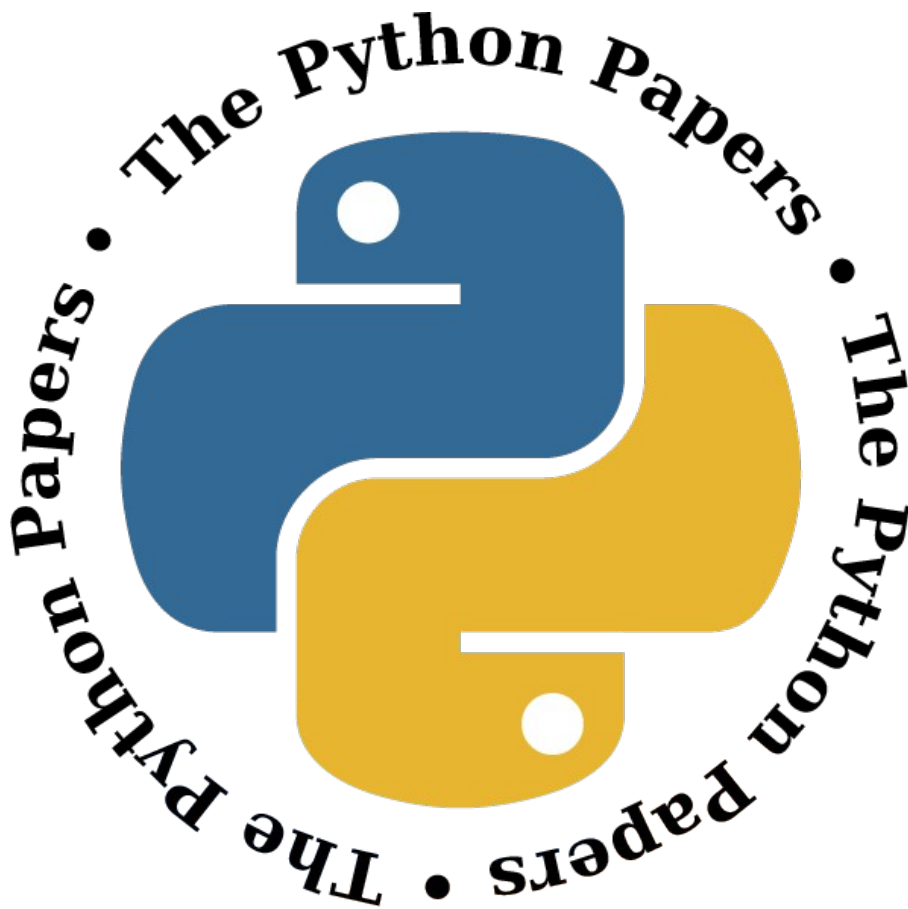


The Python Papers



Volume 2, Issue 4
pythonpapers.org

Journal Information

The Python Papers

ISSN: 1834-3147

Editors

Tennessee Leeuwenburg
Maurice Ling
Richard Jones
Stephanie Chong

Referencing Information

Articles from this edition of this journal may be referenced as follows:

Author, "Title" (2007) *The Python Papers*, Volume N, Issue M, pp. m:n

e.g. Maurice Ling, "Firebird Database Backup by Serialized Database Table Dump" (2007) *The Python Papers*, Volume 2, Issue 1, pp. 7:15.

Copyright Statement

© Copyright 2007 The Python Papers and the individual authors
This work is copyright under the Creative Commons 2.5 license subject to Attribution,
Noncommercial and Share-Alike conditions. The full legal code may be found at
<http://creativecommons.org/licenses/by-ncsa/2.1/au/>

The Python Papers was first published in 2006 in Melbourne, Australia.

Referees

An academic peer-review was performed on all academic articles. A list of reviewers will be published in each December issue. This has been done in order to ensure the anonymity of reviewers for each paper.

The Python Papers

Volume Two, Issue Four : December 2007

In this Edition:

Articles and Editorials:

Editorial

Page 4

Editor-In-Chief, Tennessee Leeuwenburg

This issue marks some new beginnings. Gloria W. Jacobs joins us as a regular columnist, chiefly addressing the issue of women in computing. We also include an interview with Giles Thomas of Resolver Systems, our first article aimed equally at developers and the business community. It is hoped that we will be able to include a lot more content which is of interest to business. If we can stimulate more business interest in Python, it is believed this can help developers to link with the business community. As many of our readers are involved with open-source software, finding ways to effectively link with business will be of great interest. Understanding the challenges faced by businesses which seek to use and indeed contribute to the open-source Python software is a highly relevant task.

GrrlCamp: An Open Source Python Project for Women

Page 6

Gloria W. Jacobs

Today, from where I stand as an experienced software developer, I survey the current landscape and see a similar picture to that of ten years ago. Only this time it strikes me as sad. Ten years ago, I considered it more to be happenstance, but that excuse is no longer valid. There really should be more female F/OSS enthusiasts, hobbyists, tinkerers and hackers. I have heard many women echo my concerns, and many more ask where they can find such projects, so that they may become those enthusiasts, hobbyists, tinkerers and hackers.

News from the Python Software Foundation

Page 9

Stephan Deibel, Chairman of the Board, Python Software Foundation

"The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of the international community of Python programmers."

The Python Software Foundation (PSF) is the non-profit legal entity that holds and protects the intellectual property rights behind Python, keeping it free and open for all to use. The PSF also contributes to the Python community by underwriting or sponsoring conferences, funding grants and special projects, and leading volunteer efforts.

Everyone that programs with Python should know at least a little about the PSF, especially how the Python license and trademarks work. While this may seem boring to many programmers, there is much that an aware Python users base can do to help protect Python, and keep the community behind Python vital and productive.

An Interview with Michael Foord on IronPython

Page 16

Michael Foord

I've always loved programming. I started with Basic on the BBC microcomputer back when 32k was a lot of memory (actually 64k was a lot of memory, but 32k wasn't bad). I moved onto assembly language on the Amiga (a truly beautiful computer and operating system),

and then took a break from programming for ten years.

I got involved with Python completely by accident about five years ago and immediately fell in love with the elegance and expressiveness of the language. I've been writing articles and involved in open source projects ever since.

This article includes a sample section from Michael's upcoming book, "IronPython in Action"!

Screen Scraping Web Pages

Page 22

by Corey Goldberg

This tutorial shows how to programmatically retrieve a stock quote from Google Finance. It uses Python's high level Web API and screen scraping with regular expressions.

Interview with Resolver Systems

Page 24

The Python Papers interviews Giles Thomas

The Python Papers presents an interview with Managing Director and CTO of Resolver Systems, Giles Thomas. Giles gives us an in-depth look at his company and how Python supports the business. This is a great case study into how Python can be used commercially. It includes a discussion of .NET technologies, the pros and cons of Python as a language and other great topics.

Eight Tips to Start With Python

Page 30

Tarek Ziade

A friend of mine is starting Python. I tried to sum up some tips for him, that may be useful to others. Don't hesitate to comment it if you think something important is missing.

Python Events

Page 80

A list of upcoming Python events.

Peer Reviewed Submissions:

Acknowledgment of Reviewers

Page 34

Editorial Board

In order to preserve the anonymity of reviewers, those who have been a part of the peer-review process are acknowledged each year in our December issue (starting with this issue). Those listed here have our thanks for helping to pioneer *The Python Papers*.

pypk - A Python extension module to handle chemical kinetics in plasma physics modeling

Page 35

N. Pinhão¹

Nuclear and Technological Institute, Physics Dept., Estrada Nacional 10, 2685 Sacavém, Portugal

PLASMAKIN is a package to handle physical and chemical data used in plasma physics

¹ Electronic address: npinhao@itn.pt; URL: <http://plasmakin.sourceforge.net>

modeling and to compute gas-phase and gas-surface kinetics data: particle production and loss rates, photon emission spectra and energy exchange rates... pypk is a new addition to the package and provides access to libpk from Python programs. It is build on top of the ctypes foreign function library module and is prepared to work with several Fortran compilers. However pypk is more than a wrapper and provides its own classes and functions taking advantage of Python language characteristics. Integration with Python tools allows substantial productivity gains on program development and insight on plasma physics problems.

The When and How for Design Patterns

Page 48

Olemis Lang

Patterns are powerful tools in order to build trustworthy systems quickly, therefore the article reveals that the core Python system incorporates the GoF design patterns comprehensively. Additionally, it portrays how their structure can also describe interactions among artifacts other than classes. All the way through, several illustrative examples (many of which are extracted from Python Enhancement Proposals) are described using either Python implementations or UML diagrams. They are the starting point to sketch a simple (perhaps naive) method for pattern hatching, and also to unveil the relevant subjects that determine when and how patterns are found.

Python Switch Statement

Page 58

Lance Finn Helsten

The Python programming language does not have a built in switch/case control structure as found in many other high level programming languages. It is thought by some that this is a deficiency in the language, and the control structure should be added. This paper demonstrates that not only is the control structure not needed, but that the methods available in Python are more expressive than built in case statements in other high level languages.

An Introduction to Test-Driven Code Generation

Page 64

Raphael Marvie

Agile Software Development promotes the use of techniques such as *Test-Driven Development* (TDD) and *Automation* in order to improve software quality and to reduce development time. Code generation represents a way to achieve automation, reducing repetitive and error-prone tasks.

Code generation is well accepted, writing a code generator is not necessarily that hard, however it is not trivial to decide when and how to embrace code generation. Moreover, it is even harder to embrace at the same time code generation and TDD, wondering for example *How to build a generator following a test driven approach?* or *How to test drive generated code?*

This paper aims at providing hints to answer these questions. It presents an agile approach named *Test-Driven Code Generation*. The main principle is to gain knowledge about the application during the first iterations of the development process and then to identify how to implement code generation. As code generation should not drive you out of TDD, our proposal, based on a *double TDD Loop*, provides hints to marry both approaches in order to empower your developments.

Letter from the Editor

Tennessee Leeuwenburg



Hello to the readers of *The Python Papers*! Here we are at the commencement of our second year of publication. This issue is stuffed full of content. It feels great to be putting our best foot forward after a difficult but rewarding first year.

This issue contains more original academic work than ever before, showing that our efforts in that area are beginning to pay off. We are continuing to pursue associations with major academic databases in an effort to lift the profile of our authors and of Python. We really appreciate the authors who have submitted papers to our journal as it helps us to present a good case for consideration as a serious academic journal as well as being a home for hobbyists and business users.

This issue marks some new beginnings. Gloria W. Jacobs joins us as a regular columnist, chiefly addressing the issue of women in computing. We also include an interview with Giles Thomas of Resolver Systems, our first article aimed equally at developers and the business community. It is hoped that we will be able to include a lot more content which is of interest to business. If we can stimulate more business interest in Python, it is believed this can help developers to link with the business community. As many of our readers are involved with open-source software, finding ways to effectively link with business will be of great interest. Understanding the challenges faced by businesses which seek to use and indeed contribute to the open-source Python software is a highly relevant task.

The next year will be, for me, a real litmus test for the longer-term future. We have a big agenda with the further expansion of our academic and business links. In this issue, we miss our Python User Group Highlights, but we have lined up regional representatives in a number of countries and hope to continue turning the journal's spotlight on the groups of people which make up the Python user community.

Readers of our blog will already be aware that we are launching a companion publication, *The Python Papers Monograph Series*. *TPPM* will carry longer publications, conference proceedings, cutting-edge articles and other content which is inappropriate for the regular journal to contain but which deserves a good home. We hope thereby to provide our experiences with publishing this journal to community groups which may need help with publishing their works. This has the potential to be a significant task however, and will need the support of the community.

We are also going to be expanding our editorial board further during the next year, hopefully before our next issue. We would love to include more people in the production of this journal, at high levels or otherwise. There is definitely a role for anyone who is interested in being a part of our team.

A special thanks go out to those who help us by proof-reading our articles, pointing out errors and helping us to manage the workload of maintaining a good editorial standard in a volunteer publication.

Introducing The Team

Tennessee Leeuwenburg – Editor-in-Chief

Tennessee Leeuwenburg is a software developer working at the Australian Bureau of Meteorology on automatic text generation of weather forecasts. Prior to this he spent time working on meteorological data transfer standards in the form of the OpenDAP database system.

Maurice Ling – Associate Editor

Maurice Ling is a PhD candidate in the department of Zoology of The University of Melbourne working on text analysis of biological literature for the purpose of understanding hormone interactions in the mouse mammary cell.

Richard Jones – Associate Editor

Richard is Blue Box Device's lead OpenGL developer with over 10 years of broad experience working with multiple languages and tools in web-based management systems, data archive, meta-data systems, computer graphics, business systems, e-commerce and communications. He also runs the bi-annual PyWeek Game Programming Challenge and is involved with the organisation of the Open Source Developer's Conference in Melbourne, Australia.

Stephanie Chong – Associate Editor

Stephanie is currently studying Arts/Law at The University of Melbourne.

Contacting The Python Papers

The editors may be contacted via email at: editor@pythonpapers.org. We are always happy to receive feedback, suggestions for improvement and ideas for future articles and topics.

Contribute to The Python Papers

If you would like to contribute an opinion piece, an article, participate in an interview or submit a paper for review and publication, please don't hesitate to contact us at editor@pythonpapers.org

GrrlCamp: An Open Source Python Project for Women

Gloria W. Jacobs

The evolution of Free/Open Source Software was as much fun to watch as it was to help push along. When I started designing and developing software professionally in 1989, compilers cost thousands of dollars and freeware was just a tiny dream held by a small group of UNIX developers and college professors.

I worked for a company which ported their software to the AT&T 3B series, the Pyramid symmetric multiprocessor and the IBM XT. The XT version sold for \$200 while the versions for monster-sized machines sold for tens of thousands of dollars. When I asked why, the response I got was essentially that companies which can afford these monster machines can also afford monstrous software and licensing fees. Still no fair, I thought, especially since the features on all versions were the same. But I was green and scooped up into this very good job six months before I graduated, so I did not protest. I was happy to have a company want to hire and train me, as well as allowing me to finish my studies simultaneously.

There simply were no other options in 1989. People paid what companies wanted for software, even down to the compiler and its supplemental components. The hobbyist in this field was either rich, or moonlighting on university equipment to design and develop the software of their dreams (often with great difficulty). Cross-platform testing was nearly impossible. Lab time had to be negotiated. It was not fun, to say the least.

In the early 90's, I began to see some small embedded-systems companies play with software from the Free Software Foundation. The "gas" (GNU Assembler) was often tweaked to work with proprietary hardware. The GNU C compiler (gcc) was maintained privately by a tiny group, as well as professionally by companies charging \$7000 a year for support. The professional bug fixes were not shared with the public and were not rolled back into the free code base, thus defeating the purpose of the free GNU tools in the first place. We were far from a solution, but knew we would get there eventually.

As hardware dropped in price, more enthusiasts and hobbyists sprung forth, and more freeware was developed. Now moonlighting could be done in the comfort of one's own basement, on a better PC, or even on a \$700 Motorola ethernet-ready prototype board. The quality got progressively better, and the projects bigger, to the point where we are today. Free/Open Source Software, or F/OSS, dominates the application development world (does it?) and has infiltrated, but not yet consumed, the embedded systems world. In both cases, software design and development is easier, cheaper and takes less time to ramp up. We have a plethora of tools to draw upon and start with, so that project time and cost is incredibly less than it was 15, 10, or even 5 years ago. F/OSS has exploded all over the world, enabling everyone with a PC and an idea to explore and create.

I saw this evolution and helped push it along wherever I could. I helped embedded systems companies hack the GNU assembler and use other GNU tools to automate building and testing. I helped set up SAMBA servers for companies, showing them how Linux provides affordable and reliable file servers (and authentication services?) for Windows clients. I taught people how to use GNU make, wrote Perl wrappers, web interfaces and automation tools for various companies over many years.

It was great fun, but lacking one aspect: women. I was often the only female for miles in many of these environments. I didn't mind; I learned how to handle the bad, enjoy the good and focus on my ultimate goals of learning and getting the work done. My hobbyist tendencies also flourished as I could afford to buy better hardware, run free software on it and have a few side projects at any moment in time. It was and still is great fun for me.

Today, from where I stand as an experienced software developer, I survey the current landscape and see a similar picture to that of ten years ago. Only this time it strikes me as sad. Ten years ago, I considered it more to be happenstance, but that excuse is no longer valid. There really should be more female F/OSS enthusiasts, hobbyists, tinkerers and hackers. I have heard many women echo my concerns, and many more ask where they can find such projects, so that they may become those enthusiasts, hobbyists, tinkerers and hackers.

This inspired me to start something called *GrrlCamp*. It is an Open Source project which will be written mostly in Python, my favorite language to date. The intuitive nature of Python syntax, plus it's diversity, robustness and overall speed makes it the perfect choice for every project I've done since 2001. Python is ideal for *GrrlCamp* also because it is fairly easy to teach online. Since I run my own publicly accessible servers from my basement, host commercial software from these servers and have taught online before, I figured it was time to dedicate one server (and much of my time and energy) into starting this project.

This project is unique in several ways. It is entirely voluntary, even to the point of turning down offers for corporate sponsorship to ensure that it is guided only by volunteers and contributors. It is only for women, but it does not matter how novice or experienced they are. The purpose of the project is for training as well as fun, so a few *GrrlCamp* members offer online classes to members for free. It is entirely hosted from my home on my own server and all participants take part in everything, including system administration. I started by opening up services on my server, advertising for *GrrlCamp* participants on the net, and preparing and organizing online group material.

I was fortunate to have a volunteer come forward and offer her outstanding organizational skills to our group. This helps greatly. I have quickly learned that having a "kick-ass" group manager on volunteer projects of this size is as essential as a right arm.

Within several weeks, I had a total of 31 participants from all over the world and in so many time zones that we need to hold weekly repeat chat sessions ten hours after our first ones, simply to ensure that we reach out to everyone and everyone can reach back. I started by passing out system admin assignments and the work began.

Everything from the definition of the groups' operating rules and responsibilities to the build and installation of all of the necessary Open Source packages is done by the group. *GrrlCamp* has a mostly flat management structure with the exception of one small "bump", being the "kick-ass" group manager's absolute veto power over anything absurd. Everything is decided by consensus and discussion. Needless to say, this is verbose and time consuming, but so is any volunteer project where people contribute great time, energy and effort.

We are currently at the end of the system admin and group principle definition phases, finishing up the bash shell training and beginning OO design and development basic training, as well as Python basic training. We now also begin the design phase of the project source code and the content evaluation phase of how we document the process as well as the project. Our plan is to continue online until May 2008, when we meet in person in New York City for a week, wrap up the project and release a Beta version to the public under a freeware license.

My hopes with *GrrlCamp* are multifaceted. Firstly I want to help a group of women learn and experience Open Source development from the ground up, just as it is done by most startups. This is an experience many women in IT want, but have not yet had for various reasons. Luckily it is an experience I have enjoyed twenty times or more. I design, develop and administer projects for startups from scratch, all the way to production, which has, in hindsight, made me wonder why I did not start *GrrlCamp* sooner

With *GrrlCamp*, I also have a bigger goal of documenting the process as well as the project, to try and preserve the learning experience and simplify it for future groups. The system admin tasks, the discussions on participant principles, the design and development process as well as the online classes, the code and the project, are being categorized, summarized

and “wiki-fied.” This is only possible by having 31 people and an outstanding project manager, and it is so far very successful.

Within *GrrlCamp*, all information is shared, all thoughts are shared and all questions are aired. General consensus provides the answers and direction. For this reason it is much more than just another Open Source development project. It is an evolving social experiment, a project becoming more and more cohesive and substantial by the day. It has only been three weeks since *GrrlCamp* started, and we have come a long way.

As time progresses I'll talk more about the tools and the technical aspects of the project. For now I'll leave off by saying that we'll be using CherryPy and SQLAlchemy, two of my favorite development tools, on the back end portion of this project. More information about GrrlCamp will follow in subsequent issues of *The Python Papers*.

News from the Python Software Foundation

Stephan Deibel, Chairman of the Board, Python Software Foundation

"The mission of the Python Software Foundation is to promote, protect, and advance the Python programming language, and to support and facilitate the growth of the international community of Python programmers."

The Python Software Foundation (PSF) is the non-profit legal entity that holds and protects the intellectual property rights behind Python, keeping it free and open for all to use. The PSF also contributes to the Python community by underwriting or sponsoring conferences, funding grants and special projects, and leading volunteer efforts.

Everyone that programs with Python should know at least a little about the PSF, especially how the Python license and trademarks work. While this may seem boring to many programmers, there is much that an aware Python users base can do to help protect Python, and keep the community behind Python vital and productive.

Organization

The PSF was formed in 2001 as a non-profit corporation in the state of Delaware, USA. Although it was formed in the US, the PSF's mission is world-wide, extending to wherever there are Python programmers. Its membership is international, including individuals that have played a significant role in the Python community and companies that sponsor the PSF because they have an investment in Python. Both individual and sponsor members have voting rights in the organization and thus are selected based on their ability to take part in the PSF's stewardship role. It is the existing members that vote to accept new individual and sponsor members to the organization.

The officers and board of directors of the PSF take care of most of the day-to-day business of running the organization. The board of directors is voted in by the members and the officers are appointed by the board. Some of the officers are paid a small salary since their work requires constant availability and many hundreds of hours of effort per year. The other officers and directors are unpaid volunteers. Currently, we have officers and board members in the US, Canada and Germany.

The board of directors meets monthly, via IRC, and the PSF members meet once a year in person at the North American PyCon conference. Members that cannot be present in person at the annual meeting can vote by ballot submitted ahead of time or by proxy (assigning their vote to someone they trust).

More information, including the membership roster and public records, is available at <http://python.org/psf>

Financial Overview

The PSF is funded from a variety of sources. For example, in 2006 a total of \$168K in revenues came from: Individual donors (13%), PSF sponsor fees (17%), PyCon registration and PyCon sponsor fees (51%), participation in the Google Summer of Code program (14%), and interest on investments and other minor sources (5%).

In 2006 the PSF spent \$158K total, for: PyCon 2006 related expenses (67%), funded grants (13%), the advocacy special project (10%), payroll for officers (5%), and other expenses such as bank charges, legal fees, and insurance (5%).

