Python Magazine

APRIL 2008

Diving into the Gene Pool with BioPython

Learn about bioinformatics with Python

Portable Data with PyTables

Managing large datasets with PyTables

A Database in the Cloud

Google Spreadsheets with Python

Web-based data storage and retrieval with Python and the Google Data API

Scripting Xcode with Python

Take control of Your Favorite OS X IDE with Python

Inside:

- Tech Communities, and the Python Community
- PyCon 2008 in Review
- Web Frameworks, Python 3k, PyCon, and more!
- Lower the Complexity of GUI Programming with AVC

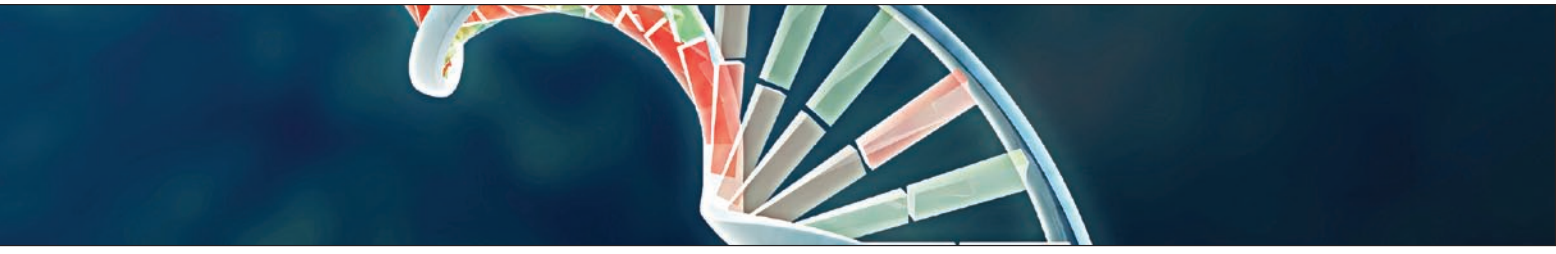
Python Magazine

CONTENTS

FEATURES

Download this month's code at:
http://code.pythonmagazine.com/2/4

COLUMNS



WRITE FOR US!

If you want to bring a Python-related topic to the attention of the professional Python community, whether it is personal research, company software, or anything else, why not write an article for Python Magazine? If you would like to contribute, contact us and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine. Visit www.pythonmagazine.com/c/p/write_for_us or contact our editorial team at editors@pythonmagazine.com and get started!

EDITORIAL

>>>import this

The world of technology looks, from the outside, like a massive sea of wires and code. I was initially intimidated by a move into the IT world, because I had no idea where to focus my energy. The result was that I focused my energy in a lot of different directions, and found that it's possible to earn an honest living without being pigeon-holed into some highly-specialized position, which I would only grow bored with.

Depending on the client and the needs of my full-time employer, I work as a database administrator, system administrator, infrastructure architect, project manager, network administrator, and/or developer. Within these various tech genres are subcultures devoted to the various technologies and paradigms of getting work done in a particular way or using particular tools, which I find fascinating. Within the system administration community are subcommunities devoted to OpenLDAP and Fedora Directory Server, Puppet and CFEngine, Apache and lighttpd, Squid and pound. Within the web development community there are groups devoted to .NET, PHP, Python, and Java. Within the PHP community are devotees of Smarty, Drupal, Joomla, and the various Nuke derivatives. Within the Python community are communities for Django and TurboGears, PyQT and wxPython, Wing IDE and SPE. The list goes on indefinitely.

Through my interactions with these communities and others, and especially in my work trying to support the Python community (through the creation of Python Magazine), PHP community (through past work as Editor in Chief of php|architect), and the Linux community (through various writings on a number of topics for a number of outlets), what I've witnessed is that not all communities are the same. They all have their own unique quirks and idiosyncrasies, and they all cope with their evolution in different ways.

After some time observing and interacting with the various communities, you can start to build a profile of a community's "personality". You can collect enough experiences with each community to start to see trends in their responses to different stimuli. How do they deal with disruptive change? How do they react to ideas that originate from competing communities? How do they dole out recognition? How do they communicate their purpose? Do they have a sense of humor? What is their propensity toward in-fighting? What percentage of a decision is weighted on technical merit as opposed to stature, reputation, or other non-technical factors? How do ideas move from the end user to the core developer? There are millions of observations anyone can make, and we probably make them subconsciously to some degree. I have a strong tendency to want to understand what makes a community tick, whether it's technical or not. I don't know why. It's a blessing and a curse. I call it a hobby. I like to people-watch, too.

So what makes the Python community special or different? I think one of the great strengths of Python that will fuel its continued success is the relative maturity and lack of religion-like or cult-like tendencies. This is not to say there are no Python zealots. But if you take the community's personality as a whole and try to envision a person who exhibits this personality, this person is probably not a zealot.

Python Magazine

APRIL 2008
Volume 2 - Issue 4

Publisher
Marco Tabini

Editor-in-Chief
Brian Jones

Technical Editor
Doug Hellmann

Contributing Editor
Patrick Bryant, Steve Holden

Columnist
Mark Mruss

Graphics & Layout
Arbi Arzoumani

Managing Editor
Emanuela Corso

Authors
Doug Hellmann, Eugen Wintersberger, JC Cruz, Jeff Scudder, Mark Mruss, Steve Holden, Zachary Voase

Python Magazine (ISSN 1913-6714) is published twelve times a year by Marco Tabini & Associates, Inc., 28 Bombay Ave., Toronto, ON M3H1B7, Canada.

For all enquiries, visit
http://pythonmagazine.com/c/p/contact_us

Printed in Canada

At a high level, Python users talk about ideas from other languages all the time. As a community, we are curious about technologies and languages that compete with our own, and acknowledge what they get *right* in addition to what's lacking. I personally know several Python developers who also work regularly with Java and C++. The ideas they gain from working with those languages are met with open arms in the Python community as compared to other communities, where I've witnessed responses like "well, then go use that language" on a more regular basis than I'd like to admit.

Diving slightly deeper, there is even quite a bit of cross-pollenation in the various sub-communities within the larger Python community. The Django community has brought in ideas from other web frameworks – even those in other languages. Heck, TurboGears and Pylons apparently took so many ideas from each other that they've made strides toward increasing the number of moving parts they have in common as a means of reducing duplication of effort, and accelerating the evolutionary process of both projects by working in unison on those parts. In other communities, this level of self-awareness and willingness to collaborate across competing projects is just about unheard of.

Traits like this, I believe, lead to a more cohesive overall community of developers, as opposed to a more fragmented model seen in some other areas of technology. I believe the existence of the Python Software Foundation also aids in supporting this cohesiveness by producing the only volunteer-run conference organized not by a local user group, but at a tier above local user groups, so that a single, known, recognizable conference, PyCon, gains one of the benefits of conferences run by some commercial entities: location-independence. PyCon is in Chicago this year, and has been held in Washington DC and Texas in the past. This location-independence insures that people in different parts of the country get to experience PyCon, where they learn things they probably never knew about Python, as well as the community. Face-to-face events strengthen the bonds of the community members, and do much to foster a more cohesive community.

Aside from evolutions in the language itself, how will the Python community evolve? It's interesting to speculate on. In the PHP world, a huge influx of new programmers and projects led to a somewhat interesting collection of projects and subcommunities, and (in my opinion) a greater distance between the core PHP developers and the users they're supposed to support. In the Linux world, the progress in the desktop environment and the excitement surrounding its progress seems to have hogged a lot of the development effort, which (again, my opinion) has slowed the progress of projects that cater to making *servers* easier to manage, administer, troubleshoot, and deploy. Another problem in the Linux server realm is a religion-like splintering of adherents to different distributions.

Of course, there are different Python distributions. There also seems to be a growing number of new developers choosing Python. The python-tutor mailing list and various IRC channels seem to do a pretty good job of offering a consistent message to new developers in terms of best practices as applied to their task du jour. Rather than seeing a random collection of blogs and external documentation links being thrown at new developers, a great many of the links point directly to python.org, which acts as a very authoritative common point of reference for the entire community. The basic ideals laid out in the Zen of Python are also consistently repeated and impressed upon those coming to Python new, or from different areas of development.

With all of these things considered, I believe the Python community is well-poised for growth, with little risk of things getting out of control or splitting into a million little pieces. It's quite refreshing to witness, and I'm proud to be a part of such a welcoming community supporting a fantastic language for newcomers and experienced developers alike.

BRIAN JONES is a system/network/database administrator who writes a good bit of Perl, PHP, and Python. He's the co-author of Linux Server Hacks, Volume Two from O'Reilly publishing, founder of Linuxlaboratory.org, contributing editor at Linux.com, and, in a past life, worked as Editor in Chief of php|architect Magazine. In his spare time, he enjoys brewing beer, playing guitar and piano, writing, cooking, and billiards.

AND NOW FOR SOMETHING COMPLETELY DIFFERENT

by Doug Hellmann

PyCon 2008 was held March 12-20 in Chicago. The explosive growth in attendance of the conference translated to an increase in energy and enthusiasm, from both attendees and organizers.

I haven't made it to a Python convention for several years, not since before they changed from being called *International Python Conference* to the community organized *PyCon*, so I was excited to have a chance to go to PyCon 2008 in Chicago last month. I wasn't the only one, either – turnout was over 1000 this year (quickly dubbed a *kiloPythonista*). According to the organizers, that is a 70% increase over last year's attendance. The growth trend is strong, so organizers are anticipating even more participation next year. Another statistic we were able to estimate based on registration is that 100 women attended the conference this year. While still lower than it should be, that number is up from 26 last year, and there has been a concerted effort to continue the upward trend to enhance the diversity of our community.

Tutorials

On the first day of PyCon attendees have the opportunity to participate in a wide variety of tutorial sessions. The topics this year included build tools, OLPC, ORMs, advanced Python programming tips, scientific computing, and a variety of web development frameworks. There were tutorials targeted at different levels of expertise, from new Python developers to those with years of experience. Most of the feedback I received

REQUIREMENTS

Useful/Related Links:

- US PyCon http://us.pycon.org
- PyCon UK http://uk.pycon.org
- PyCon Blog http://pycon.blogspot.com/
- Python 3.0 http://www.python.org/download/releases/3.0/
- PyCon Organizer List http://mail.python.org/mailman/listinfo/pycon-organizers
- PyCon 2010 Bid Information http://wiki.python.org/moin/PyConPlanning/BidRequirements

from tutorial attendees was positive, with the only complaints related to networking issues that occurred early in the day. Given that more people attended tutorials this year than *all of PyCon 2006*, that widespread enthusiasm is impressive.

Keynotes

In typical conference style, each morning started with keynote presentations. On Friday morning Chris Hagner of White Oak Technologies, Inc. told us about some of the problems they have encountered since selecting Python as their primary development platform. For example, in response to the complaint that "Not enough developers know Python", he answered that "No one learns Python 'by accident'", and those that do tend to be better developers who push themselves to learn new tools. Although few of their customers already know Python, they took that as an opportunity to build trust while teaching them about the technology. In all cases White Oak was able to convert the challenges into opportunities or competitive advantages. These types of success stories bode well for other Python consulting shops.

Guido's keynote, also on Friday, covered Python 3.0. The take away quote is: "It's coming. Don't worry. It's going to change everything. *Don't worry*." You can find the material he presented about the changes in 3.0 and everything being done to make the upgrade path as smooth as possible on python.org, so I won't repeat it here.

Saturday's keynote from Van Lindberg was an excellent discussion of intellectual property issues and how they relate to open source or free software licenses. I've been working with open source licenses of one form or another for several years now, but his description of the difference between rivalrous and excludable goods, and solutions for handling the "tragedy of the commons" and "free rider" problems clarified in my own mind the reasons why using open source licenses can be challenging in the short term but rewarding over the long term.

PyCon 08: Guido van Rossum during his keynote

On Sunday morning, Mark Hammond discussed how Mozilla uses Python, including the news that version 1.9 includes some support for scripting Mozilla with Python. He also gave several examples of syntax and language constructs being lifted from Python and adopted in JavaScript.

Later, Ivan Krstić gave an update on the One-Laptop-per-Child project, including some uplifting stories of his work with kids in Peru and Uruguay during the first OLPC deployments. Ivan has written extensively about the trips on his blog, and the details of the OLPC project he has shared are inspiring (http://radian.org/notebook/first-deployment).

Scheduled Talks

I tried to attend as many of the scheduled talks as possible, but I can't possibly write reviews of all of them in the space I have here. Each of the sessions was digitally recorded, and the volunteers doing the A/V work put in heroic efforts to transfer the digital tapes to files that eventually will be available online. Watch the PyCon blog for announcements; the videos will be posted as they are converted.

One of the more interesting sessions I attended was Brett Cannon's talk entitled *How Import Does Its Thing* – a deep-dive examination of the intricacies of Python's module and package re-use mechanism.

Later that morning Jeff Rush presented tips and tricks for managing and growing Python user groups. In addition to Jeff's ideas, the questions and suggestions from the audience afterwards showed the size and diversity of the Python community. There seem to be vibrant, active user groups all *over* the place, each with their own unique personalities. Some groups are heavily populated by students, while others are (ahem) older. Each type of group has its own challenges, so sharing knowledge about what works for some may help others.

Matt Harrison talked about managing code complexity, including how to use metrics to ensure you're measuring something useful. His diagrams and explanation of cyclomatic complexity gave me an entirely new perspective on the way I plan my own testing. One of the GHOP projects this year was a cyclomatic complexity analyzer for Python code. It is open source, but not necessarily packaged for easy release, yet.

Another session worth mentioning was given by the folks at Resolver. Their Resolver One spreadsheet/Python interpreter mash-up feels like what we've always wanted a spreadsheet to be – a full-blown programming environment with charts and graphs. It will be interesting to see how the product does, and whether non-programmers are ready for the power and flexibility afforded to them by exposing the Python interpreter. I'd like to give it a try, but it uses IronPython and only runs on Windows.

Jason Pellerin's talk on nosetest illustrated quite a few features that make it a powerful tool for running tests. The support for module and package-level fixtures using generators for data-driven test creation especially caught my eye. One of the issues we have with our test suite at work is the length of time it takes to run, mostly due to the expensive fixtures. Making it easier for a group of tests to use the same fixtures could help us cut down on the repetitive operations.

Titus Brown's presentation on integrating code coverage analysis and UI test automation tools with the OLPC Sunday afternoon was intriguing. His approach to instrumenting a remote process to collect code coverage data, then collecting it and presenting the results in a UI shows great promise for testing and debugging distributed apps. Titus is well known for his work with testing tools and techniques, and never fails to deliver an entertaining and informative talk.

Open Spaces

The scheduled talks are a good source of information about what's going on in the community, but the *hallway track* this year was one of the best parts of the weekend. The less formal nature of the Birds-Of-a-Feather (BOF) meetings and other open space sessions resulted in more collegial and participatory conversations. Some, such as Steve Holden's *Teach me Twisted* were informative and hilarious at the same time. But the sessions weren't all work related. There were spaces dedicated to board games, online gaming tournaments, and plenty of good food and drink. Meal times were an excellent opportunity to get to know the other attendees in small group conversations.

Sprints

The increase in overall attendance carried over to an estimated 40% increase in sprint attendance. Unfortunately, I had to leave while the sprints were still being organized, but there was a large crowd entering the hall for the pre-sprint tutorials as I was leaving for the airport. The participating projects included Django, TurboGears/Pylons, Python core, and many others.

Advice for Attending Next Year's Conference

Here are a few tips if you plan to attend PyCon next year, in no particular order:

Obviously, *do* register early to take advantage of the discount rates and make sure you are able to reserve a room in the conference hotel. Space is limited, and the rooms fill up quickly.

Once you're there, *don't* get too wrapped up in the scheduled sessions. Make sure to check the open space schedule a couple of times a day to see what spur-of-the-moment events are planned.

For goodness sake, *don't* sit in the corner. Mingle! Better yet, pitch in and help out. I arrived on Thursday evening, but hadn't signed up for tutorials due to my travel time constraints. So after I registered for the conference I found myself with a big block of free time. The guys handing out tote bags and t-shirts, herded by Don Spaulding, accepted my offer of help and over the course of Thursday evening and Friday morning I was able to interact, at least in a limited fashion, with a large number of the conference attendees.

If you're not interested in a session, *don't* sit in the back of the room reading your email. Give up the seat for someone who *is* interested, and go to the lounge area to mingle or use your laptop there. Sitting through a talk you're not listening to just wastes your time.

Get up a little earlier than you normally do to snag the best selection of breakfast pastries (or don't, and leave

PyCon 08: Day 1

them for me). Food quantity and quality were good this year, so missing out isn't really an issue, but coming downstairs early also gives you a head start on the day. You'll be fresh and ready for the early sessions. Breakfast is another good time to share a table with someone you don't know and find out about interesting projects going on in the community. For example, I learned about a project Dr. Johnny Stovall is doing in Indonesia using Python to build an application to teach tribal people the national language by showing them words in both their tribal language and the national language at the same time.

Oh, and leave space in your suitcase for the schwag and t-shirts. The vendors and sponsors this year were especially generous with the goodies.

Changes I Would Like to See

The conference went smoothly, but nothing is perfect and there is room to improve PyCon, too. Luckily, since it is an volunteer-run event, so *we* can change it.

One point that has been brought up is the commercial nature of some of the keynotes and lightning talks. It has been discussed rather extensively online, so I won't belabor the point here. I will say that balancing the content of a sponsored presentation can be tough, and some worked out better than others. But balance *can* be achieved, and sponsorship is *important* to controlling costs, so we as a community need to work that out. Presentations about successful commercial apps written in Python that talk about the technology involved, development "war stories", deployment, and other techniques that can be shared without giving away the "secret sauce" all can be useful and interesting from an advocacy and technology perspective. Not all projects are, or need to be, open source.

The layout of the conference hall had the open spaces rooms downstairs from the "main" conference. A few of the BOFs and other meetings I went to were well attended, but placing more emphasis on announcements about what was going on downstairs would have helped make it feel less like a sub-conference. Some sort of webcam system to project the schedule upstairs might have helped with that, but I don't know how realistic such an idea is. I'm sure someone out there could come up with a more creative solution.

