

The Python interpreter as a framework for integrating scientific computing software-components

Michel F. Sanner

The Scripps Research Institute,
Department of Molecular Biology, TPC26
La Jolla, CA 92037

Abstract:

The focus of the Molecular Simulation Laboratory is to model molecular interactions. In particular, we are working on automated docking and molecular visualization. Building and simulating complex molecular systems requires the tight interoperation of a variety of software tools originating from various scientific disciplines and usually developed independently of each other. Over the last eight years we have evolved a strategy for addressing the formidable software engineering problem of integrating such heterogeneous software tools. The basic idea is that the Python interpreter serves as the integration framework and provides a powerful and flexible glue for rapidly prototyping applications from reusable software components (i.e. Python packages). We no longer think in terms of programs, but rather in terms of packages which can be loaded dynamically into the interpreter when needed, and instantly extend our framework (i.e. the Python interpreter) with new functionality. We have written more than 25 packages (>2200 classes) providing support for applications ranging from scientific visualization and visual programming to molecular simulations and virtual reality. Moreover, some of our components have been reused successfully by other laboratories for their own research. Applications created from our software components have been distributed to over 15000 users around the world. In this paper we describe our approach and various applications, discuss the reasons that make this approach so successful, and present lessons learned and pitfalls to avoid in order to maximize the reusability and interoperability of software components.

Keywords: scripting languages, interpretive languages, interpreters, programming paradigms, code reusability, modularity, interoperability, component-based software development, software engineering, computational biology.

Introduction

Molecular biology is evolving from the study of macromolecules in isolation towards complex environments, potentially as large as complete cells. The amount and heterogeneity of information that needs to be processed and integrated in order to understand and simulate such complexity requires a leap in the level of sophistication of our software tools. The next generation of bioinformatics programs will have to: (1) support the inter-operation of rapidly evolving and sometimes brittle, computational software developed in a variety of scientific fields and programming languages; (2) support the integration of data across biological experiments and scales; (3) adapt to rapidly evolving hardware environments; and (4) empower users by allowing them to carry out operations unanticipated by the programmers such as trying new combinations of algorithms or new visualizations. These combined

requirements of flexibility, portability, resilience to failure, programmability by the user, and responsiveness to changes in the computational models and the hardware pose a tremendously challenging software engineering problem.

So far, this challenge has mainly been address by the creation of software frameworks that provide Application Program Interfaces (APIs) for integrating new software components. These frameworks usually provide support for managing software components (e.g. plug-in technology) and the overall workflow and communication between components. We have found that this approach has several drawbacks. First of all, it promotes extensible environments rather than code reuse. Users and developers often have their own software environment and would prefer to include a new method in their own environment rather than having to switch to a completely new framework in which many of their usual tools might not exist. The integration of a given computational method into one of these frameworks only benefits this particular framework and this integration work will have to be repeated for each new framework. More importantly, each framework wants to act as the central piece of the software environment, and be in charge of scheduling operations and the execution flow, thus making it difficult to combine functionality available in various frameworks. These frameworks also often impose restricting data types or data models, and grant limited control to components over execution flow and other components. Finally, many of these frameworks link their extensions into the environment rather than dynamically loading them when needed, thus leading to bloat-ware.

Command languages and scripting capability are often available in complex software environments. On rare occasions, these languages are powerful enough to be used to implement new extensions. However, these languages are often highly biased toward a particular scientific domain and lack generality. Moreover, a framework's language is only usable in that particular application which limits the size of the user community, and in turn reduces potential contributions from the user community. Some applications embed scripting languages such as TCL or Python which is a better approach than developing a custom language. However, the framework is still the central piece of the environment and dictates the way extensions are developed, thus making these extensions framework specific. Moreover, the functionality available in the application itself is not re-usable in other applications, but merely scriptable through the interpreter.

Our Approach

Here we present a solution we have evolved and implemented over the past eight years and which has proven to be successful in addressing some of the shortcomings discussed above. Rather than developing our own interpretive language we used Python, and instead of embedding Python into a framework or developing yet another framework, we decided to make the Python interpreter itself our framework. This idea of using an interpretive language as an integration framework could work with other interpretive languages as well. This Python-centric framework is extended by developing dynamically-loadable Python packages implementing specific functionality. This approach promotes code compartmentalization and encourages developers to switch their mindset from writing programs to writing components that carry out specific tasks. Applications are then generated by combining these components at a high level, leading to true component-based applications. This architecture makes swapping software components, comparing implementations and integrating new methods trivial. Such a high level of flexibility is particularly

interesting in a research environment where the goals are constantly shifting as new discoveries are made, and where new computational methods need to be integrated in our simulation environments as they become available. While this difference in mindset might appear trivial, experience has shown that developing software components in isolation of an application or a framework leads to increased re-usability. When developing for a framework it is always tempting to take advantages of some of its features with the detrimental consequence of tying the extension to this particular framework. Using our Python-centric approach, we have demonstrated unprecedented reuse of software components in several applications that we have developed and distributed. Some of these components have also been used successfully in other laboratories within their own software environments.

