Python Success Stories

8 true tales of flexibility, speed, and improved productivity

Industrial Light & Magic

LivingLogic AG

Rackspace

Siena Technology

United Space Alliance

Wing IDE

University of St Andrews

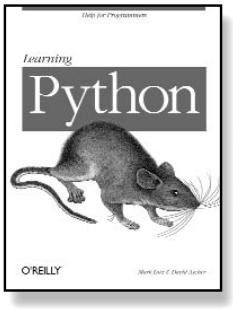
Journyx

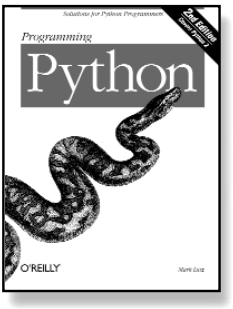
"We achieve immediate functioning code so much faster in Python than in any other language that it's staggering."

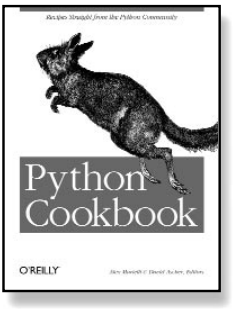
—Robin Friedrich *United Space Alliance*

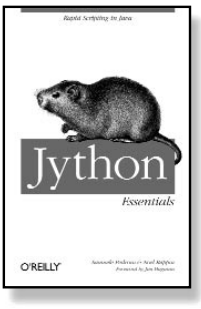
O'REILLY®

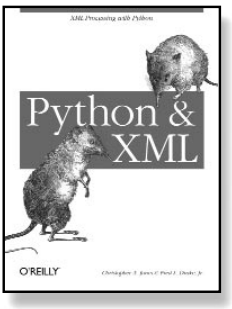
Essential Tools for Success with Python

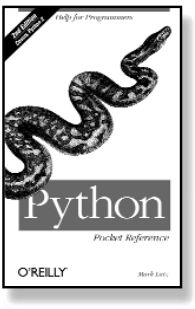


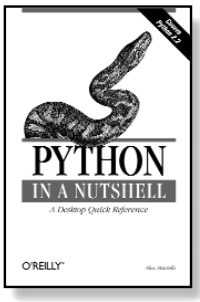


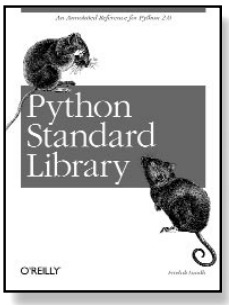


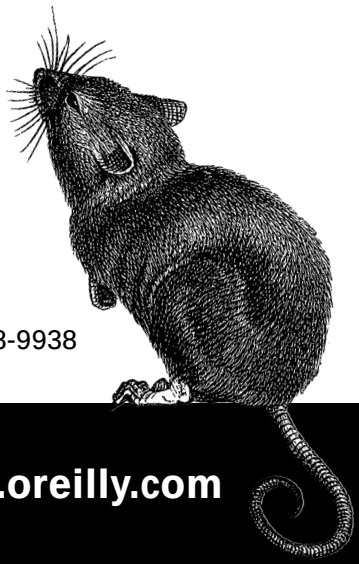












Order Now

See back page for order form. Or call 800-998-9938

O'REILLY®

python.oreilly.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.**

Industrial Light & Magic Runs on Python

Tim Fortenberry

About the Author

Tim Fortenberry joined Industrial Light & Magic in 1999 as an intern. Later that same year he began to work full time in the Resources department. He worked as a scripts/tools programmer. Shortly after, Fortenberry joined the Research and Development department. He is one of the founding members of the Pipeline and TD Tools groups that helped bridge the gap between artists and technology.

As an engineer, Fortenberry is responsible for developing and maintaining the myriad of applications used for rendering and pipeline control flow of images at ILM. Prior to joining ILM, Fortenberry worked as a Linux® systems administrator for VA Linux Systems.

Originally from Southern California, Fortenberry received his Bachelor of Arts degree from the University of California at Berkeley in Anthropology with an emphasis in Archaeology.

www.ilm.com

Background

Industrial Light & Magic (ILM) was started in 1975 by filmmaker George Lucas in order to create the special effects for the original *Star Wars* film. Since then, ILM has grown into a visual effects powerhouse that has contributed not just to the entire Star Wars series, but also to films as diverse as *Forrest Gump*, *Jurassic Park*, *Who Framed Roger Rabbit*, *Raiders of the Lost Ark*, and *Terminator 2*. ILM has won numerous Academy Awards® for Best Visual Effects, not to mention a string of Clio awards for its work on television advertisements.

While much of ILM's early work was done with miniature models and motion controlled cameras, ILM has long been on the bleeding edge of computer-generated visual effects. Its computer graphics division dates back to 1979, and its first CG production was the 1982 Genesis sequence from *Star Trek II: The Wrath of Khan*.

In the early days, ILM was involved with the creation of custom computer graphics hardware and software for scanning, modeling, rendering, and compositing (the process of joining rendered and scanned images together). Some of these systems made significant advances in areas such as morphing and simulating muscles and hair.

Naturally, as time went by many of the early innovations at ILM made it into the commercial realm, but the company's position on the cutting edge of visual effects technology continues to rely on an ever-changing combination of custom in-house technologies and commercial products.

Today, ILM runs a batch processing environment capable of modeling, rendering and compositing tens of thousands of motion picture frames per day. Thousands of machines running Linux, IRIX®, Compaq® Tru64®, Mac OS® X, Solaris®, and Windows® join together to provide a production pipeline with

approximately eight hundred users daily, many of whom write or modify code that controls every step of the production process. In this context, hundreds of commercial and in-house software components are combined to create and process each frame of computer-generated or enhanced film. Making all this work, and keeping it working, requires a certain degree of technical wizardry, as well as a toolset that is up to the task of integrating diverse and frequently changing systems.

Enter Python

Back in 1996, in the *101 Dalmation* days, ILM was exclusively an SGI IRIX shop, and the production pipeline was controlled by Unix® shell scripting. At that time, ILM was producing 15-30 shots per show, typically only a small part of each feature length film to which they were contributing.

Looking ahead towards more CG-intensive films, ILM staff began to search for ways to control an increasingly complex and compute-intensive production process.

It was around this time that Python version 1.4 came out, and Python was coming into its own as a powerful yet simple language that could be used to replace Unix shell scripting. Python was evaluated, compared to other technologies available at the time (such as Tcl and Perl), and chosen as an easier to learn and use language with which to incrementally replace older scripts.

At ILM, speed of development is key, and Python was a faster way to code (and re-code) the programs that controlled the production pipeline.

Python Streamlines Production

But Python was not designed just as a replacement for shell scripting and, as it turns out, Python enabled much more for ILM than just process control.

Unlike Unix shell scripting, Python can be embedded whole as a scripting language within a larger software system. In this case, Python code can invoke specific functions of that system, even if those functions are written in C or C++. And C and C++ code can easily make calls back into Python code as well.