The primary complaints I heard in the hallways this year related to the wireless networking in the conference area. There were glitches early on in the first day or two, but I never had any trouble myself so I'm not sure how bad they really were. Less complaining and more pitching in should solve that problem, too.

Participating as an Organizer

Speaking of helping out, you really should consider getting involved. The conference will be in Chicago again next year, though the exact venue and setup has not been finalized. The best way to have input into those decisions and otherwise keep up with what is going on is to join the pycon-organizers mailing list. See the PyCon web site for details.

The bid process for PyCon 2010 was launched during a meeting on Saturday during lunch. There are several user groups pitching their cities to host the conference. One of the most important selection criteria is the enthusiasm and energy of the local organizing committee, so if you want the convention in your home town *you need to be involved*.

PyCon 08: Vendor room

DOUG HELLMANN is a Senior Software Engineer at Racemi. He has been programming in Python since version 1.4 on a variety of Unix and non-Unix platforms. He has worked on projects ranging from mapping to medical news publishing, with a little banking thrown in for good measure.

Diving into the Gene Pool with BioPython

by Zachary Voase

Bioinformatics is on the rise in the world of science. More and more computer scientists have begun to gravitate towards this exciting field, and as tools and libraries such as BioPython evolve, this number will only increase. In this article I introduce developers and dabblers who are familiar with Python to the biology powering this exciting subject, and the utilities available for Pythonistas to work with biological data.

Introduction

DNA: The blueprint of life. Surely an oversimplified description of such a complex and important molecule, but a necessity nonetheless. What most people overlook is the effort that goes into determining, analyzing and using this blueprint to gain even more information about the world, both around us and within us. The creation of phylogenetic trees, for example, is one pursuit of the field of bioinformatics. These trees show, with a high probability, how organisms evolved from the thriving soup of the Earth into amazingly intricate beings, and expose genetic relationships between different organisms or proteins.

Let's begin with the basics. The DNA molecule is essentially a string of small units known as *nucleotides*, held together by a sugar-phosphate backbone. Nucleotides have attached to them chemical groups known as *bases* that distinguish the different types of nucleotide. In each of the nucleotides that make up DNA, the base can be one of four possibilities: Adenine (A), Guanine (G), Thymine (T) or Cytosine (C) (see Figure 1). So, we can look at DNA as being a string of information in base-4 notation. Figure 2 shows a strand of nucleotides with the sugar-phosphate backbone in grey and the bases in

REQUIREMENTS

PYTHON: *2.3+*

Other Software:

- BioPython v1.4.4
- C compiler (if compiling BioPython from source)
- mxTextTools v2.0
- Numerical Python v24.2

Useful/Related Links:

- AMY1.fa - http://tinyurl.com/2cgzrg
- AMY1.gb - http://tinyurl.com/yppgow
- BioPython Home Page - http://biopython.org
- SwissProt Protein Database - http://www.expasy.ch/sprot/
- FASTA Format Wikipedia Entry - http://en.wikipedia.org/wiki/FASTA_format
- PubMed Home Page - http://www.pubmed.gov/
- BioPython SeqIO Wiki Page - http://www.biopython.org/wiki/SeqIO

color. The strands of bases join up along their length in order to make a complete molecule of DNA, in the famous double-helix structure. Given one strand, the second strand's composition is predictable because A will only join up with T, and C with G. The name given to this other strand is the *complementary strand* or sequence. Figure 3 illustrates a strand of DNA along with its complementary strand. There is no real chemical difference between the two strands; the strand on the left is complementary of that on the right, and vice versa. The difference comes in the interpretation of the information held by the DNA. Some information is encoded in a strand-dependent way, and other information is not; the factors affecting this are too numerous and complex to describe here.

FIGURE 1

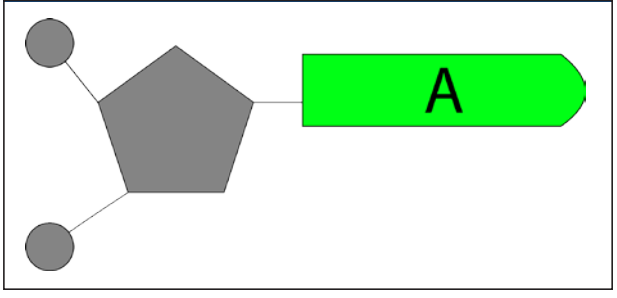
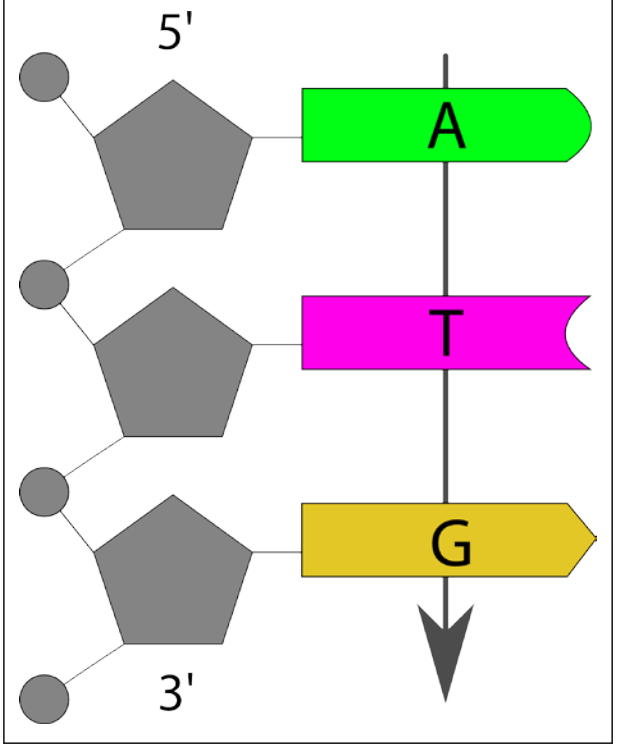


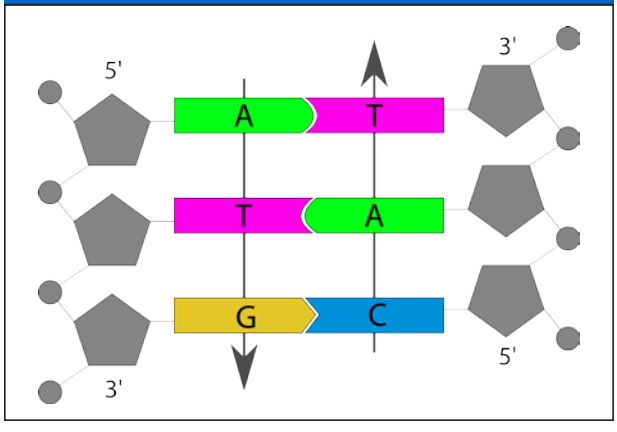
FIGURE 2



Another feature of the DNA molecule is its directionality. The structure of the molecule is such that a strand has two chemically distinct endpoints that tell the DNA-reading machinery in the cell which end to start from. These two ends are called 5′ and 3′ (pronounced "five prime" and "three prime"), and conventionally DNA sequences are written from 5′ to 3′. In Figures 2 and 3 the arrows run in the direction the sequences are read and written. The complementary strands run in opposite directions, so in order to represent data in a conventional manner it is necessary to give complementary sequences in reverse order, the so- called *reverse complement*. This can be seen in Figure 3 also; the strands running opposite to each other run in different directions, because their 5′ and 3′ ends have been swapped.

Essentially, DNA provides a code for creating proteins, the machinery of our cells. There are other regions of DNA which perform several other complex and, at the moment, not entirely understood functions, but the main concern of this article is in the protein-coding regions. This code is understood by each and every cell in our body. Different sequences of three bases, known as *codons*, correspond to one of 20 different chemical subunits called *amino acids*. The amino acids string together to form *proteins*. A long sequence of DNA, therefore, codes for many of these units. The units are strung together to form the more complex molecules making up proteins via a process called *translation*. During translation a sequence of codons is literally translated into a sequence of amino acid subunits. These proteins carry out specific functions in our cells. Some, called *enzymes*, allow particular chemical reactions to occur in our cells. Others, known as *hormones*, send messages to other cells in our body. Still others perform one of a host of other unique abilities.

FIGURE 3



That's enough background material on DNA coding; it's time to get back into the Python coding.

"DNA provides a code for creating proteins; the machinery of our cells."

Basics of Sequences

Let's look at how easy it is to manipulate a sequence of nucleotides. Initialize your environment by importing the Seq class, then create a sequence by instantiating a Seq with a sequence as the only argument.

```
>>> from Bio.Seq import Seq
>>> sequence = Seq('AAACAACTTCGTAAGTAAGTATA')
>>> print sequence
Seq('AAACAACTTCGTAAGTAAGTATA',
SingleLetterAlphabet())
```

The reverse complement of a sequence can be found by calling `reverse_complement()`.

```
>>> print sequence.reverse_complement()
Seq('TATACTTACTTACGAAGTTGTTT',
SingleLetterAlphabet())
```

To reiterate, the reverse complement is a representation of the complement as it would be read from the 5' to 3' end, due to the opposite directionality of the complementary strand of DNA.

Alphabets

The problem with just using simple Seq objects as a way of storing sequences is that there is no meaning to the data. One way of adding metadata to a sequence is to associate a particular *alphabet* with it. The alphabet specifies which characters should be present in a sequence, and protects it from several illegal operations which we will see later. A lot of very commonly used alphabets are held in `Bio.Alphabet`, with the ones we want located in `Bio.Alphabet.IUPAC`.

```
>>> from Bio.Alphabet import IUPAC
>>> print IUPAC.unambiguous_dna
IUPACUnambiguousDNA()
>>> print IUPAC.unambiguous_dna.letters
GATC
```

From this, it's clear that `unambiguous_dna` is an instance of the `IUPACUnambiguousDNA` class, which is held inside the same module. It has a set of letters, which represent the valid characters forming a sequence of unambiguous DNA. There is also an alphabet for ambiguous DNA, in which there are several more letters to represent ambiguity in bases – for example, 'R' indicates that a base may be either 'A' or 'G', and 'Y' represents 'T' or 'C'. The whole specification may be found at the Wikipedia entry for the FASTA format. To add an alphabet to a sequence, simply give the alphabet instance as the second positional argument when initializing the sequence. Alternatively, this may be done by changing the `alphabet` attribute of the sequence object.

```
>>> sequence.alphabet = IUPAC.unambiguous_dna
>>> print sequence
Seq('AAACAACTTCGTAAGTAAGTATA',
IUPACUnambiguousDNA())
```

Having access to the alphabet when working with a lot of sequences helps because you can quickly see what sequence is what.

Using Sequences with Alphabets

The Seq object is very versatile. It supports slices, concatenation, and the majority of operations which may be performed on strings. Using the usual slicing syntax, and the sequence object we created previously, let's cut out the first five characters of the sequence.

```
>>> sub_seq1 = sequence[:5]
>>> print sub_seq1
Seq('AAACA', IUPACUnambiguousDNA())
```

The alphabet of the parent sequence is preserved in the newly created sequence. Create another sub-sequence of the last five characters, and stick the two together.

```
>>> sub_seq2 = sequence[-5:]
>>> new_seq = sub_seq1 + sub_seq2
>>> print new_seq
Seq('AAACAGTATA', IUPACUnambiguousDNA())
```

Here, we've taken two sequences and combined them to make a new one, using the concatenation operator, just as we would with a string. The alphabet for the new object is inherited from the original sequences again.

Alphabets also act to validate sequence data and can help prevent illegal operations between incompatible molecules. For example, RNA molecules are similar to DNA molecules except that they do not form double

stranded molecules normally, and that the grey section in Figure 1 varies between the two. Another major difference is that Thymine (T) on the DNA molecule is a counterpart to Uracil (U) on the RNA molecule. Due to their differing structures the two are incompatible with one another; you cannot have a mixed strand of DNA and RNA. We can make a new sequence using the `IUPAC.unambiguous_rna` alphabet, the alphabet for sequences of RNA, and try to concatenate the two.

```
>>> rna_seq = Seq('GUAAGUAUA',
... IUPAC.unambiguous_rna)
>>> print rna_seq
Seq('GUAAGUAUA', IUPACUnambiguousRNA())
>>> new_seq + rna_seq
...
<type 'exceptions.TypeError'>: ...
```

An error message is returned: it is impossible to concatenate a DNA sequence with an RNA sequence. In this way, using alphabets helps to avoid errors which may arise from trying to do something which is physically impossible.

Mutable Sequences

Sequences are like primitive strings or lists in that they allow slicing and access to particular positions along the sequence. On the other hand, sequences do not allow you to replace slices or give positions within the sequence different values; in fact, they don't actually have a `__setitem__()` method.

```
>>> sequence[5] = 'G'
Traceback (most recent call last):
...
AttributeError: Seq instance has no attribute
'__setitem__'
```

Objects like this are said to be *immutable*. So, if you want to make changes to slices of a sequence, it is necessary to change the sequence so that it becomes mutable, using BioPython's `MutableSeq` class. `Seq` instances can be transformed into `MutableSeq` instances by calling their `tomutable()` method.

```
>>> mut_seq = sequence.tomutable()
>>> print mut_seq
MutableSeq('AAACAACTTCGTAAGTAAGTATA',
IUPACUnambiguousDNA())
```

Using a mutable sequence, you can reassign whole slices of a sequence, or just single characters, and the newly-changed mutable sequence can be made immutable again by calling `toseq()`.

```
>>> mut_seq[:5] = 'TCAGG'
>>> print mut_seq
MutableSeq('TCAGGACTTCGTAAGTAAGTATA',
IUPACUnambiguousDNA())
>>> changed_seq = mut_seq.toseq()
>>> print changed_seq
Seq('TCAGGACTTCGTAAGTAAGTATA',
IUPACUnambiguousDNA())
```

The functionality offered by the `MutableSeq` class is very useful, as it allows sequences to be modified in order to carry out mutations, as would happen in the body.

Reading Sequences From Files

Of course, all of this manipulation of genetic sequences would be useless if you couldn't store your data after manipulating it. One of the most popular sequence storage formats is FASTA, which was originally developed for use with the FASTA sequence alignment algorithm (a method for finding similarity in sequences). It's a very simple format, essentially consisting of a line beginning with a > character, followed by information about a sequence, and then several lines containing the sequence itself. This allows many sequences to be placed in the same file, and it is trivial to implement a parser and writer for this format. BioPython contains tools for working with FASTA files.

We can begin by getting an exemplar sequence file from the internet. In the related links for this article, I have provided a TinyURL version of a link to a query on a public database called PubMed. Download that file and save it as `AMY1.fa` so you can use it with the following examples.

As an example, we'll write this sequence's reverse complement to an output file. The first step is to import the necessary BioPython resources. `Bio.SeqIO` contains even more tools to read and write a variety of formats, but I'm only going to demonstrate FASTA now. The `IUPAC` module contains needed alphabets, and the `SeqRecord` class will be used later to write the sequence to a file.

```
>>> from Bio.SeqIO import FastaIO
>>> from Bio.Alphabet import IUPAC
>>> from Bio.SeqRecord import SeqRecord
```

The generator function `FastaIO.FastaIterator()` reads the sequence data. BioPython makes extensive use of generators, because they allow procedures to be carried out on streams and provide more intuitive interfaces. We can assign an alphabet to the data returned by this generator by specifying an `alphabet` keyword argument. Invoke the generator using the sequence file as the argument and specifying an alphabet.

```
>>> seq_file = open('AMY1.fa')
>>> seq_reader = FastaIO.FastaIterator(seq_file,
...     alphabet=IUPAC.unambiguous_dna)
```

Records are returned by successive calls to the generator's `next()` method. Get the first (and only) record from the file, and close the file.

```
>>> seq_record = seq_reader.next()
>>> seq_file.close()
```

The returned objects are instances of `Bio.SeqRecord.SeqRecord`. In addition to the sequence itself, sequence records contain metadata about a particular sequence, as parsed by the FASTA reader, including the name, id, and description of the sequence. In the case of the FASTA format, the name and id are the same when read. Other formats provide different names and ids, with ids primarily being used for accessing records in large databases.

```
>>> print seq_record.__class__
Bio.SeqRecord.SeqRecord
```

Writing Sequences To Files

The sequence held in a `SeqRecord` instance is available through its `seq` attribute. We can now get the reverse complement by extracting `seq` from the original record, finding its reverse complement, and assigning it to a new variable.

```
>>> seq = seq_record.seq
>>> seq_rc = seq.reverse_complement()
```

In order to write sequences out, you need to package them in a `SeqRecord` object. `SeqRecord.__init__()` accepts the sequence as the first positional argument, with a host of other keyword arguments usually used for more complex things such as database cross-references and annotations to the sequence. Sequence records to be written to FASTA-formatted files only need `name`, `id` and `description`. Here I've created some variables to hold the information before creating the record, in the interest of clean code.

```
>>> name = seq_record.name
>>> id = seq_record.id
>>> description = seq_record.description + \
...     ' (Reverse Complement)'
>>> seq_record_rc = SeqRecord(seq_rc,
...     id=id, name=name,
...     description=description)
```

In addition to passing them to the constructor, you can assign these attributes to the `SeqRecord` object after instantiating the object, like this:

```
>>> seq_record_rc.name = name
>>> seq_record_rc.id = id
>>> seq_record_rc.description = description
```

The next step is to write this information in the FASTA format using `FastaIO.FastaWriter`. It accepts a file-like object as its only positional argument, and has a `write_file()` method which takes a list of sequence records and writes them out in the FASTA format. Create a writable output file called 'AMY1_RC.fa' to which we will write both the original sequence and its reverse complement.

```
>>> seq_file_rc = open('AMY1_RC.fa','w')
```

Instantiate the `FastaWriter` instance around the output file like this:

```
>>> seq_writer = FastaIO.FastaWriter(seq_file_rc)
>>> print seq_writer.handle is seq_file_rc
True
```

The file object is held in the writer instance and can be accessed from the `handle` attribute, meaning there is usually no need to use an intermediate `seq_file_rc` file instance. This applies to the reader, too.

