# Python for Scientific Gateways Development

**Randy Heiland**
Research Technologies
Indiana University

heiland@indiana.edu

**Sean D. Mooney**
Center for Computational Biology and
Bioinformatics, Department of Medical
and Molecular Genetics, Indiana
University School of Medicine

mooney@mail.compbio.iupui.edu

**Joshua Boverhof, Keith Jackson**
Distributed Systems Department
Lawrence Berkeley National Laboratory

{jrboverhof, krjackson}@lbl.gov

**Maciek Swat, Ariel Balter**
Department of Physics
Indiana University

{mswat, abalter}@indiana.gov

**Marcus Christie**
Computer Science Department
Indiana University

machrist@cs.indiana.edu

**Joseph Insley**
Mathematics & Computer Science
Division, Argonne National Laboratory

insley@mcs.anl.gov

## ABSTRACT
A scientific gateway is an interface that addresses some fundamental needs of a scientific community. This typically involves remote computation and/or data access. The interface might be Web-based or it might be some other type of workstation application that uses client-server technology. This paper focuses on tools and technologies that are based on the Python programming language and can be quite useful for scientific gateway development.

## Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications – *extensible languages, object-oriented languages, very high-level languages.*

H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *web-based services.*

## General Terms
Design, Experimentation, Standardization, Languages.

## Keywords
Python, Web services, Gateways, Web development frameworks.

## 1. INTRODUCTION
A scientific gateway is, in layman's terms, an entryway into some scientific domain from a networked computer. The analogy of an entryway into a home seems appropriate: one enters through a (locked) door [security]; once inside, it is easy to explore publicly-accessible rooms [interface layout and interaction]; the rooms themselves [tabs, panels, gadgets] accommodate different aspects of work, education, and leisure.

For a more concrete reference point, a TeraGrid Science Gateway [1] is defined as "a community interface that enables access to high end resources, usually through a web portal or desktop client server arrangement". We will present examples of both web and desktop applications that may or may not qualify as a gateway, depending on your point of view.

The landscape of technologies and tools for developing gateways is quite vast. However, what are known as Enterprise Frameworks has dominated much of that landscape for the past several years. In this paper, we offer some alternatives that, we believe, are lighter weight in terms of their development and deployment costs yet empower scientific communities more than ever. Our approach should appeal immediately to scientific communities who are already using Python and thereby provide them with an economy of scale in terms of software engineering costs. More to the point, developers of scientific codes should be able to prototype their own gateways using some of the Python-based projects discussed here. Part of the motivation for this paper can be found online [2].

This paper will be part survey, part unapologetic evangelizing (for Python), and part discussion of a particular gateway (web) development framework.

## 2. PYTHON: BATTERIES INCLUDED
Python [3][4] is an open source, cross platform, high level programming language. It has a rich set of features that makes it well-suited for scientific applications. A recent issue of CiSE [5] was devoted to that very topic. Some basic features of the language itself include: very readable syntax, dynamic data types, object orientation, and modularity (allowing for easy extensibility via community-contributed modules). In addition, the "batteries included" phrase refers to Python's standard library, offering a wealth of functionality. Moreover, it is usually straightforward to wrap existing codes (C, C++, Fortran) with Python, i.e. make them accessible from Python.

Not surprisingly, some of the earliest adopters of Python as an interface to large computational scientific codes were in the U.S. National Labs. Various groups at both Lawrence Livermore and Los Alamos National Labs, in particular, were using Python to make it easier to interact with physics codes. For example, Dubois [5] has shared his personal experience for performing computational steering at LLNL using Python. Another early project at LANL led to the development of SWIG [6], now one of the most popular tools for automatically wrapping low-level languages in Python.

We will not attempt to provide a Python tutorial in this paper. Rather, for beginners in Python, we refer them to helpful online material [7]. For those particularly interested in scientific Python packages, we recommend additional material [8], some of which we refer to in the next section.

# 3. SURVEY OF PYTHON-BASED PROJECTS

Since we are discussing the development of scientific gateways in general, using Python-based tools, a table of some Python projects that may be useful for such gateways seems relevant. Table 1 may seem an odd mix of tools and, admittedly, reflects some of our personal projects and interests. Nevertheless, it serves as a starting point for newcomers who might consider Python for science-related projects.

