Python

Success Stories

Part two

12 more true tales of flexibility, speed, and improved productivity

AstraZeneca Uses Python for Collaborative Drug Discovery

Verity Ultraseek: Building Successful Enterprise Solutions with Python

Carmanah Lights the Way with Python

Maritime Industry Increases Efficiency with Python

ForecastWatch.com Uses Python To Help Meteorologists

Cog: A Code Generation Tool Written in Python

ERP5: Mission-critical ERP/CRM with Python and Zope

EZTrip.com Chooses Python for Enterprise Integration

Frequentis TAPtools®—Python in Air Traffic Control

LoveIntros Uses Python to Help Northwest Singles Click

Python and Zope in the EZRO Content Management System

Suzanne: Python Handles Critical Data in a Domain Name Landrush

O'REILLY®

Python Resources

International Python Meetup Day
python.meetup.com

Python Software Foundation
python.org/psf

Pythonology
pythonology.org

O'Reilly Media, Inc.
python.oreilly.com

O'Reilly Network Python Dev Center
onlamp.com/python

Python Advocacy HOWTO
amk.ca/python/howto/advocacy/advocacy.html

Python Starship
starship.python.net

Vaults of Parnassus: Python Resources
py.vaults.ca/parnassus

Daily Python-URL
pythonware.com/daily/index.htm

Pyzine
pyzine.com

Python Journal
pythonjournal.cognizor.com

Introduction

by Alex Martelli & Guido van Rossum

Python is widely admired for its simplicity and elegance. However, these undeniable qualities must not overshadow Python's usefulness: with Python, you *can get the job done*!

Python is highly scalable–suitable for large projects as well as for small ones, excellent for beginners yet superb for experts. It is stable and mature, with large and powerful standard libraries, and a wealth of third-party packages for such widely varied tasks as numerical computation, Internet and web programming, database use, graphical user interface development, XML handling, image processing, three-dimensional modeling, and distributed processing with CORBA or SOAP. Python plays well with others, easily integrating with either C/C++ or Java™ you can extend it, you can embed it in your existing applications, and you can use it to integrate multiple applications.

This booklet illustrates Python's usefulness by presenting a small sample of Python success stories, real-life examples where Python played a central role in the development and delivery of working, useful software systems of substantial size. The systems are in a wide variety of application areas, and they go up the scale, all the way to enterprise-wide integrated systems. Thus, you'll see that Python is not just a scripting language (not limited to coding small and simple scripts, even though it's quite suitable for such frequent, small tasks), nor is its usefulness confined to any one application area. In the hands of experienced software developers, Python offers high productivity for projects of all sizes, in all application areas.

Python's usefulness comes exactly from its simplicity and elegance, harmonized into a seamless whole by its highly pragmatic design. Engineers since the Roman Vitruvius have recognized that a good design exhibits solidity (firmitas), delight (venustas), and usefulness (utilitas). Italian architect, Leon Battista Alberti showed during the Renaissance that these characteristics are achieved by harmony (concinnitas) the art of ordering disparate parts into an organized whole. Python's design makes these theories come to life, no less than Alberti's Tempio Malatestiano.

Managers love Python because its simplicity minimizes programmers' learning efforts, but the same simplicity underpins Python's solidity, avoiding any bugs related to obscure, misunderstood, and ad-hoc features, in both the language's implementations and programs coded in Python. Designers are drawn to Python's elegance, which allows concise and readable expression of design ideas, but the same elegance and readability ease maintenance, modification, and reuse for programs coded in Python. The ease of interfacing Python to existing C/C++ or Java libraries and applications, plus Python's large standard library and the wealth of existing third-party Python extensions, combine to make Python the ideal language for challenging integration tasks, no less than for "green-field" development projects. All together, these characteristics make Python the language of choice for high-productivity software development—one of the most rapid development environments on the planet.

Once this booklet has whetted your appetite, you can explore Python further, starting at **www.python.org**. For many excellent books about Python, and for additional Python success stories, see **python.oreilly.com**.

AstraZeneca Uses Python for Collaborative Drug Discovery

Scott Boyer, Andrew Dalke, and Pierre Bruneau°

About the Authors

Scott Boyer is a principal scientist in the Enabling Science and Technology section of AstraZeneca Discovery R&D, Mölndal, Sweden.

Scott received his Ph.D. from the University of Colorado, Boulder and has worked at both Pfizer and AstraZeneca.

Andrew Dalke is the founder of Dalke Scientific Software, LLC, a software consulting company located in Santa Fe, New Mexico, USA. Andrew has been developing computational chemistry and biology software since 1992.

Pierre Bruneau studied chemistry at the Ecole Nationale Supérieure de Chimie de Strasbourg. He is now a principal scientist in the Cancer and Infection Research Area of AstraZeneca Discovery in Reims, France.

www.astrazeneca.com

Introduction

AstraZeneca is one of the world's leading pharmaceutical companies. With over 54,000 employees world-wide, it provides innovative, effective medicines designed to fight cancer, provide pain control, heal infection, and fight diseases of the cardiovascular, central nervous, gastrointestinal, and respiratory systems.

Finding a new drug often takes over a decade and more than $800 million. A big problem early in the process is identifying those candidates more likely to be good drugs from the vast universe of possible molecules.

Computational chemists have developed many techniques to predict molecular properties. These can be used to evaluate the likelihood that a molecule will be stable in the stomach (for pills that are swallowed), and that it can travel through the blood stream, cross the cell membrane, and eventually be broken down and eliminated, all without being too toxic to the body.

If these computational techniques were good enough there would be no need to do actual experiments. But today's computer models cannot fully characterize a molecule's behavior in the body, nor replace the intuition of a skilled pharmaceutical chemist. Real molecules must still be tested in the laboratory to see how they react.

To save time and money on laboratory work, experimental chemists use computational models to narrow the field of good drug candidates, while also verifying that the candidates to be tested are not simple variations of each other's basic chemical structure.

Process Improvements Needed

Much of the work on drug identification actually takes place through collaboration between many research groups scattered around the world. As part of this process, experimental chemists send a list of compounds to the computational chemist, who works on the data set and sends back the results.

Historically, experimental chemists were forced to rely on computational chemists and other staff to run computer predictions. Each prediction technique required running a separate program, some commercial and others developed in-house by different groups around the company, and each program had its own set of inputs, options, configurations, and failure behaviors. An experimental chemist usually didn't have the training to work with them, which meant that the computational chemists were forced to take time out of their work on developing new techniques to run routine models.

In 2000, AstraZeneca wanted to improve this process so that experimental chemists could make better computational predictions on their own so that the research of the computational chemists could progress at a faster rate, and make its way into the lab more quickly.

Pierre Bruneau, a principal scientist at AstraZeneca, had worked on this problem while at Zeneca, which merged to form AstraZeneca. He developed a web-based interface called H2X, named after the allied navigation systems used during the second world war. H2X was based on an in-house molecular property calculator called Drone. This system used a Perl script which computed some of the simpler molecular properties by calling the appropriate prediction program, usually through a wrapper written in Perl, *csh*, or a domain-specific control language.

Python Chosen

H2X using Drone was a successful experiment and it was used by many people. In 2001, AstraZeneca decided to develop it further and brought in Andrew Dalke as a consultant to improve the back-end code by making it more robust, extensible, and maintainable. Dalke, a well-known advocate for Python in computational chemistry and biology, convinced the group that Python was the appropriate language for the next generation back-end, which was named PyDrone.

Python was chosen for this work because it is one of the best languages available for physical scientists, that is, for people who do not have a computer science background. Many other powerful and expressive high level languages exist, including Perl, Lisp, Scheme, Ruby, CAML, and Haskell. Of all these, Python is one of the few that is based on research into usability and the factors that make a programming language easy to learn and use. Yet Python was also designed to solve real-world problems faced by an expert programmer. The result is a language that scales well from small scripts written by a chemist to large packages written by a software developer.

Python's Error Handling Improves Robustness

The first iteration of PyDrone refactored the existing Perl code into more appropriate functions, classes, and modules while translating the code base into Python. Refactoring the Perl code without moving to Python would have produced comparable architectural results, but Python's explicit error handling and stronger type checking helped to considerably improve the code's robustness.

The current version of PyDrone uses about 20 different external binaries and scripts to predict various molecular properties. When an external program works correctly, then the output is easy to parse into the desired results. However, these programs don't always work correctly, are not fully documented, and it's often hard to determine all the possible failure cases from the outside. To compensate for this, the PyDrone developers wrote tests to anticipate as many error cases as possible, but it was impossible to rule out additional unexpected error cases after deployment.

From experience dealing with this issue first

in Perl (Drone) and then in Python (PyDrone), we found that Python is better at catching many types of errors and at managing unexpected problems in a deployed application. This is a result of the way in which the two languages approach error handling in general. For example, Perl's I/O routines are quiet and failures must be checked explicitly, usually with the "or die" idiom. A conscientious programmer will always add those, but they take up space, are easy to forget, and hard to test. In contrast to this, Python functions are noisy and almost always raise an exception automatically when there is a problem in code.

After rewriting in Python, we initially thought this noisy behavior was a nuisance because Python kept raising exceptions and stopping where the old Perl code had kept on going. However, we soon found that nearly every exception indicated a previously undetected error case for which we needed to add new error handling code. Python was helping us find problem spots and preventing us from letting silent errors into our data.

One example of an error case that Python uncovered for us was caused by an external prediction program that would usually return a numerical error code but in some cases was found to return the string "error" instead. In Perl, strings and numbers are converted automatically, for example the string `"123"` becomes the integer 123. Unfortunately, Perl also converts non-numerical strings, such as `"error"`, to `0`. As a result of this, Drone was treating `"error"` as a successful return value, leading to incorrect results. Python's stronger typing uncovered the error as soon as the rare case that caused it was executed.

Another way in which Python helped us to improve our code was by its inclusion of a complete stack traceback with each exception report. We found this very useful in helping us understand the source of a problem without needing to run a debugger or add extra code instrumentation. This feature of Python was particularly helpful in remote debugging of rare cases. Andrew Dalke is in the United States and Pierre Bruneau isin France. When an error occurred, Bruneau's email with the traceback often contained enough information to pinpoint and fix the problem.

Adding Powerful Extensibility with Python

The next stage in PyDrone's development was to improve its extensibility. Some molecular properties depend on other properties. For example, a molecule's mass depends on its composition. The older Drone code maintained these dependencies manually with a set of `"if"` statements that specified which prediction routines should be called, and in which order, during execution of an analysis. In this approach, adding new dependencies soon led to a combinatorial nightmare.