The `write_file()` method accepts an iterable containing several records and writes them all out to the file. It can only be called once for a writer instance. Write out the original sequence too, for the purpose of demonstration.

```
>>> seq_writer.write_file([seq_record,
                           seq_record_rc])
```

You now have a new FASTA-formatted file containing both the original and reverse complement of the genetic sequence of the human alpha-amylase 1 enzyme.

Proteins

The last Biology section touched on proteins lightly, but it is relevant to go slightly more in-depth now before doing more work with them.

Proteins, like DNA, are strands of units which are each different but link together in a similar fashion. The units of proteins are known as *amino acids*, and several of these join end-to-end to form a large molecule known as a *polypeptide*. Often, proteins will be made up of only one polypeptide, but some are composed of several polypeptides that combine in a very specific way. Haemoglobin, the pigment which allows our blood to transport oxygen, is one such protein; it is composed of four polypeptides known as *haem groups*, with an atom of iron trapped between the molecular entanglement.

DNA has many functions in our cells. In this article, we are concentrating on how it acts as a blueprint for the creation of proteins. As mentioned before, there is a code which maps 3-base-long codons of DNA onto particular amino acids in a process known as *translation*. Translation is carried out by a structure in cells called the *ribosome*, which joins this string of amino acids up to form a polypeptide. Because there are 64 possible codons and only 20 amino acids, the code is called *degenerate*, as several different codons will code for the same amino acid. Nevertheless, due to the structure of polypeptides it is possible to write them as sequences, albeit using 20 letters instead of 4. If you take a look at the `Bio.Alphabet.IUPAC` module, you will notice an alphabet named `protein`. This is the alphabet of amino acids, the specification for which may also be found at the FASTA format Wikipedia entry. It is also worth noting that because polypeptides do not join up with another complementary molecule, the concepts of complements and reverse complements do not apply. If you try calling the `reverse_complement()` method of a protein sequence, you will receive an error.

Querying From Public Databases

Due to the rapid expansion of the field of bioinformatics, and the necessity for information interchange between research parties around the world, several resources have been created which contain quickly growing repositories of information for bioinformaticists. PubMed, mentioned earlier, is a database of articles, genes, whole genomes (the gene sequences of entire organisms), and protein sequences run by the US National Center for Biotechnology Information. Swiss-Prot, part of the Swiss Institute of Bioinformatics' ExPASy server, is a publicly available database of protein structures and information, complete with large amounts of annotation and references. BioPython offers ways to query these (and other) resources from within your Python programs, using several modules included with the main distribution. In this section, we're going to grab a protein sequence from the Swiss-Prot protein knowledgebase.

The first step is to import `Bio.ExPASy` and `SwissIterator`.

```
>>> from Bio import ExPASy
>>> from Bio.SeqIO.SwissIO import SwissIterator
```

Then, using `ExPASy.get_sprot_raw()`, create a handle, that returns the data fetched from Swiss-Prot on the protein specified by the identifier 'P04745', which is essentially the index of a record in the database.

```
>>> connection = ExPASy.get_sprot_raw('P04745')
```

Instantiate the `SwissIterator` class around this connection. Because the connection acts like a file object, it is perfectly safe to iterate over it, instead of reading data into a file or StringIO buffer and then using that. The SwissIterator is a generator function which returns `SeqRecord` objects.

```
>>> sprot_reader = SwissIterator(connection)
>>> sprot_record = sprot_reader.next()
>>> connection.close()
```

Due to the large amount of annotative data held in the Swiss-Prot database, this sequence record will have its other possible attributes filled; `annotations`, `dbxrefs` and `features`, for example, will all contain a lot of information.

```
>>> print sprot_record.name
AMY1_HUMAN
>>> print sprot_record.annotations['organism']
Homo Sapiens (Human)
```

Protein P04745's name is 'AMY1_HUMAN', and it comes from the 'Homo Sapiens' organism, otherwise known as the Human. ExPASy's entries have a lot of metadata bundled with them, including references to papers which talk about the protein, and the location of the protein on other databases. You have successfully downloaded and parsed the polypeptide sequence for the alpha-amylase 1 enzyme, the genetic sequence of which we just worked with a short while ago.

Proteins are not simply mapped from the gene to the polypeptide sequence. Before being translated, DNA is copied onto a strand of RNA in a process called 'transcription'. This *messenger RNA* (mRNA) then moves from the nucleus of the cell, where the DNA is stored, out into the cell's cytoplasm. There a process called *splicing* occurs. Splicing is essentially the removal of particular fragments of code from the sequence, which has the effect of changing the polypeptide sequence produced. The fragments which remain, known as *exons*, join together to form a new sequence, and those that were removed, called *introns*, move off and are recycled by the cell. The spliced mRNA strand is used as the template to make the polypeptide.

Splicing is still not fully understood. Biologists spend a lot of time and effort figuring out what parts of a gene are spliced, and why. It is useful to have the gene and the protein sequences available at the same time because by using the protein sequence it is possible to

figure out what parts of the gene were spliced out, and hence which parts of the gene are introns and which are exons. Splicing is responsible for the large size difference between the human genome (the collective set of all human genes) and the human proteome (the set of all human proteins). This difference in size is because it is possible for one gene to code for several different polypeptides by alternative splicing. New evidence also suggests that some segments of RNA have a catalytic action which causes them to splice themselves out of the mRNA strand, without external help from proteins. The rapid advances occurring in this field every day, combined with its incredible and exciting complexity, are what is making it so attractive to computer scientists around the world.

Data Included With BioPython

In addition to a large collection of procedures, BioPython comes with a rich set of useful data. The `Bio.Data` module includes data such as the molecular weights of bases for both DNA and RNA and the codon mappings used by several different organisms to translate mRNA into polypeptide sequences. The specification for ambiguous DNA is included in the dictionary `Bio.Data.IUPACData.ambiguous_dna_values`. Individual character keys are mapped onto strings of several letters. For example, 'R' is mapped onto 'AG' and 'Y' onto 'CT', with a whole host of other ambiguous letters.

As an example of the other features of `Bio.Data`, we're going to write a small application to calculate the molecular weight of a specific protein, given its SwissProt identifier. The program will have to download a SwissProt entry, parse it, and then use the molecular weight data included in BioPython to calculate the weight of the protein. For this, it will need the `Bio.ExPASy.get_sprot_raw()`, `Bio.SeqIO.SwissIO.SwissIterator` and `Bio.Data.IUPACData.protein_weights` respectively. Listing 1 shows the whole program, lines 3-7 of which contain the necessary imports.

A good starting point is the small function to download the sequence record for a specified protein identifier (lines 9-14). It is merely a repetition of the procedure used earlier for the alpha-amylase 1 enzyme. It accepts a protein ID, connects to SwissProt using `get_sprot_raw()`, and uses `SwissIterator()` to grab the sequence record. It then closes the connection and returns the record. The reason I chose not to make it return only the sequence, which is really all that is needed, is so this function can be used again by other programs. Next, the program needs another function to calculate the weight of a protein given a sequence (lines 16-18). Because `calc_weight()` only requires that its argument implements iteration, it can accept strings, unicode objects, `Seq`, and `MutableSeq` instances. The protein letter-to-weight mappings are held in a simple dictionary, called `protein_weights`, located in `Bio.Data.IUPACData`. This dictionary allows `calc_weight()` to simply map the weight of each amino acid to a list, and then return the sum of this list. Again, fragmenting this rather simple program up into several functions enables other programs to use various parts of this one.

Finally, the wrapper function `id_to_weight()` cements it all together (lines 20-23). If `id_to_weight()` is fed the protein identifier for alpha- amylase 1, 'P04745', it gives back a floating point number of approximately equal to 66955.810. I've also added a small section which will allow the function to be run from the command line, and called with several identifiers. As you can see, BioPython comes with a real treasure trove of data, and I recommend you have a look around at the `Bio.Data` module in the interpreter and on-line documentation as much as you can to see what can be done with it.

LISTING 1

```
1. #!/usr/bin/env python
2.
3. import sys
4.
5. from Bio.Data.IUPACData import protein_weights
6. from Bio.ExPASy import get_sprot_raw
7. from Bio.SeqIO.SwissIO import SwissIterator
8.
9. def get_prot_record(prot_id):
10.     connection = get_sprot_raw(prot_id)
11.     reader = SwissIterator(connection)
12.     prot_record = reader.next()
13.     connection.close()
14.     return prot_record
15.
16. def calc_weight(prot_seq):
17.     weight_list = map(protein_weights.get, prot_seq)
18.     return sum(weight_list)
19.
20. def id_to_weight(prot_id):
21.     prot_record = get_prot_record(prot_id)
22.     weight = calc_weight(prot_record.seq)
23.     return weight
24.
25. if __name__ == '__main__':
26.     args = sys.argv[1:]
27.     for arg in args:
28.         print '%s: %.3f' % (arg, id_to_weight(arg))
29.
```

Sequence Features

The sequences we have examined so far have all been pretty simple; just strings of letters, with optional metadata if wrapped in a SeqRecord instance. But, of course, different segments of sequences often have different functions or roles. Sometimes you will have a sequence of pre-mRNA (i.e. mRNA that has not yet been spliced) and you will want to know which parts are introns and which parts are exons, etc. It is easy to do this with BioPython's SeqFeature module. Sequence Features are annotations to sequences which can optionally span a segment of the sequence. In this section we will use sequence features to separate out a DNA sequence into several different parts. Each of these parts will correspond to an exon on the DNA sequence. For us to do this, the FASTA format will not be enough; we're going to have to use a format known as GenBank. This format allows for verbose detail and annotation of a sequence, and BioPython can parse the GenBank format's annotations into sequence features. I've made available a link to the data we will need. Download the AMY1.gb file as you did for the FASTA-formatted sequence, and make sure to save it under that name.

> " BioPython offers ways to query these resources from within your Python programs."

Listing 2 starts by importing the necessary modules and functions. In this case, we need to import only two functions that are held in the Bio.SeqIO module: parse() and write(). These functions offer a general I/O ability, with the type of file specified as the second positional argument. For example, on line 4 we open the file using the 'genbank' format string, meaning it will read this as a GenBank file. Other file types include 'fasta' (the file format of which should be obvious) and 'swiss' (for SwissProt files). The list of supported file formats, along with their capabilities (with respect to reading and writing) can be found at the BioPython wiki page for the SeqIO module.

Once the file is open, we grab the only record in the file, and close the file. Lines 7-8 extract all the exon data from each feature in the sequence record.

SeqRecord instances contain a features attribute which is a list of SeqFeature instances. Each feature has a type attribute; for example, a type of 'gene' means that the feature contains information on where that particular sequence can be found within an organism's genome. Features of type 'exon' contain information on a particular exon within that sequence, including the exon number for that exon within that gene and its location, given as a range of positions along the sequence. The location attribute of the first exon shows that it is made up of the bases between positions 0 and 168 along that sequence, with 0 being the first base (as with all iterables in Python).

The next step is to create a list of the sequence records cut out of the master sequence so we can write them to a file afterwards. This list is called seq_records. The loop processes the exons. First, it gets the start and end positions by accessing its location attribute. Then, it retrieves the exon's number, or index on the strand, from the 'qualifier' with title 'number'. The first item in the list is used because the values are held in lists so keys can be specified several times for a particular feature. Qualifiers are essentially key/value pairs of metadata associated with features, which mean you can specify your

LISTING 2

```
1. from Bio.SeqIO import parse, write
2.
3. gb_file = open('AMY1.gb')
4. gb_iterator = parse(gb_file, 'genbank')
5. gb_record = gb_iterator.next()
6. gb_file.close()
7. exons = [ feat for feat in gb_record.features
8.           if feat.type == 'exon' ]
9.
10. seq_records = []
11. for exon in exons:
12.     start = exon.location.start.position
13.     end = exon.location.end.position
14.     number = exon.qualifiers['number'][0]
15.     sequence = gb_record.seq[start:end]
16.     record = SeqRecord(sequence)
17.     record.name = 'AMY1.%s' % (number,)
18.     record.id = record.namev
19.     record.description =
20.         'Human amylase gene, exon %s' % (number,)
21.     seq_records.append(record)
22. outfile = open('AMY1_exons.fa','w')
23. write(seq_records, outfile, 'fasta')
24. outfile.close()
25.
```

own qualifiers without having to adhere to a globally-defined standard.

The sequence for the exon is then obtained by slicing out the region between the start and end positions previously obtained (line 15). A new record is created around this sequence, and its name is set to our custom format, which will be different for each exon. Because we are writing this out to the FASTA format, the `id` and `name` attributes should be the same, so we can set the record's `id` to be its `name`. In addition, we can also give each record a description, which we've set to give a basic summary of the whole gene's function and the particular exon number. Finally, each newly created record is appended to the list.

Now that it has a list of fully created sequence records, the program can write them all out to 'AMY1_exons.fa' (lines 22-25). This example uses the standard form of `write()`, which is usually used for writing output in a quick-and- dirty way. For more precise control, it is often necessary to use the more specialized writers such as the `FastaWriter` class used before. There you have it: a FASTA-formatted file which contains all of the exons which will go on to make the alpha- amylase 1 enzyme.

Conclusion

BioPython is an incredibly versatile and well documented tool for bioinformaticists. In this article I have barely scraped the surface of all that it has to offer. There are some very well-written tutorials out there that go into a much greater depth. BioPython is constantly adapting to both the large and the small discoveries being made every day, and it will continue to do so with time. I can only wish you good luck getting to know it in the future.

ZACHARY VOASE is a high school student currently studying at the English International College in Marbella, Spain. He programs in Python in his spare time and plans to read Software Engineering in University. Previous experience has taught him that computers don't like drinking Coca-Cola.

Scripting Xcode with Python

by JC Cruz

This article will teach you how to write an Xcode menu script in Python. You will learn how to provide data to the script and how to handle its output. Next, you will learn to use an Xcode script macro or utility scripts. And you will get four script examples, written in Python, which you can change for your own use.

The Xcode Tool

Xcode is the de facto tool for building software products for the MacOS X platform. This is the tool to use to build applications based on the AppKit framework, or drivers based on the IOKit framework. Xcode's main compiler is gcc, which supports a wide range of languages including C/C++, ObjC, and Java. It uses the Gnu Debugger, or gdb, to trace or alter program execution, view and change program variables, or run functions separate from the normal execution flow. Xcode also supports CVS and Subversion for tracking changes made to project files, creating source branches, and generating project histories.

In addition to these editing features, Xcode comes with a complement of support tools. With Interface Builder you can build and edit your project's graphical user interface. The Computer Hardware Understanding Developer Tools (CHUD) helps you to locate any issues that will degrade a product's performance. With PackageMaker, you can combine your product and its support files into a package that you can give to your users.

But Xcode has its own set of issues. First, it does not support Git, the new source code management tool from

REQUIREMENTS

PYTHON: *2.4+*

Other Software: *Xcode 3.0, MacOS X 10.5.x*

Useful/Related Links:

- *The Xcode User Guide:* *http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeUserGuide*

Linus Torvalds. It lacks any metrics to measure source code complexity. It does not have an integrated backup feature for its projects and SCM repositories. And it does not support more advanced SCM features, forcing you to use the command-line.

Future releases of Xcode will address some of these limitations. You can, of course, address some of them yourself by writing an Xcode menu script.

The Xcode Script Menu

The Xcode Script menu gives you access to menu scripts to enhance the features of Xcode. It comes with a default set of scripts, which are grouped in terms of functions. For instance, the **Open** scripts open the path selected in the active document. The **Comments** scripts manages the comment tags and metadata on the document. The **HeaderDoc** scripts add placeholders for headerdoc info. The **Text** scripts run basic text processes on the active document.

The scripts are stored in `/Developer/Library/Xcode/XCUserScripts.plist`. This file uses the `.plist` XML format, the same format that most OS X applications use to store their preferences. You can view the contents of the file by opening it with your favorite text editor, or you can use the Property List Editor tool, which is part of the Xcode suite of tools.

You can also use Python and its XML modules to parse the contents of `XCUserScripts.plist`. But keep in mind that this is an Apple-specific format that may change in later revisions of Xcode.

You can write menu scripts using a wide range of scripting languages. The default scripts, in fact, are written in either Bourne shell or Perl. But there are advantages to writing a menu script in Python. Python's syntax is cleaner and more consistent than either Bourne or Perl. It is also easier to read, write and maintain than Perl. In addition, it is an object-oriented language by design.

The Xcode Script Editor

To make changes in the Xcode Script menu, choose **Edit User Scripts** from that menu. Xcode then displays the script editor shown in Figure 1.

The editor has five basic parts. On the left, highlighted in green, is a list showing the items displayed on the Script menu. The first column of the list shows the name of each script or category on the menu. To display the scripts under each category, click the triangle next to the category name. To select a script for editing, click its name. To change the name of the script or category, double-click its name on the list.

The second column of the list shows the script's shortcut key. To assign a shortcut key to a script, double-click the field next to that script's name. Enter the shortcut key by pressing a character key while holding down one or more control keys. For example, to assign the shortcut `Shift-Ctrl-Q`, hold down the Shift and Control key, and then press the `Q` key. To remove a shortcut key, click the **-** widget on the field.

On the right of the window, shaded in grey, is the editor field. This field displays the script that you selected from the list. Notice that Xcode uses syntax coloring to make the script easier to read.

At the top of the editor field, highlighted here in purple, are the input controls. The first pop-up menu sets the *input source* of the script. There are three choices: none, the entire text of the active document, or the selected text from that document. To read the input string, use `sys.stdin.readlines()`.

The second pop-up menu sets the working directory of the script. Again, this menu gives three choices: your home directory, the project's directory, or system root (`/`). To get the working directory from within your script, use either `os.getcwd()` or `os.getcwdu()`.

