degrades performance!

Parameters

- **name** (*str*) – A username.
- **password** (*str*) – The authentication password.
- **password2** (*str*) – The authorization password, if required.
- **key** (PrivateKey) – A private key, if required.
- **needs_lock** (*bool*) – True if the account will be locked during login.

acquire(*signal=True*)
Locks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the acquired_event signal.

context()
When you need a 'with' context for an already-acquired account.

get_authorization_password()
Returns the authorization password of the account.

Return type *string*

Returns The account password.

get_key()
Returns the key of the account, if any.

Return type PrivateKey|None

Returns A key object.

get_name()
Returns the name of the account.

Return type *string*

Returns The account name.

get_password()
Returns the password of the account.

Return type *string*

Returns The account password.

release(*signal=True*)
Unlocks the account. Method has no effect if the constructor argument *needs_lock* wsa set to False.

Parameters **signal** (*bool*) – Whether to emit the released_event signal.

set_authorization_password(*password*)
Changes the authorization password of the account.

Parameters **password** (string) – The new authorization password.

set_name(*name*)
Changes the name of the account.

Parameters **name** (string) – The account name.

set_password(*password*)
Changes the password of the account.

Parameters **password** (string) – The account password.

class Exscript.account.**AccountManager**
Bases: future.types.newobject.newobject

Keeps track of available user accounts and assigns them to the worker threads.

__init__()
Constructor.

acquire_account(*account=None*, *owner=None*)
Acquires the given account. If no account is given, one is chosen from the default pool.

Parameters

- **account** (Account) – The account that is added.
- **owner** (*object*) – An optional descriptor for the owner.

Return type *Account*

Returns The account that was acquired.

acquire_account_for(*host*, *owner=None*)
Acquires an account for the given host and returns it. The host is passed to each of the match functions that were passed in when adding the pool. The first pool for which the match function returns True is chosen to assign an account.

Parameters

- **host** (Host) – The host for which an account is acquired.
- **owner** (*object*) – An optional descriptor for the owner.

Return type *Account*

Returns The account that was acquired.

add_account(*account*)
Adds the given account to the default account pool that Exscript uses to log into all hosts that have no specific *Account* attached.

Parameters **account** (Account) – The account that is added.

add_pool(*pool*, *match=None*)
Adds a new account pool. If the given match argument is None, the pool the default pool. Otherwise, the match argument is a callback function that is invoked to decide whether or not the given pool should be used for a host.

When Exscript logs into a host, the account is chosen in the following order:

Exscript checks whether an account was attached to the Host object using Host.set_account()), and uses that.

If the Host has no account attached, Exscript walks through all pools that were passed to Queue.add_account_pool(). For each pool, it passes the Host to the function in the given match argument. If the return value is True, the account pool is used to acquire an account. (Accounts within each pool are taken in a round-robin fashion.)

If no matching account pool is found, an account is taken from the default account pool.

Finally, if all that fails and the default account pool contains no accounts, an error is raised.

Example usage:

```
def do_nothing(conn):
    conn.autoinit()

def use_this_pool(host):
    return host.get_name().startswith('foo')

default_pool = AccountPool()
default_pool.add_account(Account('default-user', 'password'))

other_pool = AccountPool()
other_pool.add_account(Account('user', 'password'))

queue = Queue()
queue.account_manager.add_pool(default_pool)
queue.account_manager.add_pool(other_pool, use_this_pool)

host = Host('localhost')
queue.run(host, do_nothing)
```

In the example code, the host has no account attached. As a result, the queue checks whether use_this_pool() returns True. Because the hostname does not start with 'foo', the function returns False, and Exscript takes the 'default-user' account from the default pool.

Parameters

- **pool** (AccountPool) – The account pool that is added.
- **match** (*callable*) – A callback to check if the pool should be used.

get_account_from_hash(*account_hash*)
Returns the account with the given hash, if it is contained in any of the pools. Returns None otherwise.

Parameters **account_hash** (*str*) – The hash of an account object.

release_accounts(*owner*)
Releases all accounts that were acquired by the given owner.

Parameters **owner** (*object*) – The owner descriptor as passed to acquire_account().

reset()
Removes all account pools.

class Exscript.account.**AccountPool**(*accounts=None*)
Bases: future.types.newobject.newobject

This class manages a collection of available accounts.

