Case Study: Python in a Commercial Environment

Greg Stein, gstein@microsoft.com
Microsoft

Abstract

This paper discusses how Python was selected and used at a small software vendor to produce a commercial software product. It highlights some of the benefits and drawbacks of Python found at various points of the product's life, how some of the problems were solved, and discusses some possible options for increasing Python's use within a commercial environment.

Background: eShop, Inc.

This case study concentrates on a software vendor named eShop, Inc. It was originally formed in May, 1991 as Ink Development Corporation with the intent of producing mobile software (pen-based, typically). This paper's author, Greg Stein, was one of the co-founders of Ink. In December, 1993 Ink reformed itself as eShop, Inc. to produce electronic commerce software. At this time, the Internet was an unknown phenomenon and all online activities were primarily occurring over private X.25 packet-switched networks such as SprintNet or Compuserve's network. At first, eShop produced commerce software for General Magic's MagicCap OS for shopping online through AT&T's PersonaLink online service. Simultaneously, eShop was began work on a Windows-based shopping client, which would communicate with the servers over the X.25 networks. As the MagicCap project was winding down in the fall of 1994, eShop began concentrating entirely on the Windows-based client and redeploying and extending the Unix-based server portion of its system to target the new client. By the summer of 1995, it was obvious that the Internet was having a large impact on the direction of online activities and eShop made some adjustments in its client software to operate as a Web browser helper application. This was marginally acceptable and the software was released to the public in November, 1995. However, Pandora's Box had been opened - the Internet was something that had to be addressed directly, rather than simply accounted for, and eShop had to move quickly or lose its competitive advantages.

The Introduction to Python

Sometime during February, 1995, Greg heard of Python through a good friend of his, John Viega. For the next two months, Greg learned and used Python primarily for hobby-related activities. However, It soon became apparent that Python would be ideally suited to solve many of the operational needs of managing and controlling a 24 hour, 7 day a week, Unix-based server environment. By developing many example scripts to solve certain problems, Greg was able to demonstrate Python's unique advantage, for the problem space, to Steve Peterson, the VP of Development at eShop. Steve agreed with the benefits and agreed to use Python for some development. The deployment of Python was going to be limited to eShop's internal use, as it was unproven technology, but Python's foot (tail?) was in the door.

The Subtle Invasion Begins

Having convinced Steve of Python's utility, eShop decided to hire an intern for the summer to develop a lot of the Python-based utilities that we were looking for. Unfortunately, the intern declined at the last moment, leaving Greg as the only Python programmer at eShop. This was in May, 1995, and Greg decided to find a way to get others to learn Python.

Most of the example scripts that had been built revolved around the use of Tom Culliton's Oracle module. Data was inserted, manipulated, and extracted from Oracle databases containing information about shoppers, products, traffic information, and store layouts. While interesting, and certainly fast to develop, modify, and maintain, it still did not quite have that edge over the C++ classes that were being used to access the database. The other programmer's were still not convinced of Python's overall utility.

The breakthrough came through the use of publishing the database information to a web page. Greg began building some CGI scripts in Python that selected the data and returned it as an HTML page. After showing these small, simple 20 line scripts to a few other people, they became quite interested. Within a couple weeks, two other people were writing CGI scripts in Python to extract data and publish it via CGI. Python's utility had been demonstrated and accepted, but eShop's use of Python remained pretty much at this level for a long time - it was fine for internal use, but it was still taboo for product development.

The "Web-Lite" Concept

In the Fall of 1995, eShop was somewhat at a crossroads. Its proprietary client and server were due out soon, yet the Web was taking off. On one hand, eShop wanted to continue to provide the best possible shopping experience possible, which required the custom client software; on the other hand, it was becoming obvious that a custom client was going to limit the acceptance of the eShop system. Customers were beginning to demand entirely Web-based software. Greg was charged with trying to find a way to create this "Lite" version of a shopping experience via a Web browser.

Prototyping with Python

By the Fall, Python had become well-known, if not fully accepted, at eShop. Over the next three months (from September through the beginning of December), Greg began to build various prototypes to test out ways to shop via a Web browser and ways to leverage our existing investment in tools and the server.