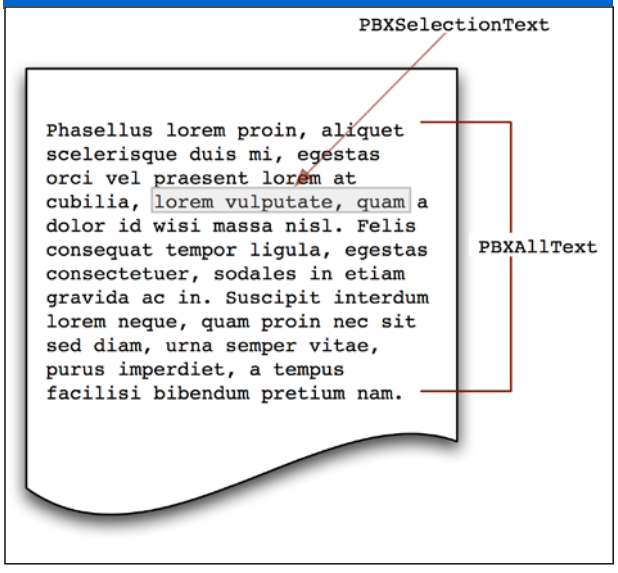
At the bottom of the editor field are two output controls, shaded blue. The first pop-up menu sets the *output handling* for anything written to `sys.stdout` by your script. You can have Xcode send the output to the active document, to a separate document, or to an alert dialog. You can also have Xcode discard the output.

The second pop-up menu sets the *error handling* for all messages written to `sys.stderr` by the script. Xcode will merge the errors with stdout, display them on an alert dialog, or discard them entirely.

At the bottom of the script list are the edit controls. Click the **-** button to remove a selected script from the list. Use it also to remove a script category and all the scripts in that category.

Click the **+** button to display the drop-down menu used to manage the script list. To add an empty script to the list, choose **New Shell Script** from the menu. For a new script category, choose **New Submenu**. Both menu actions create a placeholder name on the list that you can rename. You can further organize the menu with dividing lines, using the **New Separator** option. To copy an entry in the list, select the entry and choose **Duplicate Script**. When the editor makes a copy of a script or category, it adds the word "copy" to the original name.

FIGURE 2



Script Macros

Xcode has eight script macros that you can use in your menu scripts. To use a macro, make sure to enclose its name in %%%{ and }%%% tokens. Then assign both name and token to a variable as a quoted string. For example, to use the macro PBXAllText, enter it as follows.

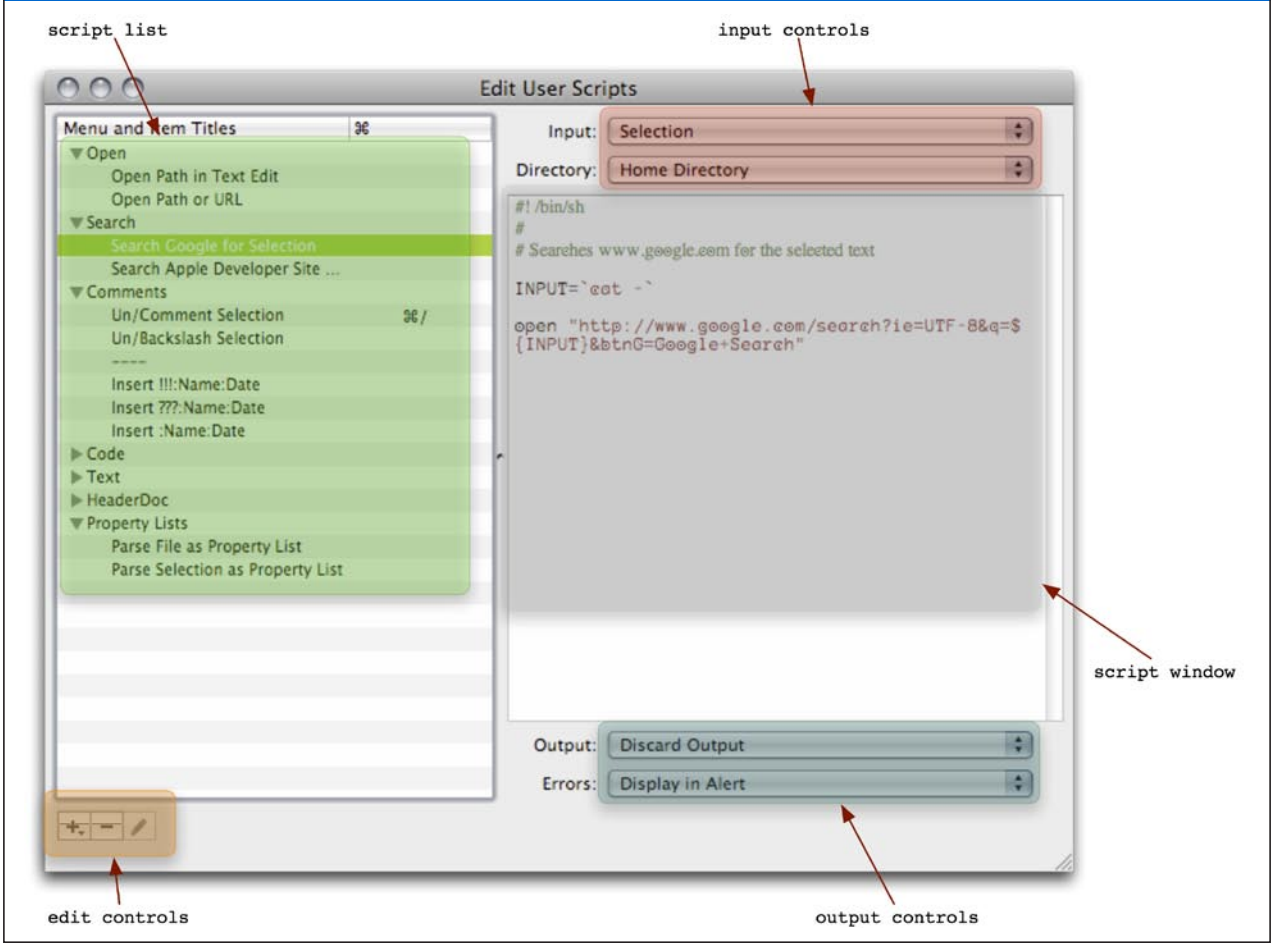
```
tTxt = "%%%{PBXAllText}%%%"
```

The token identifies the name as a macro. Xcode then replaces the macro with the right string data. But if the name is not of a valid macro, Xcode leaves both name and token unchanged.

There are three groups of script macros. The first group define the *input data* for the script, as illustrated in Figure 2. PBXAllText expands to the entire text in the *active document*. PBXSelectedText includes only the *selected text*.

Notice that these two macros provide the same service as the editor's input controls. Both use the same

FIGURE 1



active document as their data source. In fact, you can use both macros and input controls to give two input sources to your script. But whereas the input controls pass the input string as an stdin argument, the macros expand in-line to the specified string. This can be a problem if the string contains quotes. Consider the following code snippet.

```
tTxt = "%%%{PBXAllText}%%%"
```

If the active document has the phrase `The big brown fox "jumps" over the lazy dog`, then `PBXAllText` will expand as follows.

```
tTxt = "The big brown fox "jumps" over the lazy dog"
```

Python will return a syntax error when it finds the second " on the line. One way to fix this problem is to use a backslash (\ ) to escape the quotes in the text within the editor.

```
The big brown fox \"jumps\" over the lazy dog
```

But this means changing the source string, something to be avoided. A better way is to enclose the macro in triple quotes in your script

```
tTxt = """%%%{PBXAllText}%%%"""
```

to produce:

```
tTxt = """The big brown fox "jumps" over the lazy
dog"""
```

The second group of macros deal with *text metrics* (Figure 3). `PBXTextLength` gives the total number of characters in the active document. There are three macros for working with the selection: `PBXSelectionLength` is replaced with the number of selected characters in the document, `PBXSelectionStart` the starting index of the selection, and `PBXSelectionEnd` the ending index.

Xcode counts only the number of printable characters for these macros. It then returns the count as an integer. You can store the count as either an integer or a string. For example, to store the length of the selected text as a string, type the macro as a quoted string.

```
tSel = "%%%{PBXSelectionLength}%%%"
```

To store the same length as an integer, leave out the quotes.

```
tSel = %%%{PBXSelectionLength}%%%
```

The four macros related to text metrics are the only ones that do not need the use of enclosing quotes.

The third group of macros are those that do not fit the previous two groups. `PBXFilePath` gives the absolute file path to the active document. It allows you to operate directly on the active document. Use it to read the data of the document, or to overwrite the document with new data. Note that this macro works only if the active document is a text file. If the document is a binary file such as a `.png` or a `.icns`, or if it's a bundle such as a `.xib` or `.lproj`, the macro will return a null string.

`PBXSelection` sets the selection marks on the output string. Use one copy of the macro to mark the start of the selection, and another to mark the end of the selection. For example, assume you have the following output string.

```
The big brown fox jumps over the lazy dog.
```

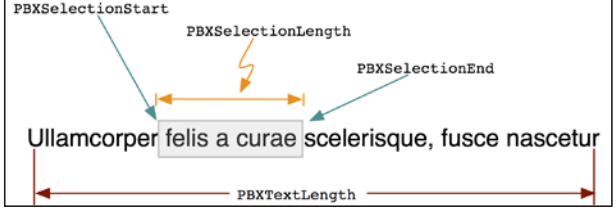
To select the word `jump` in the string, use the `PBXSelection` macro as follows.

```
tTxt = "The big brown fox "
tTxt += "%%%{PBXSelection}%%%"
tTxt += "jumps"
tTxt += "%%%{PBXSelection}%%%"
tTxt += " over the lazy dog."
print tTxt
```

Xcode, however, will highlight the marked text only if the standard output is either the active document or a new blank document. If the script output is being directed to either an alert dialog or the clipboard, Xcode leaves the marked text unselected.

Precisely how Xcode displays the output text on the active document depends on the script's output control settings. When the setting is **Replace Selection**, Xcode replaces the selected text with the output. If there is no selection, Xcode places the output text after the text cursor. If you choose **Replace Document Contents**, Xcode replaces the entire text in the active document with the output. For **Insert after Document Contents**, Xcode appends the output text after the document's text. Finally, when the output setting is **Insert after Selection**, Xcode appends the output text after the selected text. Again, if

FIGURE 3



there is no selection, Xcode places the output text after the text cursor. In all four output settings, Xcode places the text cursor right after the output text.

The Xcode Utility Scripts

Xcode has five utility scripts that you can call from your menu script. These utilities allow your script to interact with other users by the use of dialog windows. In fact, they work in the same way as the **EasyDialogs** module that comes with the MacOS version of Python. But the utility scripts have some advantages over EasyDialogs. Their dialog windows have a more consistent OS X look and feel since they use the **Standard Additions** library. They also place their windows in front of the active Xcode window, while EasyDialogs places its windows behind.

`AskUserForNewFileDialog` displays a **Save File** dialog to the users and returns the absolute path to the named file. It takes two arguments, the *prompt* used for the dialog's message and a *default_name* for the name of the file to create. If you leave either one blank, the script will use its own default value.

The calling syntax for `AskUserForNewFileDialog` is:

```
AskUserForNewFileDialog [prompt [default_name]]
```

The second utility is `AskUserForExistingFileDialog`. It displays an **Open File** dialog to the user and returns the absolute path to the selected file. The **Open File** dialog displays hidden and locked files, as well as hidden directories. It shows bundles, which are special directories, as single files.

`AskUserForExistingFileDialog` takes a single *prompt* parameter for the dialog's prompt message. But if you leave it blank, the script displays the dialog without a prompt message.

Call `AskUserForExistingFileDialog` as follows:

```
AskUserForExistingFileDialog [prompt]
```

`AskUserForFolderDialog` displays a **Select Directory** dialog to the user and returns the absolute path to the selected directory. The **Select Directory** dialog displays only visible directories in the target volume. It displays the files in each directory, but leaves those files disabled, and it treats bundles as single files. As with `AskUserForExistingFileDialog`, if the parameter *prompt* is blank, the script displays the dialog without a prompt message.

The calling syntax for `AskUserForFolderDialog` is:

```
AskUserForFolderDialog [prompt]
```

The script `AskUserForApplicationDialog` displays a **Select Application** dialog and returns the absolute path to the application selected by the user. How it returns the path depends on the type of application. If the application is a bundle, the script returns the path with a trailing / (like a directory) and then adds a `.app` suffix to the application name. If the application is a single binary, the script returns the path without a trailing / and `.app` suffix. The **Select Application** dialog lists all the applications in the target machine. But it does not include command-line tools in its list.

Run `AskUserForApplicationDialog` with two possible arguments:

```
AskUserForApplicationDialog [title [prompt]]
```

The *prompt* is the dialog's prompt message and *title* is the dialog's title. If either one is blank, the dialog is displayed with default values.

`AskUserForStringDialog` displays a basic input dialog and returns the string entered in the field. It takes two optional arguments.

```
AskUserForStringDialog [default [prompt]]
```

Pass *default* to set the initial value of the dialog. Use *prompt* to set the dialog's prompt message. If either one is blank, the script displays the dialog with default strings. The basic input dialog has its own set of issues: it does not let you change its title string; it does not have a secure input mode, like that of a password field; and it does not have a time-out feature, so your script hangs until the user responds.

Calling an Xcode utility script from Python can be a little tricky, but if you follow a few simple rules, you will have no problems calling the desired script. First, to call a utility script, use `os.popen()` with the script's location on the OS X volume. Use the special macro `PBXUtilityScriptsPath` to include the location in the call. For example, to call the utility script `AskUserForStringDialog`, use `os.popen()` as follows.

```
tCmd = "%%%{PBXUtilityScriptsPath}%%%"
tCmd += "/AskUserForStringDialog"
tOut = os.popen(tCmd)
```

Note the forward slash (/) in front of the script's name in line 2. When Xcode expands `PBXUtilityScriptsPath`, it does not end the path with a /. Adding a forward slash before the script's name fixes this issue.

Second, to read the script's result, use the following code snippet.

```
tOut = tOut.read().strip()
```

All five utility scripts return their results as a string printed to standard output. This snippet reads and trims out any whitespace at the end of that string.

It is important to pass all parameters to the utility script as quoted strings. Also, make sure to append each parameter to the script call in the correct order. For example, to display a **Save File** dialog, call `AskUserForNewFileDialog` as follows.

```
tMsg = '"Save your backup as" '
tNom = '"foobar.tar.gz"'

tCmd = "%%%{PBXUtilityScriptsPath}%%%/"
tCmd += "AskUserForNewFileDialog "
tCmd += tMsg
tCmd += tNom
# Hide error messages
tCmd += ' 2>/dev/null'

tPth = os.popen(tCmd).read().strip()
```

Here, the first two lines set the utility script arguments. Note the use of single quotes to enclose the two strings. The `tCmd` variable contains the script call, and the last line executes the call and stores the results in the variable `tPth`.

Finally, all five scripts display a **Cancel** button on their dialog windows. When users click on that button, the script prints an error message to stderr, and exit with an error code. To prevent the error message from showing up in the output of your script, redirect it to `/dev/null` as in the example above.

Menu Script Basics

An Xcode menu script starts with the same shebang (`#!`) header found in a typical Unix shell script. This header sets the path to the Python interpreter, which is usually `/usr/bin/python` on MacOS X 10.5. This path, however, may change in future versions. So test your menu scripts on every major release of OS X.

A menu script should return an exit code when it ends. An exit code of `0` means the script ran without errors; other numbers means an error occurred. Use `sys.exit()` to return the right code. If this call is absent, the menu script returns an exit code of `0` by default.

Now, the best way to learn how to write an Xcode menu script is to get a working script and learn how it comes together. The following are five menu scripts that I use in most of my Xcode sessions. I have tested these scripts on Xcode 3.0. Feel free to modify them for your own use.

Example: Counting Lines

Listing 1 is a very simple menu script. It uses the active document as its input and an alert dialog as its output. First, the script reads the entire contents of the document (line 7). Then it counts the number of lines of readable text (line 10), and prints the results to output (line 15).

A variation of this script is shown in Listing 2. This time, the script counts only those lines that form the source, skipping over comment and blank lines (lines 16-18). Basically, this script returns a software metric called SLOC (*source lines of code*).

Listing 2 handles only sources files written in languages that use # as the comment prefix, like Python. Another variation would be to modify this script to handle other source languages. The script could determine the right language by looking at the file's suffix, then filter out the comments for that language.

LISTING 1

```
1. #!/usr/bin/python
2. # -*- coding: utf-8 -*-
3. #
4. import sys
5.
6. # read the data in STDIN
7. tInp = sys.stdin.readlines()
8.
9. # count the number of lines in STDIN
10. tCnt = len(tInp)
11.
12. # display the count results
13. tOut = "There are " + repr(tCnt) + " lines in the document."
14.
15. print tOut
16.
```

LISTING 2

```
1. #!/usr/bin/python
2. # -*- coding: utf-8 -*-
3. #
4. import sys
5. import re
6.
7. # read the data in STDIN
8. tInp = sys.stdin.readlines()
9.
10. # count the number of lines in STDIN
11. tCnt = 0
12. for tLne in tInp:
13.   tTst = tLne.strip()
14.   tLen = len(tTst)
15.   if (tLen > 0):
16.     tTst = re.search('^#', tTst)
17.     if (tTst is None):
18.       tCnt += 1
19.
20. # display the count results
21. tOut = "SLOC: " + repr(tCnt)
22. print tOut
23.
```

Example: Searching For Text

Listing 3 is a little more complex. This script uses two input sources: the selected text in the active document, and the Xcode script macro `PBXFilePath`. The script's output is directed to a blank Xcode document.

First, the script reads the selected text from stdin (line 8). Then it locates and reads the contents of the active document (lines 15-20). Next, the script checks each document line. When it finds a line that has the selected text, the script adds the line and its line number to an output string buffer. It then prints the buffer to stdout.

Note that the script prints its results after it checks all the lines. You can have the script print each line as it goes, but Xcode displays the script's output only after the script ends its run. Because of this behavior, how the menu script handles its output is of little matter.

One useful variation of Listing 3 would be to have it do context-sensitive parsing. For instance, if the selection is part of a comment line, the script would parse only the comment lines in the document. If the selection is part of a source line, the script would parse only the source lines.