__init__(*accounts=None*)
Constructor.

Parameters **accounts** (*Account | list[Account]*) – Passed to add_account()

acquire_account(*account=None*, *owner=None*)
Waits until an account becomes available, then locks and returns it. If an account is not passed, the next available account is returned.

Parameters

- **account** (Account) – The account to be acquired, or None.
- **owner** (*object*) – An optional descriptor for the owner.

Return type *Account*

Returns The account that was acquired.

add_account(*accounts*)
Adds one or more account instances to the pool.

Parameters **accounts** (*Account | list[Account]*) – The account to be added.

get_account_from_hash(*account_hash*)
Returns the account with the given hash, or None if no such account is included in the account pool.

get_account_from_name(*name*)
Returns the account with the given name.

Parameters **name** (string) – The name of the account.

has_account(*account*)
Returns True if the given account exists in the pool, returns False otherwise.

Parameters **account** (Account) – The account object.

n_accounts()
Returns the number of accounts that are currently in the pool.

release_accounts(*owner*)
Releases all accounts that were acquired by the given owner.

Parameters **owner** (*object*) – The owner descriptor as passed to acquire_account().

reset()
Removes all accounts.

class Exscript.account.**AccountProxy**(*parent*)
Bases: future.types.newobject.newobject

An object that has a 1:1 relation to an account object in another process.

__init__(*parent*)
Constructor.

Parameters **parent** (*multiprocessing.Connection*) – A pipe to the associated account manager.

acquire()
Locks the account. Returns True on success, False if the account is thread-local and must not be locked.

context()
When you need a 'with' context for an already-acquired account.

static for_account_hash(*parent*, *account_hash*)
Returns a new AccountProxy that acquires the account with the given hash, if such an account is known to the account manager. It is an error if the account manager does not have such an account.

static for_host(*parent*, *host*)
Returns a new AccountProxy that has an account acquired. The account is chosen based on what the connected AccountManager selects for the given host.

static for_random_account(*parent*)
Returns a new AccountProxy that has an account acquired. The account is chosen by the connected AccountManager.

get_authorization_password()
Returns the authorization password of the account.

Return type *string*

Returns The account password.

get_key()
Returns the key of the account, if any.

Return type PrivateKey|None

Returns A key object.

get_name()
Returns the name of the account.

Return type *string*

Returns The account name.

get_password()
Returns the password of the account.

Return type *string*

Returns The account password.

release()
Unlocks the account.

class Exscript.account.**LoggerProxy**(*parent*, *logger_id*)
Bases: future.types.newobject.newobject

An object that has a 1:1 relation to a Logger object in another process.

__init__(*parent*, *logger_id*)
Constructor.

Parameters **parent** (*multiprocessing.Connection*) – A pipe to the associated pipe handler.

add_log(*job_id*, *name*, *attempt*)

log(*job_id*, *message*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

Exscript.host module

Representing a device to connect with.

class Exscript.host.**Host**(*uri*, *default_protocol='telnet'*)
Bases: future.types.newobject.newobject

Represents a device on which to open a connection.

__init__(*uri*, *default_protocol='telnet'*)
Constructor. The given uri is passed to Host.set_uri(). The default_protocol argument defines the protocol that is used in case the given uri does not specify it.

Parameters

- **uri** (string) – A hostname; see set_uri() for more info.
- **default_protocol** (string) – The protocol name.

account

address

append(*name*, *value*)
Appends the given value to the list variable with the given name.

Parameters

- **name** (string) – The name of the variable.
- **value** (*object*) – The appended value.

get(*name*, *default=None*)
Returns the value of the given variable, or the given default value if the variable is not defined.

Parameters

- **name** (string) – The name of the variable.
- **default** (*object*) – The default value.

Return type object

Returns The value of the variable.

get_account()
Returns the account that is used to log in.

Return type *Account*

Returns The account.

get_address()
Returns the address that is used to open the connection.

Return type *string*

Returns The address that is used to open the connection.

get_all()
Returns a dictionary containing all variables.

Return type dict

Returns The dictionary with the variables.

get_dict()
Returns a dict containing the host's attributes. The following keys are contained:

•hostname

•address

•protocol

•port

Return type dict

Returns The resulting dictionary.

get_name()
Returns the name.

Return type *string*

Returns The hostname excluding the name.

get_option(*name*, *default=None*)
Returns the value of the given option if it is defined, returns the given default value otherwise.

Parameters

- **name** (*str*) – The option name.
- **default** (*object*) – A default value.

get_options()
Return a dictionary containing all defined options.

