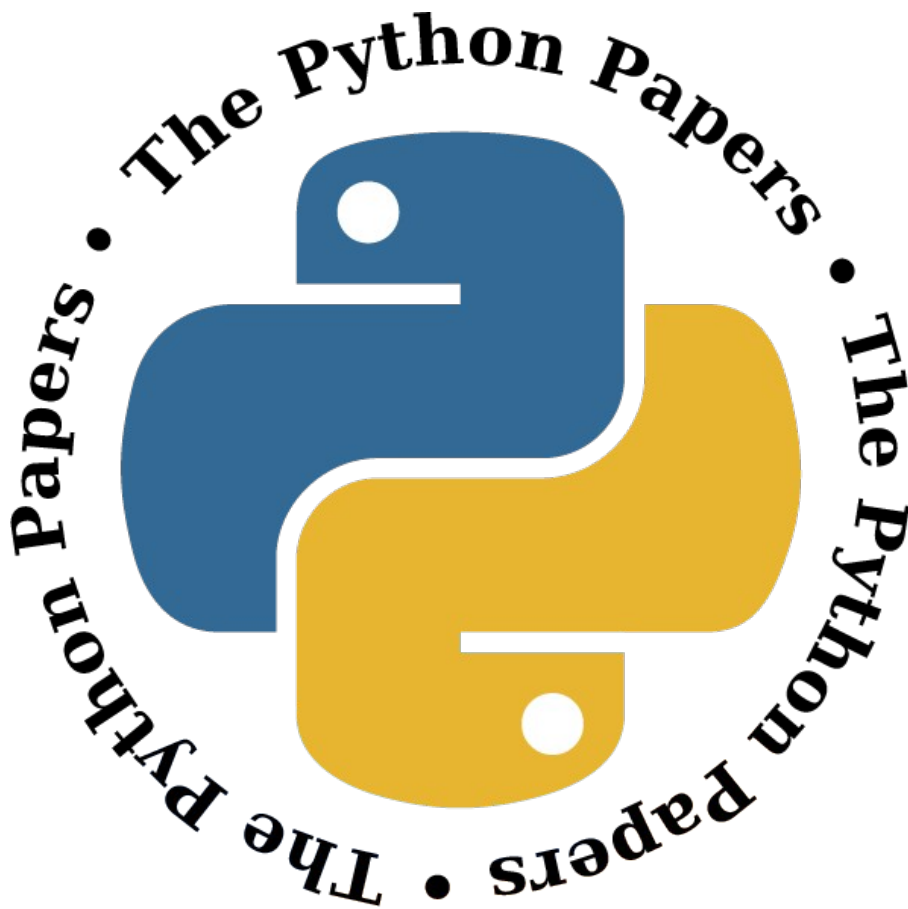


The Python Papers



Volume 3, Issue 1
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 Three, Issue One : March 2008

In this Edition:

Articles and Editorials:

Editorial

Page 3

Editor-In-Chief, Tennessee Leeuwenburg

Pyfundamentalism: The Emotions Provoked by Python

Pages 5-6

Gloria W. Jacobs

When doing a search for the term "Pyvangelism", I found this link: <http://webpages.cs.luc.edu/~mt/Python-First/Py1.html>, which is a small series of slides, succinctly written by yet another Python fanatic. The words fanaticism and evangelism imply the crossing of a boundary from rational to purely emotional state of mind. Fanaticism, or any form of fundamentalist allegiance to any one person, place or thing, usually worries me. It implies that the person may have relinquished all mental power to re-evaluate other tools, techniques and options. At the same time, it's fascinating when a programming language elicits deep emotion, and causes people to cross the line of rational thought into emotional feeling and expression.

What is ShowMeDo?

Pages 7-19

Ian Oswald and Kyran Dale

ShowMeDo is a tutorial video site mostly for open-source software with a strong Python focus. Many videos are made by our open-source Authors and we also make our own tuition-focused series especially for Python via a subscriptions package.

You can think of us as a 'YouTube for education' but note that we take a very strong anti-YouTube approach to new material and comments. Whilst anyone can make a video, all videos are previewed before being accepted, we don't believe that a wisdom-of-the-crowds approach is suitable when dealing with education. We strive to provide our Authors with a trustworthy portal through which their knowledge is shown.

Over 43,000 videos are played each month, we host all of them under liberal licenses (the authors typically choose a Creative Commons license). We refuse to have the all-too-typical adverts so you experience a clean site that is focused on teaching you new skills. You can even embed the videos in your own site just like with YouTube - except that ours are crystal clear!

An Efficient Scalar Package in Python

Pages 20-25

Russell A. Paielli

A Python class was designed to represent physical scalars and to eliminate errors involving implicit physical units (e.g., confusing angular degrees and radians). The standard arithmetic operators are overloaded to provide syntax identical to that for built-in numeric types. The

scalar class does not define any units itself but is part of a package that includes a complete implementation of the standard metric system of units and many common non-metric units. It also allows users to define a specialized or reduced set of appropriate physical units for any particular application or domain. Once an application has been developed and tested, the scalar class can easily be switched off, if desired, to achieve the execution efficiency of operations on built-in numeric types, which can be nearly two orders of magnitude faster. The scalar class can also be used for discrete units to enforce type checking of integer counts, thereby enhancing the built-in dynamic type checking of Python. The scalar package is available from <http://RussP.us/scalar.htm>.

PyGTK, PyQT, Tkinter and wxPython comparison

Pages 26-37

Guilherme Polo

Python offers a multitude of GUI toolkits, much more than described here, for assisting on development of graphical applications, and by having so many options available the chances are you will be unable to make a good decision. A good decision would be one that fulfill your requirements, and in order to achieve this it is necessary an understanding of choices available.

Python Events

Page 81

*A list of upcoming Python events.***Peer Reviewed Submissions:****Doctest and unittest... now they'll live together happily**

Pages 38 - 51

Olemis Lang

This article presents a novel way to verify doctests while unittest runs. A new object oriented API was crafted, and an underlying glue layer was built in order to reconcile both frameworks' interfaces. In the process design patterns have been applied. The tool has been used to test several examples. It provides new loaders to extract test cases from doctests. It also records separately into unittest test results the match made for individual examples. Therefore reports are consistent with unittest's, and automated test analysis require no further parsing. Doctest readability and unittest flexibility are finally together.

The Python interpreter as a framework for integrating scientific computing software-components

Pages 52 - 64

Michel F. Sanner

Building and simulating complex molecular system requires the tight interoperation of a variety of software for tools originating from various scientific disciplines and usually developed independently of each other... the Python interpreter serves as the integration framework and provides a powerful and flexible glue for rapidly prototyping applications from reusable software components.

Parts-of-Speech Tagger Errors Do Not Necessarily Degrade Accuracy in Extracting Information from Biomedical Text

Pages 65 - 81

Maurice HT Ling, Christophe Lefevre and Kevin R Nicholas

An ongoing assessment of the literature is difficult with the rapidly increasing volume of research publications and limited effective information extraction tools which identify entity relationships from text. ...This study aims to evaluate the parts-of-speech (POS) tagging accuracy and attempts to explore whether a comparable performance is obtained when a generic POS tagger, MontyTagger, was used in place of MedPost, a tagger trained in biomedical text.

Letter from the Editor

Tennessee Leeuwenburg



Hello to the readers of *The Python Papers*!

This issue marks the beginning of Volume 3. It almost feels like we are no longer in “startup” mode. This issue is stuffed full on content, so thanks to all those in the Python community who have contributed to this journal over its lifetime. We have a fantastic, in-depth article on ShowMeDo who are setting a real example for online Python content delivery. Ian Oswald and Kyran Dale have kindly put a lot of work into giving a tour of what they have to offer.

Our newest editor, Guilherme Polo also gives us an up-do-date exploration of a number of choices for Python GUI development, showing the pros and cons of the major toolkits. Gloria W. Jacobs continues her fine column, with this issue covering “PyVangelism”. Most Python developers that I talk with are passionate about their choice of language, so I'm sure this will spark a great deal of interest. Unfortunately, I can't simply list every article in the journal here, but read on for more great content.

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.

Stephanie Chong – Associate Editor

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

Guilherme Polo – Associate Editor

I like variety, in my somewhat short time (around 7 years) as developer I have developed programs related to Network, Data compression, Database, Web, Desktop, Data Modeling, Text Parsers, among others, in diverse languages. Almost always you will find me programming, reading, writing about something I enjoy, or even get me studying Computer Science at State University of Maringá (Brazil).

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

Pyfundamentalism: The Emotions Provoked by Python

Gloria W. Jacobs

When doing a search for the term “Pyvangelism”, I found this link: <http://webpages.cs.luc.edu/~mt/Python-First/Py1.html>, which is a small series of slides, succinctly written by yet another Python fanatic. The words fanaticism and evangelism imply the crossing of a boundary from rational to purely emotional state of mind. Fanaticism, or any form of fundamentalist allegiance to any one person, place or thing, usually worries me. It implies that the person may have relinquished all mental power to re-evaluate other tools, techniques and options. At the same time, it's fascinating when a programming language elicits deep emotion, and causes people to cross the line of rational thought into emotional feeling and expression.

I think the reasons why this happens become obvious if you've experienced this yourself. Python's learning curve is quick, even for novice software developers, hobbyists and casual enthusiasts who like to dabble in software. Advanced, fairly complex operations can be learned and done in Python very quickly. And by now, the number of “dabblers” is so high that the vast amount of examples makes it easy to get a running start on any aspect of Python. There are numerous new blogs, comprised of only a few new pages, touting the “Hey! Look what I just did in Python!” epiphanies people have experienced. It's so shockingly easy for so many, that they feel the need to show the world how shockingly easy it was.

The appeal of Python to the seasoned developer is more relief than epiphany, but elicits a similar reaction. The great relief of having to spend only two days writing and testing socket I/O code which would have taken you first a week to write, and then another week to test, in C, goes beyond words. For the seasoned developer, it give us new hope of actually having time to visit the beach, or walk in the woods once again.

The new, true test of the viability of a programming language today has become that feeling of empowerment, the “ah-ha” moment of “Hey, I can do this too, even though I have not achieved wizard status”. In the very recent past, programming languages were what they were. Everyone complained about their quirks and horrors, but we all simply accepted them, warts and all. If a language made our programming tasks torturous, we shrugged it off, hunkered down behind our desks, and did what we had to do to get the driver, app, network protocol, or GUI working.

I have to admit, there was a great pride in achieving “wizard status” in UNIX environments back in the 1980's. I relished the fact that I was the only person in my group back then who knew what an alignment error was in C, and understood what caused it, and how to fix it. Another example is Endianness. Most developers today have probably heard of Endianness, but will probably never have to deal with it, or genuinely understand it. Back then it was a common issue, and your Endian knowledge was one respectable way of proving your UNIX prowess, and showing your “battle scars” from the trenches of UNIX software development.

The UNIX wizard analogy was very real in the 1980's; so real in fact, that a series of UNIX wizard posters were printed and handed out at UNIX Expo, and other UNIX conventions. People stood in line for them, and even laminated them and hung them over their desks in pride. They are now collectors items. I am auctioning off two of these Unix Magic posters (<http://www.devchix.com/2008/01/20/two-mint-condition-original-unitech-unix-magic-posters-for-auction/>). Yes, I too stood in line for them, so often that I have extras. I was very young, and admired whole wizard imagery, as many of us young role paying gamers did. It was “so cool” back then, as cool as being Dungeon Master. Now it's laughable, but the seriousness of the imagery was genuine back then. The stereotypical smart AT&T employee was a DMTS (Distinguished Member of Technical Staff), did not need to wear a badge because everyone read his book and knew who he was, and had the physical appearance of a gray haired

“wizard”. Many of us young neophytes were not beholden to the wizard imagery. It didn't properly represent us, although we could embody the role. But it was an image which paid great respect to our UNIX elder pioneers. For that reason we too respected the wizard image, and kept it alive. But in our hearts, we knew we would become “wizards” much faster than many of our elders.

To some, these new, easy-to-learn languages were a threat to their treasured knowledge. They worked long and hard to achieve their UNIX wizard status, and felt that everyone else should also. RTFM was “the way”, the only way to properly learn, in many UNIX environments back then. But this was rapidly changing, to the dismay of many elitist “wizards”.

I saw, and experienced a definitive change in mindset from wizard status to casual user with the advent of the Sun Microsystems “Pizza Box”. Their machines did not arrive on crates, like all other UNIX machines. These were workstations, which arrived in boxes which you could lift by yourself, with complete and easy to read manuals. Much like a PC of today, you unpacked, it, flipped through the manual, set it up, and you were up and running.

I can honestly say that I learned most of my UNIX system administration skills by reading the Sun Microsystems manuals of the late 80's. I was contracted out to law firms, to set up X terminals, transceivers, networking cable, modems, UUCP, cron and email daemons and services, etc. I learned all of it directly from the manuals. It was empowering, to be 22 years old, and have achieved “wizard” status from a series of manuals.

The languages to follow were just as empowering, allowing rapid development (a phrase which was an oxymoron in the 1980's). Over these years, I had never heard developers evangelizing C++, Java, or PHP. They were better languages than their predecessors – handy, but not worth preaching about. So, why does Python elicit the preacher in the software developer, and cause them to cross the line from rational acceptance to fundamentalist adoration?

I can tell you from my personal experiences that Python seems to be the easiest language to teach online. In my GrrlCamp group, the programming skill varies from novice to very experienced, but in other languages. When I demonstrate things in Python, everyone gets it. Very few discussions are about syntax and language nuance. Most discussions are about our own design, implementation and use of the language. That alone is a beautiful thing, and for this reason, our progress through this project is amazing.

I recently started another, smaller group online called PySheep. We are a small group of female volunteers, writing supplemental image display and manipulations tools in Python and Android, for the Electric Sheep project. For this project, I've chosen WxPython for the cross platform GUI. WxPython is a robust series of Python wrappers written using SWIG, around the WxWidgets libraries. In one day, I got the basics of mpeg to png conversion, and png frame display working. Rapid development is an understatement. This is lightning fast development, and it is most definitely empowering.

Could this empowerment make me feel like writing about my experiences, and encouraging the use of Python. Certainly. It motivated me to write this article. Am I a pyfundamentalist? Well, my answer is, right now, yes. I keep an open mind, I am aware of, and was bitten by the GIL¹ issue on one recent project. I do re-evaluate my choices on a regular basis, and Python still rises to the top of my list as the easiest, most powerful and functional general purpose language out there right now.

1 Global Interpreter Lock

What is ShowMeDo?

Ian Oswald and Kyran Dale

ShowMeDo is a tutorial video site mostly for open-source software with a strong Python focus. Many videos are made by our open-source Authors and we also make our own tuition-focused series especially for Python via a subscriptions package.

You can think of us as a 'YouTube for education', but note that we take a very strong anti-YouTube approach to new material and comments. Whilst anyone can make a video, all videos are previewed before being accepted, we don't believe that a wisdom-of-the-crowds approach is suitable when dealing with education. We strive to provide our Authors with a trustworthy portal through which their knowledge is shown.

Over 43,000 videos are played each month, we host all of them under liberal licenses (the authors typically choose a Creative Commons license). We refuse to have the all-too-typical adverts so you experience a clean site that is focused on teaching you new skills. You can even embed the videos in your own site just like with YouTube - except that ours are crystal clear!

70 authors share their knowledge with thousands of daily visitors. Python was our original focus (for the sake of history see: <http://showmedo.com/videos/series?name=PythonDownloadInstallTest>) and forms 1/3 of our 500 videos. Videos are typically 5-30 minutes long and are shown in sharp VGA resolution backed with a source-code listing and blog-like Comments for conversation. New videos are added every week.

Whilst our main focus is Python we have a range of other topics extending to Perl, Ruby, Blender, Inkscape, OpenOffice and even OpenStreetMap. We welcome new screencasts on any computing topic.

A brief history

We wanted a way to share geek-knowledge around the world 24/7 as if the friendly geek was sitting next to you. What we wanted didn't exist so we built our own!

The video tutorials are the foundation stone from which we'll build some larger ideas including on-line live group tuition. Kyran and I are a pair of Artificial Intelligence researchers and Python programmers who turned to teaching and we've attracted a group of keen authors who choose to work with us. Backed by years in academia and industry, we know how to teach so that our users learn about new topics quickly and easily.

Notable Python Series and Authors

Over the next few pages, we will be introduced to some of the authors who have contributed to ShowMeDo.

Notable Author: Kyran Dale

Kyran (<http://showmedo.com/videos/?author=8>) is the co-founder of ShowMeDo and has authored our most-popular-ever series which is an introduction to wxPython for new GUI programmers (<http://showmedo.com/videos/series?name=PythonWxPythonBeginnersSeries>).


watch videos ▾ make videos ▾ home about contact us resources ▾ Google™ Custom Search Search

Python GUI Programming with wxPython (5 videos) a series by [Kyran Dale](#)












A series of videos for beginners showing how easy it is to quickly develop professional looking graphical user interfaces exploiting the ease and power of the Python programming language. wxPython is fast becoming one of the most popular cross-platform GUI programming solutions. This set of videos aims to show you why.

How to Download and Install wxPython


Here we take you through the simple steps required to have wxPython up and running on your Windows machine.



11 Members Liked this Video










 Ivan Ozsvald	 Brewdos	 Ronan	 rj.walsh	 Kyran Dale
 Anders Persson	 Rhochi Phiri	 Ryan Galbraith, GISP	 toastandlove	 Jack Palmer
 Steven				

A 'Hello World' Window in Five Easy Lines



Name: [Kyran Dale](#)
Member: 28 months
Authored: 6 videos
Description:


67 Members' Votes for this Series

 Ivan Ozsvald	 Brewdos	 Ronan
 rj.walsh	 Kyran Dale	 Anders Persson
 Rhochi Phiri	 Ryan Galbraith, GISP	 toastandlove

[show me more...](#)

Other Videos You Might Like

By Same Author	Recommended	Related
-----------------------	--------------------	----------------



none
 Kyran Dale and Ivan Ozsvald
 Doug asked us to prepare a talk for the 2007 PyCon plenary sessions to explain our goals. Kyran and I talk about why we [...]

Notable Author: Ian Oswald

Ian (<http://showmedo.com/videos/?author=2>) is the other co-founder of ShowMeDo. Ian has authored 64 videos. Most are for Python including long Python tutorials with exercises for subscribers.

One of Ian's most popular series is the subscriber-only 'Python 101 - easygui and csv' which takes a new Python programmer through the task of building a simple user-interface to read a comma separated file of numbers. The numbers are manipulated, a new file is generated and unit-tests are used to verify that the program works exactly as expected.

watch videos ▾
make videos ▾
home
about
contact us
resources ▾
Google™ Custom Search
Search

Python 101 - easygui and csv

(17 videos)

a series by [Ian Ozsvald](#)

Aimed at Python Beginner Programmers, this fully-worked series builds a complete application that utilises a User Interface, the comma-separated-values (csv) module, reads and writes files, handles exceptions and gives useful error messages, uses test-driven development, nosetests and refactoring. We do all of this inside Wingware's excellent Wing IDE.

"Very well explained. I've learned a lot about debugging code. The ability to print the code, work through the problem and review the screencast, makes for clear understanding of what's going on. Thanks." - [Jshack](#)

"It's a NO BRAINER to get this quality of tutorial [...]. Everyone, who starts learning python, should get it. Excelet job. Keep up with a good work." - [nicholas](#)



Name: [Ian Ozsvald](#)
Member: 28 months
Author: 64 videos
Description: I am co-founder of ShowMeDo (see <http://showmedo.com/about>). By trade I'm a programmer, A.I. resear ...

105 Members' Votes for this Series

 Jshack	 notatestuser	 cybervolk
 Peter M Samson	 Thomas	 MickL
 dami1976	 JZA	 richard1956

[show me more...](#)

Subscribe to Start Learning Python the Easy Way!


Series Overview in 5 minutes (Free)

We look over what is covered in the series and what you'll learn in the following 1 hour 20 minutes of tutorial screencasts.



12 Members Liked this Video

 Jshack	 notatestuser	 cybervolk	 Peter M Samson	 Thomas
--	--	---	--	--

Other Videos You Might Like

By Same Author
Recommended
Related



none
Ian Ozsvald

Take a tour through Wingware's Wing IDE Professional. See how to use Code Completion when writing code, use the

Notable Author: Siddhi

Siddhi (<http://showmedo.com/videos?author=1982>) and 'empty' (<http://showmedo.com/videos?author=4405>) have authored a number of Django tutorials (<http://showmedo.com/videos/django>). Siddhi's 'Create a wiki in 20 minutes' (<http://showmedo.com/videos/series?name=v7kABKL6R>) has been one of our most popular series since its addition last year.

The screenshot shows a video player interface. At the top, there is a navigation bar with links: watch videos, make videos, home, about, contact us, resources. A search bar is on the right. The main content area is titled 'Learn Django: Create a Wiki in 20 minutes tutorial by Siddhi'. The video frame shows the Django project website with a play button in the center. The website content includes the Django logo, a navigation menu (Home, Download, Documentation, Weblog, Community, Code), and a 'Meet Django' section. To the right of the video frame, there is a sidebar with a profile picture of Siddhi, his name, member status (7 months), and number of videos (3). Below this is a description of his work at Silver Stripe Software. There is also a PayPal donation link with the text 'Encourage the author(s) - reward them with a paypal transfer! 100% goes directly to the author(s)'. At the bottom of the sidebar, there is a 'Leave Some Feedback' section with a paragraph of text and a 'Quick comment' form with several checkboxes for feedback.

Notable Author: André Roberge

André Roberge (<http://showmedo.com/videos/?author=22>) is 'a Theoretical Physicist by training, a University President by accident, and a hobbyist Python programmer' and creator of the excellent Crunchy which runs a Python interpreter inside Firefox. His videos show you how to try Python code samples using a nice editor directly inside a web page.

The screenshot shows a web browser displaying a tutorial page for 'Crunchy Python'. The page has a green navigation bar at the top with links: 'watch videos', 'make videos', 'home', 'about', 'contact us', and 'resources'. A search bar is on the right. The main content area has a red header 'Introduction to Crunchy' and 'rompting Python' (partially visible). Below this, there's a video player showing a Python interpreter prompt with the code: `print "Hello World!"`. The video player has a play button and a 'show code' button. The text below the video says: 'I am assuming that you are viewing this page while using Crunchy. If not, all you will see is a somewhat traditional page with some black text in it. Any reference to a Python interpreter that I make will seem rather odd.' Below that, it says 'An almost typical introduction to Python' and 'I will not begin this introduction to Python the standard way, like it is done with all other languages, by asking you to write the traditional'. Then it shows the code: `print "Hello World!"`. Below that, it says 'No, this would just be too boring and neither worthy of Python nor of Crunchy. Instead, I suggest that you type the following code in the interpreter prompt that appears just below it.' Then it shows the code: `>>> answer = 211 // 5` and `>>> print answer`. Below that, it says 'Crunchy: Borg Interpreter (Python version 2.5.1). Type `crunchy.help` for more information.' Then it shows the code: `>>> print "Hello world"`. Below that, it says 'If you haven't made any mistake, Python will respond with **The Answer to The Ultimate Question of the Universe and Everything**, at least according to the Hitchhiker's Guide to the Galaxy. The Python interpreter you just used was actually inspired by and adapted from some code work by Robert Brewer. You may...'. The sidebar on the right has a profile for André Roberge, a 'Leave Some Feedback' section, and a 'Quick comment' form.

Notable Author: Jeff Rush

Jeff Rush (<http://showmedo.com/videos/?author=709>) has been the chair of PyCon 2006 and 2007, the Python Advocacy Co-ordinator and organises the Dallas-Ft. Worth Pythoners. His 15 videos will teach you Python via walk-throughs, show you IPython in action and teach you to screencast.

The screenshot shows a video player interface. At the top, there is a navigation menu with links: watch videos, make videos, home, about, contact us, resources. A search bar is also present. The main content area features a video player with the title "The 'IPython' Interactive Shell - Part 1" and "tutorial by Jeff Rush". Below the title, there is a Python logo and the text "A Demonstration of the 'IPython' Interactive Shell - Part 1". The author information is listed as: Author: Jeff Rush, Copyright: 2007 Tau Productions Inc., License: GNU Free Documentation License, Date: July 17, 2007, Version: 1. A description follows: "A usage walkthru showing the use of the enhanced interactive shell named IPython. Divided into five parts, this talk in the series will cover tab completion, namespace management, logging, the help system and introspection." The video player controls are visible at the bottom. On the right side, there is a sidebar with a profile picture of Jeff Rush, his name, membership status (11 months), and a description of his role. Below this is a PayPal link to encourage the author, followed by a "Leave Some Feedback" section with a "Quick comment" form containing several checkboxes for feedback.

Notable Author: Lucas Holland

Lucas Holland (<http://showmedo.com/videos/?author=79>) covers both English and German open-source tools and languages with his friend Marius Meinert and has produced 35 tutorials. They make screencasts from two locations in Germany joined via Skype and a collaborative recording environment.