While we concede that running a Python interpreter is a constraint for developing and reusing our software components this approach has several advantages. Python is a general purpose programming language; therefore it is not biased toward any application domain. There is a large user community contributing Python packages that can be readily integrated for the most part, and we are using several third party packages in our applications. Having a fully-fledged, high-level, object-oriented programming language as a glue to connect software components greatly reduces the need for stringent APIs for components to be inter-operable because adapters can always be written. Finally, since our software components are developed independently of each other and no component is allowed to assume control of the application. The Python interpreter remains central to the software environment, naturally providing scripting capability over the applications and the data they manipulate. An additional benefit of programming in Python is platform independence which is a great advantage when supporting multiple platforms. While there were initial concerns about the performance impact of using an interpretive language, we found that Python is very well suited for developing entire software components. This is due to the fact that we and other developers have of a very wrong impression about how much and where the utmost performance is needed and where the true bottlenecks are.

Below we will briefly describe the key differences between compiled and interpretive languages and stress the advantages of interpretive languages in general and Python in particular as integration platforms. We describe some software components we have developed and illustrate their integration into end-user applications. We conclude with some lessons learned, pitfalls to avoid and discuss some of the challenges associated with our approach.

Programming languages

In compiled languages (FORTRAN, C, C++, *etc.*) an executable has to be built (i.e. compiled and linked) from the source code before it can be executed. Extending or modifying these programs requires re-building the application after the source code has been modified. This cannot be done while the program is running. Moreover, these languages require complete and syntactically correct source code before compilation and testing become possible. Interpretive languages (Perl (Tisdall 2003), Python (Lutz and Asher 1999), Tcl (Ousterhout 1994), Ruby (Thomas 2001), Scheme (Dybvig 1996), Lisp (Steele 1990), Basic (Lien 1986), *etc.*) use a program called an interpreter to evaluate source code as it is read from a file or typed in interactively by a user. These languages are often referred to as scripting- or interpreted-languages. Applications written in these languages can be extended and modified while the program is running without having to quit, recompile or restart the

application. Code can be executed and tested as it is written, even if only a small fraction of the complete application has been implemented, thus allowing the rapid exploration of novel hypothesis and the early determination of design flaws.

Programming languages can also be classified and described by placing them on a scale ranging from low- to high-level languages. We will deem a language “high-level” if it provides: high-level data structures such as lists and associative arrays; mechanisms for preventing errors from aborting the application (exceptions); array boundary checking, and support for automatic memory management as part of the language. The more such features any particular language provides, the higher it will be ranked. Interpretive languages usually are higher level than compiled languages.

Advantages of Interpretive Languages

Rapid development: The high-level and dynamic nature of interpretive language offers several advantages for scientific computing. The ability to quickly prototype software is particularly interesting in a research environment where the software requirements are constantly shifting as the understanding of the underlying science evolves. **Easy to learn:** Their easy syntax and high-level data structures make it easier for non-professional programmers such as computational biologists to develop programming skills enabling them to interact with data programmatically and eventually develop code on their own. For instance, Python is becoming an increasingly popular language for teaching computational skills (Schuerer and Letondal 2004). **Open-ended application extensibility:** Software environments enabling end users to interact programmatically with their data and with the application using a simple yet fully fledged programming language provides the highest level of extensibility. While GUIs are an excellent way to abstract programming syntax and data structures, with the exception of visual programming (see below), GUIs can only be implemented for anticipated ways of using a program. This lack of programmability associated with GUI-based interfaces greatly limits the application’s range of capabilities and its extensibility. Modifying an application’s source code enables unrestricted modifications; however, it is usually a difficult task and always presents the danger of corrupting the application. General purpose scripting languages provide a safer and easier way to extend applications, while providing the open-ended extensibility which is missing in GUI-based interfaces. Scripting languages blur the line between users and developers. **Glue languages:** Interpretive language can be used to integrate heterogeneous pieces of software written in a variety of compiled languages and let them interoperate.

Interpretive languages provide new solutions to the challenges mentioned in the introduction and are becoming increasingly popular for developing bioinformatics applications (BioPerl (BioPerl), BioPython (BioPython), Chimera (Huang, Couch et al. 1996), PMV (Coon, Sanner et al. 2001), Vision (Sanner, Stoffler et al. 2002), PyMol (Delano 2002), WhatIf (Vriend 1990), PHENIX (Adams, Grosse-Kunstleve et al. 2002), MMTK (Hinsen 1997), VMD (Humphrey, Dalke et al. 1996)) and for teaching programming concepts to biologists (Schuerer and Letondal 2004).