Using this capability, ILM integrated Python into custom applications written in C or C++, such as ILM's in-house lighting tool, which is used to place light sources into a 3D scene and to facilitate the writing, generation, and previewing of shaders and materials used on CG elements. It is the lighting tool that is ultimately responsible for writing the 3D scene out to a format that a renderer can interpret and render.

At the same time, more and more components, such as those responsible for ILM's many custom file formats and data structures, were re-wrapped as Python extension modules.

As Python was used more widely, extending and customizing in-house software became a lot easier. By writing in Python, users could recombine wrapped software components and extend or enhance standard CG applications needed for each new image production run. This let ILM staff do exactly what production needed at any given time, whether that meant allowing for a specific look for an entire show, or just a single CG character or element.

As it turned out, even some of ILM's non-technical users were able to learn enough Python to develop simple plug-ins and to create and modify production control scripts alongside the technical users.

Python Unifies the Toolset

Encouraged by its successes in batch process control and in scripting applications and software components, ILM started to use Python in other applications as well.

Python is now used for tracking and auditing functionality within the production pipeline, where an Oracle database keeps track of the hundreds of thousands of images that are created and processed for each film. DCOracle2, one of the Oracle integration libraries available for Python, has performed well in this task and is now in use on Linux, IRIX, Tru64, and Solaris.

Python is also used to develop the CG artist's interface to ILM's asset management system. Designed to be modular, this tool has been enhanced by a large group of developers and non-developers alike to extend well beyond its original mandate. The application is now used not only to manage CG assets and elements, but also in daily shot review, as a network-based whiteboard, as an instant messenger, and even allows an occasional game of chess.

As Python was applied in more ways, it slowly crowded out a plethora of competing technologies for shell scripting and batch control, embedded scripting, component software development, database application development, and so forth. Python's versatility ultimately simplified the developers' toolset and reduced the number of technologies that needed to be deployed to ILM's thousands of production computers. This new, simpler toolset translated directly into an easier to manage and more reliable development and production process.

Hardware Costs Reduced

Although chosen initially for its ease of use and integration capabilities, Python's portability to many other operating systems eventually became one of its key strengths. Once Python was in use, it made the production control system portable. This gave ILM additional freedom in making hardware technology choices, including a large-scale introduction of commodity PC hardware and Linux, a move that has saved the company substantial amounts of money in recent years.

Source Code Access Important

After having used Python intensively for six years, ILM has yet to run into significant bugs or portability issues with the language. As a result, since Python 1.5 ILM has been able to rely on stock distributions in unmodified form.

However, availability of source code for the language acts as an important insurance policy should problems arise in the future, or should custom extensions or improvements become necessary. Without this, ILM could never have bought into Python so heavily for its mission-critical production process.

One case where access to source has already been beneficial was in ILM's continued use of Python 1.4, which is generally considered obsolete. Because the production facility is under continuous use, upgrading systems to new Python versions would result in significant disruption of the production process.

Instead, ILM installs new systems with newer versions of Python but maintains older systems only so they can run the same scripts as the newer systems. Supporting this mix has required access to the Python sources in order to back-port some changes found in newer Python versions, and to reimplement portions of newer support libraries under older versions of Python. ILM is currently running a mix of Python 1.4, 1.5, and 2.1.

Python Tested by Time

The visual effects industry is intensely competitive. To stay on top of the heap, ILM continuously reviews its production methods and evaluates new technologies as they become available.

Since its adoption in 1996, the use of Python has also been reviewed numerous times. Each time, ILM failed to find a more compelling solution. Python's unique mix of simplicity and power continues to be the best available choice for controlling ILM's complex and changing computing environment.

XIST: An XML Transformation Engine Written in Python

Dr. Walter Dörwald and Dr. Alois Kastner-Maresch

About the Authors

Before receiving his Ph.D. in 2000, Dr. Walter Dörwald researched forest growth simulations and artificial life at BITÖK (the Bayreuth Institute of Forest Ecosystem Research), and developed a large C++ framework for simulation and visualization. In 2000, he co-founded LivingLogic AG, and has been responsible for the company's fundamental tools and technologies ever since.

After getting his Ph.D. in Mathematics in 1990, Dr. Alois Kastner-Maresch lead the Forest Ecosystem Simulation research team at BITÖK. In 2000, he co-founded LivingLogic AG, where he is CEO.

www.livinglogic.de/Python/xist/

Summary

XIST is a XML transformation engine written completely in Python at LivingLogic AG, a software development company specializing in web technology. XIST was designed to facilitate the task of creating and maintaining large web sites.

Background

Soon after we began creating web pages in 1994, it became clear that typing HTML files by hand is tedious and cumbersome, and we began to search for tools to simplify the repetitive task of HTML generation.

Early on, we discovered and started to use an HTML preprocessor named hsc. This tool supported generation of pages from templates by defining new markup tags and controlling how these tags would be transformed into HTML, somewhat like XML/XSL does now.

Unfortunately hsc had certain limitations: It didn't support local variables, and there were no control structures except conditionals. Even arithmetic was not possible. Our first web sites developed with this system consisted of a mix of hsc macros and Perl scripts that generated hsc source files.

In 1998, hsc's author halted further development, and we became quite motivated to find an alternative. At first we decided to continue development of hsc ourselves, and planned to make it compatible with XML, which was beginning to become popular at the time. But extending hsc, which is written in C, proved quite difficult. For example, adding Unicode support required rewriting the entire I/O system. It became clear that we needed to find another toolset for our web development.

XIST is Born

Around this time we discovered Python and decided that it might be a good way to completely rewrite hsc from scratch. Python includes XML parsing capabilities that we felt could be used as the basis for our work:

Instead of writing macros in hsc, we could write XML that could be processed through a simple mapping from XML element types to Python classes.

In this approach, XIST generates an extended Document Object Model (DOM) on parsing each XML file. Classes defined for each element in the file are instantiated as the DOM is generated, and methods on the classes are used to perform the necessary XML transformations during page generation. This allows us to realize our web templates with the full power of an object-oriented scripting language.

During implementation, we found that all of the key features of hsc could be supported quite easily in Python:

- Automatically calculate image sizes? The Python Imaging Library does this with ease.
- Parse XML files? There are several XML parsers available in Python.
- Load and store XML to and from databases? The Python DB-API is standardized and modules exist for MySQL, Postgres, Interbase, Oracle, ODBC, Sybase, and others.
- Fetch XML from the web? Python's urlparse and urllib standard libraries were made for that. Handle Unicode? Python 2.0 fully supports Unicode out of the box.

Implementing the first prototype version took a few weeks of spare time programming and turned out to be very successful. Python provided a much shorter path from concept to implementation than any of the other programming languages we have used. And so XIST was born.

XIST's development continued in Python and today XIST is the basis of a successful company which employs 15 people. XIST is now used in all of our web projects at LivingLogic AG.

