PYTHON: A PROGRAMMING LANGUAGE FOR SOFTWARE INTEGRATION AND DEVELOPMENT

M. F. SANNER
The Scripps Research Institute
10550 North Torrey Pines Road, La Jolla, CA-92037
sanner@scripps.edu

Over the last decade we have witnessed the emergence of technologies such as libraries, Object Orientation, software architecture and visual programming. The common goal of these technologies is to achieve software reuse. Even though, many significant advances have been made in areas such as library design, domain analysis, metric of reuse and organization for reuse, there are still unresolved problems such as component inter-operability and framework design[1]. We have investigated the use of interpreted languages to create a programmable, dynamic environment in which components can be tied together at a high level. This work has demonstrated the benefits of such an approach and has taught us about the features of the interpreted language that are key to a successful component integration.

The problem

One of the challenges in bio-computing is to enable the efficient use and inter-operation of a wide variety of rapidly-evolving computational methods to simulate, analyze, and understand the complex properties and interactions of molecular systems. In our laboratory we investigates several areas, including protein-ligand docking, protein-protein docking, and complex molecular assemblies. Over the years we have developed a number of computational tools such as molecular surfaces, phenomenological potentials, various docking and visualization programs which we use in conjunction with programs developed by others. The number of programs available to compute molecular properties and/or simulate molecular interactions (e.g., molecular dynamics, conformational analysis, quantum mechanics, distance geometry, docking methods, ab-initio methods) is large and growing rapidly. Moreover, these programs come in many flavors and variations, using different force fields, search techniques, algorithmic details (e.g., continuous space vs. discrete, Cartesian vs. torsional). Each variation presents its own characteristic set of advantages and limitations. These programs also tend to evolve rapidly and are usually not written as components, making it hard to get them to work together.

The “traditional” solution

Typically, researchers have been using tools such as AWK and shell scripts to make such programs work together. If that approach appears tempting initially, it has many inherent problems and limitations that will surface in the long run. These include, a very low level of inter-operability: i.e. usually data is transferred between programs using files or pipes allowing only inter-operation at the program level rather than the function level or at least functionality level. This makes its difficult, for instance, for a molecular dynamics code to use some third party electrostatic or molecular surface calculation package to derive a term used to drive the simulation, or to use someone’s visualization program tools to steer the simulation or monitor or play back trajectories. Such developments usually require substantial coding and often access to and understanding of the source code. Such an approach also requires the creation of a large number of interfaces between different tools. This makes it very hard to incorporate new methods into the tool set and therefore stifles the researcher’s creativity. The level of code reuse offered by this approach is very low. For instance, every program operating on molecules will need to implement its own parser for different molecular file formats, each having its own bugs and weaknesses and each requiring coding effort. Finally, this approach often lead to very large scripts that are difficult to maintain, extend and debug.

Other solutions: Visual programming, specialized software suites ...

The frustration of developing under these conditions has prompted us to investigate better methods for developing code and integrating computational methods. Our first approach was based on using AVS (Advanced Visualization System, from AVS Inc.). This environment has proven very useful for us over the last ten years, in terms of code reuse and capturing developments done by a set of transient collaborators, typically post doctoral fellows who spend a few years in the laboratory before moving on. We have also encountered some limitations. AVS is a data-flow driven computation and visualization environment that comes with a large number of processing modules for a wide variety of operations such as: data input, image processing, surface and volume rendering, etc. These modules can be linked together graphically using a network editor to create a processing stream for a particular visualization or computation. It also offers a mechanism for adding custom-

designed modules for new computational methods. AVS users roughly fall into three classes distributed in a pyramid. At the high end is the module programmer, typically writing C programs and making this code available as AVS modules. The second, and larger class of users, are those who produce their own networks using existing modules. Although networks do not have constructs for loops or conditional execution, many visualizations can be done at this level without writing a single line of code. The third, and largest class of users, are those who use their own data with an existing network. One of the reasons AVS has worked well for us is probably due to this visual programming paradigm creating this intermediate class of users which fits quite well scientists in need of custom visualization who do not want to become programmers. Of course AVS' modular nature promotes code reuse that leads to rapid prototyping. This has enabled the scientist to concentrate on the visualization process rather than the program used to visualize the data.