Performance

A drawback of interpretive languages is that they are slower than compiled languages, typically by a factor of 15 to 40 (Cowell-Shah 2004). When compiled languages appeared, they rapidly replaced assembly languages. The substantial increase in readability and portability they offered outweighed the associated potential

decrease in runtime performance. Today, this pattern is repeating itself: the sustained increase of computational power in typical desktop computers makes interpretive languages a practical alternative to compiled languages for a variety of tasks. Moreover, we often have a wrong intuition about how much and where performance is needed. A key observation is that the bulk of the source code of programs used in computational biology deals with input and output, handling events, and bookkeeping of data structures. The computationally intensive parts are often confined to a few functions that amount to a small percentage of the total number of lines of source code, typically less than 10%. Hence, large amounts of code can be implemented in high-level interpretive languages without affecting the overall performance of the application.

Which scripting language is right for my application?

Scripting languages such as the various UNIX shell scripting languages and awk (Aho, Kernighan et al. 1988) have been available for a long time. However, these languages are arcane and limited in generality. They have been reserved to computer savvy users. A number of high-level scripting languages are available today. Perl was one the first to be widely used in scientific computing. It is a powerful language for writing highly concise scripts. However, the way its syntax and notation tends to promote obfuscation is not a desirable feature for reusing and sharing code. Perl is mainly used for writing relatively short, “use-once”-type scripts, but is ill suited for developing large and complex software applications (Raymond 2001). It also is lacking a good interpreter shell for interactive code development. The Tcl language is another example of a widely used scripting language. Unfortunately, its one strength – all data are represented as strings – is also its main weakness for scientific computing. This makes Tcl cumbersome and inefficient for numerical computations. The Tcl-shell is also relatively simple and underdeveloped.

Python appeared in 1991 and was designed from the outset as an object-oriented language while allowing for simple scripting. It supports multiple inheritance, introspection, self-documenting code, high-level data-structures (i.e. lists and associative arrays called dictionaries), and exceptions and warning mechanisms. Its syntax was designed to be free of arcane symbols and to look like English text (i.e. pseudo-code). Python is open source and runs on virtually any computer from super-computers to PDA's. New functionality can be added to the interpreter by loading extensions to the language at runtime. Such extensions can be organized into sets called packages. The standard Python distribution comes with a comprehensive library of extensions covering needs in areas as diverse as regular expression matching, GUI toolkits, databases, network protocols, numerical calculations and XML processing to name a few. In addition, an active community of developers provides a variety of packages spanning all areas of computing, reflecting the diversity of Python's user community and application areas. Python extensions can be written both in the Python programming language and in compiled languages such as FORTRAN, C or C++, thus allowing the incorporation of legacy code. Code written in compiled languages must be “wrapped” (i.e. turned into a Python extension) before it can be called from a Python interpreter. Semi-automatic tools (SWIG) facilitate the process of wrapping compiled code. These extensions execute faster than those written using the Python language. However, they are platform-dependent (i.e. they need to be compiled for every hardware platform and operating system). With Python, it is possible to have “the best of both worlds” by implementing those functions that require the utmost in performance using a compiled

language while the remainder of the application can be written in a high-level and platform-independent language.

Python has been used as a “glue language” to integrate monolithic programs. With the wrapping mechanism described above, a scripting layer can be added to such programs, allowing them to inter-operate within a Python interpreter (Sanner 1999) (Humphrey, Dalke et al. 1996); (Huang, Couch et al. 1996). Using Python for scripting applications should be contrasted with using a custom scripting language (i.e. SVL (Santavy and Labute), BCL (Daelen), SPL (Tripos), batchmin (Schrodinger), MATLAB (Hanselman and Littlefield 2001), *etc.*). Such languages often lack extensibility and tend to be domain specific since they are created by developers whose strengths are in a specific application domain. Because of their specificity, these languages will not benefit from the input of a large community. Designing a language is better left to people whose primary occupation is designing languages.