Return type dict

Returns The options.

get_protocol()
Returns the name of the protocol.

Return type *string*

Returns The protocol name.

get_tcp_port()
Returns the TCP port number.

Return type *string*

Returns The TCP port number.

get_uri()
Returns a URI formatted representation of the host, including all of it's attributes except for the name. Uses the address, not the name of the host to build the URI.

Return type str

Returns A URI.

has_key(*name*)
Returns True if the variable with the given name is defined, False otherwise.

Parameters **name** (string) – The name of the variable.

Return type bool

Returns Whether the variable is defined.

name

options

protocol

set (*name*, *value*)
Stores the given variable/value in the object for later retrieval.

Parameters

- **name** (*string*) – The name of the variable.
- **value** (*object*) – The value of the variable.

set_account (*account*)
Defines the account that is used to log in.

Parameters **account** (*Exscript.Account*) – The account.

set_address (*address*)
Set the address of the remote host the is contacted, without changing hostname, username, password, protocol, and TCP port number. This is the actual address that is used to open the connection.

Parameters **address** (*string*) – A hostname or IP name.

set_all (*variables*)
Like set(), but replaces all variables by using the given dictionary. In other words, passing an empty dictionary results in all variables being removed.

Parameters **variables** (*dict*) – The dictionary with the variables.

set_default (*name*, *value*)
Like set(), but only sets the value if the variable is not already defined.

Parameters

- **name** (*string*) – The name of the variable.
- **value** (*object*) – The value of the variable.

set_name (*name*)
Set the hostname of the remote host without changing username, password, protocol, and TCP port number.

Parameters **name** (*string*) – A hostname or IP address.

set_option (*name*, *value*)
Defines a (possibly protocol-specific) option for the host. Possible options include:

verify_fingerprint: bool

Parameters

- **name** (*str*) – The option name.
- **value** (*object*) – The option value.

set_protocol (*protocol*)
Defines the protocol. The following protocols are currently supported:

•telnet: Telnet

•ssh1: SSH version 1

•ssh2: SSH version 2

•ssh: Automatically selects the best supported SSH version

•dummy: A virtual device that accepts any command, but that does not respond anything useful.

•pseudo: A virtual device that loads a file with the given "hostname". The given file is a Python file containing information on how the virtual device shall respond to commands. For more information please refer to the documentation of protocols.Dummy.load_command_handler_from_file().

Parameters **protocol** (string) – The protocol name.

set_tcp_port (*tcp_port*)
Defines the TCP port number.

Parameters **tcp_port** (*int*) – The TCP port number.

set_uri (*uri*)
Defines the protocol, hostname/address, TCP port number, username, and password from the given URL. The hostname may be URL formatted, so the following formats are all valid:

•myhostname

•myhostname.domain

•ssh:hostname

•ssh:hostname.domain

•ssh://hostname

•ssh://user@hostname

•ssh://user:password@hostname

•ssh://user:password@hostname:21

For a list of supported protocols please see set_protocol().

Parameters **uri** (string) – An URL formatted hostname.

tcp_port

vars

Exscript.key module

Represents a private key.

class Exscript.key.**PrivateKey** (*keytype='rsa'*)
Bases: future.types.newobject.newobject

Represents a cryptographic key, and may be used to authenticate useing *Exscript.protocols*.

__init__ (*keytype='rsa'*)
Constructor. Supported key types are provided by their respective protocol adapters and can be retrieved from the PrivateKey.keytypes class attribute.

Parameters **keytype** (string) – The key type.

static from_file (*filename*, *password=''*, *keytype=None*)
Returns a new PrivateKey instance with the given attributes. If keytype is None, we attempt to automatically detect the type.

Parameters

- **filename** (string) – The key file name.
- **password** (string) – The key password.

- **keytype** (string) – The key type.

Return type *PrivateKey*

Returns The new key.

get_filename()
Returns the name of the key file.

Return type *string*

Returns The key password.

get_password()
Returns the password for the key.

Return type *string*

Returns The key password.

get_type()
Returns the type of the key, e.g. RSA or DSA.

Return type *string*

Returns The key type

keytypes = set([u'rsa', u'dss'])

set_filename(*filename*)
Sets the name of the key file to use.

Parameters **filename** (string) – The key filename.

set_password(*password*)
Defines the password used for decrypting the key.

Parameters **password** (string) – The key password.