To solve this problem in Python, we developed a simple rule base which acts like a Python dictionary. It contains a data cache and a mapping from property name to prediction function. If a requested property name (the dictionary key) is in the cache, we reuse it. Otherwise, we find the associated function, call it to compute the value, store the result in the cache, and return it. The functions themselves are given a reference to the Properties manager so they can recursively request any additionally needed dependencies. To add a new prediction we register the new function in the function table—and let the functions themselves handle the dependencies. The cache is needed because some properties are expensive to compute or are needed by many other properties.

The resulting new architecture made a simple but profound difference to the project. We now have a single system that can accommodate all current and future prediction methods, that computes only the minimum needed to yield the requested results, and that is easy to understand and maintain. Before we built it in Python, several people in the company had argued it impossible to build such a system at all.

The Benefits of Python's Type System

The PyDrone architecture could have been implemented in many languages, but Python's dynamic typing made it much easier to build our Property manager. Some molecular properties are numbers, others strings, or lists and dictionaries, and still others are class instances. A statically typed language would have required extra hassle to allow a mixture of return types to be inserted into the Property manager. Even Perl, which is also dynamically typed, would have required some way to distinguish between references to a `$scalar, %hash`, or `@list`. In Python it just worked, and we could mix the data types of the keys in the Property manager dictionary without any extra effort at all. Yet, as described above, Python does at the same time provide sufficient data type checking to find many kinds of common type mismatch errors.

One of the factors that made our Property manager so successful was that Python lets user-defined types emulate the behavior of built-in types. Our Property manager acts very much like a lookup table that maps property name to value, so we designed it to emulate a Python dictionary. In Python, this is done by implementing specific special methods such as `__getitem__()`, `__setitem__()`, and so forth. By emulating a dictionary, nearly every other Python function that operates on a dictionary would work with our manager. It also made the code easier to understand and debug because the syntax and point-of-call usage fit naturally with what people expect.

Python facilitated our Property manager implementation in other ways as well. One PyDrone feature that had been requested by users was the ability to describe a new prediction using an equation based on existing properties. For example, an equation might look like:

```
0.34 + (0.78*CLOGP) -
(0.018*HBA) - (0.015*HB_TOT) -
(0.11*MM_HADCA) - (0.017*MM_QON)
+ (0.012*VDW_POL_AREA)
```

where the variables are keys in the Property manager. This was quite easy to implement in Python, and we would be hard pressed to find a language that makes it any easier. Python's mathematical expressions are almost identical to the standard form used in the sciences, so we could use Python's `"eval"` statement to parse and evaluate the user-defined expressions. Because our Property manager acts like a Python dictionary, it could (at least in theory) be provided directly to the eval statement as the locals dictionary used for variable lookup during expression evaluation.

As it turned out, for performance reasons, the `eval()` implementation in Python accepts only built-in dictionary types and not an emulated mapping type, so we had to engage in some extra trickery to make our on-demand dependency system work with equations. Nevertheless, the entire implementation was quite easy.

Results

PyDrone took about 3 months of development time, another 3 months of QA, and 3 weeks of documentation time to produce about 5,600 lines of finished Python code.

Overall, PyDrone has been a wonderful success for AstraZeneca. As a result of using Python, we were able to quickly and easily develop a great tool that is both very simple to use and that adapts well to new prediction methods.

The biggest problem we've had with Python is that relatively few people at AstraZeneca use it for development. The IT group prefers either Perl (systems people) or Java™ (architecture people) so we occasionally get requests to rewrite parts of the project in one of those languages. Even so, we have found developers are interested in learning Python, especially when they see comparisons of development time and effort, resulting code size, and other metrics.

Verity Ultraseek: Building Successful Enterprise Solutions with Python

Ryan Weisenberger

About the Author

Ryan Weisenberger is a software developer and project lead for Verity Ultraseek. He has been involved with the product for four years. Weisenberger is originally from Southern California, and has been working in the high-tech industry for seven years.

www.verity.com

Background

Successful information search is mission-critical. It increases worker efficiency and saves money. On an external web site, it also increases customer satisfaction and helps retain users to the web site. The average company can waste millions of dollars because its employees cannot locate and retrieve the information that is needed for their jobs.

Infoseek released the original version of Ultraseek Server on March 31, 1997. It was built by wrapping Infoseek's core search technology with Python, in order to make it a complete search engine for enterprises. Verity Inc. acquired the Ultraseek product through its recent acquisition of Inktomi's enterprise search business in December 2002. The product is now known as Verity Ultraseek.

Verity Ultraseek is an advanced search engine that gives employees of departments and small to mid-size enterprises the power to quickly find the information they need to get their jobs done.

Downloadable for free 30-day trials, Verity Ultraseek can be downloaded and installed on a corporate intranet or web site typically within 20 minutes, and delivers extremely low total cost of ownership. The software's administrative and user interfaces are also extremely user-friendly.

Choice of Python

Python, a language that was previously used to build the Ultraseek Server and several back-end tasks at Infoseek, was used in creating the Ultraseek spider, as well as the GUI that provides the end-user search box, results pages, and all administrative functions.

Python was chosen for several reasons:

1. Python (like Java™) is a modern language with objects, modules, threads, exceptions, and automatic memory management. At the time, both C and C++ were rejected as missing at least some of these features.
2. The solution needed to be multi-threaded. Neither Perl nor Java was chosen as the solution at the time. Perl was not considered to be as robust. Java did not allow control over the core interpreter. The language was subject to the quirks of the particular Java™ Runtime Environment (JRE) installed on each system. Although a JRE could have been bundled with the product, it would have substantially bloated the size of the download.
3. Python reference counting was found to be superior to Java garbage collection, especially in an environment where response time was critical. Java's approach resulted in brief periods where the search engine was unavailable for several seconds at a time. Python's reference counting method allowed incremental recovery of unused memory, which meant that server threads would always be available to do a search.
4. From previous experience with the language, we knew that Python can provide faster development, smaller code volume, access to a rich set of libraries, and fewer memory management bugs to deal with. Python's built-in serialization features would also simplify saving and restoring state in the engine.
5. Python could be used to build a set of internal interfaces, and as a scripting language would allow customers to programmatically alter the behavior of the indexing spider and parser to meet local requirements. This extensibility without the need to learn proprietary scripting languages or APIs has been highly regarded by many customers.
6. Python could be embedded into the HTML pages that make up the product's user interface and used during page generation. This was before JSP™ existed, so our evaluation of Java didn't include that technology.

Some other Python benefits:

7. Python is open source which means we can fix bugs in the interpreter ourselves.
8. Ease of integration with C allows for connections to complex C functionality.
9. High syntax readability helps decrease development time.

Implementation

Verity Ultraseek now runs on Windows NT®, Linux®, and Solaris™.

The overall architecture includes a built-in HTTP server with an underlying spider to collect web documents, and a query engine to serve search results back to the user. The interface, including the administrative console, is served over the built-in HTTP server using Python-scripted HTML. Some of the modules, such as the search spider, GUI, and HTTP server, were written in Python. For performance reasons, others, such as the query engine, indexer, and HTML parser, use the Python/C API to incorporate functionality written in C. The C-based search engine core was also wrapped as a Python extension module. Some C++ is used in the interfaces to key third party technologies. Python's flexibility in this regard gave us a wide degree of flexibility in integration.

Other software packages that are integrated into the product include Basistech's JMA, KMA, and Rosette; DataDirect's ODBC Connect; XLT from InXight; KeyView document filters from Verity; and Adobe Acrobat® libraries.

Results

Python was facilitated in a cost competitive environment. Today, more than 2,500 worldwide customers have deployed Verity Ultraseek that uses the Python language.

The source code, not including third party technology, is approximately 80,000 lines of

Python, and less than 150,000 lines of code overall. This includes the Python-scripted HTML that makes up the user and administrative interfaces which represents approximately 40,000 lines of the total.

Conclusion

This project has been incredibly successful. The Verity Ultraseek development team has been flexible and responsive in developing, implementing, and improving features according to customer feedback. Python has been very useful in keeping the development environment simple, flexible, and easy to learn.

During the course of development, only a few disadvantages were found in Python. One was inability to detect some errors that would typically be uncovered during compilation (although static error checking tools like PyChecker are now available for this problem). Another was a result of the very seamless way in which Python casts between object and data types. While this feature allows for quick and flexible code, it is the responsibility of the programmer to make explicit the data types that a function expects. A standard coding style that names variables according to their expected data type is an easy way to address this problem.

Carmanah Lights the Way with Python

George Belotsky and Thomas Major

About the Authors

George Belotsky is a software architect who has done extensive work on high performance Internet servers as well as hard real time and embedded systems. His technology interests include C++, Python, and Linux. Belotsky has written a number of articles (see oreillynet.com/pub/au/1204) including a series on Python and Network I/O. He is also the author of the Flightdeck-UI open source project (openlight.com/fdui). You can reach him at questions@openlight.com.

Thomas Major is the product development manager at Carmanah, an electrical engineer by schooling with broad product design experience acquired at Visteon and Philips Electronics. His interest started in analog circuit design, later embracing digital, software, and embedded design.

www.carmanah.com

Introduction

This is a story about how Python's elegant design can make the language useful in an unexpected way.

Carmanah Technologies Inc. was conceived in the middle of the Pacific Ocean. The founder, David Green, was sailing his boat from Fiji, bound for Victoria, British Columbia. He was running low on battery power for his navigation lights, and had an insight.

The eventual result of that mid-ocean idea was the world's first self-contained, fully autonomous marine navigation light. During the day, each device uses solar power to recharge its integrated batteries. The light then operates from the stored charge at night. In place of conventional bulbs, Carmanah's systems use long lasting, high-efficiency LEDs. The overall result is an exceptionally rugged device, sealed against the elements and requiring no maintenance.

On occasion, lost Carmanah lights have drifted for thousands of miles upon the ocean currents, only to be recovered later–in full working order. This sort of reliability has made them famous amongst waterway authorities, sailors, and the coast guards of various countries.

The technology born of the harsh marine environment has proven useful in many other areas, and today Carmanah is the world leader in solar-powered LED lighting. The company manufactures a whole line of such lights for a variety of different uses including transit, roadway, railway, industrial markers, and airfield illumination, as well as the original marine application. The lights are sold all over the world, and must often withstand extreme conditions (the open ocean, desert climates, the far north, urban vandals, etc.). Carmanah's lights have even been the subject of a documentary on national television.

Electric lighting has become so commonplace, it seems like a simple idea. An autonomous, self-contained light, however, turns out to be a complicated thing indeed. The amount of usable solar radiation varies with the

weather, the seasons, one's position on the earth, and the orientation of the solar panel. The battery state of charge must be carefully managed to ensure long life and correct operation. Available power has to be monitored, and possibly rationed through the night.

Depending on the application, the light may also have to be programmable to emit internationally recognized flash codes, react to user input, etc. In even more complex situations, wireless networking is required to allow the lights to communicate with each other, or with a central base station.

