

Scripts

Table of contents

1 Jython and Python.....	2
2 Scripting for The Grinder.....	2
2.1 Script structure.....	2
2.2 Canonical test script structure.....	3
2.3 Automatically generating scripts.....	3
3 Tests.....	3
4 The Grinder script API.....	4
5 The Jython distribution and installation.....	5
5.1 Setting the Jython cache directory.....	5

This section describes The Grinder 3 scripting API. If you've used The Grinder 2 for HTTP testing and you're not a programmer, you might be a bit daunted. Don't worry, it's just as easy to record and replay HTTP scripts with The Grinder 3.

1. Jython and Python

The Grinder 3 scripting engine is Jython - the Java implementation of Python. Python is powerful, popular and easy on the eye. If you've never seen any Python before, don't panic. Take a look at the [script gallery](#) (`../g3/script-gallery.html`) and Richard Perks' [tutorial](#) (`../g3/tutorial-perks.html`) to get a taste of what its like. There are plenty of resources on the web, here are a few of them to get you started:

- [The Jython home page](http://www.jython.org/) (`http://www.jython.org/`)
- [The Python language web site](http://www.python.org/) (`http://www.python.org/`)
- [Ten Python pitfalls](http://zephyrfalcon.org/labs/python_pitfalls.html) (`http://zephyrfalcon.org/labs/python_pitfalls.html`)

I recommend the [Jython Essentials](#)

(`http://www.amazon.com/exec/obidos/tg/detail/-/0596002475/qid%3D1044795121/103-7145719-311822`) book; you can read the [introductory chapter](#) (`http://www.oreilly.com/catalog/jythoness/chapter/ch01.html`) for free.

2. Scripting for The Grinder

2.1. Script structure

Scripts must conform to a few conventions in order to work with The Grinder framework. I'll lay the rules out in fairly dry terms before proceeding with an example. Don't worry if this makes no sense to you at first, the examples are much easier to comprehend.

1. Scripts must define a class called `TestRunner`

When a worker process starts up it runs the test script once. The test script must define a class called `TestRunner`. The Grinder engine then creates an instance of `TestRunner` for each worker thread. A thread's `TestRunner` instance can be used to store information specific to that thread.

Note:

Although strongly recommended, strictly `TestRunner` doesn't need to be a class. See the [Hello World with Functions](#) (`../g3/script-gallery.html#helloworldfunctions`) example.

2. The `TestRunner` instance must be callable

A Python object is callable if it defines a `__call__` method. Each worker thread performs a number of *runs* of the test script, as configured by the property `grinder.runs`. For each run, the worker thread calls its `TestRunner`; thus the `__call__` method can be thought of as the definition of a run.

3. The test script can access services through the `grinder` object

The engine makes an object called `grinder` available for the script to import. It can also be imported by any modules that the script calls. This is an instance of the [Grinder.ScriptContext](#) (`script-javadoc/net/grinder/script/Grinder.ScriptContext.html`) class and provides access to context information (such as the worker thread ID) and

services (such as logging and statistics).

2.2. Canonical test script structure

This is an example of a script that conforms to the rules above. It doesn't do very much - every run will log *Hello World* to the output log.

```
from net.grinder.script.Grinder import grinder

# An instance of this class is created for every thread.
class TestRunner:
    # This method is called for every run.
    def __call__(self):
        # Per thread scripting goes here.
        grinder.logger.output("Hello World")
```

2.3. Automatically generating scripts

If you are creating a script for a website or web application, you can use the [TCPProxy](#) (`../g3/tcpproxy.html#HTTPPluginTCPProxyFilter`) to generate an HTTPPlugin script suitable for use with The Grinder.

3. Tests

Although our simple test script can be used with The Grinder framework and can easily be started in many times in many worker processes on many machines, it doesn't report any statistics. For this we need to create some tests. A [Test](#) (`script-javadoc/net/grinder/script/Test.html`) has a unique test number and description. If you are using the [console](#) (`../g3/console.html`), it will automatically update to display new Tests as they are created.

Let's add a Test to our script.

```
from net.grinder.script import Test
from net.grinder.script.Grinder import grinder

# Create a Test with a test number and a description.
test1 = Test(1, "Log method")

class TestRunner:
    def __call__(self):
        log("Hello World")
```

Here we have created a single Test with the test number *1* and the description *Log method*. Note how must import the Test class in a similar manner to Java. We've also explicitly imported the grinder object, this is good practice as it allows our script to be called from other scripts as a Python *module*.