While an excellent scripting- and glue-language, Python is also a general-purpose, object-oriented programming language. This aspect has been key for our purpose. It can be used as the primary language for the implementation of complete packages and applications which have the great advantage of platform-independence (i.e. the same source code runs on all platforms). Our software development effort has grown to over 15 packages written in pure Python, amounting to over 2000 classes and well over 800'000 lines of code. Such a colossal software base would be extremely difficult to manage without object orientation and hierarchical names spaces. Unfortunately, the often wrong intuition about where and how much performance is needed tends to deter programmers from using interpretive languages. This misconception about performance is difficult to overcome. However, designing and implementing components in the Python programming language first, provides several advantages: (1) The development cycle of a prototype is greatly accelerated through the use of a high-level language and the reuse of software components. It is not uncommon for a Python program to have 3 to 10 times fewer lines of source code than the same program written in C; (2) The design of software components can be validated rapidly. Python code can be tested as it is written. The ability to run code after only a fraction of an application or component has been implemented helps identify design problems early on; (3) Performance bottlenecks can be identified using Python's profiler and resolved by optimizing the Python code, or by implementing such parts in C or C++; Finally, (4) a potentially slow but working Python implementation is always available in the case that a C or C++ implementation cannot be loaded.

Software-components and their integration

We have developed a number of Python packages (Table 1). To illustrate our approach, we describe two independently developed software components: MolKit and DeJaVu, and demonstrate their integration for molecular visualization.

MolKit: This software component has no dependencies on other Python packages and provides objects for reading and writing common data file formats such as PDB, Mol2 and mmCIF, describing biological molecules. When a molecule is read, a hierarchical data-structure is built reproducing the natural hierarchy found in molecules such as proteins. Molecule objects can be queried and can also generate information such as atom types, covalent bonds, atomic radii, etc. This component is written in pure Python.

Package	Description
<i>MolKit</i>	Read, write and build hierarchical representation of molecular data structures.
<i>ViewerFramework</i>	Visualization application template. Uses <i>DejaVu</i> for rendering 3-D geometry.
<i>PyBabel</i>	Re-implementation of Babel 1.6 (molecular file formats conversion). Supports assigning of atom type and bond order, ring detection, Gasteiger charges, protonation.
<i>Mglutil</i>	Various packages containing mathematical functions, GUI widgets, etc.
<i>FlexTree</i>	Provides hierarchical, high-level representations of molecular flexibility.
<i>shapefit</i>	Protein-Protein docking software package based on the complementarity of smoothed protein shapes.
<i>Volume</i>	A package for representing and manipulating 3D regular grids of volumetric data.
<i>WebServices</i>	Generic support for web services.
<i>symserv</i>	A set of objects for describing point symmetries.
<i>Vision</i>	A component supporting visual programming.
<i>DejaVu</i>	An OpenGL-based general purpose 3D geometry rendering component.
<i>PyARTK</i>	A software component for virtual reality.
<i>PyQslim</i>	Python wrapper of the polygonal mesh decimation library QSLim (Garland 1999).
<i>Mslib</i>	Python wrapper of the MSMS library (Sanner, Olson et al. 1996) for computing molecular surfaces.
<i>UT-packages</i>	A suite of Python wrapper of C++ libraries for fast iso-contouring, volume rendering, pseudo density maps calculations, signed distance fields, meshing techniques, etc. (Bajaj, Pascucci et al. 1996).
<i>SFF</i>	Python wrapper of a C-implementation of the AMBER Force Field (Simple Force Field).
<i>SpatialLogic</i>	Python wrapper of a C library performing BSP-Tree-based CSG operations.
<i>Gle</i>	Python wrapper of the GL Extrusion library (Vepstas 1991).
<i>openglTk</i>	Python wrapper for OpenGL, generated on the fly from .h files.
<i>memoryObject</i>	A native extension for speeding up communication between C and C++ and Python code.
<i>PyAutoDock</i>	objects implementing the AutoDock forcefield.
<i>SurfDock</i>	Python wrapper of the protein-protein docking program SurfDock.
<i>Stride</i>	A wrapper of the Stride program for assigning secondary structure to proteins.
<i>BHtree</i>	A Python wrapper of a C library implementing binary spatial divisions.

Table 1: Most important Python packages developed in our laboratory. A shaded background indicates packages that are platform-independent (i.e. written entirely in Python).

DejaVu is our OpenGL-based, platform-independent, general-purpose 3-D geometry visualization component. It defines classes implementing objects such as Viewer, Camera, Light, ClippingPlane, ColorEditor, Geometry, etc. The Viewer class implements a fully-fledged visualization application. It provides control over a large number of rendering parameters including: user controllable depth-cueing; global anti-aliasing; perspective and orthographic projection modes; multiple light sources; as well as per geometry: rendering modes (points, lines, polygons, outlined), shading modes (flat, Gouraud), culling modes (back, front, none), arbitrary clipping planes, magic lenses that reveal the geometry only inside the lens, blending functions for transparency, etc. Each *DejaVu* Viewer object maintains a hierarchy of geometrical objects. Rendering attributes and 3-D transformations can be defined for any particular geometry in this hierarchy, or can be inherited from a parent. A Viewer object contains a virtual trackball object for rotating, translating and scaling. This trackball can be bound to any geometry, camera, light source, clipping plane, or texture.

