The Perl^H^H^Hython Review

Volume 0 Issue 2 April 1, 2002

Idle hands make idle minds

Web Access http://www.ThePerlReview.com/ **Email** letters@theperlreview.com **Publisher** brian d foy **Editors** David H. Adler, Andy Lester, Adam Turoff **Technical Editors** Kurt Starsinic, Adam Turoff **Copy Editors** Beth Linker, Jack Hattaway, James Tillman **Contributors** brian d foy, Mike Giroux, David Ascher, Adam Turoff, Guido van Rossum, Andy Lester, Paul Prescod, Dave Hoover, Jérôme Quelin

Letters

Here be monsters

I can't be the only one that noticed that Buffy Summers is credited as a contributor to TPR on the cover of the (0, 1) issue. Can it be that—already—TPR has some demons and vampires to vanquish?
– *Paul Barry, Ireland*

brian writes: *Already? We probably had demons and vampires before we started! However, we also have Dave Adler as an editor so it all works out. A Beautiful Mind just won the Academy Award for Best Picture. Ever wonder what secret messages might be hidden in your magazines? Or in the first issue of The Perl Review? We are actually surprised that London.pm did not notice it first. :)*

About this issue

We decided to ditch Perl for something much easier to typeset—Python. Not many people know that Python started about the same time as Perl, and Adam Turoff touches on that in an interview with Guido van Rossum. Paul Prescod shows us why dynamically typed languages, like Python, have some advantages over other options, which you can apply to Perl advocacy itself. David Ascher takes us behind the scenes at the Python Software Foundation to tell us how "There's more than one way to do it" applies to more than just Perl constructs. If you absolutely need a Perl article, Mike Giroux tells you why his first Perl golf attempt does not shatter any records but taught him a lot about Perl.

Python on the web

The Perl Review
http://www.theperlreview.com – the website for this magazine with information for readers and authors

The Python Language
http://python.org/ – All things Python

Vaults of Parnassus
http://www.vex.net/parnassus/ – Python's repository of reusable code

Community News

ActivePerl 5.6.1 Build 632

http://www.activestate.com/Products/ActivePerl/

ActiveState reports that this build of Perl, which they will release mid-April, fixes the recent zlib security issue, adds some support for thread-safety on Unix platforms, and bundles newer versions of some modules including libwww and libnet, as well as version 3 of their Perl Package Manager.

MacPerl v5.6.1 released

http://dev.macperl.org/

You can download MacPerl 5.6.1r1 from the MacPerl Development website. MacPerl v5.8.0 is under development as well.

New Parrot Pumpking

http://dev.perl.org/

Simon Cozens released Parrot 0.0.4, and passes the pumpkin to Jeff Goff. Simon wants to spend more time finishing some of his other projects.

search.cpan.org upgraded

http://search.cpan.org/

Elaine Ashton reports that Sun Microsystems donated a Sun Enterprise 250 server with one gigabyte of RAM and two processors. She configured in Boston then drove to St. Louis—2600 miles—to finish installing it.

Perl Conference 6 news

http://conferences.oreilly.com

Larry Wall and Damian Conway hope to introduce Perl 6 at the O'Reilly Open Source Convention, July 22-26, although Nat Torkington says the Perl 6 compiler may not be ready in time. Richard Stallman and Lawerence Lessig will give keynote addresses.

New books

Publishers: to have us list your book, send us a note at book_reviews@theperlreview.com.

Apache Administrator's Handbook
Rich Bowen; Sams Publishing; 0-672-32274-9; 326 pages; January 2002

Sams Teach Yourself Ruby in 21 days
Mark Slagell; Sams Publishing; 0-672-32252-8; 560 pages; March 2002

JavaScript Developer's Dictionary
Alexander J. Vincent, James Davis; Sams Publishing; 0-672-32201-3; 600 pages; May 2002

Transact-SQL Cookbook
Ales Spetic, Jonathan Gennick; O'Reilly & Associates; 1-56592-756-7; 302 pages; May 2002

Free as in Freedom
Sam Williams; O'Reilly & Associates; 0-596-00287-4; 240 pages; March 2002

802.11 Wireless Networks: The Definitive Guide
Matthew Gast; O'Reilly & Associates; 0-596-00183-5; 464 pages; April 2002

Perl in a Nutshell, 2nd Edition
Ellen Siever, Stephen Spainhour, Nate Patwardhan; O'Reilly & Associates; 0-59600-241-6; 700 pages; May 2002

Perl & XML
Erik T. Ray, Jason McIntosh; O'Reilly & Associates; 0-596-00205-X; 224 pages; April 2002

Perl Golf

Dave Hoover & Jérôme Quelin

The March challenge required golfers to find an integer's *secret number*, an idea taken from numerology. Valid solutions took an integer argument from the command line, calculated its secret number and displayed all intermediate steps. To determine this mystical number, golfers added all pairs of adjacent digits and concatenated the sums. For any sum greater than 9, golfers added the digits to produce a single digit. In the following example, 9 is the secret number of 248:

```
$ perl secret.pl 248
248
63
9
```

Ton Hospel won with a 47 stroke solution:

```
#!perl -l
s//pop/e;print^s/./hex($`%10+$&)%15/eg&&do$0
```

This solution was the evolutionary pinnacle of 14 submissions that began with a humble score of 67. Ton had several 47 stroke submissions, but this final solution improved his tiebreaker score by replacing the loop construct `while+` with the recursive `&&do$0`. The tiebreaker for March favored the solution with the least amount of `[\w\s]` characters.

Bitwise exclusive OR, `^`, controls the recursion. It XORs its operands, `print` and `s///g`, and returns the value to `&&`. Ton's XOR was a golf-ish way of writing `$a!=$b`. Since `print` returns a value of 1, XOR evaluates to true until `s///g` returns a value of 1. `s///g` returns the number of substitutions, meaning that recursion ends when one digit remains.

Ton won the competition with his ingenious dual-purpose use of `s//pop/e`. On the first pass, `s//pop/e` places the command line argument into $_. On all subsequent iterations, the empty regular expression `//` repeats the last successful regular expression, `/./`. Since @ARGV is now empty, pop() returns undef, which becomes the empty string in this context. Thus, `s//pop/e` acts like `s/.//`, trimming off the first character of $_. This behavior cleans the residue left by Ton's otherwise elegant secret number algorithm:

```
hex($`%10+$&)%15
```

Ton used hex() to assure that sums of 10 returned 1 rather than 0 as they would have if he had used the 6 stroke shorter, `($`+$&)%10`. The hex() function converts a hexidecimal number to base-10 (hex(10) == 16) and `%15` maps the result to the final sum, 1.

Stepping through the algorithm using the initial value 248, $& holds 2. $` holds the value of all characters preceding the current match, in this case the empty string. The empty string becomes 0 in the context of %10, leaving:

```
hex(0%10+2)%15
```

For the first digit, the result is the orginal 2. Taking the next digit, 4, the result is:

```
hex(2%10+4)%15
```

This produces 6, the sum of the first two digits, 2 and 4. Continuing on to 8, things get more interesting:

```
hex(24%10+8)%15
```

Step-by-step: `24%10==4`, `4+8==12`, `hex(12)==18`, and `18%15==3`. The brilliance of the algorithm is now apparent. Ton took the two final digits, 4 and 8, and reduced them to 3 ($4 + 8 == 12$, $1 + 2 == 3$). `s///g` is finished and $_ holds 263. On the next iteration of the program, is stripped off by `s//pop/e` and due to the -l switch, 63 is printed with a newline appended.

Lars Mathiesen won the beginners' category with a score of 57 and beat many noted veteran golfers:

```
#!perl -l
$_=pop;print,
s/(?=(..)?)./$1%9||$1&&0+$1&&9/gewhile/./
```

To see all 590 solutions submitted by the 133 participants, along with information for this month's challenge, visit http://perlgolf.sourceforge.net/.

Guido van Rossum: Benevolent Dictator for Life

Adam Turoff

Abstract

I interviewed Guido van Rossum, creator of Python and leader of the Python community, also known as Python's *BDFL*, the Benevolent Dictator for Life. My goal was not to ask the same boring questions that Guido has answered many times before, but to ask the questions whose answers that you—Perl programmers—would be interested in reading.