Example: Backing Up Your Project

Listing 4 shows how you can use an Xcode menu script to backup an Xcode project. This script uses the **tarfile** module to create the backup. It uses `AskUserForNewFileDialog` to interact with the user.

First, the script gets the location of the Xcode project (line 9). Then it asks the user to set the name and location of the backup (lines 14-25). Next, the script opens the tarball where it will write the backup (line 29). It then uses `os.walk()` to list and store each project file into the backup (lines 31-48). Once done, the script closes the output, and displays a success alert to the user (lines 50-56).

This script suffers from two flaws. First, it assumes that all source files are on the root level of the project directory. If the script is invoked while you are viewing a source file that is in a subdirectory of the project, the script might backup only that subdirectory, thus

LISTING 3

```
 1. #!/usr/bin/python
 2. # -*- coding: utf-8 -*-
 3. #
 4. import sys
 5. import re
 6.
 7. # read the data in STDIN
 8. tTgt = sys.stdin.readline()
 9. tTgt = tTgt.strip()
10.
11. if (len(tTgt) < 1):
12.   print "You need to select something."
13. else:
14.   # read the source file
15.   tSrc = "%%%{PBXFilePath}%%%"
16.
17.   # open the file for reading
18.   tSrc = open(tSrc, "r")
19.   tInp = tSrc.readlines()
20.   tSrc.close()
21.
22.   # parse each line of text
23.   tOut = ""
24.   tCnt = 0
25.   for tLne in tInp:
26.     # does the line have the target word?
27.     tCnt += 1
28.     tFnd = re.search(tTgt, tLne)
29.     if (tFnd is not None):
30.       # update the output string
31.       tStr = str(tCnt) + ":\t" + tLne
32.       tOut += tStr
33.
34.   # display the results
35.   print tOut
36.
```

LISTING 4

```
 1. #!/usr/bin/python
 2. # -*- coding: utf-8 -*-
 3. #
 4. import sys
 5. import os
 6. import tarfile
 7.
 8. # retrieve the current working directory
 9. tCWD = os.getcwd().strip()
10.
11. # set the following string constants
12. tMsg = "Save the tarball backup in this directory"
13.
14. # prepare the default name for the tarball backup
15. tNom = os.path.basename(tCWD)
16. tNom = tNom + ".tar.gz"
17.
18. # ask the user where to save the tarball backup
19. tCmd = "%%%{PBXUtilityScriptsPath}%%%"
20. tCmd += "/AskUserForNewFileDialog \""
21. tCmd += tMsg + "\" " + tNom
22. tCmd += " 2>/dev/null"
23.
24. tBck = os.popen(tCmd)
25. tBck = tBck.readline().strip()
26.
27. if (len(tBck) > 0):
28.   # create the tarball backup
29.   tTar = tarfile.open(tBck, "w:gz")
30.
31.   # get a list of files in the working directory
32.   for tOrg, tDir, tItm in os.walk(tCWD):
33.     # parse the file list
34.     for tNom in tItm:
35.       # add the file to the tarball backup
36.       tPth = os.path.join(tOrg, tNom)
37.       tTar.add(tPth)
38.
39.     # parse the directory list
40.     tIdx = 0
41.     for tNom in tDir:
42.       if (tNom.find(".git") >= 0):
43.         # exclude the Git subdirectory
44.         del tDir[tIdx]
45.       elif (tNom.find("build") >= 0):
46.         # exclude the build subdirectory
47.         del tDir[tIdx]
48.       tIdx += 1
49.
50.   # close the tarball backup
51.   tTar.close()
52.
53.   # inform the user
54.   tMsg = "Backed up the project at: "
55.   tMsg += tBck
56.   print tMsg
57.
```

missing the rest of the project. One solution is to have the script walk outwards each directory, checking for an `.xcodeproj` bundle. A project directory will always have this bundle in its root level.

The second flaw in this script is that it expects the active document to be a text file. If that document is a binary file or a bundle, `os.getcwd()` will return the path to the home directory. So far, the only workaround is to inform the user that the working directory is incorrect.

This backup script could be enhanced to ask the user to choose the backup format. Some format choices include ZIP and Pax. Another variation is to have the script do incremental backups. If a project file is newer than its backed up version, the script updates the backup with the new version.

Example: Working with Version Control

Listing 5 shows how to use a menu script to export an Xcode project from a CVS repository. The script does not need any input or output. Instead, it uses `AskUserForStringDialog` and `AskUserForNewFileDialog` to interact with the user.

First, the script checks if CVS is tracking the project directory (line 16). If this is true, the script gets the location of the CVS repository (lines 18-20). Next, the script asks the user what project revision to export, and where to store the export (lines 22-36). Finally, the script prepares and executes the CVS export command (lines 39-48). It then displays any errors during the export in an alert dialog.

This script, however, works only if the project is tracked by CVS. It could be enhanced to check what SCM tool is in use and perform the correct export action for that tool.

This script also exports the project in its original form, that is, as a collection of files and directories. You can modify the script to store the exported project in an archive file such as a tarball or send the exported project to a specified user via e-mail.

Final Remarks

A menu script is an effective way of enhancing an Xcode session. A script can add a new feature to Xcode or extend an existing feature. It can aid Xcode in managing a project's files, or allow users interact with an ongoing Xcode process.

The Xcode Script menu gives quick access to all menu scripts. It handles the flow of data to and from a menu script. Its editor makes it easy to add a new script, edit an existing one, or assign a shortcut key to a script.

I hope this article gives you an idea of how to write an Xcode menu script in Python. So, until next time, relax and have a cup of tea.

LISTING 5

```
1. #!/usr/bin/python
2. # -*- coding: utf-8 -*-
3. #
4. import sys
5. import os
6.
7. # initialize the following locals
8. tTag = "Which revision do you want to export?"
9. tXpt = "Save the exported project in:"
10.
11. tPth = os.getcwd().strip()
12. tPrj = os.path.basename(tPth)
13.
14. # is CVS tracking this project?
15. tCVS = os.path.join(tPth, "CVS/Root")
16. if (os.path.exists(tCVS)):
17.   # get a path to the CVS repository
18.   tSrc = open(tCVS, "r")
19.   tCVS = tSrc.readline().strip()
20.   tSrc.close()
21.
22.   # ask the user for the revision tag
23.   tCmd = "%%%{PBXUtilityScriptsPath}%%%"
24.   tCmd += "/AskUserForStringDialog \"\" \""
25.   tCmd += tTag + "\" 2>/dev/null"
26.   tTag = os.popen(tCmd)
27.   tTag = tTag.readline().strip()
28.
29.   if (len(tTag) > 0):
30.     # ask the user where to export the project
31.     tCmd = "%%%{PBXUtilityScriptsPath}%%%"
32.     tCmd += "/AskUserForNewFileDialog \""
33.     tCmd += tXpt + "\" " + tPrj
34.     tCmd += " 2>/dev/null"
35.     tXpt = os.popen(tCmd)
36.     tXpt = tXpt.readline().strip()
37.
38.     if (len(tXpt) > 0):
39.       # prepare the CVS export command
40.       tCmd = "cvs -d tCVS -q export "
41.       tCmd += "-r tTag -d tXpt tPrj"
42.       tCmd = tCmd.replace("tCVS", tCVS)
43.       tCmd = tCmd.replace("tTag", tTag)
44.       tCmd = tCmd.replace("tXpt", tXpt)
45.       tCmd = tCmd.replace("tPrj", tPrj)
46.
47.       # execute the command
48.       tCVS = os.system(tCmd)
49.
50. else:
51.   print "The project is not being tracked by CVS"
```

JC is a semi-retired software engineer. Years ago, he worked for companies like Apple and Adaptec, designing software that never saw the light of day (i.e. drivers). He now lives in British Columbia, working as a freelance tech writer. He wrote various computer articles, most of which are published in MacTech and REALbasic Developer. He also spends quality time with his little nephew.

Portable Data with PyTables

by Eugen Wintersberger

Saving data to disk is a common job for every program. PyTables offers a simple interface to the great HDF5 library, saving your data in a portable way. This article shows a real life example of how to convert an ASCII format to HDF5 using PyTables.

Introduction

At some point nearly every program has to write data to disk. The data might be log information or numerical data from a computer simulation or a scientific experiment. Though any kind of data can be stored using PyTables and HDF5, the focus of this article is numerical data. This is what HDF5 was originally made for and is therefore its most common application.

Scientists and engineers tend to prefer writing data from experiments or simulations to simple ASCII files. The argument for this is that ASCII files can be opened on every platform, in the worst case with a simple editor. However, writing numerical data to ASCII files can cause two problems in particular with floating point values:

- Floating point numbers can be written only with limited precision; thus, rounding errors can occur if the number of digits written to the file is truncated.

REQUIREMENTS

PYTHON: *2.5+*

Other Software:
- Numpy: >=1.03 http://www.scipy.org
- PyTables: http://www.pytables.org/moin

Useful/Related Links:
- HDF5 homepage: http://hdf.ncsa.uiuc.edu/products/hdf5/index.html
- LZO data compression: http://www.oberhumer.com/opensource/lzo/
- bzip2: http://www.bzip.org/
- EDF documentation: http://www.esrf.fr/computing/expg/subgroups/general/format/Format.html

- Files can quickly become very large if an application is data-intensive, especially when high precision of the numbers is required.

To get a better feel for what we are talking about, let us assume a simple eight byte double value. Such a floating point number usually provides a precision of 15-16 digits (if using IEEE 754 coding). Storing that number in ASCII with full precision would occupy at least 15-16 bytes (one byte per digit – and I do not take into account the two bytes for the comma and the sign) instead of the original eight bytes required for binary storage. From this simple consideration, one could assume that the best method to store numerical data is to use a binary format. However, binary representation of numbers is highly platform-dependent: writing binary files in a naive way can make your files unreadable on other machines.

Another problem is how to handle metadata and whether to store several data sets in a single file. The data in a file must be organized somehow, and we probably have to save some additional information like the date the measurement was taken or the environmental parameters under which the data was recorded. How can we manage all of these things? Of course, if you're working alone, you can define your own file format and provide your own toolchain to read and write your data. However, nowadays most scientists and engineers work together in cooperation rather than in isolation. In such a case, conversion and import/export tools must be provided and maintained to handle a custom binary file format.

One solution would be to use XML to organize the data in a file and store the metadata. XML files are plain ASCII files and will work on any platform. The size of the files can be reduced by means of external compression programs like bzip2 or gzip (this is how OpenOffice is storing data, for example). But even this approach has some disadvantages:

- it does not solve the rounding problem for floating point numbers
- compressing/unpacking the file can take a lot of time
- the entire file must be compressed.

HDF5 (Hierarchical Data Format) addresses all of these problems. HDF5 is an open and binary file format developed by the NASC (National Association of Super Computing). It can deal with any kind of data, though its strength is clearly in storing numerical data. Besides simply storing binary data, HDF5 also allows you to structure data in any kind of hierarchy, and it provides a very gentle way to store metadata.

HDF5 comes as a C library with bindings for C++ and Fortran. The HDF5 API is as complex as HDF5 itself is powerful. Fortunately, a simple interface called PyTables exists for Python that allows you to store data in HDF5 files and use the most common features of HDF5 without navigating the complex API.

HDF5 - Hierarchical Data Format

Before talking about PyTables, some fundamental knowledge about the concepts behind HDF5 is required. All objects stored in an HDF5 file can be sorted into "groups." This gives your file some structure in case you have to save many objects into a single file. These groups act a bit like directories on a file system. In addition, every object can have several attributes holding further information about a particular object. Imagine that we have a group with arrays of data measured during a certain day. An attribute of the group object could be the date on which the data was recorded or the temperature at which it was recorded, etc.

To store data, a so-called "dataset" must be created, along with two other objects: a "datatype" and a "dataspace." A "datatype" can be one of the native numeric types HDF5 provides, though it's not restricted to scalar values. Vector types and user-defined compound types (such types can be used to store the content of a C structure) are also allowed. In fact the "datatype" defines what should be saved in the "dataset." The "dataspace" defines how the data elements, defined by the "datatype," should be stored. For instance as a `1024x768` array or as a vector with 10 entries. The "dataspace" must not be static. It is also possible to define an extendable dimension in a dataspace so that additional data can be added to the "dataset." Together, the "datatype" and the "dataspace" form the "dataset." You might think that this is unnecessarily complex, and you are completely right! At first glance, it really is. But treating dataspaces and datatypes as independent objects makes them reusable throughout your code. Once you have defined a dataspace of a certain shape, you can use it within your entire program. Finally, the dataset object is responsible for how the data is stored to the disk. For instance, whether or not an array of data should be compressed or how the data should be chunked before being passed to the compression and writing routines.

A particularly nice feature of HDF5 is that it allows the compression of single objects (by means of zlib). This is a great advantage over compressing the entire file since compression also decreases IO performance. In the case of HDF5, only extremely large data must be compressed while smaller ones can be left uncompressed. This would decrease the IO performance only on the large data sets but not on the small ones or on metadata stored in addition to an object.

Obtaining and installing PyTables

PyTables can be downloaded from the homepage of the project (see references). Ubuntu and Debian Linux users can install PyTables simply by installing the "python-tables" package via the apt package management system. For all other platforms, refer to the PyTables documentation.

A few words about PyTables

Before starting with PyTables, you should know a few basic concepts. PyTables does not support all features of HDF5. For instance, multidimensional atoms for CArrays (see below) are not possible. For many applications, however, the possibilities offered by PyTables are completely sufficient.

In most cases we can distinguish two classes of data:

- tabular data – spreadsheet-like data (i.e. scalar data organized in columns)
- array data – scalar data organized in *n* dimensional arrays (think of an image where color values are stored in a two dimensional array).

Basically, PyTables supports two major object classes to store data: tables and arrays. Array objects can be subdivided in four classes:

- normal arrays (class **Array**) which cannot be compressed
- chunked arrays (class **CArray**) which can be compressed
- extendable arrays (class **EArray**) which are compressible and can be extended along one dimension
- variable length arrays (class **VLArray**) which are compressible one-dimensional arrays where every entry can be of different length.

FIGURE 1

```
#F ./data/sige_crystal.spec                    SPEC file global header
#E 1140534969
#D Mon Feb 27 20:52:12 2006
#C sige_crystal  User = opid10
#O0    delta     gamma     omega     theta       mu      sigma    sigmat       xt
#O1       zt       zt1       thd       chid     rhod         xd        yd      PhiD
#O2       zd      arcf        zf       att0   gslitT    gslitB    gslitR    gslitF
#O3  gslitho   gslithg   gslitvo   gslitvg    phigH     chigH      xgH       ygH
#O4      zgH     phigV     chigV      xgV       ygV       zgV      s0hg      s0ho
#O5     s0vo      s0vg    slit1T    slit1B    slit1R    slit1F   slit1ho   slit1vo
#O6  slit1hg   slit1vg   slit2hg   slit2vg       TRT      chia       tha      ttha
#O7    picou     picod        pi      rien     zlens
```

```
#S 2  ascan  xgV -1.42663 -0.826625  30 0.5
#D Mon Feb 27 20:52:31 2006
#T 0.5  (Seconds)                          Header and data section of scan 2
#G0 0 0 0 0 0 0 0 0 0 0
#G1 1.54 1.54 1.54 90 90 90 4.079990459 4.079990459 4.079990459 90 90 90 2 0 0 4 1 0 0 47.28 23.64 0 0 0 0 69.1 79.55 0 0 0 1.54 1.54
#G3 -4.079990459 4.432114633e-16 6.561207576e-16 -2.129092944e-16 4.079990459 -2.49819112e-16 0 0 -4.079990459
#G4 0 0 0 1.54633 0 0 0 0 0 -180 -180 -180 -180
#Q 0 0 0
#P0 0 0 0 0 0 9.7131 -0.005255 -6.51625
#P1 4.4771875 92.1048 -5e-05 -0.2042 -1e-05 -0.88 0.023 1.37
#P2 4.36895 0 4.205 11 -2.3999704 0.49999998 0.24999998 0.25000012
#P3 -7.0921986e-08 0.50000009 -1.4499852 -1.8999705 0.115 -0.04 0 0
#P4 0 0 0 -1.126625 0 -3.25 0.20003582 0.27849285
#P5 -0.3810371 0.2 10 10 0.25 0.25 0 0
#P6 0.5 20 7.9999999 9.0000758 3.0013225 0 13.7179 27.3782
#P7 0 0 3.9990234 450 2.5
#UMI0 Current  AM   SH  1=U27-2=U35_Gap  1=U27-2=U35_Taper U27_Gap  U27_Taper U35u_Gap  U35u_Taper XBPM1X  XBPM1Z  XBPM1Q XBPM2X  XBPM2Z  XBPM2Q
#UMI1  193.60  ON AUTO    13.2008    1.0000   11.9882   -0.5325   22.5100    0.0000  -1.26  -0.10   0.00 -99.99 -99.99  -2.00
#UMI2 Refill in 40054 sec, Fill Mode: uniform / Message: Feb 27 18:34 Delivery:Next Refill at 08:00; End of U.S.M. at 8h00.
#UAUTOF active=YES  detector=all0  filtset=filtW
#N 24
#L xgV  H  K  L  Epoch  Seconds  Ion_m1  Ion_m2  all0  psd0  dir0  refl0  yoneda0  fb_Pi  fb_A  foil  srcur  curratt  detcorr  ratio  mcaLt0  ccdint  Monitor  Detector
-1.426625 0 0 0 534985 0.5 15354 10878 1878 1878 1 0 1878 -4.4839483 179.02223 2.1633902 193.59813 11 5.9446276e+09 0 0.48 0 11674 0
-1.406625 0 0 0 534986 0.5 15353 10878 1934 1934 0 0 1934 -4.4815746 179.13289 2.1679857 193.59813 11 6.1218902e+09 0 0.48 0 11859 0
-1.386625 0 0 0 534987 0.5 15352 10877 1952 1952 0 0 1952 -4.4841178 179.05614 2.1640683 193.59813 11 6.1788674e+09 0 0.48 0 11661 0
-1.366625 0 0 0 534988 0.5 15355 10878 1903 1903 0 0 1903 -4.4841446 179.06864 2.1686282 193.59813 11 6.0237627e+09 0 0.48 0 11824 0

....

#ATT xia: 0
#R Scan 2 Peak at -1.30662 is 6.35296e+09              COM at -1.27814   FWHM is 0.186793 at -1.11983
```

```
#S 3  ascan  phigV -0.5 0  50 0.5
#D Mon Feb 27 20:56:54 2006                Header and data section of scan 3
#T 0.5  (Seconds)
....
```

A particularly nice feature of PyTables is the fact that it maps **NumPy** arrays directly to PyTables arrays. Therefore, if you want to write or read an array you can use NumPy arrays directly as sources or targets respectively.