The first system built used Python's socket facilities to connect a server to the eShop server. There was a relatively significant connection cost involved with the eShop server (the eShop clients would connect and maintain a single connection for the entire online shopping session). The protocol that was used was stateless, but did have some setup costs. This intermediate server would maintain a connection to avoid these setup costs, but would still be capable of multiplexing multiple clients through the single connection, due to the stateless nature of the protocol that was used. Next, this server exposed its own listening socket and a higher level protocol. This higher level protocol was modeled after the internal interfaces used within the custom client. Finally, CGI scripts were written to connect to the intermediate server, fetch information, and generate pages with that information. The intent was that the intermediate server would simplify the process of writing CGI scripts for creating a Web-based storefront. This architecture is pictured in Figure 1.

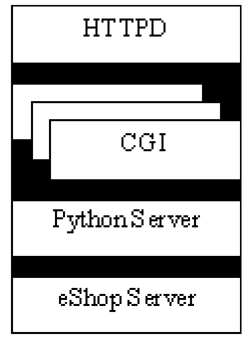

Figure 1: Web-Lite Prototype #1 (CGI-based)

This system worked very well to demonstrate Python's capabilities with networking, protocol handling, HTML generation, and many other facets. Python's ability to be used as a rapid development environment was critical to be able to test out the architecture. Unfortunately, the cost of invoking the CGI scripts was a bit exorbitant to claim a "best of breed" system. Even though Python-based CGI scripts were easy to write and maintain for presenting a storefront, there was no way a customer was going to be able to do this. It became obvious rather quickly that this approach was a dead end.

The second prototype was developed near the end of November. The prototype was more of a way to validate the use of Python than to try out a new architecture for Web-based shopping. In this prototype, Greg recoded the C++ -based server in Python. Having the server written in Python provided a way to quickly take our system in new directions without the cost of writing and maintaining the equivalent C++ code. The direction to take it was still an unknown, but the prototype provided options and a basis for trying out the new directions. In terms of validating Python's capabilities, it was a mixed success. It didn't provide any real change in functionality, any new extension, or any obvious applicability to Web-based shopping. However, it did provide a hard measurement on the benefits of using Python instead of C++: the lines of Python code was 10% of the equivalent C++ code. Both versions of the code were developed by experienced programmers, using the respective languages properly and to their full extent, and both versions had the same percentage and style of commenting. Essentially, all aspects were equal, yet the Python code had much fewer lines. From a software engineering standpoint, this was a tremendous success. Bug counts are always proportional to the number of lines of code, meaning that the Python version should have 10% of the bugs of the C++ version. Further, the fewer lines of code meant that it would have a smaller and more understandable "footprint" in the developers' minds. The Python code was arguably more maintainable due to its improved readability and rapid edit-test cycle (no compile and link step). Lastly, the server could also be shown to be more robust - being entirely in Python, it was not subject to memory-related coding errors such as null pointers, buffer misallocation and overruns, or unfreed or doubly-freed memory. Even if an error occurred within the code, an exception would be raised and could simply be caught at the top level of the system (where the transaction was received from the socket and dispatched). The server would never crash and would never leak memory; uptimes would be indefinite. This prototype also served to help us classify the kinds of errors that would occur within Python software. Primarily, the largest class of errors were typographical errors that would raise a NameError or AttributeError at runtime. Compiled languages would catch this kind of error, but it had to wait until runtime for Python. Luckily, this is a very easy error to detect, diagnose, and fix. Standard testing methodology uses code coverage to verify that all the various paths through the code work as advertised. By using statement coverage (the simplest kind of code coverage), a test and development team can isolate nearly all potential typographic errors in Python code. Various research tools and utilities that have been posted to the public Python forums can also detect these kinds of errors even before reaching the code coverage phase of software development.

Python had proven itself to be capable of implementing a fully functional, highly available server.

The Resistance Fights On

From February through the end of November, Python had been poked and prodded. It was used in many situations and scenarios and was able to cleanly implement every one of them. It was a marked success in being able to create more maintainable software in less time than any other solution we had available. Despite this, it was still not fully accepted as a development language.