The PSF's assets currently stand at about \$310K. A healthy balance enables the foundation to take on the financial risk of underwriting PyCon each year, and provides a reserve for any unexpected expenses, legal or otherwise.

The PSF has obtained the status of 501(c)3 non-profit, which makes contributions from individuals deductible from US federal taxes. The PSF is not yet officially registered as a non-profit in other countries. However, in many cases, donations and sponsor fees paid by a business are also pre-tax, but this depends on local rules.

Licensing

Python is licensed under a "stack" of open source licenses, which represent its history as an open source project under the sponsorship of a number of different organizations. The topmost license on this stack is the PSF License (<http://opensource.org/licenses/PythonSoftwareFoundation.php>). Everything added to Python since version 2.0.1 is covered by the PSF License, while some older portions of the code are covered by the older licenses lower down on the stack.

For most users, the details of the license stack are irrelevant. All of the licenses grant permission to use, modify, and redistribute the code in a non-restrictive way. However, it is a good idea to have a basic understanding of how the licensing works, how it relates to copyright ownership, and what factors make an open source release legally valid.

In fact, the PSF does not directly own the copyright to any of the Python source code. The copyrights are retained by the organizations and individuals that contributed the code. The PSF obtains its rights through *licensing*. An important concern here is that the licensing was made by those that had a legal right to do so, in other words by the original authors or copyright holders of the code.

The legal history of the code in Python is thus defined in two ways:

(1) In the earlier days of Python, employee contracts and related law ensured that the organizations that released Python under the various licenses had the rights to the code that was added in each release. The open source licenses used by those organizations granted all the necessary rights to the world in general, which of course includes the PSF. The only significant right the PSF does not have to this code is to re-license it. This is the reason for the continued existence of the older licenses in the license stack. Since some of the organizations involved no longer exist, it is unlikely that the PSF could obtain relicensing rights in the future. Thus, the older licenses cannot be removed from the license stack.

(2) For newer contributions, including all those since the PSF License was introduced, the PSF obtains the necessary rights to the code first from the author through a contribution agreement. The author licenses the code to the PSF under one of several specially selected open source licenses and simultaneously signs an agreement that gives the PSF the right to relicense the code without restriction. The code is then re-licensed under the PSF License.

The initial licenses that accompany the contribution currently can be either the Academic Free License version 2.1 or the Apache License version 2.0. These were selected because they contain a patent clause and avoid unnecessary restrictions on the licensee.

The important thing to note here is that the PSF can demonstrate that Python contains only code licensed to it by organizations and individuals that had the necessary rights to the code in order to make that license legally valid. Python's case is further strengthened by the existence of an intact revision control record that dates back to its very beginning, and extensive email and other records that help to establish when and by whom the code was developed. This body of evidence makes any legal claim against Python's open source status difficult and very unlikely to succeed.

For more information, see the PSF License FAQ:

<http://wiki.python.org/moin/PythonSoftwareFoundationLicenseFAQ>

Trademarks

The PSF also holds and protects the trademarks behind the Python programming language. This includes the "Python" name, when used in the domain of programming languages, and also the Python logos. "Python" is a registered trademark in the US, while the logos have not yet been registered.

There is much confusion about the nature and purpose of trademarks, and they are often incorrectly viewed as incompatible with open source. This is simply not true. Trademarks are another way to help keep Python free and open for all to use, and protecting the trademarks helps to prevent confusion about the nature and purpose of Python.

For example, if an entirely different programming language were named "Python" or even "Python Plus", it could cause considerable confusion, making life more difficult for the developers and users of either language. Any other Pythons should be the same programming language, within reason, even if they implement the language differently as do IronPython and Stackless Python.

Similarly, the Python logos must refer only to the Python programming language and they should not be modified in any way (color, font, shape, etc.) as this can cause confusion. It dilutes the "brand" that helps to make Python distinctive and identifiable to the user (the "consumer").

In fact, trademarks are intended primarily as a way to protect the consumer, and the burden of keeping the trademark clear of confusion falls to the trademark owner. Because of this, the PSF has no choice but to defend the Python related trademarks. Failing to do so would mean that we lose rights to the marks and would be unable to prevent use of the name for another programming language, or misleading uses such as those that obscure the fact that Python is open source.

Because Python is free and open, our enforcement of the trademarks is indeed mostly about avoiding confusion and not at all about holding the trademarks close as would be done with some proprietary marks. We *want* the trademarks to be used as widely as possible. This includes also the unaltered logos, which are often used to indicate suitability for Python or implementation in Python.

One important and useful idea that applies to trademarks in general is the notion of "nominative use" of the trademark. A trademark can *always* be used to name the trademarked entity itself so long as the use does not imply some special relationship with the trademark holder. Thus, the word "Python" can be used anywhere to refer to Python without asking the PSF for permission, regardless of whether or not the use is commercial.

The PSF has extended the idea of nominative use also to the unaltered Python logos, making it possible to use the logos in most contexts without any special permission. Again, we *want* the trademarks to be used widely.

For details, see the PSF Trademark Policy:

<http://www.python.org/psf/trademarks/>

PyCon and Other Conferences

In 2003, the Python community started to run its own low-cost developer-focused Python conference called PyCon. This conference format proved to be quite successful and the conference has experienced substantial growth in size and offerings each year. The original PyCon conference is held in North America. In 2007, PyCon-style conferences were added in Italy, the UK and Brasil.

The PSF has been closely involved with the North American PyCon since its creation. It provides the financial backing that is necessary to plan and run the conference, and has helped in other ways, for example by providing grants to pay need-based travel expenses for selected attendees (\$6.2K in 2007).

Any profit or loss from the conference is absorbed by the PSF. To keep the conference affordable, the organizers attempt to break even or sustain a small loss. However, due to the continued unexpected growth in the conferences, each of the five conferences held so far has turned a profit. For example, PyCon 2007 drew in 593 attendees (seven short of the venue's limit) and an unexpected six platinum sponsors, as well as sixteen other sponsors. While the registration fee for attendees is set below the cost of running the conference, the added sponsor fees more than made up the difference each year.

While unintended, turning a profit at PyCon has helped the PSF build a reserve fund which now makes it possible to take on the significant financial commitment of booking ever-larger conference venues.

More recently, PSF has started to fund some other Python conferences by sponsoring (or donating funds to) the organizations that run the conferences. In 2007 this included EuroPython (\$5K), PyCon Italia (1.5K Euros), PyCon UK (\$2.5K) and the Argentinian Python conference (\$500).

The PSF will continue to provide support for Python conferences around the world, certainly by making donations to each event and possibly also by underwriting the major financial commitments made by these other community-organized conferences as they grow.

For more information, see <http://python.org/community/pycon/>

Grants Program

In 2004, the PSF issued its first call for grant proposals, with a total budget of \$40K. In response to that call, over 60 proposals were submitted. Three of them were selected for funding:

1) *Software Carpentry* (Greg Wilson, \$27K) – an intensive introductory course in basic software development practices for scientists and engineers. The proposed portion of the work was completed in October 2006 and the course is still actively being improved and expanded. For details, see <http://www.swc.scipy.org/>

2) Implementation of PySNMPv3 (Ilya Etingof, \$1.5K) – an SNMP library for Python. This was completed in October 2005. For details, see <http://pysnmp.sourceforge.net/>

3) *Moving Jython Forward* (Brian Zimmer, \$10K) -- to bring Jython up to date with CPython. Due to unexpected changes in the proposer's career, the work for this grant was only partially completed, and only a subset of the funding (\$2.3K) was paid. Even so, the Jython project has seen progress with the release of version 2.2 in August 2007. For details, see <http://jython.org/>

This initial experiment with a formal grant process, although largely successful, has not yet been repeated because of the amount of volunteer labor that the grant review and oversight process consumed.

The PSF board is, however, always open to receiving proposals for funded work, which will be considered along with our other initiatives. Proposals that are likely to be successful are those that will bootstrap a new project or sub-section of the Python community in a relatively cost effective manner. Projects that won't be maintained after the funding ends are unlikely to be considered.

For more information, see <http://www.python.org/psf/grants/>

Special Project: python.org redesign

The PSF has funded several special projects outside of the formal grants process. One of these was the redesign of the python.org website in 2005 and 2006. This included the graphic design for the site and Python logo, repackaging of the site content in a uniform format, and development of a new build system for the site.

The rationale for funding this change was that the volunteer site maintainers did not have the time available to make these major labor-intensive changes and the community had failed to agree on several earlier redesign attempts (both graphical design and improvement of the technology behind the site).

Yet, a general consensus existed that the old site needed some intensive work.

This contract (valued at 4K GBP) was awarded to the proposer, Tim Parkin of Pollenation. The graphical design and the repackaging of content (with help from volunteers during the 2006 PyCon Sprints) were both successful. The new general purpose build system, called Pyramid, was however found to be slow, complicated, and hard to install. This made it difficult for volunteers to be able to get started with maintaining the site. Some improvements were made, but ultimately Pyramid was replaced with a much simpler and faster build system that is custom-designed for python.org.

Despite these initial problems with the build software, this project was a success. The look of the site has been improved and moving to a CSS based design and uniform content format will make any future changes in design or toolset much easier.

To compare the new and old designs, visit <http://python.org/> and <http://web.archive.org/web/20060207222027/http://www.python.org/>

Looking ahead, there is still much work to be done in improving the site, adding and keeping content up to date, and even rethinking some of the structure and presentation (for example, to make it more accessible to new Python users, or to allow annotation). Please consider volunteering as a site maintainer.

For more information, see <http://www.python.org/dev/pydotorg/website/>

Special Project: Advocacy Coordinator

Another special project undertaken by the PSF was the funding of a paid Advocacy Coordinator for 13 months during 2006 and 2007 (\$46K total). This was a proposal from Jeff Rush, who had played a significant role in the organization and promotion of the 2006 and 2007 PyCons. The goal was to bootstrap a community of people interested in advocating and promoting Python.

Unfortunately, this project failed in its primary goal – relatively few volunteers were found to contribute to the advocacy effort and no sustained advocacy community appeared.

However, several important tasks were accomplished and these more than justified the overall expense of the project:

- 1) Targeted promotion of PyCon 2007 led to a 50% increase in the number of attendees at the conference, compared to 2006 (from 410 to 593 attendees). The PSF earned an unexpected profit of \$32.5K from PyCon 2007.
- 2) Intensive effort on the Forrester Wave report on dynamic languages led Forrester to conclude that among all the dynamic languages "Python represents the best combination of features, breadth of applicability, and strong community." These reports are used by application development and management professionals to make technology choices. For details see: <http://www.forrester.com/Research/Document/Excerpt/0,7211,41386,00.html>

3) A number of high-quality screencasts about Python were developed. These are designed to make it easier for new users to get started with Python. They can be seen, along with videos contributed by other authors, at <http://www.showmedo.com/videos/python>

The lesson learned from this experiment is that, although many individual advocates for Python exist, the Python community as a whole is not interested or able to engage in organized advocacy. On the other hand, funding specific projects aimed at promoting Python is a fruitful way for the PSF to approach advocacy in the future and the PSF board is always open to new proposals in this area.

For more information, see <http://wiki.python.org/moin/PythonAdvocacyCoordinator>

Become a Volunteer

The exact total size of the Python user base is unknown, but even the known user base utterly dwarfs the number of people that actively contribute to Python itself, or to the community that surrounds it.

If you are not already working on Python or its website or contributing in some other way, please consider getting involved! Python is open source, so it is – of course – the community that makes it all work.

Helping out is fun, and it can be great for your career as well. Some of the many options to consider are:

1. Work on python.org: <http://www.python.org/dev/pydotorg/website/>
2. Work on updating and expanding the documentation for Python:
<http://www.python.org/doc/>
3. Work on other infrastructure, such as the software behind <http://us.pycon.org/> or the wiki at <http://wiki.python.org/> or by helping with system administration
4. Become an advocate: Blog, write, talk, or screencast about Python, and show your friends what it can do. Casual peer recommendation is the number one most trusted source of information used in making technology choices.
5. Promote Python to your boss and encourage its use in internal projects. A company's use of Python in software testing, build scripting, and intranet applications is often followed by use of Python in its core products.
6. Support your local Python user group, or start a new group in your area:
<http://wiki.python.org/moin/LocalUserGroups>
7. Attend or help out with your regional Python conference:
<http://www.python.org/community/workshops/>
8. Work on Python or its standard libraries: <http://www.python.org/dev/>

Notice that software development is just one part of the overall picture, and there is much that you can do without becoming an expert in programming language design.

Regardless of how you get involved, the best way to get started is to find some useful task or project where you can contribute. Take the initiative to work on it (no need to ask for permission or instructions), announce it, and you'll soon find yourself engaged and having fun as part of the world-wide Python community.

Make a Donation

Even if you cannot get involved as a volunteer, please consider donating to the PSF and Python financially in one or more of the following ways:

1. Donate to the PSF to help fund our activities: <http://www.python.org/psf/donations/>
2. Ask your company to sponsor PyCon. This is a great way to find employees that know Python, or to gain exposure for your company:
<http://us.pycon.org/2008/sponsors/>
3. If your company uses Python extensively, becoming a sponsor of the PSF helps protect that investment: <http://www.python.org/psf/sponsorship/>

More information on the PSF, including the membership roster and public records, is available at <http://python.org/psf>

The author of this article welcomes questions and suggestions and can be contacted at psf@python.org

An Interview with Michael Foord on IronPython



by Michael Foord

Blog: <http://www.voidspace.org.uk/python/weblog/index.shtml>

Book: <http://www.manning.com/foord>

Job: Senior Software Engineer and Community Champion

Employer: Resolver Systems <http://www.resolversystems.com/>

Publisher: http://www.manning.com/affiliate/idevaffiliate.php?id=282_94

The Python Papers> How did you first get involved with software development?

I've always loved programming. I started with Basic on the BBC microcomputer back when 32k was a lot of memory. I moved onto assembly language on the Amiga (a truly beautiful computer and operating system), and then took a break from programming for ten years.

I got involved with Python completely by accident about five years ago and immediately fell in love with the elegance and expressiveness of the language. I've been writing articles and involved in open source projects ever since.

TPP> How did you get involved with IronPython?

About two years ago I started working for Resolver Systems, a new startup in London. They were (well... are) developing a new spreadsheet application aimed at the financial services market.

Since this market is very conservative technologically, Resolver had to be built on an already accepted platform. This basically meant Java or .NET and for a desktop application .NET was the logical choice. The two developers assumed they would be working with C#, the default .NET language. Having an embedded interpreted language is a core part of Resolver and so they started to evaluate .NET scripting language engines. At the time IronPython was at version 0.7, but Giles and William were very impressed with it - particularly the level of integration with the .NET framework. They decided to see how far they could get writing Resolver in IronPython. I was the fourth Resolver developer and the first with any experience of Python!

Two years later, Resolver consists of 30 000 lines of IronPython code, plus another 100 000 lines of Python in the test framework.

Resolver is a great application for anyone who does data processing with Python or spreadsheets. Our public beta is now available and you can download it from our website.

TPP> IronPython is just for Windows users, right?

IronPython runs on both .NET and Mono. Mono is cross platform, running on various operating systems like Linux Mac OS X, Solaris, NetBSD, FreeBSD, OpenBSD and even Windows.

There is a community distribution of IronPython created by Seo Sanghyeon and called FePy (<http://fepy.sourceforge.net>). This includes patches for Mono compatibility and comes with the full Python standard library.

The Mono VM is an impressive platform, and there is a surprisingly healthy community of people who use IronPython on non-Windows platforms.

Advantages of IronPython over CPython, whichever VM you are running on, include:

- *Performance.* Some aspects of IronPython run much faster than CPython, particularly because of the just-in-time compiler.
- *Access to the .NET libraries, including third party extensions.* For Windows development the Windows Forms user interface library is better than anything I have seen available for Cpython. *[There are gtk and other interface libraries available for linux and also Windows - Ed]*
- *No global interpreter lock.* Multi-threaded programs can take advantage of multi-core processors, which they can't on Cpython.
- *Unicode strings.* This is coming in Python 3, but really does make working with text more pleasant.
- *.NET may be an easier corporate sell than Cpython.*
- *IronPython is dramatically easier to extend with C# than CPython is with C.*
- *The .NET platform supports a wide range of 'native' languages (C#, [VB.NET](#), and functional languages like F# and Nemerle just to name a few). Inter-operation between these languages is basically straightforward. The new DLR (Dynamic Language Runtime) extends this range of languages to include IronRuby, managed Javascript, Smalltalk and a port of Lua called Nua.*
- *Through Silverlight the browser can be scripted with IronPython.*

TPP> *You work for Resolver Systems, who are also featured in this issue. Could you describe how you have found the experience of software development using IronPython? How does it compare with other languages you have used professionally?*

My development experience is almost entirely with Python (plus a smattering of Javascript which is hard to avoid these days). The nice thing about developing with IronPython is that it is just Python, but you have this whole range of new libraries to use.

Working with IronPython was my first experience of using Windows APIs. I really didn't know what to expect and feared the worst! To my surprise the majority of the .NET APIs are very pleasant. A few are a bit over-engineered and make you jump through hoops, but in general they are very good to use - almost Pythonic (except for all the camel case method names and static methods instead of functions).

Since working at Resolver I have also regularly used C#. It is a fairly nice static typed language and is sometimes described as 'Java done right'. I have actually read a lot more C# than I have written. The .NET documentation, and most of the online examples, all use C# (which is why I created the IronPython Cookbook). It is very easy to read and not much harder to write, though I still prefer Python. Although they are very different languages, the core object models are similar which helps.

TPP> *You have been writing a book called "IronPython in Action" for Manning Publications Co. How did you come to work with them?*

I first got in touch with Manning by reviewing a couple of books from their "In Action" series on my blog. I've always enjoyed writing, and after programming at Resolver for a few months I thought IronPython would be the good subject for a book. I approached them, and over six months or so we hammered out a proposal for "IronPython in Action". They've been very good to work with and I would recommend them to any aspiring technical writers.

TPP> We have been lucky enough to be able to include a sample section in this

issue. Perhaps you'd just like to introduce it...

This is an excerpt from the first chapter. It is an introduction to IronPython for Python programmers and explains why Python programmers should be interested in IronPython.

TPP> *A lot of our readers will use Linux systems. IronPython can run through Mono. Will your book have something that Linux users will be able to make use of?*

Most of the book will be relevant to Mono. There is very little that is Windows specific in the book, except perhaps for one chapter!

One of the outcomes of the recent Mono summit was the Mono team showing off the native drivers for OS X, accessible through both Windows Forms and GTK#. This means that IronPython is a viable choice for writing genuinely cross platform applications. Windows Forms gets quite a lot of coverage in the early chapters of the book as it is used for the example application. Most of the rest of the examples, including embedding and extending IronPython, should 'just work' on Mono.

Seo Sanghyeon is helping review the book, so if there is anything that is different on Mono, or doesn't work, then hopefully he will point it out for me.

TPP> *What are the major communities for IronPython support and cooperation?*

Central to the community are obviously the IronPython and FePy websites. On the IronPython site you can report bugs and vote on ones that are important to you:

<http://www.codeplex.com/IronPython>
<http://fepy.sourceforge.net>

Most of the support happens on the IronPython mailing list. There is a very friendly community there with a good combination of those using IronPython on .NET and Mono. It's the best place to go for answers to specific questions:

<http://lists.ironpython.com/listinfo.cgi/users-ironpython.com>

Other useful places for information and articles are:

The IronPython Cookbook - <http://www.ironpython.info/>
A wiki with examples of using the .NET framework with IronPython, including embedding IronPython.

IronPython-Urls - <http://ironpython-urls.blogspot.com/>
A blog that covers the IronPython world linking to news and articles.

My pages on IronPython and Silverlight -
<http://www.voidspace.org.uk/ironpython/index.shtml>

TPP> *When is your book due to be published?*

I hope to finish writing the book early in the new year. It will be available in hardcopy two or three months after that.

In the meantime the first six chapters are already available via the Manning Early Access Program. Another four chapters will be added very soon. I'm particularly proud of the chapters on testing with IronPython, metaprogramming with IronPython and the integration with the .NET framework. There will also be a chapter on [ASP.NET](#) with IronPython written by my colleague Christian Muirhead. He has several years development experience writing web applications with both Python and .NET and is also writing the chapter on databases and web services.

Book Excerpt: “1.1.3 IronPython for Python Programmers”

We were lucky enough to gain permission to reproduce a sample section of Michael's upcoming book.

As I mentioned before, IronPython is a full implementation of Python 2.4. If you've already programmed with Python there is nothing to stop you experimenting with IronPython straight away.

The important question is; why would a Python programmer be interested in using IronPython? The answer is basically twofold, the platform and the platform. Let me try and make a bit more sense. First of all I mean the underlying platform that IronPython runs on; the CLR. Secondly, along with the runtime comes the whole .NET framework, a huge library of classes a bit like the Python standard library.

There are several reasons why the Common Language Runtime is an interesting platform. The CLR has had an enormous amount of work to make it fast and efficient. Multithreaded programs can take full advantage of multiple processors, something that CPython programs can't do because of a tricky creature called the 'GIL'[1]. Because of the close integration of IronPython with the CLR, extending IronPython through C# code is significantly easier than extending CPython with C. There is no C API to contend with, you can pass objects back and forth across the boundary without hassles and with no reference counting[2] to worry about. On top of all this, .NET has a concept called 'AppDomains'. These allow you to run code with reduced privileges, like preventing it from accessing the file system, which is a feature that has long been missing from CPython.

IronPython uses .NET classes natively and seamlessly, and there are a lot of them. Two of the gems in the collection are Windows Forms and the Windows Presentation Foundation, which are excellent libraries for building attractive and native looking user interfaces. As a Python programmer, you may be surprised by how straightforward the programmers interface to these libraries feels. Whatever programming task you are approaching, it is likely that there is some .NET assembly available to tackle it. This includes third party libraries for sophisticated GUI components, like data grids, where there is nothing comparable available for CPython. Table 1.1 shows a small selection of the libraries available to you in the .NET framework.