Earlier this winter, I had the opportunity to hear Guido van Rossum speak to the New York Perl Mongers when he was in town for LinuxWorld. The discussion was quite lively, and the mood was quite civil—thankfully devoid of the holy wars that previously dominated the Perl versus Python discussion. Last month, I caught up with Guido in an email interview to follow up on some of the issues raised at that meeting. The questions I most wanted to ask related to why Python is different, and what lessons Perl programmers should be trying to learn from Python.

1 Creating Python

Programming languages come and go, yet Python is alive and well ten years after its initial release. What do you think Python got right, and what aspects of Python do you feel need improvement?

As you know, Python's predecessor was ABC, developed in the early eighties. When I created Python, I started out with strong opinions and good intuition on what was good programming language design. Much of this was rooted in the philosophy of ABC's designers, and the rest came from my own (then) 15 years of programming experience. I was lucky that I wasn't bound by a desire or need for backwards compatibility with older tools, so I could freely pick and choose only the best from other languages. I also had good firsthand experience with the areas where ABC failed, both in terms of language design (e.g. its innovative terminology was a failure, and it had too much emphasis on keeping things sorted) and in terms of implementation (ABC was a single monolithic program that was hard to extend).

I think the areas most in need of improvement are not in the language's design or implementation, but in the community support around it. For example, Python's equivalent to CPAN (the Vaults of Parnassus) could use work, and we're desperately looking for a catchy marketing slogan to explain Python's benefits in eight words or less. "import this" just doesn't strike me as a particularly good catch phrase.

What problems did you want to solve that led you to create Python? What goals do you have for Python today?

The specific problem back in 1989, was to have a programming language that would run on an operating system that was very different from Unix, and would let me write relatively small often throwaway programs that would require too much time to code in C, but weren't easily done as shell scripts either. Because of the non-Unix requirement, porting Perl wasn't an option (apart from the fact that it offended my good taste.

Today, Python has shown it's good for quite a bit more than that. It's a glue language, it's a scientific steering language, it's an extension and end-user programming language, it's a prototyping and testing language, it's a web, database and GUI language, it's used for building distributed systems consisting of hundreds of thousands of lines, and it's used in primary computer science education. My goals are to keep the language usable and make it even more usable, for all those purposes. All this without getting any group of users to revolt.

It's well known that you started Python as a language to teach computer programming. Many of your efforts over the years have involved using Python to teach programming as well. What's the best non-Python environment that you have seen for teaching programming? How close are Python and IDLE to your ideal environment for teaching programming?

Actually, while Python descended from a teaching language (ABC), it was originally not intended for that purpose. I just kept enough ideas from ABC that it turned out to be a good teaching language—in other words, teaching was in its genes.

Some of the better environments I've seen for teaching are Smalltalk (squeak) and DrScheme. Python the language is pretty close, but for teaching a parser with much better diagnostics (especially warnings while you type, unintrusively marked a la spell checking in MS Word) would be nice. IDLE the environment is pretty close too, but it needs project management, an easier way to run a script (and the script should be run in a separate process), and a better debugger.

2 Improving Python

Beginning programmers and professional programmers have two different sets of requirements for a programming language or a development environment. What does Python offer for the beginner? For the professional? For the casual programmer?

For the beginner: easy-to-learn syntax, easy-to-use data structures, and an interactive interpreter that's open to experimentation. Also a large set of readable examples in the form of the standard library, a friendly community that's eager to answer questions, good free on-line tutorials, and plenty of books to learn from.

For the pro: a large standard library and an even larger library of third party add-on modules and packages, exceptions, several layers of programming structuring devices (packages, modules, classes), a choice of several different GUI toolkits, and the ability to write your own extension modules in C, C++ or Fortran.

For the casual programmer: a syntax that's easy to remember, a large standard library with pre-built solutions, a vast amount of documentation, and real power under the hood when you need it.

What major features have been added to Python over the last few releases? What major features are slated for inclusion in the next few releases?

In 2.0, we've added list comprehensions, augmented assignment, a real garbage collector, string methods, Unicode and XML support, and a new, more Perl-compatible regular expression engine. In 2.1, we added nested scopes and weak references, amongst others. In 2.2, we added iterators, generators, the ability to subclass (most) built-in types, and unified ints with long ints. For more on all these changes, I recommend the series of articles by Andrew Kuchling, "What's new in Python 2.x", with URLs:

```
["http://www.amk.ca/python/2.%s/" % x for x in "012"]
```
[1]

[1] or, just start at http://www.amk.ca/python/

(BTW, that was a list comprehension, introduced in 2.0.[2])

In 2.3, we plan to do mostly consolidation work, and add to the library. There are a lot of proposals under consideration, and some of them will make it into 2.3, some will make it into 2.4 or later, and some will be shot down or replaced by something better. I've written a highly personal and subjective review of most of the proposals recently, on view at

```
http://www.python.org/doc/essays/pepparade.html
```

I'm also thinking about adding a new 'bool' type to the language, which will make it a little easier for programmer to express their intention of using a variable to hold a truth value. To my surprise, it brought out a lot of controversy: the responses cover the entire spectrum, from "at last, but you're not going far enough" to "who needs it" and "this would ruin the language". So I'm still thinking about that one.

3 Python and Perl

One of the things that distinguishes Python when compared to Perl is that Python has been implemented multiple times: CPython, JPython/Jython, Stackless Python, etc. Have multiple implementations of Python been a goal for the Python language, or is it an interesting side effect of Python's standard implementation? Where would you like these alternative implementations to take Python?

These really are only two implementations, Jython and CPython: Stackless is just an add-on for CPython. But there's a third implementation, Python for .NET. There are also several projects aiming at translating Python to C code, which will eventually become separate implementations in their own right, with different semantics in corner cases.

It wasn't an original goal to have multiple language implementations, but I've always resented language features that were too closely tied to a particular implementation technique. I've actively encouraged the development of alternate versions, in particular Jython. I've enjoyed the opportunity it gave me to become conscious of the difference between accidents of an implementation from the intentions of the language designer, and it has helped clarify issues that might change in future versions of CPython (like the reliance on reference counts). It has affected the design of some newer features. For example, we decided that generators would use a new keyword, so they would be recognizable by the parser, which was a requirement for a 100% pure Java translator.

I hope the alternatives take Python to places that CPython cannot reach. Jython is an obvious example; the C translators will eventually break through the speed barrier.

What lessons do you feel Python can teach Perl programmers (both as a language and as an environment)?

I think you'd have to ask a converted Perl programmer; I don't know enough Perl to be able to say for sure. But I guess that it wouldn't hurt to emphasize the importance of readable code for projects that extend over time or space (e.g. number of developers). Too much cleverness in the parser can turn against you.

[2] or, just start at http://www.amk.ca/python/

What lessons do you feel Perl can teach Python programmers (both as a language and as an environment)?

Sometimes, speed matters. Regular expressions are handy. String interpolation (the ability to substitute variables into strings) is useful. CPAN rules.

4 Thoughts about Python

What lessons has Python taught you about the design of a programming language?

Don't be afraid to follow your intuition. At least, that worked for me.

What use of Python has made you most proud to be recognized as the Python's creator?

That's a tie between two opposite ends of the spectrum: at one end, the existence of a very large system written in Python, Zope, which now supports a company with a growing number of consulting customers while the software itself is open source; and at the other end, the Python programs written by middle and high schoolers, especially girls—a very underrepresented group in our geek world.

5 Resources

The Python Homepage – `http://www.python.org/`

Jython, the Java implementation of Python – `http://www.jython.org`

IDLE, an Integrated DeveLopment Environment for Python – `http://www.python.org/idle/`

Squeak, a cross-platform Smalltalk implementation – `http://www.squeak.org`

DrScheme, a beginner-friendly environment for the Scheme programming language; part of the PLT Scheme project – `http://www.plt-scheme.org/`

The Python Software Foundation: A Primer

David Ascher

Abstract

The Python Software Foundation is the youngest cousin of the open source foundations such as the Apache Software Foundation and the Perl Foundation. In this article, I present the PSF—who its ancestors were, why it was created, what it is today, and what we hope to accomplish with it.