Part of this was due to historical, developer attitudes. There is a highly prevalent desire on many developers' minds to require the code to be as fast as possible. Python, being interpreted, could never be as fast as C++; therefore, they reasoned, it was not a viable language. However, this attitude does not take into account market realities: customers want the software yesterday; they want it fully featured; they want it to work. Customers always want the software as soon as possible; they don't want to wait for those extra 90% lines of code to be developed and debugged. In very few cases, will a customer trade off features for performance. Under no circumstance will they accept broken software (say, because it crashes due to null pointer problems). All of these things point to a requirement to be able to develop the software quickly and with high quality; they do not point to a need for arbitrary performance requirements such that C++ is necessitated. They can always throw more iron at the problem, especially if the design allows for scaling to fill a given machine and then across multiple machines. Even ignoring customer requirements, the developers always said "it must be as fast as possible" rather than "it must be as fast as we need it to be." This demonstrates a disconnect between beliefs and requirements. Lastly, Python's rejection was also due to it simply being different and new. Needless to say, all developers are most comfortable with what they know.

In general, these subtle attitudes can only be relieved by direct experience with the language. Getting a developer to actually sit down, learn Python, and then use it, will usually suffice to overcome most barriers. Without knowing all the details of Python, there will be an irrational response to reject it. This was the hardest lesson to learn in getting Python accepted at eShop. However, by making it available and demonstrating its utility (e.g. the CGI examples), it broke down some barriers and got people interested.

At the end of November, we had one item left on our list of reasons to not use Python, and one consideration to take into account. The reason to reject Python was the talent pool - Python is not a common, popular language, so it would be difficult to find experienced Python programmers. For a software company, this reason alone could be enough to reject Python out of hand without a backward glance. However, this is exactly where Python shines. In every single case where a developer has had to learn Python, it has taken only a couple hours for them to be able to read it, a couple days to start writing code, and a maximum of a week to be a contributor. When the training costs for a new employee are considered (in terms of learning the culture, HR, product development processes, the existing code base, etc), the time to learn Python is negligible. Having resolved that one last item on our rejection list, we had one consideration left: we found that some developers simply didn't want to learn Python. They would rather expand and continue their C++ skills than to let them "atrophy" while working on Python projects.

We decided to allow Python as a means of development for actual products where it made sense (mainly on server-related software; we weren't quite ready to cross the GUI bridge yet). There was still some basic attitudinal resistance; for better or worse, the next step at eShop went brute-force: the third Python-based prototype was developed and it was Right; the development team had to adapt.

Web-Lite is Born

At the beginning of December, the Third Python Conference was held in Reston, Virginia at the United States Geographical Survey offices (courtesy of Jim Fulton for making the space available). At that conference, Paul Everitt of Digital Creations demonstrated a unique system. Little did Paul know what the final impact of that system would bring about.

Digital Creations had written an extension module for the Apache web server. This module would take incoming requests, format an ILU method invocation, and dispatch the request to a long-lived object in a separate process. This system solved the critical problem in the first prototype: avoiding the overhead of CGI when requests arrived at the web server. ILU has a nearly seamless language binding for Python; the long-lived objects could easily be written in Python. These objects would effectively act as servers, responding to page requests from a Web-based shopper. Even better, the system was inherently scalable to absurd levels. The Web server already scaled on a machine through multiple processes; through standard DNS round-robin techniques, multiple machines could be used to scale that portion of the system. Since ILU could perform method invocations between machines, the long-lived object could live on another machine. By specifying different objects within the URL for the request, multiple objects could be targeted, each living on a separate machine if needed. At Greg's request, Digital Creations added a feature to the extension module to round-robin, effectively, through a target set of objects. By distributing the objects in this fashion, the long-lived object portion of the system was scalable. Assuming those objects accessed a large database (such as Oracle), standard database scaling techniques would provide scaling. This architecture is shown in Figure 2.

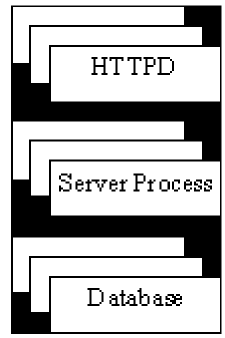

Figure 2: Web-Lite Prototype #3 (ILU-based)

All the pieces had dropped into place. An efficient mechanism for brokering Web requests among long-lived Python-based objects was available. The system was robust and could be scaled to arbitrary levels to support any number of shoppers visiting the site. Moreover, the technology was available immediately and was very easy to use.

Greg returned from the Python Conference and presented the idea to Will Poole, eShop's President, with the additional note that a prototype could be built within two weeks. Almost without hesitation, Will agreed that two weeks was nothing if eShop could have a prototype architecture for Web-based shopping. The third Web-Lite prototype was demonstrated one week later.