**Table 1. Some potentially useful Python-based projects for scientific gateways**

| | |
|---|---|
| NumPy | Numerical module providing array objects |
| UCSF Chimera, VMD, PyMOL | Molecular graphics and modeling |
| CompuCell3D | Environment for cell-based modeling |
| ZSI | Web services toolkit |
| pyGlobus | Interface to the Globus Toolkit |
| pyGridWare | WSRF toolkit (GT4 compatible) |
| SciPy | Science and engineering modules |
| Biopython | Tools for computational molecular biology |
| VTK, ParaView | Scientific visualization |
| matplotlib | 2D plotting |
| TurboGears, django | Web development frameworks |
| SWIG, SIP | Tools for wrapping code in Python |
| Pyrex | Tool for writing extension modules in C |
| VisTrails | Scientific workflow system |

While some of the projects listed here appear to be standalone workstation applications and therefore not seem to fit into a gateways scenario, it may be the case that either the tool can also be used as a non-GUI Python scripting engine or that the GUI application offers client-server functionality (recall the definition of the TeraGrid Science Gateway). We refer to these projects by their commonly used names and provide a simple description. A web search will provide more information. Obviously, there are many more Python projects that could be listed here as potential contributors to scientific gateways (and many non-Python based projects that could probably be easily wrapped in Python). Remember, our goal is to minimize the burden on the scientists who might like to develop their own gateways.

# 4. WEB SERVICES

Though not a requirement for building gateways, one approach for performing remote data I/O and remote method execution is via Web services [9]. By incorporating standards-based Web services into the gateway model, one can provide dynamically updated results. There exist a panoply of Web services technologies that one could write about. For our purposes, we highlight one key component, namely, the Web Services Description Language (WSDL). A WSDL essentially defines the API for a Web service.

We offer, as examples, some projects that incorporate Web services and might be classified as scientific gateways. The first example uses a Python-based workstation application, UCSF Chimera, for molecular graphics. The science behind this project is to perform a structure-based local environment search [10] on user-supplied proteins against a set of databases containing known protein structures and functions. The gateway infrastructure incorporates a Python Web services toolkit (a precursor to the ZSI toolkit) to provide both the server hosting functionality (database access, algorithm execution) and the client functionality (job submission, visualization of results, etc). Thanks to Python's easy extensibility, we were able to augment the core Chimera application with a "plugin", a set of GUI panels that communicated with the server (Figure 1, plugin is comprised of the panels surrounding the 3-D view).
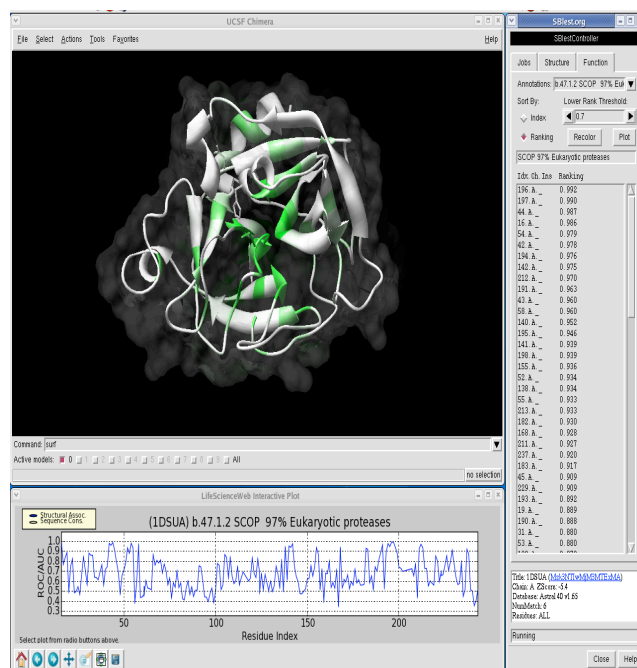


**Figure 1. An SBLEST gateway using UCSF Chimera**

We have also developed a Chimera plugin that communicates via Web services provided by the ChemBioGrid project [11] (Figure 2).
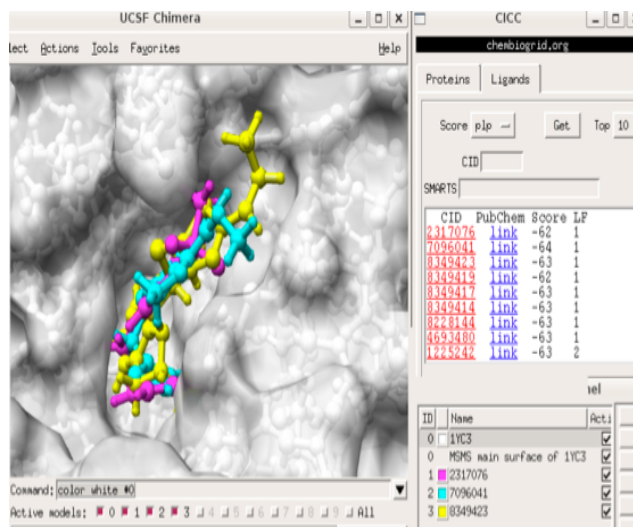
**Figure 2. A ChemBioGrid gateway using UCSF Chimera**