Content Management with Python and JSP

On top of XIST, LivingLogic has developed a content management system called XIST4C (4C means Content, Community, Collaboration and Commerce). This system combines the advantages of XIST's abstracted page layouts with pre-compilation of page templates to Java Server Pages that are ultimately used to deliver the content to the web.

By using XIST tag libraries instead of JSP tag libraries we are able to build optimized Java Server Pages that run considerably faster than their JSP tag library counterparts, without any changes to the JSP files. This performance gain is a result of the fact that the XIST preprocessor takes care of many compute-intensive operations, that might require dynamic type introspection, string processing etc. and would be executed by the Java tag library code during page load.

Fast development combined with low hardware requirements make XIST4C especially suitable for small and medium-sized enterprises. This has allowed us to achieve a unique competitive advantage, and to realize projects at a much lower cost.

Prototyping Java Systems with Python

In 2000, LivingLogic was engaged to develop a product for modeling business process work flow and to automate business processes. Inspired by our earlier successes with XIST, we decided to develop a Python prototype. This decision was made even though our contract required us to produce a Java-based prototype for delivery to a large group of developers that would turn it into a marketable product.

The approach of using Python early in prototyping made it possible for us to study concepts using working code almost from the start of this project. Although we had to rewrite our Python prototype in Java, the overall time spent on prototyping was lower than we have seen in other projects that used only Java.

Conclusion

Python is easy to learn and use, produces maintainable code, and packs enough power to make it a suitable choice for many application domains and project sizes.

Some of the features that we like best about Python include:

- Python's extensive standard library and a considerable range of available third party packages support development in many application domains.
- An unsurprising syntax and the widespread and consistent use of a few basic concepts, like namespaces, help to make Python code readable and maintainable.
- Extensive and easy to use introspection facilities make Python easy to learn interactively by discovering its capabilities, including documentation, from the command prompt.
- Python is readily extensible in C or C++, so it is easy to incorporate non-Python modules into an application.

Python has played an important role in the success of LivingLogic AG, and will continue to be the basis for most of our software development efforts.

Python is Rackspace's CORE Technology

Nick Borko

About the Author

Nick Borko is the Director of Internal Application Development and the project manager for Rackspace's enterprise database application, CORE. Rackspace Managed Hosting is the leader in delivering managed hosting services to small and medium enterprises. All customer platforms include state-of-the-art data centers, customized servers, burstable connectivity, 99.999% uptime SLA, a dedicated account manager, instant emergency response and access to live expert technicians 24x7 for support of all hardware and core software. Founded in 1998 and headquartered in San Antonio, TX, Rackspace manages servers for customers in more than eighty countries.

www.rackspace.com

Introduction

To be the industry leader in managed hosting, you have to be fast and flexible. By using Python to implement our enterprise data systems, Rackspace can quickly and effectively change its internal systems to keep up with shifts in the industry and in our own business processes. We do this through a central customer information system called "CORE," which is used both for Customer Relationship Management (CRM) and Enterprise Resource Planning (ERP). Python and CORE are key factors that enable Rackspace to provide our Fanatical Support(tm) and faster customer service.

Background

Rackspace's central customer database started as a simple ERP system to provision and track managed servers. It began humbly, as a small collection of PHP pages that did the job nicely for the few hundred servers that was the beginning of Rackspace's customer base.

As Rackspace grew, that small PHP system became the center of business at Rackspace. Every time an opportunity to automate a process presented itself, it was rolled into that system.

After a couple of years, the result was a big, unmaintainable mess of thousands of PHP pages and modules that had been written and maintained primarily by one person. The limits of PHP (then version 3) had been stretched thin, the system was too much for one person to maintain, and it was difficult to bring in new people to help with it.

Our first attempt to update the system came when PHP version 4 was released. This release promised better object-oriented capabilities, and the time was right for Rackspace to dedicate more people to the project.

The system was totally redesigned from the ground up, including new database schemas and application design strategies. At this time we re-dubbed the project "CORE," an acronym for Core Objects Reused

Everywhere, in order to reflect the overall design goal for CORE: modularity and reusability across all systems in the company. With that goal in mind, our team went to work using the object-oriented features of PHP.

While we were able to re-fit the application and add increased functionality, the project ultimately failed due in large part to the problems encountered while using the object framework provided by PHP.

Memory leaks, inconsistent interfaces, inconsistent internal data models, randomly freed objects, multiple object copies despite explicit use of references, internal PHP errors, and untraceable code failures made the task all but impossible to accomplish in PHP.

Even after we achieved a relatively stable code base, we were nowhere near our goal of Core Objects Reused Everywhere because we had to depart from pure object-oriented methods just to work around the problems inherent in PHP. It became clear that PHP was unsuitable for our large scale, mission critical projects. A new solution had to be found.

Python in CORE

We had always considered Python to be an excellent candidate for implementing our enterprise system, but it was initially passed over in favor of building upon the existing (vast) code base we already had in PHP. At that time, we felt that PHP could be used successfully in CORE by introducing a better structured system design.

Unfortunately, that wasn't enough to overcome our other problems with PHP, so we re-evaluated our situation. The first alpha version of Python 2.2 had recently been released, and we decided to begin work on a new CORE framework using the new features that were available in that version.

The Power of Introspection

One of the first tasks in writing the new framework was to build its database interface.

Python's introspection model had been significantly enhanced with the release of Python 2.2. We decided to use it to build a generic database interface class, based on a DBI 2.0 compliant database connector. In this approach, rather than writing queries or table-specific wrappers by hand, a meta-class abstracts all database queries into a single clean API.

We create descendents of this meta-class to make an API for each table. Each table's class contains a few class constants that describe the columns in the database. This way we can add new tables to the overall API quickly and simply without having to worry about implementation details for any specific table.

The API also uses meta-data to automatically validate and convert values passed to the database. This is done by a "normalizer" function that converts the Python data types being passed through the API into valid SQL values. The function also verifies types and formats that are not necessarily checked by the database or by Python, such as phone numbers and ZIP codes.

Reusing Objects Everywhere

Once the database API was complete, we created a second layer of classes on top of it. This higher level API implements the business logic for specific applications, such as contact management or trouble ticket handling. It also prevents users from performing operations that are inconsistent with Rackspace's business practices, or assigning data that would result in other types of high-level corruption of the data in the database.

With the creation of this second layer, we

achieved our original goal of Core Objects Reused Everywhere. Programmers throughout the company began to use this API to implement interfaces to application functionality. This required little interaction with our API development team, and it could be done without fear of misusing the API.

While we designed the API primarily for CORE, the central enterprise application, it is reused in a number of other systems at Rackspace. For example, one group built a SOAP server on top of the API, in order to access it from their PHP applications. Other applications use the API directly, and it has been extremely gratifying to see our work reused and integrated so easily with other systems.

Integrating Python with Apache

With the API in place, our next task in developing CORE was to find a useful templating module to integrate our Python code with HTML pages running on the Apache web server.