Development of Web-Lite

By mid-January, it was apparent that this new system exceeded all expectations on the kind of shopping experience that could be created for Web-based shoppers. Given the huge drive to the Internet and the viability of using this system for them, most of the development team had been shifted to turn the prototype into a real product. Development of the proprietary system virtually halted.

All of the prototypes had been built by a single person. eShop now had a team working on the product. This exposed Python's second weakness: its lack of type declarations removed a subtle means of communication between two component developers. In languages such as C++, a developer could look at a function or method prototype in a header and understand the data types needed to invoke that function. Within Python, headers do not exist, so a developer must look to the source code. Unless that code is commented very well, it will usually not be obvious what the expected type is for a variable named, say, "formArgs" (in this case, it was typically a dictionary with string keys and a list of strings for the value). Our partial solution was to be as consistent as possible in variable naming. If the name "formArgs" occurred somewhere, then you could infer the type; conversely, if you had a dictionary of strings mapping to lists, and it came from an HTML form, then it should be called "formArgs". Needless to say, developers did not comment as well as they could or should have, variables sometimes did not match with a convention or the convention was unknown, or new structures were used that had no precedent for a convention. At this stage of development, the typical error was due to incorrect parameter typing. Python 1.5, with its "assert" statement, should help to improve this aspect of team-based development drastically. On one hand, an assert statement can immediately detect a failure in a calling convention and raise an appropriate and descriptive error; on the other hand, when a developer looks into a source module for the method they want and its parameters, those asserts will be available to provide additional information about the parameters. Of course, valid documentation will always be desired and will always improve the communication between developers on a team, but historical reality states that you cannot rely solely on documentation.

The Web-Lite server was fully functional and very capable by the start of February. Feature development continued until the end of March; partly this was due to changing and expanding requirements (since the system originated from a prototype, it had no rigorous specifications to work against), and partly due to Python's nature - it was simply very easy to extend the feature set. In April, Web-Lite was essentially complete and entering into a true beta test phase.

Oops, eShop is Acquired

eShop's plans to enter into a beta test with participating merchants and service companies was effectively put on hold due to the impending acquisition by Microsoft, which concluded near the end of June, 1996.

By the time of the acquisition, eShop had built an entire Web-based shopping system in about four months (mid-December to mid-April). The resulting server was stable, maintainable, robust, and scalable. The development team was relatively small (less than half a dozen engineers at the high point). It would be fair to say that this feat would have been impossible except for the use of Python as a development language.

Python in a Commercial Environment

Based on the eShop experience, one of the largest impediments to using Python in a commercial or corporate environment is simply historical attitudes. There is no single, simple solution to overcoming this barrier. It is impossible to simply require developers to learn and to use Python. An autocratic decision like that would strike right at the heart of most developers. It would seem that the better approach is to require them to learn Python, so that the developers can make the tradeoffs and choices themselves.

Barriers within the corporation must be removed so that a developer is free to choose Python if it makes sense for the task at hand. These barriers are found primarily in available tools, organizational processes, and a team's acceptance of other developers' choices. Management plays a crucial role in the decision process for allowing Python to be used for product development. If management does not accept it, or resents its use, then Python will never be available to developers to use when the task requires it.

From a sheer technical perspective, Python is usable across a huge variety of problem spaces. While it may not be suitable for a spreadsheet recalculation engine, its extensibility easily allows for that engine to be coded in C and incorporated into a Python-based spreadsheet application. Python's single largest weakness, in the eyes of the author, is Python's native threading capabilities. Until these are improved dramatically, Python will forever be barred from many server-based environments that demand threading. Otherwise, Python excels in all aspects in its capabilities to incorporate complex data structures, control mechanisms, extensibility, interfaces to GUI subsystems, and the like. Python can be appropriate for any application, and pending threading improvements, for any server application.

Python currently lacks many tools that modern developers are looking for. Specifically, these include tightly-integrated source level debuggers (right out of the box), GUI development packages for WYSIWYG creation of user interfaces, packaging methods for software distribution, and readily available tools to solve the typographical and type mismatch classes of errors. Many of these tools are available, but they are not integrated as well as they could be, or they do not come with the standard distribution. Some of the tools (such as a debugger) are needed only as a preference item for some developers; a development team may not need them.

In summary, Python is a viable language for a commercial or corporate environment, for use in simple operational scripting or for full product development, but your needs should be assessed and matched against what is readily available to you from the Python community.