A great deal of mechanical, electrical, electronic, and optical design is required to create these lights. As is typical of modern complex devices, an embedded software program running on a microcontroller operates each unit, making it come to life. Like a miniature version of the Monolith from *2001: A Space Odyssey,* each light maintains itself, ready to perform its function whenever the need arises.

The Future of Practical Computing

Current practical computer applications are dominated by the PC. Yet, just like the mainframe before it, the PC will lose its central place in the use and deployment of computer technology. Embedded systems—computers that are part of other devices—will be the most prevalent in the future.

The mainframe is far from being obsolete today, and the PC will likewise remain important. Most computers, however, will be incorporated into something else, rather than standing alone. This process has already begun. Automobiles employ multiple embedded systems, some of which communicate with each other. Embedded systems also operate many household appliances. Such systems are likewise very important in industry, where they form a crucial element of many instruments and tools.

The catalyst for the future expansion of embedded systems is the rapid advance of network technology. As the hardware becomes less expensive, smaller, faster, and more efficient in the use of electrical power, networks of embedded devices will proliferate and grow. Such devices—in collaboration with one another—will control homes, offices, and industrial facilities. Preparations for this new world of computing are underway now, as evidenced by the Cambridge-MIT Institute's Pervasive Computing Initiative (www.cambridge-mit.org/cgi-bin/default.pl?SID=6&SSID=76&SSSID=446&SSSSID=366). The initiative has also been covered in a BBC story (news.bbc.co.uk/1/hi/technology/3583479.stm).

The emergence of Pervasive Computing will make it necessary for each system to be very low maintenance. When every user requires hundreds of devices, it is simply not feasible to service them all on a regular basis. Of course, fully self-contained systems that manage their own power would be ideal, because changing batteries or attaching wires is a serious challenge if it needs to be done on a large scale. Thus, the miniature Monolith becomes a swarm.

The Importance of Python

Large scale deployment of embedded systems demands inexpensive components. Considerations such as small size, high reliability, and low power consumption are also very important. Specialized processor chips, called microcontrollers, have been developed to meet these goals. Combining CPU, memory, and peripherals (such as UARTs) on a single chip, modern microcontrollers are marvelous devices. These features, however, come at a significant price. A typical microcontroller has only a few hundred bytes of RAM, several K of ROM (to store the program), and orders of magnitude less processing capability than a conventional desktop microprocessor. Hardly an environment for running Python!

There are projects to adapt Python for embedded applications, but they require significant resources on the microcontroller, and are still in the very early stages. Surprisingly, however, it turns out that standard Python is of tremendous value throughout an embedded system's lifecycle. This is because the highly resource-constrained nature of embedded devices make them dependent on standard PCs for many tasks—both during development and during deployment.

For example, embedded software is compiled on conventional desktop systems, and the resulting object code is then loaded onto the target microcontroller. Another example is troubleshooting a device in the field, which usually requires additional hardware to run a diagnostic utility. Ordinary laptops are a very attractive platform for this application, due to their ready availability and relatively high level of standardization.

Thus, a major part of any embedded software development effort is writing the required support code to run on a standard PC. There are quite a few languages available for this task, but the advantages of Python are many. Python is quite easily learned by people from various programming backgrounds, such as Java™, C, or Visual Basic®. After becoming familiar with Python, development proceeds very quickly—perhaps faster than with any other language. At the same time, Python lends itself to the creation of highly readable, compact, and well-structured code.

Python's particular mix of features also helps embedded developers be more effective when programming a PC. While these developers are well familiar with C (by far the most popular high level language for embedded systems), a C program written for a standard desktop or server is quite different in style from one for a microcontroller.

The compactness of Python programs is especially important, because embedded developers have, by necessity, learned to express their designs in a very small amount of code. Python's automated memory management also helps, because many embedded developers have little experience with dynamic memory allocation—a technique that is not practical in most embedded environments. In addition, the object-oriented facilities of Python are simple, powerful yet not coercive. This allows embedded developers (who are often less familiar with OOP) to gradually adopt the object-oriented paradigm in their work.

As embedded systems grow in complexity, the advantage of using Python to augment traditional techniques becomes more and more important. At Carmanah, Python adoption (which began with the crosswalk traffic beacon, a sophisticated device that includes wireless networking) has spread to several key areas of the embedded system's lifecycle.

A Python program controls the software build process, allowing firmware for different products to be put together from a large number of shared components. The build system is simple yet a lot more flexible than makefiles, as well as easier to customize, configure, and extend. Unlike the build tools supplied by compiler vendors, the Python-based build system can work with different compilers.

Python is also used for stress tests and unit testing—an aspect of development particularly vital in embedded systems, due to the inherent difficulty of upgrading devices once they are in the field. Additional uses of Python, such as for control panels and code generation, are being considered as well.

One very exciting application of Python at Carmanah is in the role of a device simulator. A simulator can act as a node in an embedded network, while displaying the internal system state via animated images on the screen. Simulators are very valuable during the early stages of

an embedded project, when little actual hardware is available. By taking the place of missing devices, simulators can allow software development to continue even before the hardware design is completed.

At Carmanah, Python has been used not only by experienced engineers, but by student interns as well. Even interns with little prior programming experience can accomplish quite a lot with Python, while requiring less supervision than other languages demand.

Conclusion

The exciting work of creating self-contained, autonomous devices continues at Carmanah. As the swarm of micro-Monoliths becomes reality, the importance (and complexity) of the embedded software grows. By making such software much easier to develop, test, control, and deploy, Python is really lighting the way to the future of computing.

Maritime Industry Increases Efficiency with Python

Henrik Wimmerstedt

About the Author

Henrik Wimmerstedt is product manager for the Tribon Developer's Toolkit and chairman of the Tribon Developer's Network. He joined the company in 1997, after studies in naval architecture.

www.tribon.com

Introduction

Tribon Solutions develops, markets, and supports CAD/CAM/CIM software solutions, with the mission of increasing overall efficiency in the maritime industry.

For more than 30 years the company has provided shipyards, design agents, and maritime equipment suppliers with new ways to improve cost efficiency, quality, and performance. Our solutions are proven to generate time savings and to increase speed to market.

With its head office in Sweden and offices in the Republic of Korea, China, Japan, Germany, the UK, Russia, Singapore, and the USA, Tribon Solutions serves the global maritime industry and customers in more than 43 countries.

The Need for a Ship Building API

The building of a ship is initiated by the need to transport goods, people, or firepower. The particular need dictates the dimensions, layout of the cargo area, engine power, and other basic attributes for the ship. The design for the ship then meshes these requirements with strength and safety regulations. Because most transport needs are unique, many ship designs are built only once or only in a small series.

The Tribon software suite covers the ship building process from the first concept to the finished ship. The design and construction of a ship involves a high degree of concurrent engineering. To solve this problem Tribon uses a Product Information Model (PIM), which is the central repository and single source of information for designers, planners, administrators of material, manufacturing personnel, and others working on design and construction.

Even though most ship designs are unique, shipyards try to lower cost by using as much standardized and parameter-driven design as possible. The problem for a software vendor is that design principles are different at each yard due to factors such as ship type, production facilities, prior experience, and national regulations and standards.

Tribon's solution to this problem was to make it easy for shipyards to develop their own functionality based on Tribon core technology, including the Tribon PIM. To achieve this, Tribon Solutions had to create an API that was platform independent, easy to use, had all the strengths of a modern programming language, and was extendable and embeddable.

Choosing Python

Tribon began work on its API in 1995. Two different paths were considered at this stage: Either to publish directly the libraries used by Tribon, or to create a wrapper on top of existing code.

The first approach would make all our functionality available to the user, but users would have to use the same development environment as Tribon Solutions, change compiler versions when Tribon Solutions did so, and so forth. This would have been an expensive and complex solution, only usable by the largest shipyards in the world, those that had their own large IT and development departments.

The second approach was preferable, as long as a tool could be found or developed that covered most of the given criteria. Tribon already had a geometry macro language that was developed in-house, but to extend it to the desired level of functionality would have been costly to implement and maintain. The remaining option was to find a third party solution that fulfilled the API's needs.

During investigation of options, Python was discovered quite early when a member of the development team read about Python in a computer magazine. After some initial experimentation there were really no other contenders. Python had it all. It was a beautiful programming language that was extensible, embeddable, platform independent, and had no license cost.

When it came to incorporate Python into the Tribon software, we found the integration to be quite easy and problem-free, and it was achieved with very little effort. The result of this merge between Tribon and Python was named Tribon Vitesse, and the first version of Python used was 1.2.

Experiences

During the last seven years Tribon Solutions has been able to move with the updates of Python without any major inconvenience to our customers or ourselves. Today we have customers that have developed hundreds of modules over the years. Python's platform independence, and the fact that it is an interpreted language, have been a major benefit to customers that have migrated from the VMS and Unix® to Windows®. They have been able to port their code with no changes or at most only minor changes.

Today Tribon Solutions has customers that have, by utilizing the power of Tribon Vitesse, been able to reduce design time of certain complex ship structures from four weeks to two days, while improving overall quality. This enormous reduction in design time has been possible by automating more of the design, calculations, information search, and result checking. Other customers have created entire applications, based on Tribon technology, that fit perfectly into their way of working and thinking.

Python is also used to customize the Tribon system through the implementation of hooks and event-driven triggers that allow the user to control the graphical user interface, information display, design standards, and much more.

In the fall of 2002, as a further service to customers, Tribon Solutions formed a Developer's Network that supports third party vendors building maritime solutions based on Tribon Vitesse.

Summary

Today Tribon Vitesse consists of over 500 functions and more than 50 different classes, and it is still growing. Development is to a large extent driven by customer requests.

Python has proven to be a perfect tool for creating an API to existing applications because it is:

- Extendable
- Embeddable
- Platform independent
- Easy and logical to learn, even for non programmers
- A beautiful programming language

Visit Tribon Solutions on-line at www.tribon.com for more information or contact info@tribon.com.

ForecastWatch.com Uses Python To Help Meteorologists

Eric Floehr

About the Author

Eric Floehr specializes in large-scale data collection and analysis, and consumer Internet software, having worked with such companies as MCI, Datalytics, and Battelle. He holds a degree in computer and information science from The Ohio State University. He has been in the technology industry for over 13 years, and is founder of Intellovations, LLC, a technology consulting company focused on building software for discovery—challenging projects that bring new information and knowledge into businesses for competitive advantage, higher productivity, and greater profit. Intellovations has offices in Marysville, Ohio, and can be found on the web at **www.intellovations.com**.