1 Python's History

Python, like a large number of open source projects created in the early 1990s, has a fairly complex history. While Guido van Rossum has an unchallenged position as "creator and principal developer" for the language, Guido did most of the work on Python while being employed by a series of organizations with very different interests in and approaches to Python. Guido came up with the first version of Python while working for CWI, a research institute in the Netherlands. CWI therefore owned the original copyright to the Python language, name, *etc.*, collectively referred to as the Python Intellectual Property. Python's original license was a fairly "loose" license—a copy of the MIT X11 license at the time, it granted Python users pretty much every right except the right to claim Python as their own invention. Much later, CWI would grant Guido some rights to the portions of Python developed while he was a CWI employee.

In 1995, Guido moved to Virginia in the United States to work for the Corporation for National Research Initiatives (CNRI), a mostly government-funded non-profit corporation chartered with developing Internet infrastructure. CNRI viewed Python as a key technology that they wanted to use for their research and hired several other key Python developers, providing key financial support for Python's development. One of the developers hired in 1997 was Jim Hugunin, who had started the JPython port of Python to the Java platform. Thanks to CNRI's support, JPython went from being a rough prototype to a successful compatible port of Python. After Jim left CNRI for Xerox PARC, Barry Warsaw took over the work on JPython.

The CNRI years coincided with the massive explosion of interest in Python around the world. CNRI hosted the python.org website, owned the CVS source code repository for Python (behind their firewall), and took over the management of the Python mailing list from CWI (although CWI still runs a gateway to Usenet). Part of this explosion of interest led to the desire to form a "user group," with the goal of both building a more formalized community as well as a means to gather financial support for Python development. Two such user organizations were started—the Python Software Association, and the Python Software Consortium. The PSA was a mostly informal organization (http://www.python.org/psa/FAQ.html), consisting of individuals who gave nomimal membership fees ($25/year for students, $50/year otherwise), and "cheap" corporate memberships ($500 or more). Some of the early Python conferences also provided some income. The PSA's funds were used to partially defray the costs of running the website and the Python conferences. While a nice, friendly organization, the PSA did not really aim to accomplish much, and neither could it—it had no legal identity and no bank account. The PSA also did not provide the right forum for funded development of Python. Several large organizations (Hewlett-Packard, Lawrence Livermore National Labs among others) were looking for a mechanism through which they could pay Guido and other staff to add features they needed in Python. Thus the Python Consortium (http://www.python.org/consortium/) was created in

1999, modeled on the successful W3C and X Consortia. Python Consortium members were by definition corporations, with membership fees based on the corporation's size. Python's support for Unicode strings and the development of the new regular expression library was partially funded by Hewlett-Packard, through the Python Consortium, but that model did not grow successfully.

In May 2000, Guido, Barry, Jeremy Hylton and Fred Drake left CNRI for a California startup company, BeOpen.com (although they stayed in the Washington, D.C. area). BeOpen's business plans involved financing Python's development through a group called PythonLabs, which combined the abovementioned four ex-CNRI employees and Tim Peters, a veteran of software and hardware companies Cray, Kendall Square Research, and Dragon Systems. BeOpen was an exciting opportunity for Guido and the developers, since the promise was that the dot-com's business would provide the salaries for several full-time developers devoted to nothing but Python development. With the move to BeOpen, much changed in the Python world:

- The key developers got to work on Python full-time.
- Python shifted from version 1.6 to version 2.0, with all of the technical and non-technical changes implied by such a change in an open source project.
- Python development was decentralized, with a public CVS server at SourceForge. CVS access given to a larger number of developers (about 30).
- Public bug and patch databases hosted by Sourceforge also became key parts of the Python development process.
- A formalized "Python Enhancement Proposal" process was defined, through which anyone could suggest changes to Python.
- Much legal wrangling ensued over license details.

The last issue is one which alone could take dozens of pages to recount in detail, but we will save you the pain those details represent. It can be summarized thus:

Due to Python's development history, the current Python license is a layered license, with CWI's license at the bottom, CNRI's license in the middle, BeOpen.com's license on top of that (and as we'll see, the Python Software Foundation's license on top of all the others). The current Python license is deemed compliant with the OSI definition of open source licenses, as well as compatible with the GNU General Public License (which means that one can use Python to build GPL systems, not that the terms of the GPL apply to Python). Those interested in the details of the Python license are welcome to read it at http://www.python.org/2.2/license.html.

In October of 2000, the PythonLabs team left BeOpen.com (as you will remember, that was not a good time for dot-coms), and was hired by Digital Creations (now Zope Corporation). Part of the move to Digital Creations involved the realization in many people's minds that the intellectual property state of Python could not depend on who Guido happened to be working for. Each company has a different legal staff, different business priorities, and different views on open source in general and Python in particular. The Python community at large needed a stable, long-term, solid intellectual property basis on which to continue developing the Python software toolset. Thus, as part of the PythonLabs team's employment contract with Digital Creations, it was agreed that all of the intellectual property rights from PythonLabs' Python work would go to a third party, the Python Software Foundation (which was yet to be created at the time of the contract signing).

2 The Python Software Foundation

The Python Software Foundation was officially started in February 2001, with initial support from the PythonLabs team, Digital Creations, and ActiveState—another company with a strong desire to see Python's intellectual property issues clarified. The PSF bylaws were drafted, modeled on the Apache Software Foundation, which is a widely recognized, successful foundation responsible for the Apache HTTP server as well as a variety of other open source projects.

2.1 Bylaws

The bylaws of the foundation are rather boring reading material, so only the truly dedicated will take the effort to read them—but if you are one of those, feel free: http://www.python.org/psf/bylaws.html.

2.2 Membership

The membership currently consists of thirty-five Nominated Members, individuals identified by the founding members as being significant members of the Python community. Most have CVS access, or have otherwise made significant contributions to Python (important non-core modules, books, advocacy, etc.). There are also currently six corporate Sponsor Members. These are corporations who were invited by the membership and contribute a $2000 yearly fee. Sponsor Members and Nominated Members each get a single vote on any matter requiring membership votes. Members vote to select new members, and may serve on PSF committees. Once the PSF gets going, it is the committees who will have the real day-to-day power in the PSF (because they will do the real day-to-day work).

2.3 Board of Directors, Officers

The Board of Directors represents the membership and is in charge of day-to-day governance of the organization. Each board member is elected by the membership periodically. The board currently consists of myself, Jeremy Hylton, Marc-André Lemburg, Martin von Löwis, Tim Peters, Guido van Rossum and Thomas Wouters. The board appoints officers, who are charged with various administrative duties. Jeremy Hylton is Secretary and Treasurer, and Guido van Rossum is President and Chairman.

2.4 Mission Statement of the PSF

The Python Software Foundation (PSF) is a non-profit membership organization devoted to advancing open source technology related to the Python programming language. It intends to qualify under the US Internal Revenue Code as a tax-exempt 501(c)(3) scientific and educational public charity, and will conduct its business according to the rules for such organizations.

By "open source" we mean freely available technology licensed under terms compatible with Version 1.9 (or later) of the Open Source Definition established by the Open Source Initiative (http://www.opensource.org/).

The PSF:

- Produces the core Python distribution, made available to the public free of charge. This includes the Python language itself, its standard libraries and documentation, installers, source code, educational materials, and assorted tools and applications.
- Establishes PSF licenses, ensuring the rights of the public to freely obtain, use, redistribute, and modify intellectual property held by the PSF.
- Works with the Open Source Initiative to ensure that PSF licenses conform to the Open Source Definition.
- Holds Python's intellectual property rights for releases 2.1 and following.
- Seeks to obtain the intellectual property rights for Python releases prior to 2.1, for relicensing under the PSF Python license, to relieve the legal burden on Python's users. The PSF may also seek rights to other Python-related software for relicensing under a PSF license.
- Protects the Python name, and the names, service marks and trademarks associated with all other intellectual property held by the PSF.
- Solicits and manages contributions to the Python codebase, and may perform these services on behalf of other open source Python-related codebases.
- Raises funds to support PSF programs and services. The regulations for public charity funding are complex. Some consequences are that the vast bulk of funding must come from private contributions (including sponsoring memberships) and public grants, must come from a broad base, and that no single private donor can supply a substantial percentage of the PSF's total funding. Additional revenue may be pursued in ways consistent with then-current rules for public charities and with Python's standing as an open source project. For example, the PSF may offer to sell conference proceedings, special Python distributions, or merchandise with distinctive insignia.
- Publicizes, promotes the adoption of, and facilitates the ongoing development of Python-related technology and educational resources. This includes, but is not limited to, maintaining a public web site, planning Python conferences, and offering grants to Python-related open source projects.
- Encourages and facilitates Python-related research in the public interest.