Finally, a few words of caution about compression: while HDF5 only supports compression via zlib, PyTables supports also bzip2 and the LZO library. In the case that the files created by PyTables should remain valid HDF5 files, only zlib should be used. The PyTables documentation has an elaborate section dealing with the different compression algorithms and how they influence writing and reading performance of PyTables. If you need more information about this topic have a look there. Let's have a look at how it works.

Using PyTables: a real life example

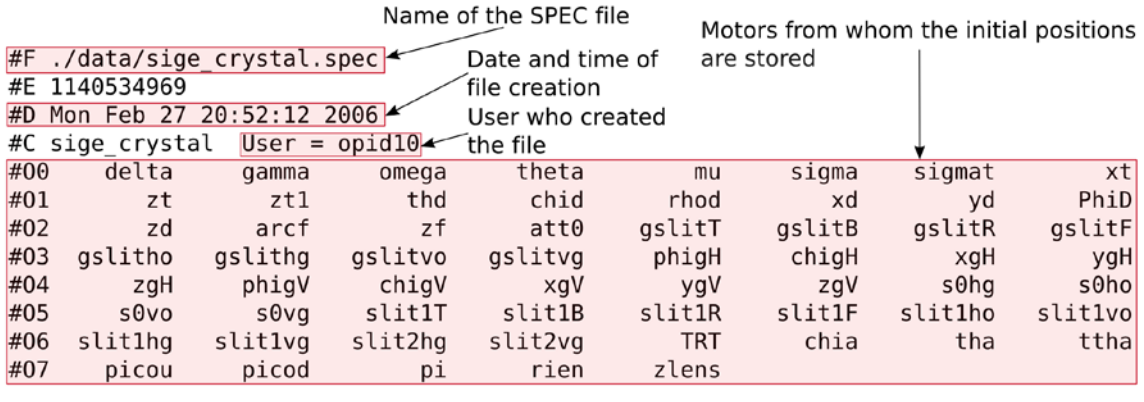
The author works from time to time at the European Synchrotron radiation facility (ESRF) in Grenoble (A synchrotron is a large facility for generating x-rays). Data recorded there is stored either as ASCII data in a "SPEC" file (so called after the experiment control system SPEC) or in a binary data format called EDF (ESRF Data Format). The latter is mainly used for array data recorded by CCD detectors (these detectors can be imagined as digital cameras, except that they are sensitive to x-rays). In accordance with the classification of the previous section, the EDF files contain array data while the SPEC files hold mostly tabular data (with an exception we will see later). In addition to the recorded data, these files contain plenty of metadata that might be of importance for data evaluation. In this and in the following sections I will show how to use PyTables to convert both file formats to HDF5 and by doing so make access to the recorded data much easier. A good starting point is to think about how we want to structure our data in the HDF5 file. For this purpose we should first look at how data is organized within SPEC and EDF files respectively.

As shown in Figure 1, a SPEC file consists of a global header followed by several so called "scans." The global header of a SPEC file holds general metadata as shown in figure 2. The block denoted as `initial motor positions` in figure 2 contains the names of the motors in the order that their "initial positions" are stored in the scan headers (more about this later). Figure 3 shows a simple SPEC scan holding only scalar data. Every scan consists of a header who's lines start with a # character and a data section. We find here again a block named `initial motor positions` where every number denotes the initial position (before the measurement starts) of a motor. The motor corresponding to a particular position can be obtained from the "initial motor positions" block in the global header of the SPEC file. The most important part of a SPEC scan is the header section with the column names and the data section below it. As already mentioned, a SPEC scan can also hold multidimensional data as shown in figure 4. When such is the case, the array data is stored in a block starting with `@A` after the record holding the scalar data. The array data comes from an MCA (Multi Channel Analyzer). In the case that such MCA data is stored in a SPEC file we find additional metadata in the scan header starting with `#@`.

The SPEC format is extremely flexible and therefore every file can look a bit different. This short description is far from complete, but will suffice for our purposes.

The EDF file used to store CCD data (at least in the project under discussion here) is as complex as the SPEC

FIGURE 2



file format. Therefore, we will restrict our case to the format used to save CCD data at the *ESRF beamline ID01* (beamlines are a particular kind of laboratory for x-ray experiments at a synchrotron). In this case the file starts with an ASCII block holding metadata in the form of key value pairs separated with an "=" sign. The header block is encapsulated by curly braces. After the header block the data is stored in binary form within the file. Again, the EDF format is much more complex as the simple example treated here. For more information about the EDF format, refer to the EDF documentation.

The final arrangement of the data in the HDF5 file is depicted in figure 5. To fully understand this structure, two additional hints are necessary:

- More than one SPEC file should be stored in the HDF5 file; therefore a top level group for every SPEC file is needed.
- One or more CCD records are taken for every data record in the SPEC file, so the CCD data will be stored in a separate group below the scan group.

The basic skeleton of such a structure can be easily created with the following lines of code:

```
import tables
f = tables.openFile("datafile.h5",mode="w")
fgroup = f.createGroup("/","sige_crystal",
         "Data of SPEC file sige_crystal.spec")
```

FIGURE 3

```
#S 2  ascan  xgV -1.42663 -0.826625  30 0.5          <- scan number and scan command
#D Mon Feb 27 20:52:31 2006                          <- date and time the scan has been started
#T 0.5  (Seconds)                                    <- integration time
#G0 0 0 0 0 0 0 0 0 0 0 0 0 0
#G1 1.54 1.54 1.54 90 90 90 4.079990459 4.079990459 4.079990459 90 90 90 2 0 0 4 1 0 0 47.28 23.64 0 0 0 0 69.1 79.55 0 0 0 1.54 1.54
#G3 -4.079990459 4.432114633e-16 6.561207576e-16 -2.129092944e-16 4.079990459 -2.49819112e-16 0 0 -4.079990459
#G4 0 0 0 1.54633 0 0 0 0 0 -180 -180 -180 -180
#Q 0 0 0
#P0 0 0 0 0 0 9.7131 -0.005255 -6.51625
#P1 4.4771875 92.1048 -5e-05 -0.2042 -1e-05 -0.88 0.023 1.37
#P2 4.36895 0 4.205 11 -2.3999704 0.49999998 0.24999998 0.25000012
#P3 -7.0921986e-08 0.50000009 -1.4499852 -1.8999705 0.115 -0.04 0 0
#P4 0 0 0 -1.126625 0 -3.25 0.20003582 0.27849285
#P5 -0.3810371 0.2 10 10 0.25 0.25 0 0
#P6 0.5 20 7.9999999 9.0000758 3.0013225 0 13.7179 27.3782
#P7 0 0 3.9990234 450 2.5
                                                     <- initial motor positions
#UMI0 Current  AM   SH  1=U27-2=U35_Gap  1=U27-2=U35_Taper U27_Gap  U27_Taper U35u_Gap  U35u_Taper XBPM1X  XBPM1Z  XBPM1Q XBPM2X  XBPM2Z  XBPM2Q
#UMI1  193.60  ON AUTO    13.2008    1.0000   11.9882   -0.5325   22.5100    0.0000  -1.26  -0.10   0.00 -99.99 -99.99  -2.00
#UMI2 Refill in 40054 sec, Fill Mode: uniform / Message: Feb 27 18:34 Delivery:Next Refill at 08:00; End of U.S.M. at 8h00.
#UAUTOF active=YES  detector=all0  filtset=filtW
#N 24                                                <- number of columns
#L xgV  H  K  L  Epoch  Seconds  Ion_m1  Ion_m2  all0  psd0  dir0  refl0  yoneda0  fb_Pi  fb_A  foil  srcur  curratt  detcorr  ratio  mcaLt0  ccdint  Monitor  Detector
                                                     <- column names
-1.426625 0 0 0 534985 0.5 15354 10878 1878 1878 1 0 1878 -4.4839483 179.02223 2.1633902 193.59813 11 5.9446276e+09 0 0.48 0 11674 0
-1.406625 0 0 0 534986 0.5 15353 10878 1934 1934 0 0 1934 -4.4815746 179.13289 2.1679857 193.59813 11 6.1218902e+09 0 0.48 0 11859 0
-1.386625 0 0 0 534987 0.5 15352 10877 1952 1952 0 0 1952 -4.4841178 179.05614 2.1640683 193.59813 11 6.1788674e+09 0 0.48 0 11661 0
-1.366625 0 0 0 534988 0.5 15355 10878 1903 1903 0 0 1903 -4.4841446 179.06864 2.1686282 193.59813 11 6.0237627e+09 0 0.48 0 11824 0
....                                                 <- column data

#ATT xia: 0
#R Scan 2 Peak at -1.30662 is 6.35296e+09             COM at -1.27814   FWHM is 0.186793 at -1.11983
```

FIGURE 4

```
#S 19  a2scan  om 32.7266 36.7266  gam 63.4428 71.4428  800 1
#D Mon Feb 27 21:31:43 2006
#T 1  (Seconds)
#P0 0 69.4428 35.7266 0 0 9.7131 -0.005255 -6.51625
#P1 4.4771875 92.1048 -5e-05 -0.2042 -1e-05 -0.88 0.023 1.37
#P2 4.36895 0 4.205 11 -2.3999704 0.49999998 0.24999998 0.25000012
#P3 -7.0921986e-08 0.50000009 -1.4499852 -1.8999705 0.115 -0.04 0 0
#P4 0 -1.082875 -2.632875 -1.119875 0.056125 -3.25 0.20003582 0.27849285
#P5 -0.3810371 0.2 10 10 0.25 0.25 0 0
#P6 0.5 20 7.9999999 9.0000758 3.0013225 0 13.7179 27.3782
#P7 0 0 3.9990234 450 2.5
#@MCADEV 0
#@MCA %16C
#@CHANN 1024 0 1023 1
#@CALIB 0 1 0
#@ROI all0 0 1023
#@ROI refl0 623 627
#@ROI yoneda0 0 1023
#@ROI psd0 0 1023
#@ROI dir0 1 200
                                                     <- MCA parameters -
                                                        %16C = 16 numbers per data row
#N 25
#L omega  gamma  H  K  L  Epoch  Seconds  Ion_m1  Ion_m2  all0  psd0  dir0  refl0  yoneda0  fb_Pi  fb_A  foil  srcur  curratt  detcorr  ratio  mcaLt0  ccdint  Monitor  Detector
32.7266 63.4428 1.04711 0.0183725 0 537350 1 30513 21553 2644 2644 0 1 2644 -4.4765876 179.17441 2.1492518 191.80591 0 2644 0 0.98 0 21661 0
@A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 ...
  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0                     <- MCA data block after first
                                                        row of column data
32.7316 63.4528 1.04726 0.0183751 0 537352 1 30518 21565 2474 2474 0 2 2474 -4.4780018 179.16697 2.1457534 191.80591 0 2474 0 0.97 0 21499 0
@A 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0\                  <- MCA data block after second
 ...                                                    row of column data
```

```
f.createGroup(fgroup,"scan2","Data of scan 2")
f.createGroup(fgroup.scan2,"ccd","EDF ccd data")

f.createGroup("/sige_crystal","scan3",
    "Data of scan 3")
f.createGroup("/sige_crystal/scan3","ccd",
    "EDF ccd data")

print f
f.close()
```

This yields the following output:

```
datafile.h5 (File) ''
Last modif.: 'Fri Feb  1 12:41:31 2008'
Object Tree:
/ (RootGroup) ''
/sige_crystal (Group) 'Data of SPEC file
sige_crystal.spec'
/sige_crystal/scan2 (Group) 'Data of scan 2'
/sige_crystal/scan3 (Group) 'Data of scan 3'
/sige_crystal/scan3/ccd (Group) 'EDF ccd data'
/sige_crystal/scan2/ccd (Group) 'EDF ccd data'
```

The `openFile()` function is pretty simple to understand: the mode argument can be either "w", "r", or "a" opening a file for writing (overwriting an existing file), reading, or appending data to an existing file respectively. Subsequently, the `createGroup()` method creates groups according to its arguments: the first argument is the location where the group should be created in the file, the second argument is the group's name, and the third argument is its description. The location can be given either by passing the parent group object as the first argument or by using a UNIX-like directory path representation where "/" denotes the "root" group. The root group is a special group which is always present in a HDF5 file acting as the toplevel object for all other data and group objects.

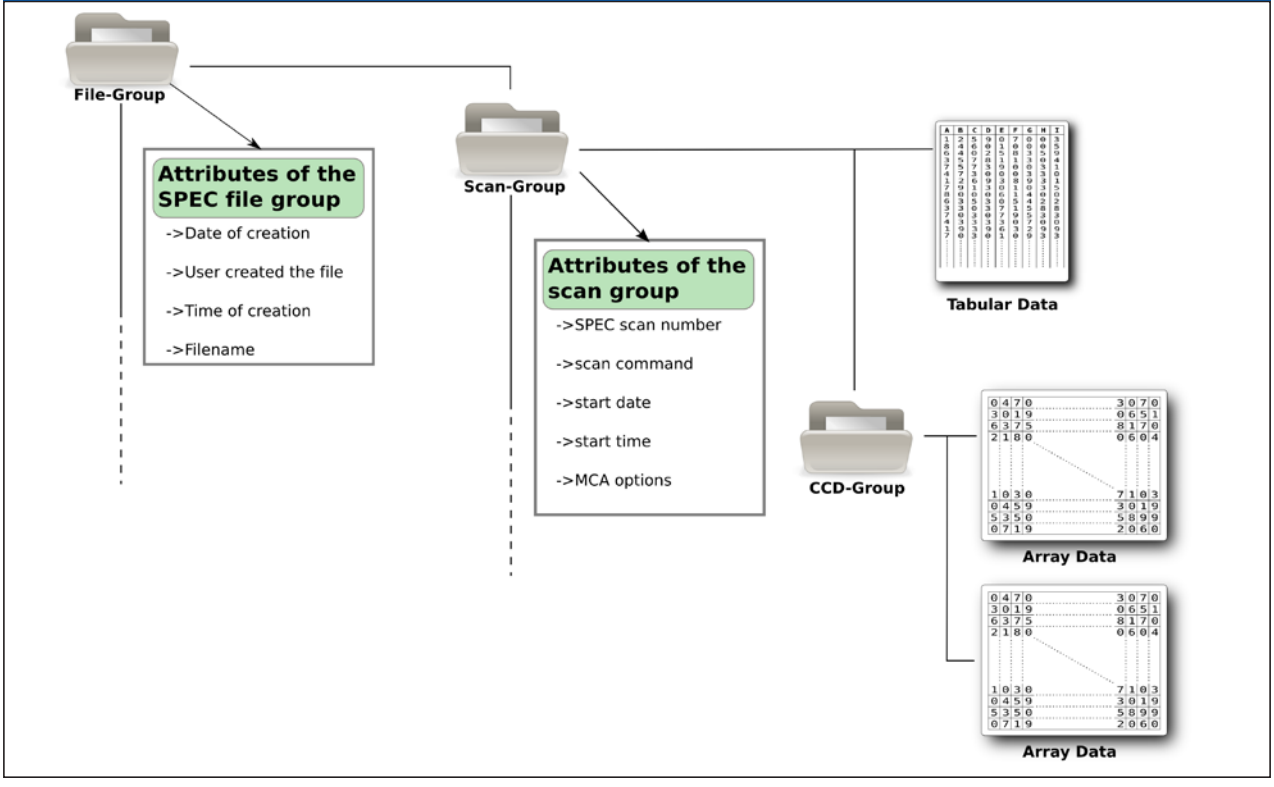
One nice feature of PyTables is, objects in the file hierarchy are attributes of their parent object. The file object is at the top level, so in the above example we can access the group `scan2` using the following notation:

```
scan = h5.root.sige_crystal.scan2
```

This does imply that object names have to follow the conventions for valid python variable names. In practice, it only means that object names must not include blanks or dots and may not start with digits.

Since the parsing of EDF and SPEC files is pretty complex, I will not show the code here but rather define two virtual modules called **spec** and **edf**. Header data should be represented by dictionaries where the name of the header entries are the keys and the dictionaries' values are the data belonging to a header entry. Throughout

FIGURE 5



NumPy record array

Record arrays are a conveniant way of storing tabular data. Record arrays posses keys to identify columns and an index to access a record which can be treated as the row of a table. The following short code sequence shows how to use record arrays.

```
import NumPy as n

rtype = {"names":["column1","column2"],
         "formats":[n.float32,(n.int,(2,2))]}
a = n.rec.array([
                [1.0,[[1,2],[3,4]]],
                [2.0,[[7,8],[9,19]]],
                ],dtype=rtype)
for rec in a:
    print rec

print a["column1"]
print a["column2"][0]
```

The output of this sequence is then

```
(1.0, array([[1, 2],
       [3, 4]]))
(2.0, array([[ 7,  8],
       [ 9, 19]]))
[ 1.  2.]
[[1 2]
 [3 4]]
```

Clearly, the entries of such a record array are writable too.