Introduction

ForecastWatch.com, a service of Intellovations, is in the business of rating the accuracy of weather reports from companies such as Accuweather, MyForecast.com, and The Weather Channel. Over 36,000 weather forecasts are collected every day for over 800 U.S. cities, and later compared with actual climatological data. These comparisons are used by meteorologists to improve their weather forecasts, and to compare their forecasts with others. They are also used by consumers to better understand the probable accuracy of a forecast.

The Architecture

ForecastWatch.com is built from four major architectural components: An input process for acquiring forecasts, an input process for acquiring measured climatological data, the data aggregation engine, and the web application framework.

There are two main input processes in the system: The forecast parser, and the actuals parser. The forecast parser is responsible for requesting forecasts from the Web for each of the forecast providers ForecastWatch.com tracks. It parses the forecast from the page and inserts the forecast data into a database until it can be compared to the actual data. The actuals parser takes actual data from the National Climatic Data Center of the National Weather Service, which provides high, low, precipitation, and significant weather events for over 800 United States cities, and inserts the data into the database. This process also scores the forecasts with the actual weather data, and places that information in the database.

Once the data has been collected and scored, it is processed by the aggregation engine, which combines the scores into yearly and monthly blocks, sliced by provider, location, and the number of days into the future for which the forecasts were predicting. In its first year, 2003, the system only gathered forecasts for 20 U.S. cities, or about 250,000 individual forecasts, so most of the data output was based on the raw scoring data. The aggregation engine was added once the system was scaled up to 800 cities, increasing the data stream by almost 4000%. In the first half of 2004, the system has already scored over 4 million forecasts, all collected, parsed, and displayed on the Web.

The last component in ForecastWatch.com's architecture is the web site itself. This is the interface through which customers access the collected and aggregated forecast accuracy information.

Implemented with Python

ForecastWatch.com is a 100% pure Python solution. Python is used in all its components, from the back-end to the front-end, including also the more performance-critical portions of the system.

Python was chosen initially because it comes with many standard libraries useful in collecting, parsing, and storing data from the Web. Among those particularly useful in this application were the regular expression library, the thread library, the object serialization library, and gzip data compression library. Other libraries, such as an HTTP client capable of accepting cookies (ClientCookie), and an HTML table parser (ClientTable), were available as third party modules. These proved invaluable and were easy to use.

The threading library turned out to be very important in scaling ForecastWatch.com's coverage to over 800 cities. Grabbing web pages is a very I/O bound process, and requesting a single page at a time for roughly 5000 web pages a day would have been prohibitively time-consuming. Using Python's threading library, the web page retrieval loop simply calls `thread.start_new()` for each request, passing in the necessary class instance method that retrieves and processes the web page, along with the parameters necessary to describe the city for the desired forecast. The request classes use a Python built-in Event class instance to communicate with the main controlling thread when processing is complete. Python made this application of threading incredibly easy.

Python is also used in the aggregation engine, which runs as a separate process to combine forecast accuracy scores into monthly and yearly slices. The aggregation process uses queries via MySQLdb to the MySQL database where the input modules have placed the forecast and climatological data they have harvested. Colorized maps, showing forecast accuracy by geographical area, are then generated for use on the web site and in printed reports.

This forecast accuracy map uses intensity of blue and red to indicate the degree of error in predicting temperatures by geographical area.

ForecastWatch.com's web interface was originally written in PHP but later changed to Python to simplify the toolset and improve integration with the other components of the system. Quixote, a Python web application framework, was selected as the basis for the entirely Python-based web front-end. The Quixote-based web application runs on Linux using Apache with mod_scgi, and was able to serve pages as fast as the PHP-based implementation. Python made it easier to make changes and add features than the PHP implementation. Quixote also provided for more flexible crafting of URLs, replacing PHP URLs like this one: www.forecastwatch.com/drilldown.php?s=2&m=2&d=1&p=1&st=33

with easier to manage URLs like this:

www.forecastwatch.com/drilldown/awx/2004/02/1/AL

This is much nicer for the customer to bookmark or share with clients or potential clients. Finally, Quixote was easy to use and integrates well with Apache. For example, a redirect in Quixote is simply:

```
request.redirect(path-or-URL)
```

Python Made It Possible

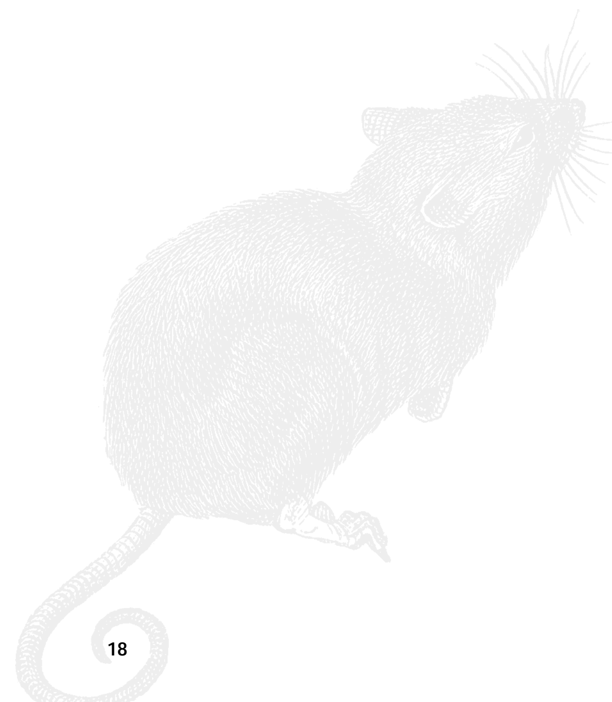
Python played a significant role in the success of ForecastWatch.com. The product currently contains over 5,000 lines of Python, most of which are concerned with implementing the high-level functionality of the application, while most of the details are taken care of by Python's powerful standard libraries and the third party modules described above. Many more lines of code would have been needed working in, for example, Java™ or PHP. The integration capabilities of those languages are not as strong, and their threading support is harder to use.

Python is impressive as an object-oriented rapid application development language. One of Python's key strengths lies in its ability to produce results quickly without sacrificing maintainability of the resulting code. In ForecastWatch.com, Python was used for prototyping as well, and those prototypes were able to evolve cleanly into the production code without requiring a complete rewrite or switching toolsets. This saved substantial effort and made the development process more flexible and effective.

Because of the clean design of the language, refactoring the Python code was also much easier than in other languages; moving code around simply requires less effort.

Python's interpreted nature was also a benefit: Code ideas can easily be tested in the Python interactive shell, and lack of a compilation phase makes for a shorter edit/test cycle.

All of these factors combine to make Python a terrific alternative to C++ and Java as a general purpose programming language. ForecastWatch.com was made possible because of the ease of programming complex tasks in Python, and the rapid development that Python allows.



Cog: A Code Generation Tool Written in Python

Ned Batchelder

About the Author

Ned Batchelder is a professional software developer who struggles along with C++, using Python to ease the friction every chance he gets. A previous project of his, Nat's World, was the subject of an earlier Python Success Story.

www.kubisoftware.com

Introduction

Cog is a simple code generation tool written in Python. We use it or its results every day in the production of Kubi.

Kubi is a collaboration system embodied in a handful of different products. We have a schema that describes the representation of customers' collaboration data: discussion topics, documents, calendar events, and so on. This data has to be handled in many ways: stored in a number of different data stores, shipped over the wire in an XML representation, manipulated in memory using traditional C++ objects, presented for debugging, and reasoned about to assess data validity, to name a few.

We needed a way to describe this schema once and then reliably produce executable code from it.

The Hard Way with C++

Our first implementation of this schema involved a fractured collection of representations. The XML protocol module had tables describing the serialization and deserialization of XML streams. The storage modules had other tables describing the mapping from disk to memory structures. The validation module had its own tables containing rules about which properties had to be present on which items. The in-memory objects had getters and setters for each property.

It worked, after a fashion, but was becoming unmanageable. Adding a new property to the schema required editing ten tables in different formats in as many source files, as well as adding getters and setters for the new property. There was no single authority in the code for the schema as a whole. Different aspects of the schema were represented in different ways in different files.

We tried to simplify the mess using C++ macros. This worked to a degree, but was still difficult to manage. The schema representation was hampered by the

simplistic nature of C++ macros, and the possibilities for expansion were extremely limited.

The schema tables that could not be created with these primitive macros were still composed and edited by hand. Changing a property in the schema still meant touching a dozen files. This was tedious and error prone. Missing one place might introduce a bug that would go unnoticed for days.

Searching for a Better Way

It was becoming clear that we needed a better way to manage the property schema. Not only were the existing modifications difficult, but new areas of development were going to require new uses of the schema, and new kinds of modification that would be even more onerous.

We'd been using C++ macros to try to turn a declarative description of the schema into executable code. The better way to do it is with code generation: a program that writes programs. We could use a tool to read the schema and generate the C++ code, then compile that generated code into the product.

We needed a way to read the schema description file and output pieces of code that could be integrated into our C++ sources to be compiled with the rest of the product.

Rather than write a program specific to our problem, I chose instead to write a general-purpose, although simple, code generator tool. It would solve the problem of managing small chunks of generator code sprinkled throughout a large collection of files. We could then use this general purpose tool to solve our specific generation problem.

The tool I wrote is called Cog. Its requirements were:

- We needed to be able to perform inter esting computation on the schema to create the code we needed. Cog would have to provide a powerful language to write the code generators in. An existing language would make it easier for devel opers to begin using Cog.
- I wanted developers to be able to change the schema, and then run the tool without having to understand the complexities of the code generation. Cog would have to make it simple to combine the generated chunks of code with the rest of the C++ source, and it should be simple to run Cog to generate the final code.
- The tool shouldn't care about the language of the host file. We originally wanted to generate C++ files, but we were branching out into other languages. The generation process should be a pure text process, without regard to the eventual interpretation of that text.
- Because the schema would change infrequently, the generation of code should be an edit-time activity, rather than a build-time activity. This avoided having to run the code generator as part of the build, and meant that the generated code would be available to our IDE and debugger.

Code Generation with Python

The language I chose for the code generators was, of course, Python. Its simplicity and power are perfect for the job of reading data files and producing code. To simplify the integration with the C++ code, the Python generators are inserted directly into the C++ file as comments.

Cog reads a text file (C++ in our case), looking for specially marked sections of text, that it will use as generators. It executes those sections as Python code, capturing the output. The output is then spliced into the file following the generator code.

Because the generator code and its output are both kept in the file, there is no distinction between the input file and output file. Cog reads and writes the same file, and can be run over and over again without losing information.

In addition to executing Python generators, Cog itself is written in Python. Python's dynamic nature made it simple to execute the Python code Cog found, and its flexibility made it possible to execute it in a properly-constructed environment to get the desired semantics. Much of Cog's code is concerned with getting indentation correct: I wanted the author to be able to organize his generator code to look good in the host file, and produce generated code that looked good as well, without worrying about fiddly whitespace issues.