Now the console knows about our Test, but we're still not using it to record anything. Let's record how long our `grinder.logger.output` method takes. To instrument the `grinder.logger.output` method we use our Test to wrap it with a proxy *wrapper*. The wrapper object looks exactly like the `grinder.logger.output` method and can be called in the same manner. If we call the wrapper the call will be delegated through to the `grinder.logger.output` method but additionally the time taken to do the call and the number of calls will be recorded and reported to the console.

```

from net.grinder.script import Test
from net.grinder.script.Grinder import grinder

test1 = Test(1, "Log method")

# Wrap the log() method with our Test and call the result logWrapper.
logWrapper = test1.wrap(grinder.logger.output)

class TestRunner:
    def __call__(self):
        logWrapper("Hello World")

```

This is a fully functional test script that works within The Grinder framework and reports results to the console.

You're not restricted to wrapping method calls. In fact, its more common to wrap objects. Here's an example using The Grinder's [HTTP plug-in](#) (`../g3/http-plugin.html`) .

```

# A simple example using the HTTP plugin that shows the retrieval of a
# single page via HTTP.

from net.grinder.script import Test
from net.grinder.script.Grinder import grinder
from net.grinder.plugin.http import HTTPRequest

test1 = Test(1, "Request resource")
request1 = test1.wrap(HTTPRequest())

class TestRunner:
    def __call__(self):
        result = request1.GET("http://localhost:7001/")

```

4. The Grinder script API

With what you've seen already you have the full power of Jython at your finger tips. You can use practically *any* Java or Python code in your test scripts.

The Grinder script API can be used to access services from The Grinder. The [Javadoc](#) (`script-javadoc/index.html`) contains full information on all the packages, classes and interfaces that make up the core API, as well as additional packages added by the shipped plug-ins. This section provides overview information on various areas of the API. See also the [HTTP plugin documentation](#) (`../g3/http-plugin.html`) .

The [net.grinder.script](#) (`script-javadoc/net/grinder/script/package-summary.html`) package

An instance of [Grinder.ScriptContext](#) (`script-javadoc/net/grinder/script/Grinder.ScriptContext.html`) called `grinder` is automatically available to all scripts. This object provides access to context information and acts a starting point for accessing other services. The instance can be explicitly imported from other Python modules as `net.grinder.script.Grinder.grinder`.

We have described the use of the [Test](#) (`script-javadoc/net/grinder/script/Test.html`) class [above](#).

The [Statistics](#) (`script-javadoc/net/grinder/script/Statistics.html`) interface allows scripts to query and modify [statistics](#) (`../g3/statistics.html`) , provide custom statistics, and register additional views of standard and custom statistics.

The [net.grinder.common](#) (script-javadoc/net/grinder/common/package-summary.html) package

This package contains common interfaces and utility classes that are used throughout The Grinder and that are also useful to scripts.

5. The Jython distribution and installation

The Grinder is shipped with a version of Jython but does not package the Jython distribution of the standard Python library. If you want to use the standard library, or if you want to use a different version of Jython, obtain and install Jython and tell The Grinder where you installed it. You can do this either by adding the following to your [properties](#) (../g3/properties.html) file:

```
grinder.jvm.arguments = -Dpython.home=/opt/jython/jython2.2.1
```

or on the agent command line:

```
java -Dgrinder.jvm.arguments=-Dpython.home=/opt/jython/jython2.2.1
net.grinder.Grinder
```

In both cases, change /opt/jython/jython2.2.1 to the directory in which you installed Jython. You must install Jython on all of the agent machines. If the version of Jython is different to that included with The Grinder (2.2.1), you should also add the installation's jython.jar to the start of the CLASSPATH used to launch the agent.

Note:

Adding the installation's jython.jar to the start of the CLASSPATH used to be sufficient for Jython to calculate its install directory, but this no longer appears to work with Jython 2.2.1. You must explicitly set python.home as described above.

Jython picks up user and site preferences from several sources (see <http://www.jython.org/docs/registry.html>). A side effect of setting python.home is that the installed registry file will be used.

5.1. Setting the Jython cache directory

Another feature that is not working in later versions of Jython is the correct calculation of the cache directory. If you don't have a Jython cache directory, wild card imports of Java packages (e.g. from java.util import *) may not work, The Grinder will take a little longer to start, and ugly error messages will be displayed:

```
28/09/08 17:57:11 (agent): worker paston01-0 started
*sys-package-mgr*: can't create package cache dir,
'/home/philipa/performance/round2/grinder-3.1/lib/jython.jar/cachedir/packages'
```

You can specify the cache directory either by setting the python.home as above (in which case the directory will that specified in the Python registry), or by setting the Java property python.cachedir in your [properties](#) (../g3/properties.html) file:

```
grinder.jvm.arguments = -Dpython.cachedir=/tmp/mycache
```

or on the command line:

```
java -Dgrinder.jvm.arguments = -Dpython.cachedir=/tmp/mycache  
net.grinder.Grinder
```

You can only set `grinder.jvm.arguments` once, so if you want to set both the cache directory and `python.home` either use the registry, or do this:

```
grinder.jvm.arguments = -Dpython.home=/opt/jython/jython2.2.1  
-Dpython.cachedir=/tmp/mycache
```