In a similar fashion, the ParaView scientific visualization application (like the VTK libraries on which it depends) can also be extended via Python [12]. This opens the door for more interesting extensions to the TeraGrid Visualization Gateway [13]. We plan to explore VMD [14] as a rendering engine for this gateway in the future.

# 5. PYTHON WEB FRAMEWORKS - TURBOGEARS

We now focus on Web-based gateways, as opposed to workstation applications operating as gateways. When discussing Web gateways, one enters the realm of Web development frameworks. Our criteria for choosing a suitable framework includes: Python-based, active developer and user communities, and ease of use. In our opinion, the two open source Python-based frameworks that are viable options are django and TurboGears (incidentally these frameworks are similar to the popular Ruby on Rails application). We will discuss and evaluate the suitability of TurboGears as a web framework for creating scientific gateways.

Like Enterprise Frameworks mentioned above, TurboGears offers a Model-View-Controller (MVC) paradigm for Web development. TurboGears is designed for rapid development and, not surprisingly, the MVC tools are all Python-based. Each of the three MVC components uses existing Python modules; there is no need to reinvent the wheel. For the Model component, a TurboGears project can connect to a relational database using SQLObject [15], the View component uses Kid templating [16] and the Controller component uses CherryPy [17].

For the following exercise, we have installed the latest version of Python (currently 2.5) and the easy_install Python module [18]. We are also running the Apache web server (version 2.2.4) and the mod_python Apache module. To install TurboGears, one simply runs the command (probably as admin/root or sudo):

$ *easy_install turbogears*

After installing TurboGears, one can start a "project", i.e. a gateway project for our discussion, simply by running the command:

$ *tg-admin quickstart*

and be prompted for a project name. This will create a directory/folder structure that includes a Python script called *start-<project-name>.py*. After running *start-<project-name>.py*, one can point a browser to *http://localhost:8080* (user configurable) and see the default Web page ("gateway") for this project (Figure 3).
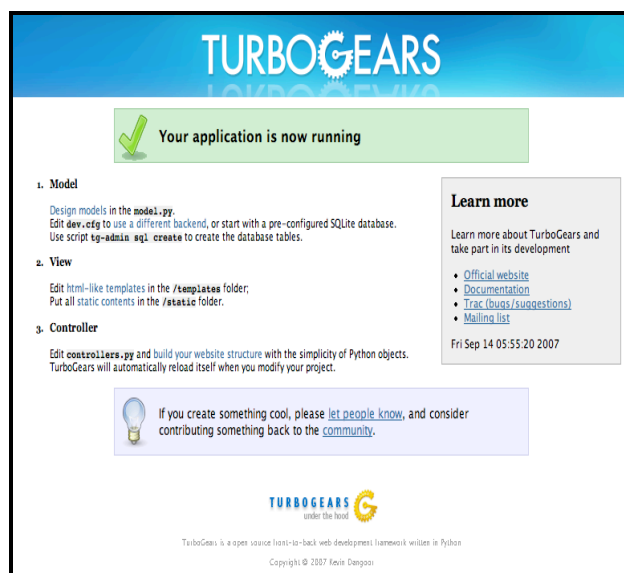