The screenshot shows a video player interface for a tutorial titled "Objektorientierte Programmierung (OOP) tutorial by Lucas Holland and Marius Meinert". The video content displays the title "Objektorientierte Programmierung" and "OOP" on a green background. The video player includes a "show medo" logo in the bottom right corner and a progress bar at the bottom. To the right of the video player, there is a sidebar with author information for Lucas Holland and Marius Meinert, a "Leave Some Feedback" section, and a "Quick comment" section with a list of feedback options.

watch videos ▾ **make videos** ▾ **home** **about** **contact us** **resources** ▾

Objektorientierte Programmierung (OOP)
tutorial by [Lucas Holland](#) and [Marius Meinert](#)

Objektorientierte Programmierung
OOP

show medo

Name: [Lucas Holland](#)
Member: 13 months
Author: 35 videos
Description: Hey, I'm a geek just like you ;-)
No, seriously I do have a real life... I enjoy programming, screencasting and spending time with my friends. I maintain a blog together with Marius at: <http://www...>

Name: [Marius Meinert](#)
Member: 13 months
Author: 30 videos
Description: I am just one of this internet-addicted persons, that lives in the net. I have a website with Lucas, that's on monroebrothers.org

Leave Some Feedback

Our authors tell us that feedback from you is a big motivator. Please take a few moments to let them know what you think of their work.

Quick comment

anonymous quickly comments:

- You get my vote!
- Please make more on this subject!
- *Really* useful, thanks!
- Thanks - you saved me some time
- I appreciate your effort
- Feedback - Volume is too low

Notable Author: Horst Jens

Horst Jens (<http://showmedo.com/videos/?author=71>) was previously featured in Volume 2 Issue 2 of the Python Papers. He has authored 25 videos with various students at HIT (www.hit.co.at) and other schools on Python and open-source topics.

Whilst many of Horst's videos are in English, some are English-subtitled with spoken German or Chinese. Horst often uses an interesting technique - his students are shown in a small web-cam window to the side of the screencast so you can see them talking as they present.

The screenshot displays a video player interface for a tutorial titled "Ubuntu: Making a .deb package out of a python program" by Horst Jens. The video content is as follows:

- from .py to .deb : Preparation**
- 2006/11 by Horst JENS, pagamebook@gmail.com, technical Advisor: Alexander Hungenberg
- rockpaperscissors16.tar.gz
- Prepare applications to make a .deb package out of a python program:
 - Terminal
 - File-Manager
 - Text Editor
- Install programs:
 - dpkg-dev
 - dh-make
- Learn how to use commands:
 - cd ..
 - cp
 - mv
 - rm
- Note: in all commands on the next pages, replace rockpaperscissor, rockpaperscissors16, sandbox etc. with your own python program or folder names.

Terminal snippets shown in the video:

```
$ sudo apt-get install dpkg-dev
$ sudo apt-get install dh-make
```

```
$ man cd
$ man cp
$ man mkdir
$ man rm
```

The video player interface includes a navigation bar with links: watch videos, make videos, home, about, contact us, resources. A search bar is present with "Google Custom Search". The video progress bar shows 01:02.

Name: [Horst Jens](#)
Member: 13 months
Author: 25 videos
Description: Hi, my name is Horst JENS, living in Vienna, Austria. I work as a programmer (python) and computer teacher and hope to introduce my students to the joy of making showmedo videos and getting good ...

Encourage the author(s) - reward them with a paypal transfer!
 100% goes directly to the author(s)

PayPal

Leave Some Feedback

Our authors tell us that feedback from you is a big motivator. Please take a few moments to let them know what you think of their work.

Quick comment

anonymous quickly comments:

- You get my vote!
- Please make more on this subject!
- *Really* useful, thanks!
- Thanks - you saved me some time
- I appreciate your effort
- Feedback - Volume is too low
- Feedback - Visual quality could be improved
- Feedback - Your speech is a bit too quick

Case study - Erik and 'developing emol!'

Erik Thompson (<http://showmedo.com/videos/?author=116>) has authored 2 series for ShowMeDo. The first was on the 3D Python physics engine VPython, the second is for a 3D molecule viewer (using wxPython and PyOpenGL). Both series are well over an hour long. Erik writes on why he chose to host his videos with ShowMeDo...

I started listening and watching podcasts a couple years ago and had the urge to put something out there myself. I wound up making a video tutorial on the easy to use 3D Python programming library called VPython (<http://vpython.org>). In the video series I used VPython to perform various physics simulations. I wanted to review my physics and I figured a good way to review is to try and explain the subject to someone else. So, rather than go find someone nearby who would listen to me ramble, I made the video tutorial series. Creating a video tutorial geared towards a Python programming library certainly limited my potential audience compared to creating something like a general tech podcast, but niche subjects allow for deeper and more interesting content than is generally found in the more popular podcasts out there. I'm continually amazed at the number of people who actually do watch the video tutorials and have posted and emailed positive feedback.

That series was created before I knew about ShowMeDo and I pretty much had to maintain a website for it myself and viewers had to download large avi files, because I had no idea how to embed a flash player of decent quality into my website. A major reason I decided to use ShowMeDo to host my video series is that ShowMeDo makes screencasts actually look reasonably good when watched through a flash player. Screencasts on YouTube tend to look like a blurry mess compared to ShowMeDo because YouTube uses of a smaller player and codecs that are optimized for general video rather than for a screencast. ShowMeDo also allows users to download the videos so they can watch them in their favorite media player. Another big reason I wanted my videos on ShowMeDo is that they have a Python and FLOSS heavy community which makes it easy for potential viewers who might be interested in my Python videos to come across them.

Recently I started a new series called Developing emol! that is meant to be a cross between a tutorial and a documentary for creating an Open Source application. The series differs from many other Python tutorials out there in that rather than simply focusing on a single programming library it attempts to show some of the other things involved with creating a program. Developing emol! intends to cover everything from design, having the project hosted, using the Bazaar version control software, coding, and deb package creation. The final program will be a 3D molecule viewer built with PyOpenGL, wxPython, and SQLite and hosted on Launchpad (<http://launchpad.net/emol/>). My secret hope for the video series is to attract future developers interested in enhancing or branching off their own version of the molecule viewing program. What better way to learn about getting involved with an Open Source program than by seeing a documentary of actually how it was designed and coded? As a student at California State University Long Beach, I'm involved with research involving computational chemistry and I plan on eventually getting into computational drug design. The program has the potential to become useful for my research.

My other hopes for the series are simply to encourage others to create their own Open Source programs and to encourage others to create their own interesting video tutorials/documentaries for me to enjoy watching. It's pretty easy to get started. All you need is a microphone and to install any of a number of free software packages available to create the screencast. Once you've recorded your tutorial (usually as an avi file) ShowMeDo makes the rest of the job extremely easy by re-encoding it for use in their flash player and making it available in their friendly community. It's pretty fun to have stuff out there that you've created so I encourage the readers of The Python Papers to give it a try!

Learn Python the Easy Way with a ShowMeDo Subscription

Our goal has always been to help document open-source software (especially Python!) and to ease the learning-curve for a new user. Kyran and I decided during late 2007 to work on ShowMeDo full-time, both building the environment for our authors and delivering new tutorials under a subscription package.

We are completely focused on our goal of making Python easy to learn for everyone - especially for new users. We'll show you the important modules, walk you through building

useful applications, help you learn to debug and unit-test and we'll test your knowledge with exercises and worked-solutions. We've done the research so you don't have to.

"As Ian says somewhere, he's found, organized, and presented information that would take weeks or months to learn in the ordinary way. Even if your time is worth only \$10 per hour, I figure the value in time saved is at least \$800. The skills are priceless." - Vincent DiCarlo on our first Subscriber series (Python Development on XP).

Our current topics include graphical user interface with wxPython and easygui, file reading/writing, using IDEs, debugging, and refactoring. Future topics will include writing web and desktop applications, reliably testing websites, controlling Windows applications using pyWin32, writing GUIs, talking to databases, writing games and more.

The most recent subscriber-only series teaches the viewer to build a wxPython-based Image Viewer (<http://showmedo.com/videos/series?name=To3wW0reK>) and runs for over an hour. The viewer learns how to build a skeleton wx application which they can reuse, then add menus, a splash screen and more. The series includes 3 exercises with fully-worked solutions.

The screenshot shows a video player interface. At the top, there are navigation links: 'watch videos', 'make videos', 'home', 'about', 'contact us', and 'resources'. A search bar with 'Google Custom Search' is on the right. The video title is 'Overview - building your own Image Viewer' by Ian Ozsvald. The video content shows a Windows command prompt running 'python ImageViewer.py', which opens a window titled 'C:\wxPythonImage\ C:\wxPythonImageViewer\337736441_a7095cdd33.jpg'. The window displays a sunset image of a pier over the ocean. The video player has a play button and a progress bar. On the right side, there is a profile for Ian Ozsvald, including his name, membership duration (28 months), number of videos (64), and a description. Below the profile is a 'Leave Some Feedback' section with a 'Quick comment' form containing several checkboxes for feedback options like 'You get my vote!', 'Please make more on this subject!', etc. There is also an 'Email' field and a 'Write a review' button.

The plan costs the same as several cups of coffee per month. If you need to refresh your skill-set and you like the idea of learning at your own pace, backed by helpful admins who can answer your questions and guide you forward then please visit our Subscriptions page to learn more (<http://showmedo.com/subscribe>).

Additions to the main site

We run two Google Groups, use a blog for video-announces (which goes out to the Planet Pythons) and have a wiki for additional material.

Our first Google Group is focused on Authors - come here to talk to the Authors and learn about screencasting on Linux, Mac and Windows:

<http://groups.google.com/group/showmedo>

Our second Group, just recently established, is focused on users who are learning new skills. It enables you to ask questions to other users who have watched the same videos as you:

<http://groups.google.com/group/showmedo-learners>

The blog is used to post notifications to the two Planet Python aggregators announcing new Python series to the world at large:

<http://blog.showmedo.com/>

The wiki holds additional information including some source-code, wallpaper, notes on video making and editing and background material on some of the videos:

http://wiki.showmedo.com/index.php/Main_Page

We also announce videos to relevant mailing lists on behalf of our authors including wxPython-users and Python's Tutor list. Whilst we encourage our Authors to publicise their own videos we help them out as much as possible.

Getting Involved

Why would you want to make videos? Perhaps you want to give back to a favoured open-source project by showing how it works - this is especially useful for the more complex tools and modules.

You might want to encourage others to try the project, maybe you want to show why this project is better than another. Whatever your reason - we are interested in hosting your screencasts and helping you find the right viewers.

Users love to sit back and be shown exactly why they should use a tool and they're not afraid to say Thank You - our authors receive over 160 comments each week (<http://showmedo.com/recentComments>) from satisfied users. By sharing your knowledge you'll help to promote the project you show and you'll build your own reputation in the community.

Seen recently on <http://showmedo.com/recentComments>:

"I think this is by far the most excellent understanding I have had on how python works. I bought multiple books, but the problem with every book is that they repeat the same fundamentals without giving an overall perspective of how a programming setup works."

"Yet another very nice & useful episode. The explanation of destroying objects and why you are supposed to destroy them was very clear."

"I really enjoyed the tutorial. I'm new to python, and programing in general. I picked python because after looking over C# Java and Python, the general view is that Python is a good begininer langauge"

You'll find 'Add a video to ShowMeDo' in the navigation bar under 'make videos' (<http://showmedo.com/addVideoInstructions>). This page includes notes on topics, resolutions, video-lengths, techniques and screencasting videos. It also links to the Author's Google Group where you can ask questions and learn how other's make their videos.

When you submit a video note that we check everything that comes into the site. It can take us up to a day or so to get back to you. We only do the check to make sure no rubbish gets posted. Don't be afraid to post a work-in-progress if you'd like feedback too.

For your first video we recommend making just 1 short video, about 5 minutes in length, showing how to use a favoured tool or module. The reason for doing just 1 video is that it is much easier to complete a good single video rather than contemplate and possibly fail to finish a much more ambitious series on the first attempt.

You can embed the resulting video in your own blog or the site of your chosen module so it can be shown to exactly the right users.

You are free to set your own license (we recommend Creative Commons, ShareALike, NonCommercial). You own your own content, we don't ask for any rights-transfer (unlike some

other big-name video sites that could be mentioned...)

The Future

We are focusing our efforts on making it easier for people to learn Python at first (via our Subscriptions scheme) with a view to covering other topics (like Linux, Perl, JavaScript and OpenOffice) later. We'd love to help Python by having free videos covering all the major modules and tools - you could help us out here!

We plan to have videos covering all of the major packages within the next year, along with a really solid beginners video guide that introduces programming with Python for everyone.

One exciting new feature we're toying with is the integration of André Roberge's Crunchy into ShowMeDo so that you can write Python code inside Firefox whilst following a tutorial. Extra features which help learning - such as learning guides and tests - are also on our agenda.

What should you do now?

If you haven't seen ShowMeDo then please come by and take a look. Remember to leave a Thank-You comment for an Author - authors really love to know that you've appreciated their efforts. Next Visit one of our Google Groups and think about getting involved. We'd love to hear from you and help you out as you make your first ShowMeDos.

Useful Links

Series and Topics

<http://showmedo.com> - 500 tutorial videos

<http://showmedo.com/videos/python> - 177 Python videos, more every week

<http://showmedo.com/videos/wxpython> - 3 wxPython series

<http://showmedo.com/subscribe> - "Learn Python the Easy Way"

<http://showmedo.com/videos/series?name=L3dNy3tjR> - '5 Minutes with Python'

<http://showmedo.com/videos/series?name=DKKuA1cT6> - 'My Favourite App'

<http://showmedo.com/videos/series?name=vXJsRwIBX> - Erik's 'Developing emol!'

<http://showmedo.com/videos/series?name=PythonWxPythonBeginnersSeries> - 'Python GUI Programming with wxPython'

<http://showmedo.com/videos/series?name=To3wW0reK> - 'Build a wxPython Image Viewer'

<http://showmedo.com/videos/series?name=v7kABKL6R> - 'Create a [Django] wiki in 20 minutes'

Named Authors

Kyran Dale (<http://showmedo.com/videos/?author=8>)

Ian Ozsvald (<http://showmedo.com/videos/?author=2>)

Siddhi - <http://showmedo.com/videos?author=1982>

'empty' - <http://showmedo.com/videos/?author=4405>

André Roberge - <http://showmedo.com/videos/?author=22>

Jeff Rush - <http://showmedo.com/videos/?author=709>

Lucas Holland - <http://showmedo.com/videos/?author=79>

Horst Jens - <http://showmedo.com/videos/?author=71>

Creating Screencasts and Adding to ShowMeDo

<http://showmedo.com/addVideoInstructions>

<http://groups.google.com/group/showmedo>

Site Links

<http://showmedo.com/recentComments> - Over 600 comments to authors each month

<http://showmedo.com/mostPopular> - Over 43,000 video-plays each month

<http://pythonpapers.org/> - to see Horst's entry on ShowMeDo in Volume 2, Issue 2

An Efficient Scalar Package in Python

Russell A. Paielli

Abstract-- A Python class was designed to represent physical scalars and to eliminate errors involving implicit physical units (e.g., confusing angular degrees and radians). The standard arithmetic operators are overloaded to provide syntax identical to that for built-in numeric types. The scalar class does not define any units itself but is part of a package that includes a complete implementation of the standard metric system of units and many common non-metric units. It also allows users to define a specialized or reduced set of appropriate physical units for any particular application or domain. Once an application has been developed and tested, the scalar class can easily be switched off, if desired, to achieve the execution efficiency of operations on built-in numeric types, which can be nearly two orders of magnitude faster. The scalar class can also be used for discrete units to enforce type checking of integer counts, thereby enhancing the built-in dynamic type checking of Python. The scalar package is available from <http://RussP.us/scalar.htm>.

Introduction

Physical units and scalars are fundamental to scientific and engineering calculations and computations. All scientists and engineers learn to add, subtract, multiply, divide, and convert units, and to keep track of them with pencil and paper. When they program a computer, however, they usually drop the explicit units and leave them implicit in the actual numerical calculations, with perhaps a comment to document the units. They do that for two basic reasons: (1) software that automatically tracks and checks units, although available, is not widely standardized and well known, and (2) that software can, in some cases, drastically reduce computational efficiency. The scalar class presented here addresses those problems.

As a consequence of the lack of explicit units in most scientific and engineering software, mistaken units are a common source of error. Perhaps the most common such error involves passing an angle in degrees to a trigonometric function that takes it in radians. Humans tend to think in degrees, but standard trig functions take radians, and the conversion is often forgotten until its omission is discovered after time-consuming debugging. In one case, aircraft simulation results over a period of six months were corrupted by such an error. In another case, confusion between seconds and minutes was found in Python code three years after the results from it were published.

In air traffic management (ATM), horizontal distance is usually specified in nautical miles, whereas altitude is specified in feet, hundreds of feet, or thousands of feet. Horizontal speed is usually given in knots, whereas vertical speed is usually given in feet/minute. Such units can easily get confused if comments in the code are the only mechanism for enforcing consistency. Permitted units can be restricted by software coding standards, but draconian restrictions on permitted units can be inelegant and inconvenient, forcing error-prone conversions on input and output.

The approach taken here to prevent such confusion is to allow the user to select units that are appropriate for the job, then to track those units explicitly in software just as an engineer or scientist would do on paper. The scalar class itself does not specify any units, but a unit definition file can be used to define the preferred units and the relevant conversion factors for the particular application or domain. Two such definition files are included with the scalar package, one for metric (and other common units) and one for traditional ATM units. These files can be extended easily and also serve as examples for defining other sets of units.

When run in the interactive Python interpreter, the scalar class can serve as a “calculator”

that tracks units and checks consistency. No longer must engineers enter only the numbers into a calculator and manipulate the units separately in their heads or on paper. But the larger benefit of the scalar class is in automatically catching unit errors in Python software. For computationally intensive applications, the scalar class can be easily switched off for efficient production runs that produce the same results. Thus, the scalar class can be used during development and testing to guarantee unit correctness, then turned off to obtain the execution efficiency associated with built-in numeric types. The resulting improvement in efficiency, which will be discussed later, is nearly two orders of magnitude.

Several other software packages are also currently available in Python for representing and manipulating physical scalars with units, but this scalar class was developed independently. Any resemblance to other classes or packages is coincidental. The scalar class presented here is believed to be the first to provide the option to be switched off to realize the efficiency of operations on built-in numeric types. That capability could be a key to widespread adoption, because the large computational overhead involved with tracking and checking units is no longer a reason to avoid using them.

Usage

The scalar class is actually called “_scalar” because is not intended to be used directly. Instead, a function called “unit” (which calls the _scalar class constructor) is intended to be used to define units. But the user need not even call the unit function directly unless a new unit is needed that is not already available in the unit definition file included with the package. The units module (file units.py) defines a comprehensive set of standard units, including the complete metric (SI) system and many common non-metric units. To access those units, simply import the units module. Depending on how the installation was done, this import could be done either of the following two lines:

```
from scalar.units import *
from units import *
```

The latter form will be assumed for the remainder of this guide. The unit module imports the scalar module, hence the user should *not* import the scalar module directly. Doing so could cause problems if the scalar class is later disabled for efficiency, which will be discussed later.

The pre-defined units in the units module are as consistent as possible with standard unit abbreviations such as “s” for seconds and “m” for meters. Thus, for example, the scalar 23 m/s² would be “constructed” as

```
accel = 23 * m/s**2
```

These short identifiers are convenient for interactive “calculator” sessions or small applications, but they may be inappropriate for larger applications because short identifiers at global scope can cause problems with name clashes and inadvertent shadowing or overwriting of unit names. To avoid such problems, the units module can be imported with

```
import units
```

The meter object would then be referenced as “units.m.” A cleaner form can be realized with

```
import units as u
```

The meter object is then referenced simply as “u.m,” which saves on both typing and code clutter, while maintaining the separate namespace. Another way to avoid the short identifiers at global scope is to use the long unit names, such as “meters,” “seconds,” “milliseconds,” etc., which are also provided. See the units module for the complete set of pre-defined unit names.

In addition to the units module, which is fairly comprehensive, another smaller module called “ATMunits” is also included. It was designed for basic air traffic management (not actual operational ATM, but prototyping, testing, and data analysis). It provides an example of a simplified unit definition file for a particular application or domain without the overhead of all the unneeded units in the units module. The smaller number of units and the lack of single-letter unit names allows the ATM units to be used globally without causing significant problems. Users are also free to copy the units module and strip out what they don't need, of course.

As in the standard metric system of units, the base unit for length in the units module is the meter, and several common scaled variations of it are defined:

```
m = meter = unit("m") # base unit for length
mm = millimeter = unit("mm", m/1000)
um = micrometer = unit("um", mm/1000)
cm = centimeter = unit("cm", m/100)
km = kilometer = unit("km", 1000*m)
```

If only one argument is passed to the unit function, it creates a new base unit, but if two arguments are passed to it, it creates a derived unit that is defined by the second argument. Although not strictly required, the first argument should be the string version of the unit name on the left side of the assignment, as shown above (otherwise the “repr” function will not work properly). Only one base unit should be defined for each unit type, such as time or length. When inconsistent units are added or subtracted, an “InconsistentUnits” exception is raised.

Outputs will be printed by default in the base unit, such as meters for length, unless the user specifies otherwise. For example:

```
>>> dist = 5.2 * m
>>> dist += 27 * mm
>>> print dist
5.227 m
>>> print format(dist, "ft", "%.2f")
17.15 ft
```

The third argument for numeric formatting is optional. The format function raises an exception if the unit specified in the second argument is not of the correct type for the scalar passed in the first argument. The available output units are the keys of the “to_” dictionary in the units module. To print a scalar as a number without units, simply divide by the desired units to convert to a float for output. For example:

```
>>> print dist/ft
17.1489501312
>>> print "%.2f" % (dist/ft)
17.15
```

Note that the parentheses are needed here to get the correct operator precedence.

Users are free to define any set of units, but unit names should contain only alphabetic characters (lowercase and/or uppercase letters). Note however, that unit names with non-alphabetic characters will not raise an exception until they are printed out. [This allows output conversions to units with non-alphabetic characters, such as “ft*lb.”]

The convention for printing units with the “str” (string conversion) function is to show multiplication with asterisks, division with slashes, and powers with double asterisks, as expected. For example, “kilogram-meters/second-squared” would be shown as “kg*m/s**2.” Only one slash is used, and if the denominator has multiplied units they are placed in parentheses, as in “kg/(m*s**2).” These output conventions are consistent with the

requirements of the “repr” (representation) function, which produces a string representation of the object that can be parsed by “eval” to create a copy of the object. The repr function is identical to the str function except that it places an asterisk (multiplication operator) between the coefficient and the units, as in “23 * m/s.”

No function is provided to extract the numerical coefficient of a scalar independent of the units because that would depend on the base units chosen, and the output should not depend on the choice of base units. To write numerical data to a file without the units, or to pass data to a third-party function or application without the units, simply divide the scalar by the required unit. Suppose, for example, that a distance needs to be written or passed as a number with implicit units of feet. If the variable is named “dist,” then “dist/ft” will be automatically cast to a built-in type (typically a float) corresponding to the distance in feet.

A scalar cannot be cast to a built-in numeric type such as “float” unless it is dimensionless. The predefined units of “rad” (radians) and “deg” (degrees) in the units module are dimensionless, for example, and the base unit is radians. This convention guarantees that standard trigonometric functions will be passed arguments in terms of radians, as expected, and it prevents any other unit from being erroneously passed to a trigonometric function. For example, “sin(30*deg)” returns 0.5 as expected rather than being converted to “sin(30),” which would be wrong.

To facilitate the switching off of the scalar class for efficiency (to be discussed in the next section), the scalar class has no public member functions (other than the overloaded arithmetic operators). The reason is that calls of member functions using standard “dot” notation cannot work on built-in numeric types.

The scalar module imports the standard math module, and it re-defines the functions “sqrt,” “hypot,” and “atan2” so as to work correctly for both float and scalar arguments. Depending on how the unit definition module was imported, these functions may need to be accessed through the units module name. For example, if the import was done with “import units,” then the sqrt will need to be accessed as “units.sqrt.”

An interesting question arises with regard to the handling of units when the numerical coefficient is zero. When the coefficient is zero, the units shouldn't matter, and the scalar class was designed to reflect that fact. Thus, when a variable is initialized to zero, the units need not be specified, and any units can be added to zero. Also, an expression such as “0*m+4*s” will execute without an exception even though the units are inconsistent.

Disabling the Scalar Class for Efficiency

The scalar class can be used as a units “calculator” in the interactive Python shell, and it is also computationally efficient enough for many non-interactive applications. However, it may be too slow for some computationally intensive applications. Computational “overhead” can make arithmetic operations one to two orders of magnitude slower than corresponding operations on built-in numeric types such as “float” and “int.”

The source of that overhead is twofold. First, the character-string manipulation involved with tracking and checking the units obviously takes some time. But just disabling the unit tracking cannot increase the efficiency to the level of built-in types. The mere fact that scalar is a class rather than a built-in type also adds substantial overhead, slowing the execution speed by up to an order of magnitude even if unit checking is disabled within the scalar class.

Fortunately, a simple method has been devised to eliminate both sources of overhead. After an application has been tested and its unit consistency verified, the scalar class can be switched off or disabled for production runs. During testing and development the scalar class can be enabled all the time, but if the tests themselves are computationally intensive, it needs to be enabled only occasionally to catch the vast majority of unit errors.