Exscript.logger module

Logging to memory.

class Exscript.logger.**FileLogger**(*logdir*, *mode=u'a'*, *delete=False*, *clearmem=True*)
Bases: *Exscript.logger.Logger*

A Logger that stores logs into files.

__init__(*logdir*, *mode=u'a'*, *delete=False*, *clearmem=True*)
The logdir argument specifies the location where the logs are stored. The mode specifies whether to append the existing logs (if any). If delete is True, the logs are deleted after they are completed, unless they have an error in them. If clearmem is True, the logger does not store a reference to the log in it. If you want to use the functions from *Exscript.util.report* with the logger, clearmem must be False.

add_log(*job_id*, *name*, *attempt*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

class Exscript.logger.**Log**(*name*)
Bases: future.types.newobject.newobject

__init__(*name*)

aborted(*exc_info*)
Called by a logger to log an exception.

get_error(*include_tb=True*)

get_name()

has_ended()

has_error()

started()
Called by a logger to inform us that logging may now begin.

succeeded()
Called by a logger to inform us that logging is complete.

write(**data*)

class Exscript.logger.**Logfile**(*name*, *filename*, *mode=u'a'*, *delete=False*)
Bases: *Exscript.logger.Log*

This class logs to two files: The raw log, and sometimes a separate log containing the error message with a traceback.

__init__(*name*, *filename*, *mode=u'a'*, *delete=False*)

aborted(*exc_info*)

started()

succeeded()

write(**data*)

class Exscript.logger.**Logger**
Bases: future.types.newobject.newobject

A QueueListener that implements logging for the queue. Logs are kept in memory, and not written to the disk.

__init__()
Creates a new logger instance. Use the *Exscript.util.log.log_to* decorator to send messages to the logger.

add_log(*job_id*, *name*, *attempt*)

get_aborted_actions()
Returns the number of jobs that were aborted.

get_aborted_logs()

get_logs()

get_succeeded_actions()
Returns the number of jobs that were completed successfully.

get_succeeded_logs()

log(*job_id*, *message*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

class Exscript.logger.**LoggerProxy**(*parent*, *logger_id*)
Bases: future.types.newobject.newobject

An object that has a 1:1 relation to a Logger object in another process.

__init__(*parent*, *logger_id*)
Constructor.

Parameters **parent** (*multiprocessing.Connection*) – A pipe to the associated pipe handler.

add_log(*job_id*, *name*, *attempt*)

log(*job_id*, *message*)

log_aborted(*job_id*, *exc_info*)

log_succeeded(*job_id*)

Exscript.queue module

The heart of Exscript.

class Exscript.queue.**Queue**(*domain=''*, *verbose=1*, *mode='threading'*, *max_threads=1*, *host_driver=None*, *exc_cb=None*, *stdout=<open file '<stdout>', mode 'w'>*, *stderr=<open file '<stderr>', mode 'w'>*)
Bases: future.types.newobject.newobject

Manages hosts/tasks, accounts, connections, and threads.

__init__(*domain=''*, *verbose=1*, *mode='threading'*, *max_threads=1*, *host_driver=None*, *exc_cb=None*, *stdout=<open file '<stdout>', mode 'w'>*, *stderr=<open file '<stderr>', mode 'w'>*)
Constructor. All arguments should be passed as keyword arguments. Depending on the verbosity level, the following types of output are written to stdout/stderr (or to whatever else is passed in the stdout/stderr arguments):

•S = status bar

•L = live conversation

•D = debug messages

•E = errors

•! = errors with tracebacks

•F = fatal errors with tracebacks

The output types are mapped depending on the verbosity as follows:

•verbose = -1: stdout = None, stderr = F

•verbose = 0: stdout = None, stderr = EF

•verbose = 1, max_threads = 1: stdout = L, stderr = EF

•verbose = 1, max_threads = n: stdout = S, stderr = EF

•verbose >= 2, max_threads = 1: stdout = DL, stderr = !F

•verbose >= 2, max_threads = n: stdout = DS, stderr = !F

Parameters

- **domain** (*str*) – The default domain of the contacted hosts.
- **verbose** (*int*) – The verbosity level.
- **mode** (*str*) – 'multiprocessing' or 'threading'

- **max_threads** (*int*) – The maximum number of concurrent threads.
- **host_driver** (*str*) – driver name like "ios" for manual override
- **exc_cb** (*func(jobname, exc_info)*) – callback function to call on exceptions
- **stdout** (file) – The output channel, defaults to sys.stdout.
- **stderr** (file) – The error channel, defaults to sys.stderr.

add_account(*account*)

Adds the given account to the default account pool that Exscript uses to log into all hosts that have no specific Account attached.