DejaVu's set of geometry objects is extensible and currently includes: Polylines, IndexedPolylines, IndexedPolygons, QuadStrips, TriangleStrips, Spheres, Cylinders, Ellipsoids, Arcs3D, Arrows, Box, Points, CrossSets and TextLabels. Such objects can be instantiated and added dynamically to a viewer. *DejaVu* is written entirely in Python but relies on the presence of the Numeric package as well as *openglTk*, our OpenGL wrapper for Python.

Combining MolKit and DejaVu for molecular visualization: Figure 1 shows the code necessary for reading a molecule using the MolKit component and displaying a CPK model of this molecule (i.e. a single sphere per atom). Nine lines of Python code are sufficient for achieving basic molecular visualization with software components that have been developed independently of each other.

```

from MolKit import Read
mols = Read('1crn.pdb')
coords = mols[0].chains.residues.atoms.coords
radii = mols[0].defaultRadii()

from DejaVu import Viewer
vi = Viewer() # create a viewer

from DejaVu.Spheres import Spheres
s = Spheres('sph', centers=coords,
            radii=radii,quality=10)
vi.AddObject(s) # display atomic spheres

```

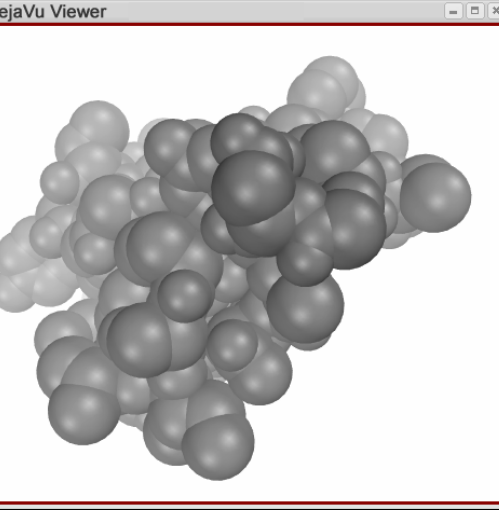


Figure 1: Nine lines of Python code are needed to read a molecule and display a sphere for each atom using two software components developed independently of each other: MolKit and DejaVu. The Read factory function from MolKit builds molecular objects from which data, such as atomic centers can be retrieved, and which can infer attributes such as atomic radii. A Viewer object from DejaVu provides a fully-fledged 3D geometry viewer in which geometry such as a set of spheres can be displayed.