However, molecular modeling and bio-molecular visualization pose many challenging problems for data-flow environments. Molecules have a high-level of internal organization which it is often desirable to reproduce in the programs operating on them. This is not always compatible with the simple data-types typically available in these environments. There are also problems of data duplication and inter-module communication. But, what we felt was the most restraining limitation in AVS was its lack of scripting capabilities. The AVS Command Language Interface (CLI) merely consists of a set of commands and moreover it exposes only a subset of the kernel's functionality creating some serious limitations. We have lifted some of these problems by embedding a Python interpreter in an AVS module thus adding scripting capabilities [1].

Commercial and academic molecular modeling packages address these problems more specifically than AVS, however, most of these packages are monolithic programs providing only a limited set of options for altering the style of the visualization or extending the program to accept new types of data or to do new computations. Since one of our missions is to investigate new computational methods and visualizations they did not appear to be the right tools.

Language centric approach and interpreted languages

Programs are usually developed in a "self centric" way, meaning that theya re written to be self contained units aimed at solving a given problem of fulfill a given task. Some programs, like AVS, are designed to be extended by adding new modules that can encapsulate new computational methods, but this always has to be done within the program's framework. And since programs are inherently specialized, this is bound to create problems. To address this problem we decided to experiment with a "language centric" approach. We use a high-level language as the core of our framework. Rather than writing programs we now extend this language with modules or components implementing specific functionality. The high-level language serves as a "glue" to tie modules and components together to rapidly create specialized applications. In some sense, the language becomes a "scripting framework" allowing fast prototypying of new applications. Developing extension modules for the language corresponds to postponing specialization of code as much as possible.

We felt, that an interpreted language would provide the flexibility, interactivity and extensibility needed for such an approach and we started exploring using the three most popular interpreted languages: Perl, TCL and Python. There are a number of articles comparing these three languages to each other as well as to compiled languages such a C, C++ and Java [see http://www.python.org/doc/Comparisons.html for a list of articles]. After some experimentation with these different languages we learned that "all interpreted languages are not create equal" and each has its specific strengths and weaknesses. Of course they all provide a scriptable framework that is interactive, flexible, extensible and embeddable but there are differences in style and philosophy that make one or another more compelling for a given task.

Perl has the largest user base and is excellent for surprisingly short scripts that do a lot of work, which unfortunately can also be quite challenging to understand. This language offers good support for common application-oriented tasks whereas Python's elegant, and not overly cryptic, syntax emphasizes support for common programming methodology and promotes code readability and thus maintainability. Tcl, like Python can be used as an extension language and a stand-alone programming language but it's support for data structures is rather weak (traditionally everything is a string). Moreover, the lack of modular name spaces before version 8.0 hindered the development of large programs. All these languages span multiple platform and often provide more platform independence than Java. They all can be extended in C or C++.

We settled on Python for a number of reasons, including: it's concise and almost pseudocode-like syntax; its modularity; its object oriented design; its profiling, debugging, reflection, introspection and self documentation capabilities; and the availability of a Numeric extension allowing the efficient storage and manipulation of large amounts of numerical data. Python is as good a glue as any other interpreted language but in addition it can be used to develop substantial extension components.

Python

Python is an interpreted, interactive, object-oriented programming language. It provides high-level data structures such as list and associative arrays (called dictionaries), dynamic typing and dynamic binding, modules,

classes, exceptions, automatic memory management, etc.. It has a remarkably simple and elegant syntax and yet is a powerful and general purpose programming language. It was designed in 1990 by Guido van Rossum. Like many other scripting languages it is free, even for commercial purposes, and it can be run on practically any modern computer. A python program is compiled automatically by the interpreter into platform independent byte code which is then interpreted. We are running unmodified components written in Python under linux, Windows NT, 98, 95, IRIX, SunOS, OSF.