After looking at a number of available Python-based templating modules, we opted to create a simple parser of our own. Our approach was to convert server-side template pages into Python servlets whose output is sent by the HTTP server to the user's browser.

Although this was a fairly simple exercise, we did run into some problems stemming from our design of the CORE database meta-class. We found that altering classes and modules at runtime, as is done by the meta-class, violates guidelines imposed by Python's optional restricted execution environment. Since we felt that restricted execution was a necessary component in supporting a persistent web module, we opted to deploy CORE using CGIs rather than mod_python or similar persistent solutions.

Since fast hardware and multiple servers are readily available, and since our template parser pre-compiles and caches the Python servlet code that it produces, the CGI solution is sufficient for our needs. It also allows us to resolve issues such as database connection pooling and restricting the execution environment outside of Python.

Unit Testing

Thanks to the unit testing module that comes with Python, our projects are reaching production with far fewer bugs than we had ever thought possible when we were using PHP. During maintenance with PHP, there was always a question of whether a change in one place would break something else in another part of the application.

We now write unit tests for each and every API as the API is being designed. This means that we can verify the changes in one module as well as its effects on all the others simply by running the unit tests for the entire API.

Since introducing Python and unit testing, the nature of the bugs that we see in deployed applications has shifted to include primarily those in the user interface, such as layout problems or faulty event handling.

These days, very few bugs come from the API itself, and even those are generally the result of poor revision management or DBA coordination during application deployment. Python can't solve _all_ problems during development, but it certainly has reduced the number of critical system defects for us.

Documentation

Lack of documentation has been a major problem with our previous development efforts. We tried several tools and policies to document our PHP efforts, but in the end these failed. Code changed too quickly, and the code-level documentation tools available

for PHP at the time were too finicky to justify the amount of effort required to get the documentation to parse correctly. Additionally, despite careful planning and coding strategies, the mixture of PHP and HTML made deciphering and understanding the code more difficult.

Fortunately, Python was designed with documentation in mind, with the use of "doc strings" for modules, classes and methods. Since documentation is actually a part of the language itself, and pydoc is a standard module in the Python distribution, it was easy to extract API documentation to HTML and other formats.

Over time, we have found that the syntactic structure of Python makes for extremely readable code, and that in itself helps in the overall task of documenting and maintaining code.

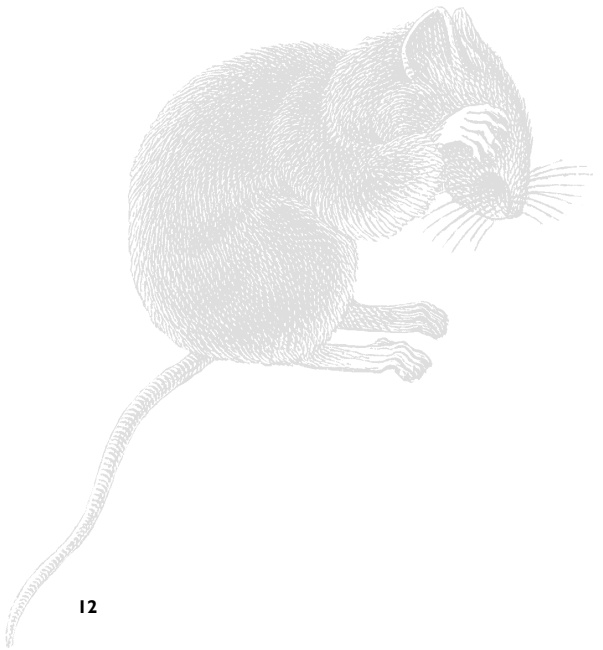
Conclusion

Python has dramatically improved development processes for the CORE project, and it has led to the faster development times and more rapid releases that allow us to keep up with Rackspace's ever-changing business processes.

Python enabled us to create a sophisticated dynamic data model that is flexible and easy to use for abstracting database operations. With it, we realized our goal of Core Objects Reused Everywhere.

Python's integrated unit testing and documentation tools greatly enhance our ability to deploy and maintain a more stable, error-free product.

The result is a successful enterprise application that is instrumental in the delivery of Rackspace Managed Hosting's promise of Fanatical Support, Unmatched Speed, and Unlimited Flexibility in the managed hosting industry.



Putting Web Services to Work with Python

Dr. Tim Couper, Marc-Andre Lemburg,
and Siena Technology Ltd.

About the Authors

Dr. Tim Couper (tim@siena-tech.com) is the chairman of Siena Technology. He holds a mathematics D.Phil. and has twenty years' experience running software companies. He now spends most of his time consulting as architect and technical lead for large development projects, and has been extensively involved in planning and coordinating the 2002 and 2003 Python UK conferences.

Marc-Andre Lemburg (mal@siena-tech.com) is the Chief Technical Officer (CTO) of Siena Technology. He holds a degree in mathematics from the University of Duesseldorf. Marc-Andre has been working with Python since 1993, is a Python Core Developer, board member of the Python Software Foundation (PSF), author of the well-known eGenix.com mx extensions (mxODBC and mxDateTime), and was one of the executive organizers of EuroPython 2002.

www.siena-tech.com

Introduction

Consultants naturally try to provide their customers with the best solutions for a problem. Sometimes this means exploring new areas together with the customer or directing the project into a solution space that better fits the problem than the usual "buzzword-compliant" approaches. We've seen these fail too often, misleading the project into solving problems relating to the selected technology, rather than meeting the original project plan.

Python Goes Fortune 500

In a recent project for one of our customers, we faced a problem that is quite common in Fortune 500 companies: Multiple clients running on unmanageable client machines are tied closely to complicated database relationships, making management and further development very difficult.

Thanks to the marketing efforts of several large server vendors, our solution space was quickly identified. What the company needed was Web Services, or put simply, a way for a client application to talk to the server side in a reasonably standardized manner. Our customer was already using two of the dominant technologies in this area: Microsoft .NET and BEA's WebLogic® J2EE server.

When we came into the project, a team was already trying to solve some of the company's problems using the J2EE platform. However, we found that they were spending more time trying to work around design problems in the technology than actually writing code for the services. Since the company had already been introduced to Python in a smaller XML project, we were able to convince the project lead to try a new technique based on Python.

The idea was to leverage the efficiency of Python programming together with its good database connectivity to compete against the J2EE team. What we

needed was a stable and robust server implementation and a flexible way to write and publish Web Services. Fortunately, the server had already been written in the form of the easily extensible eGenix.com Application Server, so our task was simply one of adapting this server to make writing services as easy as possible.

Keep IT Simple, But No Simpler

Following the Python paradigm of "obvious is better than obscure, explicit is better than implicit," we chose the most straightforward possible way of dealing with Web Services: Each service was mapped to a Python class, which provided the public methods to call from the client application. There were two reasons that we felt that service implementations should look no different than any standard class implementation in Python: (1) a programmer should not need to learn a new way of coding just to be able to write services, and (2) existing integrated development environments (IDEs) should be used to make coding services even easier.