In our SPEC example the functions of our virtual SPEC module return such an array holding the tabular data of a single scan. For the construction of the PyTables table we need to know the names of the columns, their data type, and if necessary their shape. This can be achieved by the following code:

```
for d in a.dtype.descr:
    print "name: %s",d[0]
    t = n.dtype(d[1])
    print "type: %s",t.name
    #handle shape
    try:
        s = d[2]
        print "shape: ",s
    except:
        s = None
```

Which will produce the following output:

```
name: %s column1
type: %s float32
name: %s column2
type: %s int64
shape:  (2, 2)
```

Finally, the number of records in a NumPy array can be obtained from its shape attribute.

the following sections we assume that the spec module provides the following functions:

```
#return global SPEC file header as dict
spec.get_file_header(file)

#return scan data as NumPy record array
spec.get_scan_data(file,nr)

#return scan header as dict
spec.get_scan_header(file,nr)
```

In the sidebar you will find more information about NumPy record arrays. Since we assume that the EDF files contain only a single header and data block, the edf module provides only the following functions:

```
#return EDF header as dict
edf.get_header(filename)

#return EDF data as NumPy array
edf.get_data(filename)
```

Saving tabular data - SPEC

We assume now that we have already created the file structure discussed above. In the next step we will create the table object holding the data stored in a single SPEC scan. We will call this table *data* by convention. The fist step in creating a table object is to create its descriptor, which is either a class of type `IsDescription` or a dictionary. If we use the class approach, we can defined a table descriptor like this:

LISTING 1

```
 1. import tables
 2. #import virtual spec module
 3. import spec
 4.
 5. #numpy record array with the data of a single scan
 6. data = spec.get_scan_data("sige_crystal.spec",2)
 7. scangroup = h5.root.filegroup.scangroup
 8.
 9. #create the dictionary describing the table
10. tab_desc_dict = {}
11. col_name_list = []
12. for d in self.data.dtype.descr:
13.     cname = d[0]
14.     col_name_list.append(cname)
15.     if len(d[1:])==1:
16.         ctype = numpy.dtype((d[1]))
17.     else:
18.         ctype = numpy.dtype((d[1],d[2]))
19.     tab_desc_dict[cname] = tables.Col.from_dtype(ctype)
20.
21. #create the table in the scan group
22. tab = h5file.createTable(scangroup,"data",tab_desc_dict,"scan data")
23.
24. #write data to the table
25. for rec in data:
26.     for cname in rec.dtype.names:
27.         tab.row[cname] = rec[cname]
28.     tab.row.append()
29.
30. tab.flush()
31.
```

```
import tables
class MyTable(tables.IsDescription):
    column1 = tables.Float32Col()
    column2 = tables.IntCol()
    column3 = tables.Float64Col()
```

As you see from the example, the attributes of `MyTable` define the column names and the type of each column that will make up the table.

Since the column names in a SPEC file can vary from scan to scan, we have to create the descriptor dynamically from the NumPy record array we obtain from the `get_scan_data()` function of the spec module. Listing 1 shows how to create a table from a record array and save its data to a table. In lines 12 to 19, the table descriptor dictionary is assembled from a NumPy record array. Again we see in line 19 how nicely NumPy is integrated into PyTables: the type of a column can be derived directly from the data type of a record arrays column. This makes it possible to keep the code for saving data to a table quite generic. In line 22 the table object is finally created. In lines 25 to 28, the data is written to the table using the Row object, accessing every column in each row by name just as you would address the contents of a dictionary. Once all data is written to a row, it will be appended to the table by calling its `append()` method. After appending all rows, calling `flush()` on the table object will free all temporarily used memory and ensure the integrity of the HDF5 file, even if the program crashes and the file is not closed regularly.

You may wonder how to access the data now that it's stored in a table. This is fairly simple. Calling the `read()` method of a table object will return the table's data as a NumPy record array. Alternatively, you can access the data of every column via the `cols` attribute of a table as shown below.

```
a = tab.read()
x = a["column1"]
y = a["column2"]

x = numpy.array(tab.cols.column1[:])
y = numpy.array(tab.cols.column2[:])
```

The objects obtained in this way are NumPy arrays and can therefore be used directly for further processing. The last approach to access column data, however, requires the column names to be valid Python identifiers (no blanks, dots, or leading digits).

We know already that SPEC scans can also contain multidimensional data like an MCA. This is not a problem for PyTables since table objects can handle columns whose data type is non-scalar too. In fact, we do not even have to change anything in listing 1, since we derive the data type of a column from its data type in the record array. That is pretty nice, isn't it? However, there is one thing which might be of interest. Tables holding array like columns will consume plenty of space on the hard drive. So this is a good time to introduce the compression features of PyTables. The only thing we have to do is create a compression filter and pass it as a keyword argument to the table creation method of the HDF5 file. It looks like this:

```
f = tables.Filter(complevel=5,complib="zlib")
tab = h5file.createTable(scangroup,"data",
                         tab_desc_dict,
                         "scan data",
                         filters=f)
```

" HDF5 is an open and binary file format developed by the NASC (National Association of Super Computing)"

Here, *complevel* sets the compression level and *complib* is the library used for compression. Again, if you want to stay compatible with native HDF5, only zlib should be used. The effect of compression can be tremendous. For instance, on the last ESRF experiment in Grenoble, the author converted a SPEC file holding 50 scans to HDF5. Using no compression the resulting HDF5 file was around 17 MB in size, while with compression the HDF5 file was only 1.3 MB (using zlib and a compression level of 9). Take note, though, that using compression will slow down the read/write performance.

Adding metadata in attributes

As we have seen, a SPEC file contains plenty of metadata. This data can be stored as attributes to a group, file, array or table object. The most general method to add an attribute to an object is to use the `setNodeAttr()` method of the file object.

```
globhead = spec.get_file_header(filename)

scan2 = h5.root.sige_crystal.scan2
```

```
for k in globhead.keys():
    h5.setNodeAttr(scan2,k,globhead[k])
```

The first argument of `setNodeAttr()` can be a path or an object, and it refers to the node to which the attribute should be attached. The remaining two arguments are the attribute's name and its value. To read an attribute from an object use the `getNodeAttr(where,attrname)` method of a file object. Another possibility to access a group attribute would be to use

```
scan2._v_attrs.date
```

Where _v_attrs is an object of type `AttributeSet` class. If you want to use this method, your attribute name must be a valid Python identifier. If the object you want to attach an attribute to is a table/array object, there is also an alternative method to read/write attribute data. We take a look at that in the next section.

In principle, every Python object can be saved to an object's attribute. If the attribute should be readable by an application that is not aware of PyTables or Python (for instance a native C application), the attribute data must be of type *bool*, *int*, *float*, *complex*, or *str*. These values are converted to native HDF5 data types. Furthermore, all NumPy arrays of such a type will be converted to arrays of native HDF5 data types and therefore can be read by native HDF5 applications as well. To ensure that attribute data is stored correctly, scalar data can be converted to NumPy scalar values and lists to NumPy arrays as well.

Saving array data - EDF

EDF data is of array type and should therefore be stored in the CCD group of every scan. Listing 2 shows the fundamental code to do this job. The data will be stored in a **CArray**, which allows compression. For the creation of a CArray we have to define an **Atom** object. Atoms are, in fact, what data types are in native HDF5 (see above). They define how the element that should be stored in an array should look. Though Atoms can be declared as being multidimensional, CArrays do not support Atoms at present.

In the case of array/table objects, an alternative approach is available to attach attributes:

```
ca.attrs.myattribute = numpy.int(1234)
```

where `numpy.int(1234)` ensures that the attribute is saved as a native HDF5 type.

Conclusion

PyTables is an easy to use interface to the powerful HDF5 library. Developers should think about using PyTables before inventing their own file format, especially if they need to store numeric data. A particular problem of this case study is that parsing the EDF and SPEC files is extremely tedious. The parser code, for instance, must ensure that header information is stored with the proper data type to the header dictionary to make attributes readable by native HDF5 applications. From this it follows that it would be better to store data directly into an HDF5 file rather then converting existing formats to HDF5. The x-ray and neutron scattering community has realized this problem and is actually developing a new file format called NEXUS based on HDF5 which has, at least in the opinion of the author, a very gentle design. One can hope that other scientific communities and hardware vendors will follow this example and therefore make reading their data files much more painless.

LISTING 2

```
 1. import tables
 2. import edf
 3.
 4. h5 = tables.openFile("test.h5",mode="a")
 5. g = h5.root.test.scan2.ccd
 6.
 7. #read CCD header and data
 8. data = edf.get_data(filename)
 9. hdr = edf.get_header(filename)
10.
11. #save the data
12. f = tables.Filters(complevel=5,complib="zlib")
13. a = tables.Atom.from_dtype(data.dtype)
14. ca = h5.createCArray(g,"ccddat",a,data.shape,
15.                      "CCD data frame",filters = f)
16. ca[...] = data[...]
17.
18. #save attributes
19. for k in hdr:
20.     h5.setNodeAttr(ca,k,hdr[k])
21.
22. h5.close()
23.
```

EUGEN WINTERSBERGER is a PHD student at the department of semiconductor physics at the Johannes Kepler university in Linz/Austria where he works in the x-ray diffraction group. Four years ago he started with Python - and became addicted to it immediately.

e-xact transactions

Payment Processing Technology

MOVING BUSINESS. REAL TIME. REAL SMART

They don't get IT!

- Banks do what banks do.
- Take your money.
- Use your money.
- Charge you to access your money.

We get IT!

- We build software.
- Our code works.
- Download via the Web.
- Bank Agnostic.
- 24 x 7 Test System

You get IT!

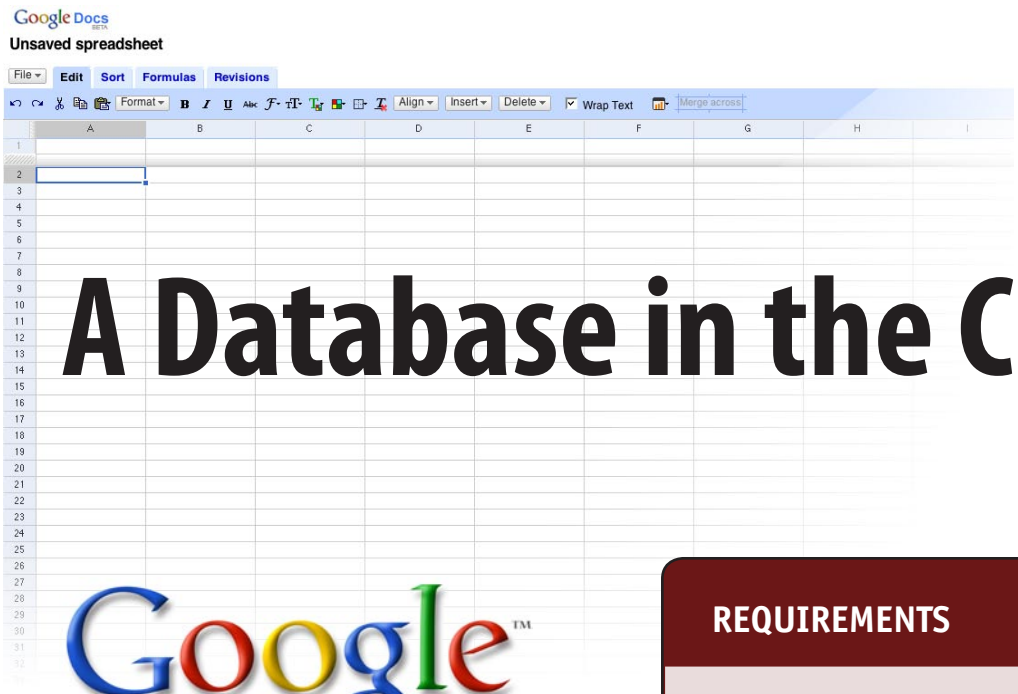
- **FREE**
- http:developer.e-xact.com
- Influence our developement

E-xact Transactions (Canada)
Suite 304 - 134 Abbot Street
Vancouver, BC, V6B 2K4
Canada

Contact Information
Toll Free: 1-877-303-9228
Ph: 604.691.1670
Fax: 604.691.1678

Web: www.e-xact.com
Email: comments@exact.com

Trading Symbol: EXZ.U
Listing: (CDNX)



A Database in the Cloud

Spreadsheets with Python

REQUIREMENTS

PYTHON: 2.x, Google Docs API - *http://code.google.com/apis/spreadsheets*

Useful/Related Links:

- Google Spreadsheets - *http://spreadsheets.google.com/*

by Jeff Scudder

It seems more and more data is moving into the Cloud. The Internet, that is. The benefits are manifold: less disk usage on your machine along with the added benefit that your data is accessible anywhere you can get on the Net. The next time you are writing an application, think about how happy your users would be if they could use it on any machine and still have the same configuration and data. There are, of course, a few drawbacks to storing everything in the Cloud: there is the problem posed by offline access, the problem of finding affordable online storage, and finally, the performance cost of fetching data from the web. While these limitations can present significant challenges, none of them is insurmountable. In this article, I will walk you through steps to create a remote storage solution using Python and Google Spreadsheets.

Why Google Spreadsheets?

Since this is Python Magazine, you probably won't question my first choice in technology, but why Google Spreadsheets? I have a few reasons in mind. First of all it is free; you can't beat that. Second, Google Spreadsheets keeps all of its data in the cloud, and it provides a web service interface for programmatic access (more on that later). Third, Google Spreadsheets has a simple user interface that many people are already familiar with. The user interface is more important than many people think for a database. Having full access and the ability to view/edit everything easily can drastically reduce time spent debugging.

My goal is to create an online database: something easy, quick, and free. When I hear database, the first thing I think of is an SQL database, so why not run one? Because using an SQL database involves not only running your own server, but also time spent defining schemas and relationships and managing the data with interfaces that are often less user-friendly than Google Spreadsheets. In a spreadsheet, you can use each worksheet as a table: the first row can define the table fields and each subsequent row will contain a single record. Another distinction from SQL is that a spreadsheet database will not necessarily be relational. Often all your application requires is a large collection of key-value records (something like a list of dictionaries, to use Python terminology). Hey that sounds like a pretty good objective. Why not create a hierarchy of dictionaries whose data is stored in Google Spreadsheets?

Talking to Google: AtomPub

The first thing we need to tackle is to figure out how we can get Python talking to Google Spreadsheets. Luckily enough, spreadsheets has a web services interface based on AtomPub. Through this web service you are able to upload files and convert them into Google Documents, find spreadsheets, read and modify the worksheets in a spreadsheet and change the contents of the cells in a worksheet. For specific details on the available functionality of this API, point your browser at http://code.google.com/apis/spreadsheets.

I mentioned that the Google Spreadsheets API is based on AtomPub, but what does this mean exactly? AtomPub is a REST style protocol for interacting with data using XML for the data format and HTTP for the transport and program interface. Google has extended AtomPub to add a few features and common mechanisms. The extensions are used to provide programmatic access for a growing number of Google services. Taken together, these services are called "Google Data APIs," and Google has released a set of open source libraries for interacting with these services in a variety of languages. For this project, we will focus on the Python library and use it as the foundation for our spreadsheet database. You can get the source code for the Google Data Python client library on Google Code in the open source project hosting section under the project name gdata-python-client.

The fact that the Google Spreadsheets API uses XML to express cells, worksheets, and other spreadsheet data shouldn't be something you need to worry about. Some of the objectives of the client libraries are to translate XML blobs into easy to use native class objects and to turn potentially complex HTTP calls into methods which abstract most of the requests going on behind the scenes. This makes the process of writing a spreadsheet database module quite a bit cleaner, but you will need to keep in mind that some of the method calls in the library can be expensive in terms of execution time because they involve making an HTTP request to a remote server.

First Steps

Let's jump into a basic design. Our spreadsheets database will have a three-tiered design consisting of databases, tables, and records. These three elements map nicely to spreadsheets, worksheets, and individual rows. Keeping this in mind, we will begin by trying to create a new database by creating a new Google Spreadsheet.

The process of creating a new Google Document is easy enough on http://docs.google.com, but we're going to try to manage the whole process from within the spreadsheets database library. Lets define a `DatabaseClient` class to manage the creation process.

The first thing we'll need to put into this class is the username and password for your Google Document's account so that it can authenticate with the Google Data servers. The spreadsheet creation step introduces one of the small complexities of working with the Google Documents APIs. It turns out that there are two web services which we will be working with in this library.

The first is the Google Document List API which allows you to manage your spreadsheets, word processing documents, and presentations, and the second is the Google Spreadsheets API, which allows you to read and modify the contents of Google Spreadsheet. Spreadsheet creation is only available through the Google Document

List API so we will begin in Listing 1 by creating a client for the Google Document List and storing it in our `DatabaseClient`.

In the listing, `SetCredentials()` call to `ProgrammaticLogin()` will perform an HTTPS request to exchange your username and password for an auth token which is used in all subsequent (unencrypted) requests. Now that we have authenticated with the Documents List server, we are ready to create a new database by uploading a Spreadsheet. The Document List API supports document creation by means of uploading an existing file from your computer. Since we're creating a new Spreadsheet to initialize an empty database, we can upload an empty file, or better yet, use **StringIO** to create a virtual file and avoid touching the local disk. One of the many types of files that the document list accepts is a .csv file so we can write a method to create a new spreadsheet that looks something like this:

```
def CreateDatabase(self, name):
    # Create a Google Spreadsheet to form the
    # foundation of this database by uploading a
    # file to the Google Documents List API.
    virtual_csv_file = StringIO.StringIO(',,,')
    virtual_media_source = \
        gdata.MediaSource(
            file_handle=virtual_csv_file,
            content_type='text/csv',
            content_length=3)
    db_entry = \
        self.__docs_client.UploadSpreadsheet(
                        virtual_media_source, name)
    return Database(spreadsheet_entry=db_entry,
                    database_client=self)
```

Database Classes

Notice we introduced a new class, a **Database**, which represents a Google Spreadsheet and abstracts interactions with the Google Spreadsheets servers. One of the most important pieces of information that we need to know about the Google Spreadsheet is it's ID key. The spreadsheet's key uniquely identifies it within the Google Documents domain and it is one of the components in each URL which is related to this spreadsheet. I mentioned that the Google Data APIs are RESTful web services and one of the concepts often used in RESTful protocol design is the identification of each resource with a unique URL. In all of the HTTP requests our library makes from here on out this spreadsheet key will become part of the target URL. We can abstract the spreadsheet key details from the users of this library within the Database class's constructor which look like this:

```
class Database(object):
  def __init__(self, spreadsheet_entry=None,
                     database_client=None):
    self.entry = spreadsheet_entry
    id_parts = spreadsheet_entry.id.text.split('/')
    self.spreadsheet_key = \
         id_parts[-1].replace('spreadsheet%3A', '')
    self.client = database_client
```

You may notice that I passed the `database_client` into the database object's constructor. The instance of the database will contain a reference to the client object so that it can perform operations against the Spreadsheets and Documents List server.