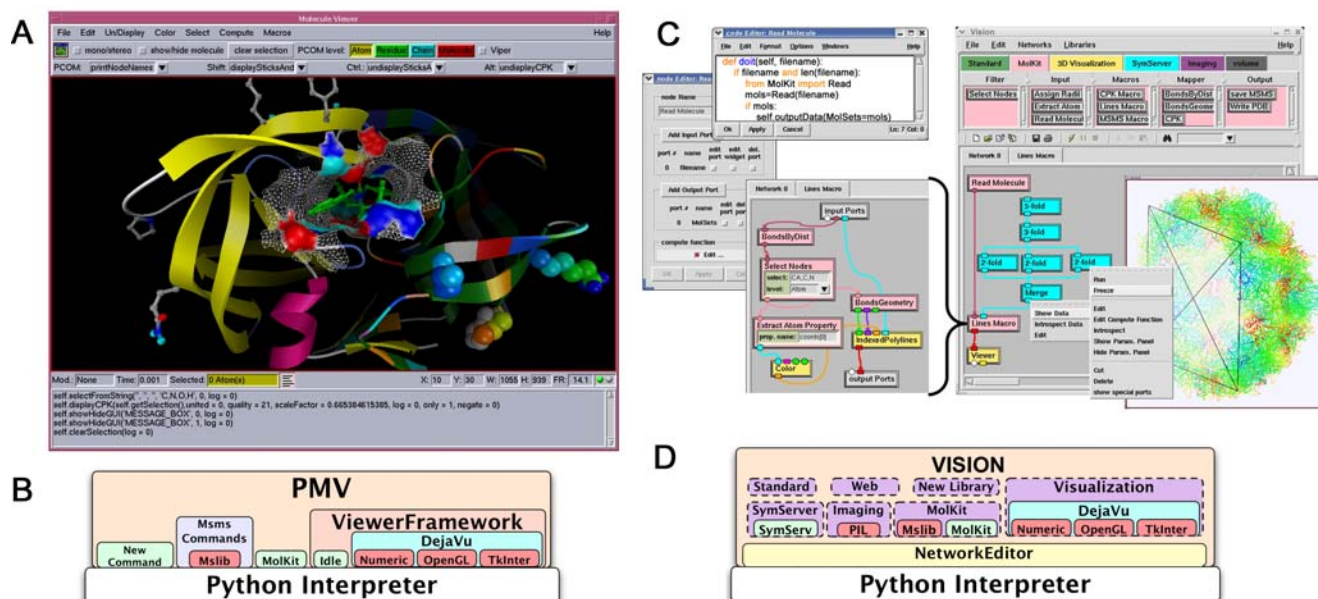


Figure 2: Two applications PMV and Vision built from, and sharing many software components. (A) PMV: a general purpose molecular visualization application. (B) Architectural layout of PMV. Nested boxes denote dependencies between Python packages. Packages with dark pink background are platform-dependent. (C) A molecular visualization application built using the Vision visual programming environment. A network used to display a viral capsid is shown. The sub-network embedded in the “Lines Macro” is shown as an inset. The “node Editor” allowing inspecting and modifying nodes interactively has been started on the “Read Molecule” node. (D) Architectural layout of Vision. Libraries of Vision nodes are shown with dashed outlines. Note the number of shared software components.

More complex applications: Using the software components described in Table 1, we have created complex applications, including: Pmv (Figure 2A), a general purpose molecular visualization and manipulation environment; AutoDockTools, a GUI for setting up and launching AutoDock-based automated docking calculations (Morris, Goodsell et al. 1998), and analyzing the results; and PyARTK, an augmented reality software in which we combine live video of physical models manipulated by the user with computer graphics enhancing the displayed video (Gillet, Goodsell et al. 2004). Both AutoDockTools and PyARTK are specializations of PMV.

Extensibility by users

While the Python interpreter is the foundation of all the applications we

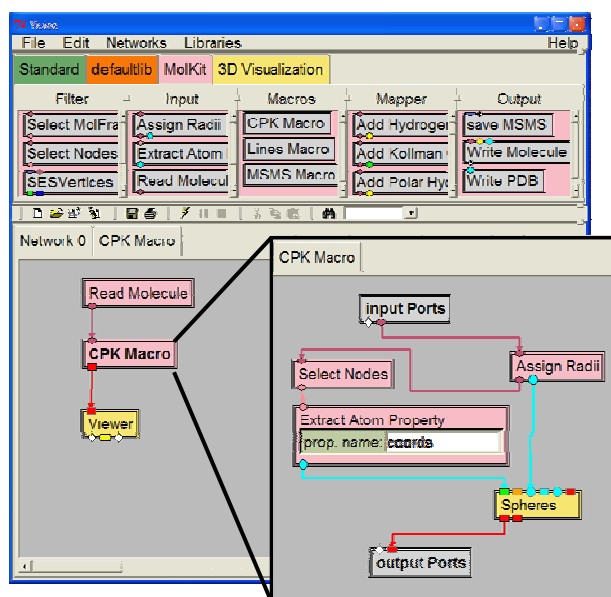


Figure 3: a simple Vision network corresponding to the Python script in Figure 1. The nodes are colored by the library they originate from. The CPK macro node is expanded in the inset. A user can create this network without any knowledge of the Python syntax or the data structures use to store molecules.

develop and always is accessible for advanced scripting, we have come to realize that asking domain experts such as biologists to learn a programming language, even as simple as Python, can be asking too much. In order to address this problem we have explored a couple of solutions. First, in our ViewerFramework component which is underlying PMV and AutoDockTools, all commands invoked through the graphical user interface generate a log string which corresponds to what a user would have to type in the Python shell to achieve the same result. This can help some of the more sophisticated users to prototype a series of operations, capture the log, and for instance place it inside a loop to apply these operations to a sequence of objects. However, the user still has to worry about proper indentation and other syntactical details of the Python language. In order to hide this level of complexity completely from the user we have explored the concept of visual programming. We have implemented a software component called Vision (figure 2C) (Sanner, Stoffler et al. 2002) which support dragging and dropping computational nodes onto a canvas and connecting their input and output ports for creating workflows and computational networks. Libraries of Vision node expose the functionality from other Python packages such as MolKit and DejaVu. Figure 3 shows a Vision network that is equivalent to the Python script shown in figure 1. This paradigm empowers the end user with the ability to extend the application with new functionality without having to learn the syntax of a programming language of getting intimate with the data structures used to store objects. A key difference between Vision and other visual programming languages such as Open DX (IBM 2002) or AVS (Upson, Faulhaber et al. 1989) is that Vision is a software component that can be reused in other applications. We have demonstrated its integration with PMV (Stoffler, Coon et al.