Since Python is an object-oriented programming (OOP) language in all respects, the implementation we chose placed basic shared functionality into a Service base class that hides networked server interaction from the programmer. As a result, the service developer only needs to think about the business logic in the service methods and can rely on the server to automatically provide database connection pooling, protocol handling, transaction control, and all the complicated interactions that are needed to make a server side implementation robust.

Client-side Bliss

Another design goal for the system was to simplify client-side programming as much as possible, in order to make it easy to adopt the new technique. Just as service writers shouldn't need to think about low-level database connectivity, we felt that client application programmers shouldn't be bothered with the details of setting up connections and talking to the server side. We wanted to design a very simple application programmer's interface (API) which would hide all the complications inherent to networked client/server interaction.

Client side agents for Java® and Windows' COM interface made this possible by enabling access to the Web Services from all major client application environments such as Visual Basic (VB), Visual Basic for Applications (VBA as used in Word, Excel, and Access), Delphi, C++, Java, C# and others. These not only hide the protocol level from the application programmer, but also provide the key to enabling security and fail-over solutions.

Web Services standards are still in the planning stage, and it is not at all clear which of the proposals will be accepted by standards organizations. By creating a true middle tier, we were able to hide the particular methods and protocols we chose inside of the Siena client and server, and were free to use existing security standards and authentication modes to build a secure communication channel.

Above and Beyond Web Services

With these basic building blocks, we were moving towards realizing the Siena Web Services Architecture, which included a Python-based server side, a COM client side agent (also written in Python), and a Java client side agent.

One distinct advantage of writing both the server side and COM client side of our Web Services architecture in Python was that we could automatically replicate services from the server to the client side. This was

possible whenever a service did not depend on database connectivity or other elements specific to the server-side environment. Python's data packaging facilities and portable byte code format made this operation quite easy to implement. The result was a significant boost in application performance, reduced network bandwidth requirements, reduced network latency, and increased server performance, all without sacrificing the efficiency of centralized server-side management that makes Web Services so attractive to IT management.

The Python Mantra

Still missing in our plan were the skills needed to code Python servers and clients. Most of the programmers in our team knew only a mix of Java, Visual Basic, and C++. While the J2EE group was working on solving J2EE problems, we invested a day in teaching Python to the rest of the team. Python wasted no time making its way into the hearts and minds of these programmers. It was a thrill to hear fellow programmers chiming in with our own Python mantra: "This is what I've always been looking for."

Results

Happy programmers are good programmers, and good programmers work efficiently. That's what project management learned at this point in our effort. The group's Web Services programmers quickly caught on to the new Python-based system and development progressed at amazing speed. Services could now be implemented in a few minutes rather than the days needed using the typical J2EE approach. Now most services were completed and deployed in less than a day, and the ease and speed with which they can be modified and tested has made an incremental approach to service development possible. And IT management was excited to see the overall high performance of our solution.

The Siena Web Services Architecture has become a crucial mission-critical component for this customer, as it moves from a two-tier to three-tier architecture and adds fail-over and security to their Web Services.

The Siena Web Services Architecture will soon become part of our product line. If you are interested in the solution, please visit our website at www.siena-tech.com/ or contact us directly.

PS: After several months of effort, the J2EE team never did get their Web Services working.

Python Streamlines Space Shuttle Mission Design

Daniel G. Shafer

About the Author

Dan Shafer is a freelance author and sometime Python coder who hangs out on California's central coast. He is a member of the PythonCard Open Source development team creating a GUI-building framework for Python applications. He makes his living as a writer and a product development consultant. A founder and former editorial director of Builder.com, Shafer has been part of the web development community almost from its inception.

This article was previously publised on Builder.com

Introduction

Software engineers have long told their bosses and clients that they can have software "fast, cheap, or right," as long as they pick any two of those factors. Getting all three? Forget about it!

But United Space Alliance (USA), NASA's main shuttle support contractor, had a mandate to provide software that meets all three criteria. Their experience with Python told them NASA's demands were within reach. Less than a year later, USA is nearing deployment of a Workflow Automation System (WAS) that meets or exceeds all of NASA's specifications.

"Python allows us to tackle the complexity of programs like the WAS without getting bogged down in the language," says Robin Friedrich, USA's Senior Project Engineer. Friedrich conceived of the WAS project in response to a significant gap in the way shuttle mission planning was handling data management. "Historically," Friedrich says, "this data has been communicated using paper and, more recently, data file exchange. But both of these approaches are error-prone. Catching and fixing errors as well as responding to frequent change requests can bog such a system down." Complicating the issue was the challenge of finding money to improve the flight design process in an era of declining budgets for space activities.

"Just in time" Provides a Solution—and More Problems

USA decided they needed a way to "minimize data changes and the resulting rework." The shortest route to that goal would be to shift the design work to the end of the process so that flight characteristics would have a good chance of already being finalized. In other words, as Friedrich says, "We decided we needed to do this data management work 'just in time'."

A just-in-time solution, however, generally puts more stress on both people and systems to get things right

the first time because postponing these activities to the end of the process means a loss of scheduling elasticity.

"The obvious answer," according to Friedrich, "was to create a central database repository to help guarantee consistency and to provide historical tracking of data changes." An Oracle database was designed to store the information, but a graphical front end to manage the process of workflow automation was clearly an essential component of an effective solution. "We knew from experience—we do a good bit of Java coding in our group–that using C++ or Java would have added to the problem, not the solution," Friedrich maintains.

Python a Mainstay Since 1994

Enter Python. "I literally stumbled across Python as I was searching the pre-Web Gopher FTP space for some help with a C++ project we were doing," says Friedrich. Being an inveterate systems engineer, Friedrich "just had to investigate it." He was stunned by what he discovered.

"Twenty minutes after my first encounter with Python, I had downloaded it, compiled it, and installed it on my SPARCstation. It actually worked out of the box!"

As if that weren't enough, further investigation revealed that Python has a number of strengths, not the least of which is the fact that "things just work the first time. No other language exhibits that trait like Python," says Friedrich.

He attributes this characteristic to three primary language features:

- Dynamic typing
- Pseudocode-like syntax
- The Python interpreter

The result? "We achieve immediate functioning code so much faster in Python than in any other language that it's staggering," says Friedrich. "Java and C++, for example, have much more baggage you have to understand just to get a functioning piece of software.

"Python also shines when it comes to code maintenance," according to Friedrich. "Without a lot of documentation, it is hard to grasp what is going on in Java and C++ programs, and even with a lot of documentation, Perl is just hard to read and maintain." Before adopting Python, Friedrich's team was doing a good bit of Perl scripting and C++ coding. "Python's ease of maintenance is a huge deal for any company that has any significant amount of staff turnover at all," says Friedrich.

The team had already developed a moderately large number of C++ libraries. Because of Python's easy interface to the outside world, USA was able to retain these libraries. "We wrote a grammar-based tool that automatically interfaced all of our C++ libraries," says Friedrich.