**Figure 3. Initial startup screen for a new TurboGears project**

At this point, it is mostly a matter of becoming familiar with the syntax of the automatically generated template for this view (welcome.kid) and script for the controller (controller.py) in order to create the desired look and functionality of a gateway. Fortunately, TurboGears provides several "extras" that can help. These include AJAX functionality (via MochiKit) such as drag-and-drop, the ability to serialize objects using JSON instead of XML, and numerous predefined widgets (several being AJAX-based). Some of the widgets are common: buttons, checkboxes, input forms, tables, file browser, etc. However, some widgets are more complex: data grids, 2-D plots, multiple-value selection between two lists, etc. (Figure 4).
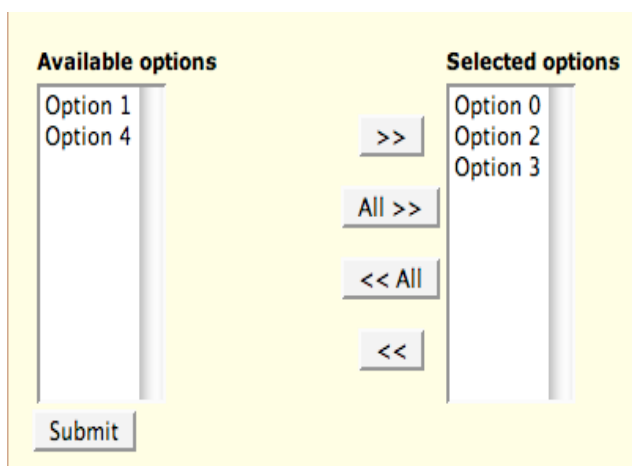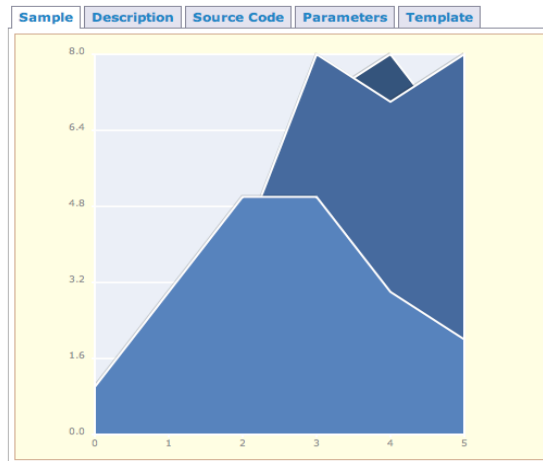
**Figure 4. 2-D plotting (top) and file transfer (bottom) widgets in TurboGears**

# 6. SIMPLE EXAMPLES

Now that we have more or less laid the foundation for developing a Python-centric scientific gateway, let's actually build some simple examples. We begin with something that might be a useful component for a TeraGrid-related gateway. The Network Weather Service (NWS) [19] offers a batch queue prediction Web service. It includes several TeraGrid computing resources. Using the ZSI toolkit (version 2.1), one can automatically generate a Python client script from the NWS WSDL (*Nws.xml*) that is provided, as follows:

% *wsdl2py --complexType Nws.xml*

This will automatically generate the Python script, *NwsService_client.py*. One could then use an (Python) IDE to interactively develop functionality that may be specific to your gateway's needs and then insert this functionality into the appropriate MVC component.

```python
from NwsService_client import *

loc = NwsServiceLocator()
stub=loc.getNws()

req = qbetsGetMachinesRequest()
resp = stub.qbetsGetMachines(req)
machineInfo = resp._qbetsGetMachinesReturn
#print 'available machines = ',machineInfo

req = qbetsPredictBoundRequest()
# set input parameters
req._in0 = 0          # timestamp for time of prediction (0=now)
req._in1 = "bigred"  # machine internal tag (rf. qbetsGetMachines())
req._in2 = "MED"    # queue of interest
req._in3 = 4         # how many nodes job will request
req._in4 = 341       # walltime max the job will request
req._in5 = 0         # quantile of interest (0= 0.95)
req._in6 = 0         # startDeadline (# of secs from 'now' of job's desired start time)

# call the remote method
resp = stub.qbetsPredictBound(req)
#print 'qbetsPredictBound-------- ',resp

bigredXML = resp._qbetsPredictBoundReturn

# use an XML parser from the standard library to parse the output results
import xml.etree.cElementTree as etree
root = etree.fromstring(bigredXML)
print '----- Big Red results ------'
print etree.tostring(root)
```

This produces the result:

```
----- Big Red results ------
<boundPrediction>
    <status>SUCCESS</status>
    <statusLong>Success</statusLong>
    <prediction>3658</prediction>
</boundPrediction>
```

After incorporating the above client code into the TurboGears controller script and making a small change in the style sheet template, we trivially transform the default gateway in Figure 3 to that of Figure 5. Obviously, a more eye-pleasing style is possible if we had used a predefined TurboGears form widget.