The scalar class can be switched off by simply setting an environment variable called “scalar_off.” To switch it back on, simply unset that variable. A couple of bash shell aliases are useful here:

```
alias scalar_off='export scalar_off=yes'
alias scalar_on='unset scalar_off'
```

[The value assigned to scalar_off is irrelevant. It could just as well be nothing, but a non-null value makes it easier to determine whether the scalar class is enabled or disabled using the “echo” command in bash.] Note that the scalar class defaults to enabled if scalar_off is not set.

The use of an environment variable to enable or disable the scalar class means that it can be done with no changes to the Python client code. If the user prefers to enable and disable the scalar class from within the Python client code itself, that can be done as follows:

```
from sys import argv
from getopt import getopt
from os import environ # dict of environment variables

_scalar_off = 1 # default to disabled scalar class

opts, args = getopt(argv[1:], "", ["scalar_on", "scalar_off"])

for opt in opts:
    if opt[0] == "--scalar_on": _scalar_off = 0
    if opt[0] == "--scalar_off": _scalar_off = 1

_off = "scalar_off"

if _scalar_off: environ[_off] = "yes"
elif _off in environ: del environ[_off]

from units import *
```

The command line argument “--scalar_on” enables the scalar class, which is disabled by default here. This is most likely the proper default for “production” code, particularly if it will be shipped to a customer. If the scalar class is set to enabled by default, then “--scalar_off” can be used to disable it. This code provides another convenient way to control the status of the scalar class without modifying any client Python code. It should be placed in the “main” module before any unit definition file gets imported (either directly or indirectly through another import).

When the scalar class is enabled, the “scalar.py” module gets imported, but when it is disabled the “scalar_off.py” module gets imported instead. The latter replaces the unit function with a function of the same name that simply returns 1 for base units. Thus, the expression “25*m” would be replaced with “25*1,” or “25*km” would be replaced with “25*1000.” This replacement of the scalar class eliminates its overhead. It may leave a few unnecessary multiplications and divisions by one when base units are used, but that should not be significant unless they occur in high-rate, numerically intensive loops. If they do, the multiplication by the unit should be done ahead of the loop. In practice, explicit multiplication by units tends to occur mainly on input, and explicit division by units tends to occur mainly on output, so neither typically occur in high-rate loops. Assuming no unit inconsistencies, all outputs with the scalar class disabled should be identical to what they were with it enabled.

To quantify the speedup resulting from disabling the scalar class, a simple timing test was done in which scalars were added ten million times. The time required on a 2.0 GHz Dell Linux workstation was approximately 140 seconds. When the scalar class was switched off,

that time dropped to approximately 2.9 seconds, for a speedup factor of nearly 50, or nearly two orders of magnitude. Very similar results occurred for multiplying scalars.

Note that most of the benefits of the scalar package are still available even when the scalar class is “disabled” for efficiency. The only thing missing with the scalar class disabled is the automatic checking for addition, subtraction, or comparison of scalars with incompatible units. With the scalar class disabled, seconds can be added to meters without raising an exception, for example. But with the scalar class disabled, the scalar package still automatically converts everything to standard base units, so the user need not worry about base units or conversion factors. Thus, confusion between different units representing the same physical quantity will still be avoided, and those kinds of errors are probably much more common than adding or subtracting incompatible units. Confusion between angular degrees and radians will still be avoided, for example, as will confusion between seconds and minutes or feet and meters.

When using the scalar package in the interactive Python shell, efficiency is unlikely to be an issue, hence the user is unlikely to want to disable the scalar class. If the `scalar_off` environment variable is unintentionally left set, however, the scalar class will be disabled, which could be annoying. To prevent such an annoyance, a short file called “units_on” is provided. If the “PYTHONSTARTUP” environment variable is set to point to this file, it will automatically execute whenever the interactive Python shell is started. This file not only guarantees that the scalar class is enabled for interactive usage, but it also automatically imports the units module for convenience. To use this file, place the following in your “.bashrc” file:

```
export PYTHONSTARTUP=~/.scalar-x.x/scalar/units_on.py
```

Enhanced Dynamic Type Checking

The scalar class can also be used to implement a stronger form of dynamic type checking than is provided by the built-in dynamic type checking in Python. Suppose, for example, that a count of “bars” needs to be maintained and perhaps passed to a function or object. The user can simply define a unit for it and use it like any other unit:

```
bar = unit("bar")
count = 0
...
count += bar
```

If a count of type “bar” is then erroneously added to an incompatible type, the error will be detected and flagged immediately, possibly saving substantial debugging time.

PyGTK, PyQt, Tkinter and wxPython comparison

Guilherme Polo

Python offers a multitude of GUI toolkits, much more than described here, for assisting on development of graphical applications, and by having so many options available the chances are you will be unable to make a good decision. A good decision would be one that fulfill your requirements, and in order to achieve this it is necessary an understanding of choices available.

If you are just beginning GUI development, sometime you will need to select a toolkit and a lot of questions will eventually pop up, so I expect this article to help you making a sane decision. And if you already do GUI development and are considering learning another toolkit, or maybe you are moving from another language to Python, I, again, expect this text to help you choose your next tool.

Writing about every possible point of comparison is not possible (maybe it would fit in a book, a large one), so I have chosen to talk about some topics that you may face in your role as GUI developer. Options will be given, and you will balance them according to your requirements.

Each toolkit presented here has both strong and weak points, and it is, in fact, up to you to decide which one fits your needs. All the four major GUI toolkits available for Python are discussed on this text: PyGtk², PyQt³, Tkinter, wxPython⁴.

Look and Feel

The main task of a GUI developer is to build applications that are easy to use and, to achieve that, they have to be designed to be familiar. Knowing your users is a big step towards success, and this is not really easy to achieve, it involves research, experience, dedication, effort, and goes on.... Here follows a good paragraph that will remind you why it is important to know your users, taken from "Designing Interfaces, By Jenifer Tidwell. Chapter 1: What Users Do":

It starts with an understanding of people: what they're like, why they use a given piece of software, and how they might interact with it. The more you know about them, and the more you empathize with them, the more effectively you can design for them. Software, after all, is merely a means to an end for the people who use it. The better you satisfy those ends, the happier those users will be.

As a developer it should be interesting to follow some guidelines to make it easier to develop pleasant interfaces. Namely there is the Gnome HIG⁵ (Human Interface Guidelines) and Apple HIG⁶ that are widely used. Both describes good design tips that you should use while developing user interfaces, independent of what platform you are planning to focus on.

If you are going to use any of the toolkits presented here, it is good to know that your application will not look the same across platforms and may not feel the same. If you want this kind of thing, you should be looking for Lightweight GUI toolkits. They provide uniform behavior on all platforms with the disadvantage of slower execution.

2 PyGtk site: <http://www.pygtk.org/>

3 PyQt site: <http://www.riverbankcomputing.co.uk/pyqt/>

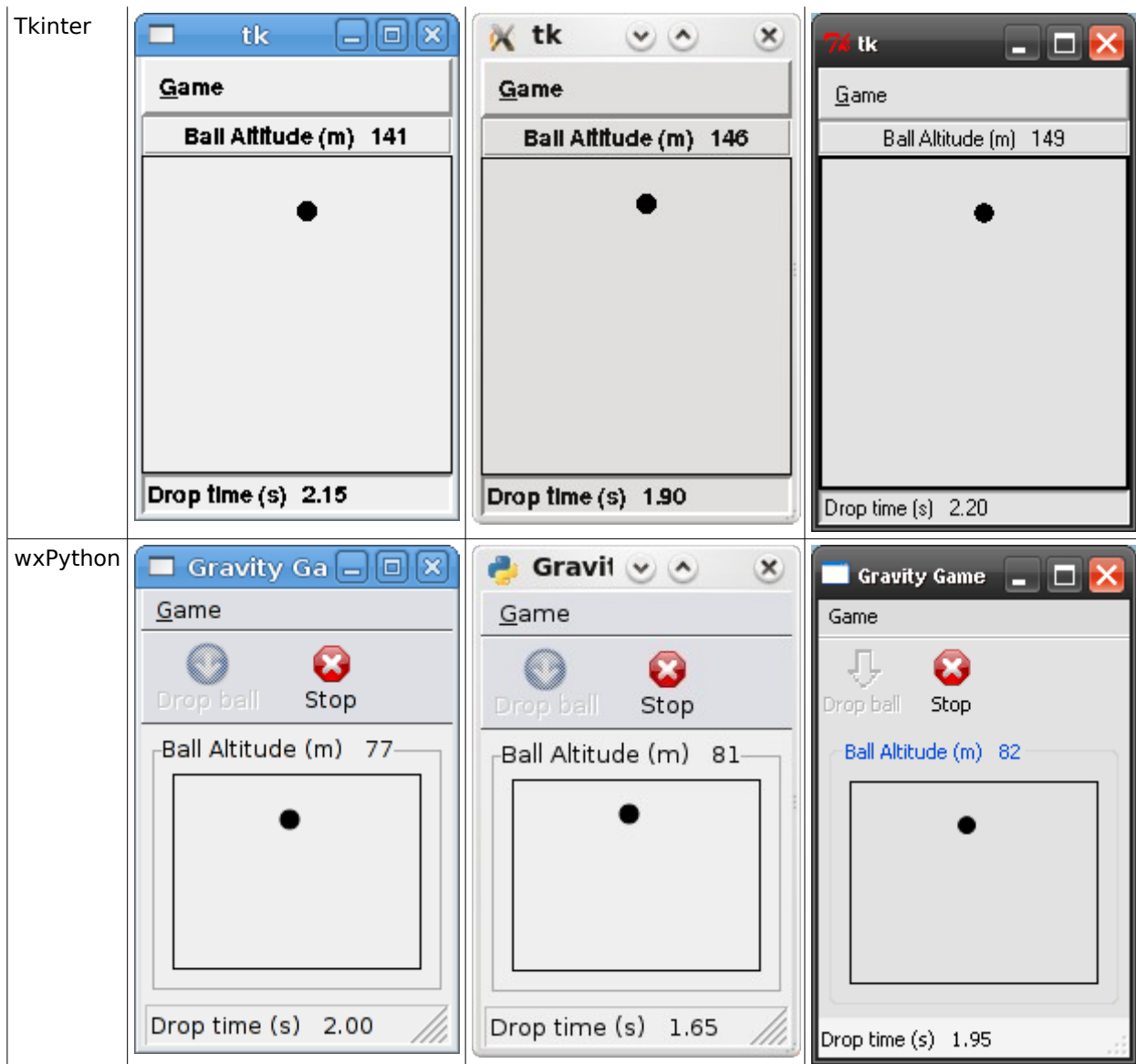
4 wxPython site: <http://www.wxpython.org/>

5 Gnome HIG documentation: <http://developer.gnome.org/projects/gup/hig/>

6 Apple HIG documentation: http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/XHIGIntro/chapter_1_section_1.html

By now, I believe it is important to show some screen shots. They were taken in three different platforms and should give an idea of how different your applications may look across platforms. This is too a first chance to select which one is more likely to fit your users.

	Gnome 2.20	KDE 4	Windows XP
PyGtk			
PyQt			



Source code for shown applications are available at http://gpolo.ath.cx:81/download/gravity_game. This game is a simplified version of "The Gravity Game", you can find the original source at book "Physics for Game Programmers", Chapter 3 (in Java).

GUI Customization

The word "customization" can take different meanings according to the context, so, for this section, it's meaning will be defined as: "capacity to modify default behavior of the widgets contained within the toolkit, be these modifications beneficial or even harmful to users."

Said that, be sure customizations are needed before doing them. Remember it is important to keep your application familiar to the users, so there must be a really strong point to opt for customized look and feel. But depending on the kind of program you are developing, it may make sense to choose for this. For example, music and video players usually uses custom interfaces, therefore it is possible you could want customized look and feel too.

I classify customizations based on what they do: Compromising customizations, Pleasant customizations and Runtime customizations.

- Compromising customizations
 - Reasoning for this naming: Removes application's familiarity.

- Typical changes: Shaped widgets, changing widget colors, custom fonts.
- Pleasant customizations
 - Reasoning for this naming: Any toolkit should provide in order to be useful and the user probably will feel better with these changes.
 - Typical changes: Possibility to change the label position relative to an image in a button, changing toolbar position, general changes that could be done to widgets available so they fit better your project.
- Runtime customizations
 - Reasoning for this naming: Customizations are done while the program is running.
 - Typical changes: Widget placement, applications's appearance.

Note that Runtime customizations lists as typical change applications's appearance, this would fall in Compromising customizations. But it was user's choice to do it, and he probably feels better with these changes, making it a good thing.

It is not interesting to discuss Pleasant customizations here since all toolkits presented provide them. Some of them supports more than others, mainly because they contain more widgets thus allowing more possibilities of customizations.

Lastly note that Compromising customizations may be useful too, they can help low vision users (for example). So they are not always dangerous or bad.

PyGtk customization

- Compromising
 - Themes are the way to go for large customizations under PyGtk. Users can choose a new look for your application just by setting a different theme (requiring no code changes).
 - PyGtk supports creation of shaped widgets.
- Runtime
 - It is possible to change themes easily while the application is running.
 - There is no support for dock widgets, so the user cannot freely relayout his applications. If you need this kind of customization, there are wrappers for GDL library which supports this functionality.

PyQt customization

- Compromising
 - PyQt uses Styles instead of Themes to achieve large customizations.
 - This toolkit also supports creation of shaped widgets.
- Runtime
 - Like in PyGtk, it is pretty easy to change styles while the application is running.
 - PyQt contains a widget called QDockWidget allowing your users to freely move and reposition widgets.

Tkinter cstomization

- Compromising
 - Tkinter allows widgets fonts and colors to be customized, but not as easily as using Themes or Styles.
 - Shaped widgets are not supported by this toolkit.
- Runtime
 - Not available at all.

wxPython Customization

- Compromising

- Like Tkinter, this toolkit supports changing widgets fonts and colors but not as easily as using Themes or Styles. wxPython may adapt its look and feel based on changes done on your platform.
- Shaped widgets are supported.
- Runtime
 - wxPython provides a library called AUI (Advanced User Interface) allowing dockable floating frames (like PyQt dock widget) and much more.

Summary

Designing good user interfaces requires a decent amount of effort, following some guidelines may help you and will make your users happier.

Customizations are not always welcomed by users, make sure you need it and your applications benefits from this before diving into this area. If you believe they are needed, both PyGtk and PyQt got more general solutions. If you are not planning changing the entire interface through themes, wxPython may be good enough to you as well. Lastly, if you do not need fancy customizations, Tkinter may be a choice.

Web Integration

As time passes by, it gets harder to ignore how much the Web is part of our daily lives, and if it is not important for you yet, it is possible that in some time it will. It just keep growing (only Google indexes several billions of pages), all the time.

For this reason, it may be possible that you will want (or need) to integrate Web resources into your Desktop applications. Some programs already do it, be it a small part or even the largest and most important part of the GUI application.

If you are planning to use Web resources (HTML, CSS, Javascript, Flash, ...) in your Desktop applications, you need to be more careful when choosing among the toolkits just in case frustration, anger, and co. are not your friends. But, be warned: depending on the level of resources and portability desired, there may be no choices at all.

PyGtk

PyGtk comes with Pango that uses a SGML-like markup language that allows you specify attributes with the text they are applied to by using a small set of markup tags. This is helpful for displaying applications's documentation or simple HTML pages, for example. With Pango it is possible to build a humble HTML renderer.

Next options are external libraries.

For rendering and/or printing simple HTML documents, you would need GtkHtml (python-gtkhtml bindings⁷). If you need CSS 1,2 and/or DOM 1,2 then GtkHtml2 would be more appropriate, but unfortunately I couldn't find any documentation for GtkHtml2. Even if you do not need documentation at all to learn how to use it, you should worry about its lifetime. It looks like both GtkHtml and GtkHtml2 are not being developed anymore (but it still works), so I wouldn't pick them. Platform supported (GtkHtml, GtkHtml2): Linux.

And there is GtkMozEmbed that allows you to embed a Mozilla browser window into your Gtk application. Its API is severely limited, it doesn't provide printing or viewing the page source (as examples of the limitation). But if you do not need this kind of service, I would opt for using it. The reason for this choice is the support for all Web resources your standard web browser does, and it's very easy to use. Lastly, it seems to have been used only in Linux for now.

⁷ python-gtkhtml bindings site: <http://www.fcoutant.freesurf.fr/python-gtkhtml.html>

PyQt

PyQt's components QLabel, QGraphicsTextItem, QTextEdit and QTextBrowser supports HTML (a subset of HTML 3.2 and 4) rendering. It isn't intended to support all features a browser does, but at least it is multi-platform.

Furthermore PyQt supports ActiveX through QAxContainer making it possible to embed Internet Explorer ActiveX control into your application. You get all features you might need in exchange of being limited to Windows and it is only available for commercial versions of PyQt.

Tkinter

Just like you could do in PyGtk (use Pango to build a simple HTML renderer), you can do one in Tkinter using its Text widget. Next to this there is the possibility of using Python's built-in module webbrowser⁸.

If using external libraries is not a problem, give TkHtml⁹ and its Python wrapper¹⁰ a try. I've ran a sample that is included but didn't enjoy it all, it doesn't allow you to select text, among other things. Lastly there is TkHtml3¹¹ that gives cooler features, like CSS support, but there is no Python wrapper yet.

wxPython

This toolkit provides some components that supports basic HTML. This includes HtmlListBox and HtmlWindow that serves as a basic and limited HTML render (like GtkHtml and QTextEdit). The good thing is that they are both multi-platform.

It also provides ActiveX support, and a component called IEHtmlWin that is an ActiveX IE window embedded in a wxWindow.

Summary

Integrating web in your desktop application using any of these toolkits and expecting them to work everywhere with whatever you desire is just a hope nowadays. If you just need plain and simple HTML rendering, wxPython and PyQt will serve you well. If you are targeting Linux, PyGtk with GtkMozEmbed is your best shot. And if you are focusing on Windows users, wxPython will provide all features you might need through ActiveX (at no charge).

Future

It seems WebKit¹² will save you (some time in the future). There is support for gtk, qt, wx and a project called wxWebKit¹³ that already provides bindings for wxPython. And hopefully it will run in all platforms you expect to.

Licenses

Licenses affect your work more than you can imagine, if you don't do already. You could want to hide the source, or keep any derivative works free, or maybe just put it in the public domain, or other uses, but to do so, you need to use the correct license. Choosing an appropriate license is not always easy, it may even require a lawyer, but this is out of scope of this article.

8 Python module webbrowser documentation: <http://docs.python.org/lib/module-webbrowser.html>

9 Tkhtml site: <http://www.hwaci.com/sw/tkhtml/>

10 Tkhtml Python wrapper: <http://tix.sourceforge.net/Tixapps/src/Python/TkHtml.py>

11 Tkhtml3 site: <http://tkhtml.tcl.tk/index.html>

12 WebKit site: <http://webkit.org/>

13 wxWebKit site: <http://wxwebkit.wxcommunity.com/pmwiki/index.php?n=Main.HomePage>

If you are worried about spending money, do not be. All the toolkits give you opportunity to not need to invest any money on licenses, but depending on your kind of application distribution and chosen toolkit, there may be some price to pay for.

PyGTK License

PyGTK uses LGPL (Lesser General Public License), this license is a middle term between GPL and permissive licenses such as BSD and MIT licenses. The main difference between the GPL and the LGPL is that the latter can be linked to a non-(L)GPLed program, which may be free software or proprietary software. This means that you may choose to not distribute the source along with your program. If you are in doubt to use LGPL, check out <http://www.fsf.org/licensing/licenses/why-not-lgpl.html>.

PyQt Licenses

PyQt follows Trolltech's license model, that means you will use different licenses based on how you use PyQt.

If your program is GPL compatible, then you do not need to buy a commercial PyQt license. Being compatible with GPL means, among other things, that:

- You will be providing the source code for your application
- All modified and extended versions of the program will continue being free
- Users are allowed to re-use, modify and re-distribute the code

Note that does not mean you can't sell copies of the program for money. There is a GPL FAQ that you should read in case of doubts: <http://www.fsf.org/licensing/licenses/gpl-faq.html>. Previous to PyQt v4, this free version was only available for Linux and Mac OSX.

In case your use of PyQt is not compatible with GPL then you will need a commercial PyQt license. Buying a license for PyQt does not include Qt licensing, so you must also purchase copies of the commercial edition of Qt from Trolltech. More informations on buying PyQt License can be found at <http://www.riverbankcomputing.co.uk/pyqt/buy.php> and for Qt at <http://trolltech.com/products/qt/licenses/licensing>.

Tkinter License

Tkinter is bundled with Python and also uses the PSFL (Python Software Foundation License). This is a permissive free software license which is compatible with the GNU General Public License (GPL). Its primary use is for distribution of the Python project software. Unlike the GPL the Python license is not a copyleft license, and allows modifications to the source code, as well as the construction of derivative works, without making the code open-source.

wxPython License

wxPython uses the same license as wxWidgets, the wxWindows License. This license is essentially the L-GPL (Library General Public License), with an exception stating that derived works in binary form may be distributed on the user's own terms. This is a solution that satisfies those who wish to produce GPL'ed software using wxWidgets/wxPython, and also those producing proprietary software.

Summary

If you don't even want to think about licenses in your way, Tkinter or wxPython would be the first option. Next to this is PyGTK, its license basically only forbids distribution of static linked libraries without neither source code nor linkable object files, so, if you are not considering this kind of distribution, it's all good. Finally there is PyQt. If you will be following GPL, you won't need to buy any licenses. But if that is not the case, you will need

to invest some money in PyQt and QT before you start developing your application.

GUI Designer Tools

As your applications grow, you will notice a lot of lines of code used just to create window components and by that time you will probably want to avoid unnecessary clutter in your code. That is where these tools should be used. GUI Designer tools also allows you to focus on the core development of your applications for the reason that GUI development will be a breeze.

Tools available for each toolkit will be rapidly described below. But you should familiarize yourself with them, or at least with the ones used by your chosen toolkit.

PyGtk tools

- Glade¹⁴ (First release: April 18, 1998 ; Latest release: December 18, 2007)
- Gazpacho¹⁵ (First release: June 30, 2004; Latest release: July 29, 2007)

Glade and Gazpacho are very similar from user perspective. They both requires you to understand how GTK containers work in order to construct interfaces. Using Glade there is a possibility of using a widget called "Fixed" that lets you drag and drop components into Toplevel widgets without caring about containers. Using this "technique" leads to a very problematic application, setting specific size and position for all widgets is a good recipe for headache. It is very likely your interface will not appear as correct in someone's else computer as it appeared on yours. For these reasons, you should learn how to use containers, it is not hard and your programs will achieve better results.

These tools generates .glade files that are then used by your application. Gazpacho also supports saving in gtkbuilder and gazpacho formats. GtkBuilder is said to take over glade format, making it the new format to use (it has been added in Gtk+ 2.12 and is available for PyGtk 2.12).

There are some reasons to pick Gazpacho over Glade, like the presence of a set of Kiwi¹⁶ widgets and the possibility to use GtkUIManager¹⁷.

And the reasons to pick Glade over Gazpacho are its development time and amount of users.

PyQt tool

- Qt Designer¹⁸ (First release: September 07, 2000 ; Latest release: October 03, 2007)

Qt Designer 4 is a very easy GUI designer to use, and very powerful as well. It is a tool that makes most people happy, you can just start dragging and dropping widgets and leave it like that or ask it to layout the widgets for you, or, you may choose to create Vertical/Horizontal/Grid Layout and drop widgets there.

Rarely you will need to change the tab order, but if you want to do it for the right reason, it provides a very attractive visual editor for that. Other nice visual features are edition of Signal/Slots and Buddies. The former provides a visual representation of the signal and slot connections in the form that can help other developers understand how the final component will behave. The later, associates a widget to its buddy. This association allows you to connect a QLabel to a QLineEdit (its buddy) and then focus the buddy widget by pressing QLabel's shortcut key combination.

This tool generates an .ui file, that you can choose to use through PyQt4.uic module or use the pyuic tool that will convert the .ui file to a .py file. It may also generate a .qrc file if you are using external resources, like images. To use the resources file, you need to use pyrcc tool so it converts to a python module with the external resources embedded into it.

14 Glade site: <http://glade.gnome.org/>

15 Gazpacho site: <http://gaspacho.sicem.biz/>

16 Kiwi site: <http://www.async.com.br/projects/kiwi/>

17 GtkUIManager reference: <http://www.pygtk.org/pygtk2reference/class-gtkuimanager.html>

18 Qt Designer site: <http://trolltech.com/products/qt/features/designer>

Tkinter tool

- GUI Builder¹⁹ (Released into open source in November, 2006)

GUI Builder is a next-generation SpecTcl alternative, as its page says, but I didn't see much difference except that GUI Builder menu creation is more straightforward to use than SpecTcl's one.

It is pretty simple and somewhat intuitive, and enforces the use of Grid geometry manager which is good because it makes you write applications with a consistent layout, that is, you will not be hand positioning the widgets.