Another aspect of Python that Friedrich found eminently significant is its shallow learning curve. "We are always under the gun on software projects, like everyone else," he says. "But for any programmer, picking up Python is a one-week deal because things just behave as you expect them to, so there's less chasing your tail and far more productivity." He contrasts that with C++ and Java, which he says takes a good programmer weeks to grasp and months to become proficient.

Friedrich says that even the non-programming engineers at USA learned to do Python coding quickly. "We wanted to

draft the coding energy of the engineering staff, but we didn't want them to have to learn C++. Python made the perfect 4GL programming layer for the existing C++ classes."

One Coder and 17,000 Lines of Code Later

The WAS project, which has required somewhat less than a man-year of effort, has been coded by a single senior software engineer, Charlie Fly, who has cranked out some 17,000 source lines of code (SLOC). Python plays the central role, managing data interactions and the task network.

In the system, user tasks communicate with a Python data server, which in turn connects to an Oracle server via DCOracle. Using Oracle's built-in trigger mechanism to send a message to WAS as data records are updated, the WAS calculates which tasks are now data-ready and notifies the appropriate user.

At the core of the design is the Task object, which stores all information relevant to a single task in the workflow network. The end user can view the network in a PERT-style chart layout, where color coding reveals at a glance which tasks are finished, which are in process, and which have not yet been started.

Two other graphical interface windows allow the user to manage the dependencies among data items in the network and to view and edit individual task details.

All of the code for the UIs was also done in Python, using the popular Tkinter library along with an open source package of supporting modules. Tkinter is included in all standard Python installations.

"USA is pleasantly surprised by how much quality software we can deliver," Friedrich says. "And each time we demonstrate success with Python, we add a few more believers to my growing list!"

Wing IDE Takes Flight with Python

Stephan R.A. Deibel and John P. Ehresman

About the Authors

Stephan R.A. Deibel has been designing and developing software for almost twenty years. He has worked extensively in medical informatics and as a software consultant. Before finding Python, his projects included one of the first Macintosh-based multimedia authoring systems and an early CORBA implementation. Stephan is now CEO and co-founder of Archaeopteryx Software, Inc., makers of Wing IDE.

John P. Ehresman has been programming for more than ten years in medical informatics and as a software consultant. He has been using Python since version 1.2 and is now co-founder of Archaeopteryx Software, Inc.

www.wingide.com

Introduction

Wing IDE™ is a commercial integrated development environment for the Python programming language. Wing provides developers with a full-featured source editor, debugger, code browser, and many other tools specifically designed for use with Python. Wing works with all forms of Python, whether running as a standalone app, under a web server, or in a custom embedded scripting environment. Several GUI layers (wxPython, PyQt, PyGTK, and Tkinter) are supported, as are Zope and mod_python for web development, and pygame for game development.

Wing was inspired in 1999 by several experiences we, its developers, had using Python alongside other technologies. At that time, we were working as consultants charged with evaluating a number of alternatives for tiered web development. Some of these were based on Java and some on Visual Basic, MTS, and ASP. Concurrently, we happened to be using Python to prototype some of the functional requirements for the web-deployed business applications we were developing.

It wasn't long before we found ourselves comparing our Python prototypes favorably to the actual systems we were developing. Python was a much more productive way to work, and it seemed to result in at least as good an end product.

Unfortunately, our clients never seriously considered Python simply because it wasn't a mainstream (namely Java or Microsoft) technology. But it was clear to us that Python could have been a significant cost saver and competitive advantage for them, and we saw a business opportunity in helping other organizations benefit by using Python.

Development Approach

Work on Wing IDE started almost right away, in mid-1999, initially on a part-time basis. We realized that

writing an entire IDE wasn't going to be easy and wanted to be sure that Python was really as good as it appeared to us at the time. The logical way to approach this was to develop the IDE itself in Python. This would give us proof of concept and let us become early users as we started to develop and debug Wing IDE with itself.

To speed development and keep costs down, we chose to base Wing on as many open source modules as we could find. The GUI was written with GTK, which is accessed from Python via PyGTK. The source editor is based on Scintilla, an open source code editor component. And printing is implemented via py2pdf from ReportLab.

Initial development was on Linux but we planned to support at least Windows and eventually other Unix-like operating systems. For this reason, we avoided platform-specific implementations and chose cross-platform technologies.

Additional development tools used in the project included gcc, Gnu make, latex, pdflatex, latex2html, emacs/xemacs (before Wing was functional), Visual C++ 6, and cygwin.

Results

Our work on Wing IDE has been quite a success. We were able to develop faster than we originally expected, and to deliver Wing IDE on Linux, Windows 98 through XP, Mac OS X with XDarwin, Solaris, and FreeBSD without major platform-specific development work. Today, our product is receiving good reviews and is selling well. All of this has been possible without any outside funding and with a development team of just two people.

The biggest benefits of using Python have been in overall productivity, cross-platform deployment, speed of the resulting application, scalability, rock-solid stability, and its strong support for mixed-language development.

Productivity

Over the course of this project, we have been able to write on average over 175 lines of debugged, documented, tested code per developer per day. Over a period of 660 FTE days, we produced a total of approximately 121K lines, of which 77K were written in Python. Even without considering that a line of Python is typically equivalent to ten or more lines of C, we were extremely pleased with this result.

The entire product, including third-party open source modules, actually contains on the order of 1.2 million lines of code, of which 274K lines are Python.

So why was using Python so productive, even when only 63% of the code we wrote was in Python? There are several answers to this question:

1) Simple syntax—Although use of indentation to indicate program structure sometimes turns off first-time Python users, the reduced typing burden that comes with avoiding {}'s and similar syntactic sugar does matter over the course of writing and rewriting hundreds of thousands of lines of code.

2) Dynamic high-level data typing—Lack of strong data typing, another commonly cited "weakness" in Python, is in practice a significant advantage for most kinds of software development. The bottom line is that you don't need strong data typing in the context of a language like Python. Common coding errors are caught anyway by the type system: You still can't add a number to a string, reference past the end of an array, or call a non-existent class method. Dynamic high-level data typing cuts out great volumes of support code

and makes it possible to write flexible and introspective code (more on this below).

3) Powerful, easy-to-use data structures—Python's built-in list and dictionary data structures can be used in combination to build just about any fast runtime data structure in a snap. This further reduces the amount of support code you need to write.

4) Extensive standard library—Python comes with a vast standard library supporting everything from string and regular expression processing to XML parsing and generation, Web Services tools, and internet protocol support. Many common programming tasks have already been built into the standard library, making it possible to do more with less code. Third-party modules are also available for database access, CORBA, COM, statistics, math, image processing, and much more.

5) Introspection—Python's flexible data typing system extends also to code documentation, classes, methods, and even the way in which methods are called. Python makes introspection extremely accessible to the programmer and, remarkably, introspective code remains readable and maintainable. This can be very useful in redirecting I/O to classes of your own design, writing a tracer that determines code coverage, packing up data to store on disk or send over the network, developing glue code, writing table-driven algorithms, extracting documentation from code at runtime, applying design-by-contract development methods, and in building various types of meta-classes. For almost every programming task, Python makes it not only possible but quite easy to build meta-code where one might otherwise end up building gobs of manually crafted code.