3 Frequently Asked Questions about the PSF

Why did the PSF board pick that specific mission statement, and why does it matter?

If (or, thinking optimistically, when) the PSF gets charity status, we will be limited in our actions to the scope defined by the mission statement and bylaws. Thus, the mission statement is as inclusive as possible while remaining "on focus" and appropriate to a non-profit charitable organization dedicated to Python's future.

Why yet another software foundation? Why not just be part of the ASF or YAS?

Representatives of both the Apache Software Foundation and Yet Another Society (the organization behind YAPC and the Perl Foundation) have generously offered to host Python within their organizations. For better or worse, we, the PSF founders, decided against it. One key reason for going it alone is that we

wanted to have sponsor members. The core purpose of the PSF is to be an organization with a mission focused on Python, not on anything else. While we are likely to share a great deal with the goals of the ASF and YAS, the overlap is not 100%, and as such, it was deemed worth the extra headache of reinventing this particular wheel. It is not unlike the decisions for doing Unicode handling differently in Python than in any other comparable language—it had to be appropriate to the rest of Python, even if it was more work than "borrowing" someone else's code. We are in touch with folks at those foundations, and do exchange information on a regular basis.

Why does the PSF have sponsor members unlike ASF and YAS?

Corporations have always had a seat at the table when it came to shaping important decisions about Python. The PSA had corporate members, the Python Consortium was an organization made up only of corporate members. Furthermore, sponsor members were seen at the foundation of the PSF as an important mechanism by which regular, predictable income for the foundation could be established. The yearly membership fee from self-selected corporate sponsors with an explicitly stated interest in Python's well-being and future seemed a good match. The model of sponsor members, however, has led to some accounting challenges. As the board has learned (after painful dredging through accounting books on non-profit accounting rules), one of the key requirements for getting charity status from the IRS is that the PSF must show evidence of broad public support. The precise rules are beyond the interest of most sane readers, but the short version of it is that financial support must come from a large number of parties and be relatively evenly distributed. Having just the sponsor members as the source of financing becomes a problem.

Given that it is so much work, why bother getting non-profit status?

There are a variety of reasons to seek non-profit status that apply to all similar organizations (donations are tax-deductible, tax laws are kinder, people are more willing to donate to a "designated good cause"). In addition, the PSF has a very specific motivation, due to the fact that public charities are prohibited by law from transfering their assets to anything but a like charity. This means that CNRI, which holds the intellectual property to several versions of Python, cannot even consider transfering those rights to the PSF until we are granted charity status.

How are you going to resolve the problem of showing broad public support given the issue with sponsor members?

Getting financial support from a few corporate members does not in itself pose a problem. Only if that is the only source of income and if there are relatively few of these donors is that a problem. The solution is therefore to establish a mechanism by which more people can donate more money. Luckily, we know that we can, provided we can figure out the accounting. The PSA received a continuous flow of membership checks, even though the PSA had no legal status at all, and the only thing members got in exchange was a listing on the website.

Sounds good! How can I help?

If you are a member, read the PSF-members mailing list and participate. The board wants all the input from the membership it can get in order to represent it as best as it can. When committees are formed, such as the "Public Support Committee", participate!

If you want to help financially, get in touch with the board (psf-board@python.org), and we will let you know when and how we are ready to accept funds.

If you are an accountant, a lawyer, or know something about running a non-profit and want to provide free advice, get in touch with the board (psf-board@python.org) and let us know.

How do I become a member?

Work hard on things Pythonic. Take care of bugs, patches, releases, newbies. Write library modules, test code, documentation, docstrings. Establish credibility with existing members. Learn about non-profit accounting rules or go to law school (kidding!).

Where can I learn more?

The PSF has its own section on the python.org website (http://www.python.org/psf). This includes the bylaws, mission statement, records of meetings, listings of members, directors and officers, and any other information we felt appropriate to put up there. A version of this article will find its way soon after this journal goes live.

4 Current Status and Future Plans

The PSF is incorporated, and is catching up on the paperwork that is involved in both being a corporation and filing for 501(c)(3) status with the IRS. We have gotten some of the sponsor fees and are expecting more in the coming weeks, which will go towards the administrative fees accrued and pending, as well as towards getting legal and accounting advice. The legal advice is particularly important with respect to the PSF contributor forms, which will go a long way towards establishing a strong claim of ownership over the Python intellectual property. The accounting advice will be particularly important to ensure that our accounting practices will help, not hinder, our non-profit status.

The emphasis of the PSF board at this point is squarely on making sure that we can secure the intellectual property which makes up Python—and that starts with getting 501(c)(3) status, since CNRI, which owns a good chunk of Python IP, will not be legally able to transfer that IP to the PSF until that happens. On a day-to-day basis, much of what is involved in that medium-term goal makes a great deal of sense both in the short term and in the long term. Getting our financial books in order, establishing a routine for the board and the membership, reaching out and establishing broad public support—these are all goals that we have anyway—but the IRS is helping us prioritize them.

The challenging issues the board is facing right now involve the mismatch between our skills and experience (mostly doing things in the mostly logical software world) and the immediate goals for the corporation—accounting, legal, and administrative details. We are learning more about Rules of Order and accounting than we had expected. Luckily, we have not had to worry too much about purely legal issues to date—although that will come, I am sure. Setting up a non-profit foundation is turning out to be a lot more work than we ever expected, but then again, that is not altogether different from any other software project, now is it?

Some of the readers probably remember "Computer Programming for Everybody" (CP4E)—a research and educational effort aimed at making Python the ideal programming language for teaching programming. The PSF could very well solicit government grants to pursue goals not unlike CP4E—and those funds would in fact help the accounting issues, since government grants are by definition evidence of public support. Other long-term goals include more educational/promotional work, possibly funding research work, etc.

5 Acknowledgements

Thanks to Greg Stein, Tim Peters, and Guido van Rossum for their feedback on a draft of this document.

6 About the Author

David Ascher is a Director of the Python Software Foundation. He is currently employed by ActiveState and works on development tools.

7 Resources

The Python Homepage – http://www.python.org

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

Computer Programming for Everybody (CP4E) – http://www.python.org/cp4e

The Apache Software Foundation – http://www.apache.org/foundation

Yet Another Society – http://www.yetanother.org/

The Perl Foundation – http://www.perl-foundation.org/

Tiger Woods isn't Worried, and Neither is abigail: A duffer's first try at (perl) golf

Mike Giroux, rmgiroux@acm.org

Abstract

In March 2002, The Perl Review held its Second Perl Golf tournament, TPR(0,1). I decided to try my hand at the sport, and I chronicle my first attempt in this article. I describe each step I took, including the ones which are so verbose now that I have seen the leaders' solutions, including Ton Hospel's amazing winning entry. Even though I did not win, or even come close, I learned quite a bit about Perl.

1 The challenge

The Perl Golf website describes the rules. In this challenge, the program must distill an integer down to its "secret number" by taking every consecutive pair of integers and adding them together to form a new number. The program continues until it reduces the number to a single digit.

For example, the program should reduce 7654 to 429 like this:

```
7+6 => 13, 1+3 => 4
6+5 => 11, 1+1 => 2
5+4 =>  9
```

The program repeats the process until a single digit number remains. At each iteration, the value is printed to standard output. The program reads from the command line the integer it should reduce, which is the only argument.

2 My solutions

I came up with my first, verbose version of the script to make sure I had a working approach. I added comments to make this clearer; otherwise, it is the code I started with.