Table 1.1 Common .NET Assemblies and Namespaces

Assembly Name	Purpose
System	Contains the base .NET types, exceptions, garbage collection classes and much more.
System.Data	Classes for working with databases, both high and low level.
System.Drawing	Provides access to the GDI+ graphics system.
System.Management	Provides access to Windows management information and events

	(WMI), useful for system administration tasks.
System.Environment	Allows you to access and manipulate the current environment, like command line arguments and environment variables.
System.Diagnostics	Interact with processes.
System.XML	For processing XML, including SOAP, XSL/T and more.
System.Web	The ASP.NET web development framework.
System.IO	Contains classes for working with paths, files and directories. Includes classes to read and write to filesystems or data-streams, synchronously or asynchronously.
Microsoft.Win32	Classes that wrap Win32 common dialogs and components including the registry.
System.Threading	Classes needed for multithreaded application development.
System.Text	Classes for working with strings (like StringBuilder) and the Encoding classes which can convert text to and from bytes.
System.Windows.Forms	Provides a rich user interface for applications.
System.Windows	The base namespace for WPF, the new GUI framework that is part of .NET 3.0.
System.ServiceModel	Contains classes, enumerations, and interfaces to build Windows Communication Foundation (WCF) service and client applications.

As we go through the book we'll use several of the common .NET assemblies, including some of those new to the .NET 3.0 release. More importantly we'll learn how to understand the MSDN documentation so that you are equipped to use *any* assembly from IronPython.

We will also do some client-side web programming with Silverlight, scripting the browser with Python. This is something that has not been possible before IronPython and Silverlight.

Most of the Python standard library works with IronPython; ensuring maximum compatibility is something the Microsoft team has put a lot of work into. Do beware though. Not all of the standard library works; C extensions don't work because IronPython isn't written in C. In some cases, alternative wrappers may be available^[3], but parts of the standard library and some common third party extensions don't work yet. If you are willing to swap out components with .NET equivalents, or do some detective work to uncover the problems, it is usually possible to port existing projects.

Where IronPython really shines is with new projects, particularly those that can leverage the power of the .NET platform. In order to take full advantage of IronPython there are a few particular features you will need to know about. These include things that past experience with Python alone won't have prepared you for. Before we turn to actually using IronPython, let's first look at how it fits in the world of the .NET framework.

^[1] The 'Global Interpreter Lock', which makes some aspects of programming with Python easier but has this significant drawback.

^[2] CPython uses reference counting for garbage collection, which extension programmers have to take account of.

^[3] Several of these are provided by IPCE – the IronPython Community Edition. We'll look at this in a later chapter.

Screen Scraping Web Pages

by Corey Goldberg

This tutorial shows how to programmatically retrieve a stock quote from Google Finance. It uses Python's high level Web API and screen scraping with regular expressions.

First, let's look at the page we want to get our content from. To get finance data for the ticker "IBM", we use this URL:

```
http://finance.google.com/finance?q=IBM
```

If you enter this URL in your browser, you can see the page we are going to scrape from.

To retrieve the content of the page, we can use Python's urllib module:

```
import urllib
content = urllib.urlopen("http://finance.google.com/finance?
q=IBM").read()
```

Now that we have the content stored, we want to scrape some data from it. If we look inside the content (this is the same as if you used "View Source" in your browser for this page), we see a line that contains the price quote:

```
<span class="pr" id="ref_18241_1">116.26</span>
```

To extract the price quote, we use a regular expression (regex) with matching groups. Regular expressions are a powerful tool for doing pattern matching and text extraction/parsing. Regexes may seem a little arcane (unless you are a Perl hacker), but they allow you to search and manipulate text using a concise syntax.

In our regex, we mark a "matching group". This is the piece of the regex enclosed in parenthesis as a section to extract data. In this case, we use `(.*)` to define the matching group. The group contains metacharacters that match a range of literal characters. This is the pattern that matches our stock quote value:

```
class="pr".*?>(.*?)<
```

Once we do the search, we can get the text we extracted with the matching group. Since we only used one group, our text will be contained in `m.group(1)`:

```
import re
m = re.search('class="pr".*?>(.*?)<', content)
if m:
    quote = m.group(1)
```

Regular expressions are compiled into `RegexObject` instances. If we are going to use a regex frequently, we can optimize it by compiling it once and then using our compiled version:


```
regex = re.compile('class="pr".*?>(.*?)<')  
m = regex.search(content)
```

We can put all of this into a function which accepts a ticker symbol and returns a price quote:

```
import urllib  
import re  
  
def get_quote(symbol):  
    base_url = 'http://finance.google.com/finance?q='  
    content = urllib.urlopen(base_url + symbol).read()  
    m = re.search('class="pr".*?>(.*?)<', content)  
    if m:  
        quote = m.group(1)  
    else:  
        quote = 'no quote available for: ' + symbol  
    return quote
```

About the author:

Corey Goldberg is a software engineer from Boston with over 10 years experience as a developer and tester. He has contributed to many open source projects and developed and maintained several on his own. Corey has a Master's Degree in Computer Information Systems from Boston University. For questions and more information, visit his web site and blog at: www.goldb.org

Interview with Resolver Systems

The Python Papers interviews Giles Thomas

The Python Papers> *How old is your organisation? How was it begun?*

Giles> Resolver Systems was started right at the very end of 2005. The three of us who founded it all had a lot of experience with how spreadsheets were used in financial companies, and knew how unfavourably the spreadsheet developer's experience compared

to the software developer's. We felt there had to be a better way, and we came up with the idea that is now our product. We developed a business plan around it, presented it to a number of investors, and received the "angel" funding we needed to make it a reality.



Photo 1: Giles Thomas (left) and Michael Foord (right)

TPP> *Resolver produces a spreadsheet application. Could you tell us a little about what it does and who your typical clients are?*

Everyone knows that a spreadsheet is just another way of writing a computer program; it is less immediately obvious that this makes spreadsheet programming the most successful tool for software development in the world. And it works really well for simple what-if analyses, or even for complex calculations so long as they are one-off things that can be thrown away after they're used.

The problem is that everything goes horribly wrong when the simple applications become complex, or the disposable ones become long-lived. We believe that things get so messy when this happens because a traditional spreadsheet consists of two loosely-coupled programs: an implicit one written as formulae in a grid, and an explicit one written in a programming language as macros. The connection between the two is so narrow and so ill-defined that it is almost impossible to write a maintainable program.

Our application, Resolver One, is designed to merge these programs together, back into one, to make the overall application more manageable. It does this:

- By making the program that is implicit in the grid become an explicit IronPython program, and by integrating it tightly with the user's own macro-like code (which is, of course, also IronPython). This clarifies the way explicit and implicit code interact, and also makes it easier for the business user to hand over a useable unit of functionality to their IT department so that it can be used elsewhere.

- By allowing the user to put arbitrary IronPython expressions and objects in the grid. This makes developing advanced but well-structured spreadsheets easier, and lets people integrate the Python and .NET components they and their colleagues develop easily into their calculations.

- Our typical clients are in investment banks and hedge funds, in particular the "quants" - quantitative traders and analysts who spend much of their time modeling incredibly complex derivatives in spreadsheets. Very technical people who need a decent graphical way to write their programs. We've also had a lot of interest from people in biotechnology, which we weren't expecting; we're working to see how to best help people in that area.



*Photo 2: Clerkenwell, St John's gate
(http://en.wikipedia.org/wiki/St_John's_Gate - just down the road from the office)*



Photo 3: A typical nearby street scene

TPP> What major competitors exist to Resolver? What makes Resolver unique?

Well, we have a spreadsheet application, so obviously the main competitor is Microsoft Excel! I think our unique selling point is what I described before; we make it possible for people to write simple, maintainable spreadsheets for complex applications. In addition, because Resolver One can produce IronPython code representing the contents of your spreadsheet, you can use it to produce backends for other systems. For example, we were very easily able to write a simple Web server - the forerunner of the Resolver Web Server that we're marketing today - that could serve up spreadsheets as HTML pages with input fields in a form. To put that another way, in a week or two we'd managed to produce something that did important parts of what Microsoft's SharePoint server - a huge enterprise monster - can do with its Excel Services module. It would have been equally easy to write a back-end for, say, an application server.

So, instead of having business people (who are happy with spreadsheets but scared of code) try to explain calculations to developers and go through multiple design iterations before getting a system that works as required, developers can work on putting together the infrastructure that surrounds the calculations - the really difficult and interesting stuff from a technical perspective - and just drop in the calculations, which are the really difficult and interesting stuff from a business perspective.

I think this is a much better division of labour.

TPP> *For how long has your organisation been using Python? I understand Resolver uses IronPython. Was the decision to use the .NET framework made before or after deciding to use Python?*

From the start, our primary market was obviously going to be the business desktop - this means Windows, and the least painful way to develop a slick Windows application is using .NET. So, when we started out back at the end of 2005, we were intending to write our application in C# and embed the traditional CPython engine as a scripting language. But while searching online for details of how to embed CPython in a .NET program, we discovered IronPython. It was still a beta, but we quickly realised that people in the finance world would be very interested in being able to seamlessly use the .NET libraries being built by their IT teams in their Resolver scripts, and so we decided to embed it into our app instead of CPython.

Then came the defining moment: because we were only just starting out, and had no existing codebase to worry about, we thought - let's try writing our application itself in IronPython instead of C#, and see how far we get. After all, if all we had was a few hundred lines of code, we'd be able to translate it over to C# pretty quickly if/when things didn't work out.

That was back in 2005, and we've not needed to do that rewrite yet. Our entire application, which is almost 30,000 lines of production code and over 100,000 lines of functional and unit tests, contains maybe a few hundred lines of C#. The rest is IronPython.

It's a choice we've never regretted. The application is fast, responsive, and fills the needs of our users precisely. Our concerns about performance have proven unnecessary - problems we've had in that area have almost invariably come from our own choice of algorithms rather than anything to do with the language. And IronPython is so nice to code in that we can respond rapidly to changing client needs and make updates to our software relatively painlessly.

TPP> *How does development happen in your organisation? Is the development group large, or small? Is it a "Python shop", or do you employ a mixture of expertise? Do you subscribe to a particular development approach (e.g. Waterfall, Agile/XP?)*

We are an 8.5-person XP shop - the half-person being me, as I'm often out talking to clients. Many of the people in the company, myself included, had no Python knowledge before they started; we believe that a sufficiently-good developer can learn Python on the job quickly, and while we're delighted we've managed to hire some Python gurus, and couldn't survive without them, we're also delighted to be using a language that's clean and simple enough that we don't need every new hire to already be expert in every detail.

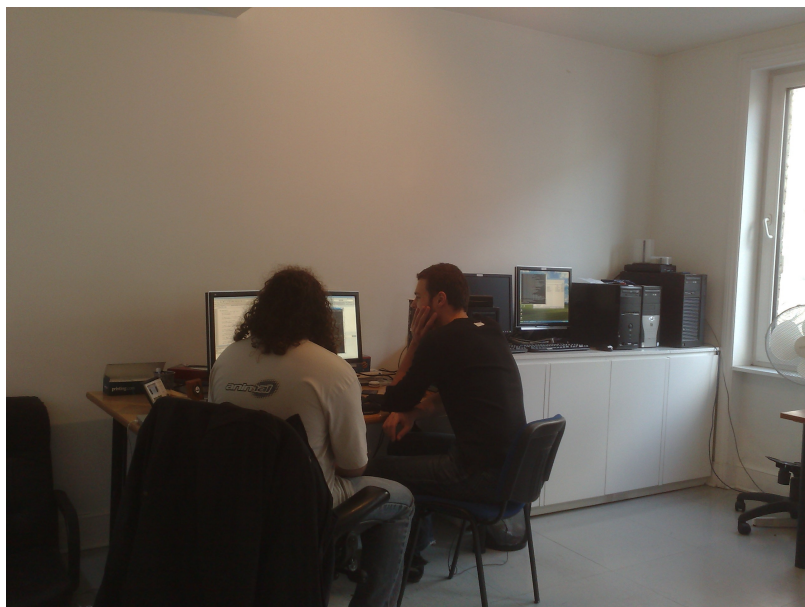


Photo 4: Resolver office interior: Michael Foord and Menno Smits pair-programming. To the right you can see part of our build farm, with a monitor showing one of our 3, 600 test cases running.

TPP> *How did Python fit with the software architecture chosen?*

Python - like, I suspect, all dynamic languages - is a perfect fit for the test-first development required by XP. There are two main reasons:

- Duck-typing and other dynamic language features make writing tests much easier than statically-typed languages.
- When you're writing unit tests for a program written in a statically-typed language, there's a big overlap between the things you test with your unit tests and the things that are checked by the static typing. To put it another way, when you write heavily-tested code in a statically-typed language, you're duplicating effort. With a dynamic language, you don't have that cost; the tests test what needs to be tested. A corollary is that unit-testing is vitally important for programs written in dynamic languages - but I'd argue that it's vital for any non-trivial program anyway, regardless of the language used.

TPP> *Were developers able to work quickly using Python? Has Python been a good choice in terms of the performance of your coders?*

Definitely. Moving from Java to Python has felt to me like my earlier move from C++ to Java - there's this sudden feeling of freedom from all this fussy detail that was holding me back.

But equally important for us, I think, has been the filtering effect. The words "all of our development is done in Python" is an excellent way of maximising the number of good programmers that apply, and minimising the... less good programmers. I think Paul Graham commented on this a long time ago.

Good programmers have either learned Python in their spare time, or aren't afraid of learning it on the job. Bad programmers, people who've read "Java for dummies" and then slouched into a career writing enterprise systems for customers who know no better, are unlikely to have learned a weird, techie language with greek letters in its list of reserved words.

TPP> *Have there been any performance difficulties using Python?*

Not so far! That's not to say that we've not encountered performance problems, but the ones we've seen have been largely due to our own choices of algorithms or to .NET's architecture.

TPP> *Have you had to employ any other technologies to work around problems that could not be solved in Python?*

Yes, we've used C# in a few places in our product and in our testing framework; this has been exclusively in places where we've wanted to call down to the underlying Windows APIs, below the .NET layer. In order to do that, you have to use a service called P/Invoke, which relies on .NET attributes; IronPython, sadly, does not support these right now.

TPP> *Have you encountered any major roadblocks and how did you get past these?*

The biggest issues have been:

- Getting a decent functional testing framework operating. The unit-testing module that comes as part of the Python standard library is great, and works fine with IronPython, but functionally-testing .NET GUIs is harder than we originally hoped. None of the frameworks on the market seem quite enough. We've spent a lot of time getting our one just right.
- Dealing with the CPython C extensions. These don't work with IronPython, and while there's a .NET equivalent for many of them, for some there's not. This is a general problem for IronPython users, not just a problem for us, so we've started an open-source project to build some kind of bridge between IronPython and the C extensions. We've got a mailing list at <http://groups.google.com/group/c-extensions-for-ironpython> and we'd be delighted if readers of the Python Papers joined us! We're working on getting a few basic libraries working right now, with the aim being to support NumPy as soon as we can – and in the long term to see if we can support C extensions in general.



Photo 5: A local pub. The editor does not dare suggest this had anything to do with major roadblocks or getting past them.

TPP> *What other major projects has Resolver undertaken? Did these other projects also use Python?*

We're a one-product company right now, but I expect our future projects to use Python in a big way.

TPP> *If you could 'fix' one thing about Python, what would it be?*

Significant whitespace!

Only kidding, I'm a big fan of stopping people from writing unreadable code. I would say my #1 would be the ability to write multi-line anonymous functions with statements in them. I use lambdas a lot in my coding, but they're just not quite enough... I think Ruby's blocks do the kind of thing I'd like to see, though I can't say I like the syntax.

TPP> *If readers would like to try or buy your application, how can they do this?*

You can download a version to try out from our website, <http://www.resolversystems.com/>

TPP> *Are you currently hiring?*

We're not actively hiring right now, but we are likely to have some open positions in early 2008.

Eight Tips to Start With Python

Tarek Ziade from a blog post dated September 24th 2007 on his blog <http://tarekziade.wordpress.com/>

A friend of mine is starting Python. I tried to sum up some tips for him, that may be useful to others. Don't hesitate to comment it if you think something important is missing.

1. Get the best online documentation.

There are a few online documentation you must read:

- the [official tutorial](#), that gives you a quite complete overview of Python;
- the standard library [module index](#). You can download it to simplify the search through greps. This is the documentation you get through the *help* command in the prompt.
- Active State's [Python Cookbook](#). There are thousands of code snippets that are created, ranked, categorized and commented by developers.
- [Dive Into Python](#) online book, that makes you discover Python features through well thought examples.

2. Read [PyCon](#), [EuroPython](#) and [Pycon UK](#) wrapups and slides.

They are the three main Python events, and a lot of things are happening there. You'll learn a lot by reading the talks slides. If you can go there, it's even better: sprints, bird of feathers and lighting talks are organized. To convince your boss to send you there, you could make a talk proposal "My first steps in Python" 😊

3. Suscribe to the right feeds.

- The mainstream is [Planet Python](#). It gathers most of the blogs out there, so it is the best place to start.
- Pythonware's [Daily Python URL](#). Human-filtered feed. It used to provide several dozains of links per week, but it seems to have slowed down, and provides a few links a week now. I think it's better this way.

4. Learn and use the rising standards.

There are a few libraries that have a deep impact on the way people write and distribute their work:

- [setuptools](#): helpers to build and distribute your code [eggs](#). A public repository à la Perl's CPAN called [Cheeseshop](#) is wired with this library so people can distribute their code there. It's one of the major innovation of last years in Python world in my opinion.
- [sqlalchemy](#): The [ORM](#) that is now used by the majority of Python frameworks. Its flexibility is impressive. I think there is no equivalent tool in any other language (please let me know if there is);
- [Python paster](#). This tool allows you to create templates that can be used to generate skeletons for your code. It is used by many web frameworks to provide people a simple way to generate a standardized boiler-plate code canvas when they start up something. This is done in Java for quite a long time (you cannot do without it in Java, otherwise it would take you years to write any program ;)), and tools like [PyDev](#) and Eclipse would provide the canvas to do similar things. But the paster is independant from any IDE;
- [reStructuredText](#): learn how to use it. It's our LaTeX. Your code documentation should use it.

I am sure they are other tools out of my domain of expertise that are major. I am thinking of libraries in the scientific world for example.

Anyone can help me on this topic to complete this tip ?

5. **Ask for help.** The three places you can get some help are:

- the [mailing list](#)
- the irc channel `#python` on freenode.
- [the tutor mailing list](#). Mihai Campean says: *"This is a list specifically for those new to Python and those interested in helping people learn the language, and the atmosphere is very friendly. It's probably a better place to start than python-list, in my opinion"*

There are some talented guys that dedicate their free time to help newcomers.

6. **Try to adapt your successful code patterns.**

When I started Python, I tried to adapt what I used to do with the tool I mastered then (Delphi). Since *There should be one- and preferably only one -obvious way to do it.* (try `import this` in a prompt), that helped me a lot to learn and understand all the subtles of Python on use cases I mastered. The most pleasant thing about it is that you quickly drop all Python books and guide to work with the language, unlike Java for example, where you need to keep many reference books on your desk.

7. **Share on your experience and participate !.**

A newcomer (yeah! fresh blood!) experience is a highly valuable material for the language advocacy: the discovering state of mind sometime reveals weaknesses or absurdities experienced users don't see anymore. Furthermore, fresh new ideas are often brought by people that comes from other communities. If you feel that something is absurd, unclear or wrong, you should start a thread on the language mailing list. If you have an idea on any kind of enhancement, maybe it worth a [Python Enhancement Proposal](#).

8. **Watch what is being done in Python 3, PyPy and web frameworks**

[Python 3](#) is the next version of Python, [PyPy](#) is Python written in Python. Web frameworks like [Django](#) or [Zope](#) are large Python codebases. These three *sub-communities* have something in common: they form the R&D of the language. [Zope](#) for example, has enhanced a lot *setuptools* and *doctest* through a massive feedback. Keeping an eye on them even if you don't use them will make you live and understand what rises in the language.

[PyPy](#) is an amazing project. Even if you don't understand everything (Python in Python ? what the... ;)), seeing one of Armin Rango talks will give you an instructive high level view of Python. Now for Python 3, even if you cannot read and understand all threads in the dedicated mailing list, keeping an eye of Guido's wrapups and thread subjects will help you to do the jump on P3k, and probably make your Python 2 code look nicer.

Posted by Tarek Ziadé

Filed in [python](#)

8 Responses to "Eight tips to start with Python"

1. [Mihai Campean](#) said:
[September 24th, 2007 at 8:59 am](#)

Thanks for the useful tips Tarek, I am also starting to learn Python and this kind of information is just what I am looking for. However there is a subject less touched by you in this post, and this is good IDE's for Python. I would really appreciate some recommendations on this subject.

2. [Kalle Svensson](#) said:
[September 24th, 2007 at 9:08 am](#)

I'd also recommend the tutor@python.org mailing list (<http://www.python.org/mailman/listinfo/tutor>). This is a list specifically for those new to Python and those

interested in helping people learn the language, and the atmosphere is very friendly. It's probably a better place to start than python-list, in my opinion.

3. [Michael Easter](#) said:
[September 24th, 2007 at 11:35 pm](#)

IMHO, the single most important tip is to get one's hands dirty! Either with quick scripts or in the console... Then find a pet problem (preferably known from a previous language) and implement it.

Also, experiment, play, and be curious. Rediscover the joy of programming.

Experimentation is always rewarding but _especially_ with Python. It's like holding a strange substance that can be liquid and solid at the same time.

4. [Alan](#) said:
[September 25th, 2007 at 12:44 am](#)

The book that helped me most was "Dive Into Python".

More info per page than most, but all explained clearly and cleanly.

Find it at <http://www.diveintopython.org/>


5. [Tennessee Leeuwenburg](#) said:
[September 25th, 2007 at 1:19 am](#)

Hi Tarek,

I thought your article was really great. Also, thanks to the commenter who mentioned tutor@python.org. I didn't know about this (or forgotten about it) and I got a real jolt of "gee, that's a good idea" when I read your comment.

If you would be interested in including your article in the next issue of The Python Papers, I'd love to include this post. Contact me at tleeuwenburg@gmail.com if you'd like to follow up...

Thanks,
-Tennessee

6.  [Tarek Ziadé](#) said:
[September 25th, 2007 at 7:11 am](#)

@Mihai: Thanks for the tip. I think this is a good starting point here: <http://spyced.blogspot.com/2005/09/review-of-6-python-ides.html>. I am going to see how I can integrate it. It's not easy because it's a topic on itself 😊

@Kalle: Thanks, great to know, I added it.