Python's OS-level integration let me execute shell commands where needed. We use Perforce for source control, which keeps files read-only until they need to be edited. When running Cog, it may need to change files that the developer has not edited yet. It can execute a shell command to check out files that are read-only.

Lastly, we used XML for our new property schema description, and Python's wide variety of XML processing libraries made parsing the XML a snap.

An Example

Here's a concrete but slightly contrived example. The properties are described in an XML file:

```
<!-- Properties.xml -->
<props>
        <property name='Id'
type='String' />
        <property name='RevNum'
type='Integer' />
        <property
name='Subject' type='String' />
        <property
name='ModDate' type='Date' />
</props>
```

We can write a C++ file with inlined Python code:

```
// SchemaPropEnum.h
enum SchemaPropEnum {
        /* [[[cog
        import cog, handyxml
        for p in
handyxml.xpath('Properties.xml',
'//property'):

cog.outl("Property%s," % p.name)
        ]]] */
        // [[[end]]]
};
```

After running this file through Cog, it looks like this:

```
// SchemaPropEnum.h
enum SchemaPropEnum {
        /* [[[cog
        import cog, handyxml
        for p in
handyxml.xpath('Properties.xml',
'//property'):

cog.outl("Property%s," % p.name)
        ]]] */
        PropertyId,
        PropertyRevNum,
        PropertySubject,
        PropertyModDate,
        // [[[end]]]
};
```

The lines with triple-brackets are marker lines that delimit the sections Cog cares about. The text between the `[[[cog` and `]]]` lines is generator Python code. The text between `]]]` and `[[[end]]]` is the output from the last run of Cog (if any). For each chunk of generator code it finds, Cog will:

1. discard the output from the last run,
2. execute the generator code,

3. capture the output, from the `cog.outl` calls, and
4. insert the output back into the output section.

How It Worked Out

In a word, great. We now have a powerful tool that lets us maintain a single XML file that describes our data schema. Developers changing the schema have a simple tool to run that generates code from the schema, producing output code in four different languages across 50 files.

Where we once used a repetitive and aggravating process that was inadequate to our needs, we now have an automated process that lets developers express themselves and have Cog do the hard work.

Python's flexibility and power were put to work in two ways: to develop Cog itself, and sprinkled throughout our C++ source code to give our developers a powerful tool to turn static data into running code.

Although our product is built in C++, we've used Python to increase our productivity and expressive power, ease maintenance work, and automate error-prone tasks. Our shipping software is built every day with Python hard at work behind the scenes.

More information, and Cog itself, is available at www.nedbatchelder.com/code/cog.

ERP5: Mission-critical ERP/CRM with Python and Zope

Dr. Jean-Paul Smets

About the Author

Dr. Jean-Paul Smets is CEO of Nexedi. Nexedi develops ERP5, a high end ERP/CRM/ eCommerce solution based on Zope application server and published under open source/ free software licenses. Nexedi provides complete enterprise consulting, software development, and professional training services for ERP5. Nexedi clients include the apparel industry, consumer electronics industry, telecommunication companies, and government agencies.

www.nexedi.com

Introduction

Nexedi is a leader in high-end enterprise services, providing solutions for Enterprise Resource Planning (ERP), Customer Relationship Management (CRM), and eCommerce. Nexedi has built its business on open source, and has designed and released an ERP/CRM framework called ERP5 under the GPL Free Software license.

ERP5 is in production in the apparel industry and government agencies with multigigabyte databases that track millions of warehouse stock movements. ERP5 is written entirely in Python and leverages the Zope Enterprise Objects framework to provide high performance and availability on clusters of inexpensive PCs.

Background

The ERP5 project started in 2002 when Coramy, an apparel industry leader in Europe, decided to migrate its in-house ERP solution to a new open source model. The choice of open source for Coramy ERP was a strategic move to reduce software maintenance costs and to allow Coramy to retain complete control over its custom developments, something that would have been impossible with standard proprietary ERP solutions.

Nexedi was created as an independent company in charge of developing, implementing, and disseminating the ERP5 technology. Nexedi was given a budget of 80,000 EUR to develop a generic ERP framework published under GPL license and customize it for Coramy's specific needs.

Python: Clean Objects from Scripting to Metaprogramming

Working with this small budget, ERP5's developers needed to look to innovative approaches and code reuse to cut development costs. At the most abstract level, ERP5 is based on five generic classes used by all modules: Resource, Node, Movement, Item, and Path. This model, known as the ERP5 Universal Business

Model, makes it possible to reuse code by abstracting away from the specific domain and encapsulating the generic relationships and actions common to many business processes. As a result, modules as different as Payroll and Invoice can share almost all of their code.

The ERP5 abstract model architecture requires a well-designed object language that supports high level abstraction. Selecting an appropriate implementation language became strategic to the project's success. The project also required complete multiplatform OS abstraction, a rich library for web applications, a rapid GUI development toolkit, support for internationalization, wide community support, and proven maturity. The short list of candidates quickly became: Java™ or Python. Or, more precisely, Java+Jakarta or Python+Zope.

Two key factors led to the choice of Python. First, was the need to make extensive use of metaprogramming. Second, for simplicity's sake, ERP5 needed to be implemented in a single language from core architecture to scripting.

Metaprogramming is a technique that allows the programmer to redefine the semantics of the implementation language at runtime. It can be used to endow extremely abstract implementations with domain-specific behaviors that are modeled in properties or tables, rather than hand-coded. In ERP5, this powerful technique allows 95% of the class methods to be generated automatically from lists of properties that define each unique custom ERP implementation. This has reduced maintenance costs by an order of magnitude: The typical ERP5 system contains 100,000 lines of code, rather than the 1,000,000 lines of code required in similar projects based on traditional programming techniques.

Python supports efficient metaprogramming through powerful introspection features that allow programs to inspect and modify code at runtime. Java's introspection are by contrast quite poor and inflexible. Metaprogramming in Java requires writing preprocessors, which would have added prohibitively to the cost and complexity of implementing the ERP5 system.

Another advantage that Python had over Java was that it could be used at all levels of the system, from core implementation to scripting. Most ERP systems, while written in one language, use another scripting language to allow flexible configuration at runtime by ERP administrators. Python is equally well suited both for scripting and core development, reducing complexity and increasing the flexibility of the system. Using Python allowed code initially written as scripts to be incorporated afterward into core components, and vice versa, wherever this made sense. With Java, it would have been necessary to provide a separate scripting environment based on a different language, such as Jython or ECMAScript, and reusing scripting code in core components would have been much more difficult.

Enterprise Objects with Zope

In addition to leveraging these high-level language capabilities, the ERP5 project also needed to base on an existing turn-key application server with support for transparent object persistence, transactions, and workflow. This would allow the project to focus its limited resources on application design, rather than on application server design.

Zope, a Python-based application server, fulfilled this need. In 2002, Zope was already a mature application server environment while Jakarta was still rather immature. Zope provided clustering, object storage, object publication, transactions, security, workflows, and a web-based management interface, all in a turn-key package. Zope and Python, unlike Java, also provided the licensing needed for complete freedom in distributing code on LiveCD, in RPM packages, and so forth.

In some applications needed in ERP, such as in Point-of-Sales (POS), an autonomous client/server GUI application provides better results than a pure web-based solution. For these cases, the ERP5 project team selected PyQt, which supports the rapid development of multiplatform client/server applications with native look and feel. Autonomous applications written in PyQt could interoperate with Zope through the XML-RPC and SOAP implementations available for Python.

ZSQLCatalog: Querying and ROLAP of Zope Object Database

In January 2003, the initial ERP5 modules went into production at the Coramy factory and design center.

The ERP5 architecture relies completely on Zope for data storage, transactions, and workflow management. The user interface is based on an extended version of Martijn Faassen's Formulator component. ERP5 itself is implemented as a set of Zope components. Overall, thanks to massive code reuse and fast coding, ERP5's initial development was completed in less than one year. The choice of Python and Zope proved to be a good one.

Part of this success was due to Nexedi's ZSQLCatalog component which leverages Zope's object database to implement an innovative approach for querying objects and data mining.

ZSQLCatalog solves several major problems often encountered in Enterprise applications: locks, slow reports, and inconsistent transactions. Most Enterprise applications use tables in a relational database to store data. In ERP5 data is stored instead in Zope's object database. Zope eliminates the need for storage adapters or attribute mappings by providing transparent persistence of Python objects. Zope also speeds up object access: Reading the Zope object database is 10 to 100 times faster than retrieving a row from the fastest relational databases available on the market today. Finally, unlike relational databases, transactions in a Zope object database can last minutes without causing deadlocks.

One limitation encountered in Zope was that it does not provide an efficient tool for querying the millions of objects that are needed in an ERP solution. To solve this, Nexedi's ZSQLCatalog maps between attributes or methods of objects and relational columns, tables, and databases. This mapping is not intended to store actual object attributes into relational tables, but rather only those attributes relevant to facilitating fast queries. In this way, the relational database is used as an index into the Zope object database.

ZSQLCatalog can be viewed as a kind of Relational Online Analytical Processing (ROLAP) component for object databases, and it provides a nice interface for extracting reports from the object database into OpenOffice or Microsoft Excel spreadsheets.

ZSQLCatalog was very successful in this project: a Zope database with more than 2,000,000 objects can be queried with statistical methods in a few milliseconds.

ERP5SyncML: Synchronized Distributed Objects

Coramy's requirements for ERP5 included the need to deploy applications in its factories in Tunisia. This posed some additional challenges, since Internet connectivity between Africa and Europe is not always perfect and can be very expensive. Given Coramy's modest budget for network connectivity, the project had to provide a solution for synchronizing two ERP5 servers over slow and unreliable transcontinental Internet connectivity.

To do this, Nexedi implemented the SyncML protocol in Zope, using email or HTTP as the transport layer. Early prototypes were developed in a few weeks, using Python's rich library for network protocols

and XML parsing.

Turning those prototypes into an industrial-strength solution turned out to be a bit more complex, due to the many possible failure cases the code had to handle. Nexedi used Python's unit testing framework to write extensive tests of the SyncML implementation. These tests could test every part of the SyncML component, especially those that would be difficult to test in the field, and report any problems back to the developer. Once this was done, the ERP5SyncML component became reliable. In the process of testing, some minor bugs occurring in complex situations were found in the Python libraries. Because Python is open source, the ERP5 developers could find and fix the bugs quickly and contribute the fixes back to the Python community.

ERP5SyncML is now being used not only by ERP5 but also by Nuxeo CPS, an advanced Content Management System, as a way to synchronize documents between French government agencies.