With Figure 5, we have set the stage for our next example application, CompuCell 3D [20] (CC3D). CC3D is an open source tissue simulation environment. Like UCSF Chimera, it is intended to be used primarily as a standalone workstation application (Figure 6). However, because it provides a Python API, CC3D can be used in a non-interactive fashion as well. We will take advantage of this and present an example that incorporates CC3D into a gateway. Our goal in doing this will be to allow for easy user input of simulation parameters and subsequent remote execution of the simulation, providing a gateway component for parameter studies. It is quite often the case that modeling efforts are aimed at finding an optimal set of parameters for which the model mimics reality. Unfortunately, in many cases, a search for parameters is purely heuristic and requires many trial-and-error attempts. Providing a gateway component for parameter sweeps could greatly improve a researcher's productivity.
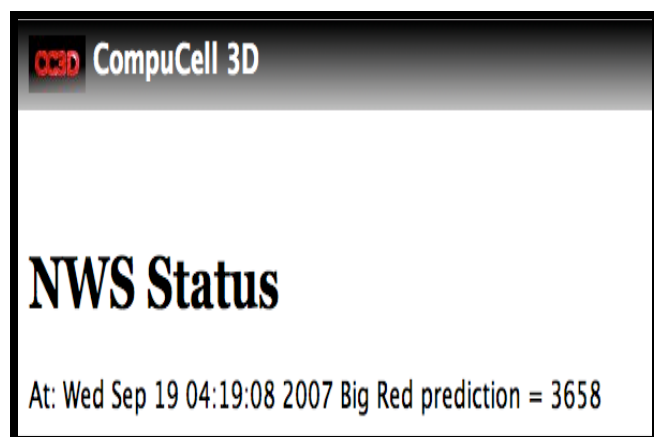


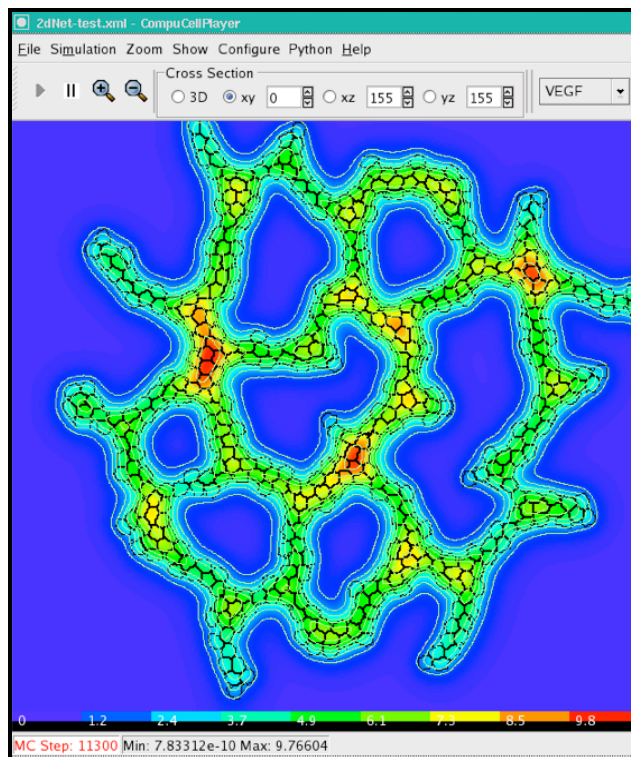**Figure 5. Simple example for real-time NWS results**



**Figure 6. CompuCell3D application**

Making minor changes to the *controller.py* and *template.kid* files, we can create a simple gateway component (Figure 7) to accept input parameters, select a machine (and queue), and submit a remote job. To execute a remote job, we follow the same approach used in the (Python-based, ZSI-enabled) VisPort [21] TeraGrid Web service, namely, invoke the command *globus-job-run* from a Grid(Globus)-enabled server. We also make the simplifying assumption that there exists a community credential and we take advantage of the MyProxy service [22]. Assuming the job runs successfully, we could then use a file transfer gateway component as depicted in Figure 4 to retrieve output files.

**Figure 7. Mock-up for parameter input and job submission**

## 7. SECURITY

To prevent resources exposed by a gateway from being available to the general public, securing the web-based gateway is a necessary deployment step. We have a fairly straightforward out-of-the-box solution used to secure web gateways using Grid Security Infrastructure (GSI) authentication. Our solution assumes: the GSI file systems (e.g. */etc/grid-security*) are available on the host computer [23], a newer version of M2Crypto (>=0.16) is installed, and a client certificate exists and has been imported into the browser. Due to space constraints, we will not elaborate on the steps involved but are more than willing to share this information with interested readers.