This tool generates two files, a `_ui.py` file and other `.py` file based on the name you gave when you saved the your `.ui` file.

wxPython tools

- wxGlade²⁰ (First release: July 31, 2002 ; Latest release: February 02, 2008)
- XRCed²¹ (First release: ~ August 31, 2001 ; Latest release: March 10, 2007)

Before moving on let me say that there are more GUI builders available for wxPython, like: Boa Constructor²² (IDE and GUI Builder), VisualWx²³ (Windows only), wxDesigner²⁴ (Commercial). But I have chosen to talk only about wxGlade and XRCed because they are under active development, both works cross-platform, and they are open source projects. Now we may continue.

wxPython uses Sizers, and for the same reasons you should understand containers if you are using PyGtk, you should invest some time understanding sizers. Sizers are a bit harder and you will probably need to read some tutorial in order to use them perfectly. As suggestion, read this page: <http://wiki.wxpython.org/UsingSizers>.

wxGlade looks like Glade-2, it is not as pretty as Glade-3 (but works), except that it is able to generate XML-based resource system (or just XRC) files. Note that it is also able to generate direct Python code, but the advantage of opting for XRC format is that they can be used by XRCed as well.

XRCed is actually a resource editor that supports creating and editing files in XRC format, but I am including it in this section anyway. It will be your preferred tool when you get accustomed with wxPython, it is the fastest tool to develop an UI if you know enough about wx. XRCed doesn't involve drag & drop of widgets into a beautiful window, on the other hand you will drag & drop components into a XML tree, making it a bad choice if you are not comfortable on wxPython.

If you prefer to separate the UI layout from the code, be sure to use a XRC file. Working with this file is like working with glade files, but this is wxPython so it has several differences in use.

Both wxGlade and XRCed misses a lot of widgets included in wxPython, a consequence of this toolkit having so many widgets.

Summary

If you know your tool, and if the name of that tool is wxPython then XRCed is waiting for you. If you are considering learning some GUI toolkit and it is important that it has a powerful and featureful GUI Designer application, Qt Designer will be your favorite tool. Next to this there is Gazpacho, followed by Glade and wxGlade. Gazpacho takes the lead for supporting action based menus and toolbars (Qt Designer supports this too). Lastly, if you just want something simple and good looking is not important at all, GUI Builder is there for you.

Built-in widgets

Unlike other sections this one does not contains subsections dedicated to each toolkit, all

19 GUI Builder site: <http://spectcl.sourceforge.net/>

20 wxGlade site: <http://wxglade.sourceforge.net/>

21 XRCed site: <http://xrced.sourceforge.net/>

22 Boa Constructor site: <http://boa-constructor.sourceforge.net/>

23 VisualWx site: <http://visualwx.altervista.org/>

24 wxDesigner site: <http://www.roebling.de/>

the discussion will take place here.

All toolkits discussed provides a set of basic widgets that can construct any GUI application. But as you start doing more advanced user interfaces the number of used widgets will tend to increase and it is very likely that you will see yourself creating new widgets, or at least using some external libraries, every time you have a different application to build, if you are using Tkinter. That is because this toolkit has a very limited set of widgets and the chances of it growing are rare (Tk 8.5 has new widgets, meaning they will eventually be available in Tkinter) because it is part of Python and that means it is stable, which is not bad, but it stays the same for a long time. This is also one of the reasons why it is unlikely that some other GUI toolkit will take Tkinter's place in Python, because they are changing too often and possibly causing some form of instability.

wxPython has the largest set of widgets between the four toolkits. This implies that the chances of needing to construct a new widget are lower than if you were using PyGTK, PyQt or Tkinter. Of course if you need a very specific widget, no toolkit will provide it, so, you can either check if someone has already done it, or do it yourself. wxPython can save a great amount of time depending on the application you are planning to do, one of the reasons being the addition of wx.AUI library (AUI stands for Advanced User Interface). This library instantly adds a huge amount of flexibility to your application, like: floating/docking frames, customizable look and feel, perspective saving and loading, optional transparent window effects while dragging and docking, and others.

PyGTK and PyQt provides a similar amount of widgets, but PyQt leads between the two because QTextEdit supports HTML rendering, it also supports OpenGL in PyQt applications through QtOpenGL (requiring PyOpenGL) and allows docking widgets, for example.

Documentation Available

This last section points out some documents that you will be visiting while developing a GUI application. Note that there is much more "Documentation Available" than described here, this was just an attempt to collect most of the interesting documents existing relative to the toolkits discussed here.

Nevertheless, I hope these pointers will answer most of your question.

PyGtk

Document	Audience
PyGtk 2.0 Tutorial ²⁵	Anyone starting in PyGtk
PyGTK FAQ ²⁶	If you use PyGtk, but don't know everything about it yet, this FAQ will solve most of your questions that are likely to arise
PyGtk Wiki ²⁷	Both beginners and more experienced users will find this useful. Its main purpose is to collect links for several other documentations.
<ul style="list-style-type: none"> • Others <ul style="list-style-type: none"> • PyGTK 2.0 Reference Manual²⁸ • Articles and Tutorials about PyGTK²⁹ • Beginning Python, Chapter 13³⁰ 	

PyQt

Document	Audience
Introduction to PyQt4 ³¹	Anyone starting in PyQt

²⁵ PyGtk 2 Tutorial: <http://www.pygtk.org/pygtk2tutorial/index.html>

²⁶ PyGtk FAQ: <http://faq.pygtk.org/index.py?req=index>

²⁷ PyGtk Wiki: <http://live.gnome.org/PyGTK>

²⁸ PyGtk 2 Reference Manual: <http://pygtk.org/docs/pygtk/index.html>

²⁹ Articles and Tutorials about PyGtk: <http://pygtk.org/articles.html>

³⁰ Beginning Python book: <http://www.amazon.com/Beginning-Python-Programmer-Peter-Norton/dp/0764596543>

³¹ Introduction to PyQt4 site: <http://www.rkblog.rk.edu.pl/w/p/introduction-pyqt4/>

- Rapid GUI Programming with Python and QT³² PyQt 4 book, one of the best options available for learning how to program with this toolkit
- The PyQT and PyKDE community Wiki³³ Good link collections, anyone interested in PyQt will find this useful
- Qt Quarterly³⁴ For those already developing in Qt/PyQt interested in learning more about this toolkit.
- Others
 - PyQT v4 - Python Bindings for Qt v4³⁵
 - PyQt's Classes³⁶

Tkinter

Document	Audience
An Introduction to Tkinter ³⁷	Anyone starting in Tkinter
Python and Tkinter Programming ³⁸	A good read, in form of a book, for those wanting to learn Tkinter
Tkinter Wiki ³⁹	Contains links for some tutorials, reference documentation, and others. Better suited for intermediate users
<ul style="list-style-type: none"> • Others <ul style="list-style-type: none"> • Tkinter reference: a GUI for Python⁴⁰ • Tkinter -- Python interface to Tcl/Tk⁴¹ 	

wxPython

Document	Audience
The wxPython tutorial ⁴²	Anyone starting in wxPython
wxPython in Action ⁴³	Very good book, just a bit dated but still serves as one of the best ways, if not the best, to learn wxPython
wxPyWiki ⁴⁴	I would say this is the best wiki among the others mentioned here. Beginners, intermediate and experienced users will find good and interesting informations here
<ul style="list-style-type: none"> • Others <ul style="list-style-type: none"> • wxPython API⁴⁵ 	

Summary

There is a decent amount of documentation for all the toolkits. But it is important to ask yourself if you are willing to learn. It does not matter how many books, tutorials, articles, etc.. may exist for a toolkit (or anything else) if you just do not take time to learn what you want/need. The excuse of documentation not being good enough, in this case, is hardly a reason for preventing you from learning any of these toolkits.

32 Rapid GUI Programming with Python and QT book: <http://www.qtrac.eu/pyqtbook.html>

33 The PyQT and PyKDE community Wiki: <http://www.diotavelli.net/PyQtWiki>

34 Qt Quarterly site: <http://doc.trolltech.com/qg/>

35 PyQT v4 - Python Bindings for Qt v4 reference: <http://www.riverbankcomputing.com/Docs/PyQt4/pyqt4ref.html>

36 PyQt's Classes reference: <http://www.riverbankcomputing.com/Docs/PyQt4/html/classes.html>

37 PyQT's Classes: <http://www.pythonware.com/library/tkinter/introduction/index.htm>

38 Python and Tkinter Programming book: <http://www.manning.com/grayson/>

39 Tkinter Wiki: <http://tkinter.unpythonic.net/wiki/>

40 Tkinter Reference site: <http://infohost.nmt.edu/tcc/help/pubs/tkinter/index.html>

41 Tkinter Python module documentation: <http://docs.python.org/lib/module-Tkinter.html>

42 wxPython Tutorial site: <http://www.zetcode.com/wxpython/>

43 wxPython in Action book: <http://www.manning.com/rappin/>

44 wxPython Wiki: <http://wiki.wxpython.org/>

45 wxPython API site: <http://www.wxpython.org/docs/api/>

Doctest and unittest... now they'll be merrily together

Olemis Lang (olemis@gmail.com)

Testing is a very important discipline to ensure and validate software quality. Python includes two standard modules to perform functional testing. Prior to version 2.4 both tools were unrelated, leading to scattered testing code. From this version on a `unittest` API is provided by `doctest`. The present work aims to propose some enhancements to this API in order to achieve a better `unittest-doctest` integration. Although there are more complex testing tools (e.g. `nose`) which allow performing both kinds of tests, they are outside the scope of this article. Firstly, they are not standard modules. Besides, the intent is to load and execute doctests just like if we were using `unittest`.

Even though many types of tests exist these days⁴⁶, functional testing is very important. Firstly, it validates whether software behavior matches the business rules documented in the software requirements. Besides, for continuously evolving systems and iterative development processes it is also crucial to perform regression testing. Thereby introduced defects are handled as soon as possible and the defect does not propagate to future versions. These tests are also the main building block for test-driven development and extreme programming techniques. A scripting language like Python ought to be aware of this since it is often used to write test scripts (e.g. for Java⁴⁷ and .NET⁴⁸). Because of this, the paper also covers the standard modules available nowadays in Python for functional testing.

Outline

Before explaining the whole new solution to integrate both major testing frameworks for Python, it will be helpful to talk about them separately. This can be also useful for pythoneers wishing to get a fuller understanding of the options available these days. Nevertheless, the explanation will cover only the features needed to understand the work done. Afterward they are compared so as to illustrate the need and the idea leading to their integration in version 2.4. Its usage will also be portrayed, thereby clarifying the strongest as well as the weakest points inherent to this new feature the way we know it today. These facts help to establish a motivation for the proposition presented thereafter. Firstly the novel implementation is discussed in detail. This can be useful for developers, and all those aiming to understand how an object oriented API can merge both these frameworks. Next some useful use cases are explained. They are helpful for testers because therein they will find guidelines to face some testing scenarios. All the way through the emphasis made on object orientation as well as other distinctive features are explained, and compared with respect to the current standard.

Functional tests with doctest

The standard module `doctest` was included in Python 2.1. It is perhaps the most intuitive way available in this language to write functional tests. In order to gain a deeper insight about this framework it is important to know how tests are written. Once this is fully understood, the next step is to know the API which allows running tests. Finally, it is relevant to know the elements included in the framework in order to execute one test after another, and report the results. Let's briefly talk about these topics.

46 Several types of tests are described comprehensively in G. D. Everett, R. McLeod, Jr "Software testing : testing across the entire software development life cycle" (2007) John Wiley & Sons, ISBN 978-0-471-79371-7.

47 See M. Nadel "Use Jython to build JUnit test suites" available on-line at <http://www.ibm.com/developerworks/java/library/j-jythtest.html>.

48 A. Henderson "Integrating NUnit & IronPython..." link available at <http://ironpython-urls.blogspot.com/2006/10/integrating-nunit-and-ironpython.html>.

Specifying tests with `doctest` is very easy as illustrated in Figure 1. They reside in docstrings and therefore are expressed in textual form. This means that any element able to be explicitly documented can contain such tests. Usually the documentation for one such element includes only the tests needed to ensure that its implementation fulfills its expected behavior. The syntax involved to declare them resembles interactive sessions with the Python interpreter. In fact, coders can copy the characters outputted to the console during one such session and paste them into docstrings. That would be enough to specify the tests. But in real development it is often better, and more useful, to write the test before actually implementing the code behind it. However, the former characteristic is what causes using `doctest` to feel intuitive, because every Python programmer have interacted with the interpreter. This means that anyone knowing the language can write tests.

```

1.  def shuffle(seq):
2.      """
3.      >>> seq = range(10)
4.      >>> shuffle(seq)
5.      >>> seq.sort()
6.      >>> seq #doctest: +NORMALIZE_WHITESPACE
7.      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
8.      """
9.      # code omitted
10.
11. def sample():
12.     """
13.     >>> seq = range(10)
14.     >>> sample(seq, 20) # doctest: +ELLIPSIS
15.     Traceback (most recent call last):
16.         ...
17.     ValueError: sample larger than population
18.
19.     >>> [x in seq for x in sample(seq, 5)]
20.     ... #doctest: +NORMALIZE_WHITESPACE
21.     [True, True, True, True, True]
22.     """
23.     # code omitted
24.
25. def choice(seq):
26.     """
27.     >>> seq = range(10)
28.     >>> elem = choice(seq)
29.     >>> elem in seq
30.     True
31.     """
32.     # code omitted
33.     # Running doctests
34.     # from the command line
35.
36. if __name__ == '__main__':
37.     import doctest
38.     doctest.testmod()

```

Figure 1: Testing functions in standard `random` module with `doctest`.⁴⁹

After the tests are written, programmers must invoke one of the functions defined inside the `doctest` module in order to execute the tests. The one most frequently used for this purpose is `testmod`, but it is also possible to use either `run_docstring_examples` or `testfile`. So... what happens immediately after they get called? The first thing that takes place after such call is test hatching. This process is usually divided into two stages. When we test modules (`testmod`), `DocTestFinder` instances are created. Their role is to enumerate the functions (methods) and classes reachable from the module specified as a parameter, and also belonging to it. For each such object found, `DocTestFinder` instances extract its docstrings and trigger the second stage: parsing. If the tests reside in text files (`testfile`) or if a single object is to be tested (`run_docstring_examples`) then parsing is started right away since there is no need to use `DocTestFinder` objects. In the first case the contents of the text file will be parsed, whereas in the second one the string bound to the object's `__doc__` attribute will be considered instead.

Then you may ask... why do we need parsing in this case? Well, mainly because tests are intermingled with documentation, most of which is usually written in natural language. That's why the outcome of the previous step are the input to `DocTestParser` objects. Their duty is to separate the text representing interactive sessions from intervening text. Interactive sessions look like a collection of statements inputted by the programmer followed by the result outputted by the interpreter in consequence of its execution. The instances of `Example` class encapsulate such information (statement to execute + expected interpreter output). `DocTestParser` objects are thus also responsible for creating the `Example` instances

⁴⁹ Throughout the text the different frameworks are compared against versions of the test code provided in <http://www.python.org/doc/2.5.1/lib/minimal-example.html>.

representing the interactive sessions they found.

After all parsing has been performed, the resulting `Example` objects are put together into a container. Actually `DocTest` objects are liable for that, besides referencing a namespace (global namespace) which will be used in the next phase: test execution.

The processing class used to execute and verify the interactive examples in a `DocTest` is `DocTestRunner`. In order to do so, it uses an output handler (a callable object). Firstly, instances of the later class notify that testing is about to start. Next, they extract the `Example` objects included in the incoming `DocTest`. Each example is later on compiled. `DocTestRunner` requests the interpreter to execute them in the context of the namespace held by the `DocTest` instance under testing. The requesting object also gathers the interpreter output. At this time `OutputChecker` instances get into action.

`OutputChecker` instances compare the aforesaid output with the expected result (contained in `Example` objects). If both of them match, success is reported by the runner. If they do not match then failure notification occurs. A third case is still possible if an unexpected exception is raised during the execution. All these situations are reported by the aforementioned output handler, `sys.stdout.write` being the default one.

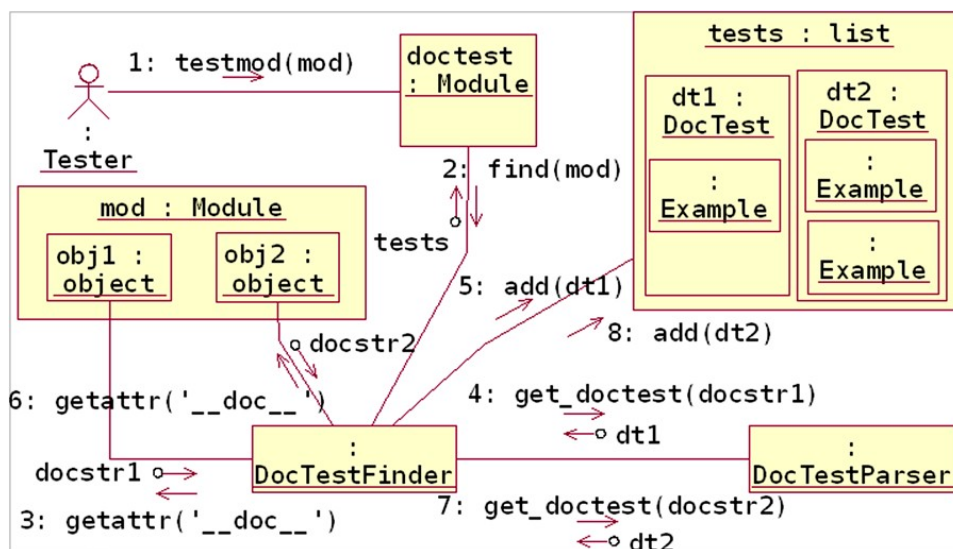


Figure 2: Retrieving `DocTest` instances from module docstrings.

The term unexpected exception was used before because by using `doctest` it is also possible to specify that the expected behavior is an exception to be raised. The documentation for function `sample` in Figure 1 shows how this is done. This kind of tests follows the same philosophy of being similar to interactive sessions. However, in case of exceptions the interpreter also outputs some fields which depend on the execution context and can vary from time to time. The use of ellipsis is compulsory under these conditions. They are wildcard characters which tell output checkers that a portion of the resulting string has to be ignored because either it is not important for the check to perform, or perhaps because this segment can vary from one time to another. Ellipsis ought to be used in together with “boolean” *option flags*. The comments in lines 6 and 14 in Figure 1 portray how to set them. They allow to customize how `OutputChecker`s verify the examples and the contents of the subsequent report. A `DocTestRunner` can use another output checker so there is space to write tests using a different syntax which supports user needs.

Assessing `doctest`

A direct consequence of the aforesaid layout for tests is that they can be considered as a behavioral description of the element which is to be tested. In this case, the same language used to code the feature is used to sketch its behavior, so there is no need for further

translations to get executable testing artifacts. The tests also tell us explicitly how to use the different functions, classes, methods and so on. It is also clear what should be expected in return. Since all this information is located in docstrings, the documentation of these objects is enhanced. Example usage is available, being possible to record it in the form of a simple tutorial combining source code written in Python, together with explanations written in natural language. This is practical because “sometimes a single line of code is better than one thousand words”.

Another benefit of `doctest` is its encapsulation. Coders need not to know what happens behind the scene so as to test the different examples. This contributes a lot to make `doctest` a user-friendly tool. On the other hand, its results are truthful since it relies on the interpreter to execute the examples. So whatever you see happening during the test is what you actually get once the statements in the docstrings are executed in a real scenario. This could be the case when other coders use the library code being checked to build their own systems.

However some enhancements to `doctest` are still possible. For example, `DocTest` class is meant to represent a *Composite* but is not implemented considering this pattern. This implies that it's not easy to put together different instances of this class, as well as instances of other classes representing other kinds of functional tests. This also means that it could be difficult or unnatural to mix in the examples found in different modules in order to test them altogether. In this case scattered test reports could be obtained while testing a populated package, for example. Furthermore:

“test setup has to be either copied or hidden away from the test, making the overall environment harder to understand.”⁵⁰

Another valuable characteristic is that different subclasses of those used by `doctest` in test execution can allow for customization of the whole process. However the API supplied for this purpose restrains this flexibility and, for example, only allows sending reports to the standard output. This means that test upshots for individual examples are intermingled. Therefore if they were of interest, `sys.stdout` needs to be redirected perhaps via `StringIO`. Besides further analyses are made difficult since test reports ought to be parsed.

PyUnit tests in action

The module `unittest` is included among the standard modules since version 2.1. It inherits a long tradition started by Kent Beck's *Smalltalk testing framework* (a.k.a. *SUnit*), and followed by others like *JUnit* (Java), *CppUnit* (C++), *JSUnit* (JavaScript), *HUnit* (Haskell), *mUnit* (MATLAB), *utMySQL* (MySQL), *NUnit*, *csUnit* (.NET programming languages), and many other implementations for at least 51 programming languages of different nature. Python could not be the exception.

The notion of *test cases* is central to the testing process defined by *xUnit* frameworks. In a simplistic way they could be viewed as the most atomic unit of testing. Broadly speaking, they define the features under test and how to carry out the assessment. The most naive way to implement a test is to extend the `TestCase` class and redefine its `runTest` method. Every statement to check has to be asserted using the methods available in the prior base class. If no such assertion fails then the test succeeds; otherwise, it is reported as a failure. Failures represent anticipated problems. Once again a third case is possible when an unanticipated exception (one not caught by `assertRaises` method) is raised while conducting the test. These conditions are reported as errors, and are more catastrophic since they are unchecked bugs.

Even a simple application can demand several functional tests. That's the reason for having the `TestSuite` class. Its instances group the test cases being part of a testing scenario. These suites are implemented according to the Composite pattern. Therefore they can contain other suites, as well as separate test cases. Infinite nesting levels are possible in theory. A test suite can be built explicitly by hand. Nonetheless, in real life there will be several

⁵⁰ Wikipedia page for `doctest`, available on-line at <http://en.wikipedia.org/wiki/doctest>.

classes whose methods define tests over the target application. Suites construction would be very tedious in these situations. Therefore `unittest` includes the `TestLoader` class. Its objects automatically gather the tests defined within a module or class. In modules containing many `TestCase` descendants, the tests found within each such class are returned wrapped in a `TestSuite`. Well, what happens once loaders receive classes as input?

Table 1: *TestCase* methods to check for and report failures.

Method Name	Description
<code>fail</code>	Signals a test failure unconditionally.
<code>assert_</code>	Signals a test failure if an expected condition is not met. A message describing the failure can be supplied.
<code>assertEqual</code>	Test that two values are equal. If not, the test will fail with a given explanation.
<code>assertNotEqual</code>	Test that two values are not equal. If they do compare equal, the test will fail with a given explanation.
<code>assertAlmostEqual</code>	Used to test for equality of two instances of inexact types like float. An explanation may be given.
<code>assertNotAlmostEqual</code>	Used to test if two instances of inexact types differ so much as to be considered different. An explanation may be given.
<code>assertRaises</code>	Test that an exception of a given type (or any of a group of exceptions) is raised when a callable is invoked with known positional and keyword arguments.
<code>failIf</code>	Signals a test failure if an abnormal condition is met. A message describing the failure can be supplied.

The fact is that `unittest` supports test fixtures. In practice, as software evolves testers find groups of similar test cases. Often they require the same initialization and cleanup. If this is the case a single `TestCase` inheritor can contain multiple test methods. When a test loader is about to build a suite out of a `TestCase` descendant, it examines whether it contains any method whose name starts with the “test” prefix like shown in Figure 3 lines 10, 18, and 22. If this is not the case, only `runTest` is executed as explained before. Otherwise, for each method the loader creates one instance of the class in order to execute the former. The method name is bound to the `_testMethodName` attribute of this instance. All the objects thus created are collected into a test suite.

Having nothing but a test suite is not enough. Its test cases should be executed. Test runners are used for this purpose. They decide how to test the suite and what to do with test results. `PyUnit` incorporates the class `TextTestRunner` in order to output the results in textual form to a file-like object (`sys.stderr` by default).

The first things runners do is to create an instance of the class `TestResult`. Later, `TestCase` instances contained into the target test suite are processed one by one. Immediately before each test is actually performed, the test case's `setUp` method gets invoked. Since test cases derived from the same fixture share this method, the common initialization steps should be coded therein. Afterward, the test method bound to the test case instance gets called. The framework monitors whether a failure, an error or a successful test occurs. In any case, the result is stowed into the `TestResult` object created for this run. Immediately after the test method has been called and the result recorded, the `TestCase.tearDown` method gets executed. Since instances of the same fixture also share this method, it can be used to release the resources allocated from within `setUp`.

```

1.  from random import shuffle,
2.      choice, sample
3.  import unittest
4.
5.  class TestSequenceFunctions(
6.      unittest.TestCase):
7.
8.      def setUp(self):
9.          self.seq = range(10)
10.
11.     def testshuffle(self):
12.         # make sure the shuffled
13.         # sequence does not lose
14.         # any elements
15.         shuffle(self.seq)
16.         self.seq.sort()
17.         self.assertEqual(self.seq,
18.             range(10))
19.
20.     def testchoice(self):
21.         element = choice(self.seq)
22.         self.assert_(element in self.seq)
23.
24.     def testsample(self):
25.         self.assertRaises(ValueError,
26.             sample, self.seq, 20)
27.         for elem in sample(self.seq, 5):
28.             self.assert_(elem in self.seq)
29.
30.     if __name__ == '__main__':
31.         unittest.main() # Command-line test
32.
33.     def explicit_test_run():
34.         # Finer level of control to run tests
35.         loader = unittest.TestLoader()
36.         suite = loader.loadTestsFromModule(
37.             sys.modules[__name__])
38.         unittest.TextTestRunner(
39.             verbosity=2).run(suite)

```

Figure 3: Writing `unittest` test cases for the `random` module.