Code Listing 1: The verbose version

```
#!/usr/bin/perl

$_=shift;               # place the command line argument into $_
while($_>9){            # while we have more digits to process
    $pd=10;             # initialize "previous digit" to "none"
    print "$_\n";       # print the current number
    while(/(\d)/g){     # for each digit in $_
        if($pd>9){      # if we don't have a previous digit,
            $pd=$1;     # save this one and then
            next        # restart the loop
        }
        $nd=$pd+$1;     # "next" is "previous" plus the current one
        $pd=$1;         # save current digit as next "prev digit"
        if($nd>9){      # if next digit is greater than 9
            $nd=~/(\d)(\d)/ or die;
            $nd=$1+$2;  # repeat the addition
        }
        $nn.=$nd;       # concatenate next digit onto "new number"
    }
    $_=$nn;             # update $_ with new number
    $nn='';
}
print "$_\n";           # print the final answer
```

The first try illustrates the main parts of the problem.

1. The program has to extract the command line argument from @ARGV, because $ARGV[0] is too verbose for a golf challenge.
2. The program has to loop until the solution is complete, and print the result at each iteration
3. The program has to deal with the digits two by two, but `/(.)(.)/g` will not work since the second character of the first pair is the first character of the second pair. Given 1234, `/(.)(.)/g` will match 12 then 34, while we need 12, 23, and 34.
4. The program has to convert each pair in `(00..99)` to its corresponding secret digit between 0 and 9. For some pairs, there will be an intermediate step between 10 and 18.

3 Shaving strokes

Initially, I took the worst possible approach to all four of those problems, unfortunately. But the tpr01.pl test program reported that it passed all tests, scoring it at 368 strokes. I removed all whitespace, except newlines, and shrank all the two character variable names down to one character. This program still works, and I scored a much-improved 184 strokes.

Code Listing 2: Removing whitespace

```
#!/usr/bin/perl

$_=shift;
$p=10;
while($_>9){
print"$_\n";
while(/(\d)/g){
if($p>9){
$p=$1;
next
}
$x=$p+$1;
$p=$1;
if($x>9){
$x=~/(\d)(\d)/;
$x=$1+$2;
}
$n.=$x;
}
$_=$n;
$n='';
$p=10;
}
print"$_\n";
```

However, the scoring program counts newline characters in the score, so from here on, I write everything as one-liners. I had to break the lines for printing, but I submitted each of these scripts as a one-liner and I do not count the newlines in my score.

Code Listing 3: Removing newlines -- 162 strokes

```
#!/usr/bin/perl
$_=shift;$p=10;while($_>9){print"$_\n";while(/(\d)/g){if($p>9){$p=$1;next}$x=$p+$1;$p=$1;
if($x>9){$x=~/(\d)(\d)/;$x=$1+$2;}$n.=$x;}$_=$n;$n='';$p=10;}print"$_\n";
```

This is still the same approach as the commented version, essentially. I scored 162 strokes with this version, but still was not competitive, so I knew I had to find some way to cut down on the size of the program. My next idea was to try to find repeated strings, put them in variables, and use the string form of the eval function to expand the variables.

Code Listing 4: Remove repeated variables -- 206 strokes

```
#!/usr/bin/perl
$w='while(';$p='print"*_\n";';$v='*p=10;';$i='if(';$d='(\d)';$a="*_=shift;$v$w*_>9)
{$p$w/$d/g){$i*p>9){*p=*1;next}*x=*p+*1;*p=*1;$i*x>9){*x=~/$d$d/;*x=*1+*2;}*n.=*x;}*_=*n;
*n='';$v}$p";$a=~s/\*/\$/g;eval$a;
```

Fortunately for my sanity, that turned out to be worse, and yielded 206 strokes. I had two problems with this approach. I had to replace all the $ symbols in the string to be eval'ed with another symbol, so they

would not be prematurely evaluated during the double quoted interpolation at the `$a=""` assignment. Then I had to do a global substitution to replace those symbols with "$". And since I moved common operations out of line into the substitution variables, maintaining and improving this version would have been difficult. I found that the overhead required for this tack was too high for a low-par golf course like this one. I might have to revisit this idea in the future for longer problems.

By doing the "compression attempt" in code listing 4, I realized I was setting `$p=10` twice in my code, so I went back to the 162 stroke version and corrected that problem, which cut my score down to 156.

Code Listing 5: Removing repitition -- 156 strokes

```
#!/usr/bin/perl
$_=shift;while($_>9){$p=10;print"$_\n";while(/(\d)/g){if($p>9){$p=$1;next}$x=$p+$1;$p=$1;
if($x>9){$x=~/(\d)(\d)/;$x=$1+$2;}$n.=$x;}$_=$n;$n='';}print"$_\n";
```

At this point, I read the postmortem for *The Perl Review*'s first golf tournament and after looking at Ton Hospel's winning entry, I realized that I could use the pop function just as easily as shift to get the command line argument, saving two strokes, and that by using the -l switch, I could save two newlines, saving one stroke net. I was now at 134 strokes.

Code Listing 6: Using pop -- 134 strokes

```
#!/usr/bin/perl -l
$_=pop;while($_>9){$p=10;print"$_";while(/(\d)/g){if($p>9){$p=$1;next}$x=$p+$1;$p=$1;$x-=9
if($x>9);$n.=$x;}$_=$n;$n='';}print"$_";
```

Of course, I forgot that $_ is the default argument to print, and that I did not need the double quotes. I corrected that and removed some extra semicolons to get me down to 118 strokes.

Code Listing 7: Default arguments -- 118 strokes

```
#!/usr/bin/perl -l
$_=pop;while($_>9){$p=10;print;while(/(\d)/g){if($p<=9){$x=$p+$1;$p=$1;$x-=9if($x>9);$n.=$x}
$p=$1}$_=$n;$n=''}print
```

At this point, I could not squeeze my solution any farther. I needed a better approach to one of the four problems. I decided to focus on the problem of getting the pairs of digits.

I tried using `/(.)(?=.)/g`, the lookahead assertion, to match a second digit but to not consume it. That worked, but unfortunately, I found out that `(?=)` are not capturing parentheses, so I had to use `/(.)(?=(.))/`. I also figured out that I could sum the new digits with the ternary operator, `($1+$2)>9?($1+$2):($1+$2-9)`. I precomputed `$x=$1+$2`, so that became `$x>9?$x:$x-9`.

Code Listing 8: Ternary operator -- 87 strokes

```
#!/usr/bin/perl -l
$_=pop;while($_>9){print;s/(.)(?=(.))/$x=$1+$2;$n.=$x>9?$x-9:$x/eg;$_=$n;$n=''}print
```

That got me down to 87 strokes. Switching to `$x-9*($x>9)` saved a stroke, and got me to 86.

Code Listing 9: No ternary operator -- 86 strokes

```
#!/usr/bin/perl -l
$_=pop;while($_>9){print;s/(.)(?=(.))/$x=$1+$2;$n.=$x-9*($x>9)/eg;$_=$n;$n=''}print
```

I had another major breakthrough when I realized how stupid I was to build up $n all the time, when the substitution already modified $_ almost exactly the way I wanted. However, the substitution left the last digit unchanged, but a simple chop corrected that. Now I was down to 75 strokes.

Code Listing 10: Losing $n -- 75 strokes

```
#!/usr/bin/perl -l
$_=pop;while($_>9){print;s/(.)(?=(.))/$x=$1+$2,$x-9*($x>9)/eg;chop}print
```

I switched to a "do while" instead of the while loop which allowed me to get rid of the `>9` and one of the print statements. I saved two more characters by getting rid of the () around the while condition, shrinking the code to 68 strokes.

Code Listing 11: do {} while -- 68 strokes

```
#!/usr/bin/perl -l
$_=pop;do{print;s/(.)(?=(.))/$x=$1+$2,$x-9*($x>9)/eg;chop}while$_
```

I realized that now that I was not using the ternary operator, $x cost me more than it gained, so I ditched it. That got me to 65 strokes.

Code Listing 12: Dropping the ternary operator -- 65 strokes

```
#!/usr/bin/perl -l
$_=pop;do{print;s/(.)(?=(.))/$1+$2-9*($1+$2>9)/eg;chop}while$_
```

From here, I tried all kinds of things to improve my solution, but this is the best answer I came up with, placing 18 strokes behind the winner and I am pretty sure this means I missed the cut. But I did try a couple of other avenues which were interesting.