@Michael: You are right, and the prompt is really neat for that.

@Alan: Added, thanks.

@Tennessee: Thanks ! Sure i'll send you a mail

7. [osKanpur](#) said:
[September 25th, 2007 at 2:12 pm](#)

Thanks for the Python tips. I am looking for an easy to "get hands dirty" resource for introducing children to Python programming. Is there any resource which you could point to for children ages 10 - 15 learning their first programming language ? Maybe some simple interactive resource which sums up great ways to do maths or

algebra or in fact any typical school problems with Python ?
Many Thanks osKanpur

8. [Mihai Campean](#) said:
[September 28th, 2007 at 10:05 am](#)

The review of the 6 python IDE's is a good article, thanks for the pointers.

Acknowledgment of Reviewers

The editorial board would like to thank the following reviewers for their assistance during this past year. In order to prevent people reasoning through who reviewed each paper, the reviewers are listed in the December issue only.

1. David Bouchard, Massachusetts Institute of Technology, United States of America <davidb@media.mit.edu>
2. Helen Borrie, IBPhoenix, France <helebor@iinet.net.au>
3. Bruce Cropley, Melbourne eXtreme Programming Enthusiasts Group (MXPEG), Australia <cropleyb@yahoo.com.au> Review Panel. My career has been mainly spent in stockbroking, aviation simulation and security software.
4. Zoltán Donkó, Research Institute for Solid State Physics and Optics, Hungary <donko@sunserv.kfki.hu>
5. Alan Green, Cirrus Technology Pty Ltd, Australia <alang@bright-green.com>
6. Ann W. Harrison, IBPhoenix, France <aharrison@ibphoenix.com>
7. Hugo Liu, Massachusetts Institute of Technology, United States of America <hugo@media.mit.edu>
8. Michael YJ Loke, BearingPoint, Australia <jymloke@gmail.com>
9. Raphael Marvie, University of Sciences and Technologies of Lille <Raphael.Marvie@lifl.fr>
10. James Stroud, University of California - Los Angeles, United States of America <jstroud@mbi.ucla.edu>
11. Sitalakshmi Venkatraman, Victoria University of Wellington, New Zealand
12. Jingxian Wang, The University of Melbourne, Australia
13. Helena Nowakowska, Centre for Plasma and Laser Engineering, Institute of Fluid Flow Machinery, Polish Academy of Sciences
14. Kelvin KL Yeo, Telstra, Australia
15. Roberto de Almeida
16. Flavio Coelho
17. Kevlin Henney
18. Jason Diamond
19. Jessica Wang
20. Art Haas
21. Harry George
22. Graham Ellis
23. Jim Crutchfield
24. Bruce Sherwoord

pypk - A Python extension module to handle chemical kinetics in plasma physics modeling

N. Pinhão²

Nuclear and Technological Institute, Physics Dept., Estrada Nacional 10, 2685 Sacavém, Portugal

Abstract

PLASMAKIN is a package to handle physical and chemical data used in plasma physics modeling and to compute gas-phase and gas-surface kinetics data: particle production and loss rates, photon emission spectra and energy exchange rates. A large number of species properties and reaction types are supported, namely: gas or electron temperature dependent collision rate coefficients, vibrational and cascade levels, evaluation of branching ratios, superelastic and other reverse processes, three-body collisions, radiation imprisonment and photoelectric emission. Support of non-standard rate coefficient functions can be handled by a user-supplied shared library.

The main block of the *PLASMAKIN* package is a Fortran module that can be included in an user's program or compiled as a shared library, *libpk*. *pypk* is a new addition to the package and provides access to *libpk* from Python programs. It is build on top of the *ctypes* foreign function library module and is prepared to work with several Fortran compilers. However *pypk* is more than a wrapper and provides its own classes and functions taking advantage of Python language characteristics. Integration with Python tools allows substantial productivity gains on program development and insight on plasma physics problems.

1. Introduction

A plasma is the state of ionized gases and is the most pervasive state of matter in Nature: It is found anywhere from stars and interstellar gas clouds to semi-conductor processing and fluorescent lamps. It encompasses a wide range of characteristic parameters: length scale (10^{-2} – 10^{16} m); charged particle densities (10^6 – 10^{21} particles/m³); time scale (10^{-10} – ∞ s); electron temperature (10^{-1} – 10^5 eV); magnetic field (10^{-10} –5 T) and involves the interaction of different types of particles – neutral atoms and molecules, excited species, electrons and photons – between them and with electromagnetic fields³.

Modeling of plasma physics problems requires the solution of conservation equations (mass, momentum and sometimes, energy) for the species involved together with electromagnetic field equations, subject to appropriate boundary conditions. Source and sink terms in the conservation equations include the chemical interactions of these particles in a wide range of processes.

Thus, from electron kinetics studies to collisional-radiative models or large multifluid and time dependent models, whichever the numerical method used, simulation of plasma and gas discharges invariably requires the reading, classification, sorting and manipulation of particles and reactions and, frequently, the evaluation of reaction rates and power transfer

² Electronic address: npinhao@itn.pt; URL: <http://plasmakin.sourceforge.net>

³ In some problems the gravitational field has also to be taken into account.

rates.

The handling of these data frequently requires a significant amount of code, development time and effort. It is clearly advantageous to have a package able to deal with that data regardless of the number or nature of the species and chemical reactions involved and of the problem being solved or the method used. Such a package can be used as a “black box” moving the description of particles and reactions from code to a data file, thus allowing the user to concentrate on the algorithm, and once the code is developed, to easily modify and test different chemical models.

The need for a “language” to write chemical reactions and to compute the kinetic terms in a generic way is a subject with very broad application of which several approaches have been developed. Several packages have been published that are directed to specific fields - plasma physics [1]; atmospheric chemistry [2-3] - or to general purpose chemistry [4]. All of the above packages include ODE (ordinary differential equations) solvers. However, the first does not include surface reactions, the atmospheric chemistry codes do not cover the needs of plasma physics and the last, although quite complete, is a proprietary, commercial product. More recently, a Chemical Markup Language schema to describe reactions in XML has been finalized [5].

The *PLASMAKIN* package is designed to provide a framework to handle species and reactions in a way that is not dependent on a user's program, the number of space dimensions or the nature of the problem being solved. At the same time, taking into account the rather different applications that can benefit from *PLASMAKIN* and the continuous improvement of numerical algorithms, an ODE solver was not included, leaving this choice to the user.

These design options allow *PLASMAKIN* to focus on the treatment of chemical kinetics data and be useful in a large range of codes - Boltzmann equation solvers, collisional-radiative codes, fluid and hybrid codes, Monte Carlo or PIC codes, etc.

The central element of the package is a library - *libpk* - providing data reading and computational functions⁴.

The library can be extended “downwards” to support non-standard reaction rate laws by a user-supplied library.

The library has also been extended “upwards” through a Python module - *pypk* - with higher level functions and classes and allowing the development of complete Python programs.

The development of a Python module allows integration with the broad range of services provided in Python (numeric libraries; plotting services; testing frameworks; graphical user interfaces) and an increase of productivity on application development. However the most significant advantage is the perceived increase of insight into the problems.

Section 2 gives a description of the problems *PLASMAKIN* is able to address. However, only a brief survey of the physics and chemistry bases is discussed. More detailed information can be found in [6]. The following sections discuss the architectural design including the *libpk* structure and the data model, the range of properties supported, the characteristics of the *pypk* module and finish with a summary and discussion of future developments. In the Appendix we present a sample data file.

2. Plasma Physics Problems

Plasma processes occur in a wide range of conditions. To understand these phenomena, the first set of information needed are – gas density or pressure; initial temperature; and whether the process occurs at constant pressure or at constant volume.

⁴ The first version of this library was published with the name *PLASMAKIN*. However, when *pypk* was developed it was decided to reserve the name *PLASMAKIN* for the package and use the name *libpk* for the library.

Depending on the problem being studied, the description of the plasma requires the solution of some form of Maxwell's equations together with conservation equations. For the sake of clarity we will focus on the particle conservation equation⁵:

$$\frac{\partial n_i}{\partial t} + \nabla \cdot (n_i \vec{u}_i) = \mathcal{G}_i - n_i \mathcal{L}_i \quad (1)$$

where n_i is the density of particle i , \vec{u}_i the mean particle velocity, \mathcal{G}_i and \mathcal{L}_i are gain and loss terms by collisions. These have the general expression:

$$\begin{aligned} \mathcal{G}_i &= \sum_j A_j^{m,i} n_m + \sum_j k_j^{ml,i} n_m n_l + \sum_j k_j^{mlp,i} n_m n_l n_p \\ \mathcal{L}_i &= \sum_j A_j^{i} + \sum_j k_j^{im} n_m + \sum_j k_j^{iml} n_m n_l \end{aligned} \quad (2)$$

A_j^m are rates for unimolecular reactions (i.e. radiative processes) and k_j^{ml} and k_j^{mlp} rates for bimolecular and termolecular reactions, respectively. The sum index j in the first equation is on all reaction producing species i and in the second equation is on all reactions consuming species i . These reaction rates are, in general, functions of the electron or gas temperature.

To study the contribution of collisions to the energy balance in the plasma we are interested in three quantities: The power density lost (or gained) by the electrons (P_e), the power density converted to heat (P_H), and the radiated power density (P_r). The computation of each of these terms is similar to the \mathcal{G}_i and \mathcal{L}_i terms above but slightly more complex and in most cases requires the use of the reaction enthalpy, H_r . As a simple example, the radiated power density is given by $P_r = \sum_{m,n} \varepsilon_{mn} A_j^m n_m$ where the sum is on all radiative transitions of all species and ε_{mn} is the photon energy.

The reaction enthalpy is calculated as $H_r(T) = \sum_p H_f^p(T) - \sum_r H_f^r(T)$, where $H_f(T)$ is the enthalpy of formation of the reaction products and reactants, respectively. We assume a simple linear dependency on temperature, $H_f(T) = H(T_0) + C_p \cdot \Delta T$, where T_0 is the standard temperature ($T_0 = 25^\circ\text{C}$) and C_p the specific heat, which is a reasonable approximation for gases at low pressure.

Finally we may also be interested in knowing the relative contribution of each reaction for the source and loss terms both in equation (1) and on the power loss terms.

The computation of the \mathcal{G}_i and \mathcal{L}_i terms, the power loss terms, (P_e, P_H, P_r) and the relative contribution of each reaction for these terms, are the main tasks of the libpk library. For that purpose we need to consider the different properties of the chemical species and reactions involved.

2.1. Chemical Species

In a plasma, different types of species are present - atoms and molecules in different excited levels, positive and negative ions, electrons, and photons. In some applications (i.e. dusty plasmas), it is also necessary to consider bigger particles.

As we are interested in processes leading to excitation of different energy levels of the same atom or molecule, atoms or molecules in different excited levels are treated as

⁵ Both the momentum and the energy conservation equations also include collisional terms. These terms, however, are small in the momentum conservation equation and thus, frequently neglected. This is not always true for the energy conservation equation although a good number of problems can be solved without including this equation.

different chemical species.

Some of the properties needed to characterize these species are common to any species - name, energy, mass, charge, formation enthalpy - while others are meaningful only to some species or needed only for some types of reactions.

Molecular species are a special case: In some problems it is important to consider the vibrational levels of molecules. The density of these levels can be estimated from a modified Treanor distribution [7-8] provided the vibrational frequency, vibrational temperature and anharmonicity constant for those levels are known. This allows them to be considered as a group and to leave the details of each level to be handled by the library.

The full range of species properties considered can be found in [6].

2.2. Chemical Reactions

A large range of reaction processes can be handled by the library:

Unimolecular processes include radiative process, radiation imprisonment and the treatment of cascade levels.

Both forward and reverse bimolecular reactions are supported.

A large number of forward rate coefficients have an Arrhenius temperature dependence,

$$k_i = K_i T_g^{\beta_i} \exp\left(-\frac{\epsilon_i}{k_B T_g}\right) \quad (3)$$

where k_B is the Boltzmann constant, T_g the gas temperature and, K_i , β_i and ϵ_i constants characterizing the reaction.

Reactions in plasma, however, can have more complex temperature dependencies or, as is the case for electron collision reactions, depend on the electron temperature. To accommodate this, *PLASMAKIN* assumes a power series dependence on temperature in the exponential term of the rate coefficients:

$$k_i = \alpha_i^0 T^{\beta_i^0} \exp\left(\sum_{j=1}^5 \frac{\alpha_i^j}{T^{\beta_i^j}}\right) \quad (4)$$

where T is the electron temperature, T_e , for electron collision reactions and T_g for other cases. The rate coefficients are characterized by a maximum of twelve parameters $(\alpha^j, \beta^j, j = 0, \dots, 5)$.

The rate coefficient for forward and reverse reactions are related through the principle of detailed balancing. Hence, once the forward rate, k_f , is known, the reverse rate, k_r (for a two-body collision, $A + B \rightleftharpoons C + D$) is determined from

$$\frac{k_f}{k_r} = \left(\frac{\mu_r}{\mu_f}\right)^{3/2} \frac{g_C g_D}{g_A g_B} \exp\left(-\frac{\epsilon_t}{k_B T_g}\right) \quad (5)$$

where μ_f and μ_r are the reduced masses for particles in “forward” and “reverse” reactions, g the degeneracies of the energy levels and ϵ_t the energy change.

Thermolecular reactions are supported both as standard reactions and as pressure-dependent reactions where different 3rd-body species can have different reaction

efficiencies.

Finally, to account for more complex type of reaction rate coefficients, *libpk* can call a dynamic library or can be linked with a user's routine.

The full range of reactions, including surface reactions, can be found in [6].

3. *PLASMAKIN* Architecture

The *PLASMAKIN* package is composed of three units: a Fortran 95 module that becomes, after compilation, the *libpk* library, a dummy or stub routine that can be replaced by user-supplied rate coefficient equations and a Python module, *pypk*. We postpone the discussion of the Python module to section 5.

The development of the Fortran module was influenced by the application of Object Oriented methodologies to Fortran [9-10]. The programming paradigms of abstraction, information hiding, data encapsulation and function overloading guided the design of data types and routines and have simplified the development of bindings for other languages. The module subroutines are organized in five groups of tasks:

Data reading and processing:

Parse the datafile, test the correctness of the data and build species and reactions;

Inquiry routines:

Allow the user to inquire about plasma, species or reaction properties and to compute the source and loss terms for the conservation equations;

Setting routines:

Allow the user to set some of the plasma, species or reaction properties (i.e. the gas or electron temperatures. In this case all the temperature dependent rate coefficients are recomputed.)

Error processing and diagnostics:

Depending on user's choice, errors are handled either by returning an error code or printing an error message and stopping the program.

Interface routines:

These routines allow the calling of module routines from C programs solving calling conventions differences between C and Fortran such as the use of a null character in C strings and the handling of assumed-shape arrays in the Fortran routine interfaces.

All names of public procedures follow a simple convention: *pk*<action><subject>, where *pk* is used to identify *libpk* procedures, <action> is a verb or an inquiry clause (Read, Get, Set, Clean, Is) and <subject> is the data acted on by the procedure (Data, Value, Species, Reactions, ReverseReaction, Sources, PhotonEmission, PowerLosses).

Details of the procedure interfaces can be found in [6].

The dummy routine is included to allow the computation of reaction rates through non-standard expressions. The routine included in the package is just a template with the proper calling convention that a user has to adopt whenever non-standard rate expressions are needed. Ideally this routine should be compiled as a shared library (a dll for MS Windows™ users). The arguments passed from *libpk* include the index of the reaction in the datafile, arrays with the vibrational quantum number of reactant and product species, the gas and electron temperatures and the densities of species involved in 3-body reactions.

4. *libpk* Data Input

Data input files are ASCII files using Fortran NAMELIST structures to take advantage of Fortran native IO support. NAMELIST structures are annotated lists of values that provide a simple and convenient mechanism of data input. Each record is specified by a namelist name preceded by an ampersand, followed by pairs of names and values separated by the equals sign, and is terminated by a slash character. These records can appear anywhere and in any number in the file and can include comments started by a '!' character.

libpk recognizes three NAMELIST:

PLASMAKIN_DATA:

used to read plasma initial values for gas density or pressure, units and gas and electron temperatures. See the example below:

```
&PLASMAKIN_DATA Pressure=5,'mbar', Gas_n=,'cm-3', Gas_T=350,'K' /
```

In this case the gas density is computed from the pressure and temperature values and the output will be specified in cm^{-3} ;

CHEM_SPECIES:

used to input individual species or groups of species in the case of vibrational levels. The amount of data that has to be written is minimized as all properties have default values and the parser is able to deduce missing values from the information already available. Three examples are shown below:

```
&CHEM_SPECIES name='Ne', constant=T, mass=20.2, initial_conc=50,'%'/
&CHEM_SPECIES name='Ne[3P2]', energy=16.6, g=5, data_file='trp.dat' /
&CHEM_SPECIES name='A2[X,v]', constant=T, mass=70
  v=0,45, omega=1580.19,'cm-1', vib_T=1000,'K',
  anharmonicity=7.58e-3, initial_conc=50.0,'%' /
```

In the first case the species *Ne* has a default value of zero for the energy. The second species, *Ne[3P2]* inherits the mass value from its parent, *Ne*. Additional data can be read from the file *trp.dat*. In this case the library just passes the file name to the calling program. Finally in the last case we have a group of vibrational species. The library converts the group into individual species from *A2[X,v=0]* to *A2[X,v=45]* and computes the initial concentrations as referred in section 2.1 using the indicated values for *vib_T*, *omega* and *anharmonicity*.

CHEM_REACTION:

used to input individual reactions or reaction groups in the case of reactions involving vibrational levels.

```
&CHEMREACTION ! Values from V.A. Ivanov J. Phys. B 31 [1998] 1765
  reaction='e + Ne[3P2] -> e + Ne[3P1]'
  value=1.603e-6,-0.3,-6.0e2,1, units='cm3s-1' /
&CHEMREACTION ! 2025 reactions
  reaction='A2[X,v] + A2[X,w-1] -> A2[X,v-1] + A2[X,w]'
  data_file='',5 /
```

The first namelist describes a single reaction following an Arrhenius law: $k = 1.603 \times 10^{-6} \cdot T_g^{-0.3} \exp\left(+\frac{6.0 \times 10^2}{k_B T_g}\right)$. The second example represents a group of

reactions with reaction rates obtained from the 5th equation of a user's routine. Taking into account the combinations of vibrational numbers in the example above, this reaction group symbolizes 2025 reactions.

5. *pypk*: The Python Module

The most recent addition to the *PLASMAKIN* package is a Python wrapper based on the *ctypes* foreign function library.

The development of a Python module serves several purposes:

1. It allows a broader audience to use the library since it does not require knowledge of Fortran;
2. It integrates the library with the rich set of Python modules. Of special importance to plasma modeling is the integration with other modules for scientific computing as `Scipy` [11] or `Matplotlib`; and
3. It allows the use of Python development tools as `doctest` or `unittest` that have proved useful to track errors in the library itself.

The module is organized into three levels:

1. The lower level comprises the loading of the shared version of *libpk* and function definitions to call the library C-compatible routines.

These function definitions depend on the compiler used to link the library as each compiler uses its own convention to assign external names of module entities. For instance **gfortran** uses the convention `__module-name__entity-name` where `module-name` and `entity-name` are in lowercase, while **g95** uses `module-name_MP_entity-name` and **Microsoft Fortran** uses `_MODULE-NAME_mp_entity-name`, with `module-name` in uppercase. This requires an initial inquire of which Fortran compiler is installed and the import of submodules with the correct function definitions for each compiler. Presently the **gfortran** and **g95** compilers are supported and a submodule for the **Intel Fortran** compiler is being developed.

The arguments and return values of these functions are `ctypes` data types. While in Fortran the public subroutines have overloaded interfaces that accept different types of data, `ctypes` do not include any mechanism to support a similar behavior. Thus the number of functions defined had to be increased. These routines are not meant to be called directly, thus all the routines names start with two underscore characters;

2. The second level are functions that use only Python built-in types as function arguments or as return values. These functions call the lower level functions to access the *libpk* library. The naming convention is similar to the equivalent library subroutines (`<action><subject>`). A closer similarity can even be obtained if the Python module is imported as `pk` (making the function calls `pk.<action><subject>`). The functions have the following characteristics:

1. They accept different type of data allowing the recovery of the level of abstraction of the Fortran routines that had been lost with the first level functions;
 2. The argument list is simplified. In the Fortran library most of the procedures are subroutines, returning values in the argument list. Because in Python we deal with functions, all the arguments in Fortran that are used only to return values have been removed from the argument list or, when these arguments were optional arguments, they have been transformed into logical flags;
 3. They use as much as possible, arguments with default values which further simplify the use of these functions;
 4. Have proper exception handling.
3. The higher level comprises class definitions. The following classes are defined:

PK

- a class to keep the information on global properties. The values of the gas temperature (`GasTemp`), electron temperature (`eTemp`) and gas density (`GasN`) can be set. All the other properties are fixed. In this case, an attempt to assign a new value raises an exception.

Species

- a class representing the chemical species. The density (`n`) or concentration can be changed by the user (changing also the values in the library) but all

other properties have fixed values. An attempt to change these values raises an exception;

Reaction

- a class representing reactions. The value of the reaction coefficient can be changed. This class defines a method `Update()` to update the value of the rate coefficient. This is necessary because if the value of rate coefficients in the library are changed, i.e. due to a change in temperature, the only way to update the corresponding values in the class instance is to call this function;

Phys_Property

- a helper class to describe a physical property, containing a float value and a string for the units;

Data_Column

- a helper class to hold the string for a filename and an integer index.

6. A Sample Case

As an illustration of the usage of this package we study the passage of an electron swarm through a gas.

The electron density has a Gaussian profile and along their passage, they excite and ionize the gas, producing several chemical species. These species take part in several reactions among themselves and with the electrons, and are transported either by diffusion or advection to the walls where they are neutralized or de-excited.

Here we consider an hypothetical gas with diffusion coefficients and rate coefficients adjusted for this example.

The species density is obtained from the solution of a system of ordinary differential equations.

The listing of the program can be found in Appendix A. To solve this problem *pypk* is used together with *scipy* and *pylab* for computational and plotting services, respectively.