Python is modular by nature. The kernel is very small and can be extended by importing extension modules. The Python distribution includes a diverse library of standard extensions (some written in Python, others in C or C++) for operations ranging from string manipulations and Perl-like regular expressions, to Graphical User Interface (GUI) generators and including web-related utilities, operating system services, debugging and profiling tools, etc. New extension modules can be created to extend the language with new or legacy code. We describe these extension capabilities below. There are a substantial number of extension modules that have been developed and are distributed by members of the Python user community. These extension modules, sometimes referred to as "packages" or components include as GADFLY, an SQL database manager written in Python; PIL, the Python imaging library; FNORB and OmniBorker, CORBA compliant Object Request Brokers (ORB) written in Python; Gendoc, an automated documentation tool; and Numeric Python, just to name a few.

The best ressource for Python along with the books that are available is probably the Python web site (http://www.python.org). It provides access to code, documentation, packages, articles, mailing lists etc. It is also worth mentioning the recent creation of the biopython.org web site, a collaborative software effort for computational biology and chemistry very much like bioperl.

Finally, it worth mentioning that, besides the C implementation of the Python interpreter, there is also a 100% pure Java implementation called JPython, allowing the use of Python as an interpreted language for programming in the Java world. This interpreter allows to instantiate Java class, and Java code can call Python code. The native extension first need to be made available in the Java world before they become available in JPython.

The Python Numeric extension

Numeric Python is an extension module for efficient storage and data-parallel manipulation of numerical data. Using this Module, many simple operations which would be too slow in Python can be performed very efficiently (basically as fast as in C) without having to implement the code in C. This module very often allows us to write extensions in Python (using Numeric) that are efficient enough for our purposes and do not need to be re-coded in C or C++.

For example, if the coordinates of the N atoms consituting a molecule are stored in a matrix of Nx3 floating point values called "Coordinates", operations like the ones demonstrated in the following Python code become syntactically trivial and as efficient as if they were programmed in C or C++:

- translate the molecule using a 3-vector T. This illustrates data-parallel operations, the 3-vector is added to each 3 vector in Coordinates

```
>>> Coordinates = Coordinates + T
```

- compute the center of gravity of the molecule. First sum all values while collapsing the array's first dimension, then divide by the number of atoms.

```
>>> import Numeric
>>> g = Numeric.sum(Coordinates) / len(Coordinates)
```

- Compute the distance from all atoms to a given 3D point P

```
>>> d = Coordinates - P
>>> # now sum over the 2nd dimension and take the sqrt
>>> d = Numeric.sqrt(Numeric.sum(d*d, 1))
```

Performance issues

Very often there is a concern about the price paid in terms of performance when using an interpreted language. We have learned that, first of all, we had a wrong conception about where, when and how much performance we need. In addition, Python code runs with reasonable performance sufficient for many common task. Finally, having the Numeric extension helps preserving good performance even when working with large arrays of numbers. An example I like to give to illustrate these points is our first PDB parser for Python we which developed as a C extension because we though that Python code would not be efficient enough. The C extension we wrote to read PDB files ended up being very complex and became difficult to maintain and extend. So we decided, to develop a Python version that turned out to comes very close in performance to the

C version for parsing a PDB file, a tree-like structure and computing the connectivity. It also was much smaller, simpler, easy to maintain, and platform independent. We feel that the small performance loss was well worth what we gained in flexibility and portability. Since then, we prototype all new extensions in Python and then profile them before deciding what parts really need to be re-implemented in C or C++.

Python as an integration tool

As we mentioned before, one of the key feature of all modern interpreted languages is their extensibility. I would like to briefly review the different ways in which our Python based scripting framework can integrate legacy code and be extended with new functionality. There are basically three ways to add new functionality to this scripting framework: by implementing it in Python, by "wrapping" existing code or by creating an interface to existing code that has some communication capabilities.

- Implementing the functionality in Python:

This is the method of choice for all new developments we undertake. Indeed, for code that has not yet been written the high-level nature of Python and the already available extensions make it generally much easier to implement in Python than in C or C++. Even when some code already exists there are a number of cases where we decide to re-implement in Python. This happens when re-coding in Python requires a relatively small amount of code (which is not unusual) and the performance of the Python code is expected to be acceptable. This approach provides the major advantage of platform independence.