I thought of precomputing the answers to the digit reduction. However, that only got 88 strokes. After the contest ended, by looking at one of the other solutions I realized I could have just assigned `@a=(0..9,1..9)` but that would only have gotten me to 72 strokes.

Code Listing 13: Precomputation -- 88 strokes

```
#!/usr/bin/perl -l
@a=map{int($_%10+$_/10)}0..18;
$_=pop;do{print;s!(.)(?=(.))!$a[$1+$2]!eg;chop}while$_
```

I figured out that the calculation could be expressed as `($1+$2-1)%9+1` for all `($1,$2)` except (0,0). That only scored 68.

Code Listing 14: Another way to sum -- 68 strokes

```
#!/usr/bin/perl -l
$_=pop;do{print;s/(.)(?=(.))/$1+$2&&($1+$2-1)%9+1/eg;chop}while$_
```

I thought of using a subroutine for `$1+$2`. I could only shrink that down to 71, though.

Code Listing 15: Using a subroutine -- 71 strokes

```
#!/usr/bin/perl -l
sub x{$1+$2}$_=pop;do{print;s/(.)(?=(.))/&x-9*(&x>9)/eg;chop}while$_
```

Next, I tried attacking the do-while loop, by rerunning the program with the exec function. I could only get this to 68 strokes, and it would not have worked in the tournament environment since the scripts are run with mode 0644, which means that I would have had to do something like `exec$^X,$0,$_`, which is even worse.

Code Listing 16: Using exec -- 68 strokes

```
#!/usr/bin/perl -l
$_=pop;print;s/(.)(?=(.))/$1+$2-9*($1+$2>9)/eg;chop;$_&&exec$0,$_
```

I thought that maybe it was a *lack* of goto's that was harmful in this case. This was 67 strokes.

Code Listing 17: Using goto -- 67 strokes

```
#!/usr/bin/perl -l
$_=pop;a:print;s/(.)(?=(.))/$1+$2-9*($1+$2>9)/eg;chop;$_&&goto a
```

Once the contest ended and I saw the other solutions, the biggest and most obvious improvement I could make to my script was to get rid of the first capturing parentheses, and use $&. That would have gotten me to 63 strokes.

Code Listing 18: Using $& -- 63 strokes

```
#!/usr/bin/perl -l
$_=pop;do{print;s/.(?=(.))/$1+$&-9*($1+$&>9)/eg;chop}while$_
```

I had a great deal of fun competing in the tournament. My experience of avoiding default arguments for clarity's sake in my scripting was a handicap in golf, and it was interesting to look at every cranny of the program trying to find one more character to squeeze.

I hope you will join me in the next round! Next time, I'm going to be brave, or foolhardy, and compete with the big kids in the expert ladder.

4 References

Perl Golf website – http://perlgolf.sourceforge.net

TPR(0,0) overview – http://www.theperlreview.com/Issues/The_Perl_Review_0_1.pdf

Background about the history of perl golf – http://archive.develooper.com/fwp@perl.org/msg01949.html

The post where the name appears to have originated – Greg Bacon, Message-ID: 7imntimjh1@info2.uah.edu

5 About the author

Mike Giroux is a programmer originally from Quebec, now living in Connecticut. If you think the perl golf was ugly, you should see him try to play *real* golf.

Python and the Golden Mean

Paul Prescod

Abstract

Python strikes a balance between competing design ideas to combine elegance, expressiveness, succinctness, power, and simplicity. I discuss some interesting features of Python, including multiple assignment, list comprehensions, and generators, which demonstrate these design decisions.

1 A Little Fibbing

The Fibonacci series is a mystical and fascinating corner of the numeric world. Each successive number is the sum of the previous two numbers. The quotient of successive numbers in the series has an interesting property I show later. The first few numbers in the Fibonacci series

1 1 2 3 5 8 13 21 34 ...

I can generate the first 25 numbers in the Fibonacci series with the short Python program in code listing 1. In the example I assign two things at once, a neat Python trick called "multiple assignment". I can even assign to a variable that I use on the right-hand side. Python sets up temporary variables under the covers so this works correctly.

Code Listing 1: Fibonacci number generator

```
thisnum, nextnum = 0, 1   # set up two variables
for i in range(25):       # loop 25 times
   print thisnum, nextnum # print two adjacent Fibonaccis
   thisnum, nextnum = nextnum, thisnum + nextnum
```

To me, that code is the clearest possible expression of the series with a minimum of noise characters, indentation to support the logic of the algorithm, and no explicit temporary variables. I can increase the loop bound to 1000 or 10000 or 100000 and Python does not complain. Python does not limit the size of integers.[1]

My 800 MHz Pentium III takes only a couple of seconds to generate 100,000 Fibonacci numbers. Listing 2 shows an interactive Python session that computes the 100,000th Fibonacci number and counts the number of digits in it.

[1] As of Python 2.2. To get similar behaviour in older versions of Python, add an "L" to the 1 in the first line: `thisnum, nextnum = 0, 1L`. Older versions of Python required the programmer to specifically ask it to use arbitrarily sized integers.

Code Listing 2: The length of the 100,000th Fibonacci number

```
prompt% python

>>> for i in range(100000):
...     thisnum, nextnum = nextnum, thisnum + nextnum
...
>>> print len(str(thisnum))
20899
```

Some hints of Python's design balance shows here. A purely minimalist language would not include the multiple assignment syntactic sugar. But Python includes it because it is simple to understand, clear to read, and useful in practice.

2 List Comprehensions

Most programs consist of operations on data structures and one of the most common structures is the list. Scripting languages tend to make this very easy to work with.

To demonstrate how far these languages have come, consider the idiomatic Java program in code listing 3. It loops over the characters in a string to generate a variable sized list.

Code Listing 3: Variable-sized arrays in Java

```
    static Vector chars_to_ordinals(char [] letters){
        Vector out = new Vector();
        char achar;
        for(int i=0; i<letters.length; i++){
            achar = letters[i];
            if(!Character.isWhitespace(achar)){
               out.add(new Integer(achar));
            }
        }
        return out;
    }
```

It seems as if I should be able to use a statically-typed Java array but since I do not know the size of the list I am creating in advance it gets a little bit tricky. Of course any scripting language could do a better job than this. Unoptimized Ruby or Perl equivalents would not be very different from the Python 1.5.2-era code in listing 4:

Code Listing 4: Variable-sized arrays in Python

```
def chars_to_ordinals(letters):
    ordinals = []                  # create the empty list
    for character in letters:      # loop over characters
       if not character.isspace(): # check character is not whitespace
           ordinals.append(ord(character)) # if OK, add to list
    return ordinals
```

In version 2.0, Python added an interesting feature that allows us to express this as a single expression, called a *list comprehension*, which allows conditional construction of list literals using for and if clauses. A list comprehension has the following basic syntax:

```
[ EXPRESSION for INDEX in LIST if CONDITION ]
```

The list returned by this expression is a collection of values (EXPRESSION) for each element in LIST that satisfies the CONDITION. In code listing 5, the EXPRESSION is ord(character). The INDEX, character, is used to evaluate the CONDITION, character.isspace(), for each element in the LIST represented by letters.

In code listing 5, the first thing in the list comprehension is an expression describing how to process each item, which can be arbitrarily complex. In this case I get the ASCII ordinal value for each character. Next I have a for loop that iterates over the items to be processed. In this case I am looping over the set of letters. I use the `character` variable to represent each character. I could call it whatever I like. The list comprehension packages all of the values into a single list and then the function returns it.