Assessing `unittest`.

Testing frameworks like `unittest` are quite popular since long time ago. One of the reasons behind this success is perhaps that it shows a mature object design supported by a high pattern density. Only the `TestCase` class is involved in at least four design patterns. This implies that this tool is easier to use, but harder to change. For instance, test suites allow different kinds of test cases to be tested altogether. Test cases of variate nature can dwell inside a single suite. A larger group can be formed after appending this same suite to another one perhaps containing other arbitrary suites. Regardless of their possibly different nature, they all are attached and tested the same way. Consequently it is possible to say that `unittest` encourages easy assembly and smooth integration.

Another key feature is the code reuse made possible thanks to an object oriented API. This is possible mainly to a deep separation of concerns among test retrieval, test procedure, test execution, and finally result gathering for later analysis. Firstly, this means that custom test loaders can load test cases from diverse sources, can be represented in different formats, or even follow different conventions. Separately, the system under test can be assessed in many different ways. To do so it is only necessary to add new test methods to `TestCase` descendants. Besides the testing process may be performed in dissimilar manners without interfering with the test code that actually checks the target system. To illustrate this point let's consider the example of the peer library `JUnit`. It is possible to run the same test suites reporting the outcomes in text mode via `junit.textui.TestRunner`, or graphical mode via one of `junit.awtui.TestRunner` or `junit.swingui.TestRunner`. Examples of special runners are those used by IDEs (e.g. Eclipse) to represent a test run in their interface.

It is also possible to customize the way test outcomes are stored by using personalized `TestResult` subclasses. Therefore besides volatile storage, either RDBMS, ORM, proprietary files, or anything else can be used for this purposes. A test repository being part of the project measurements could hence be deployed. The data gathered this way might give support to test analyses which can illustrate continuous displays and evolution of project status, the capacity to progress towards goals, and the efficacy of the development process. Enterprises interested in moving their CMM⁵¹ level up, can take advantage by automating key process areas from levels 2 (*Repeatable*) to 5 (*Optimizing*).

Nonetheless people usually spends far more time reading test code than actually writing it. That's why the challenge is writing readable tests⁵². In this respect `unittest` code can be hard

⁵¹ Capability Maturity Model

⁵² J. Fulton, T. Peters "Literate Testing: Automated Testing with doctest" (2004), *PyCon 2004*.

to understand, demanding from the reader previous knowledge about the framework. Test code is usually separated from source code, which can possibly difficult this task even more. This testing toolkit by itself makes no contribution to software documentation either. Another controversial topic is the way exceptions are asserted. Maybe this is the most notorious case illustrating that test code does not look like client code⁵². In this respect `doctest` seems to be more natural.

The gathering

Before Python 2.4, `doctest` included the `Tester` class. It provided simple means to combine the doctests retrieved from different modules (e.g. a package), and test them thoroughly. From version 2.4 and on, this class has been deprecated. Now it is clear that both frameworks complement each other. The weaknesses of the former turn into strengths of the later, as explained before. At the moment, the `unittest` API provided by `doctest` makes possible to create test suites from modules and text files containing doctests. The later can be combined with tests from multiple sources. Consequently `unittest` runners can run them altogether at once.

```

33. # Override the statements in Figure 1 from line 33 on.
34. import doctest, unittest
35.
36. if __name__ == '__main__':
37.     unittest.main(defaultTest='suite')
38.
39. def suite():
40.     return doctest.DocTestSuite(sys.modules[__name__])
41.
42. def run_tests():
43.     # Finer level of control to run tests
44.     unittest.TextTestRunner(verbosity=2).run(suite())

```

Figure 4: Verifying doctests using the standard `unittest` API.

Suites are created from doctests out of modules via the `DocFileSuite` function. It accepts the `setUp` and `tearDown` optional parameters. Both should be bound to a function object. In this case a `DocTest` object is built as formerly explained with the help of `DocTestFinder`. Next it is wrapped by an instance of `DocTestCase` (a `TestCase` descendant) and a regular suite containing it is returned. During this process the aforementioned parameters are bound to attributes of the new `DocTestCase` instance. Besides the `unittest`-oriented API consents testers to specify custom instances of `DocTestParser` to extract `doctest Examples` out of docstrings. Personalized `OutputCheckers` are welcomed as well in order to match differently the interpreter output against the expected result. They both are supplied in the form of keyword arguments to `DocFileSuite`. This feature is definitely an enhancement over the preceding API.

Running the new test case is a process which reuses the prior `DocTestRunner` class. However, before the execution the function supplied in the `setUp` argument is invoked with the wrapped `DocTest` instance as its sole parameter. After the test is performed the same happens with the priorly mentioned `tearDown` argument, thereby imitating test fixtures. The outcome given by the `doctest` runner is stored into a character string through `StringIO` objects. If success was not accomplished, a failure is reported to the `unittest` runner carrying out the global test. The `doctest` details are provided as the descriptive message. That's why all the issues highlighted for `DocTestRunner` reports are also valid in this context. Eventually both `unittest` and `doctest` formats will be interleaved. This could be annoying. Long reports might be confusing especially in view of the fact that multiple summaries are made. Moreover the number of individual `doctests` that failed or behaved erroneously are not considered for the final statistics reported by the `unittest` runner. A single failure abbreviates them all.

The function `DocFileSuite` rescues us when doctests lie within text files. The whole procedure is very similar to the one already explained. A characteristic common to both these functions is that they have extensive signatures. The API itself lacks on object orientation, and is not compliant with `unittest` test loaders. The main difference is that `DocFileCase` objects are used instead of `DocTestCase`. Nothing new happens in practice since the former only overrides

cosmetic features of the later.

One object oriented API to join them

After analyzing the former arguments the focus turned out to evolve the API available to retrieve test cases from doctests. This new interfaces aims to allow programmers to write doctests the same way they have done so far, but handle the tests like `unittest` users do. Hence it mostly reuses both frameworks. It also overrides the classic `doctest` interface and the one given for Python 2.4, but reuses important implementation details. So they can be considered as a useful facilitator for this work.

Table 2: Mapping from `unittest` integration API items to `doctest`'s.

doctest	Standard unittest API	OO unittest API
Example		<code>DocTestCase</code>
<code>DocTest</code>	<code>DocTestCase</code>	<code>DocTestSuite</code>
<code>DocTestFinder</code>	<code>DocTestSuite</code> function	<code>DocTestLoader</code>
<code>DocTestRunner</code>	<code>DocTestRunner</code>	<code>_Doc2UnitTestRunner</code>
<code>testmod</code> function	<code>unittest.TextTestRunner</code>	<code>unittest.TextTestRunner</code>

The main variation introduced with respect to the 2.4 version is how legacy `doctest` classes map to `unittest` concepts. Since one goal was to gather separately the information resulting from testing individual `Example` instances (i.e. single statements), the unit of testing could be no longer bound to `DocTest` object like before. Rather than this, the target for new test cases are `Example` objects. Let's explain the whole in more detail.

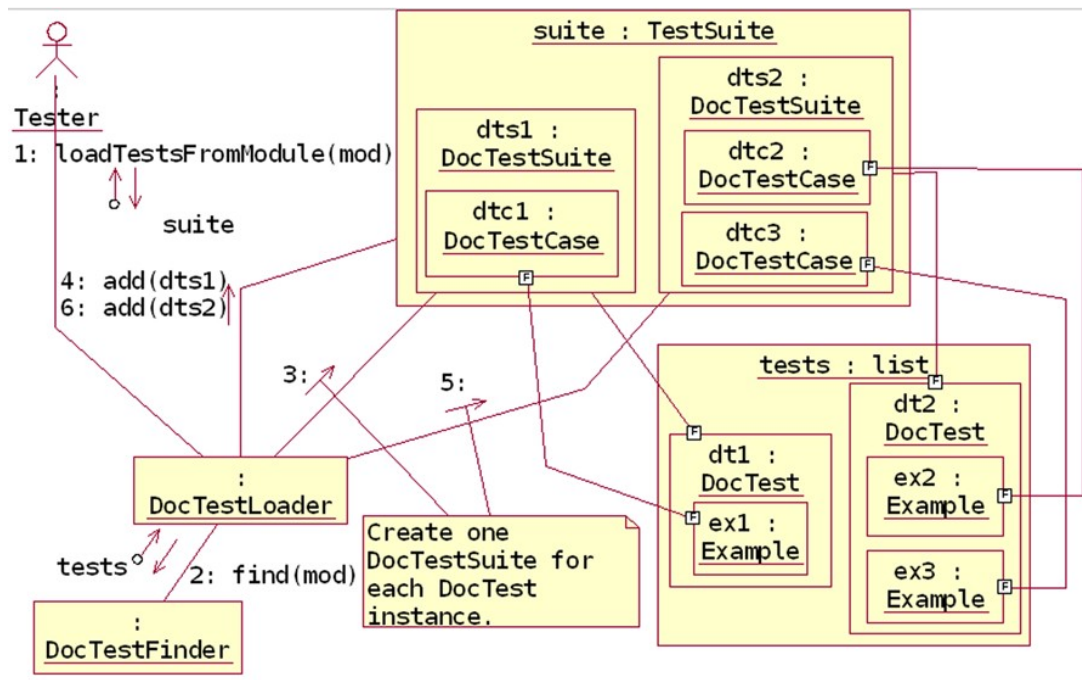


Figure 5: Retrieving test cases from doctests with loaders.

A radical new feature is that test cases are no longer loaded through functions. They have been replaced with the class `DocTestLoader`. It is a novel test loader introduced to achieve better object orientation and conform to `unittest` rules. The loading process starts when legacy `DocTestFinder` functionality is reused. And here we have the first feature contributing to flexibility. The type of finder to use for this purposes can be specified when loaders are

created. The subsequent step is to wrap the resulting `DocTest` instances with specialized suites and group the later into the `TestSuite` object returned by the loading process. The aforementioned specialized suites are represented by the also new `DocTestSuite` class and its descendants. And yes, since the suite type used in practice is bound to loaders' `docTestSuiteClass` attribute, subtypes of `DocTestLoader` can override this value and instantiate some other suites. Testers can thereby introduce their own features to meet particular needs. This solution is inspired in the usage given to the `suiteClass` attribute in `TestLoader` class.

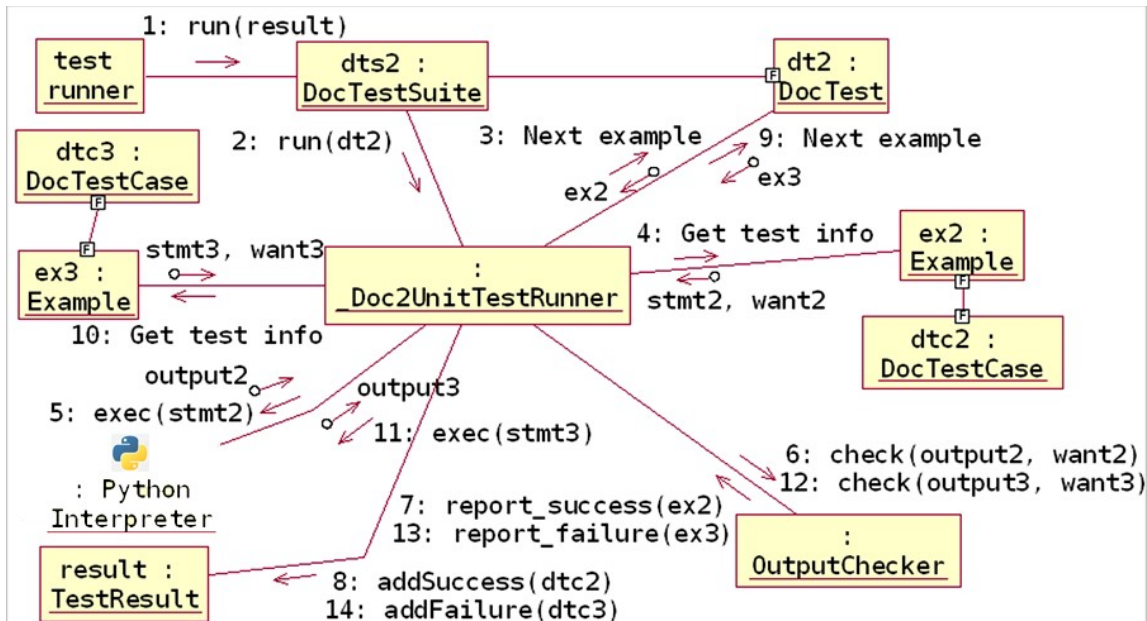


Figure 6: The custom-made `doctest` run yields more precise reports.

`DocTestSuite` class has a determining role in the integration and acts like `DocTest` peer. It maps a `unittest` run carried out by an enclosing `unittest` runner to the run needed to verify `doctest` examples. Thus It matches the *Adapter* pattern, being `DocTestRunner` the *adaptee*. In the background a tailored `DocTestRunner` descendant (`_Doc2UnitTestRunner`) executes and verifies interactive examples. Nonetheless before doing so, the suite instantiates a test case for each `Example` contained within the `DocTest` wrapped by itself. A bidirectional association is established among both examples and test cases. The novel `DocTestCase` class is used by default. Once again it is possible to supersede this decision by overriding the type object (i.e. `DocTestCase` subtype) bound to the `docTestCaseClass` attribute of the suite.

Next, when the interactive examples are executed and verified, the *Adapter* pattern is also employed to hook `_Doc2UnitTestRunner` report methods. This allows to record the outcomes in `TestResult` objects. Testers can refine the default runner by overriding the type object (i.e. a subtype of `_Doc2UnitTestRunner`) bound to the `docRunnerClass` attribute of the suite (preferably by sub-classing `DocTestSuite`).

Use cases

So far the focus has been placed in explaining how the different classes collaborate to achieve the desired goal. Let's dedicate some time to illustrate how to use the novel API.

Basic usage

There are some changes with respect to running tests by using classic `doctest`, Python 2.4 `unittest` API, and the current solution. Assuming the same functions from line 1 to 32 in Figure 1 have been declared, Figure 7 shows how doctests are run with the object-oriented API. Observe there is almost no difference with respect to `unittest` usage.


```

33. # Override the statements in Figure 1 from line 33 on.
34. import doctest
35.
36. if __name__ == '__main__':
37.     doctest.main() # Command-line test
38.
39. def run_tests():
40.     # Finer level of control to run tests
41.     suite = doctest.DocTestLoader().loadTestsFromModule(sys.modules[__name__])
42.     unittest.TextTestRunner(verbosity=2).run(suite)

```

Figure 7: Using the object oriented API to run tests.

Using optional doctest features

Anybody can wonder “How can I control doctest's behavior via option flags?”. The initializer in `DocTestLoader` accepts the extra keyword arguments shown in Figure 8. These arguments allow the use of tailored output checkers, as well as the use of legacy `doctest` options. All these parameters flow from `DocTestLoader` to `DocTestSuites`, where they are stored. At test runtime, the `_Doc2TestUnitRunner` object involved employs them to perform the `doctest` run. Besides the keyword arguments for option flags, it is also possible to supply in to the initializer a `dict` to be used as the globals when executing examples. Parameterized doctests are also possible by supplying in another `dict` to be merged into the globals used to execute examples.

```

1. import doctest, doctest
2. from __future__ import CO_FUTURE_WITH_STATEMENT
3.
4. loader = doctest.DocTestLoader(
5.     DocTestFinderSubClass(), # compatible with previous DocTestFinder subtypes
6.     {'glob1': 1, 'glob2': 2, 'glob3': 3}, # globals used when executing examples
7.     {'extra1': 1, 'extra2': 2}, # extra globals to execute examples
8.     optionflags = doctest.REPORT_UDIFF, # doctest options flags to use
9.     checker = MyOwnCheckerClass(), # override how examples are verified
10.    runopts = dict(
11.        compileflags = CO_FUTURE_WITH_STATEMENT, # options to compile examples
12.        clear_globs = True) # clear global namespace after testing
13. )

```

Figure 8: Specifying doctest optional features.

Combining test cases and doctests

“What if my module contains both doctests and test cases?” Thanks to the object oriented nature of the present API, a simple solution is at hand. Firstly, we need a loader whose purpose is to retrieve various types of tests and assemble them into a test suite. An implementation resembling the *Chain of Responsibility* pattern is shown in Figure 9c. An instance of this loader holding a legacy `TestLoader` and a `DocTestLoader` does what we need.

Since testers might frequently face this situation in practice, the `MultiTestLoader` class has been included into the API. It is important to notice that a single step is needed to set up the testing scenario. Besides this all happens like we are used to with `unittest` loaders. Hence this style encourages uniformity. The class `MultiTestLoader` is extremely reusable, even in other contexts.

```

1. def choice(seq):
2.     """
3.     >>> seq = range(10)
4.     >>> elem = choice(seq)
5.     >>> elem in seq
6.     True
7.     """
8.     # code omitted
9.
10. def shuffle(seq):
11.     # code omitted
12.
13. def sample():
14.     """
15.     >>> seq = range(10)
16.     >>> sample(seq, 20)
17.     ... # doctest: +ELLIPSIS
18.     Traceback (...):
19.         ...
20.     ValueError: ...
21.     >>> [x in seq for x in \
22.         ...     sample(seq, 5)]
23.     ... #doctest: +NORMALIZE_WHITESPACE
24.     [True, True, True, True, True]
25.     """
26.     # code omitted
27.
28. class TestSequenceFunctions(
29.     unittest.TestCase):
30.
31.     def setUp(self):
32.         self.seq = range(10)
33.
34.     def testshuffle(self):
35.         shuffle(self.seq)
36.         self.seq.sort()
37.         self.assertEqual(
38.             self.seq, range(10))
39.
40.     def testsample(self):
41.         self.assertEqual([],
42.             sample(self.seq, 0))

```

a) Writing the tests

```

43. import unittest
44. from doctest import DocTestLoader,
45.     main, MultiTestLoader
46.
47. loaders = [unittest.defaultTestLoader,
48.            DocTestLoader()]
49.
50. if __name__ == '__main__':
51.     main(testloader=
52.         MultiTestLoader(loaders))
53.
54. def run_tests():
55.     # Finer level of control to run tests
56.     loader = MultiTestLoader(loaders)
57.     suite = loader.loadTestsFromModule(
58.         sys.modules[__name__])
59.     unittest.TextTestRunner(
60.         verbosity=2).run(suite)

```

b) Running doctests and test cases altogether

```

1. class MultiTestLoader(
2.     unittest.TestLoader):
3.     def __init__(self, loaders= []):
4.         self.loaders= loaders
5.
6.     def loadTestsFromModule(self,
7.         module):
8.         return self.suiteClass(
9.             [loader.loadTestsFromModule(
10.                 module) for loader in
11.                 self.loaders])
12.     # further code omitted

```

c) A loader to retrieve different kinds of tests

Figure 9: Asserting doctests and test cases found in a single module.

Defining fixtures

The concept of fixtures pioneered by *xUnit* frameworks can be used to hide away test setup code from docstrings, thereby obtaining more concise documentation. Nevertheless at the same time readability might be jeopardized at some extent. Since test cases work at the example level in this solution, the legacy `setUp` and `tearDown` methods are executed respectively before and after the interpreter executes each example.

The current `doctest unittest` API behaves differently. It calls fixture methods once before and after asserting all the examples. However the same behavior can be obtained using test cases implemented with test patterns like *Shared Fixture* and *Chained Tests*⁵³.

⁵³ Test patterns are presented in G. Meszaros "XUnit test patterns : refactoring test code" (2007), Addison-Wesley, ISBN 0-13-149505-4.

<pre> 13. """ 14. range(10) is assigned to seq before 15. executing each statement. 16. 17. >>> shuffle(seq); seq.sort(); seq 18. ... #doctest: +NORMALIZE_WHITESPACE 19. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 20. 21. >>> sample(seq, 20) # doctest: +ELLIPSIS 22. Traceback (most recent call last): 23. ... 24. ValueError: sample larger than population 25. 26. >>> [x in seq for x in sample(seq, 5)] 27. ... #doctest: +NORMALIZE_WHITESPACE 28. [True, True, True, True, True] 29. 30. >>> choice(seq) in seq 31. True 32. """ </pre>	<pre> 33. from doctest import main, 34. DocTestCase, DocTestSuite, 35. DocTestLoader 36. 37. class RandomTestLoader(38. DocTestLoader): 39. class doctestSuiteClass(40. DocTestSuite): 41. class docTestCaseClass(42. DocTestCase): 43. def setUp(self): 44. exec 'seq = range(20)' in \ 45. self._dt.globs 46. 47. # self._dt returns the DocTest 48. # object containing the Example 49. 50. if __name__ == '__main__': 51. main(loader=RandomTestLoader()) </pre>
a) Concise docstrings	b) Compact fixture code

Figure 10: Per example preparation and cleanup actions.

Conclusions

In 2007 Python has been considered by TIOBE as the language of the year⁵⁴. According to the same source in January 2008 it has scaled up to the sixth place among the most popular programming languages. It is also considered as the glue language by excellence, and the community behind it is undoubtedly healthy. Besides being a recognition to the work made by many since years ago, all these arguments are so moving for new developers captured by its beauty. Nonetheless there is still a place for enhancements.

The standard module `doctest` is one example of such a beauty, whereas `unittest` is a typical case of strength, flexibility and stability. The later is full of design patterns, and sustains a large number of testing patterns. The former reflects the strong support provided for meta-programming in Python. It is related to many well-known *xUnit* patterns, especially test automation patterns (e.g. *Data-Driven Test*, *Recorded Test*, *Scripted Test*), and result verification patterns (e.g. *Behavior Verification*, *State Verification*, *Delta Assertion*)⁵³.

The main ambition of the present work is to run the `doctest` machinery while performing `unittest` runs. This has been accomplished after interleaving a layer which reconciles their respective interfaces. Given the object oriented nature of the solution, it is not bizarre that core classes be subjects of the *Adapter* pattern. Notably, `_Doc2UnitTester` takes part twice. It is also the main gateway between `doctest` and `unittest` in our solution. This confirms the fact that pattern density gets higher around key classes. As already said, the novel API also allows the use of many *xUnit* testing patterns while testing doctests.

Perhaps the strongest arguments in favor of this solution are related to its contribution to automated test analysis. This discipline is very important because it is a powerful indicator of a project's progress towards its goals. The number of attempted test cases over time highlights how effective the testing activities perform. Otherwise it could be found that they do not behave accordingly to the test plan. Test analysis can be helpful to adjust schedules, assign tasks, track defects, prioritize goals, monitor the development process, discover root causes, and many other dissimilar activities. In all cases, the main input consists of test results. The process of obtaining detailed information via the current `unittest` API is more complex in view of the fact that a full `doctest` report is stored. First, we need to extract the report from a `TestResult` instance. Next, useful information is retrieved through parsing. In our case, the same can be done by directly inspecting `TestResult` objects. This could ease tasks

⁵⁴ News found at TIOBE's home page <http://www.tiobe.com/>.

contributing to CMM key process areas like *Software Project Tracking and Oversight*, *Software Quality Assurance (Repeatable level)*, *Peer reviews*, *Software Product Engineering*, *Organization Process Focus*, *Intergroup Coordination (Defined level)*, *Quantitative Process Management*, *Software Quality Management (Managed level)*, and finally *Defect Prevention (Optimizing level)*.

```
F..F.FF
=====
FAIL: testshuffle (__main__.TestSequenceFunctions)
-----
Traceback (most recent call last):
  File "<stdin>", line 8, in testElems
AssertionError: [0, 1, 3, 4, 5, 6, 7, 8, 9] != [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

=====
FAIL: __main__.choice (line 5)
-----
AssertionError: Failed example:
  elem in seq
Expected:
  True
Got:
  False

=====
FAIL: __main__.sample (line 4)
-----
AssertionError: Failed example:
  sample(seq, 20) # doctest: +ELLIPSIS
Expected:
  Traceback (most recent call last):
    ...
  ValueError: sample larger than population
Got:
  []

=====
FAIL: __main__.sample (line 9)
-----
AssertionError: Failed example:
  [x in seq for x in sample(seq, 5)]
  #doctest: +NORMALIZE_WHITESPACE
Expected:
  [True, True, True, True, True]
Got:
  []

-----
Ran 7 tests in 0.015s

FAILED (failures=4)
```

Figure 11: Test report obtained with the object-oriented [unittest](#) API.

The implementation also contains plenty of interesting ideas, even for cosmetic features. For example, Meyer's *principle of uniform access* was exercised while coding [DocTestCase](#). Built-in [property](#) objects assisted in hiding the complex computation to derive test method's names from the corresponding [Example](#)'s attributes.

Acknowledgments

The thorough reviews made by the editors were crucial to enhance the contents and catch some mistakes. Their dedication and timeliness were highly valuable. The emphasis made on readability while writing the code snippets, is mainly due to Medardo Rodríguez. Moreover, the anonymous reviewers offered concise and very helpful comments.