Parameters **account** (Account) – The account that is added.

add_account_pool(*pool*, *match=None*)

Adds a new account pool. If the given match argument is None, the pool the default pool. Otherwise, the match argument is a callback function that is invoked to decide whether or not the given pool should be used for a host.

When Exscript logs into a host, the account is chosen in the following order:

Exscript checks whether an account was attached to the Host object using Host.set_account()), and uses that.

If the Host has no account attached, Exscript walks through all pools that were passed to *Queue.add_account_pool()*. For each pool, it passes the Host to the function in the given match argument. If the return value is True, the account pool is used to acquire an account. (Accounts within each pool are taken in a round-robin fashion.)

If no matching account pool is found, an account is taken from the default account pool.

Finally, if all that fails and the default account pool contains no accounts, an error is raised.

Example usage:

```
def do_nothing(conn):
    conn.autoinit()

def use_this_pool(host):
    return host.get_name().startswith('foo')

default_pool = AccountPool()
default_pool.add_account(Account('default-user', 'password'))

other_pool = AccountPool()
other_pool.add_account(Account('user', 'password'))

queue = Queue()
queue.add_account_pool(default_pool)
queue.add_account_pool(other_pool, use_this_pool)

host = Host('localhost')
queue.run(host, do_nothing)
```

In the example code, the host has no account attached. As a result, the queue checks whether use_this_pool() returns True. Because the hostname does not start with 'foo', the function returns False, and Exscript takes the 'default-user' account from the default pool.

Parameters

- **pool** (AccountPool) – The account pool that is added.

- **match** (*callable*) – A callback to check if the pool should be used.

destroy (*force=False*)
Like shutdown(), but also removes all accounts, hosts, etc., and does not restart the queue. In other words, the queue can no longer be used after calling this method.

Parameters **force** (*bool*) – Whether to wait until all jobs were processed.

enqueue (*function*, *name=None*, *attempts=1*)
Places the given function in the queue and calls it as soon as a thread is available. To pass additional arguments to the callback, use Python's functools.partial().

Parameters

- **function** (*function*) – The function to execute.
- **name** (*string*) – A name for the task.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

force_run (*hosts*, *function*, *attempts=1*)
Like priority_run(), but starts the task immediately even if that max_threads is exceeded.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

get_max_threads ()
Returns the maximum number of concurrent threads.

Return type int

Returns The maximum number of connections.

get_progress ()
Returns the progress in percent.

Return type float

Returns The progress in percent.

is_completed ()
Returns True if the task is completed, False otherwise. In other words, this methods returns True if the queue is empty.

Return type bool

Returns Whether all tasks are completed.

join ()
Waits until all jobs are completed.

priority_run (*hosts*, *function*, *attempts=1*)
Like run(), but adds the task to the front of the queue.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

priority_run_or_raise(*hosts*, *function*, *attempts=1*)
Like priority_run(), but if a host is already in the queue, the existing host is moved to the top of the queue instead of enqueuing the new one.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns A task object, or None if all hosts were duplicates.

reset()
Remove all accounts, hosts, etc.

run(*hosts*, *function*, *attempts=1*)
Add the given function to a queue, and call it once for each host according to the threading options. Use decorators.bind() if you also want to pass additional arguments to the callback function.

Returns an object that represents the queued task, and that may be passed to is_completed() to check the status.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns An object representing the task.

run_or_ignore(*hosts*, *function*, *attempts=1*)
Like run(), but only appends hosts that are not already in the queue.

Parameters

- **hosts** (*string|list(string)|Host|list(Host)*) – A hostname or Host object, or a list of them.
- **function** (*function*) – The function to execute.
- **attempts** (*int*) – The number of attempts on failure.

Return type object

Returns A task object, or None if all hosts were duplicates.

set_max_threads (*n_connections*)
Sets the maximum number of concurrent connections.

Parameters **n_connections** (*int*) – The maximum number of connections.

shutdown (*force=False*)
Stop executing any further jobs. If the force argument is True, the function does not wait until any queued jobs are completed but stops immediately.

After emptying the queue it is restarted, so you may still call run() after using this method.

Parameters **force** (*bool*) – Whether to wait until all jobs were processed.

Exscript.version module

Warning: This file is automatically generated.

Python Module Index

e

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z