Python-GLPK: Linear Programming in Python

Some aspects of an ERP system's database query functionality require mathematics that go beyond the capabilities of the relational query model. For example, ERP5 uses linear programming to determine resource capacities. Although Python includes excellent numerical frameworks, C or FORTRAN implementation of complex scientific algorithms is usually much more efficient.

Nexedi found the GNU Linear Programming Kit (GLPK) to be a good starting point for ERP5's linear programming needs. GPLK is written in C, and interfacing it to Python was achieved in only a couple of hours using the SWIG glue libraries. Nexedi now distributes a Python GLPK module, python-glpk, which provides the power of linear programming in Python.

Conclusion

The ERP5 abstract model has been found to reduce development costs by an order of magnitude when compared to traditional ERP architectures, and it has performed well in large mission-critical Enterprise applications. ERP5's largest system runs on a cluster of eight CPUs. It serves 50 simultaneous users, each of them with 8 parallel sessions, and it handles more than 2,000,000 Python objects. Its ZSQLCatalog relational index holds more than 10,000,000 rows.

Python and Zope were key to this success. Python provided a powerful object language and a rich set of libraries which allowed quick development of clean and compact code. Zope provided a mature application server and object database. Nexedi is sometimes asked: Why not Java™ and J2EE™? While it would be possible to create a similar system with Java and J2EE, development costs would be much higher. Jakarta and ObjectWeb have both matured but are still poorly integrated when compared with Zope, and require a much more complex development style. Java's poor introspection features are also still a serious limitation for efficient metaprogramming, for using Java itself as a scripting language, or for flexible object persistence. Using Java is simply not consistent with today's trend of cost cutting in Enterprise development. If the choice were made again today, Nexedi would still opt for Python and Zope.

In June 2004, ERP5 was nominated for "best enterprise project" by *Decision Informatique* professional magazine. Nexedi is now working to simplify the ERP5 setup and configuration process, in order to ease its adoption by a wider audience of open source developers.

EZTrip.com Chooses Python for Enterprise Integration

Michael Engelhart

About the Author

Michael Engelhart is the CTO and lead software engineer for EZTrip.com. He previously worked at Apple computer as senior software engineer for worldwide corporate travel, and has been a travel technology consultant to several major travel industry suppliers over the past 10 years.

www.eztrip.com

Introduction

EZTrip.com is an online travel reservation company based in the United States. Our core business is developing online booking engines and related systems that communicate with large global distribution systems (GDS), as well as directly to suppliers' central reservation systems (CRS).

Most GDSs were built in the late 1960s and early 1970s and are mainframe based OLAP systems that handle millions of transactions per day. GDSs work incredibly well and are testament to the skill of the early developers that built them, but they have severe limitations when it comes to the data that can be entered into them, and the formats that the data is allowed to take. Although some CRSs used by individual suppliers are more modern, they inherit many of the limitations of the GDSs as a result of their tight data-level integration with them.

This is a problem when using these systems in the context of a web-based applications like EZTrip.com. Also, GDSs are not relational systems and lack a query language like SQL®, so queries are limited by the API provided by its developers.

During the past two years our online processing systems have been developed in Java™ because many of the GDSs provide a low level Java or C API, and most of our developers have experience building J2EE™ applications. While Java is a great language for building a large web presence with persistent data, many aspects of our development would quickly become unmanageable and prohibitively expensive using Java alone. Python and Jython are used instead for many of the day-to-day integration tasks and the large amount of data "cleaning" that are required to provide customers with a user-friendly interface.

Green Screens

The travel industry is much like the financial industry in that "screen scraping" is often required to integrate older systems that provide no interface other than the textual screen interface used by its original human users.

Screen scraping makes heavy use of text processing tools, regular expressions, and string manipulation, all of which are built into Python and are very easy to work with. Python also provides the ability to communicate easily with other operating system processes, which made it possible to leverage external tools like *sed* and *awk* for the processing tasks. This flexibility in choosing the right tool for the job was critical to the rapid development of the custom screen scrapers needed to interface to the very many unique travel supplier's systems.

EZTrip.com uses Python's OO aspects heavily and has built an extensive class library that allows new developers to come up to speed quickly when working on new GDS data access tasks.

Text, Text, Text

Text processing is where Python really shines for us. Almost all the data handled at EZTrip.com is text-based. From screen scraping GDS data to data mining vendors' web sites, Python tools munge text day in and day out, and have yet to run into any problems with the language or its performance. For these tasks, Python is fast.

One of the tools currently being worked on is a localization system that allows generation of localized versions of hotel property descriptions without requiring human translators for every one of the 100,000 hotels in the EZTrip.com network. This tool will increase market size tenfold, and is a project that is coming along much faster than anticipated. Python's text processing capabilities helped make it possible to build a solution with much higher productivity per man hour than would have been possible using Java as the development language.

Web Services to the Rescue

In the past year or so, the larger suppliers have been slowly rolling out web services for some of their products and services. These also work seamlessly with our Python code. The XML processing tools in Python are, like everything else included with Python, well thought out, spec compliant, and powerful.

EZTrip.com often builds automated test suites in Python, in order to validate a new supplier web service before rolling it into its web presence. Python's rapid development times and capacity for automating this kind of testing makes it possible to quickly find and work around bugs in the supplier's code, resulting in a more robust web application.

Why Python?

The author originally came across Python in 1999 while working on a large Java-based web site. Like many developers new to Python, the use of indentation to indicate program structure was a stumbling block during this initial contact and Python was not adopted for use with that project. On second look, after actual experience with the language, the language's indentation-defined structure became a virtue, an important part of the overall power and simplicity of the language.

In addition to Python's clean design, the following factors make Python a good choice for enterprise integration tasks, like those undertaken at EZTrip.com:

- OS Independence—The ability to develop code on one operating system, email it to a travel supplier's IT group, and have it work seamlessly on another operating system has been a godsend in deploying and maintaining the many components of EZTrip.com.
- Database Integration—Python's database tools are top notch, allowing for quick and painless development of data mining tools in a matter of hours, rather than the days or weeks it would take in a language like Java.
- Batteries Included—Except for a few database libraries and domain-specific libraries developed in-house, almost everything needed by EZTrip.com developers is included in the Python distribution. Nevertheless, Python has managed to avoid the bloat seen in many other languages.
- Community—The Python community is fantastic. The amount of online technical information available for Python is vast. A good Internet search engine will almost always find the answers a developer needs, any time of the day or night. In those cases where EZTrip.com developers have had questions not answered by a web search, for example about the LDAP library or a database library, the developers of those projects have always been willing to provide an answer quickly. This has been invaluable as a time saver, and in keeping development costs down.
- Jython—Jython is an implementation of Python that is written in Java and runs on the Java Virtual Machine. It provides a powerful tool for scripting Java, and a more productive way to develop components for use in a Java system. For EZTrip.com, Jython has bridged the gap between the front-end Java-based web systems and the back-end Python tools that do much of the heavy lifting.

Newbies

One unexpected bonus discovered in the short two years EZTrip.com has been using Python is its support for new developers and interns, and its ability to make existing code more approachable and maintainable. Python has exhibited an uncanny ability to teach and encourage good coding skills, enabling developers to write clear and concise code. Python is very well designed, and this quality tends to transfer into the software that is written with it.

Summary

Python has helped EZTrip.com in countless ways. It has reduced costs by speeding development time, improved integration with myriad suppliers, provided a solid backbone to the behind-the-scenes development that continues to strengthen EZTrip.com, and made it possible to meet the many goals faced as the business has grown. Without Python, EZTrip.com would not be as successful in the online travel space as it has been in such a short period of time.

Frequentis TAPtools®—Python in Air Traffic Control

Michael Bartl

About the Author

Michael Bartl initially joined Frequentis in 2000 as a software engineer to test telecommunication hardware and later moved on to develop weather information systems. After several years he is now Product Manager for the TAPtools® product family. His main addictions are Java™, Python, and chess.

www.frequentis.com

Introduction

Frequentis is one of the world's leading providers for safety-critical solutions in the field of Air Traffic Management and Public Safety & Transport. With over 500 employees world-wide, it provides innovative, user-centered solutions to its customers.

Frequentis has been using Python in its TAPtools® product family, which focuses on the tower and airport tools segment of Air Traffic Control. These tools are used by air traffic controllers to track weather conditions, control runway lighting, and to monitor and control navigational aid instruments.

A Brief History

One of the problems in developing air traffic control solutions is that each customer's unique airport, regulatory picture, and methodologies impose specific and different requirements for user interface look and behavior. A significant part of the deployment of an air traffic control system is the customization of its interface.

Instead of developing each user interface from the ground up on a per-customer basis, Frequentis has developed a user interface layout tool called PanView, similar to products like QDesigner or Visual Studio. This tool is used to design and build the user interface which is then executed by a piece of software called the PanMachine. The PanMachine runs on in-house developed hardware called the PowerPanel, a 66MHz PowerPC with 32 MB RAM and a 12" touch screen entry device.

With these tools, Frequentis developers can rapidly prototype a layout in front of the customer, greatly reducing the number of customer design workshops necessary in the deployment of a solution.

The Customer is King

PanView and the PanMachine were originally developed using Lua as the scripting language used to connect the user interface to the underlying functionality of the air traffic control system.

This choice was found to be problematic for the layout implementors for a variety of reasons:

- Limited information is provided on errors, making ithard for developers to locate bugs.
- Variables are global, not local, by default. Python is exactly the other way around, making programs less prone to error.
- Lua has no list data structure. Although its dictionaries can be used as lists, this caused unnecessary complexity in practice.
- Lua code is easy to follow for short scripts, but its syntax and minimal standard library makes it unmanageable for larger programs.

In a very important project, the Finnish Civil Aviation Administration (FCAA) wanted to run their user interface layouts not just on the PowerPanel, but also in the context of a web browser. This requirement was important enough that it led to re-implementation of the PanMachine in Java, so it could be run as an applet in the browser. Because Lua could not be run under Java, this was a good time to replace it.

Python and Jython, the Java implementation of Python, were chosen because they would allow both the PowerPC and Java implementations of the PanMachine to execute the same user interface layouts. Python, implemented in C, was used on the PowerPC, and Jython, implemented in Java, was used in the browser applet.

Implementation in Python

Re-implementation using Python went smoothly. The Python language interpreter and support libraries are written in C and can be compiled with most C compilers. In this project, the ability for Python to be embedded into other code worked very well. The documentation is excellent and the examples are easy to follow.

However, the PowerPanel hardware has no hard disk and thus could not itself compile the Python interpreter's C sources. To work around this problem, the developers cross-compiled Python from another machine, producing object code for the PowerPC using a compiler running on another type of hardware. Once this was done, Python byte code produced on any machine could be run on the PowerPanel without modification; only the initial compilation of the language itself required cross-compilation.