## 8. NON-PYTHON COMPONENTS USING AJAX

While there are a lot of great libraries for scientific gateway development written in Python, there are also several others available in different languages and runtimes. For example, a great deal of grid middleware is available in Java, and in some situations it would be beneficial to leverage this software. One way to create a web application using a Python web framework where the backend is partially non-Python would be to create web components that are written to the browser runtime environment and that communicate with the backend via an AJAX mechanism. These web components could be implemented as JavaScript widgets, using a JavaScript library like MochiKit, Yahoo UI, Dojo, etc., or as Java applets or Adobe Flex/Flash components. The main focus in this section is on JavaScript widgets talking to a Java backend. To begin, the web components are added to the view of the Python web framework (in TurboGears via the Kid templates). They expect to make AJAX-like calls to the originating server. To make this work, the Apache HTTP server that is used to connect to the TurboGears server is also made to connect to and proxy certain requests to an instance of Apache Tomcat. The Java backend would run as a web application inside of Apache Tomcat.

As an example, imagine a very simple MyProxy web component implemented with a JavaScript library. This can be added directly to a Kid template in a TurboGears application as a snippet of JavaScript (perhaps needing a configuration option specifying a path for AJAX requests). The MyProxy web component presents a username and password login form. When the user fills in and clicks the submit button, it generates a request to */grid/MyProxyHandler*, where */grid* is the path that Apache HTTP server has been configured to map to the Apache Tomcat instance, and */MyProxyHandler* is a path to a servlet running in Tomcat that has been designed to handle these requests. This servlet could then be implemented using the JGlobus API, for example.

Note that the web components themselves are agnostic as to their backend. They expect to be able to communicate via some XML or JSON schema with configured handlers, but whether those handlers are implemented in Java, Python or some other language is of no concern to them.

## 9. CONCLUSIONS

It is our belief that there is plenty of room in the scientific gateways landscape for more ideas and more development tools. Furthermore, we believe that scientific communities should be empowered to build their own gateways, if they so desire. A novel idea would be to provide some sort of self-organizing mechanism in the gateway itself. Imagine, during the prototyping phase, that end-users (scientists) make incremental changes to the gateway layout and functionality, evolving it to be optimal for their needs.

This paper has focused on Python-based tools and has presented examples using two different approaches to gateways: extending workstation applications to be client-server and using the TurboGears web development framework, both taking advantage of Web services. For scientific communities who are already familiar with Python (or are considering learning), we hope we have inspired you to experiment with some of the ideas and tools discussed here. For communities using other approaches for developing gateways, we hope we can share ideas and that science and science education will benefit.

## 10. ACKNOWLEDGEMENTS

## 11. REFERENCES

[1] http://www.teragrid.org/programs/sci_gateways/

[2] http://communitygrids.blogspot.com/2007/02/rethinking-science-gateways.html

[3] http://www.python.org

[4] http://en.wikipedia.org/wiki/Dive_into_Python

[5] Dubois, PF. Python for Scientific Computing, Computing in Science & Engineering, vol. 9, no. 3, May/June 2007.

[6] D.M. Beazley and P.S. Lomdahl, "Building Flexible Large-Scale Scientific Computing Applications with Scripting

Languages", 8th SIAM Conference on Parallel Processing for Scientific Computing, March 14-17, Minneapolis, Minnesota. (1997). CD-ROM.

[7]     http://wiki.python.org/moin/BeginnersGuide

[8]     http://wiki.python.org/moin/NumericAndScientific

[9]     http://en.wikipedia.org/wiki/Web_service

[10]    Peters B, Moad C, Youn E, Buffington K, Heiland R, Mooney SD.  Identification of Similar Regions of Protein Structures Using Integrated Sequence and Structure Analysis Tools.  BMC Structural Biology 2006, 6:4.

[11]    http://www.chembiogrid.org

[12]    http://www.paraview.org/Wiki/Python_Programmable_Filter

[13]    http://tg-portal.uc.teragrid.org

[14]    http://www.ks.uiuc.edu/Research/vmd/

[15]    http://www.sqlobject.org/

[16]    http://www.kid-templating.org/

[17]    http://www.cherrypy.org/

[18]    http://peak.telecommunity.com/DevCenter/EasyInstall

[19]    http://nws.cs.ucsb.edu/ewiki/ (http://nws.cs.ucsb.edu:8180/axis/services/Nws?wsdl)

[20]    http://compucell3d.org/

[21]    Baker MP, Heiland R, Bachta E, Das M. VisPort: Web-Based Access to Community-Based Visualization Functionality. Proceedings in TeraGrid Conference, Madison WI, June 4-8, 2007.

[22]    http://grid.ncsa.uiuc.edu/myproxy/teragrid.html

[23]    http://www.globus.org/security/overview.html