The Python interpreter as a framework for integrating scientific computing software-components

Michel F. Sanner

The Scripps Research Institute,
Department of Molecular Biology, TPC26
La Jolla, CA 92037

Abstract:

The focus of the Molecular Simulation Laboratory is to model molecular interactions. In particular, we are working on automated docking and molecular visualization. Building and simulating complex molecular systems requires the tight interoperation of a variety of software tools originating from various scientific disciplines and usually developed independently of each other. Over the last ten years we have evolved a strategy for addressing the formidable software engineering problem of integrating such heterogeneous software tools. The basic idea is that the Python interpreter serves as the integration framework and provides a powerful and flexible glue for rapidly prototyping applications from reusable software components (i.e. Python packages). We no longer think in terms of programs, but rather in terms of packages which can be loaded dynamically into the interpreter when needed, and instantly extend our framework (i.e. the Python interpreter) with new functionality. We have written more than 30 packages (>2500 classes) providing support for applications ranging from scientific visualization and visual programming to molecular simulations and virtual reality. Moreover, some of our components have been reused successfully by other laboratories for their own research. Applications created from our software components have been distributed to over 15000 users around the world. In this paper we describe our approach and its various applications, discuss the reasons that make this approach so successful, and present lessons learned and pitfalls to avoid in order to maximize the reusability and interoperability of software components.

Keywords: scripting languages, interpretive languages, interpreters, programming paradigms, code reusability, modularity, interoperability, component-based software development, software engineering.

Introduction

Molecular biology is evolving from the study of macromolecules in isolation towards complex environments, potentially as large as complete cells. The amount and heterogeneity of information that needs to be processed and integrated in order to understand and simulate such complexity requires a leap in the level of sophistication of our software tools. The next generation of bioinformatics programs will have to: (1) support the inter-operation of rapidly evolving and sometimes brittle, computational software developed in a variety of scientific fields and programming languages; (2) support the integration of data across biological experiments and scales; (3) adapt to rapidly evolving hardware environments; and (4) empower users by allowing them to

carry out operations unanticipated by the programmers such as trying new combinations of algorithms or new visualizations. These combined requirements of flexibility, portability, resilience to failure, programmability by the user, and responsiveness to changes in the computational models and the hardware pose a tremendously challenging software engineering problem.

So far, this challenge has mainly been address by the creation of software frameworks that provide Application Program Interfaces (APIs) for integrating new software components. These frameworks usually provide support for managing software components (e.g. plugin technology) and the overall workflow and communication between components. We have found that this approach has several drawbacks. First of all, it promotes extensible environments rather than code reuse. Users and developers often have their own software environment and would prefer to include a new method in their own environment rather than having to switch to a completely new framework in which many of their usual tools might not be available. The integration of a given computational method into one of these frameworks only benefits this particular framework and this integration work will have to be repeated for each new framework. More importantly, each framework wants to act as the central piece of the software environment, and be in charge of scheduling operations and the execution flow, thus making it difficult to combine functionality available in various frameworks. These frameworks also often impose restricting data types or data models, and grant limited control to components over execution flow and other components. Finally, in many of these frameworks link their extensions into the environment rather than dynamically loading them when needed, thus leading to bloat-ware.

Command languages and scripting capability are often available in complex software environments. On rare occasions, these languages are powerful enough to be used to implement new extensions. However, these languages are often highly biased toward a particular scientific domain and lack generality. Moreover, a framework's language is only usable in that particular application which limits the size of the user community, and in turn reduces the potential contributions for the user community. Some applications embed scripting languages such as TCL or Python which is a better approach than developing a custom language. However, the framework is still the central piece of the environment and dictates the way extensions are developed, thus making these extensions framework specific. Moreover, the functionality of the application is not re-usable in other applications, but merely scriptable through the interpreter.

Our Approach

Here we present a solution we have evolved and implemented over the past ten years and which has proven to be successful in addressing some of the shortcomings discussed above. Rather than developing our own interpretive language we used the Python programming language ¹, and instead of embedding Python into a framework or developing yet another framework, we decided to make the Python interpreter itself our framework. This idea of using an interpretive language as an integration framework could work with other interpretive languages as well. This Python-centric framework is extended by developing dynamically-loadable Python packages implementing specific functionality. This approach promotes code compartmentalization and forces developers to switch their mindset from writing programs to writing components that carry out

specific tasks. Applications are then generated by combining these components at a high level, leading to true component-based applications. This architecture makes swapping software components, comparing implementations and integrating new methods trivial. Such a high level of flexibility is particularly interesting in a research environment where the goals are constantly shifting as new discoveries are made, and where new computational methods need to be integrated in our simulation environments as they become available. While this difference in mindset might appear trivial, experience has shown that the development of software components in isolation of an application or a framework is greatly increasing their re-usability. When developing for a framework it is always tempting to take advantages of some of its features with the detrimental consequence of tying the extension to this particular framework. Using Python-centric approach, we have experienced an unprecedented level of reuse of software components in several applications that we have developed and distributed. Some of these components have also been used successfully in other laboratories and software environments.

We recognize that the requirement of a Python interpreter for developing and reusing our software components is a constraint in our approach. However, the numerous advantages of this approach largely outweigh this constraint. Python is a general purpose programming language, therefore the Python language is not biased toward any application domain. There is a large user community contributing Python packages that can be readily integrated for the most part and we are using several third party packages in our applications. Having a fully-fledged, high-level, object-oriented programming language as a glue to connect software components greatly reduces the need for stringent APIs for components to be inter-operable because adapters can always be written. Finally, since our software components are developed independently of each other and no component is allowed to assume control of the application, the Python interpreter remains central to the software environment. Python naturally provides scripting capability over the applications and the data they manipulate. An additional benefit of programming in Python is platform independence which is a great advantage when supporting multiple platforms. While there were initial concerns about the performance impact of using an interpretive language, we found that Python is well suited for developing entire software components. This is due to the fact that we and other developers often have a wrong impression about how much and where the utmost performance is needed, and where the true bottlenecks are.

Below we will briefly describe the key differences between compiled and interpretive languages and stress the advantages of interpretive languages in general and Python in particular as integration platforms. We describe some software components we have developed and illustrate their integration into end-user applications. We conclude with some lessons learned, pitfalls to avoid and discuss some of the challenges associated with our approach.

Programming languages

In compiled languages (FORTRAN, C, C++, *etc.*) an executable has to be built (i.e. compiled and linked) from the source code before it can be executed. Extending or modifying these programs requires re-building the application after the source code has been modified. This cannot be done while the program is running. Moreover, these

languages require complete and syntactically correct source code before compilation and testing become possible. Interpretive languages (Perl², Python¹, Tcl³, Ruby⁴, Scheme⁵, Lisp⁶, Basic⁷, etc.) use a program called an interpreter to evaluate source code as it is read from a file or typed in interactively by a user. These languages are often referred to as scripting- or interpreted-languages. Applications written in these languages can be extended and modified while the program is running without having to quit, recompile or restart the application. Code can be executed and tested as it is written, even if only a small fraction of the complete application has been implemented, thus allowing the rapid exploration of novel hypothesis and the early determination of design flaws.

Programming languages can also be classified and described by placing them on a scale ranging from low- to high-level languages. We will deem a language “high-level” if it provides: high-level data structures such as lists and associative arrays; mechanisms for preventing errors from aborting the application (exceptions); array boundary checking, and support for automatic memory management as part of the language. The more such features any particular language provides, the higher it will be ranked. Interpretive languages usually are higher level than compiled languages.

Advantages of Interpretive Languages

Rapid development: The high-level and dynamic nature of interpretive language offers several advantages for scientific computing. The ability to quickly prototype software is particularly interesting in a research environment where the software requirements are constantly shifting as the understanding of the underlying science evolves. **Easy to learn:** Their easy syntax and high-level data structures make it easier for non-professional programmers such as computational biologists to develop programming skills enabling them to interact with data programmatically and eventually develop code on their own. For instance, Python is becoming an increasingly popular language for teaching computational skills⁸. **Open-ended application extensibility:** Software environments enabling end users to interact programmatically with their data and with the application using a simple yet fully fledged programming language provides the highest level of extensibility. While GUIs are an excellent way to abstract programming syntax and data structures, with the exception of visual programming (see below), GUIs can only be implemented for anticipated ways of using a program. This lack of programmability associated with GUI-based interfaces greatly limits the application’s range of capabilities and its extensibility. Modifying an application’s source code enables unrestricted modifications; however, it is usually a difficult task and always presents the danger of corrupting the application. General purpose scripting languages provide a safer and easier way to extend applications, while providing the open-ended extensibility which is missing in GUI-based interfaces. In fact, scripting languages blur the line between users and developers. **Glue languages:** Interpretive language can be used to integrate heterogeneous pieces of software written in a variety of compiled languages and let them interoperate.

Interpretive languages provide new solutions to the challenges mentioned in the introduction and are becoming increasingly popular for developing bioinformatics applications (BioPerl⁹, BioPython¹⁰, Chimera¹¹, PMV¹², Vision¹³, PyMol¹⁴, WhatIf¹⁵, PHENIX¹⁶, MMTK¹⁷, VMD¹⁸) and for teaching programming concepts to biologists⁸.

Performance

A drawback of interpretive languages is that they are slower than compiled languages, typically by a factor of 15 to 40¹⁹. When compiled languages appeared, they rapidly replaced assembly languages. The substantial increase in readability and portability they offered outweighed the associated potential decrease in runtime performance. Today, this pattern is repeating itself: the sustained increase of computational power in typical desktop computers makes interpretive languages a practical alternative to compiled languages for a variety of tasks. Moreover, we often have a wrong intuition about how much and where performance is needed. A key observation is that the bulk of the source code of programs used in computational biology deals with input and output, handling events, and bookkeeping of data structures. The computationally intensive parts are often confined to a few functions that amount to a small percentage of the total number of lines of source code, typically less than 10%. Hence, large amounts of code can be implemented in high-level interpretive languages without affecting the overall performance of the application.

Which scripting language is right for my application?

Scripting languages such as the various UNIX shell scripting languages and awk²⁰ have been available for a long time. However, these languages are arcane and limited in generality. They have been reserved to computer savvy users. A number of high-level scripting languages are available today. Perl was one of the first to be widely used in scientific computing. It is a powerful language for writing highly concise scripts. However, the way its syntax and notation tends to promote obfuscation is not a desirable feature for reusing and sharing code. Perl is mainly used for writing relatively short, “use-once”-type scripts, but is ill suited for developing large and complex software applications²¹. It also is lacking a good interpreter shell for interactive code development. The Tcl language is another example of a widely used scripting language. Unfortunately, its one strength – all data are represented as strings – is also its main weakness for scientific computing. This makes Tcl cumbersome and inefficient for numerical computations. The Tcl-shell is also relatively simple and underdeveloped.

Python appeared in 1991 and was designed from the outset as an object-oriented language while allowing for simple scripting. It supports multiple inheritance, introspection, self-documenting code, high-level data-structures (i.e. lists and associative arrays called dictionaries), and exceptions and warning mechanisms. Its syntax was designed to be free of arcane symbols and to look like English text (i.e. pseudo-code). Python is open source and runs on virtually any computer from super-computers to PDA's. New functionality can be added to the interpreter by loading extensions to the language at runtime. Such extensions can be organized into sets called packages. The standard Python distribution comes with a comprehensive library of extensions covering needs in areas as diverse as regular expression matching, GUI toolkits, databases, network protocols, numerical calculations and XML processing to name a few. In addition, an active community of developers provides a variety of packages spanning all areas of computing, reflecting the diversity of Python's user community and application areas. Python extensions can be written both in the Python programming language and in compiled languages such as FORTRAN, C or C++, thus allowing the incorporation of legacy code. Code written in compiled languages must be “wrapped” (i.e. turned into a

Python extension) before it can be called from a Python interpreter. Semi-automatic tools²² facilitate the process of wrapping compiled code. These extensions execute faster than those written using the Python language. However, they are platform-dependent (i.e. they need to be compiled for every hardware platform and operating system). With Python, it is possible to have “the best of both worlds” by implementing those functions that require the utmost in performance using a compiled language while the remainder of the application can be written in a high-level and platform-independent language.

Python has been used as a “glue language” to integrate monolithic programs. With the wrapping mechanism described above, a scripting layer can be added to such programs, allowing them to inter-operate within a Python interpreter^{23 18; 11}. Using Python for scripting applications should be contrasted with using a custom scripting language (i.e. SVL²⁴, BCL²⁵, SPL²⁶, batchmin²⁷, MATLAB²⁸, etc.). Such languages often lack extensibility and tend to be domain specific since they are created by developers whose strengths are in a specific application domain. Because of their specificity, these languages will not benefit from the input of a large community. Designing a language is better left to people whose primary occupation is designing languages.

While an excellent scripting- and glue-language, Python is also a general-purpose, object-oriented programming language. This aspect has been key for our purpose. It can be used as the primary language for the implementation of complete packages and applications which have the great advantage of platform-independence (i.e. the same source code runs on all platforms). Our software development effort has grown to over 15 packages written in pure Python, amounting to over 2500 classes and well over 1'000'000 lines of code. Such a colossal software base would be extremely difficult to manage without object orientation and hierarchical names spaces. Unfortunately, the often wrong intuition about where and how much performance is needed tends to deter programmers from using interpretive languages. This misconception about performance is difficult to overcome. However, designing and implementing components in the Python programming language first, provides several advantages: (1) The development cycle of a prototype is greatly accelerated through the use of a high-level language and the reuse of software components. It is not uncommon for a Python program to have 3 to 10 times fewer lines of source code than the same program written in C; (2) The design of software components can be validated rapidly. Python code can be tested as it is written. The ability to run code after only a fraction of an application or component has been implemented helps identify design problems early on; (3) Performance bottlenecks can be identified using Python's profiler and resolved by optimizing the Python code, or by providing a re-implementation of parts of the code in C or C++; Finally, (4) a potentially slow but working Python implementation is always available in the case that a C or C++ implementation cannot be loaded.

Software-components and their integration

We have developed a number of Python packages (Table 1). To illustrate our approach, we describe two independently developed software components: MolKit and Deja Vu, and demonstrate their integration for molecular visualization.

MolKit: This software component has no dependencies on other Python packages and provides objects for reading and writing common data file formats describing

biological molecules such as PDB, Mol2 and mmCIF. When a molecule is read, a hierarchical data-structure is built reproducing the natural hierarchy found in molecules such as proteins. Molecule objects can be queried and can also generate information such as atom types, covalent bonds, atomic radii, etc. This component is written in pure Python.

Package	Description
<i>MoKit</i>	Read, write and build hierarchical representation of molecular data structures.
<i>ViewerFramework</i>	Visualization application template. Uses <i>DejaVu</i> for rendering 3-D geometry.
<i>PyBabel</i>	Re-implementation of <i>Babel 1.6</i> (molecular file formats conversion). Supports assigning of atom type and bond order, ring detection, Gasteiger charges, protonation.
<i>MgUtil</i>	Various packages containing mathematical functions, GUI widgets, etc.
<i>FlexTree</i>	Provides hierarchical, high-level representations of molecular flexibility.
<i>PyBabel</i>	A Python re-implementation of <i>Babel 1.6</i> for molecular data file format conversions
<i>shapefit</i>	Protein-Protein docking software package based on the complementarity of smoothed protein shapes
<i>Volume</i>	A package for representing and manipulating 3D regular grids of volumetric data.
<i>WebServices</i>	Generic support for web services
<i>symsev</i>	A set of objects for describing point symmetries
<i>Vision</i>	A component supporting visual programming
<i>ViewerFramework</i>	A boiler plate for creating visualization applications
<i>DejaVu</i>	An OpenGL-based general purpose 3D geometry rendering component
<i>PyARTK</i>	A software component for virtual reality
<i>PyOslim</i>	Python wrapper of the polygonal mesh decimation library <i>Oslim</i> ²⁹ .
<i>Mslib</i>	Python wrapper of the <i>MSMS</i> library ³⁰ for computing molecular surfaces.
<i>UT-packages</i>	A suite of Python wrapper of C++ libraries for fast iso-contouring, volume rendering, pseudo density maps calculations, signed distance fields, meshing techniques, etc. ³¹ .
<i>SFF</i>	Python wrapper of a C-implementation of the <i>AMBER Force Field</i> (Simple Force Field).
<i>SpatialLogic</i>	Python wrapper of a C library performing <i>BSP-Tree</i> -based <i>CSG</i> operations.
<i>Gle</i>	Python wrapper of the <i>GL Extrusion</i> library ³²
<i>opengltk</i>	Python wrapper for <i>OpenGL</i> , generated on the fly from <i>h</i> files
<i>memoryObject</i>	A native extension for speeding up communication between C and C++ and Python code
<i>PyAutoDock</i>	objects implementing the <i>Auto Dock</i> forcefield.
<i>SurfDock</i>	Python wrapper of the protein-protein docking program <i>SurfDock</i>
<i>Stride</i>	A wrapper of the <i>Stride</i> program for assigning secondary structure to proteins
<i>BHtree</i>	A Python wrapper of a C library implementing binary spatial divisions

Table 1: Python packages developed in our laboratory. A shaded background indicates packages that are platform-independent (i.e. written entirely in Python).

DejaVu is our OpenGL-based, platform-independent, general-purpose 3-D geometry visualization component. It defines classes implementing objects such as *Viewer*, *Camera*, *Light*, *ClippingPlane*, *ColorEditor*, *Geometry*, etc. The *Viewer* class implements a fully-fledged visualization application. It provides control over a large number of rendering parameters including: user controllable depth-cueing; global anti-aliasing; perspective and orthographic projection modes; multiple light sources; as well as per geometry: rendering modes (points, lines, polygons, outlined), shading modes (flat, Gouraud), culling modes (back, front, none), arbitrary clipping planes, magic lenses that reveal geometry only inside the lens, blending functions for transparency, etc. Each *DejaVu Viewer* object maintains a hierarchy of geometrical objects. Rendering attributes and 3-D transformations can be defined for any particular geometry in this hierarchy, or can be inherited from a parent. A *Viewer* object contains a virtual trackball object for

rotating, translating and scaling. This trackball can be bound to any geometry, camera, light source, clipping plane, or texture. DejaVu also supports Non Photo-realistic rendering and interactive Mpeg movie generation from the Camera,

DejaVu's set of geometry objects is extensible and currently includes: Polylines, IndexedPolylines, IndexedPolygons, QuadStrips, TriangleStrips, Spheres, Cylinders, Ellipsoids, Arcs3D, Arrows, Box, Points, CrossSets and TextLabels. Such objects can be instantiated and added dynamically to a viewer. DejaVu is written entirely in Python but relies on the presence of the Numeric package as well as opengl, our OpenGL wrapper for Python.

Combining MolKit and DejaVu for molecular visualization: Figure 1 shows the code necessary for reading a molecule using the MolKit component and displaying a CPK model of this molecule (i.e. a single sphere per atom). Nine lines of Python code are sufficient for achieving basic molecular visualization with software components that have been developed independently of each other.

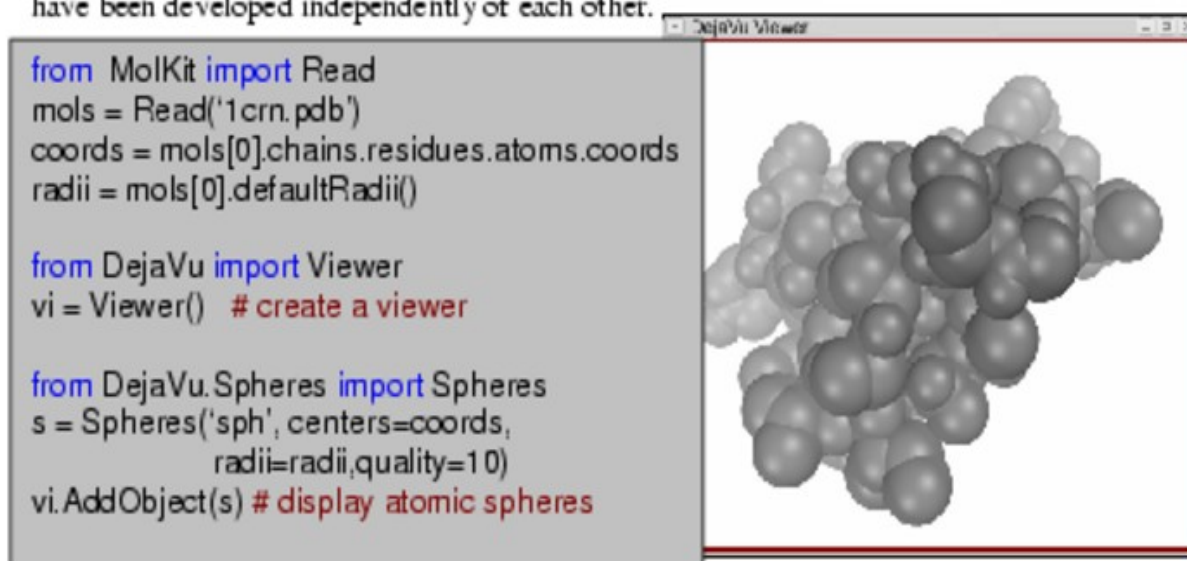


Figure 1: Nine lines of Python code are needed to read a molecule and display a sphere for each atom using two software components developed independently of each other: MolKit and DejaVu. The Read factory function from MolKit builds molecular objects from which data, such as atomic centers can be retrieved, and which can infer attributes such as atomic radii. A Viewer object from DejaVu provides a fully-fledged 3D geometry viewer in which geometry such as a set of spheres can be displayed.

More complex applications: Using the software components described in Table 1, we have created complex applications, including: PMV (Figure 2A), a general purpose molecular visualization and manipulation environment; AutoDockTools, a GUI for setting up and launching AutoDock-based automated docking calculations³³, and analyzing the results; and PyARTK, an augmented reality software in which we combine live video of physical models manipulated by the user with computer graphics enhancing the displayed video³⁴. Both AutoDockTools and PyARTK are specializations of PMV.

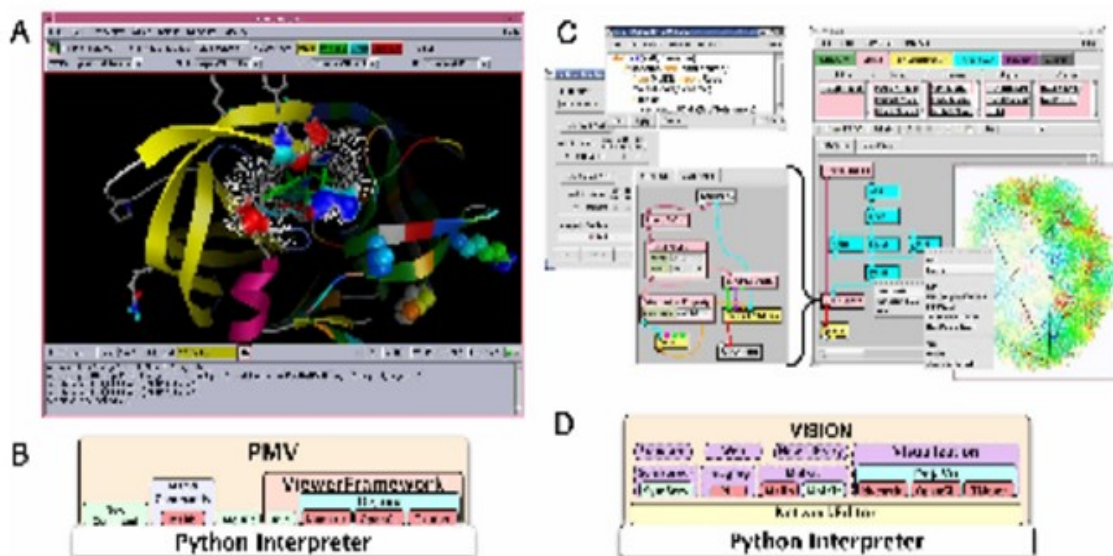


Figure 2: Two applications PMV and Vision built from, and sharing many software components. (A) PMV: a general purpose molecular visualization application. (B) Architectural layout of PMV. Nested boxes denote dependencies between Python packages. Packages with dark pink background are platform-dependent. (C) A molecular visualization application built using the Vision visual programming environment. A network used to display a viral capsid is shown. The sub-network embedded in the "Lines Macro" is shown as an inset. The "node Editor" allowing inspecting and modifying nodes interactively has been started on the "Read Molecule" node. (D) Architectural layout of Vision. Libraries of Vision nodes are shown with dashed outlines. Note the number of shared software components.