Rewriting the Lua Layouts

Once the Python-based implementation of the GUI layout tools was complete, it became necessary to rewrite the existing Lua layouts using Python. Our layout coders embraced Python with open arms because it solved all the problems they had been having with Lua, and it made new code easier to write because of Python's straightforward syntax and extensive standard libraries.

In this transition, we found that Python's syntax was very easy to learn for new users, and our coders were able to completely rewrite the Lua layouts to take advantage of the new features offered by Python, rather than just porting them at the language level.

The finished layout code contains about 5000 lines, which is half the size of the original code, is much easier to maintain, and works flawlessly with Jython in the Java port of the PanMachine.

Conclusion

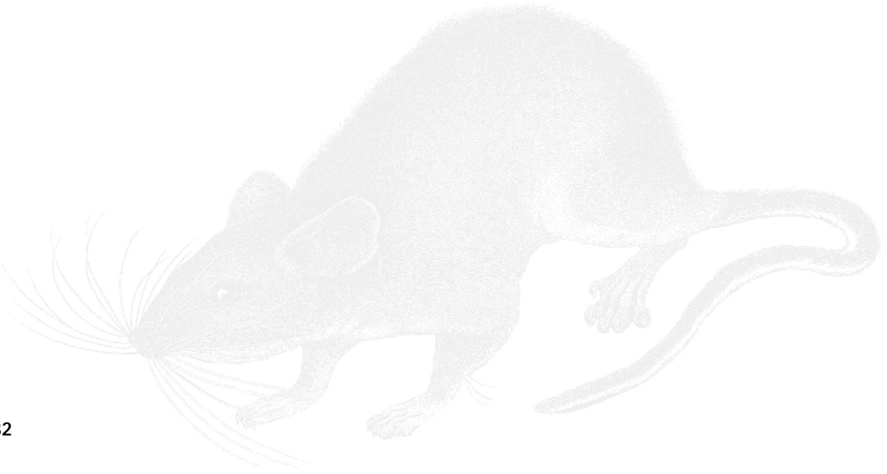
Python enabled us to fulfill customer requirements in a time-critical project, while improving the overall quality of our tools. Since changing to Python, layout implementation has become much easier for a variety of reasons:

- Python's runtime error handling makes it easier to locate and fix problems in code. The stack traces produced by Python, even when running in staged production on the custom PowerPanel hardware, have helped to speed up the

testing and debugging process.

- Python's vast standard libraries allow rapid development of functionality without resorting to re-invention of the wheel.
- Python's very clean syntax and indentation-based program structure makes code much easier to read and maintain.

Since using Python in our development, the time to write new user interface layouts for the TAPtools® product family has been reduced by a factor of three.



LoveIntros Uses Python to Help Northwest Singles Click

Greg Unrein

About the Author

Greg Unrein cofounded LoveIntros with his wife, Junel. Greg holds a degree in computer and information science from the University of Oregon. He has been in the technology industry for almost a decade, with years of experience in Java and C++, and he has found Python to be the best tool for creating a superior dating site for Northwest singles.

www.loveintros.com

Introduction

LoveIntros is an online singles community for people living in the Northwest United States. The LoveIntros web application allows members to create personal profiles, send and receive anonymous emails, create private web journals, perform advanced profile searches, and execute many other functions. By providing a safe and effective way for individuals to find each other, LoveIntros makes it easier to meet new people who have similar interests and values and who live in or near one's own community.

The Architecture

As a software project, LoveIntros contains several different modules: The end-user's web application, the administrative interface, the anonymous email daemon, and a cross-platform GUI component for editing some of the site's content. Together, these modules provide all of the functionality needed by users and administrators. The system is deployed on FreeBSD, but is portable to other operating systems. In fact, most of the project's initial development was done using Windows® 2000.

The LoveIntros web application allows users to create profiles, search for others who interest them, and communicate with these individuals in order to explore the potential for a relationship. This application is based on a three-tier architecture. Data is stored in a PostgreSQL database, and psycopg provides database connectivity. The middle tier is coded in Python and is driven by mod_python running under Apache 2. The user interface is rendered as HTML using the SimpleTAL template library.

The administrative interface to the site is written using the Quixote web application framework, and employs mod_scgi behind Apache 2. This interface provides the site administrators access to all of the functions they need to interact with users' profiles, process log files to provide reports about site activity and issues, and update site content. Quixote and mod_scgi have

performed extremely well in this application, and plans call for them to replace the current SimpleTAL and mod_python implementation of the user site.

Another important module of the LoveIntros project is its anonymous email daemon. This daemon strips the sender's email addressed from correspondence sent to other users, then forwards the emails on to their intended recipient. As a result, members can communicate without the risk of accidentally divulging their actual email addresses. Because of the power of Python and its associated libraries, this daemon consists of only a few hundred lines of code, which include several response email templates that are sent to notify senders in the event that an email could not be forwarded successfully.

Finally, LoveIntros includes a GUI application used to create a domain-specific XML editor that allows nontechnical users to edit LoveIntros' FAQ files. This was written using PythonCard, a rapid GUI development tool based on the wxPython GUI framework. Inspired by Apple's HyperCard®, PythonCard was impressive in its ability to quickly and easily create a responsive cross-platform GUI.

Implemented in Python

The LoveIntros implementors chose Python for its expressive, easy-to-maintain nature, and its rich set of built-in libraries and excellent third-party modules. Java™ and Jython, the Java-based implementation of the Python programming language, were both considered as well, but were not chosen because of the added complexity introduced by either approach, without appreciable added value. Ultimately, Python's flexibility, coding speed, solution quality, and libraries made it the language of choice for this endeavor.The LoveIntros project consumed approximately four months of full-time development work, with one developer working on the Python code, and one designer working on the UI. These four months of work were spread over more than a year of actual part-time effort. Python helped the programmers sustain this development rate in several ways.

First, Python is a very expressive language. The syntax is clean and easy to read, and basic data structures are built in. It is amazing how much less typing goes into Python code, since braces and type declarations are not used.

Second, the standard library and third-party modules available for Python are breathtaking in their coverage. Almost every time a problem was encountered, a Python library that helped speed the solution process was found. These libraries, both standard and third-party, were high-quality and tended to have helpful communities of users.

Finally, the ease with which code can be read and comprehended quickly, even months after being written, has helped to keep LoveIntros' feature set growing and maturing more quickly than any other project with which its developers have been involved.

Python has been a major factor in the success of the LoveIntros software project. It has proven to be a great choice, and LoveIntros' developers have only happy experiences to relate regarding this excellent language.

Python and Zope in the EZRO Content Management System

Andy J. Williams Affleck, M. Adam Kendall, and Development InfoStructure (devIS)

About the Authors

Andy J. Williams Affleck is the project manager for EZRO and M. Adam Kendall is the lead developer on the project. Affleck has an Ed.M. from the Technology in Education program at the Harvard Graduate School of Education and has many years background in creating solutions for online learning and teaching. Kendall is the creator of zopelabs.com, holds a BFA in digital design, and has over 9 years web programming experience.

ezro.devis.com

www.devis.com

Introduction

Development InfoStructure (devIS) is a small consulting firm in Arlington, Virginia, that is well known for its work in the eGovernment sector. This includes development of small, medium, and large-scale systems.

devIS EZ Reusable Objects (EZRO) is a content management system which can be used for many different kinds of web sites, including traditional information presentation sites such as www.devis.com, portals like www.milspouse.org, training sites like cable.devis.com, and coach style sites. A coaching site appears as a frame on the edge of the screen and drives another site in order to walk the user through that site, as in www.careeronestopcoach.org/EZRO was the outgrowth of a number of contracts with the Department of Labor over the years from 2001 through 2004. It was developed under the name of Workforce Connections. devIS contributed its copyright to the Department of Labor to allow them to release it as an open source tool in January 2004. EZRO is a more advanced version of the Workforce Connections software that has been released by devIS under the GPL.

Why Python

EZRO was originally developed as the engine to host the Department of Labor's DisabilityInfo portal, originally called Disability.gov, later renamed to DisabilityDirect.gov and now DisabilityInfo.gov. The site is a portal site that is maintained by people in each of the major agencies including the Departments of Labor, Education, Transportation, Defense, Commerce, as well as the Social Security Administration and Health and Human Services. The Department of Labor, the implementing a gency, had final editorial authority.

What was needed was a content management system that allowed for distributed editing and centralized editorial control that had to be fully accessible to the disabled. To further complicate things, an Executive

Memorandum from President Bush mandated the site go live within sixty days. While some work had already been started and a previous generation of the site existed, there was still an incredibly tight deadline to meet. As the site was to be one of a handful of sites mandated by the White House, it had to be done right the first time.

devIS decided from the start of the project that our code had to meet two general requirements. First, the software should eventually, some time after meeting the initial deadline, be released as open source. Second, it had to be portable enough to run on several platforms.

Python and Zope, a web application server written in Python, were chosen for the work. devIS had been using Zope since 1999 and already had a large staff of developers that were skilled in the ways of Zope, including the owner and developer of ZopeLabs.

At this time devIS was still developing its web applications using Zope's management interface, DTML (dynamic template markup language) tags, and short Python scripts to create dynamic sites. But Zope's management interface was not a good environment for group work: one developer could accidentally overwrite another developer's work version control.

With five developers working half-to-full time on this project, it was clear a better solution was needed. The natural conclusion was to start developing in Python using the Products framework available in Zope. A Zope product is a package of code, graphics, and templates that provides a piece of reusable web functionality. We were able to combine the integrated pieces of Zope that we loved and used on a daily basis, like user management and simple object publishing to the Web, with the flexibility of Python and its large internal library. This also allowed us to keep source code outside of the Zope object database and on the file system, where it could be used with our existing CVS infrastructure for source code control and reporting.

Implementation

Development of the EZRO content management solution started with the assumption that the application would grow over time, as most software projects do. The initial choice was between producing the software from scratch, or using an already existing framework like the Zope CMF or Plone, both of which were first released around the same time as development was started on EZRO.

After carefully reviewing the alternatives, it was decided that developing a new solution would result in a product more closely tailored to the client's needs and requirements, without the significant overhead incurred by the unneeded parts of CMF or Plone.

Because Python allows for very rapid development, this choice was not at odds with the very tight initial deadline on the DisabilityInfo project. In two weeks, a working prototype was ready to show to the client. The first production quality release of the software only took an additional three weeks, building directly upon the prototype.

By using Python, maintenance overhead was greatly reduced. Over the next year, devIS developers were able to add new EZRO features quickly and easily, in response to requests from the client. As devIS acquired new clients, EZRO was leveraged to reduce development cost of new projects. In this way, the tool grew in features and scale to what it is today.