The calls to *pypk* are limited to a few lines:

- The loading of the module in line 20;
- In line 24 the datafile with the description of species and reaction (see Appendix B) is read;
- In lines 25 - 27 we define instances of the classes `PK` and `Species`; and
- In line 47 all the “magic” is done when the G_i and L_i terms are computed.

Changing the values in the datafile and running the again the program allows a quick study of the influence of several parameters - gas density, reactions included in the model, values of the rate coefficients. The result of one simulation is shown in Figure 1.

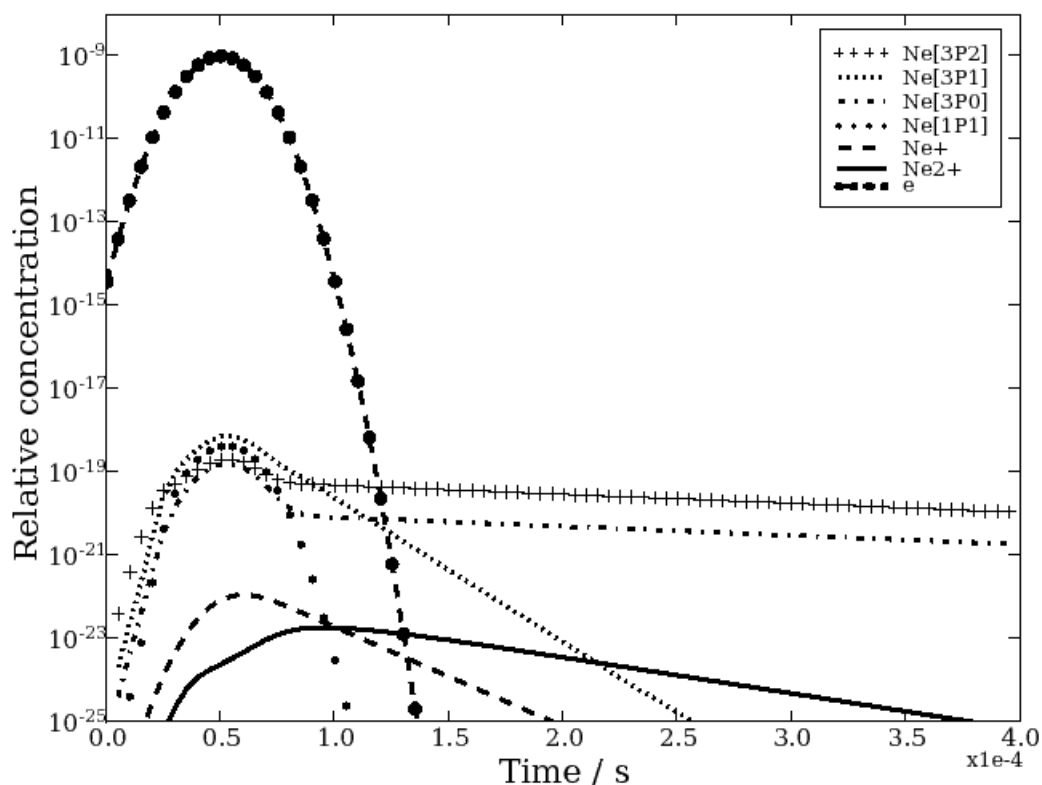


Figure 1: Time dependency of species formed during the passage of an electron swarm through a gas.

7. Conclusions

A Python extension module has been developed to access the *libpk* library and enriching the *PLASMAKIN* chemical kinetics package.

The Python module, however, is not just a wrapper around the library and adds a “Pythonic” way of problem solving to the analysis of plasma physics problems.

Taking advantage of other Python modules for debugging, numerical computation and data analysis and representation is possible to build programs quickly and reliably. The flexibility introduced by this module allows the users to concentrate on the chemical kinetics and helps to gain insight on the problems.

Further work will continue both on the extension of the *libpk* library and the *pypk* module. Future plans include the simulation of photon emission spectra taking into account line broadening; the introduction of a database for species properties and reactions, and the migration of the datafile format to XML.

Appendix

A. Sample code

```
1 #!/usr/bin/env python
2
3 """Example showing the use of pypk.py: Electron pulse in a gas.
4
```

```

5 Description :
6     This program evaluates the interaction of a short electron pulse in an ideal gas.
7     The distribution of the electron pulse in time has a Gaussian profile with tMax=5.e-5 s
8 and Std Dev= 1.e-5 s.
9     The electron pulse excites and ionizes the gas producing several species.
10    These species react through several processes and drift or diffuse to the walls, where they are
11 neutralized or deexcited. The values of rate coefficients are not realistic as sometimes they are too
12 high or too low comparing with the range of values found in real gases but serve the purpose of testing
13 the pypk module.
14    Changing the values in this file or in the data file the user can quickly see the results.
15 In this way this programs serves a pedagogical purpose.
16
17 Owner: N. Pinhao, ITN – Physics Dept . – PORTUGAL
18 Date : June 2007""""
19
20 import pypk as pk
21 from scipy import *
22 from pylab import *
23
24 pk.ReadData('test.dat')
25 gas = pk.PK()
26 gas.eTemp = 10.                # mean electron energy = 10 eV
27 sp = [ pk.Species(i) for i in range(gas.NnC, gas.NnTV+gas.NnC) ]
28
29 # Time
30 dt = 5.e-6; t = arange(0.0, 4.e-4, dt)
31
32 # Initial density for non-constant species, except electrons
33 n0 = array( [sp[i].n for i in range(gas.NnTV- 1)] )
34
35 # Electron density
36 ne = stats.norm.pdf( t , 5.e -5 ,1.e -5)
37 norm = max(ne) * 1.e3
38 ne = where( ne>1e-20, ne/norm, 0 )
39
40 # Artificial diffusion/advection for conservation equations
41 Dv = array( [5.e+3 ,5.e+3 ,5.e+3 ,5.e+3 ,4.e+4 ,2.e+4] )
42
43 def dndt(y,x) :
44     net = gas.GasN*stats.norm.pdf( x, 5.e-5, 1.e-5)/norm
45     net = net > 1e-20 and net or 0
46     tt = list(y); tt.append(net)
47     SrC , SrP = pk.GetSources(tt)
48     return array(SrC [: gas.NnTV-1]) - y * (array(SrP[: gas.NnTV-1]+Dv ))
49
50 # Integration of the ode system
51 z = integrate.odeint( dndt, n0, t, h0 =1.e-10)
52
53 # Plot the results
54 lines = [ 'k+', 'k:', 'k-.', 'k.', 'k-- ', 'k-', 'ko--' ]
55 for i in range(gas.NnTV-1):
56     semilogy( t, z[:, i] / gas.GasN, lines[i], label = sp[i].name, lw =2)
57
58 # ...add the electrons scaled by 1e-6
59 semilogy( t, ne/1.e6, lines[ gas.NnTV-1], label = sp[gas.NnTV-1].name, lw =2)
60 ylim(( 1.e-25, 1.e -8))
61 xlabel( 'Time / s' , size =16); xticks( size =12)

```

```

62 ylabel( 'Relative concentration', size =16); yticks(size =12)
63 legend()
64 savefig( 'test.eps' )
65 #show ( )

```

B. Sample Datafile

```

! *****
!
!                               test.dat
! Datafile for the Python example program
! *****

&PLASMAKIN_DATA Pressure=10., 'mbar', Gas_n=, 'cm-3', Gas_T=350, 'K'/

! 1. Gas species
&CHEM_SPECIES name='Ne', constant=T, mass=20.18, initial_conc=100, '%'/
&CHEM_SPECIES name='Ne[3P2]', energy=16.61, g=5, data_file='NeTransp.txt'/
&CHEM_SPECIES name='Ne[3P1]', energy=16.67, g=3/
&CHEM_SPECIES name='Ne[3P0]', energy=16.71, g=1/
&CHEM_SPECIES name='Ne[1P1]', energy=16.85, g=3/
&CHEM_SPECIES name='Ne[3p]', energy=18.38, cascade=T/
&CHEM_SPECIES name='Ne[3pM]', energy=18.97, cascade=T/
&CHEM_SPECIES name='Ne[4s]', energy=19.66, cascade=T/
&CHEM_SPECIES name='Ne+', energy=21.56, charge=+1/
&CHEM_SPECIES name='Ne2+', mass=40.36, charge=+1/
&CHEM_SPECIES name='photon', v=1,13/
&CHEM_SPECIES name='e', charge=-1 /

! 2. Electron excitation and ionization
! 2.1. Excitation of 3s levels
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P2]', value=5.e-4, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P1]', value=3.e-5, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3P0]', value=5.e-6, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[1P1]', value=1.e-6, units='cm3s-1'/

! 2.2. Excitation of 3p and 4s+upper levels
&CHEM_REACTION reaction='e + Ne -> e + Ne[3p]', value=5.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[3pM]', value=3.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne -> e + Ne[4s]', value=1.e-7, units='cm3s-1'/

! 2.5. Ionization
&CHEM_REACTION reaction='e + Ne -> 2*e + Ne+', value=1.e-8, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P1] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P0] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[1P1] -> 2*e + Ne+', value=1.e-7, units='cm3s-1'/

! 2.3. s_j-s_i transitions
&CHEM_REACTION reaction='e + Ne[3P2] -> e + Ne[3P1]'
value=1.603e-6,-0.3,-6.0e2,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P1] <-> e + Ne[3P0]'
value=3.1e-8,, -5.176e3,1 units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] <-> e + Ne[3P0]'
value=8.2e-9,, -1.118e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P2] <-> e + Ne[1P1]'
value=5.e-9,, 2.658e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P1] <-> e + Ne[1P1]'
value=5.e-9,, 2.054e3,1, units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne[3P0] <-> e + Ne[1P1]'
value=2.3e-7,, 1.543e3,1, units='cm3s-1'/

! 3. Radiative transitions

```

```

! 3.1. 3s radiative levels
&CHEM_REACTION reaction='Ne[3P1] -> Ne + photon1', value=0.486e8/
&CHEM_REACTION reaction='Ne[1P1] -> Ne + photon2', value=6.11e8/
! 3.2 4s radiative levels
&CHEM_REACTION reaction='Ne[4s] -> Ne + photon11', value=1.21e8/

! 3.3. Radiation imprisonment
&CHEM_REACTION reaction='Ne + photon1 -> Ne[3P1]', value=1.539e-3/
&CHEM_REACTION reaction='Ne + photon2 -> Ne[1P1]', value=1.746e-3/
&CHEM_REACTION reaction='Ne + photon11 -> Ne[4s]', value=1.0e-3/

! 3.4. Cascade reactions
&CHEM_REACTION reaction='Ne[3p] -> Ne[3P2] + photon3', value=9.24e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P2] + photon4', value=6.128e7/
&CHEM_REACTION reaction='Ne[3p] -> Ne[3P1] + photon5', value=6.722e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P1] + photon6', value=9.397e7/
&CHEM_REACTION reaction='Ne[3p] -> Ne[3P0] + photon7', value=1.691e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[3P0] + photon8', value=2.49e7/
&CHEM_REACTION reaction='Ne[3p] -> Ne[1P1] + photon9', value=6.689e7/
&CHEM_REACTION reaction='Ne[3pM] -> Ne[1P1] + photon10', value=9.208e7/
&CHEM_REACTION reaction='Ne[4s] -> Ne[3p] + photon12', value=1.034e7/
&CHEM_REACTION reaction='Ne[4s] -> Ne[3pM] + photon13', value=1.434e7/

! 4. Heavy species kinetics
! 4.1 Pooling reactions
&CHEM_REACTION reaction='2*Ne[3P2] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P1] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P0] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[1P1] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P1] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[3P0] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[1P1] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P0] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P0] + Ne[1P1] -> Ne + Ne+ + e'
value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[1P1] -> Ne + Ne+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P2] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P1] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[3P0] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P2] + Ne[1P1] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[3P0] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P1] + Ne[1P1] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[3P0] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/
&CHEM_REACTION reaction='Ne[3P0] + Ne[1P1] -> Ne2+ + e', value=3.2e-10,
units='cm3s-1'/
&CHEM_REACTION reaction='2*Ne[1P1] -> Ne2+ + e', value=3.2e-10, units='cm3s-1'/

! 3.2. Molecular ion formation
&CHEM_REACTION reaction='Ne+ + Ne + M -> Ne2+ + M' value=3.5e-31, units='cm6s-1'/

! 4. electron-ion recombination

```



```

&CHEM_REACTION reaction='e + Ne2+ -> Ne[3p] + Ne', value=1.338e-6,-0.67,
units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3pM] + Ne', value=1.338e-6,-0.67,
units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P2] + Ne', value=6.693e-7,-0.67,
units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P1] + Ne', value=6.693e-7,-0.67,
units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[3P0] + Ne', value=6.693e-7,-0.67,
units='cm3s-1'/
&CHEM_REACTION reaction='e + Ne2+ -> Ne[1P1] + Ne', value=6.693e-7,-0.67,
units='cm3s-1'/

```

Bibliography

- [1] S. A. Roberts, PLASKEM, *Comp. Phys. Commun.* **18** (1979) 363
- [2] C. J. Aro, CHEMSODE, *Comp. Phys. Commun.* **97** (1996) 304
- [3] G. D. Carver, P.D. Brown and O. Wild, ASAD, *Comp. Phys. Commun.* **105** (1997) 197
- [4] R. J. Kee, F.M. Rupley, E. Meeks and J.A. Miller, CHEMKIN - III, Sandia Nat. Laboratories, 1996
- [5] Gemma L. Holliday, Peter Murray-Rust, and Henry S. Rzepa *J. Chem. Inf. Model.* **46(1)** (2006) 145 - 157
- [6] N. Pinhao, *Comp. Phys. Commun.* **135** (2001) 105
- [7] C. E. Treanor, J. W. Rich and R. G. Rehm, *J. Chem. Phys.* **48** (1968) 1768
- [8] B. F. Gordiets, S. S. Mamedov and L. A. Shelepin, *JETP* **40** (1972) 640-646
- [9] C. D. Norton, Object Oriented Programming Paradigms in Scientific Computing, PhD thesis,
Rensselaer Polytechnic Institute, Troy, New York, August 1996
- [10] C. D. Norton, B. K. Szymanski and V. K. Decyk, *Communications of the ACM*, **38(10)** (1995) 88-100
- [11] E. Jones, T. Oliphant, P. Peterson et al. SciPy: Open Source Scientific Tools for Python, (2001-)
URL: <http://www.scipy.org>

The When and How for Design Patterns

Olemis Lang

Keywords: Design patterns, Python, OOP, concurrency, object-oriented design methods and methodologies, analysis patterns, software architecture, operating systems, process synchronization, software process models.

Previous articles⁶ have been devoted to portray Design Patterns together with useful guidelines to code them in Python. However, the former arguments could be complemented by displaying other important patterns and revealing that they can also solve large-scale issues. Also they can be part of a design strategy and it would be nice to put them in context with design methods. In this respect, it is very important to know when and how they are discovered.

This article is a first attempt to identify when design patterns come into play during system analysis and design, and also suggests how to translate their structure into Python. This text is written from the perspective of an author who applies patterns since before discovering this programming language. Readers will find hereinafter some illustrative examples where patterns go beyond the traditional class relationships. They can contribute to get a deeper insight into patterns even though the original scenarios were not coded in Python. That's because design patterns are higher-level abstractions concerned with class structures and its relationships, instead of how coders specify them by means of a particular programming language. Besides, further arguments are provided to demonstrate that Python is full of patterns, since they are comprehensively mentioned throughout Python Enhancement Proposals (PEP).

Although a lot of patterns have been discovered up to this date, our discussion remains limited to those included in the GoF catalog. It is believed that there is no need to be exhaustive in this respect to achieve the goals of this paper. Therefore, forthcoming articles shall address those left out.

Design Patterns illustrated

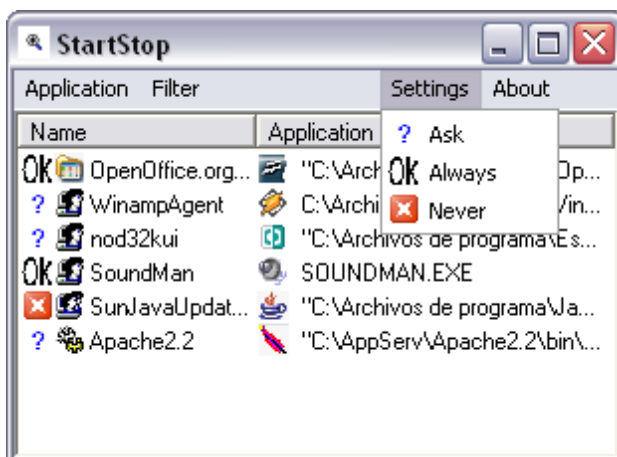
For some of us design patterns were among the greatest discoveries of our undergraduate studies. A fuller understanding of these patterns is achieved after considering them during the design of applications and systems. For this reason we start by analyzing some concrete scenarios, where they are not applied *ad-hoc* and interact with different design artifacts.

The StartStop application

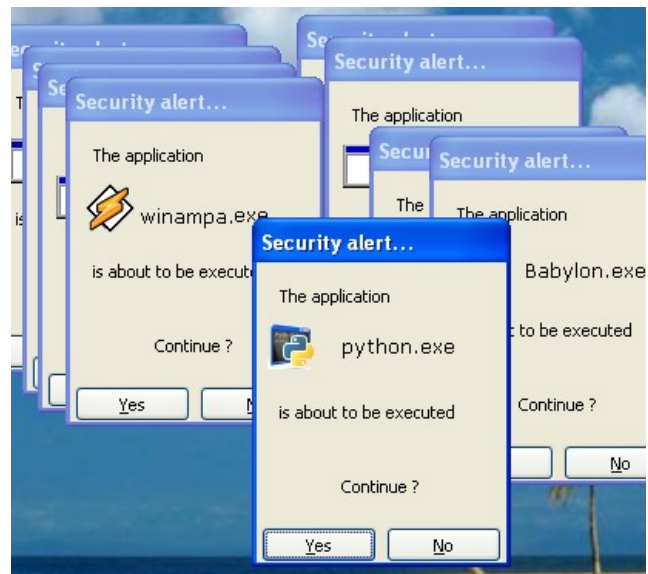
Let's first consider an application similar to the product *StartStop*⁷, whose main goal is to control the execution of applications once users start Windows sessions (thus possibly preventing virus infection). As may be seen in Figure 1a, the user can list the applications executed when he (she) starts a session in the local computer and for each one specify one of three actions: a) always execute (i.e. in case of a critical process or a "healthy" executable file), b) ask for confirmation (the default choice if a new application, perhaps infected by a virus, has not been put into another state), or c) never execute (i.e. a suspicious file or potentially useless application).

⁶ T. Leeuwenburg, "Programming Idioms pt 2 -- Design Patterns" (2007) *The Python Papers*, vol. 2, no. 1, pp 5:7

⁷ J.L. López "StartStop. Un vigía imprescindible al iniciarse Windows" (2002) *VSantivirus* no. 6 vol 542.



a) Main form



b) Multiple asynchronous confirmations

Figure 1: The StartStop application

Once a new session is started, the StartStop application intercepts the execution of each target program in a separate process and performs the requested action. Therefore, it is concerned with “controlling the execution of another process under certain conditions”, and so acts like the “protection proxy” mentioned in the applicability of the *Proxy* pattern⁸. Figure 1b shows what happens when several of these processes require a confirmation. Any user might be panicked by the amount of dialogs shown, or even think that a critical error has occurred. A successful approach is to show confirmation dialogs one by one as the user approves or rejects process execution; in other words, “ensure that only one instance of the confirmation dialog class is shown to the user, and provide a global point of access to it to retrieve the user’s choice”. Notice the match between the prior statement and the intent of the *Singleton* pattern. But, what happens if the structure proposed by the GoF⁸ is coded as-is to solve this “problem”? As the singleton instances reside in separate processes (i.e. address spaces) there is indeed a single dialog per application, but each one is still shown. So nothing has changed.

The code in Listing 1 describes a simple and correct implementation of the *Singleton* for Windows operating systems based on process synchronization. The first lines of code in Listing 1a import some modules and bind into the local name space the functions to manipulate mutexes. In Listing 1b a named mutex is acquired before each singleton dialog instance is “created” (within the `__new__` method) and is released when they are destroyed (or the application exits). Therefore there is no overlapping between the lifetimes of two different instances; hence multiple dialogs cannot be displayed at the same time. This implementation differs from the traditional structure because the “feeble” encapsulation supported by Python does not allow hiding of the initialization process. This implies that no “static” (or class) method is needed to access the singleton instance, but also motivates the introduction of a flag (`_doInit`) to prevent initializing the singleton instance more than once.

⁸ E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” (1995) Reading, Mass.: Addison-Wesley. ISBN 0-201-63361-2.

```

1  import ctypes, wx, types
2
3  kernellib = ctypes.windll.kernel32
4  CreateMutexA = kernellib.CreateMutexA
5  CloseHandle = kernellib.CloseHandle
6  ReleaseMutex = kernellib.ReleaseMutex
7  WaitForSingleObject = \
8      kernellib.WaitForSingleObject
9
10 APP_TITLE = 'StartStop confirmation...'
11 LCKNM = 'StartStop'
12
13 SL_ALWAYSEXEC, SL_NEVEREXEC,
14 SL_ASK = \
15     range(3)
16
17 class StartStopCheckApp(wx.App):
18     def main(self):
19         choice = self.getSecLevel()
20         if choice == SL_ALWAYSEXEC:
21             self.execTarget()
22         elif choice == SL_ASK:
23             dlg = StartStopDlg()
24             if dlg.ShowModal() == wx.ID_YES:
25                 self.execTarget()
26
27     def execTarget(self):
28         """ Execute the 'suspicious' application """
29     def getSecLevel(self):
30         """ Determine what to do when an
31             application is executed at boot time.
32
33             Returns one of SL_ALWAYSEXEC,
34             SL_NEVEREXEC, SL_ASK """
35
36 if __name__ == '__main__':
37     StartStopCheckApp().main()

```

a) Proxy application (process)

```

37 class StartStopDlg(wx.Dialog, object):
38     """ wxPython frame instantiated only once in the
39         context of the local host computer. """
40
41     _inst, _doInit, _hLock = None, True, None
42
43     def __new__(cls, parent= None, id= -1, \
44                 title= APP_TITLE, pos= (100, 100), \
45                 sz= wx.DefaultSize, style= wx.CAPTION):
46
47         if cls._inst is None:
48             cls._hLock = CreateMutexA(None, False, LCKNM)
49
50             # Acquire the lock... wait until "forever"
51             WaitForSingleObject(cls._hLock, -1)
52             cls._inst = wx.Dialog.__new__(cls, parent, \
53                                           id, title, pos, sz, style)
54
55         return cls._inst
56
57     def __init__(self, parent= None, id= -1, \
58                 title= APP_TITLE, pos= (100, 100), \
59                 sz= wx.DefaultSize, s= wx.CAPTION):
60
61         if not self.__class__.__doInit:
62             return
63         wx.Dialog.__init__(self, parent, id, title, pos, sz, s)
64         self.buildLayout()
65         self.__class__.__doInit = False
66
67     def buildLayout(self):
68         """ Incorporate child windows """
69
70     def __del__(self):
71         ReleaseMutex(self.__class__.__hLock)
72         CloseHandle(self.__class__.__hLock)
73         self.__cls__.__inst = None
74         self.Destroy()
75         wx.Dialog.__del__(self)

```

b) Synchronized Singleton

Listing 1: Implementing the StarStop application with wxPython.