Software component integration by non-programmers

In a research setting where goals and questions are constantly evolving with the discovery of new knowledge it is impossible for a programmer to foresee all possible way in which a user might want to combine algorithms to process his/her data and look at the results. The Python interpreter is the foundation of all the applications we develop and is always accessible for scripting thus providing a way to extend and modify any of our applications. However, domain experts such as biologists, often feel that learning a programming language (even as simple as Python) is more than what they ought to be doing. To use an analogy, it is similar to asking a driver to learn to tune his car's engine before he can drive it. We have explored a couple of ideas to address this problem. First, in our ViewerFramework component which is underlying PMV and AutoDockTools, all commands invoked through the graphical user interface generate a log string which corresponds to what a user would have to type in the Python shell to achieve the same result. This can help some of the more sophisticated users to prototype a series of operations, capture the log, and for instance place it inside a loop to apply these operations to a sequence of objects. However, the user still has to worry about proper indentation and other syntactical details of the Python language. In order to hide this level of complexity completely from the user we have explored the concept of visual programming. We have implemented a software component called Vision (figure 2C)¹³ which support dragging and dropping computational nodes onto a canvas and connecting their input and output ports for creating workflows and computational networks. Libraries of Vision node expose the functionality from other Python packages such as

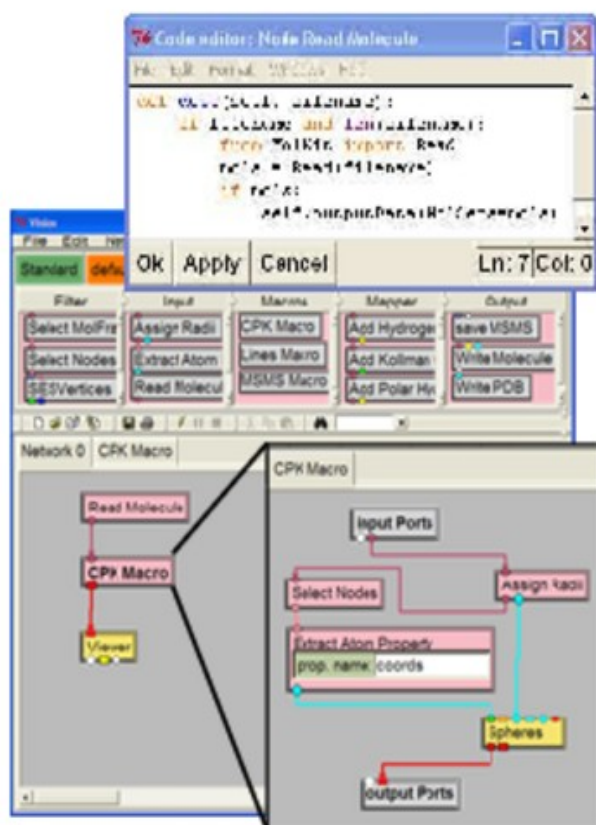


Figure 3: a simple Vision network corresponding to the Python script in Figure 1. The nodes are colored by the library they originate from. The CPK macro node is expanded in the inset. A user can create this network without any knowledge of the Python syntax or the data structures used to store molecules.

MolKit and DejaVu. Figure 3 shows a Vision network that is equivalent to the Python script shown in figure 1. This paradigm empowers the end user with the ability to extend the application with new functionality and novel combination of computational methods without having to learn the syntax of a programming language, or getting intimate with the data structures used to store objects. A key difference between Vision and other visual programming languages such as Open DX³⁵ or AVS³⁶ is that Vision is a software component that can be reused in other applications. We have demonstrated its integration with PMV³⁷. All the domain-specific knowledge is in the Vision nodes found in various libraries. The Vision nodes are lightweight wrappers for functionality otherwise available in Python. For instance the full implementation of the Read Molecule node in Figure 3 is to get the input (i.e. the file name), invoke the Read function from the MolKit package, and output the resulting Python object on the output port. In this sense, the node is merely

an adaptor for the Read function in the Vision environment.

Lessons learned and pitfalls

Python's modular nature does promote the compartmentalization of code. However, for software components to achieve their full potential in terms of re-usability special care has to be taken beside the obvious idea described earlier which is to develop components in isolation. Here we provide a short list of common mistakes we have identified over the years. Simple mistakes include using global variables. This is always undesirable in object oriented environment as it precludes the simultaneous use of multiple instances of objects referring to these variables.

One common pitfall in object-oriented languages is the tendency to create a large hierarchy of objects for all software developed by a group. While the idea is appealing from an esthetic point of view, it can be quite damaging from the software engineering aspect. The popular visualization toolkit VTK³⁸ provide a typical example of such a situation where the complete source has to be downloaded and compiled before any part of the toolkit can be used. Moreover, if the build fails half way it is often difficult to know whether or not the part that is of interest has built properly or not. We found that

keeping the package hierarchy as flat as possible and reducing inter-packages dependencies to a minimum leads to much more reusable components. The same is true with data-types. In computational biology it is often tempting to pass complex objects such as a molecule to a given computational method which might only be using a small fraction of the information stored in the molecule object. Reducing the information passed into the computational method to the strict minimum that is needed is a better approach. For instance in the example in Figure 1, we could have passed a molecule to the viewer, but this approach would make the visualization component aware of molecular objects. Alternatively, a molecule could have a display method, but here again, the MolKit component would become aware of DeJaVu. Instead, we are passing 2 numeric arrays, one for coordinates and one for radii, because that is all that is needed for drawing the spheres. By passing the simplest data types between software components, we reduce the dependencies between these components and minimize the data structures we impose onto other users of these components. Developers of libraries written in the C and C++ programming languages sometimes choose to make their code exit if input files are not found or an error is encountered. This is highly undesirable in our setting, as such a call would terminate the Python interpreter and hence our working session. We have also found that techniques such as writing unit tests before writing code lead to better designed classes as it forces the programmer to think more thoroughly about the input to methods and behavior of the object when the input is wrong. It is also helpful to think of the object in more than a single context. While this violates some extreme programming principles and can lead to “over designing” objects we have found it to be useful in identifying general aspects from domain specific ones in objects.

Conclusion

The concepts of modularity and compartmentalization of computational tasks are not new; however, time has shown that these concepts are poorly promoted by compiled languages. When programming in the latter languages, the goal is to write an application that produces the right result. There is little incentive to implement independent and therefore reusable software components for solving each particular sub-task. In fact, even if the software is properly compartmentalized, special “main” functions have to be written to ascertain the independence and correctness of the various components, which is rarely done. The intrinsic modular nature of Python on the other hand promotes the creation of self-contained components each implementing a specific functionality. This component can then be loaded into a Python interpreter, effectively extending the interpreter with new functionality. Moreover, this component naturally becomes available to any application running a Python interpreter.

We have described an approach in which the Python interpreter is the foundation of the software environment and serves as both an integration platform and a primary development language for new software components. With this approach we have witnessed extremely high levels of code reuse both within and outside our laboratory. While this approach has been successful in addressing many of the shortcomings of software frameworks, it has also revealed a new set of challenges. While applications built from software components are desirable for many reasons, they increase the complexity of software distribution, as the right set of components (i.e. compatible versions of all components) has to be used. Better support in the Python language for

package management would help with this problem. Keeping track of inter-packages dependencies is also tricky. We have found that such dependencies are easily created and often inadvertently. To address this issue we have started to explicitly declare dependencies in all of our packages. These dependencies can be critical, if the package cannot function without the package it depends on, or weak if the dependencies is only needed for a secondary feature of the package. Our nightly unit tests monitor imports and report as errors imports of packages that are not declared in the dependency lists.

As modern computational biology moves towards the study of larger systems it will require the integration of computational methods and experimental data from a variety of scientific fields. Furthermore, the complexity of the models will require an unprecedented level of flexibility in the software tools to allow investigators to formulate and validate new hypotheses. The interactive and dynamic nature of Python, its simplicity, and ease-of-use, makes the language an excellent choice for creating software tools with these advanced and complex requirements.

Acknowledgments: The development of the Python-based software components described in this article was supported through the NSF grant CA ACI9619020 and the NIH grant RR08605. This is manuscript 18323 from The Scripps Research Institute.

REFERENCES:

1. Lutz, M.; Ashet, D., *Learning Python*. March 1999 First Edition ed.; O'Reilly & Associates: 101 Morris Street, Sebastopol, CA 95472, 1999.
2. Tisdall, J., *Mastering Perl for Bioinformatics*. O'Reilly: 2003.
3. Ousterhout, J., *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
4. Thomas, D., Hunt, A., *Programming Ruby - The Pragmatic Programmer's Guide*. 1st edition (December 15, 2000) ed.; Addison Wesley Longman, Inc.: 2001; p 608.
5. Dybvig, R. K., *The Scheme Programming Language Second Edition*. Prentice Hall PTR: 1996.
6. Steele, G. L., *Common Lisp the Language, 2nd edition*. Digital Press: 1990; p 1029.
7. Lien, D. A., *The Basic Handbook: Encyclopedia of the BASIC Computer Language*. 3rd ed.; CompuSoft Publishing: 1986.
8. Schuerer, K.; Letondal, C. Python course in Bioinformatics. <http://www.pasteur.fr/techeche/unites/sis/formation/python/>
9. BioPetl BioPetl project home page. <http://bio.petl.org/>
10. BioPython BioPython project home page. <http://www.biopython.org/>
11. Huang, C. C.; Couch, G. S.; Pettersen, E. F.; Fertin, T. E. In *Chimera: An Extensible Molecular Modeling Application Constructed Using Standard Components*, Pacific Symposium on Biocomputing, 1996; 1996; p 724.
12. Coon, S.; Sanner, M. F.; Olson, A. J. In *Re-Usable components for Structural Bioinformatics*, 9th International Python conference (IPC9), Long Beach, California, 2001; Long Beach, California, 2001; pp 157-166.
13. Sanner, M. F.; Stoffler, D.; Olson, A. J. In *ViPER, a visual programming environment for Python*, Proceedings 10th International Python Conference, Alexandria, VA, February 4-7, 2002, 2002; Alexandria, VA, 2002; pp 103-115.
14. DeLano, W. L. *The PyMOL Molecular Graphics System*, Delano Scientific: San Carlos, CA, 2002.
15. Vriend, G., WHAT IF: A molecular modeling and drug design program. *J. Mol. Graph.* **1990**, *8*, 52-56.
16. Adams, P. D.; Grosse-Kunstleve, R. W.; Hung, L. W.; Ioerger, T. R.; McCoy, A. J.; Moriarty, N. W.; Read, R. J.; Sacchettini, J. C.; Sauter, N. K.; Terwilliger, T. C., PHENIX: building new software for automated crystallographic structure determination. *Acta Crystallogr D Biol Crystallogr* **2002**, *58*, (Pt 11), 1948-54.

17. Hinszen, K. In *The molecular modeling toolkit: A case study of a large scientific application in Python*, 6th International Python Conference, San Jose, CA, October 14-17, 1997, 1997; San Jose, CA, 1997.

Parts-of-Speech Tagger Errors Do Not Necessarily Degrade Accuracy in Extracting Information from Biomedical Text

18. Cowell-Shah, C. W. Nine Language Performance Round-up: Benchmarking Math & File I/O. http://www.osnews.com/story.php?news_id=5602

19. Aho, A.; Kernighan, B.; Weinberger, P., *The AWK Programming Language*. Addison Wesley:

Maurice HT Ling¹, Christophe Lefevre^{1,2}, Kevin R Nicholas¹

20. Raymond, E. S., Why Python? In *Linux Journal*, 2001.

21. SWIG *Simple Wrapper Interface Generator (SWIG)*

¹CRC for Innovative Dairy Products, Department of Zoology, The University of Melbourne, Australia

²Victorian Bioinformatics Consortium, Monash University, Australia

22. Santavy, M.; Labate, P. SVL: The Scientific Vector Language.

http://www.chemcomp.com/Journal_of_CCG/Features/svl.htm

23. Daelen, T. v. *The Insight Scripting Language*.

Corresponding email: mauriceling@acm.org

24. Tripos SYBYL, 6.9.2; Tripos Inc., .

25. Schrodinger *MacroModel*.

26. Hanselman, D.; Littlefield, B. R., *Mastering MATLAB 6, 1/e*. Prentice Hall: 2001; p 832.

27. Garland, M. *QSlip Simplification Software*, 2.0; 1999.

Abstract

28. Sanner, M. F.; Olson, A. J.; Spehner, J.-C., Reduced surface: An efficient way to compute molecular surfaces. *Biopolymers* 1996, 38, 305-320.

Background: An ongoing assessment of the literature is difficult with the rapidly increasing volume of research publications and limited effective information extraction tools which identify entity relationships from text. A recent study reported development of Muscorian, a generic text processing tool for extracting protein-protein interactions from text that achieved comparable performance to biomedical-specific text processing tools. This result was unexpected since potential errors from a series of text analysis processes is likely to adversely affect the outcome of the entire process. Most biomedical entity relationship extraction tools have used biomedical-specific parts of speech (POS) tagger as errors in POS tagging and are likely to affect subsequent semantic analysis of the text, such as shallow parsing. This study aims to evaluate the parts of speech (POS) tagging accuracy and attempts to explore whether a comparable performance is obtained when a generic POS tagger, MontyTagger, was used in place of MedPost, a tagger trained on biomedical text. **Results:** Our results demonstrated that MontyTagger, Muscorian's POS tagger, has a POS tagging accuracy of 83.1% when tested on biomedical text. Replacing MontyTagger with MedPost did not result in a significant improvement in entity relationship extraction from text; precision of 55.6% from MontyTagger versus 56.8% from MedPost on directional relationships and 86.1% from MontyTagger compared to 81.8% from MedPost on nondirectional relationships. This is unexpected as the potential for poor POS tagging by MontyTagger is likely to affect the outcome of the information extraction. An analysis of POS tagging errors demonstrated that 78.5% of tagging errors are being compensated by shallow parsing. Thus, despite 83.1% tagging accuracy, MontyTagger has a functional tagging accuracy of 94.6%. **Conclusions:** The POS tagging error does not adversely affect the information extraction task if the errors were resolved in shallow parsing through alternative POS tag use.

1. Introduction

PubMed currently indexes more than 17.5 million papers that includes 1 million papers added in both 2006 and the first half of 2007. This trend of increased volume of research papers makes it difficult for researchers to maintain a productive assessment of relevant literature. Information extraction (IE) has been used as a tool to analyze biological text to derive assertions, such as entity interactions (Abulaish and Dey, 2007). To date, there has been a number of IE tools to extract entity interactions from published text, such as MedScan (Novichkova et al., 2003), Arizona Relation Parser (Daniel et al., 2004), BioRAT (David et al., 2004) and Santos et al. (2005).

A recent article by Ling et al. (2007) has classified entity interaction IE tools by whether tools are developed with biological text in mind or adapted generic tools for biological text. Ling et al. (2007) developed Muscorian, a tool to extract protein-protein interactions from text. They also demonstrated that a generic text analysis tool chain, MontyLingua (Liu and Singh, 2004; Ling, 2006), incorporated into a two-layered generic-specialized architecture as explained in MedScan (Novichkova et al. 2003), can give rise to comparable performance in entity interaction extraction compared to those IE systems that modified existing systems, such as BioRAT (David et al., 2004), Chilobot (Chen and Sharp, 2004) and Santos et al. (2005). One of the common features of both classes of tools defined by Ling et al. (2007) is the specialization of the part-of-speech (POS) tagger. For example, Arizona Relation Parser (Daniel et al., 2004) re-trained Brill tagger (Brill, 1995) and Chilobot (Chen and Sharp, 2004) re-trained TnT tagger (Brants, 2000). POS tagging is a process of assigning grammatical roles of each word and punctuation in the source sentence. This plays a critical role in subsequent text processing tasks, such as shallow parsing, where the sequence of POS tags were used instead of the original sequence of words. At the same time, it was known that errors in POS tagging often results in misunderstanding of the sentence (Kodratoff et al., 2005; Amrani et al., 2005).

Muscorian (Ling et al., 2007) makes use of a generic POS tagger as part of MontyLingua (Ling, 2006; Liu and Singh, 2004) and performs at a comparable level to IE tools using POS taggers trained on biomedical text. This contradicts the common view that "*error propagation through cascades of processors may in aggregate severely degrade performance on the final task*" as stated in the *Call for Papers for the Tenth Conference on Natural Language Processing 2006 (CoNLL-X)*. Tateisi and Tsujii (2004) have demonstrated that generic POS taggers are only about 83% accurate when used to tag biomedical text. This suggests that MontyTagger, the generic POS tagger in MontyLingua, is unlikely to perform as well as taggers trained on biomedical text, such as MedPost (Smith et al., 2004). Therefore, it is likely that the above mentioned contradiction is resolved at the step immediately downstream to POS tagging, the shallow parsing. In MontyLingua shallow parsing (Ling et al., 2007), the input sentence is broken into noun phrase and verb phrase. The process of shallow parsing can be

seen as a collapse of a sequence of POS tags into 2 groups; hence, we expect high level of permissible substitution of POS tags within related classes. We term this permissible substitution as “alternate POS tag use”.

This study compares the performance of MedPost (Smith et al., 2004) with the generic POS tagger, MontyTagger (Liu and Singh, 2004), in Muscorian (Ling et al., 2007) and illustrates a case whereby POS tagging error does not adversely affect the final information extraction task if the errors were resolved in shallow parsing through alternate POS tag use.

2. *Methods*

2.1. Evaluating POS Tagging and Information Extraction Performance

MontyTagger was evaluated on its own using MedPost corpus (Smith et al., 2004) and its accuracy as the percentage of the number of correctly tagged tokens (words and punctuations) in the total number of tokens (n=182399). MedPost tagger was swapped in place of MontyTagger by modifying MontyLingua's *jist()* and *jist_predicates()* functions to *mpjist()* and *mpjist_predicates()*, giving MedPost-MontyLingua Muscorian:

```
def jist(self,text):
    sentences =
self.split_sentences(text)
    tokenized =
map(self.tokenize,sentences)
    tagged =
map(self.tag_tokenized,tokenized)
    chunked =
map(self.chunk_tagged,tagged)
    extracted =
map(self.extract_info,chunked)
    return extracted
```

```
def jist_predicates(self,text):
    infos = self.jist(text)
    svoos_list = []
    for info in infos:
        svoos =
info['verb_arg_structures_concise']
        svoos_list.append(svoos)
    return svoos_list
```

to

```

def mpjist(self,text):
    sentences =
        self.split_sentences(text)
    tokenized =
        map(self.tokenize,sentences)
    sourcefilename =
        random.random()*10000000000
    outfilename =
        random.random()*100000000000
        0
    source = open('temp' + os.sep +
        str(sourcefilename), 'w')
    source.writelines(tokenized)
    source.close()
    os.popen(os.getcwd() + os.sep +
        'medpost/medpost -text -token
        -penn < temp' + os.sep +
        str(sourcefilename) + '> temp' +
        os.sep + str(outfilename))

    mpout = open('temp' + os.sep +
        str(outfilename), 'r')
    tagged = mpout.readlines()
    mpout.close()
    chunked =
        map(self.chunk_tagged,tagged)
    extracted =
        map(self.extract_info,chunked)
    return extracted

def mpjist_predicates(self,text):
    infos = self.mpjist(text)
    svoos_list = []
    for info in infos:
        svoos =
            info['verb_arg_structures_concise'
            ]
        svoos_list.append(svoos)
    return svoos_list

```

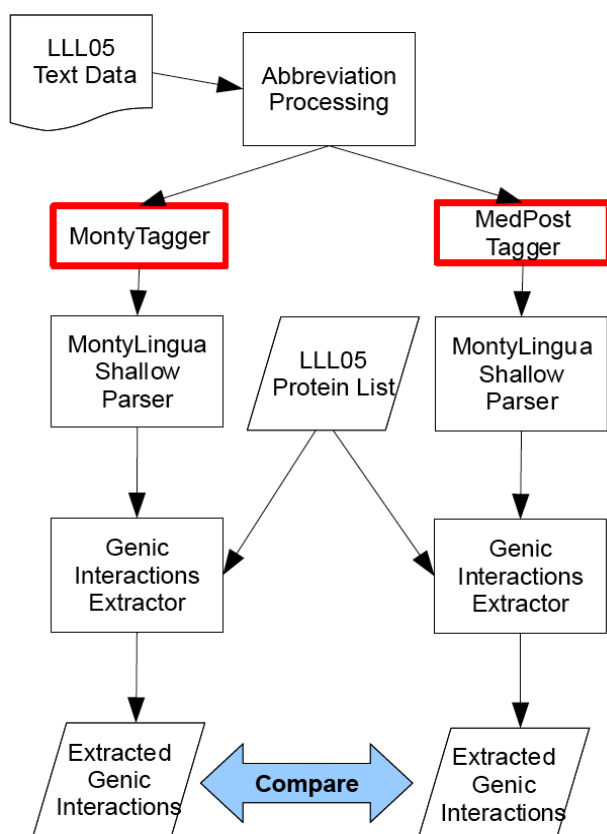


Figure 1. Flowchart of evaluation procedure for Muscorian with native MontyLingua and MedPost-MontyLingua. LLL05 test data was processed for abbreviations before feeding into each system and the extracted genic interactions (output) were evaluated for precision and recall.

MedPost-MontyLingua Muscorian's IE performance was evaluated using Learning Languages in Logic 2005 test data (Cussens and Nedellec, 2005) in the same manner as Muscorian (Ling et al., 2007) and the performances were compared (Figure 1).

2.2. Analysis of POS Tagging Errors

Wrongly tagged tokens from MontyTagger's output were first grouped by their original tags in MedPost corpus (Smith et al., 2004), then sub-grouped by MontyTagger's assigned tags (the wrong tag) and arranged in decreasing order based on the numbers of tags in both main and sub-group. First 80% of the tags in the main group where first 90% of the wrongly assigned tags were chosen for further error analysis. Each of the pairs of original tag and wrongly assigned tag were analysed with respect to the regular expressions in MontyREChunker (Ling et al., 2007), the shallow parser in MontyLingua, for the effects of the wrongly assigned tags on the operations of the shallow parser.

3 Results

3.1 Evaluating POS Tagging and Information Extraction Performance

Evaluating MontyTagger on MedTag corpus demonstrated correct tagging in 151663 of the tags representing 83.1% tagging accuracy. Using the LLL05 evaluation corpus, Muscorian with MedPost-MontyLingua on directional relationship was found to be 56.8% precise with 24.8% recall, while nondirectional relationship was estimated to be 81.8% precise with 35.6% recall (Table 1).

	<i>Directional Relationships</i>		<i>Nondirectional</i>	
	MontyLingua	mpMontyLingua	MontyLingua	mpMontyLingua
Precision	55.6%	56.8%	86.1%	81.8%
Recall	19.8%	24.8%	30.7%	35.6%
F-Score	0.292	0.345	0.453	0.496

Table 1. Summary of Muscorian's performances evaluated using Learning Languages in Logic 2005 data (Cussens, 2005).

3.2. Analysis of POS Tagging Errors

Comparison of the reference tags (MedPost corpus) with the wrongly assigned tags from MontyTagger showed the 30736 wrongly assigned tags (52.3%, n=16067) should be tagged as nouns (tag: 'NN'), 15.8% (n=4865) should be tagged as 'JJ' (adjectives), and the next four most common wrongly assigned tags were 'NNS' (n=1987, 6.5%), 'SYM' (n=1496, 4.9%), 'VBP' (n=1470, 4.8%), and 'VBD' (n=745, 2.4%). These six reference tags (NN, JJ, NNS, SYM, VBP, VBN) accounted for 26630 (86.6%) of the wrongly assigned tags, while the rest of the errors (n=4106) were distributed across 25 tags. Six tags (TO, :, (,), WP, ,) were correctly assigned in every instance

in this evaluation. A tabulation of errors is shown in Table 2 and a table providing the definition of each POS tag is given in Table 3. The confusion matrix can be found at http://ib-dwb.sf.net/Muscorian/MedPost_confuse.txt.

Tag	% Corpus	% Error in Total Error	% Error in Tag	Tag	% Corpus	% Error in Total Error	% Error in Tag
NN	28.56	52.27	30.84	VBG	0.64	0.06	1.59
IN	13.49	1.08	1.33	:	0.54	0.00	0.00
JJ	10.47	15.81	25.44	MD	0.43	0.01	0.2
DT	7.77	0.56	1.16	WDT	0.45	0.19	6.70
NNS	7.75	6.45	14.03	,	0.39	0.00	0.00
CC	6.66	1.30	3.29	PRP\$	0.28	0.01	0.40
.	3.67	0.01	0.03	FW	0.26	0.96	61.39
CD	3.13	2.02	10.84	WRB	0.23	0.59	43.33
VBN	3.05	1.70	10.13	JJR	0.17	0.17	17.74
VBD	2.81	2.42	14.56	NNP	0.14	0.03	3.53
RB	2.57	1.72	9.49	EX	0.08	0.01	1.38
)	1.89	0.00	0.00	POS	0.06	0.06	15.31
(1.88	0.00	0.00	WP	0.06	0.00	0.00
VBP	1.98	4.78	41.26	JJS	0.05	0.02	6.60
TO	1.55	0.00	0.00	RBS	0.05	0.01	4.40
VBZ	1.54	0.45	5.20	"	0.03	0.19	100.00
SYM	1.07	4.87	76.43	`	0.03	0.19	100.00
PRP	0.88	1.61	30.59	PDT	0.02	0.11	100.00
VB	0.74	0.05	1.11	RBR	0.01	0.03	44.44

Table 2. Percentage breakdown of POS tags in MedTag corpus and errors in MontyTagger as percentage of POS tags assignment. This table tabulates the POS tagging errors made by MontyTagger on MedTag corpus and the order is according to the abundance of each tag in the MedTag corpus. For example, 'NN' is the most abundant tag accounting for 28.56% or 52093 of MedTag corpus of 182399 tokens. Of which, 3084% (16067 of 52093) of the 'NN' tokens in MedTag corpus were wrongly assigned to a different POS tag by MontyTagger which accounted for 52.27% of the total wrongly assigned POS tag of 30736 tokens.