- Wrapping existing C, C++ and Fortran code:

There are many cases where it does not make sense to re-implement some legacy code in Python. Yet it is desirable to have access to this code from Python. This requires "wrapping" the legacy code for Python. Let us consider the following simple example: We have a function "foo" that takes an integer as an argument and returns a float. In the Python world integers and floats are objects. In order to be able to call the function foo from Python we have to write a function in C that will: take a Python object as an argument; extract the integer value from this object; call the function foo for this integer value; package the returned float into a Python object representing a float; and finally return that object to the interpreter. This code is called the "wrapper code". Once we have the wrapper code we can compile it along with the function foo and create a shared object (dll in the Windows world) which can then be imported into a Python interpreter. Of course, this extension is now platform dependent. This process of wrapping code can be automated to a fair extent using SWIG, Simple Wrapper Interface Generator (http:// www.swig.org). SWIG can generate wrapper code for several interpreted languages like TCL, Perl, Guile an of course Python. Recently support for Java has been added, too.

This approach is very useful to extend Python with Application Program Interfaces (API) of existing software. Once such an API has been wrapped by someone it can be made available to the community and the number of such extensions is large and growing. (Oracle, mySQL, OpenGL, DCOM, etc ...).

- Interfacing code:

For legacy code that was designed with some communication capabilities, such as through sockets or using a standard communication protocol (HTTP, NNTP, SMTP, etc.) the support provided by the standard Python modules for these protocols makes is generally quite easy to write a client in Python.

Finally, a Python interpreter can be embedded in an application as an extension language. We have describe such an approach for AVS [2]. Besides adding scripting capabilities to the program in which the Python interpreter is embedded, this also make all the tools ported to Python immediately available within this program.

Python extensions we have developed

Over the last couple of years we have developed a number of Python modules or components to deal with many aspects of our daily work. Using SWIG we have wrapped codes dealing with tasks such as: molecular surface computation; molecular mechanics and dynamics; electrostatic calculations; protein-protein docking; protein-ligand docking; etc. Besides these "traditional" computational methods we have also wrapped a number of less conventional tools for molecular modeling, such as: a convex hull calculation tool; a rapid collision detection of polygonal models library; several mesh decimation algorithms, a general extrusion library, etc.These modules are all platform dependent and we are mainly using them on SGI and Sun computers. We have also developed two platform independent packages in Python: DejaVu/DejaVuFramework which is a general 3D geometry visualization package and BioChem that provides support for manipulating and visualizing molecules.

DejaVu is a package written in Python for the visualization of 3D geometry using the OpenGL library. It

provides a set of classes describing objects such as viewer, cameras, lights, clipping planes, color editor, trackball, geometries etc. The Viewer class is a fully functional visualization application providing control over a number of rendering parameters such as, depth cueing, global anti-aliasing, rendering modes (points, lines, polygons), shading modes (flat, gouraud), multiple light sources, arbitrary clipping planes etc. An instance of a Viewer maintains a hierarchy of objects in which rendering attributes can be inherited. DejaVu also provides a number of standard geometries including lines, indexed-lines, indexed-polygons, triangle and quad strips, spheres, cylinders, labels. This list can be extended by sub-classing the geometry base class. DejaVu makes it very trivial to add visualization capabilities to any object developed in Python

DejaVuFramework is a "sub-package" of DejaVu that provides support for building visualization applications in which the overhead of creating new commands an their associated Graphical User Interface is minimal. This framework provides support for loading dynamically commands from libraries as they are needed. It has been designed to be specialized and extended in a very modular way.

The BioChem package provides classes to read molecules in a number of file formats and build a tree-like hierarchical structure reproducing the molecule's internal structure. For instance, a protein molecule is represented using the following four levels: molecule, chain, residue, atom. The number of levels can be modified. For instance, we have added a level between chains and residues to represent the secondary structure of the protein. All nodes in that tree-structure can be extended dynamically, i.e. to add a property like a radius to an instance of an atom called "A", one simply assigns the value to a member data of the atom: A.radius = 1.7.