To use the *Singleton* pattern in the former example perhaps is not accurate enough since the desired functionality should be established in a per-user basis. Otherwise, users remotely logged in could experience some troubles. A more accurate solution could be to apply the *Parametric Singleton*⁹ pattern. In this case, local users would be the parameter restraining the creation of dialog instances. The choice presented was preferred because the GoF pattern is more familiar to the audience. This decision does not spoil the underlying idea in any sense. We have seen so far design patterns solving problems in which multiple processes (i.e. architectural artifacts) are involved. Since many libraries and frameworks define classes to handle threads, they could also be mapped to a participant in a pattern.

⁹ D. Lyon and F. Castellanos "The Parametric Singleton Design Pattern", in *Journal of Object Technology*, vol. 6 no. 3, March - April 2007, pp. 13-23
http://www.jot.fm/issues/issue_2007_03/column2.

Printing formatted documents

The second example is concerned with printing a formatted document by setting up a dot matrix printer in text mode. In this mode all the features (e.g. different page formats and font styles) are set up by sending control codes to the printer through a parallel interface. In Python this can be done either by supplying control commands to Linux *parport* driver through the function `fntcl.ioctl`, or by calling the Win32 API function `DeviceIoControl` made available after using the `ctypes.windll` loader, or otherwise by means of file operations.

The main issues to handle during the design are the representation of the document structure and the means to perform basic (e.g. bold, italics, different page formats and font styles) and/or complex (e.g. hyphenation and text justification) formatting on the output.

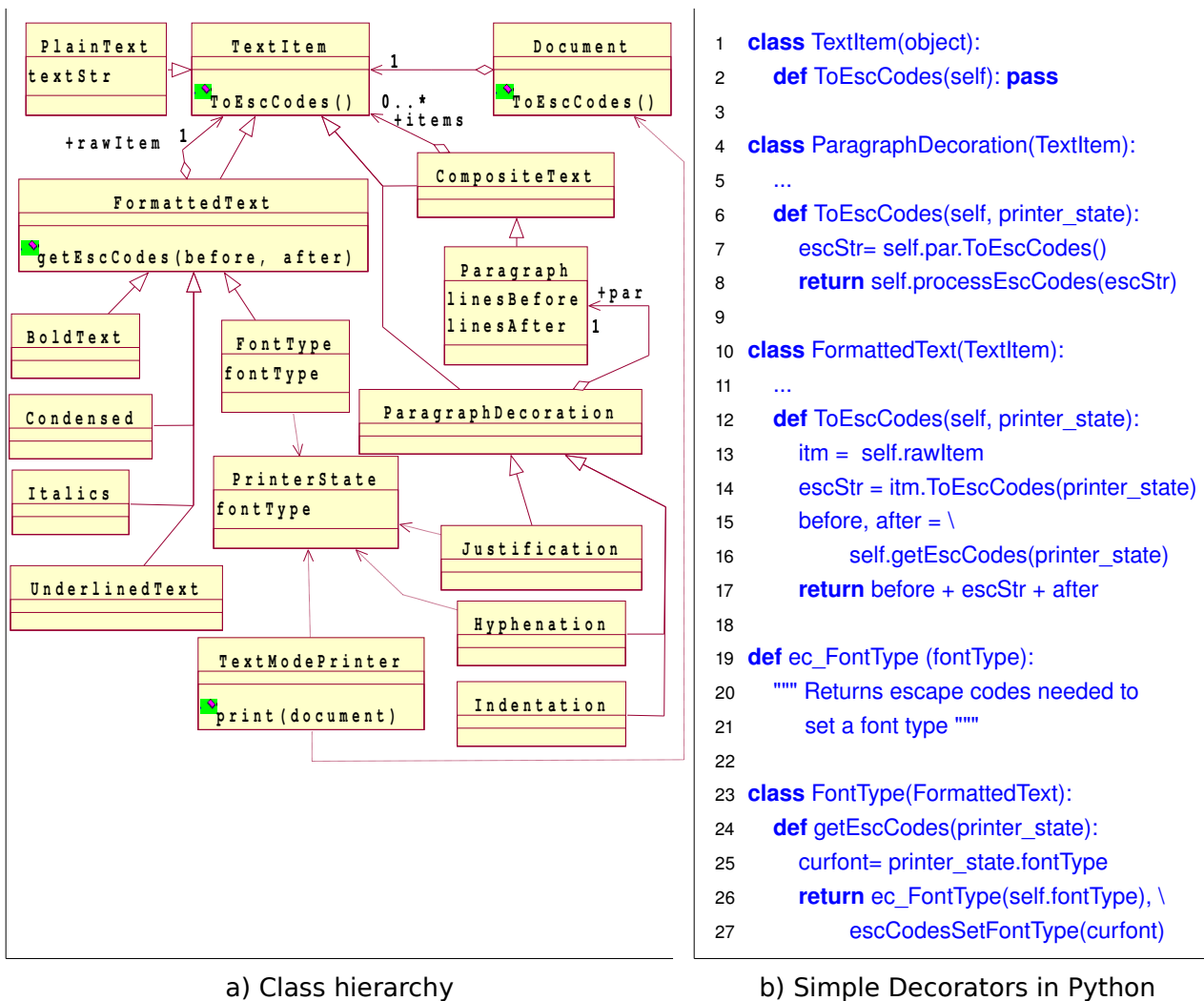


Figure 2: Printing formatted documents with text-mode printers.

The first issue is related to “composing objects into tree structures to represent documents containing paragraphs, which in turn contain pieces of plain text or text formatted in several ways.” Formatting operations require “accessing sequentially the elements in the resulting hierarchy and treating them uniformly in order to print the contents of the document without regard of their underlying representation.” This leads to the idea of using the patterns *Composite* (e.g. classes `CompositeText` and `Paragraph` in Figure 2a), *Iterator*, and perhaps *Visitor*. However, a more subtle problem must be solved in the later case. Several adornments can apply over the same text (e.g. bold, italics, and Courier fonts) or paragraph (e.g. justification, hyphenation, indentation). Thus the number of different behaviors to be supported by the classes involved is proportional to the number of combinations in which

be obtained. Other useful paragraph decorators perform hyphenation and paragraph justification. They employ the same basic algorithm, but introduce different characters at different places into lines. Figure 3 shows an example text and its representation (object tree). It highlights how these embellishments can be deeply nested within each other.

Scheduling tasks in SCADA systems

Another application of design patterns occurs when implementing a SCADA system¹¹. Its main users are Operators who are responsible for the performance of industrial processes. An important requirement to fulfill is the automation of the actions performed in response to some important events (e.g. measure process variables in regular time intervals, remote notification of critical events, complex system recovery strategies). That's why a subsystem should be liable for scheduling tasks.

Figure 4a shows the way Operators define tasks, and how the scheduling mechanism works. Therein the messages 13 to 15 reveal the *Observer* pattern. They are exchanged several times, as long as the source subject (labeled *src* in the diagram) triggers the specified event (labeled *ev*). Further refinements during system design transform *Task* (entity class) into the *Task Scheduling* subsystem. Likewise *Subject* (control class) can be traced to several subsystems which extend the former core functionality by implementing a common interface. For example, the *Timer* subsystem generates periodical events. If it is used together with the *Process* subsystem, process variables could be measured in regular intervals. The later can also generate events once one such measurement exceeds operational limits, and in turn generate different types of notifications like e-mails –*Mail* subsystem–, messages sent over wireless networks –*Communications* subsystem– or sound alarms –*Alarms* subsystem–. Moreover, the *Mail* and *Web* subsystems could notify incoming messages or HTTP requests sent by Operators. As a result the *Process* subsystem could act upon the target process. Such a structure is known as a micro-kernel architecture¹².

11 O. Lang, "Proceso de construcción de Consola de Control Local para subestaciones de distribución de energía eléctrica" (2007) *III Taller de Realidad Virtual. Informática 2007*.

12 F. Buschmann et al. "Pattern-Oriented Software Architecture: A System of Patterns", Vol. 1, John Wiley & Sons, ISBN: 0471958897.

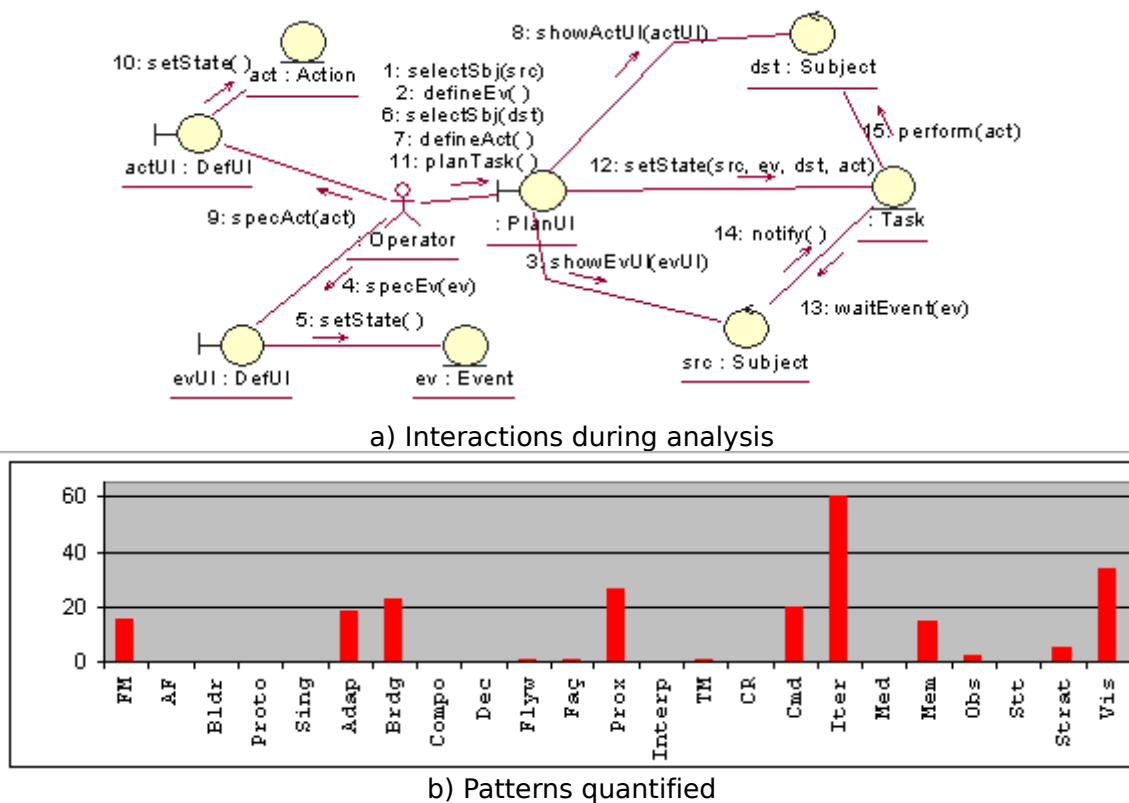


Figure 4: Patterns in the Scheduling subsystem.

The main responsibility of the scheduling subsystem is actually to “define a one-to-many dependency between different Subjects so that when one such entity triggers an event matching a task definition, all its dependent Subjects are notified and react accordingly.” Also the role of the [Subject](#) abstraction can be to “standardize an interface to relate the triggering of multiple events with performing actions as to decouple the required operations from its implementation so that the two can vary independently. Event/action associations shall be selected or switched at run-time. Besides the behavior needed to capture new events and perform new actions must be hidden to the scheduling subsystem in order to add it dynamically at run-time without changing the core task monitoring implementation.” Compare the former statements with the intent and applicability sections of the *Observer* and *Bridge* patterns, respectively. Notice that although the relationships among subsystems can be described using the former patterns, their implementation introduces many others.

Python is full of Patterns

It is incredible the job done by the community of pythonistas in order to build an extremely dynamic language upon a small set of basic concepts. And how is it possible? Do patterns contribute to this success? Several PEPs reveal that this language and its standard modules incorporate plenty of patterns. Let’s cite some relevant examples:

- The standard module [weakref](#) and PEP 205 “Weak References” introduce a mechanism suitable for implementing the *Flyweight*⁸ pattern via such references. Therein some surrogates act like weak smart references (*Proxy* pattern⁸) which attempt to behave like the object they refer to.
- The iterator protocol for Python described in PEP 234 “Iterators” and its enhancements¹³ are based on the pattern of the same name identified by the GoF⁸.
- The special methods `__copy__` and `__deepcopy__` introduced by the standard module [copy](#)

¹³ G. van Rossum, P.J. Eby “Co-routines via Enhanced Generators” PEP 342.

resemble the structure of the *Prototype* pattern⁸. This module also solves some issues identified for this pattern, e.g. full support to customize the clone operation at any point throughout the class hierarchy, provisions for default cloning semantics for “any imaginable class on earth”, and resolution of the shallow versus deep copy dilemma.

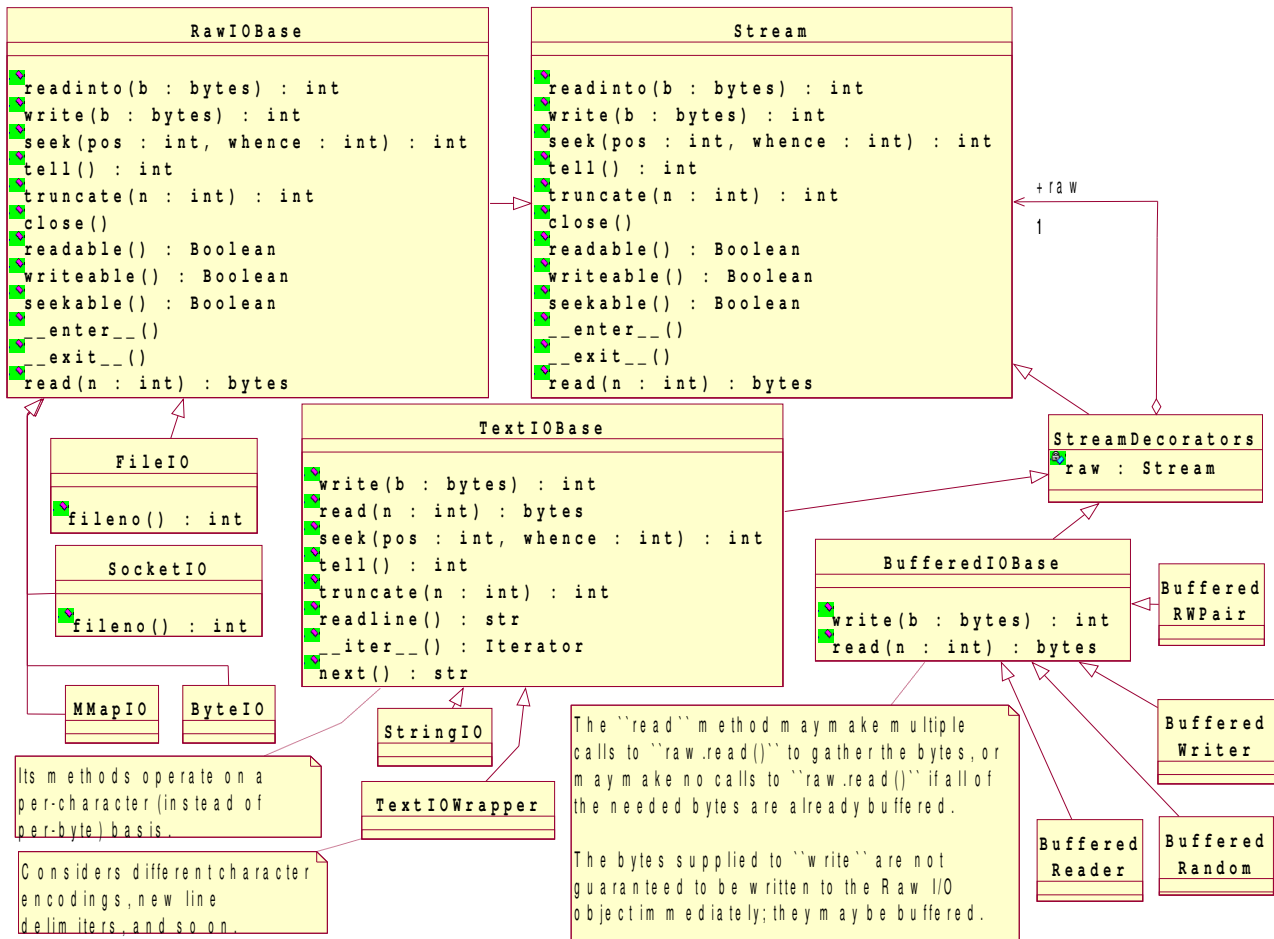


Figure 5: Overview of the Python I/O Library proposed in PEP 3116¹⁴.

- The Python I/O Library¹⁵ envisioned for Python 3000¹⁶ defines layers of stream-like objects. Therein the classes `BufferedIOBase` and `TextIOBase` follow the *Decorator* pattern⁸.
- The hierarchy of `Logger` instances defined by the `Manager` class at the standard module `logging` (depicted in PEP 282 “A Logging System”) is a peculiar way for structuring the *Chain of Responsibility* pattern⁸. The same pattern is followed by the multiple `Filter` instances that can be installed on `Logger` and `Handler` objects. Moreover `LogRecord` objects are simple examples of the *Command* pattern⁸.
- In PEP 302 “New Import Hooks” candidate importer objects in the list bound to `sys.path_hooks` are asked whether they can handle a `sys.path` item, until one is found that can. Likewise, `sys.meta_path` provides a similar mechanism for importers that don't need any entry on `sys.path` (e.g. built-in and frozen modules). In both cases we are in presence of the *Chain of Responsibility* pattern⁸.

These examples have been discovered after examining standard libraries, but further

¹⁴ Classes are not modeled exactly as in PEP 3116. An equivalent layout is presented instead in order to make clear the applicability of the pattern.

¹⁵ D. Stutzbach, M. Verdone, G. van Rossum “New I/O” PEP 3116 at <http://www.python.org/peps/pep-3116.html>.

¹⁶ G. van Rossum “Python 3000” PEP 3000 at <http://www.python.org/peps/pep-3000.html>.

examples could be found if either third party non-standard modules are also inspected or patterns beyond the GoF catalog were identified as well.

So... what's next?

After considering the former situations, some relevant conclusions arise. The first thing involved in identifying design patterns is to properly determine the problems the design should solve, the context where its found and the requirements to support. These problems might be complex. If this is the case they shall be successively decomposed into simpler ones until atomic statements are found. Later, these atomic statements should be reformulated trying to make them match the intent or applicability of different design patterns. This process can lead to several potential solutions, so each one should be assessed considering the interactions with other problems, supporting features and project constraints. As should be seen, design patterns are not a straitjacket, but a template that contributes with useful guidelines when solving a frequent problem. So while conducting the prior steps, bear in mind that singular requirements might imply further enhancements to the pattern structure.

Moreover, Figure 4b shows that consecutive refinements made to a system can bring in new patterns. Therefore all the major components of a system can be arranged following a particular pattern, but once they are implemented another design pattern(s) can be widely applied. It is important to confess that the patterns *Iterator*, *Visitor*, *Adapter*, *Proxy* and *Bridge* in that order, have been those mostly applied by the author to several projects of different nature. This statement is supported by measurements stored in a Measurements Repository. They were performed by considering stable versions of the deliverables.

The previous discussion may seem vague but the fact is that pattern selection and application can be seen like the art coders and system designers should become experts in. Therefore straight answers are either complex to fulfill or yield in generic concerns. More effective arguments can be obtained considering the notion of *pattern systems*. They put together collections of patterns and state how they relate with each other, offering guidelines for their implementation and how to combine them in order to satisfy the more complex requirements demanded for specific applications. They also organize the constituent patterns in order to speed up the process of finding those suitable for the situation at hand. Nowadays, several such systems have been crafted¹². Nevertheless, since software gets done in many different ways, software teams should adequate them according to their needs, possibilities and skills.

Furthermore, the previous examples show how the relationships established between architectural artifacts (e.g. system processes in the run-time architecture, interfaces and subsystems) look like design patterns or interact with them. Regularly design patterns are abstractions lying between architectural patterns and programming idioms (e.g. classes, algorithmic features, coding styles), but sometimes either major system components can be arranged according to their structure, or they support idioms present in programming languages (e.g. Python built-in iterator protocol, the for statement, list comprehensions, and prototyping by means of the standard module [copy](#)). So two questions arise... Is there anything else interacting with design patterns? When are design patterns discovered?

The second question is the most difficult to answer because it is not only concerned with the nature of patterns but also with how software gets done by teams of individuals¹⁷. If the development process is chaotic it could be almost impossible to know when patterns will be discovered. Besides, the development process determines the relevant support artifacts and also can promote the early depiction of an architecture (e.g. RUP¹⁸) or not (e.g. Extreme

¹⁷ O. Lang, "Emergencia de los patrones de diseño en las arquitecturas de software con ayuda de RUP" (2007) *III Taller de Calidad de Software, Informática 2007*, ISBN 978-959-286-002-5.