2003). All the domain-specific knowledge is in the Vision nodes, which are organized in libraries. The Vision nodes are light-weight wrappers for functionality otherwise available in Python. For instance the full implementation of the Read Molecule node in Figure 3 is to get the input (i.e. the file name), invoke the Read function from the MolKit package, and output the resulting Python object on the output port. In this sense, the node is merely an adaptor for the Read function in the Vision environment.

Lessons learned and pitfalls

Python's modular nature does promote the compartmentalization of code. However, for software components to achieve their full potential in terms of re-usability special care has to be taken beside the obvious idea described earlier which is to develop components in isolation. Here we provide a short list of common mistakes we have identified over the years. Simple mistakes include using global variables. This is always undesirable in object-oriented environments as it precludes the simultaneous use of multiple instances of objects referring to these variables.

One common pitfall in object-oriented languages is the tendency to create a large hierarchy of objects for all software developed by a group. While the idea is appealing from an esthetic point of view, it can be quite damaging from the software engineering aspect. The popular visualization toolkit VTK (Will Schroeder) provide a typical example of such a situation where the complete source has to be downloaded and compiled before any part of the toolkit can be used. Moreover, if the build fails half way it is very difficult to know whether or not the part that is of interest has built properly or not. We found that keeping the package hierarchy as flat as possible and reducing inter-packages dependencies to a minimum leads to much more reusable components. The same is true with data-types. In computational biology it can be very tempting to pass complex objects such as a molecule to a computational method which is only using a small fraction of the information stored in the molecule object. Reducing the information passed into the computational method to the strict minimum that is needed is a better approach. For instance in the example in Figure 1, we could have passed a molecule to the viewer, but this would make the visualization component aware of molecular objects. Alternatively, a molecule could have a display method, but here again; the MolKit component would become aware of DeJaVu. Instead, we are passing two numeric arrays, one for coordinates and one for radii, because that is all that is needed for drawing the spheres. By passing the simplest data types between software components, we reduce the dependencies between these components and minimize the data structures we impose onto other users of these components. Developers of libraries written in the C and C++ programming languages sometimes choose to make their code exit if input files are not found or an error is encountered. This is highly undesirable in our setting, as such a call would terminate the Python interpreter and hence our working session.

Conclusion

The concepts of modularity and compartmentalization of computational tasks are not new; however, time has shown that these concepts are poorly promoted by compiled languages. When programming in the latter languages, the goal is to write an application that produces the right result. There is little incentive to implement independent and therefore reusable software components for solving each particular sub-task. In fact, even if the software is properly compartmentalized, special "main" functions would have to be written to ascertain the independence and correctness of

the various components, which is rarely done. The intrinsic modular nature of Python on the other hand promotes the creation of components each implementing a specific functionality. These components can then be loaded into a Python interpreter, effectively extending the interpreter with new functionality. Moreover, these components naturally becomes available to any application running a Python interpreter.

We have described an approach in which the Python interpreter is the foundation of the software environment and serves as both an integration platform and a primary development language for new software components. Using this approach we have witnessed unprecedented levels of code reuse both within and outside our laboratory. While this approach has been very successful in addressing many of the shortcomings of software frameworks, it has also revealed a new set of challenges. While applications built from software components are desirable for many reasons, they increase the complexity of software distribution, as the right set of components (i.e. compatible versions of all components) has to be combined. Better support in the Python language for package management would help with this problem. Keeping track of inter-packages dependencies is also tricky. We have found that such dependencies are very easily, and often inadvertently created. To address this issue we have started to explicitly declare dependencies in all of our packages. These dependencies can be critical, if the extension cannot function without the package it depends on, or weak if the dependencies is only needed for a secondary. Our nightly unit tests monitor imports and report as errors imports of packages that are not declared in the dependency lists.

As modern computational biology moves towards the study of larger systems it will require the integration of computational methods and experimental data from a variety of scientific fields. Furthermore, the complexity of the models will require an unprecedented level of flexibility in the software tools to allow investigators to formulate and validate new hypotheses. The interactive and dynamic nature of Python, its simplicity, and ease-of-use, make this language an excellent choice for creating powerful modern software tools. The software tools described here are available at <http://www.scripps.edu/~sanner/software>.

Acknowledgments: The development of the Python-based software components described in this article was supported through the NSF grant CA ACI9619020 and the NIH grant RR08605. This is manuscript 18323 from The Scripps Research Institute.