This structure allowing to extend dynamically any node and perform all kinds of selections has proven the be very powerful and provides a high-level interface to the molecule. The following code instantiates a Protein object and builds its representation from a PDB file. Several selection and manipulation illustrate some the basic capabilities of these classes:

```
>>> from BioChem.protein import Protein
>>> from BioChem.pdbParser import PdbParser
>>> protein = Protein()                  ### create a protein object
>>> ### read the file with a given parser object
>>> protein.read('1crn.pdb', PdbParser() )
>>>
>>> ### Once the protein is loaded we can operate on that structure
>>> ### TreeNodeSets behave like Python arrays
>>> residues8Trhough15 = protein.chains[0].residues[8:16]
>>> allAtoms = protein.chains.residues.atoms
>>> print residues8Trhough15
<ResidueSet instance> holding 8 Residue
>>>
>>> ### select atoms all atoms with their name in list:
>>> bbnames = ['N', 'CA', 'C', 'O']
>>> backBone = allAtoms.get(lambda x, names=bbnames: x.name in names)
>>>
>>> ### perform boolean operations over set:
>>> sideChains = allAtoms - backBone
>>>
>>> ### add new members on the fly to any object or set of objects:
>>> ### here we add the a member called "bb" to all atoms and set it to 1
>>> ### for backbone atoms and 0 for side chain atoms
>>> allAtoms.bb = 0
>>> backBone.bb = 1
>>>
>>> ### build bonds using atomic distances
>>> protein.buildBondsByDistance()
```

- A Molecule Viewing Application

Building on top of the BioChem and DejaVu packages, we have developed a molecular viewer [Fig 1]. This viewer has most of the features usually expected in a molecule viewer: stick and cpk representation; different coloring schemes (by atom, by residue type, by chain, by molecule, by propreties); measuring tools; atom identification by picking; support for multiple molecules; secondary structure representation; user definable sets of atoms, residues, chains and molecules, etc. In addition to these traditional features it is dynamically extensible, i.e. new commands can be developed independently and placed in libraries. The Viewer inherits from the DejaVuFramework the capability to dynamically import these commands as needed. In fact, all commands in that viewer have been developed based on this principle. This provides an way to add features to the application that is incremental and well suited for team development. In addition this approach avoids the "feature overload" problem, i.e overloaded menus cluttered with commands that are irrelevant for the problem at

hands. Customization files allow, among other things, to specify which commands should be loaded when the application starts. This allows to define a number of molecular viewing applications just by creating different customization files, each loading different sets of commands. We have written a surface calculation command and one to selectively display pieces of a surface corresponding to a sub set of atoms. In fact in this viewer all geometries relate to the underlying molecules, therefore a subset of atoms can be used to: partially displayed/undisplay; selectively labeled; locally colored using different schemes; any of the geometries. And vice-versa, picking done on any geometry (molecule, surface, secondary structure) is mapped back onto the molecule. This application also provides a Python Shell. This Python aware, typein widget, is the standard input and standard output of the python interpreter running the application. From the Python Shell, any command can be called interactively and scripts operating on the application can be written. Commands log themselves into a file as they are executed. This file can be saved, edited and played back to restore a previous session. This application runs unmodified on any platform that has a Python interpreter with the following extensions: Numeric, Tkinter/Togl and OpenGL.

Maybe the most interesting feature of this application is that creating the application actually comes down to write the commands, and writing commands is made easy by the built-in support provided by the DejaVuFramework.

Conclusion

We are convinced that this “language centric” or scripting framework approach has a lot of strengths and benefits. We have already witnessed: a dramatic increase in our productivity, as well as a high level of code reuse. Of course, this approach is not specific to Python and could be reproduced with any interpreted language and even with compiled languages. But the fact remains that it did not happen with any of the other languages we have used previously. The most important change for us has been to shift from writing programs or scripts to writing modules or components. This concept is well know to the community of component based software development, but in a molecular modeling research environment, with limited resources for the design and implementation of software tools, Python has been instrumental. It has placed these advanced software development techniques within our reach.

References

1 sichel F. Sanner et al. (1998). Integrating Computation and Visualization for Biomolecular Analysis: An example using Python and AVS. Proc. Pacific Symposium in Biocomputing `99. pp 401-412.