Code Listing 5: Using a list comprehension

```
1  def chars_to_ordinals(letters):
2      return [ord(character)
3              for character in letters
4                  if not character.isspace()]
```

I can even combine multiple lists in a single list comprehension. Code listing 6 shows an interactive session:

Code Listing 6: Multiple lists in comprehensions

```
1  prompt% python
2
3  >>> import string
4  >>> vowels = "aeiou"
5  >>> consonants = [letter for letter in string.ascii_lowercase
6  ...                  if letter not in vowels]
7  >>> print [cons + vowel
8  ...           for cons in consonants
9  ...              for vowel in vowels
10 ...                  if cons!="q" or vowel=="u"]
11 ['ba', 'be', 'bi', 'bo', 'bu', 'ca', 'ce', 'ci', 'co', 'cu', 'da',
12 'de', 'di', 'do', 'du', 'fa', 'fe', 'fi', 'fo', 'fu', 'ga', 'ge',
13 'gi', 'go', 'gu', 'ha', 'he', 'hi', 'ho', 'hu', 'ja', 'je', 'ji',
14 'jo', 'ju', 'ka', 'ke', 'ki', 'ko', 'ku', 'la', 'le', 'li', 'lo',
15 'lu', 'ma', 'me', 'mi', 'mo', 'mu', 'na', 'ne', 'ni', 'no', 'nu',
16 'pa', 'pe', 'pi', 'po', 'pu', 'qu', 'ra', 're', 'ri', 'ro', 'ru',
17 'sa', 'se', 'si', 'so', 'su', 'ta', 'te', 'ti', 'to', 'tu', 'va',
18 've', 'vi', 'vo', 'vu', 'wa', 'we', 'wi', 'wo', 'wu', 'xa', 'xe',
19 'xi', 'xo', 'xu', 'ya', 'ye', 'yi', 'yo', 'yu', 'za', 'ze', 'zi',
20 'zo', 'zu']
```

If I want to loop over multiple lists in a single iteration, Python has a `zip` function that does that. In code listing 7, I pair lowercase letters with their uppercase counterparts.

Code Listing 7: Multiple lists simultaneously

```
prompt% python

>>> import string
>>> print [lowercase +" => " + uppercase
...               for lowercase, uppercase in
...               zip(string.ascii_lowercase, string.ascii_uppercase)]
['a => A', 'b => B', 'c => C', 'd => D', 'e => E', 'f => F',
'g => G', 'h => H', 'i => I', 'j => J', 'k => K', 'l => L',
'm => M', 'n => N', 'o => O', 'p => P', 'q => Q', 'r => R',
's => S', 't => T', 'u => U', 'v => V', 'w => W', 'x => X',
'y => Y', 'z => Z']
```

List comprehensions look like declarative expressions instead of loops. I see them as expressions of *what* I want to do—build a list with the following properties—rather than *how* I do it: "start with the empty list, then add elements from this other list". Python borrowed this elegant feature primarily from the language Haskell, but a Google search suggests that it is already strongly associated with Python.

In code listing 8, I implement a quicksort that uses list comprehensions to filter the high and low elements. This is not the most efficient way to write quicksort, and Python already has a sort method built into its list objects, so the code is illustrative more than useful.

Code Listing 8: Quick sort

```
def qsort(L):
    if len(L) <= 1: return L
    pivot   = L[0]
    lesser  = [ lt for lt in L if lt < pivot  ]
    greater = [ gt for gt in L if gt > pivot  ]
    same    = [ eq for eq in L if eq == pivot ]
    lesser  = qsort(lesser)
    greater = qsort(greater)
    return lesser + same + greater
```

If I change `for`, `in`, and `if` into mathematical symbols and list comprehensions look like set descriptors in math textbooks, and my code in listing 8 looks similar to the pseudocode in listing 9[2].

Code Listing 9: Quick sort pseudocode

```
    quicksort(L) {
        if (length(L) < 2) return L
        else {
            pick some x in L
            L1 = { y in L : y < x }
            L2 = { y in L : y > x }
            L3 = { y in L : y = x }
            quicksort(L1)
            quicksort(L2)
            return concatenation of L1, L3, and L2
        } }
```

[2]http://www.ics.uci.edu/ eppstein/161/960118.html

Granted, other languages have map, filter, or lambda functions that do similar things. Python has those also, but I do not use them if I can use a list comprehension. Given the choice of writing code that resembles elisp, or writing code that more closely resembles an algorithms book, I choose the latter.

Once again we can see how Python combines readability and succinctness, power and elegance. The Python developers rejected many potential enhancements to the list comprehension syntax which went too far in emphasizing conciseness over simplicity or scalability over readability. They developed Python not to be the most scalable or simple or concise or readable language but a balance of those factors.

3 Generators

Python stole features from not only Haskell—it shamelessly ripped off other languages also. The concepts of iterators and generators probably found their zenith in string processing languages like Icon, and they integrate nicely into Python.

In code listing 10, I rework my Fibonacci series program to collect the Fibonacci values into a list for use later in the program.

Code Listing 10: Collect values in a list

```
fibs = []
thisnum, nextnum = 0, 1
for i in range(100000):
   fibs.append(thisnum)
   thisnum, nextnum = nextnum, thisnum+nextnum
print fibs
```

The problem with my list is that it gets very large for later values in the series, and the list itself will have many items. In that case I might like to make an object that pretends to be a list, but really generates each number only when it is needed.

The Fibonacci algorithm depends on the previous `nextnum` and `thisnum` to generate the next values on subsequent calls. I want to do something similar to the code in listing 11 without creating the entire list at one time.

Code Listing 11: Generate 100 Fibonacci numbers

```
for thisnum, nextnum in fib_gen(100000):
    print thisnum, nextnum
```

Python knows the fib_gen2 function in code listing 12 is special because it contains a yield keyword which turns the function into a "generator function". I can call it in a loop context and get more than one value out of it. Every time the loop asks for a value, Python jumps back into the middle of the function to the place where it yielded and runs until the function is ready to yield another value. When the generator function runs out of values it ends, just like a regular function. Python informs the for loop that it may quit looping and the program continues.

Code Listing 12: Yielding values

```
def fib_gen2(stop_val):
    thisnum, nextnum = 1, 2
    for i in range(stop_val):
       yield thisnum, nextnum   # was "print" before. now it is "yield"
       thisnum, nextnum = nextnum, thisnum+nextnum

for thisnum, nextnum in fib_gen2(100):
     print thisnum, nextnum
```

I code the generator function as if it were just printing out values without worrying about how the values will be used, then I code the part of the program that will use the values as if it were just iterating over a list. Python hides from each of them the fact that something extraordinary is going on.

While the generator is yielding a value to the calling code, its local variables stay alive. Then, when the looping code needs another value, it jumps right back into the middle of the generator function with all of the local variable values right where they were left. The programmer does not have to list which variables should be kept and somehow jump into the middle of the algorithm. Python does that all for you.

The generator can also be used as an object so that it may be passed from one part of the Python program to another part without losing the state of the computation. When I call a generator function it returns a generator object instead of a regular value. Generator objects have `next` methods which Python creates automatically. In code listing 13 I use the next() method explicitly.

Code Listing 13: Using the generator object

```
fib = fib_gen2(100)    # get a generator object
for i in range(50)     # use 50 values:
     print fib.next()
print "Half done!"
for i in range(50):    # use the next 50
     print fib.next()
print "Done!"
```

Generators are related to, but not identical to, the concurrency feature known as “coroutines”. The Python variant “Stackless Python” has generalization of this feature and supports generators, coroutines, and even more obscure control flow mechanisms such as cooperative micro-threads and continuations. Yes, you Lisp-hackers read that correctly: continuations. Be afraid. Be very afraid.

4 The Golden Mean

The things I have shown can be either done directly or approximated in other languages, especially Ruby, which is the most recently invented of the quasi-mainstream scripting languages. Each language has unique strengths: Perl sets the bar for hackability. Ruby has innovative object-oriented and control flow features. PHP is very easy to learn and use. Java is becoming the leading language for large-scale applications.

I see Python as being uniquely guided by the philosophy of Aristotle, who postulated that every virtue is a mid-point between two extremes—“Moderation in all things.” Between indecisiveness and impulsiveness is

considered action. Between selfishness and selflessness there is cooperation.

Aristotle called the middle ground between two extremes the *Golden Mean*. The Python 2.2 interpreter can, on demand, illustrate the application of the Golden Mean to Python's design. It does so through a sort of poem written by Tim Peters and hidden like an Easter Egg under the module `this`.

