# *Software Engineering with Python for Scientists and Engineers*

Proposal submitted to Python Software Foundation

Fall 2004

Dr. Gregory V. Wilson
43 Ryerson Avenue #2
Toronto, Ontario M5T 2P4
gvwilson@third-bit.com

**Abstract**

Many scientists and engineers spend much of their time developing software, but do not think of themselves as programmers, and have never been taught how to program efficiently. Python's simplicity and extensibility, coupled with its comprehensive library, make it an ideal language for such users. However, most scientists and engineers are so busy keeping up with their own fields that they are unwilling to invest time in learning a new programming language unless they are certain that it will have significant impact on their working lives. We therefore propose to make Python more compelling for this community by creating training materials tailored to science and engineering that are suitable for classroom and self-directed use, which demonstrate how Python can be a foundation for more productive software development. We believe that doing this will increase Python's user base, and potentially lead to increased institutional support for Python.

## 1) Introduction

Many scientists and engineers spend the greater part of their working lives developing or using software. However, few of them have ever been taught how to do this effectively. After a freshman course on C or Java, and possibly a junior or senior course on numerical methods, they are left to their own devices. As a result, they do not understand how to use agile tools such as Python for rapid prototyping, data crunching, and application steering, and are unfamiliar with, or completely ignorant of, productivity aids such as version control systems, unit testing tools, and the like.

Over the past seven years, the author of this proposal has used Python extensively in classes on lightweight software engineering for scientists and engineers. That experience has shown that scientists and engineers are only willing to adopt new tools and working practices if their utility is clearly demonstrated. In particular, if potential adopters are shown how Python will help them solve their immediate problems, *and* how it fits into a larger picture of more efficient software development, a significant number will become enthusiastic users.

## 2) Proposal

We propose to create teaching material that can be used by instructors of college-level courses, and for self-directed study. All material will be made freely available under a BSD/MIT-style open license, to be selected by the PSF, in order to encourage the widest possible use and redistribution.

This material will be tailored to the specific needs of scientists and engineers: text processing exercises, for example, will show how to manipulate common scientific data formats, rather than invoices or student enrolment records, while the material on testing will include discussion of how to test numerical programs.

Specific deliverables will include:

- Lecture notes suitable for immediate in-class use (i.e., that do not require massaging or filling in before presentation). These will be prepared in XHTML (for on-line deployment) and PowerPoint (for live presentation); the PowerPoint version will be tested for compatibility with recent versions of OpenOffice.

- Explanatory material to help instructors and self-directed students.

- Well-documented example programs illustrating all important points in the above materials.

- A large number of exercises, ranging in complexity from five-minute tests of comprehension to multi-day class projects. Solutions to these exercises will *not* be included in standard distributions, but will be made available to instructors upon request. This will both lengthen the useful life of the exercises, and give users of this material a reason to provide the course material's maintainer(s) with contact information.

This material may seem redundant to people who picked the language up using existing books and on-line material. However, this impression does not take into account those people who tried Python, but found the learning curve too steep because they lacked appropriate background knowledge, or (more importantly) who didn't try at all because they didn't know they could or should. Scientists and engineers are unlikely to stumble across books such as Lutz and Ascher's *Learning Python* (O'Reilly, 2003) on their own; even if they do, only a minority have the background to recognize the value of what they have found, and to make sense of it once they do. Langtangen's

*Python Scripting for Computational Science* (Springer, 2004) goes some way toward addressing this, but its focus is on traditional scientific computing, rather than on the software development process. The author believes Python is more likely to be adopted if presented in the latter role, where it does not have to compete head-to-head with MATLAB, Mathematica, and other entrenched solutions.

## 3) Budget

The author intends to use material prepared for courses taught over the past six years as a starting point for this work; all rights to this material will be transferred to the PSF upon completion of the work. Upgrading the existing material (which does not use features added to Python since Version 2.1), extending it to cover all of the core topics listed in Appendix A in appropriate depth, and clarifying it to the point where it can be used for self-directed learning by people who aren't already familiar with basic software development concepts, will require 10 to 12 weeks of full-time work. Unexpected contingencies, the production process, coordination with the PSF and selected reviewers, etc., are expected to add two more weeks to this schedule. The total amount requested is therefore US$27,000. The author proposes that payment be scheduled as follows:

- Three payments of US$6000 each upon completion of one quarter, half, and three quarters of the material outlined in Appendix A.

- A final payment of US$9000 when all of the material outlined in Appendix A is ready for classroom use.