6) Faster development and deeper prototyping—Python increases speed of development to the point where prototyping can be integrated into and interleaved with the primary development process. When it takes only half a day to try out a new approach to a problem, rather than the week it might take in C or C++, programmers are more often empowered to rework existing imperfect code, and to try out new ideas. This results in the more rapid incorporation of experience into an application's design, and leads to higher code quality.

Cross-platform Deployment

Wing IDE runs on a variety of Posix® operating systems and Windows. Throughout our development process, we've been very happy with the way that Python performed across platforms. The same Python source or compiled Python byte code files can be shipped to clients regardless of target platform, making support quite easy.

Speed, Scalability, and Stability

When we started to write in Python, our previous experience in compiled languages led us to believe that we would be spending a fair amount of time either optimizing code or converting it into C or C++ once we had prototyped it. As it turned out, most of the time Python produced a snappy end product that didn't require any extra work.

This happened partially because most Python code is really just a thin interpreted layer over functionality that is written in C or C++. In our case, this included not just Python's fast built-in data structures and standard libraries, but also the bulk of the GTK GUI development layer and the Scintilla source editor.

In the course of development and in responding to thousands of support tickets over a three year period, we have never run into any significant problems with Python itself, either in scalability or stability. Wing IDE can handle software projects with thousands of Python files, and in many cases can run for weeks without problems. To our knowledge, we have yet to see Wing IDE crash because of a flaw in the Python interpreter or its standard libraries.

Mixed-language Development

Python is almost always fast enough but we did run into a few cases where the interpreter introduced too much overhead. The Wing IDE debugger and the source code analysis engine both contain modules that engage in extremely CPU-intensive processing. These modules needed to be written in C in order to squeeze out as much speed as possible. Fortunately, Python is designed to make it quite easy to call back and forth between Python and C or C++.

In most cases, we wrote and debugged code first in Python, and then converted by hand into C. This approach worked well for us. Working initially in Python was much more efficient and the conversion process relatively painless.

Analysis of our records shows that 360 days were spent on 77K lines of Python code and 300 days (almost as much) on 44K of C, C++, or other code. From our experience with code conversions, we believe it is roughly correct for most types of performance-critical code to equate one line of Python with ten lines of C or C++ code. This means that about 5-10% of our application functionality is in C or C++ and the rest is in Python. Even considering that the C/C++ code is somewhat more complex than most of the Python code, these results confirm without any doubt that working in Python is far more productive than working in C or C++.

In hindsight, we believe that we could have converted smaller units of code into C, by writing more general data-driven processing engines and by more carefully selecting code to convert instead of converting whole modules at a time. Our primary goal for Python in the future is to be able to use it more often, even in performance-critical sections of code. This effort should benefit from projects like pyrex, which allows the use of Python-like code in the development of compiled extension modules, and psyco, which is a just-in-time compiler for Python.

Quirks

There are just two quirks that affected our development with Python. The relative impact they had on our project was tiny compared to Python's benefits, but for balance we feel they are worth mentioning:

1) Like Java and other languages, Python occasionally deprecates old features, or fixes minor bugs in a way that can potentially break existing code. This is done over the course of a number of releases, so that programmers will first see deprecation warnings, and only later be impacted by the change. We ran into this only once when Python 2.0 began to disallow multiple arguments to the sequence append method. This problem required changing exactly three easily found calls in our code base of over 77K lines of Python.

2) Different versions of Python can produce incompatible compiled byte code and requires that C/C++ extension modules are compiled against a specific version of Python. For example, while Python 2.2.2

works happily with Python 2.2.1 or 2.2.0 byte code and extension modules, it will print warnings and may run into problems running against those compiled against Python 2.1.x or earlier. There are solid technical reasons behind this design choice, but it does require some additional work when packaging applications for distribution to users running different versions of Python. In the Wing IDE debugger, we solve the problem simply by storing separate directories for each interpreter version and importing modules accordingly at runtime. For the IDE itself, we solve it by shipping with a specific Python interpreter included; a task that's easily accomplished with support found in the Python standard library's distutils package.

No other language we have used has been this devoid of quirks, even those we have used much less intensively and across fewer language revisions.

Conclusion

Without Python, we could not have sustained the Wing IDE development effort long enough to produce what is now a successful software product. Python has been more productive, robust, and portable than any other technology we have tried. Through our experiences providing technical support for the IDE, we know that we are not alone in these assessments. Feedback from our customers often includes strong endorsements for the productivity of Python, Wing IDE, and related technologies such as Zope.

Python Enterprise-Wide at the University of St Andrews in Scotland

Hamish Lawson

About the Author

Hamish Lawson is a software developer in the IT Services department at the University of St Andrews, Scotland. He specializes in web-based information systems.

www.st-andrews.ac.uk

Introduction

The IT Services department at the University of St Andrews, Scotland, develops and maintains software systems used in a variety of capacities throughout the university.

I had several years of experience working with Perl when I took my first serious look at Python back in 1999. Our team's projects were becoming bigger and more complex, and it was obvious that we needed to bring to them more structure and clarity. I had been looking at Java for some time, but its potential benefits seemed to come at the cost of a steep learning curve, and an overall increase in development time. In contrast, Python appeared to offer the prospect of having both clarity and productivity at the same time. And if we ever needed to make use of Java's class libraries there was always Jython, an implementation of Python for the JVM. The increasing number of Python books being published testified to the language's growing popularity, and the number of available libraries was beginning to rival Perl's. This convinced me to give Python a try.

Python Finds a Home

Soon thereafter, I introduced Python to my fellow developers in the IT Services department at the University of St Andrews, Scotland. It is now the mainstay of our software development efforts.

Python has been used successfully by IT Services for a number of projects. These have included systems administration scripts, where it is used alongside Perl and shell scripts, and also sizeable enterprise systems deployed across the university.

By using it on a number of projects, we have come to understand that Python's dynamic nature, support for high-level data structures, and easy object-orientation all lower the barrier to writing well-structured reusable code in less time. The language's clear and simple syntax helps to reveal the sense (or otherwise!) of our code. This makes it easier to understand and reason

about code during development and–more critically–during later maintenance. The fact that Python is so easy to learn has been quite useful as well.

Job Vacancies Facility

One of our earliest Python projects was a facility for university job vacancies. This was implemented using Zope, an innovative Python-based web application server that provides a range of web components plus powerful facilities for templating and integration. I started the project with a colleague who had more experience in web design than programming. After some time, we found that Python's powerful simplicity enabled my colleague to improve his programming skills rapidly, to the point where he was able to continue development of the system by himself. Zope itself also helped in this regard, by reducing the amount of programming that needed to be done in the first place.