Code Listing 14: The *this* module

```
prompt% python

>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Python has many competing design principles and many of them merely argue against taking the others too far. I think that Aristotle would approve. Tim Peters says with a half-wink that Python is the language that gets all of its compromises just right. Object-oriented but not enough to be obssessive; Syntactically strict, but not enough to be aggressive; Backwards-compatible, but not enough to be regressive. From one-line hacks to large team projects, Python is not too hard, not too soft, and seems just right.

The Golden Mean is also a name used for the irrational number Φ[3]. You can approximate Φ by taking any two successive Fibonacci numbers and dividing the larger one by the smaller one. The larger the Fibonacci numbers you use, the closer you approximate Φ.

The Golden Mean appears in philosophy, mathematics, architecture, music, poetry, and nature[4]. Why should it not also be manifest in the Zen of a programming language?

In code listing 15, I calculate the Golden Mean with the rational number package mxNumber by Marc-Andre Lemburg of eGenix, by dividing successively larger Fibonacci numbers.

Rational numbers can only approximate the true value of the Golden Mean. Similarly Python can only approximate the ideal balance between different design constraints. Python will never be perfect but it gets

[3]Pronounced "phi"—not π, a different irrational number.

[4]http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/ fibInArt.html

a little bit better in each version. It gets simpler even as it gets more powerful through combining previously distinct types and concepts—for instance 32-bit integers and unbounded integers. It gets more concise and yet more readable by strategically choosing new syntaxes.

Code Listing 15:

```
from mx.Number import Rational, Float
for thisnum, nextnum in fib_gen2(100): # loop 100 times
    print Rational(nextnum,thisnum).format(10, 20)
```

5 References

Tutorial on iterators and generators –
http://www-106.ibm.com/developerworks/linux/library/l-pycon.html?dwzone=linux

Generators Python Enhacement Proposal – http://python.sourceforge.net/peps/pep-0255.html

Fibonacci Numbers and the Golden Section –
http://www.mcs.surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html

Quicksort tutorial – http://www.ics.uci.edu/ẽppstein/161/960118.html

MxNumber – http://www.lemburg.com/files/python/mxNumber.html

Zen of Python and other Python quotes and humour – http://www.python.org/doc/Humor.html#zen

Multiple assignment – http://diveintopython.org/odbchelper_multiassign.html

Center of the Python world – http://www.python.org

My home page – http://www.prescod.net

6 About the author

Paul Prescod is a long-time Python user and advocate. He works as an independent consultant from Vancouver, British Columbia and can be reached at paul@prescod.net.

Book Review: *Embracing Insanity*

Russell C. Pavlicek; Sams Publishing; 0-672-31989-6; 177 pages; September 2000;

reviewed by brian d foy

In *Embracing Insanity*, Russell C. Pavlicek provides a good primer for open source software advocacy, If you already know the value of open source software and want help advocating it, this book can help you do that, although you should use it more as the Cliffsnotes of open source rather than a definitive reference. Buy this book, but pick up a few others too.

I do not think the author supplies any compelling reasons to use open source though. He talks about the community of open source, addresses some open source myths, and highlights some open source service companies, but ignores the issues of economic value and any sort of metrics that compare open source to proprietary systems. I do not think this book can help you decide to use open source technology, but it can help you justify your use of it.

He focuses on "geek culture" to the extent that the term almost becomes synonymous with "open source", and that someone unfamiliar with open source would think that only geeks participate in the community. He equates acceptance of open source with the acceptance of "geek culture". As an open source developer, I found this offensive.

In the technology realm, he talks about GNU/Linux, but does not cover the breadth of open source. It certainly is not all operating systems and kernels, and even then he does not offer much guidance on selecting open source technologies.

I was surprised by a lack of historical perspective. The open source movement is not as revolutionary as he claims. Software started as open source before the Internet existed, and open source is not a new concept. You may want to also read Stephen Levy's *Hackers* for a better historical, although dated, perspective, and supplement some of the more recent events and philosophy with *Open Sources: Voices from the Open Source Revolution*. Karl Fogel better covers development issues in chapters 3 and 5 of *Open Source Development with CVS*. Eric Raymond's *The Cathedral and the Bazaar* makes a better case for open source over proprietary software.

reviewed by Andy Lester

Embracing Insanity is a good history book about open source and a poor book about how to incorporate the principles of open source into business. For a book that is clearly aimed at Pointy-Haired Bosses—a critical flaw.

Russell Pavlicek knows his way around the open source world. The history of Unix, FSF, Linux, Apache and other bits fit together nicely to give a lay of the land. Pavlicek also makes good arguments for how American business and these concepts that seem antithetical can coexist. The first three chapters would make a fine, if very slim, book on their own. After that, *Embracing Insanity* turns into a questionable combination of sociology and business advice.

Pavlicek spends far too much time exploring "geek culture", specific to his understanding of it. His discussions of the culture are given as standards, not generalizations. He devotes a page to the importance of beer in the community, and an entire chapter claims that the top value in the community is truth. Apparently Pavlicek thinks that this devotion to truth is the sole province of the geek community.

All the talk of culture is in proud, defiant contrast to the rest of the world. In the chapter on truth, Pavlicek explains unapologetically that to the geek, "rudeness can be forgiven when defending the truth." Perhaps that is okay to the geek, but far less likely the case to the world which will not coddle him. He does not discuss assimilation of cultures—only accommodation. Managers and co-workers are expected to bow down to geeks who will be their salvation.

The defiant and aggressive attitudes hit their worst in the chapter "Potholes: What To Avoid". Pavlicek takes a sneering attitude toward the PHBs he expects will be reading the book. Sections like "Forget Fluffy, Empty Management Speeches" and "Forget About Fudging Facts" assume that the reader is already a jerk who needs to be corrected.

The chapter "A Primer: What To Do" should have been the real meat of the book, but in an expanded form. High-level summaries and checklists would have helped greatly. Most lacking is any sort of real case-study of an organization that was able to successfully incorporate Open Source into their operations.

I would have been fascinated by front-line tales like "Petdance Inc., a small diaper service in McHenry, Illinois, used open source software to improve the bottom line. After moving to OpenWidget, network maintenance costs were halved, while at the same time Petdance programmers added four new modules to the OpenWidget codebase, all within 14 months." Benefits could have been listed, and problems discussed as cautionary tales. Most importantly, there would be the hard facts that managers look for. Without this most basic research, Pavlicek's arguments are merely unsupported anecdotes.

Finally, I would have appreciated a survey of open source software beyond the canonical Linux, GNU, Apache, sendmail and Perl, along with a discussion of those cases where there are no open source alternatives to commercial software. Open source is not the silver bullet to fix all business problems, and ignoring that lessens the credibility of the book.

Briefly reviewed

Publishers: to have us review your book, send us a note at book_reviews@theperlreview.com.

Mac OS X: The Missing Manual

David Pogue; Pogue Press; 0-596-00082-0; 583 pages; January 2002

Mac OS X has most of the power of Unix minus a lot of the easy to find documentation. This book fills that gap quite well. If you want to use Mac OS X so you can have a Mac OS and a Unix-like version of Perl, this book may save you a lot of pain and suffering moving from Mac Classic or Unix to Mac OS X. (brian d foy)

Physics for Game Developers

David M. Bourg; O'Reilly & Associates; 0-596-00006-5; 326 pages; January 2002

You cannot say that this is not rocket science, because it is. However, rocket science is not that complicated. If you want to include rockets or missiles in games then this book explains it to you. Even if you are not a game programmer, you can still learn how hovercraft, ballistic weapons, or billards work with an equal mix of code and math. Do not let the basic calculus and vector algebra scare you away—the explanations are clear and down-to-earth. (bdf)

Apache Administrator's Handbook

Rich Bowen; Sams Publishing; 0-672-32274-9; 326 pages; January 2002

This is another failed attempt at an Apache book. The author, and the authors brought in later to save the book, also work on the Apache Documentation project, so a lot of the book looks very familiar. They do not include much more advice or wisdom than the Apache documentation, and say so little on most subjects that the book is merely a list of features rather than a guide to using them. (bdf)

The mod_perl Developer's Cookbook

Geoffery Young, Paul Lindner, Randy Kobes; Sams Publishing; 0-672-32240-4; 650 pages; January 2002

This is a welcome addition to the mod_perl library, previously populated only by *Writing Apache Modules With Perl and C*. Most of the book is devoted to the mod_perl API and processing the various phases of the Apache request lifecycle. The authors clearly explain the recipes, most of which are useful on their own. The installation section is thorough and includes Win32 instructions. (Andy Lester)