2 - J. S Poulin. Reuse: been there, done that. Technical Opinion. Communication of the ACM. May 1999 Vol 45, No 5, pp98,100

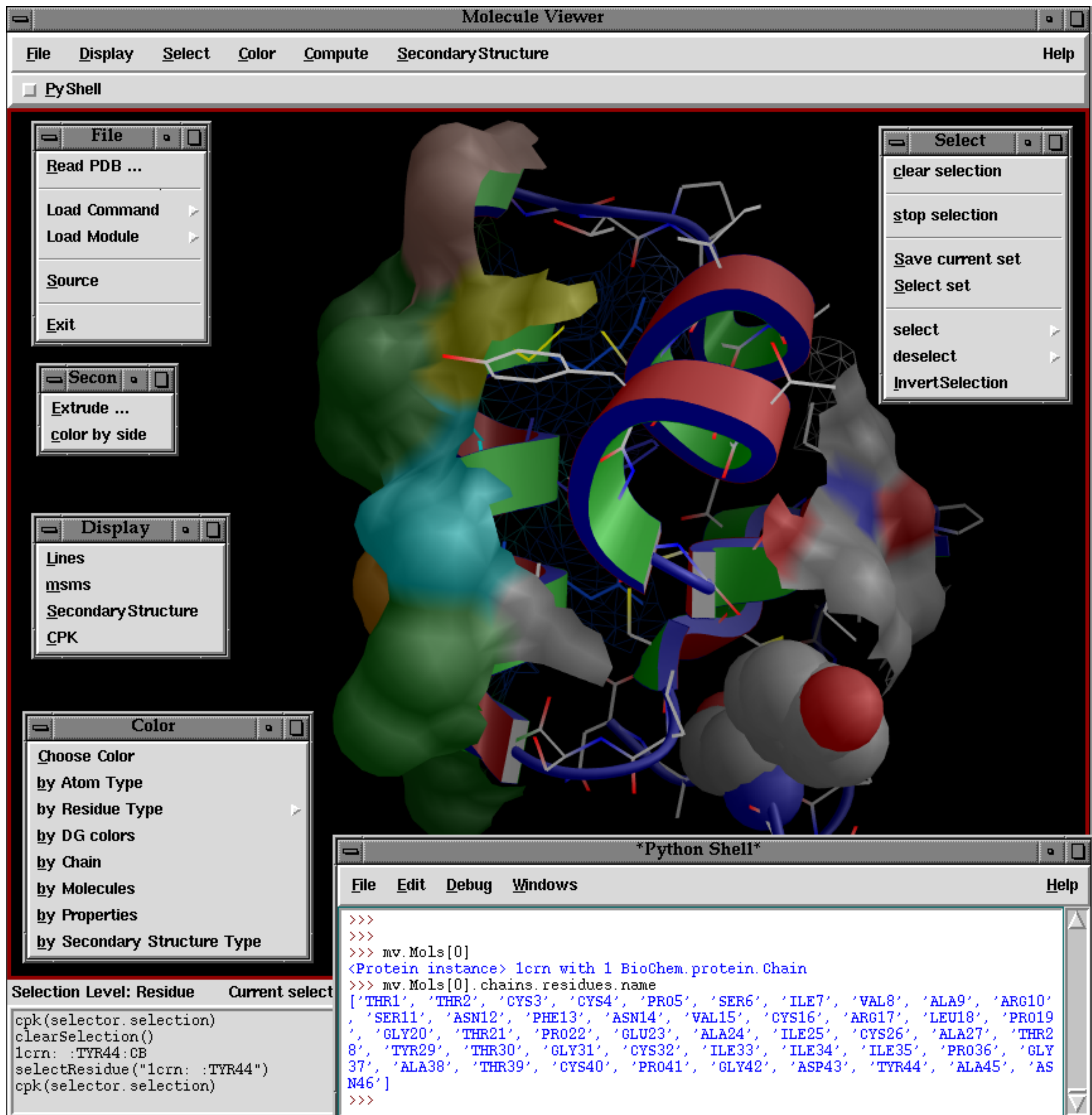

Figure 1: The molecule viewer application showing the protein Carmbin (1crn) with its secondary structure shown as a ribbon. The molecular surfaces corresponding to helix1 and sheet2 are displayed and colored using, respectively the "RASMOL residue" coloring scheme and the "by atom type" coloring scheme. Some pull down menus have been torn off to show the commands they provide. This set of commands can be extended dynamically by loading modules and commands from libraries.