¹⁸ I. Jacobson, G. Booch, and J. Rumbaugh "The Unified Software Development Process" (1998). *Addison Wesley Longman*. ISBN: 0-201-57169-2.

Programming¹⁹, alias XP). Architectures are in fact another relevant topic, since different architectural patterns encourage the presence of different subsets of the full set of design patterns. These should be the subjects to explore in order to obtain more answers.

Acknowledgments

Before the end it is necessary to thank professors Medardo Rodríguez and Luis Alberto Zarrabeitia. Their lectures on Python and the foundations of the object model contributed significantly to the understanding of the underlying concepts and technologies commented in this paper, as well as for focusing on readability.

The anonymous referees made also several useful comments, which were very important to enhance the contents and catch some mistakes. Together with the editors, they provided guidelines which undoubtedly contributed to enhance the style and layout of the document.

¹⁹ K. Beck, C. Andres. "Extreme Programming Explained: Embrace Change" (2004) Second Edition. *Addison Wesley Professional*. ISBN: 0-321-27865-8.

Python Switch Statement

Lance Finn Helsten

2007-11-20

Abstract

The Python programming language does not have a built in switch/case control structure as found in many other high level programming languages. It is thought by some that this is a deficiency in the language, and the control structure should be added. This paper demonstrates that not only is the control structure not needed, but that the methods available in Python are more expressive than built in case statements in other high level languages.

1 Introduction

The lack of *switch...case* statements in Python has caused some complaints that take the form: “Python is simple so why doesn’t it have a simple *switch...case* statement”. I will show that Python has a structure that is simpler and more expressive than the *case* constructs that exist in other imperative programming languages.

This article is written to explain the history of the *case* from its genesis in assembly language, to its current implementation in other languages, and give a formal definition of *case* then it will discuss the strengths and weaknesses of *case* as used in other languages.

Then I give two simple methods that will achieve the same objective as that desired with *switch...case* and which will have more power than *case* as defined in other languages.

All of this is necessary as the likely-hood of *switch...case* being added is remote; it has been requested and rejected by Guido van Rossum earlier this year (van Rossum, 2007). He also mentions in PEP 3103 (van Rossum, 2007) that at PyCon 2007 there was no popular support for adding *switch...case* to the language.

I do not discuss other methods of achieving the same results through the use of *if...elseif* (Beck, 2005), or the use of exceptions (Isailovski, 2005): each of those try to emulate other languages which is not the purpose of this paper.

2 Assembly

The *case* control structure had its genesis in assembly language in the form of a jump table or vector table. Jump tables consisted of an array of memory addresses to blocks of executable instructions well ordered on the values the variable could assume.

The execute a branch through a jump table required loading the variable and jump table address, then using index addressing into the table the branch address is loaded, and then a branch to that address would be taken. For the assembly (Listing 1) it requires three instruction cycles regardless of the range for the variable. The memory used, assuming four bytes for each instruction and address, is four bytes for each discrete value plus another twelve bytes for the instructions (28 bytes in this example).

Listing 1: Assembly Jump Table

```
jumpable:

    DS    address0

    DS    address1

    DS    address2

    DS    address3


case:

    lw     a0, variable

    lw     a0, jump_table(a0)

    bra    a0
```

Now compare with the *test* control structure, which most high level language programmers associate with *if...else if...else* statements in C, C++, C#, Pascal, Java, PHP, Perl, or Python. For the assembly (Listing 2) it requires two instructions for each test case and one more for the variable load, in this example that requires nine instructions. The memory used is eight bytes for each discrete value plus four for the load instruction (36 bytes in this example).

Listing 2: Assembly Test

```
case:

    lw     a0, variable

    cmp     a0,0

    beq     address_0

    cmp     a0,1

    beq     address_1

    cmp     a0,2

    beq     address_2

    cmp     a0,3

    beq     address_3
```

In neither of the above cases is a check made to determine if the variable's value is within the valid range. *case* would require two tests, for less than and greater than the range, which would add four instructions for a total of seven. *test* would require no extra testing as

we know at the end of the block the variable value is invalid.

As can be seen the *case* is subject to $O(1)$ or constant time, whereas the *test* is subject to $O(x)$ or linear time. In addition the memory required for *test* quickly approaches twice that needed for *case*. This is a rare case where a trade-off of memory to gain speed does not occur. Therefore jump tables have always been popular when programming in assembly.

There is a restriction which an attentive reader will have noticed in that the set of values must be well ordered for it to be used by a variable in the jump table. In other words each value in the set must be given a number 0,1,2,& to allow indexing into the table. This means is that if the set consisted of the values {5,19,26,135} then these values would need to be mapped to the set {0,1,2,3} or the jump table itself would need to have one-hundred thirty-six entries where most of those entries would be value error handling addresses.

3 High Level Languages

ALGOL, the father of all procedural languages, did not have a *case* statement as the concept of *case* had not been invented yet in high level languages. Then in 1965 a rarely used language ALGOL W was released that contained *switch...case* statement invented by C. A. R. Hoare which allowed for non-ordered non-overlapping values and ranges.

C and Pascal both derived from this parent language, and both had a form of *case*: C the familiar `switch (variable) case ...` and Pascal using `case variable begin value1:` C++, PHP, C#, and Java syntax are derived from the C version.

Generally the *switch...case* was converted to assembly as a sequence of test and branch structures so performance is linear. Whereas in well optimized C compilers it was not unusual for the *switch...case* to be reduced to a jump table whenever possible.

The limitation in any of the above languages is that a variable must be an integer (e.g. in C this would include char, short, int, and long).

A problem with the C *case* is that complex code, that would be better placed into a sub-routine, may be placed directly in the *switch...case* itself. The length of this code could be so large that seeing the structure of the *switch...case* would be impossible: this does not imply that placing code directly in the *switch...case* is a wrong; only that doing so reduces the readability and maintainability of the source code.

4 Formal Definition

What is needed now is a formal definition of a generalized *case*. Thus *case* is defined as let X be a countable set, let F be a set of functions, let c be a function that maps X to F where $\forall f \in F \exists x \in X : f = c(x)$.

In English, this means that for every value the *case* variable may take there is a function that will be called. This seems fairly obvious from assembly jump tables and C *switch...case* however they are limited to the countable set of integers, and not to a general countable set. The definition given here allows all countable sets to be used as the domain for the *case* variable.

This generalized *case* is not possible with language constructs in Algol, C, C++, Objective C, C#, Java, or PHP. Though it is clear that if X is limited to the set of integers then this describes a *switch...case* in C perfectly. To gain the generalized *case* requires writing special code to handle other cases (for example when the variable is a string).

I will now show that Python is able to handle the generalized *case* and is therefore more expressive.

5 Python

The key to a Python case is to use the dictionary structure with the knowledge that functions in Python are first-class. The general form (Listing 3) such that `key_x` represents an $x \in X$, and `function_f` represents a function $f \in F$. This general form represents the c in the definition, therefore it fulfills the definition.

Listing 3: Python Case General Form

```
{
    key_1:      function_1,
    key_2:      function_2,
    key_3: function_3,
    key_4: function_4
}.get(x, defaultFunction())
```

Advantages of this structure are:

1. Any first-class type may be a key: including other functions.
2. Access is in constant time ($O(1)$).
3. There is a `defaultFunction()` to handle invalid keys.
4. Late binding of the functions allows for dynamic code.
5. The dictionary may be stored after it is created to improve speed (Listing 4): the dictionary is interpreted a single time.
6. A dictionary may be selected and then a function generated from that dictionary (Listing 5).

Listing 4: Stored Switch

```
MY_SWITCH = {...}

def fastSwitch(x):
    return MY_SWITCH.get(x, defaultFunction())
```

Listing 5: Selectable Switch

```
SWITCH_1 = {...}
SWITCH_2 = {...}
```

```
def chooseSwitch(switch, x):  
    return switch.get(x, defaultFunction())  
  
chooseSwitch(SWITCH_2, x)
```

5.1 Pure Jump Table

For a pure jump table a simple sequence of functions in a list or tuple may be used (Listing 6). Functions then may be chosen based on a simple integer index in much the same way that assembly jump tables (Listing 1) operate. This has the same $O(1)$ advantage as a dictionary, does not require typing of keys, but does have the limitation on the keys being integers.

Listing 6: Python Jump Table

```
{  
    function_0,  
    function_1,  
    function_2,  
    function_3  
}[x]()
```

5.2 C Style Case

For a C style case use a dictionary that maps integer keys to functions, and if a default is needed use the `get(x, defaultFunction)` instead of placing the key in brackets (Listing 7). This is almost identical to the jump table case (Listing 6), but it is not a requirement that all the keys be well ordered and sequential.

Fall through handling needs to be dealt with in each of the functions, but this has the advantages of clarification of code, and misuse of fall through behavior. This also results in the requirement that every case be explicitly placed in the dictionary.

Listing 7: Python C Style Switch

```
{  
    0: function_0,  
    1: function_1,  
    3: function_3,  
    7: function_7  
}[x]()
```

6 Conclusion

The expressive nature of a dictionary for *case* in Python eliminates the need to have a *switch...case* statement placed in the language. A *switch...case* would obscure the usefulness of this construct that exists currently, and to make it like a C *switch...case* would require the limitation of the general case to keys that are integers.

The progress of computing has always been linked to the progress of the languages we have used: machine code to assembly to C to Python. With this progress some constructs must be left behind to allow more efficient and expressive constructs to replace them. Therefore, to continue the progress of computing and languages a feature should be added to a language when there is a demonstrable benefit in efficiency and expressiveness.

References

- Guido van Rossum. A switch/case statement, June 2007. URL <http://www.python.org/dev/peps/pep-3103/>.
- Brian Beck. Readable switch construction without lambdas or dictionaries, April 2005. URL <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/410692>.
- Zoran Isailovski. Exception-based switch-case, May 2005. URL <http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/410695>.

An Introduction to Test-Driven Code Generation

Raphael.Marvie@lifl.fr
LIFL / IRCICA
University of Lille 1 (France)

Abstract

Agile Software Development promotes the use of techniques such as *Test-Driven Development* (TDD) and *Automation* in order to improve software quality and to reduce development time. Code generation represents a way to achieve automation, reducing repetitive and error-prone tasks.

Code generation is well accepted, writing a code generator is not necessary that hard, however it is not trivial to decide when and how to embrace code generation. Moreover, it is even harder to embrace at the same time code generation and TDD, wondering for example *How to build a generator following a test driven approach?* or *How to test drive generated code?*

This paper aims at providing hints to answer these questions. It presents an agile approach named *Test-Driven Code Generation*. The main principle is to gain knowledge about the application during the first iterations of the development process and then to identify how to implement code generation. As code generation should not drive you out of TDD, our proposal, based on a *double TDD Loop*, provides hints to marry both approaches in order to empower your developments.

1 Introduction

The complexity of applications increases while the productivity of developers is still expected to improve. In the meantime, software quality is not expected to vanish. *Test-Driven Development* (TDD) [1] is one of the solutions that helps you write better software. The leitmotiv of TDD is to *write a test, write the code, and refactor* (TDD loop).

The complexity of a piece of software is partly related to the number of its functions. Looking at most pieces of software, many of these functions share a common structure or behavior inside a single application or between applications in the same context. Identifying these similarities and understanding their differences permits the development of code generators that (a) reduce the amount of repetitive work (thus the underlying potential errors), (b) improve the software quality (enforcing coding rules, not leaving unfinished copy / paste, and so on), and (c) reduce the development time. In all that, code generation is a way to achieve *Automation* [11], allowing you to focus on the value of your application more than on its implementation details. In addition, code generation can ease the integration of code into an existing architecture with defined APIs. Given rules that a piece of code must follow in a given context, code

generation can provide its structure, letting the developer free to focus on the functional implementation.

This paper focuses on *Test-Driven Code Generation*. Our proposal relies on the *Double TDD Loop*, which is made of three steps:

1. Having some working code using TDD.
2. Looking for the part that can be generated (ranging from complete functions to skeletons).
3. Developing the part of the generator that generates the code defined in the second step.

This paper can be view as an academic introduction (even if it has been used to develop an application that manages master students technical projects for two years). It does not try to compete with existing web frameworks such as *Django* [7] or *TurboGears* [16]. General points about code generation and test, as well as the *Double TDD Loop* are discussed in section 2. The three steps of the *Double TDD Loop* are illustrated in sections 3 and 4 while section 5 present the generation of tests. Section 6 concludes this paper and outlines some perspectives.

2 Principles

Extreme Programming encourages as a practice the writing of tests prior to the writing of code (see Test-First Programming in [2]). This practice has evolved to Test-Driven Development (TDD), and K. Beck has defined in [1] the *TDD mantra* or *TDD loop* (which is expected to be used iteratively and automated) as:

1. Red–Write a little test that doesn’t work, and perhaps even does not compile at first.
2. Green–Have the test pass quickly, committing whatever seems necessary in the process.
3. Refactor–Eliminate any duplication created in the second step.

Developing a piece of software is seen as a continuous flow of small steps. Going from one step to the next one is bounded by the passing of all the tests: the new one but also all the existing ones (to ensure non regression).

This section introduces an approach for writing code generators using TDD. It presents the *Double TDD loop* as an extension of K. Beck’s *TDD loop*.

2.1 Approach

A key point about code generation is that it is only about automating repetitive or critical tasks. The code to be generated must be first hand-written several times to let emerge which parts can be generated. Only then, you should write a generator. This first hand-written stage is mandatory to ensure a really useful generator and will be highly compensated by all the time saved using it afterwards.

2.1.1 Generators, Tests, and Oracles

The first step in writing a code generator is to write examples of the code to be generated. As any piece of software, this code has to work and be as clean as possible. There is no point in generating (even if it is free) non working or buggy¹ code. TDD is a good solution to write *clean code that works*².

In our approach we also use TDD to ease the development of code generators. We have to define what are the tests for these generators, thus what are the oracles. An *oracle* is a boolean function that compares an expected result with the actual one. For example, when testing a function that applies taxes to a price, the oracle tells you if the price with taxes returned by the function is the one expected for some known input values. Oracles could be implemented using assertions.

In our case, the function under test is a code generator. Then, we have to define oracles that compare, for known input values, the code produced by the generator to the expected code. It comes naturally that this expected code consists in the examples of code we have hand-written first. When generating source code, this comparison is syntactic. This approach is used whatever the generator produces whole or only parts of an application: For example, you can compare complete functions or only their synopsis.

2.1.2 Generating Tests

Most of the time, we automate the production of only parts of the application code (for example class and method skeletons, boiler plate code). Thus, once the code is generated we have to fill the holes (like functional code) with hand-written code.

Here again, the TDD loop improves our confidence in the code. If we have focused on the functions of the application rather enough to generate their skeletons, we are also able to generate test case skeletons and test suites for these functions. In that, we help automating the test process³.

Finally, we expect the development of code generators not to follow a waterfall approach, but an iterative one. The code generator cannot be completed in a simple cycle: One seeks first a generator that works for a small part of the application, then adds capabilities one by one, being sure that each one is working properly before going to the next one.

2.2 The double TDD loop

Keep in mind that code cannot be generated without having been written first by hand (at least once). The *double TDD loop* principle summarizes our proposal for the support of *Test-Driven Code Generation* (see figure 1). It is a simple extension of the *TDD loop* (which is used “as is” during step 1) and you are encouraged to use this double TDD loop iteratively, producing the generator incrementally. More often, step 1 has to be repeated several times before moving to step 2 in order to gain enough knowledge about what can be generated.

¹Bug-free software is not reachable. So we put all we can in producing code as bug-free as possible.

²Coined by Ron Jeffries.

³Complete test suites, test cases and assertions can be generated from detailed specification of a piece of software. However, this goes beyond the scope of this paper.

1. Have some working code (using one or several iterations).
 - (a) Write a test for a small part of the application, which may range from a simple function to a class. (Going bigger than a class may need several iterations.)
 - (b) Implement the selected part of the application until the test passes (as well as all the existing ones).
 - (c) On the basis of this implementation and the previous ones (if it applies) refactor the code [9].
2. Look for a part of the code that can be generated (from skeletons to complete functions).
3. Develop the part of the generator that generates the code defined in the second step (using one or several iterations).
 - (a) Write a test for the generation of the selected part of the application. This test mainly compares the code produced by the generator to the hand-written code extracted in step 2 while providing the proper input data.
 - (b) Implement the part of the generator that produces this small part of the application until the test passes (as well as all the existing ones for non regression).
 - (c) Refactor the generator to reduce code duplication and improve its clarity, ability to evolution, and maintainability. This should have no impact on the generated code.

Figure 1: The Double TDD Loop

Remark When dealing with code generation, tests, application, and generators have to remain synchronized. During step 1.c, the structure of the application may change requiring the generator to be updated accordingly. Such an update is often limited to modifying the code templates. On the contrary code generator refactoring performed during step 3.c only affects the generator structure, thus has no effect on the generated code.

3 First step: classical TDD

3.1 Illustrative Example

To illustrate our proposal, we present some excerpts of the development of a simple three tiers web application for managing student end-term projects.

The use cases of this application are quite straightforward (see figure 2):

- Any user can list the submitted projects.
- A teacher can add a project.
- A pair of student can register for a project.
- The supervisor can affect projects to pairs.

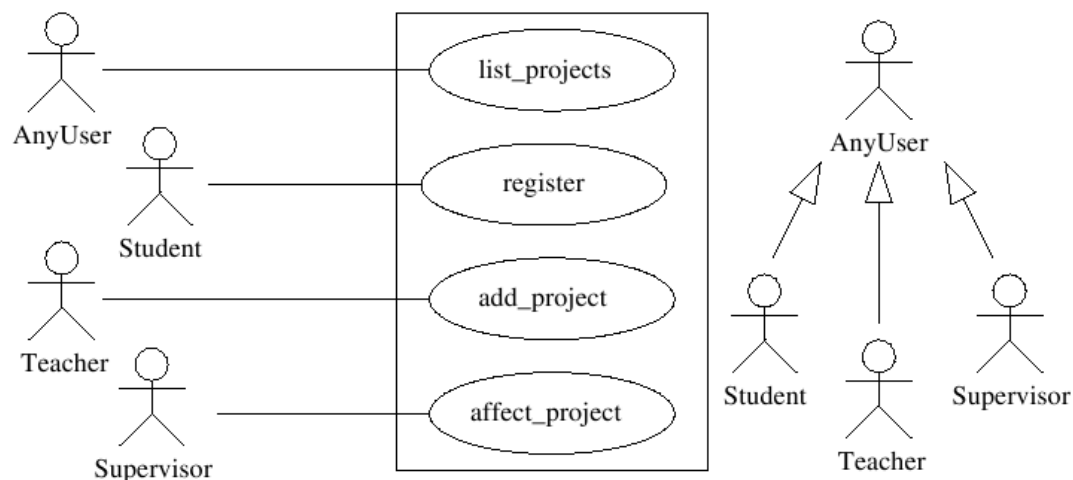


Figure 2: Use Cases for the Management of Technical Projects

The overall structure of the application follows the MVC (*Model / View / Controller*) design pattern: The presentation layer (View) is coded using *CherryPy* [4], the processing layer (Controller) is composed of pure *Python* classes, and the data layer (Model) is made of simple objects in this paper (but it could be defined as a relational / object mapping like *SQLObject* [15]).

Starting from the use cases presented in figure 2, an actor is translated as a controller which is implemented as a class. Each action that can be performed by this actor is translated as a function of this class. Actions are processing data items that can be displayed or retrieved from the presentation layer. Figure 3 presents a simplified version of the application architecture. It underlines two aspects of the application: Each use case spans the three tiers of the application, and the application can be seen as a stack of use case implementations.

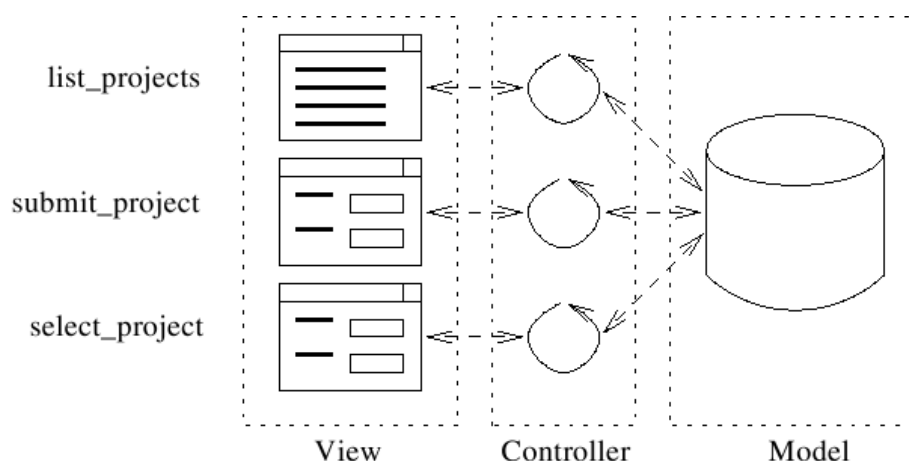


Figure 3: Architecture of the Technical Project Management System (excerpt)

3.2 Teacher view

As a representative sample of TDD, this section illustrates the implementation of the `add_project` function of the View layer for teachers. This function manages a form to add projects to the system. In this paper, a project is simply defined by its name. By default, without provided information, the function returns the form to be filled. Otherwise, it processes the provided information (filled form hence project name) in order to create the project.

First, figure 4 presents the test of the `add_project` function⁴. The test checks that the outputs are correct for expected uses of the function. When submitting information, we check that `add_project` calls the controller properly (only once, and providing the expected information). In order to test only the view and not the controller at the same time (unit testing) the test class is also used as a *mock object*. A mock object behaves as an object of the application but for specific data only⁵: In our case the test redefines the `add_project` function of the controller.

Remark For a class to be unitary tested, it has to be implemented with explicit dependencies: The dependencies of the class should be provided at construction time (like the teacher view takes a controller as parameter) or using modifiers. This requirement is strongly related to the *favor loose coupling of software components* principle (see *Orthogonality* in [10] p. 34). Explicit interactions between software artifacts eases their substitution and reuse.

Finally, figure 5 presents the implementation of the `add_project` function that passes the tests defined in figure 4.

3.3 What have we learned?

Application of TDD to develop the first functions provides the basis of the development process.

⁴This test could also be written using `doctest` [8] or `py.test` [14].