If this proposal is accepted, the author will ask the PSF to nominate a reviewer to confirm that material is complete and ready for classroom use. If, during the course of this work, the author wishes to deviate in any significant way from the outline in Appendix A, he will discuss proposed changes with the reviewer, and produce a new schedule of deliverables before proceeding.

## 4) Proposer

Born in Canada, Greg received a Ph.D. in Computer Science in 1993 from the University of Edinburgh. Since then, he has worked for companies ranging in size from six-person startups to IBM and HP, on projects ranging from data visualization to access control and identity management. Greg is the author of *Practical Parallel Programming* (MIT Press, 1995), and editor of *Parallel Programming Using C++* (MIT Press, 1996); he has also written several dozen articles and reviews for *New Scientist*, *Doctor Dobb's Journal*, and various academic journals. Greg is now a senior software developer with Hewlett-Packard, an adjunct professor at the University of Toronto (where he recently developed a new core undergraduate course on software tools and methods), and a contributing editor with *Doctor Dobb's Journal*.

Greg's involvement with Python dates to 1999, when he converted materials used in a course on software engineering from Perl to Python (and discovered that he could then cover things in two days that had previously taken three). He organized the Software Carpentry project, and was the original author of PEP 218, "Adding a Built-In Set Object Type". In the past six years, he has taught Python at Los Alamos National Laboratory, the Space Telescope Science Institute, the US Navy's Bettis Laboratory, and most recently at the University of Toronto, where he included it in CSC207, a new compulsory undergraduate course in Computer Science.

## 5) References

Gene Amdur
Hewlett-Packard Canada
901 King Street West
Toronto, Ontario M5V 3H5
gene.amdur@hp.com

Dr. David Ascher
ActiveState
580 Granville Street
Vancouver, BC V6C 1W6
davida@activestate.com

Jon Erickson
Editor-in-Chief
*Doctor Dobb's Journal*
2800 Campus Drive
San Mateo, CA 94403
jerickson@ddj.com

Brent Gorda
Lawrence Livermore National Laboratory
L314 7000 East Avenue
Livermore, CA 94551
gorda1@llnl.gov

Prof. Christopher Hogue
The Blueprint Initiative
522 University Avenue, Suite 900
Toronto, Ontario M5G 1W7
chogue@blueprint.org

Prof. Diane Horton
Dept. of Computer Science
Bahen Centre for Information Technology
Room 4236
University of Toronto
Toronto, Ontario M5S 2E4
dianeh@cs.utoronto.ca

Steve McConnell
Construx Software
11820 Northup Way #E200
Bellevue, WA 98005
stevemcc@construx.com

Irving Reid
Hewlett-Packard Canada
901 King Street West
Toronto, Ontario M5V 3H5
irving.reid@hp.com

## A) Proposed Course Outline

- One lecture on the software development lifecycle, followed by another on how and why to use version control. The first lecture is an opportunity to explain what benefits students can expect from adopting Python; the second allows instructors to manage coursework using a core software development tool.

- Four lectures on core features of the Python language (including file I/O, strings, lists, dictionaries, functions, and exceptions, but *not* classes and objects). Experience shows that this is as much "pure language" material as students are willing to absorb without further evidence of relevance.

- A lecture each on file system programming and process control, both of which they can usually apply directly in their work.

- One or two lectures (depending on previous experience) on object-oriented programming. Advanced features (such as meta-classes) are *not* included.

- A couple of lectures on testing (one on the theory, the other on the mechanics of unit testing). Experience shows that it is often easier to teach testing to scientists and engineers than to computer scientists, since the former are already "believers" in good lab practices.

- A lecture (sometimes two) on debugging (including how to use a symbolic debugger, if they aren't already familiar with one).

- Three or four lectures on data crunching (regular expressions, handling binary files, the basics of XML, and optionally an overview of working with relational databases). These are tasks to which Python is ideally suited; working with it quickly convinces students of Python's value.

- Three or four lectures in which they're shown how to build a simple version of make (which introduces them to parsing and to graphs), how persistence works, and so on. The real goal is to show them how to design non-trivial programs, and to provide examples that the next pair of lectures can refer back to. This sequence is also a good opportunity to demonstrate how much easier Python makes life…

- A lecture each on design patterns and refactoring, which refer back to the material in the previous three or four lectures for examples.

- Two or three lectures on GUI programming, building CGI scripts, performance tuning, or whatever else the class seems most interested in. At this point, students are ready to work in pairs or small teams, and are able to tackle problems such as automating their overnight build-and-test cycles, and producing RSS feeds that report the results.