REFERENCES:

- Adams, P. D., R. W. Grosse-Kunstleve, et al. (2002). "PHENIX: building new software for automated crystallographic structure determination." *Acta Crystallogr D Biol Crystallogr* **58**(Pt 11): 1948-54.
- Aho, A., B. Kernighan, et al. (1988). *The AWK Programming Language*, Addison Wesley.
- Bajaj, C., V. Pascucci, et al. (1996). "Fast isocontouring for improved interactivity." *Proc: ACM Siggraph/IEEE Symp on Volume Visualization, San Francisco, CA*.
- BioPerl. "BioPerl project home page." from <http://bio.perl.org/>.
- BioPython. "BioPython project home page." from <http://www.biopython.org/>.
- Coon, S., M. F. Sanner, et al. (2001). *Re-Usable components for Structural Bioinformatics*. 9th International Python conference (IPC9), Long Beach, California.
- Cowell-Shah, C. W. (2004). "Nine Language Performance Round-up: Benchmarking Math & File I/O." *OSnews*, from http://www.osnews.com/story.php?news_id=5602.

- Daelen, T. v. "The Insight Scripting Language." from <http://www.chem.ac.ru/Chemistry/Soft/BIOLANGU.en.html>.
- Delano, W. L. (2002). The PyMOL Molecular Graphics System. San Carlos, CA, Delano Scientific.
- Dybvig, R. K. (1996). The Scheme Programming Language Second Edition, Prentice Hall PTR.
- Garland, M. (1999). QSLim Simplification Software.
- Gillet, A., D. Goodsell, et al. (2004). A Tangible Model Augmented Reality Application for Molecular Biology. IEEE Visualization Vis04, Austin Texas.
- Hanselman, D. and B. R. Littlefield (2001). Mastering MATLAB 6, 1/e, Prentice Hall.
- Hinsen, K. (1997). The molecular modeling toolkit: A case study of a large scientific application in Python. 6th International Python Conference, San Jose, CA.
- Huang, C. C., G. S. Couch, et al. (1996). Chimera: An Extensible Molecular Modeling Application Constructed Using Standard Components. Pacific Symposium on Biocomputing.
- Humphrey, W., A. Dalke, et al. (1996). "VMD: visual molecular dynamics." J Mol Graph **14**(1): 27-8.
- IBM (2002). Open Visualization Data Explorer, OpenDX, IBM.
- Lien, D. A. (1986). The Basic Handbook: Encyclopedia of the BASIC Computer Language, Compusoft Publishing.
- Lutz, M. and D. Asher (1999). Learning Python. 101 Morris Street, Sebastopol, CA 95472, O'reilly & Associates.
- Morris, G., D. Goodsell, et al. (1998). "Automated docking using a Lamarckian genetic algorithm and an empirical binding free energy function." J. Comp. Chem. **19**(14): 1639-1662.
- Ousterhout, J. (1994). Tcl and the Tk Toolkit. Addison-Wesley.
- Raymond, E. S. (2001). Why Python? Linux Journal.
- Sanner, M. F. (1999). "Python: a programming language for software integration and development." J. Mol. Graph. Model. **17**(1): 57-61.
- Sanner, M. F., A. J. Olson, et al. (1996). "Reduced surface: An efficient way to compute molecular surfaces." Biopolymers **38**: 305-320.
- Sanner, M. F., D. Stoffler, et al. (2002). ViPER, a visual programming environment for Python. Proceedings 10th International Python Conference, Alexandria, VA.
- Santavy, M. and P. Labute. "SVL: The Scientific Vector Language." from http://www.chemcomp.com/Journal_of_CCG/Features/svl.htm.
- Schrodinger MacroModel.
- Schuerer, K. and C. Letondal. (2004). "Python course in Bioinformatics." from <http://www.pasteur.fr/recherche/unites/sis/formation/python/>.
- Steele, G. L. (1990). Common Lisp the Language, 2nd edition, Digital Press.
- Stoffler, D., S. I. Coon, et al. (2003). Integrating biomolecular analysis and visual programming: flexibility and interactivity in the design of bioinformatics tools. Proc. Thirty-Sixth Annual Hawaii International Conference on Systems Sciences, Waikoloa, Hawaii, Computer Society Press.
- SWIG Simple Wrapper Interface Generator (SWIG).
- Thomas, D., Hunt, A. (2001). Programming Ruby - The Pragmatic Programmer's Guide, Addison Wesley Longman, Inc.
- Tisdall, J. (2003). Mastering Perl for Bioinformatics, O'reilly.
- Tripos SYBYL, Tripos Inc., .
- Upton, C., T. Faulhaber, et al. (1989). "The Application Visualization System: A Computer Environment for Scientific Visualization." IEEE Computer Graphics and Applications. **9**(4): 30-42.
- Vepstas, L. (1991). GL Extrusion library.
- Vriend, G. (1990). "WHAT IF: A molecular modeling and drug design program." J. Mol. Graph. **8**: 52-56.
- Will Schroeder, K. M., Bill Lorensen The Visualization Toolkit, An Object-Oriented Approach To 3D Graphics.