

---

# Getting Started with Chimera

*Release 0.1*

**Paulo Henrique Silva**

February 17, 2009



# CONTENTS

<b>1 Introduction</b>	<b>1</b>
1.1 Chimera objects . . . . .	1



# INTRODUCTION

Chimera uses a client/server model coupled with remote procedure call (RPC) system. Chimera defines all its entities in terms of objects.

To be a valid Chimera object a class must extend `ChimeraObject` class. `ChimeraObject` class implements `ILifeCycle` interface which defines basic methods every object must have to be started and stopped. `ChimeraObject` provides a basic implementation of a control loop, a common structure in control programs.

By itself, a Chimera object is just a static bunch of code that inherits from a specific class. To be useful, every `ChimeraObject` must have a `Manager` to manage its life cycle.

**Note:** We call it instrument, controller or driver purely from a semantic point of view, as they are equal from a code point of view, there is no different base class for different kinds of objects. Every instrument, controller or driver extends `ChimeraObject`. More specifically, it must implement `ILifeCycle`, and `ChimeraObject` provides us with a basic implementation.

`Manager` class is responsible for object initialization, life cycle (start, stop) and proxy creation. `Manager` is also our server in the client/server model. It's a server of objects. We can think of `Manager` as a pool of objects available to be used.

As we need RPC support for every object and want to make the system easy to use and write, we use a Proxy class that handles all the networking for us. This way, you don't have to write networking code on your objects, just the real action, Proxy and friends add networking for you.

Before describing `Manager`'s responsibilities in more detail, let's describe all features we can have in a Chimera object.

## 1.1 Chimera objects

A Chimera object, as already said, is a normal Python class which extends from a specific base class, `ChimeraObject`, the simplest `ChimeraObject` is the following.

```
from chimera.core.chimeraobject import ChimeraObject

class Simplest (ChimeraObject):

    def __init__ (self):
        ChimeraObject.__init__(self)
```

this object has no methods, configuration or events, so it's the simplest and dumbest possible object.

As Python doesn't call base constructors per se, you need to call the base constructor from Simplest constructor (`__init__()` method).

Chimera uses the concept of `Location` all over the code. A `Location` is much like an URL, but without a scheme. Locations identify specific class instances running somewhere. The basic format is the following:

```
[host:port]/Classname/instance_name[?param1=value1,...]
```

*host* and *port*, optional fields, tell Chimera where to look for this particular object. *Classname* is the class name of the object and *instance\_name* the name given to a specific instance running on host:port. When you add objects to `Manager`, you must specify a name. Also, you can pass configuration parameters as comma separated param=value pairs.

Let's write down a class that uses this and see how to actually use this from Chimera.

```
from chimera.core.chimeraobject import ChimeraObject

class Example1 (ChimeraObject):

    __config__ = {"param1": "a string parameter"}

    def __init__ (self):
        ChimeraObject.__init__(self)

    def __start__ (self):
        self.doSomething("test argument")

    def doSomething (self, arg):
        self.log.warning("Hi, I'm doing something.")
        self.log.warning("My arg=%s" % arg)
        self.log.warning("My param1=%s" % self["param1"])
```

This example requires some explanations, but before, let's run it using **chimera** script. **chimera** script is a script to initialize `Manager` and add objects either from `Locations` given on command line or from a configuration file.

To follow Chimera conventions, a file with a class named `Example1` must be saved to a file name `example1.py` to allow Chimera `ClassLoader` to find it. We may simplify this in the future.

You can save this file anywhere on your system, let's suppose you saved it on your `$HOME` directory.

To run it, call **chimera** this way:

```
$ chimera -I $HOME -i /Example1/example
```

You'll see something like this:

```
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=test argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
```

You should use `Ctrl+C` (`SIGINT`) to stop **chimera**.

The `-I` on **chimera** tells Chimera where to look for instruments, this case to look in your `$HOME` directory (you can use any directory there, `.` for example). Then to `-i` we pass a valid Chimera `Location`.

From the Location, Chimera knows that you want to create an instance of `Example1` class and call this instance 'example'.

You can also pass configuration parameters right on the Location given in the command line. Use:

```
$ chimera -I $HOME -i /Example1/example?param1="Now for something different"
```

A few points need explanation on `Example1`:

1. `__config__` is a class attribute (class field in some circles) where you should pass a Python dictionary with any parameter you like to add to your object. Chimera uses the value you pass in as default value and also uses the type of it to do some type checking for you. Look at `src/chimera/core/config.py` for valid types.
2. `__start__()` method. This method is from `ILifeCycle` interface, `ChimeraObject` implementation just does nothing, here we use it to call a specific method on object initialization. Manager first call `__init__()` to create an instance, configure this instance passing any parameter you gave on command line, then call `__start__()` and when system is shutting down call `__stop__()`.
3. `log`. `ChimeraObject` implementation give a `log` attribute (instance field, in other circles) to every class, you can use this to log messages to default Chimera log system. It's a normal Python's logging logger, so consult [logging](#) for more information.
4. `self["param1"]`. You define your object parameters using `__config__` dict, but to access the actual value, you use the current object (`self`) as dictionary to access values from it. Thus, `self["param1"]` treat `self` as a dict and get key `param1` from it. For most purposes, `self` is a dict and normal dict. You can also set things, with normal `self["param1"] = "value1"`.

When you use **chimera** script, a `Manager` is created for you, but you can do it by yourself to learn how things work in Chimera. The following example is based on `server.py`.

```
from chimera.core.manager import Manager

manager = Manager(host='localhost', port=8000)
manager.addLocation("/Example1/example", start=True)

manager.wait()
```

Suppose you save it to `server.py` in the same directory where you put `example1.py` (this is a not a restriction, just to make things easier).

```
$ python server.py
```

You'll see exactly the same as running **chimera**.

But, as said in the first paragraph of this document Chimera is client/server, `server.py` shows how to create a server, let's see how to use it in a client.

```
from chimera.core.manager import Manager

manager = Manager()

example = manager.getProxy("localhost:8000/Example1/example")
example.doSomething("client argument")
```

Save it to `client.py`. First run, `server.py` as explained above and then run `client.py`.

```
$ python client.py
```

You will see something like this:

```
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=test argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
[date] WARNING chimera.example1 (example1) example1.py:15 Hi, I'm doing something.
[date] WARNING chimera.example1 (example1) example1.py:16 My arg=client argument
[date] WARNING chimera.example1 (example1) example1.py:17 My param1=a string parameter
```

The first three lines are from `__start__()` calling `doSomething()`, and later three from our client calling it again.

In `client.py`, you see we create a normal `Manager`, just like in `server.py`, but we only use this `Manager` to get access to `Example1` running on other `Manager` (`localhost:8000`).

`Manager.getProxy()` returns a `Proxy` object for the specifies `Location`. For all purposes this `Proxy` class acts like the original object, so you can call any method just like you would with the original object.