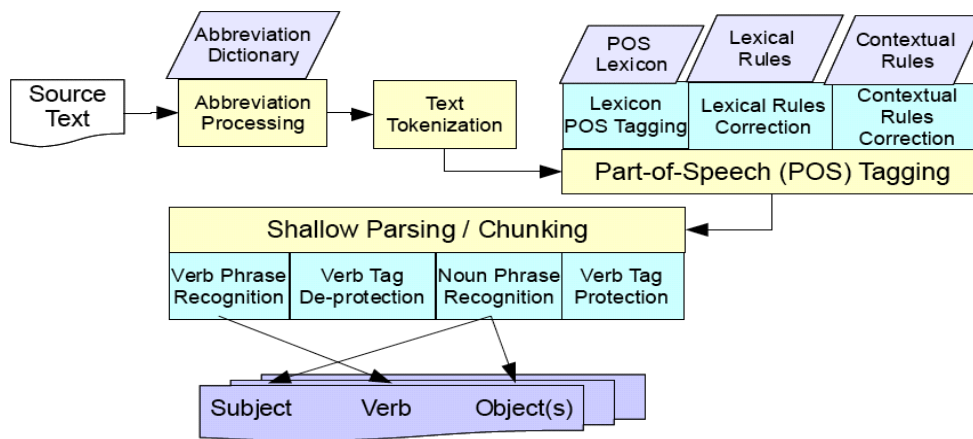


Figure 2. Muscorian's generalization layer, from source text to subject-verb-object(s) structures (Ling et al., 2007).

Tag	Description	Tag	Description
CC	Coordinating conjunction	PRP\$	Possessive pronoun
CD	Cardinal number	RB	Adverb
DT	Determinant	RBR	Adverb, comparative
EX	Existential <i>there</i>	RBS	Adverb, superlative
FW	Foreign word	RP	Particle
IN	Preposition or subordinating conjunction	SYM	Symbol
JJ	Adjective	TO	to
JJR	Adjective, comparative	UH	Interjection
JJS	Adjective, superlative	VB	Verb, base form
LS	List item marker	VBD	Verb, past tense
MD	Modal	VBN	Verb, past participle
NN	Noun, singular or mass	VBG	Verb, gerund or present participle
NNS	Noun, plural	VBP	Verb, non-3 rd person singular present
NNP	Proper noun, singular	VBZ	Verb, 3 rd person singular present
NNPS	Proper noun, plural	WDT	Wh-determiner
PDT	Predeterminer	WP	Wh-pronoun
POS	Possessive ending	WP\$	Possessive wh-pronoun
PRP	Personal pronoun	WRB	Wh-adverb

Table 3. Penn Treebank Tag Set without Punctuation Tags (Adapted from (Marcus et al., 1993))

An understanding of the general scheme of operations of MontyLingua as described in Ling et al. (2007), especially downstream process of POS tagging, the process of shallow parsing by MontyREChunker (MontyLingua's shallow parser) is crucial in our error analysis (Figure 2). Source text (abstracts) were processed for abbreviations and tokenized into sentences, then words and punctuations, before POS tagging. In POS tagging, each token was tagged first using a lexicon and corrected using lexical and contextual rules. This was where the output was 83.1% accurate compared to 96.9% in MedPost. POS tagging could be seen as a reduction of potentially unlimited human English words into 45 “words” or tags using knowledge of English grammar, and the sequence of tags was the input to the shallow parser, MontyREChunker.

Firstly, verb tags (VBD, VBG and VBN) were protected by suffixing the tags to prevent interference in subsequent noun phase recognition (Ling et al., 2007). This meant that wrong tagging between these three tags, such as VBD was erroneously tagged as VBN, had no effect on this process. However, wrong tagging of any of these three tags to any of the other 42 tags or the other way around will be detrimental to this process. Secondly, noun phrases were recognized by the following regular expression (according to Python regex engine in the Python standard library):

```
(((((PDT)?(DT|PRP[$]|WDT|WP[$]))(VBG|VBD|VBN|JJ|JJR|JJS|,|CC|NN|NNS|NNP|NNPS|CD))*(NN|NNS|NNP|NNPS|CD)+)|((PDT)?(JJ|JJR|JJS|,|CC|NN|NNS|NNP|NNPS|CD))*(NN|NNS|NNP|NNPS|CD)+)|EX|PRP|WP|WDT)POS)?(((PDT)?(DT|PRP[$]|WDT|WP[$]))(VBG|VBD|VBN|JJ|JJR|JJS|,|CC|NN|NNS|NNP|NNPS|CD))*(NN|NNS|NNP|NNPS|CD)+)|((PDT)?(JJ|JJR|JJS|,|CC|NN|NNS|NNP|NNPS|CD))*(NN|NNS|NNP|NNPS|CD)+)|EX|PRP|WP|WDT)
```

A number of relationships that potentially contribute to reduced POS tagging errors were considered. Firstly, these four tags; DT, PRP[\$], WDT, and WP[\$]; were alternatives to each other and erroneous tagging between them had no effect on shallow parsing. Secondly, the ten tags; JJ, JJR, JJS, “,” ,CC, NN, NNS, NNP, NNPS, and CD; were alternatives to each other. Lastly, these four tags; EX, PRP, WP, and WDT; were alternatives to each other.

Subsequently, verb phrases were de-protected by removing the suffix appended during the tag protection phase (Ling et al., 2007), followed by verb phrase recognition. This meant that verb tag protection had the highest precedence, followed by noun phrase recognition, and then verb phrase recognition. This meant that nullified errors in higher precedence would not affect downstream processes. Verb phrases were recognized by the following regular expression:

```
(RB|RBR|RBS|WRB)*(MD)?(RB|RBR|RBS|WRB)*(VB|VBD|VBG|VBN|VBP|VBZ)|(VB|VBD|VBG|VBN|VBP|VBZ|RB|RBR|RBS|WRB)*(RP)?(TO|RB)*(VB|VBN)|(RP)?)?
```

In terms of compensation for POS tagging errors, this meant that the four tags; RB, RBR, RBS, and WRB; and these six tags; VB, VBD, VBG, VBN, VBP, and VBZ; were alternatives to each other. However, verb phrase required terminal VB or VBN, which meant that although verb tag protection allowed for interchangeable use of VBN, VBG and VBD, erroneous tagging of VBG and VBD to VBN or VBN to VBG or VBD would be detrimental to verb phrase recognition.

A deeper analysis was undertaken to examine the errors in each reference tag (tabulated in Table 4). Firstly, by grouping close POS types, for example 'NN', 'NNP', and 'NNS' were all nouns, wrong sub-type assignment, such as 'NN' assigned as 'NNP' and 'NNS' assigned as 'NNP', accounted for 55% of the errors (n=14634). Secondly, 58% (n=2818) of 'JJ' (adjective) errors were resulted by tagging as noun (NN and NNP) while 34.9% (n=1698) of the 'JJ' errors were tagged as verb (VBN and VBG). Thirdly, about 5% (n=941) of 'NN' (noun) errors were tagged as cardinal numbers (CD). Fourthly, plural nouns accounted for 51.6% (n=1026) of 'NNS' (singular noun) errors. Fifthly, 48.7% (n=729) and 39.3% (n=587) of 'SYM' (symbol) errors were either not assigned or assigned as 'NN' (noun) respectively. Lastly, 87% (n=1927) of verb errors (VBP and VBD) were due resolution of tenses, such as non-third

party singular present tense (VBP) was assigned as infinite verb form (VB).

Error breakdown (in Table 4) demonstrated erroneous POS tagging by MontyTagger in 31 tags, with 6 tags having no errors. A total of 6 of the 32 tags (19.4%) accounted for 86.6% (n=26630) of the total errors and were chosen for further analysis. Applying these error nullification rules to each of the examined erroneous tags (86.6% of the errors), it was found that 78.6% of the errors had no effect on shallow parsing. A tabulated analysis is shown in Table 4.

Reference Tag	Wrongly Assigned Tag	Number of Wrong Assignment	Cummulative Frequency for Reference Tag	Impact on Shallow Parsing?
NN (16067)	NNP	10865	67.6%	No, NNP was an alternative match to NN in noun phrase
	JJ	2527	83.4%	No, JJ was an alternative match to NN in noun phrase recognition
	CD	941	89.2%	No, CD was an alternative match to NN in noun phrase
	VBG	812	94.3%	Yes, protected verb tag
JJ (4865)	NN	1600	32.9%	No, NN was an alternative match to JJ in noun phrase
	NNP	1218	58.0%	No, NNP was an alternative match to JJ in noun phrase
	VBN	1170	82.0%	Yes, protected verb tag
	VBG	528	92.8%	Yes, protected verb tag
NNS (1987)	NNP	1026	51.6%	No, NNP was an alternative match to NNS in noun phrase
	NN	701	86.9%	No, NN was an alternative match to NNS in noun phrase
	VBZ	128	93.4%	No, VBZ was an alternative match to NNS in noun phrase and was not a protected verb tag

Reference Tag	Wrongly Assigned Tag	Number of Wrong Assignment	Cummulative Frequency for Reference Tag	Impact on Shallow Parsing?
SYM (1496)	Not Assigned	729	48.7%	No, tokens not tagged were non-existent and SYM was not
	NN	587	88.0%	Yes, NN was matched in noun phrase
	-	115	95.7%	No, both tags was not used in shallow parsing
VBP (1470)	VB	1249	85.0%	Yes, mandatory requirement of VB in
	NN	178	97.1%	No, NN was an alternative match to VBP in noun phrase
VBD (745)	VBN	678	91.0%	Yes, mandatory requirement of VBN
	JJ	34	95.6%	Yes, protected verb tag

Table 4. Error breakdown and analysis on the effects of six most commonly mis-assigned POS tags. Six reference tags; NN, JJ, NNS, SYM, VBP, and VBD; which accounted for 86.6% of all wrong POS assignation by MontyTagger were chosen and in each tag, the assigned tags which accounted for 90% of the errors were chosen for further analysis. For example, of 16067 tags that were tagged as 'NN' in MedTag corpus, MontyTagger wrongly tagged 10865 tokens as 'NNP' and has no effect on shallow parsing, 2527 tokens as 'JJ' and has no effect on shallow parsing, 941 tokens as 'CD' and has no effect on shallow parsing, and 812 tokens as 'VBG' with an effect on shallow parsing. These 4 wrong tagging accounted for 94.3% of all 'NN' tag errors. This also meant that 922 'NN' tag errors (5.7%) were not further analyzed. A complete confusion matrix is given in <http://ib-dwb.sf.net/Muscorian/MedPost-confuse.txt>.

4. Discussion

The precision and recall of native MontyLingua Muscorian for extracting genic interactions from the LLL05 data set (Cussens and Nedellec, 2005) was 55.6% and 19.7% (F-score = 0.29) respectively for directional interactions which is about 5% higher in precision and similar in recall to that reported in LLL05 (Cussens and Nedellec, 2005). The precision and recall was 86.1% and 30.7% (F-score = 0.45) respectively for nondirectional interaction. The term “directional” means that the direction of protein activity is non-commutative, for example, “proteinA activates proteinB” does not the same as “proteinB activates proteinA”. However, nondirectional means that the protein activity is commutative, for example, “proteinA binds

to proteinB” has no different biological significance than “proteinB binds to proteinA”. This formed the baseline to evaluate a biomedical-specialized part-of-speech (POS) tagger (Smith et al., 2004) modification of Muscorian, MedPost-MontyLingua Muscorian. The main reason for examining this specialized POS tagger was that it was developed for biomedical information extraction systems (Daniel et al., 2004; Chen and Sharp, 2004) and POS tagging errors were known to be detrimental in understanding human text (Kodratoff et al., 2005; Amrani et al., 2005). In addition, POS tagger modification had been done in a number of biomedical information extraction systems, such as Jang et al. (2006) and Chilibot (Chen and Sharp, 2004).

Examining MontyLingua's source codes, the main function that processes text is the *jist_predicate()* function, which calls the *jist()* function to process text (tokenization, POS tagging and shallow parsing) and then to extract the resulting set of subject-verb-objects (SVO) from *jist*'s output (Ling et al., 2007). The Python codes for these two functions were as follows:

```
def jist(self,text):
    sentences =
self.split_sentences(text)
    tokenized =
map(self.tokenize,sentences)
    tagged =
map(self.tag_tokenized,tokenized)
    chunked =
map(self.chunk_tagged,tagged)
    extracted =
map(self.extract_info,chunked)
    return extracted

def jist_predicates(self,text):
    infos = self.jist(text)
    svoos_list = []
    for info in infos:
        svoos =
info['verb_arg_structures_concise']
        svoos_list.append(svoos)
    return svoos_list
```

As observed, *jist()* function calls *tokenize* function to tokenize the text, *tag_tokenized* function to perform POS tagging, *chunk_tagged* function to perform shallow parsing, and finally, *extract_info* function to extract SVOs from the parsed text. The systematic structure of MontyLingua's codes, especially the *jist()* function had simplified the substitution of MontyTagger (by *tag_tokenized* function) with MedPost. This implied that any of the other components in the text analysis process, like shallow parser (by *chunk_tagged* function) could be easily exchanged.

The precision and recall of MedPost-MontyLingua Muscorian evaluated using the LLL05 data set (Cussens and Nedellec, 2005) were 56.8% and 24.8% (F-score = 0.35) respectively for directional interactions, and 81.8% and 35.6% (F-score = 0.50) respectively for nondirectional interaction. Our results showed that using MedPost in place of MontyLingua's POS tagger, MontyTagger, had improved the F-score by about 5% in both directional and nondirectional interactions extraction, and recall (24.8% versus 19.7% and 35.6% versus 30.7%). However, as reasoned in Ling et al. (2007), precision was more important than recall when extracted protein-protein interactions were used to support other biological analyses and the problem with mediocre recall is resolved with large volumes of text.

Our results indicated that MedPost-MontyLingua Muscorian outperformed un-

modified-MontyLingua Muscorian in extracting directional genic interactions in terms of both precision and recall, suggesting that MedPost-MontyLingua Muscorian was more suited for this purpose. However, the precision of MedPost-MontyLingua Muscorian underperformed in extracting non-directional genic interactions, despite better recall. This suggested that errors in MontyTagger (un-modified-MontyLingua's POS tagger) resulted in more directional errors than that of MedPost. Given that our interest was in nondirectional interactions and precision was more important than recall in our case, un-modified-MontyLingua Muscorian was chosen for future work.

We conclude that our experimental results indicated that un-modified-MontyLingua Muscorian performed as well as MedPost-MontyLingua Muscorian for the purpose of processing biomedical text for the extraction of genic interactions. Thus, in contrary to the general assumption that generic text processing systems must be modified before being suitable for processing biological text for extracting genic interactions as evident from numerous systems to date, we presented a case study where comparable performance could be achieved by using generic text processing tools. This outcome is consistent with a previous study using un-modified MontyLingua for processing peer-reviewed economics papers (van Eck, 2005; van Eck and van Den Berg, 2005).

An initial evaluation of MontyTagger on MedTag Corpus (Smith et al., 2004) indicated 83.1% accuracy, which was considerably less than from MedPost's reported accuracy of 96.9% (Smith et al., 2004) and was close to the 83.0% tagging accuracy of a generic POS tagger on biomedical text (Tateisi and Tsujii, 2004). This result was expected as MontyTagger was not developed for biomedical text (Ling et al., 2007).

The POS tagging errors were expected to impact on performance of the entire text processing pipeline but this was not observed in our results. Instead, the precision of un-modified-MontyLingua Muscorian was comparable to that of MedPost-MontyLingua Muscorian on directional genic interactions (55.6% versus 56.8%) and un-modified-MontyLingua Muscorian outperformed MedPost-MontyLingua Muscorian on nondirectional genic interactions (86.1% versus 81.8%). Taken collectively the precision of both system and their respective POS tagging accuracies, seemed contradictory to general expectations as stated in the *Call for Papers for the Tenth Conference on Natural Language Processing 2006 (CoNLL-X)*.

An error analysis on MontyTagger was carried out in attempt to provide insight into resolving this contradiction. A likely hypothesis to explain why POS tagging errors did not derail the entire text processing pipeline was that the errors were nullified post-tagging. Text processing is used in Muscorian as a means to convert unstructured text into structured form for data mining - an extremely limited use of natural language processing compared to more complex uses, such as automated translation. As mentioned previously, POS tagging can be seen as a process of mapping potentially infinite number of words in the English language into a finite set of tags, based on their syntactic meanings. Shallow parsing, also known as chunking, can then be seen as a process which examines the sequence of tags and splits them into semantic phrases, of which verb phrase and noun phrase are of interest in this case. Given that MontyLingua's shallow parser parses the sequence of tags into 3 types of phrases (verb, noun, and adjectives), it is conceivable that a number of POS errors have no effect on shallow parsing.

Of the 182399 token in MedTag Corpus (Smith et al., 2004), 30736 were erroneously tagged by

MontyTagger (16.9% error) spreading over 40 tags. The top 6 most common tag errors accounted for 86.6% of the total errors and were chosen for further evaluation. In each of the 6 most abundant error tags, the top 95% of the errors were examined.

The effects of each type of errors, such as 'NN' wrongly tagged to 'NNP', were examined by analyzing the routines for shallow parsing which uses Regular Expressions. It was found that in 26630 of the examined POS tagging errors, 20928 (78.6% of 26630) had no effect on the chunking process and the remaining 5703 errors adversely affected shallow parsing, which might account for lower recall of un-modified-MontyLingua Muscorian as compared to MedPost-MontyLingua Muscorian.

Therefore, despite a low POS tagging accuracy of 83.1% by MontyTagger, more than three-quarters of the errors had no detrimental effect on chunking, suggesting a “functional POS tagging accuracy” of at least 94.6%, which was relatively close to MedPost's reported 97% accuracy (Smith et al., 2004). This apparent high “functional POS tagging performance” despite poor actual tagging accuracy might be the reason to explain un-modified-MontyLingua Muscorian's good performance in LLL05 test (Cussens and Nedellec, 2005) despite poor tagging accuracy compared to MedPost-MontyLingua Muscorian. This suggested that the nature of POS tagging errors might be more important than a single measure of POS tagging accuracy in a specific use of generic text processing tools where a shallow parser is involved. Therefore, it can be inferred that applications of biomedical literature analysis where a shallow parser is likely to be involved, such as extracting entity interactions and protein or molecule localization, POS tagging errors may not result in a decline in system performance.

At the same time, it is known that building domain-specific text processing tools requires much manual efforts (Jensen et al., 2006) suggesting that the cost and effort needed to train taggers specifically for biomedical text may not be needed, depending on the target application. However, it should also be cautioned that other applications or systems that do not involve shallow parser, such as Arizona Relation Parser (Daniel et al., 2004) which uses full sentence parsing, are likely to benefit from superior POS tagging accuracy of MedPost (Smith et al., 2004) and may experience degraded results from tagging errors.

MedTag Corpus (Smith et al., 2004) was used as a standard for evaluating MontyTagger. However, only 38 of the 45 tags in Penn Treebank Tag Set were used to annotate the corpus while the tagged output of MontyTagger illustrated the use of 45 tags. This might suggest inconsistencies or errors in MedPost Corpus, which were found in other POS tagged corpora (Peshkin and Savova, 2003; Ratnaparkhi, 1996).

5. Conclusions

In summary, analysis of the effects of MontyTagger's errors on downstream shallow parsing by MontyREChunker illustrated that 78.6% of the examined errors had no effect on shallow parsing. This implied that although the POS tagging accuracy of MontyTagger on MedPost Corpus was 83.1%, a majority of the errors had no downstream effect; thus, the functional POS tagging accuracy of MontyTagger was between 94.6% and 96.9%. A good functional POS tagging accuracy despite poor POS tagging accuracy, with respect to shallow parsing, is a likely reason for a comparative performance in extracting protein-protein interactions from text using a domain-specific or a generic POS tagger.

References

1. Abulaish M, Dey L: **Biological relation extraction and query answering from MEDLINE abstracts using ontology-based text mining.** *Data & Knowledge Engineering* 2007, 61(2):228.
2. Amrani A, Roche M, Kodratoff Y, Matte-Tailliez O: **Inductive Improvement of Part-of-Speech Tagging and Its Effect on a Terminology of Molecular Biology.** In: *18th Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2005: May 9-11, 2005* 2005; Victoria, Canada: Springer Berlin / Heidelberg; 2005: 366.
3. Brants T: **TnT - a statistical part-of-speech tagger.** In: *6th Applied Natural Language Processing Conference: 2000; Seattle, Washington, USA; 2000.*
4. Brill E: **Transformation-based error-driven learning and natural language processing: a case study in part of speech tagging.** *Computational Linguistics* 1995, 21(4):543-565.
5. Chen H, Sharp BM: **Content-rich biological network constructed by mining PubMed abstracts.** *BMC Bioinformatics* 2004, 5:147.
6. Cussens J, Nédellec C (eds.): **Proceedings of the 4th Learning Language in Logic Workshop (LLL05).** Bonn; 2005.
7. Daniel MM, Hsinchun C, Hua S, Byron BM: **Extracting gene pathway relations using a hybrid grammar: the Arizona Relation Parser.** *Bioinformatics* 2004, 20(18):3370.
8. David PAC, Bernard FB, William BL, David TJ: **BioRAT: extracting biological information from full-length papers.** *Bioinformatics* 2004, 20(17):3206.
9. Jang H, Lim J, Lim JH, Park SJ, Lee KC, Park SH: **Finding the evidence for protein-protein interactions from PubMed abstracts.** *Bioinformatics* 2006, 22(14):e220-226.
10. Jensen LJ, Saric J, Bork P: **Literature mining for the biologist: from information retrieval to biological discovery.** *Nature Review Genetics* 2006, 7(2):119-129.
11. Kodratoff Y, Dimulescu A, Amrani A: **Man-Machine Cooperation in Retrieving Knowledge from Technical Texts.** In: *AAAI 2005 Symposium on Mixed-Initiative Problem-Solving Assistants: 2005; 2005.*
12. Ling MH: **An Anthological Review of Research Utilizing MontyLingua, a Python-Based End-to-End Text Processor.** *The Python Papers* 2006, 1(1):5-12.
13. Ling MH, Lefevre C, Nicholas KR, Lin F: **Re-construction of Protein-Protein Interaction Pathways by Mining Subject-Verb-Objects Intermediates.** In: *Second IAPR Workshop on Pattern Recognition in Bioinformatics (PRIB 2007).* Singapore: Springer-Verlag; 2007.
14. Liu H, Singh P: **ConceptNet: A Practical Commonsense Reasoning Toolkit.** *BT Technology Journal* 2004, 22(4):211-226.
15. Marcus MP, Santorini B, Marcinkiewicz MA: **Building a large annotated corpus of English: the Penn Treebank.** *Computational Linguistics* 1993, 19(2):313-330.

16. Novichkova S, Egorov S, Daraselia N: **MedScan, a natural language processing engine for MEDLINE abstracts.** *Bioinformatics* 2003, 19:1699-1706.
17. Ratnaparkhi A: **A Maximum Entropy Model for Part-of-Speech Tagging.** In: *Conference on Empirical Methods in Natural Language Processing: 1996; University of Pennsylvania*: Association for Computational Linguistics; 1996: 133-142.
18. Peshkin L, Savova V: **Why Build Another Part-of-Speech Tagger? A Minimalist Approach.** In: *Recent Advances in Natural Language Processing (RANLP-2003): 2003; Borovets, Bulgaria*; 2003.
19. Santos C, Eggle D, States DJ: **Wnt pathway curation using automated natural language processing: combining statistical methods with partial and full parse for knowledge extraction.** *Bioinformatics* 2005, 21(8):1653-1658.
20. Smith L, Rindfleisch T, Wilbur WJ: **MedPost: a part-of-speech tagger for bioMedical text.** *Bioinformatics* 2004, 20(14):2320-2321.
21. Tateisi Y, Tsuji Ji: **Part-of-Speech Annotation of Biology Research Abstracts.** In: *4th International Conference on Language Resource and Evaluation (LREC2004): 2004; 2004: 1267-1270.*
22. van Eck NJ: **Towards automatic knowledge discovery from scientific literature.** *Masters*. Rotterdam: Erasmus University; 2005.
23. van Eck NJ, van den Berg J: **A novel algorithm for visualizing concept associations.** In: *16th International Workshop on Database and Expert System Applications (DEXA'05): 2005; 2005.*

Upcoming Events

The following events, taken from the python.org events wiki⁵⁵, are being held between December this year and March 2008.

April 7-11, 2008: Atlanta, GA, USA. Django Bootcamp at the Big Nerd Ranch. <http://bignerdranch.com/classes/django.shtml>

April 12-13, 2008: Poznan, Poland. RuPy 2008. <http://rupy.eu/>

May 5-7, 2008: San Francisco, CA, USA. (Comprehensive) Introduction to Python course given by Wesley Chun. <http://cyberwebconsulting.com/>

May 14-16, 2008: Longmont, CO, USA. Learning Python course taught by Mark Lutz. <http://home.earthlink.net/~python-training/longmont-public-classes.htm>

May 17-18, 2008: Paris, France. PyCon FR / Journees Python 2008. <http://fr.pycon.org/>

June 9-13, 2008: Atlanta, GA, USA. Python Bootcamp with David Beazley at the Big Nerd Ranch. <http://bignerdranch.com/classes/python.shtml>

July 7-12, 2008: Vilnius, Lithuania. EuroPython 2008.

<http://wiki.python.org/moin/EuroPython2008>

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.

⁵⁵ <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.