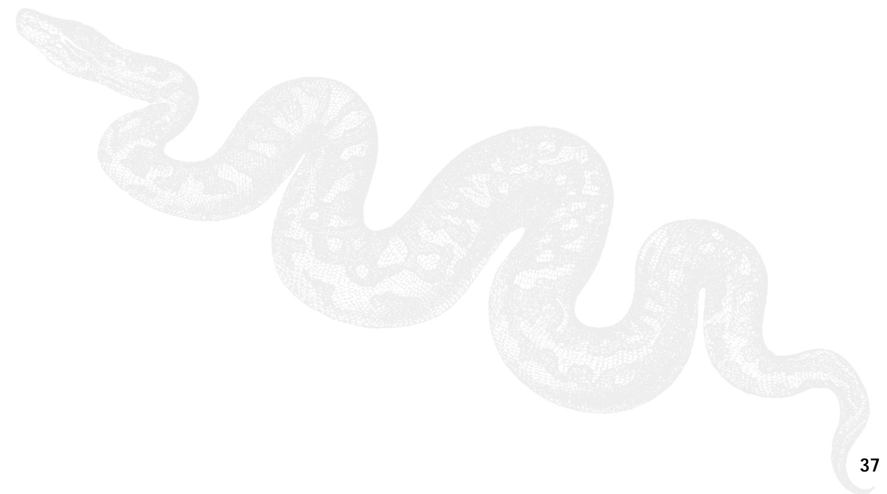
Python's clean layout and structure also made the code that was released later as open source more readable. This was a boon when bringing new developers on board, as they were able to jump in and understand the code right away.

Because of the cross-platform nature of Python, EZRO runs equally well on Microsoft® Windows® and any Unix®-like operating system, including Linux® and Mac OS® X. devIS has released a self-contained EZRO environment with installer for Windows, and distributes just the EZRO Zope Product in tar archive form for other environments. The software does rely on Zope, Aspell, and PyXML, but because of Python's portability, all of these third party applications also run on most other operating systems.

EZRO runs happily on a single machine but is deployed in the devIS production facility on three redundant servers for the public-facing sites, and a fourth secure server for the administrative back-end.

Conclusion

Python and Zope together gave devIS incredible flexibility and allowed implementing new features on a very rapid cycle, which, in turn, has greatly pleased devIS clients. The OSHA Training Institute is using EZRO to develop its training materials and they've informed devIS that they are saving thousands of dollars per day in development costs. As a bonus, the people who use it to develop their courses actually enjoy coming to work and using this tool. In no other set of tools have devIS staff found such ease of implementation.



Suzanne: Python Handles Critical Data in a Domain Name Landrush

Stephane Bortzmeyer

About the Author

Stephane Bortzmeyer (bortzmeyer@nic.fr) *is network and systems architect at AFNIC. He has worked for many Internet companies, from network operators to web agencies, and developed software in Perl, Ada, and C before adopting Python as well.*

www.afnic.fr

Introduction

AFNIC is the registry of the French *.fr* top-level Internet domain. For many years, registration rules in *.fr* were very strict. On May 11, 2004, the rules changed to a more liberal model, allowing many registrations that were not possible before. As a consequence of this change, the registry was faced with receiving a potentially unmanageable burst of requests. This is a problem known as a "landrush" faced by every DNS registry which changes its rules or introduces a new service, like Unicode domain names.

Since the rule of registries is "first come, first served," everybody wants to have their submission arrive immediately after the new domain space is available, in this case May 11, 09:00, local time. In this context, some domain names were requested by three different clients within the very same second.

Machine crashes are a sad tradition at registries resulting in unexpected delays, clients frantically hitting Submit while trying to send a completed web form, and bad press for the registry afterwards.

For a variety of reasons, this particular landrush was a technical success, with zero crashes. This article focuses on the role that Python played in this success.

Buffering the Requests

The AFNIC registration system was written years ago, mostly in Perl. It responds to registration requests, satisfies them, and then returns confirmation to the web client, all in one synchronous request/response cycle. Although this system works well during normal load, it simply cannot handle the load of a landrush. Even more modern systems that are parallelized across many machines will fail under the impossible load of a landrush, where web requests from clients must be answered in a synchronous manner.

To avoid this bottleneck without attempting to rebuild the whole registration system, AFNIC decided to place an asynchronous buffer between the client and the main registration application. This buffer was not a full-fledged system, it just received the requests, stored them in FIFO order, acknowledged them, and later handed them over to the real registration system within the limits of its processing capacity.

This asynchronous buffer was written in Python and is named Suzanne, from the novel *Un barrage contre le Pacifique* by Marguerite Duras, where the hero's mother tries to protect her land from the waves of the Pacific ocean.

The requirements for Suzanne were simple: it had to be fast, robust, reliable, and it had to work the first time around, because there was no second chance. Protecting the data was extremely critical: While the big frenzy of domain name speculation is over, many companies are still ready to spend a lot of money for a domain name. If they miss the domain name they want because buggy software dropped a request, the registry is faced with bad press at the minimum, and possibly expensive litigation.

Why Python?

AFNIC uses many programming languages (Perl, Java™, PHP, PL/SQL™, and Ruby) but this was the first use of Python. Like Perl and Ruby, Python includes the standard libraries that were required by this project. Python was chosen for its readability and ease of use.

Implementation

Most of the brutal load was handled by the Postfix mail system. All requests were received by email, which is asynchronous and thus avoids the above-mentioned limitations of synchronous processing. Suzanne was in charge of rotating mailboxes to keep them at a manageable size, and emitting acknowledgement emails. This process used Python's standard library modules mailbox, email, and *smtplib*, and the Cheetah templating system was used to create the acknowledgement messages.

When dealing with critical data accessed by several programs, all running at full speed under a heavy load, it is critical to prevent hidden race conditions. Since Python currently lacks a standard global locking module, the glock module by Richard Gruet was used to lock the mailboxes.

Once messages had been processed and acknowledged, Suzanne had to send requests to the real registration system. Since SOAP is the standard in-house inter-application protocol, AFNIC created a web service in Python, using ZSI and the standard library's BaseHTTPServer. The SOAP client was the registration system, running Perl.

This project used Python Version 2.3 and various Unix platforms. Suzanne is very small: Just 2290 lines of Python, including the testing programs and the statistics programs.

Testing was the key to success in this application, which had to perform flawlessly during its first production run. Many bugs were uncovered by testing Suzanne under load, first in-house, with several machines sending as many requests as possible, and later by volunteers sending requests from the outside.

Problems Encountered

Some problems were encountered during the implementation. Working with mail boxes and email messages in Python is somewhat complicated by the many modules in the standard library provided for this purpose. For this application, these modules were often too low-level, requiring somewhat more work than was initially expected.

ZSI also caused some trouble, and was sometimes difficult to use. The authors used SOAPpy on other projects, which in hindsight may have been a better choice. Some of these problems stem from the fact that SOAP is complicated and difficult to deal with in general, and were not always the fault of Python or ZSI.

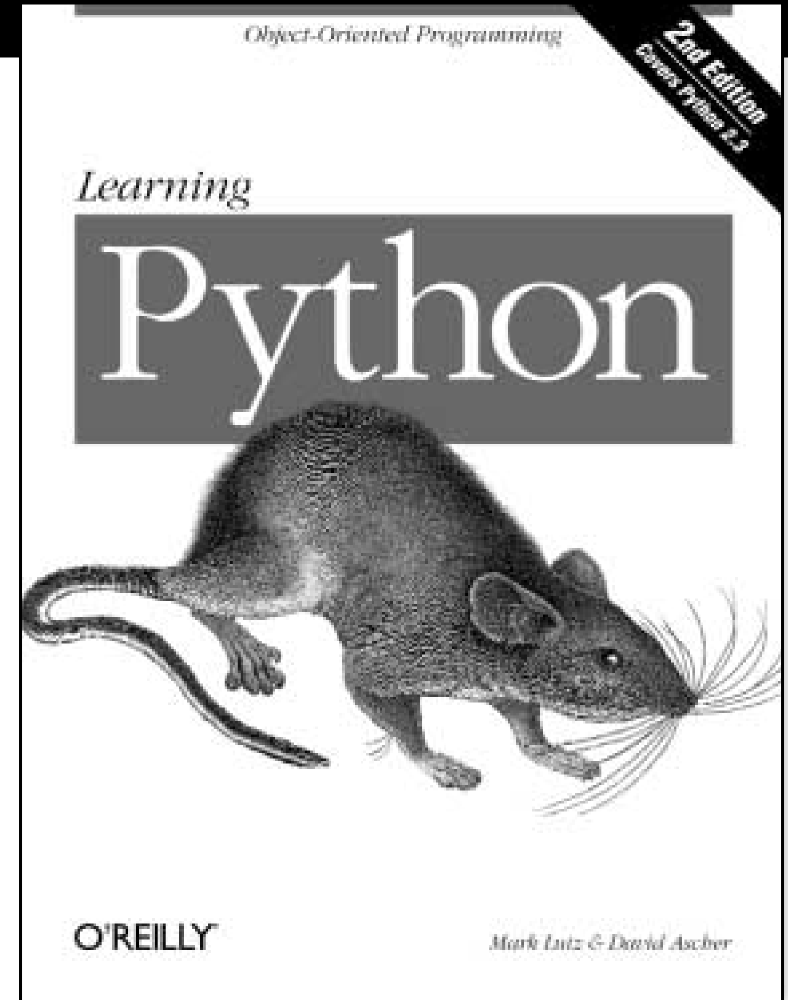
Successes

Nevertheless, development of the system was quite fast, the resulting code is simple and readable, and the system performed well.

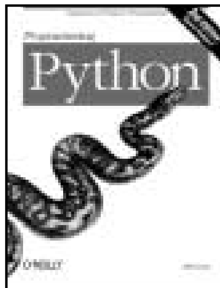
Suzanne was able to handle the required processing with acceptable system load. Mail boxes were parsed and acknowledgements sent at a very satisfactory rate. And best of all, nothing crashed during the landrush, even with 12,000 messages in the first minute, a result of the fact that many clients developed custom programs to start submitting requests at exactly 09:00.

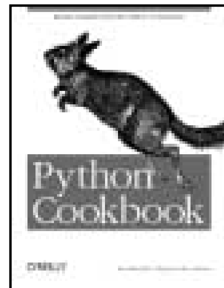
Suzanne performed so well that AFNIC is considering making it a permanent part of its infrastructure. If this is done, every request would go through Suzanne, allowing the registration system to work at its own pace regardless of unexpected peaks in future request activity.

Got Python?



Essential Tools for Success with Python





Order Now

At **python.oreilly.com** or call **800-998-9938**

©2004 O'Reilly Media, Inc. O'Reilly is a registered trademark of O'Reilly & Associates Inc.

Item #40140