Student Record System

One of our larger projects was a system for managing student records. It employs a model-view-controller architecture in order to promote reuse. Presentation is handled by Cheetah, an open-source templating technology for Python. The interface between Python and the underlying Oracle® database is handled by the high-quality DCOracle2 driver made freely available by Zope Corporation.

Matching Students with Classes

Python was also used for a web-based system that managed the process of matching students with the classes they wished to take in the coming year. The system was used by most of our students and a number of our staff for various stages of the process–from selecting preferences in the spring and summer, through to final validation during matriculation at the start of the new academic year. The system implemented work-flow and notification mechanisms.

Because of the number of concurrent users that the system was expected to support, particularly during matriculation, we felt that a traditional CGI approach could lead to performance problems. Instead, we employed mod_scgi, an Apache module that implements the client end of the SCGI protocol for long-running web processes (similar to FastCGI). The server end of the SCGI protocol was implemented by Quixote, a Python-based web framework. Quixote also provided a URL mapping mechanism that simplified the job of publishing objects on the web. Cheetah was again used for handling presentation, and Oracle was used for storage.

We were assigned this project somewhat late in the process, and it was clear that it would come into active use while parts of it were still being developed. To support this, we planned to deliver the system in stages, but we were aware from the outset that many of the system requirements still had to be discovered and understood. We knew this would lead us to rewrite parts of the system as we went along and worked to obviate this with design decisions aimed at decoupling the system's components. Python's dynamic nature and supreme flexibility made it easier to write generic interfaces, which later facilitated the rewriting and refactoring tasks we had to undertake.

Conclusion

Since switching to Python, we are writing better structured and more readable code in less time ... and it's fun! I can't think of a better testament to a programming tool.

Python Powers Journyx Timesheet

Curt Finch and John Maddalozzo

About the Authors

Curt Finch, Journyx founder and CEO, started the company in 1996 after a successful career in the consulting industry participating in and managing engagements with Fortune 100 companies such as Tivoli®, IBM®, and Prudential Securities.

John Maddalozzo, Journyx V.P. of Engineering, joined Journyx in 1999 after a twelve year career in Unix kernel development at IBM's AIX Engineering group.

www.journyx.com/

Introduction

Journyx Timesheet (tm) is a commercial application that provides time, expense, and project tracking. In 1996, Curt Finch, Journyx CEO and founder, was working in the staffing industry when he saw an opportunity to use the Web to accurately collect and store employee timesheet information.

The first version of Timesheet focused on collecting accurate cost information, with an eye towards applying that data in the formulation of new project cost projections. Since then, Timesheet has expanded considerably to facilitate tracking of time, mileage, and expenses, not just for project management but also for billing and payroll purposes. Optional modules are available for paper-less expense reporting, advanced user role management, automated billing and payroll, and to facilitate system access for disconnected traveling users.

Today, Timesheet is platform-independent, flexible enough to be reconfigured by customers to fit unique organizational needs, and scales to tens of thousands of users for the large enterprise.

Python from the Start

Curt Finch chose Python initially on the recommendation of a friend, Steve Madere, who had founded Dejanews.com (now a part of Google®). Describing the rationale for his choice, Curt said "I looked at Java and C and came to the conclusion that one line of Python is ten lines of Java or one hundred lines of C. Developers write code at basically a constant rate so I chose Python which was (and is) the highest level language I've ever seen that is also flexible enough to be generally useful."

Architecture

From the beginning, Timesheet was designed and implemented as a web application. It uses a three-tiered web application architecture with separate layers for web presentation, business logic, and data storage. As time has progressed, the application's functionality has advanced considerably, and Curt's decision to use

Python for an implementation language has proven to be a good choice.

Python is currently used for all application logic in the Timesheet application. This includes all code between the initial Apache dispatch, where mod_python is employed to expedite interpreter instantiation, through the application logic, and down to the point of call out to the database transport layer. Timesheet uses not only the Python standard library but also several independently developed open source Python subsystems, such as PyXML and ActZero's SOAP support. PyXML is used to implement certain business rules and to develop jxAPI, which is a SOAP-based API, into the application logic. Work is in progress to extend this API to define Web Services Description Language templates for the jxAPI functions. The application currently builds and ships with Python 2.1.1.

Timesheet also incorporates several non-Python technologies. The Unix and Linux® distributions are packaged with the Apache HTTP server and PostgreSQL database. The Timesheet distribution for Windows® ships with an optional Microsoft® Desktop Engine (MSDE) database and integrates with Microsoft IIS. Timesheet can be configured to use a variety of third-party databases.

Results

The Timesheet project has succeeded spectacularly, generating millions in revenue and allowing Journyx to grow every year, even under the current economic conditions. Journyx, like many of our customers, uses Timesheet internally as a mission critical part of the company infrastructure. It is used extensively for project tracking, billing, and payroll. To date, approximately 11 person-years have gone into the Journyx Timesheet product, resulting in over 150,000 lines of Python code.

In developing Journyx, the two greatest benefits of Python were the speed with which features could be written and deployed, and its true write-once-run-anywhere cross-platform capabilities. Journyx developers have found that the simplicity and clarity of Python combine with its object-oriented properties to make it a very powerful and productive language. Python's rich standard library, which includes modules for things like string manipulation and HTML generation, further supports programmers in meeting aggressive development schedules.

Because of these properties of the language, Python has enabled Journyx to add features more quickly than our competitors. We've been able to implement SOAP/XML and WSDL support and extended other aspects of the application's functionality well ahead of competitive products. Some of the key enablers of this efficiency in maintenance and improvement is the inherent clarity and readability of the Python language, a vibrant and responsive Python development community, and the high degree of backwards compatibility and stability we have seen as the language design evolves over time.

Python's cross-platform standard library and platform-independent byte code file format allow the deployment of Python modules to any platform, regardless of which platform the module was prepared on. This helped not only in avoiding per-platform development overhead but also facilitates customer support for the Timesheet software product.

Conclusion

Python has been an important competitive advantage for us, and even as our Python code base grows in complexity and maturity, the natural advantages of Python enable us to provide a high quality mission-critical application at a competitively low cost.

Python Resources

Python Software Foundation
http://www.python.org/psf

Pythonology
http://pythonology.org

O'Reilly & Associates, Inc.
http://python.oreilly.com

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

Python Advocacy HOWTO
http://www.amk.ca/python/howto/advocacy/advocacy.html

Python Starship
http://starship.python.net

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

Daily Python-URL
http://www.pythonware.com/daily/index.htm

Pyzine
http://www.pyzine.com

Python Journal
http://pythonjournal.cognizor.com

Upcoming Python Conferences and Events

PyCon DC 2003 - March 26-28, Washington D.C.
http://www.python.org/pycon

Python UK Conference 2003 - April 2-3, Oxford, UK
http://www.python-uk.org

EuroPython 2003 - June 25-27, Charleroi, Belgium
http://europython.org

Python11 at OSCON 2003 - July 7-11, Portland, Oregon
http://conferences.oreillynet.com/os2003