⁵This technique is further discussed in [1] (Self Hunt p. 144). You can also have a look at www.mockobjects.com and [12] for more information about using mocks with Python.

```

import unittest

expected_default = '''<form action="add_project" method="post">
Project: <input name="name" /><br />
<input type="submit" />
</form>'''
expected_filled = 'Your project <i>MyProject</i> has been added.'

class TeacherViewTests(unittest.TestCase):
    def setUp(self):
        self.view = view.TeacherView(self)
        self.projects = list()
    def add_project(self, project):
        """Mock the controller behavior"""
        self.projects.append(project)
    def test_add_project(self):
        self.assertTrue(self.view.add_project.exposed)
        self.assertEqual(expected_default, self.view.add_project())
        self.assertEqual(expected_filled,
            self.view.add_project('MyProject'))
        self.assertEqual(1, len(self.projects))
        self.assertEqual('MyProject', self.projects[0].name)

```

Figure 4: Test Case for the add_project Function (Teacher View)

```

class TeacherView(AnyUserView):
    def __init__(self, controller):
        super(TeacherView, self).__init__(controller)
    @cherrypy.expose
    def add_project(self, name=None):
        if not name:
            return '<form action="add_project" method="post">\n' + \
                'Project: <input name="name" /><br />\n' + \
                '<input type="submit" />\n</form>'
        else:
            project = model.Project(name)
            self.controller.add_project(project)
            return 'Your project <i>%s</i> has been added.' % name

```

Figure 5: Implementation of the add_project Function (Teacher View)

- The application architecture is outlined (see figure 3). It is made of three layers, split into four modules: model, controller, view, and main.
- A test suite is associated to each module, and a global test suite groups all the module suites (easing the test of the whole application).
- We have *Running Tested Features*⁶ ready to be demonstrated to end users for feedback⁷ or even deployed (even if we have shown only parts of these RTF).
- Finally, we can extrapolate from these functions what the other ones will look like.

It is a good starting point to move on into developing a code generator.

4 Second step: the generator using TDD

4.1 Overview

We consider in this section a generator that produces a basic skeleton of our application modules. We base its development on the implementation of controllers and views we outlined in section 3 as follows:

- We can extract, from these implementations, skeletons of controller and view classes. These skeletons represent the repetitive parts found in the hand-written code, while completing them represents specific code which cannot be generated.
- The generation of skeletons requires some information as input: The names of controllers and views as well as the name of their functions sounds interesting. This information is contained in our use cases (see figure 2).

We define test cases for the generator using the expected skeletons as oracles.

4.2 Generation of skeletons for views

A generator is as important as the applications it produces: It has to be properly developed, tested, documented and maintained.

4.2.1 Tests

Tests are defined on the basis of the expected behavior of the application. The expected behavior of our generator is to produce source code, and the current expected output is skeletons of view classes. On the basis of the view presented in figure 5 we can extract the `view_test` test case skeleton depicted in figure 6.

The string `view_test` contains the expected output of the generator when its input is the definition of the `list_project` use case. The test provides as input the name of the actor together with a list of the associated actions and the oracle compares the output of the generator to the string `view_test`.

⁶Coined by Ron Jeffries.

⁷Feedback is defined as a value by *eXtreme Programming* (see [2] p.19) and encouraged in agile and lean software development (see [5] p.66 and Tool 3 of [13] p.22).


```

view_test = '''
import cherrypy
class AnyUserView(object):
    def __init__(self, controller):
        super(AnyUserView, self).__init__()
        self.controller = controller
    @cherrypy.expose
    def index(self):
        return 'To be completed'
    @cherrypy.expose
    def list_projects(self):
        return 'To be completed'
'''
class TestViewGenerator(unittest.TestCase):
    def test_generate(self):
        self.model = {'AnyUser': ['list_projects']}
        self.assertEqual(view_test,
            generator.ViewGenerator().generate(self.model))

```

Figure 6: Test Case of the View Generator Class

4.2.2 Implementation

There are a bunch of solutions for generating code. On one hand, the use of standard *Python* template strings (like "hello %s" % name) allows the production of the code slice by slice. On the other hand, one can use template systems like *Cheetah* [3] for generating complete files⁸. We use this last solution.

Both approaches rely on two levels: an *abstract level* defines generic code with holes, and a *concrete level* answers to a specific context filling the holes with proper input.

As an example, figure 7 presents a first implementation of a view generator. The template (**view_template**) states that for each actor of the provided use case (defined in the **usecases** dictionary), a class is defined and named like the actor postfixed with "View". Then, for each action associated to the actor, a function is defined and named like the use case action (without parameters for the moment). The generator simply evaluates the template using the provided information (stored in the dictionary **searchList**).

4.3 Generation of the application

Generating partly or completely an application is not limited to the generation of source code. The code has to be organized into files, most of the time respecting a structure on the file system for which basic knowledge is required.

For the moment, our application structure is defined as follows. Files are stored in a folder named like the application. Each file is a *Python* module containing a set of related classes: a module for the data structures (**model.py**), a module for the controllers (**controller.py**), a module for the views (**views.py**), and a module for the bootstrap of the application (**main.py**).

⁸We have successfully used *Cheetah* for the generation of *Python*, *C*, *Java*, *SQL* source code, as well as plain text reports.

```

from Cheetah.Template import Template
view_template = '''
import cherrypy
#for $actor in $usecases
class ${actor}View(object):
    def __init__(self, controller):
        super(${actor}View, self).__init__()
        self.controller = controller
    @cherrypy.expose
    def index(self):
        return 'To be completed'
#for $function in $usecases[${actor}]
    @cherrypy.expose
    def ${function}(self):
        return 'To be completed'
#end for
#end for
'''
class ViewGenerator(object):
    def generate(self, usecases):
        return Template(view_template,
                        searchList=[{'usecases': usecases}]).respond()

```

Figure 7: Implementation of the View Generator Class

4.3.1 Tests

Generating the structure of the application and producing the proper code in the proper file has to be tested also, as illustrated by the test for the complete generator (**MyGenerator**) in figure 8. This test checks the integration of all the pieces together. It verifies the proper creation of the various files as well as their content (it integrates the various unit tests we previously wrote).

4.3.2 Implementation

Figure 9 presents the implementation of the main class of the generator developed on the basis of the test case defined in figure 8. This part of the generator simply creates the folder and files associated to the web application defined by the use case and uses the various generators we developed in the previous sections to produce the content of these files. This class is a simple coordinator of the various pieces of the generator.

4.4 About the input description

For all the tests we wrote, we used a simple *Python* data structure composed of dictionaries and lists in order to provide the input data to generators. This solution may not be the best one in production since it would require to check first that the input information is properly structured.

Another solution could be to use XML, as it is easily parsed in *Python*. First, the XML document can be validated using a DTD. Second, we can modify the generator to eat XML, or we can translate the XML to our simple *Python* data


```

expected_file_content = {
    'model.py': model_test,
    'controller.py': controller_test,
    'view.py': view_test,
    'main.py': main_test,
}
class TestMyGenerator(unittest.TestCase):
    def setUp(self):
        self.description = {'MasterProjects': {'AnyUser': ['list_projects']}}
        self.generator = generator.MyGenerator()
    def test_files(self):
        self.generator.generate(self.description)
        self.assertTrue('MasterProjects' in os.listdir('.'))
        files = os.listdir('MasterProjects')
        for f in expected_file_content:
            self.assertTrue(f in files)
            self.assertEqual(file('MasterProjects/%s' % f).read(),
                             expected_file_content[f])

```

Figure 8: Test Case of the Generator Main Class

```

class MyGenerator(object):
    def __init__(self):
        super(MyGenerator, self).__init__()
        self.generators = {
            'asbtraction.py': ModelGenerator(),
            'controller.py': ControllerGenerator(),
            'view.py': ViewGenerator(),
            'main.py': MainGenerator()
        }
    def _create_file(self, path, content):
        file('%s/%s' % path, 'w').write(content)
    def generate(self, model):
        for usecase in model:
            os.mkdir(usecase)
            for name, g in self.generators.items():
                self._create_file((usecase, name),
                                  g.generate(model[usecase]))

```

Figure 9: Implementation of the generator main class

structure. Two small steps (verification, translation) are better than a big one for modularity reasons: Each tool may be useful in more than one context. Figure 10 presents a simple document example for providing input data.

```
<?xml version="1.0" ?>
<usecase name="MasterProjects">
  <actor name="AnyUser">
    <action name="list_projects" />
  </actor>
  <actor name="Teacher">
    <extends>AnyUser</extends>
    <action name="add_project" />
  </actor>
</usecase>
```

Figure 10: XML version of the input information

Finally, use cases can easily be graphically defined using an UML tool. UML tools are supposed to export models as XMI (*XML Meta-data Interchange Format*) which is an XML vocabulary for representing models. Thus, a simple function may translate the XMI into the *Python* data structure. This may represent your first step towards *Model Driven Development*.

4.5 About testing generated code

In practice, generated code is not as compact as code excerpts presented in this paper (due to compactness motivations). In addition, blank lines and non significant white spaces are recommended in practice to obtain readable code but are a nightmare for testing generated code. The `assertEqual` function tests for exactly equal strings, not for strings having the same meaning. Let us see two solutions (among others) for dealing with source code comparison.

First, you can write a function that compares two code snippets or subtype `str` for redefining the `__eq__` operator. Second, you can compile the code strings using the `codeop` module [6] that permits the dynamic compilation of *Python* source code in memory and the comparison of the resulting code objects (see figure 11). This last solution has the advantage to also check that the code is correct (which should be the case, as we already wrote it using TDD).

```
import codeop
def has_same_meaning(code1, code2):
    """Do the two python code strings have the same meaning?"""
    return codeop.compile_command(code1) == codeop.compile_command(code2)
```

Figure 11: Function for Comparing the Meaning of Two Code Strings

These two solutions are not always substitutable: when you want to test incomplete code (a slice of a class or so) or non *Python* code the second solution is not usable, while it is easier to use when you want to test complete *Python* code (a function, a class, or a module).

5 Third step: TDD generated

For the moment, the generator we have produced only generates skeletons for some pieces of the application. We still need to hand-write its functional code. To apply TDD, the developer has to write tests first. Here again, code generation can ease the work by producing the skeleton of test cases for generated functions. It defines a test road map as expected test cases are identified.

5.1 Generation of the unit test skeletons

Like we generated code skeletons, we can generate test case skeletons. The oracle for testing the test generator is derived from the tests we wrote earlier (see section 3). Figure 12 presents the expected test skeleton for the AnyUser view. For each function of each view class, a test function is defined.

You may wonder why the addition of the test case to the test suite is commented out. Following Kent Beck principle *one test at a time*, all the tests should not be run at first. So, we start by commenting all test case addition in the test suite, then the developer will uncomment them one at a time, applying the *TDD loop* and never leaving a test broken. Moreover, the list of tests represents a test road map so it is better to specify them all. The uncommenting of tests represents a progress bar in the development process.

```
test_view_test = '''
import unittest
import view
class TestAnyUserView(unittest.TestCase):
    def setUp(self):
        pass
    def tearDown(self):
        pass
    def test_list_projects(self):
        self.fail('Not Implemented')
def suite():
    suite = unittest.TestSuite()
    # suite.addTest(unittest.makeSuite(TestAnyUserView))
    return suite
if __name__ == '__main__':
    unittest.TextTestRunner(verbosity=2).run(suite())
'''
```

Figure 12: Test Case Oracle for the Generation of View Test Skeletons

5.2 Using the generator

Using code generation in a project implies the use of a development process⁹ that is a little different from the Write Test / Implement / Refactor. Generators can be used in different ways, mostly depending on the generator itself (purpose, abilities, and so on).

⁹Remember we try to be agile, so we are not speaking of rigid development processes.

Code generation as we present it in this paper represents a bootstrap of the application development process. The development process associated to our generator is the following one: Steps 1 and 2 are performed only once at the beginning, then steps 3 to 5 represent an iteration of development. Completing a 3-5 cycle means adding a new *Running Tested Feature*.

1. Define the model of the application using the input format.
2. Generate the test and implementation skeletons of the application.
3. Write a test for a function of the application.
4. Implement the function associated to the test.
5. Refactor if necessary.

A more practical approach is to extend the 3-5 cycle to a 1-5 cycle (the complete development process is then iterative). It does not seem possible to have a complete model of an application in a single shot and before starting the implementation. This approach is a little bit more complicated to setup. Code organization has to be more fine grained in order to support iterations. For example, one file per class and one package instead of a module for each part of the application allow one to add new use cases without overwriting existing ones. It is also necessary to be able to add a function to an existing class without breaking the class (some of its functions may have already be implemented). This outlines the complexity of the maintenance and evolution of generated code.

Warning You have to remember when using code generation that running the generator may take less than a second while destroying hours of work. If you re-generate code for a project when hand-written code has been produced, you may erase the code you have written (for example the controller module implementation). Sometimes, you can clearly separate generated code from hand written one, sometimes you cannot. If it is not possible to separate the two kinds of source code, do not allow the (re)generation of the application on top of an existing one (that may have already been partly completed)¹⁰.

6 Conclusion

This paper introduces *Test-Driven Code Generation* (TDCG). TDCG helps the development and the integration of code generators in a TDD context. Using a simple web application as an example, we have underlined two key aspects of TDCG: (a) The TDD based writing of code examples to be generated in order to define oracles for testing generators, and (b) the generation of test skeletons in order to apply TDD when completing the application (considering that the generated code represents only part of the whole application code). Finally, we have focussed on the basis of *Test-Driven Code Generation* discussing the *Double TDD Loop*.

¹⁰This is the reason why the existence of the folder to store the files is not checked: If the folder exists then the generation fails. So you can be sure not to destroy work, as you have to wonder first: "Should I remove the folder and its content or not?"

This paper is only an introduction to TDCG. Applying TDCG in large projects would require to support iterative development as well as generating functional code.

- Agility means the ability for developers to gain feedback from customers as soon and as often as possible. To support such approach, the implementation of code generation has to be compatible with the development of use cases one by one.
- Recurrent functional behaviors can be identified. Extending the input information, some functions can be completely generated. At that point you also have to balance the use of code generation with the use of generic functions. Due to *Python* flexibility both are interesting, and should be combined to ease your development activity.

Extending the input information can improve the amount of code one can generate. But, it is necessary to find the balance between generate-able code and hand-written code, *i.e.* the cost of writing a generator compared to the cost of writing the code by hand. Generating 100% of an application does not seem to be a proper goal. Reaching 90+% of generated code is quite common. Most of the time, the last percents are the exceptional cases that happen rarely or for which providing enough description is harder than hand-writing the code (not mentioning the cost of implementing the generator itself).

Finally, having multiple generators—like a distinct generator for each layer—is another interesting improvement. Having one big tool that does everything is not very modular nor flexible. Having several small specific tools that can be composed to achieve a task is better. For example, other generators could be written for the production of *Tkinter* view classes (or any other graphical library you like), and multiple generators could be written for several persistent frameworks. Then, once the application description is defined, you choose the graphical and persistent frameworks you need and use the appropriate generators in order to produce the flavor(s) of the application you need for a given context.

References

- [1] K. Beck. *Test-Driven Development by example*. Addison-Wesley, 2002. ISBN: 0-321-14653-0.
- [2] K. Beck. *Extreme Programming Explained, Embrace Change*. Addison-Wesley, second edition, 2005. ISBN: 0-321-27865-8.
- [3] Cheetah, The Python-Powered Template Engine. Web site. <http://www.cheetahtemplate.org/>.
- [4] CherryPy is a pythonic, object-oriented web development framework. Web site. <http://www.cherrypy.org/>.
- [5] A. Cockburn. *Agile Software Development*. Addison Wesley, 2002. ISBN: 0-201-69969-9.
- [6] codeop – Compile Python code. Web site. <http://docs.python.org/lib/module-codeop.html>.

- [7] Django, The web framework for perfectionists with deadlines. Web site. <http://www.djangoproject.com/>.
- [8] `doctest` – Test interactive Python examples. Web site. <http://docs.python.org/lib/module-doctest.html>.
- [9] M. Fowler. *Refactoring, improving the design of existing code*. Addison-Wesley, 1999. ISBN: 0-201-48567-2.
- [10] A. Hunt and D. Thomas. *The Pragmatic Programmer, From Journeyman to Master*. Addison Wesley, 2000. ISBN: 0-201-61622-X.
- [11] T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988. ISBN: 0-915-29914-3.
- [12] `pMock` is a Python module for testing Python code using mock objects. Web site. <http://pmock.sourceforge.net/>.
- [13] M. Poppendieck and T. Poppendieck. *Lean Software Development, An Agile Toolkit*. Addison Wesley, 2003. ISBN: 0-321-15078-3.
- [14] The `py.test` tool and library. Web site. <http://codespeak.net/py/current/doc/test.html>.
- [15] `SQLObject` is a popular Object Relational Manager for providing an object interface to your database. Web site. <http://sqlobject.org/>.
- [16] TurboGears is the rapid web development megaframework you’ve been looking for. Web site. <http://www.turbogears.org>.

Upcoming Events

The following events, taken from the python.org events wiki²⁰, are being held between December this year and March 2008.

December 12, 2007: Cologne, Germany, monthly [pyCologne meeting](#)

December 13, 2007: IRC, Internet, [Kubuntu Tutorials Day](#) features "Get programming with PyKDE 4"

March 15, 2008: Portland, Oregon [Python workshop SIGCSE'08 \(workshop 32\)](#) led by [Atanas Radenski](#)

March 13, 2008: Chicago, Illinois, [PyCon 2008 Tutorial Day](#)

March 14-16, 2008: Chicago, Illinois, [PyCon 2008](#)

March 17-20, 2008: Chicago, Illinois, [PyCon 2008 Sprints](#)

To include your event in our next issue, or to include expanded event information, please contact us directly to ensure that your event is represented as you would like. All events available from the python.org events wiki will be included with a basic reference.

²⁰ <http://wiki.python.org/moin/PythonEvents>

The Python Papers' Review Policy

0. Preamble

The Python Papers (ISSN 1834-3147) is intended to be both a industrial journal as well as an academic journal, in the sense that the editorial board welcomes submissions from all aspects related to the Python programming language, its tools and libraries, and community, both of academic and industrial inclinations. The Python Papers aims to be a publication for the Python community at large. In order to cater for this, The Python Papers seeks to publish submissions under 2 main streams: the industrial stream (technically reviewed) and the academic stream (peer-reviewed). This policy statement seeks to clarify the process of technical review and peer-review in The Python Papers.

1. Right of submission author(s) to choose streams

The submission author(s); that is, the author(s) of the article or code or any submissions in any other forms deemed by The Python Papers editorial board (hereafter known as 'editorial board') as being suitable; reserves the right to choose if he/she wants his/her submission to be in the industrial stream, where it will be technically reviewed, or in the academic stream, where it will be peer-reviewed. It is also the onus of the submission author(s) to nominate the stream. The editorial board defaults all submissions to be industrial (technical review) in event of non-nomination by the submission author(s) but the editorial board reserves the right to place such submissions into the academic stream if it deems fit.

2. Right of submission author(s) to nominate potential reviewers

The submission author(s) can exercise the right to nominate up to 4 potential reviewers (hereafter known as "external reviewer") for his/her submission if the submission author(s) choose to be peer-reviewed. When this right is exercised, the submission author(s) must declare any prior relationships or conflict of interests with the nominated potential reviewers. The final decision rests with the Chief Reviewer.

3. Right of submission author(s) to exclude potential reviewers

The submission author(s) can exercise the right to recommend excluding any reasonable numbers of potential reviewers for his/her submission. When this right is exercised, the submission author(s) must indicate the grounds on which such exclusion should be recommended. Decisions for the editorial board to accept or reject such exclusions will be solely based on the grounds as indicated by the submission author(s).

4. Peer-review process

Upon receiving a submission for peer-review, the Editor-in-Chief (hereafter known as "EIC") may choose to reject the submission or the EIC will nominate a Chief Reviewer (hereafter known as

"CR") from the editorial board to chair the peer-review process of that submission. The EIC can nominate himself/herself as CR for the submission. The CR will send out the submission to TWO or more external reviewers to be reviewed. The CR reserves the right not to call upon the nominated potential reviewers and/or not to call upon any of the excluded potential reviewers as suggested by the submission author(s). The CR may also concurrently send the submission to one or more Associate Editor(s) (hereafter known as "AE") for review. Hence, a submission in the academic stream will be reviewed by at least three persons, the EIC as CR and two external reviewers. Typically, a submission is reviewed by three to four persons: the EIC as CR, an AE, and two external reviewers. There is no upper limit to the number of reviews in a submission. Upon receiving the review from external reviewer(s) and AE(s), the CR decides on one of the following options: accept without revision, accept with revision, reject; and notifies the submission author(s) of the decision on behalf of the EIC. If the decision is "accept with revision", the CR will provide a deadline to the submission author(s) for revisions to be done and will automatically accept the revised submission if the CR deems that all revision(s) were done; however, the CR reserves the right to move to reject the original submission if the revision(s) were not carried out by the stipulated deadline by the CR. If the decision is "reject", the submission author(s) may choose to revise for future re-submission. Decision(s) by CR or EIC is final.

5. Technical review process

Upon receiving a submission for technical review, the Editor-in-Chief (hereafter known as "EIC") may choose to reject the submission or the EIC will nominate a Chief Reviewer (hereafter known as "CR") from the editorial board to chair the review process of that submission. The EIC can nominate himself/herself as CR for the submission. The CR may decide to accept or reject the submission after reviewing or may seek another AE's opinions before reaching a decision. The CR will notify the submission author(s) of the decision on behalf of the EIC. Decision(s) by CR or EIC is final.

6. Main difference between peer-review and technical review

The process of peer-review and technical review are similar, with the main difference being that in the peer review process, the submission is reviewed both internally by the editorial board (EIC/CR and assigned AE(s)) and externally by external reviewers (nominated by submission author(s) or nominated by EIC/CR). In a technical review process, the submission is reviewed by the editorial board and any external review may be at the editorial board's discretion.

7. Umbrella philosophy

The Python Papers' editorial board firmly believes that all good (technically and/or scholarly/academic) submissions should be published and that the editorial board is integral in refining all submissions. The board believes in giving good advice to all submission author(s) regardless of the final decision to accept or reject and hopes that advice to rejected submissions will assist in their revisions.