Now that we have a database, it's time to add a table. As mentioned above, tables correspond to worksheets, and these are manipulated using the Google Spreadsheets API. With the code we've seen so far, our database client only talks to the Google Documents List API, so in Listing 2, we've modified the `DatabaseClient` to authenticate with the Google Spreadsheets service as well.

Now our Database object can talk to the Google Spreadsheets to create, update, and delete new worksheets which serve as our database tables. When creating a table, the main things we care about are the table's name and the names of the columns it has. Our database will be untyped. For now everything will be stored as a string, so we don't need to declare types on our columns. With this design in mind, we can write a `CreateTable()` method in the Database class which would look like this:

```
class Database(object):
  ...

  def CreateTable(self, name, fields=None):
    worksheet = \
    self.client._GetSpreadsheetsClient().
AddWorksheet(
        title=name, row_count=1,
```

LISTING 1

```
1. import gdata.service
2. import gdata.docs.service
3.
4. class DatabaseClient(object):
5.
6.   def __init__(self, username=None, password=None):
7.     self.__docs_client = gdata.docs.service.DocsService()
8.     self.SetCredentials(username, password)
9.
10.  def SetCredentials(self, username, password):
11.    self.__docs_client.email = username
12.    self.__docs_client.password = password
13.    if username and password:
14.      try:
15.        self.__docs_client.ProgrammaticLogin()
16.      except gdata.service.CaptchaRequired:
17.        raise CaptchaRequired(
18.            'Please visit https://www.google.com/accounts/'
19.            'DisplayUnlockCaptcha to unlock your account.')
20.      except gdata.service.BadAuthentication:
21.        raise BadCredentials('Username or password incorrect.')
22.
```

```
            col_count=len(fields),
            key=self.spreadsheet_key)
        return Table(
            name=name,
            worksheet_entry=worksheet,
            database_client=self.client,
            spreadsheet_key=self.spreadsheet_key,
            fields=fields)
```

We can now create a new table by calling `CreateTable()`, which will yield a Table object. The Table class is a bit like the Database class, also containing a `database_client` so that it's methods can talk to the spreadsheets server when we want to add new records to the database. Our Table class so far is in Listing 3.

One item worth mentioning is the `SetFields()` method in the Table class. This method sets the column headers in the worksheet to the values sent in the fields list. With the above code, the library will make a separate HTTP request for each column, so this can be a bit slow. We can speed this up by using a batch request and changing all of the required cells in one HTTP request, but for now we're going to stick with simpler code. Now that we can create tables, it's time to start adding records.

```
def AddRecord(self, data):
    new_row = \
    self.client._GetSpreadsheetsClient().
InsertRow(data,
       self.spreadsheet_key,
       wksht_id=self.worksheet_id)
    return Record(content=data, row_entry=new_row,
        spreadsheet_key=self.spreadsheet_key,
        worksheet_id=self.worksheet_id,
        database_client=self.client)
```

The data passed in to the `AddRecord()` method is a dictionary which maps the field contents in the row to the column they belong under. We have also introduced the Record class (in Listing 4) and by now you might be able to guess how this class is structured.

Now we're getting somewhere! With the above code we should be able to create a new spreadsheet, define some tables, and populate them all from within our application so let's try it out.

If you've never taken a look at Google Documents before, point your browser at *http://docs.google.com* and we'll begin. After importing the above module, try running the following in the interpreter while keeping one eye on your Google Docs. With any luck, you'll be able to see the changes you are making to your spreadsheet magically appear in your browser. First we'll create a new database.

```
client = google_spreadsheets_db.DatabaseClient()
client.SetCredentials('email@example.com',
                      'mypassword')
```

LISTING 2

```
1. import gdata.service
2. import gdata.docs.service
3. **import gdata.spreadsheet.service**
4.
5. class DatabaseClient(object):
6.
7.   def __init__(self, username=None, password=None):
8.     self.__docs_client = gdata.docs.service.DocsService()
9.     self.__spreadsheets_client = (
10.        gdata.spreadsheet.service.SpreadsheetsService())
11.    self.SetCredentials(username, password)
12.
13.  def SetCredentials(self, username, password):
14.    self.__docs_client.email = username
15.    self.__docs_client.password = password
16.    self.__spreadsheets_client.email = username
17.    self.__spreadsheets_client.password = password
18.    if username and password:
19.      try:
20.        self.__docs_client.ProgrammaticLogin()
21.        self.__spreadsheets_client.ProgrammaticLogin()
22.      except gdata.service.CaptchaRequired:
23.        raise CaptchaRequired('Please visit https://www.google.com/
accounts/'
24.                              'DisplayUnlockCaptcha to unlock your
account.')
25.      except gdata.service.BadAuthentication:
26.        raise BadCredentials('Username or password incorrect.')
27.
```

LISTING 3

```
1. class Table(object):
2.
3.   def __init__(self, name=None, worksheet_entry=None,
4.                database_client=None, spreadsheet_key=None, fields=None):
5.     self.name = name
6.     self.entry = worksheet_entry
7.     id_parts = worksheet_entry.id.text.split('/')
8.     self.worksheet_id = id_parts[-1]
9.     self.spreadsheet_key = spreadsheet_key
10.    self.client = database_client
11.    fields = fields or []
12.    if fields:
13.      self.SetFields(fields)
14.
15.  def SetFields(self, fields):
16.    self.fields = fields
17.    i = 0
18.    for column_name in fields:
19.      i = i + 1
20.      self.client._GetSpreadsheetsClient().UpdateCell(1, i, column_name,
21.          self.spreadsheet_key, self.worksheet_id)
22.
```

LISTING 4

```
1. class Record(object):
2.
3.   def __init__(self, content=None, row_entry=None, spreadsheet_key=None,
4.       worksheet_id=None, database_client=None):
5.     self.entry = row_entry
6.     self.spreadsheet_key = spreadsheet_key
7.     self.worksheet_id = worksheet_id
8.     self.row_id = row_entry.id.text.split('/')[-1]
9.     self.client = database_client
10.    self.content = content or {}
11.    if not content:
12.      self.__ExtractContentFromEntry(row_entry)
13.
14.  def __ExtractContentFromEntry(self, entry):
15.    self.content = {}
16.    if entry:
17.      for label, custom in entry.custom.iteritems():
18.        self.content[label] = custom.text
19.
```

```
database = client.CreateDatabase(
                    'My Test Database')
```

If you look at your document's list you should now see a spreadsheet named "My Test Database" which contains a single empty worksheet. Next we'll create a table and open the corresponding worksheet.

```
friends_table = database.CreateTable('My Friends',
              ['name', 'email', 'address', 'phone'])
```

Now your spreadsheet should have a worksheet named "My Friends" with the field labels in the first row. There is one important caveat when creating your table, the Google Spreadsheets List Feed, which this library uses, converts the column names from the original input values. This sometimes catches users off guard, so it warrants a brief explanation which you can skip if you'd like. As long as you use all lowercase strings with no spaces or special characters, you don't need to worry about the conversion. Everything will stay the same.

For those who are interested, the Google Spreadsheets List Feed uses the first row of the spreadsheet to create custom XML elements whose names correspond to the column names provided in the worksheet. To really understand this we need to look at the XML which is sent and consumed by the Google Spreadsheets API when you are dealing with a worksheet as a collection of rows. The XML for a row in the "My Friends" table we've created above would look something like Listing 5.

Please note the <code><gsx:...></code> tags in the above entry. The local names for these XML tags are derived from the contents of the first row in the worksheet. In order to ensure that these tags are valid XML, the contents of the first row are transformed by removing spaces, stripping leading numbers, and changing all characters to lowercase. There are a few oddities in this conversion process at the time of this writing, and the rules for conversion may be changing in the near future. For simplicity's sake, and to avoid any potential confusion, I recommend writing your column names in lowercase without spaces or special characters.

Using Your Database in the Cloud

Now that we've created our table, we can start adding data. Keep an eye on the worksheet as you are adding these records because the contents of the spreadsheet may update as the changes are pulled in to your browser.

```
friends_table.AddRecord(
        {'name':'Doug',
         'email':'doug@example.com',
         'address':'123 Imagination Way',
         'phone':'555-555-1289'})
friends_table.AddRecord(
        {'name':'Dave',
         'email':'dave@example.com',
         'phone':'555-555-1289'})
friends_table.AddRecord(
        {'address':'2 High st.',
         'name':'Fred',
         'phone':'555-555-0923'})
```

Now your data has been saved to the database in the cloud for retrieval in the future. The next time we fire up our application, we will want to begin with the data set we've already created, so we need a way to lookup existing spreadsheets, worksheets, and rows. We'll start by finding our database aka Google Spreadsheet. Within Google Documents, it is possible for one person to have many documents with the same name, so it is conceivable that searching for our database by name could give us multiple results. Each spreadsheet has a unique ID, however, so we will also allow searching by the spreadsheet key. We'll add a new method to the DatabaseClient in Listing 6.

In the above method we always return a list of Databases. You probably want to retrieve the only

LISTING 5

```
<entry>
    <id>...</id>
    <title type='text'>joe</title>
    <content type='text'>email: joe@example.com, address: 123 Oak
Street, phone: (555)555-1387</content>
    ...
    <link rel='edit' type='application/atom+xml' href='http://
spreadsheets.google.com/feeds/list/.../3aaddgi3nmbn01' />
    <gsx:name>joe</gsx:name>
    <gsx:email>joe@example.com</gsx:email>
    <gsx:address>123 Oak Street</gsx:address>
    <gsx:phone>(555)555-1387</gsx:phone>
</entry>
```

LISTING 6

```
class DatabaseClient(object):

  # ...

  def GetDatabases(self, spreadsheet_key=None, name=None):
    if spreadsheet_key:
      db_entry = self.__docs_client.GetDocumentListEntry(
          r'/feeds/documents/private/full/spreadsheet%3A' +
          spreadsheet_key)
      return [Database(spreadsheet_entry=db_entry,
                       database_client=self)]
    else:
      title_query = gdata.docs.service.DocumentQuery()
      title_query['title'] = name
      db_feed = self.__docs_client.QueryDocumentListFeed(
                    title_query.ToUri())
      matching_databases = []
      for entry in db_feed.entry:
        matching_databases.append(Database(spreadsheet_entry=entry,
                                           database_client=self))
      return matching_databases

```

Database in the list, but you might need to check additional information if you have multiple matches. Next we need to be able to find a table in a database and this method looks very similar to the above for finding a database. Within a single spreadsheet, the name and the worksheet ID are both unique as no two worksheets have the same name. I won't include the code for `GetTables()` since it is so similar. Please refer to the gdata-python-client open source project for the complete library code. Within a table, we want to be able to find specific rows. The simplest form of retrieval is to ask for a single row by number but it is also possible to retrieve by the row's unique id. Both retrieval methods are shown in Listing 7.

Retrieving a row by index may seem very foreign compared to an SQL-style database, where there is no concept of intrinsic ordering. However, a spreadsheet is naturally ordered, with rows beginning at index 1 and continuing until the first blank line of the spreadsheet. An important note: the Google Spreadsheets List Feed will only access and modify the rows from the start of the worksheet until the first blank row. This can allow you to put additional data in your spreadsheet after the first blank row. For example: you could list budget items, leave a blank row, then have formulas and totals at the bottom of the spreadsheet. Any rows that you insert will be placed just above the first blank row, so your calculations will be pushed down.

Getting just one row at a time can be a bit slow since it requires a full HTTP round trip for each request. Using the Google Spreadsheets API, we can ask for several rows at once by using the *start-index* and *max-results* parameters. You may notice that we used these same parameters in `GetRecord()` to restrict the number of results

" Retrieving a row by index may seem very foreign compared to an SQL-style database, where there is no concept of intrinsic ordering. However, a spreadsheet is naturally ordered, with rows beginning at index 1 and continuing until the first blank line of the spreadsheet."

LISTING 7

```
def GetRecord(self, row_id=None, row_number=None):
    if row_id:
      row_entry = self.client._GetSpreadsheetsClient().GetListFeed(
          self.spreadsheet_key, wksht_id=self.worksheet_id,
          row_id=row_id)
      return Record(content=None, row_entry=row_entry,
           spreadsheet_key=self.spreadsheet_key,
           worksheet_id=self.worksheet_id, database_client=self.client)
    else:
      row_query = gdata.spreadsheet.service.ListQuery()
      row_query.start_index = str(row_number)
      row_query.max_results = '1'
      row_feed = self.client._GetSpreadsheetsClient().GetListFeed(
          self.spreadsheet_key, wksht_id=self.worksheet_id,
          query=row_query)
      if len(row_feed.entry) >= 1:
        return Record(content=None, row_entry=row_feed.entry[0],
            spreadsheet_key=self.spreadsheet_key,
            worksheet_id=self.worksheet_id, database_client=self.client)
      else:
        return None

```

LISTING 8

```
def GetRecords(self, start_row, end_row):
    start_row = int(start_row)
    end_row = int(end_row)
    max_rows = end_row - start_row + 1
    row_query = gdata.spreadsheet.service.ListQuery()
    row_query.start_index = str(start_row)
    row_query.max_results = str(max_rows)
    rows_feed = self.client._GetSpreadsheetsClient().GetListFeed(
        self.spreadsheet_key, wksht_id=self.worksheet_id,
        query=row_query)
    return RecordResultSet(rows_feed, self.client, self.spreadsheet_key,
        self.worksheet_id)
```

LISTING 9

```
class RecordResultSet(list):

  def __init__(self, feed, client, spreadsheet_key, worksheet_id):
    self.client = client
    self.spreadsheet_key = spreadsheet_key
    self.worksheet_id = worksheet_id
    self.feed = feed
    list(self)
    for entry in self.feed.entry:
      self.append(Record(content=None, row_entry=entry,
          spreadsheet_key=spreadsheet_key, worksheet_id=worksheet_id,
          database_client=client))

  def GetNext(self):
    next_link = self.feed.GetNextLink()
    print 'next_link', next_link, 'href', next_link.href
    if next_link and next_link.href:
      new_feed = self.client._GetSpreadsheetsClient().Get(
          next_link.href,
          converter=gdata.spreadsheet.SpreadsheetsListFeedFromString)
      return RecordResultSet(new_feed, self.client,
          self.spreadsheet_key, self.worksheet_id)
```

to a single row. Let's write a method that accepts the starting and ending row numbers to get a series of rows in one HTTP request. You can see the example in Listing 8.

At the end of the above method, I'm returning a `RecordResultSet`, which is a new class that I've introduced to handle the special case of record collections that are part of a whole. Any time that you retrieve a feed from the Google Spreadsheets API, you are also given a "next" link which shows you the URL to fetch if you want to iterate over the entire collection. If you ever wanted to examine the entire spreadsheet, a `GetNext()` method in the RecordResultSet could make this task easier. The record result set might look something like Listing 9.

Being able to iterate through all of the records in our database may be useful in some cases, but usually your application is interested in querying the database (SQL does stand for Structured Query Language after all). The Google Spreadsheets API supports a query syntax which uses value comparisons on column names. For example, you can find all rows in a worksheet where the contents of the "hours" column is less than the value 5 (hours<5). Supported query operations include 'less than', 'greater than', 'exactly equal to', 'not equal', 'and', and 'or'. Notable query features that may be familiar from SQL such as joins and partial string matching aren't supported, so this might affect your design. I have to remind myself from time to time that this is a spreadsheet and I'm trying to keep the database design true to its simple roots. For simple uses the available query features are hopefully all you need.

For this library, we should provide a method in the Table class to allow searches within the worksheet for rows that match an expression. We take care of that in Listing 10.

This method will give us a list of matching rows and will accept an expression as a string. To search the worksheet to see which people in your employees table have worked overtime you could call this new method with 'hours > 40', and to see how many hours Todd has worked, you could use `'name == Todd'`. These queries work with dates as well. Google Spreadsheets will interpret the strings that you insert into your cells, so if you send something that looks like a floating point number, it will be treated as one. Math operations are not permitted in queries, but it is still possible to use formulas indirectly in your queries by creating calculated columns.

For example, you could add a column to calculates a percentage (`itemsperminute`) from input columns (`hours` and `itemsmade`) and use that column in your queries (`itemsperminute < 0.5`). This type of workaround lends itself particularly well to a spreadsheet.

Since this method uses the RecordResultSet, it is also possible to iterate through all of the records that match the query if there are a large number of them.

The Google Spreadsheets API has other query parameters which we have not explored yet, such as the orderby parameter. Using this parameter you can request that the results be returned sorted in ascending or descending order on a particular column.

For a full list of the query capabilities of the spreadsheets feeds, you can find more information on the reference guide for the Google Spreadsheets API.

Now that we can find the records we are interested in, it is time to tackle changing the data in a row. Since the record contains all of the information about the row, we can change it by modifying the content dictionary and sending the changes to the Google Spreadsheets server:

```
class Record(object):
  ...

  def Update(self):
    self.entry = self.client._
GetSpreadsheetsClient().UpdateRow(self.entry, self.
content)
```

That was pretty easy, but there is one thing that I haven't yet considered. Databases often have multiple users making changes and a Google Spreadsheet is not exception.

In fact, being able to collaborate with others on the same spreadsheet is my favorite part of using Google

LISTING 10

```
1. class Table(object):
2.   # ...
3.   def FindRecords(self, query_string):
4.     row_query = gdata.spreadsheet.service.ListQuery()
5.     row_query.sq = query_string
6.     matching_feed = self.client._GetSpreadsheetsClient().GetListFeed(
7.         self.spreadsheet_key, wksht_id=self.worksheet_id,
8.         query=row_query)
9.     return RecordResultSet(matching_feed, self.client,
10.        self.spreadsheet_key, self.worksheet_id)
```

LISTING 11

```
1. class Record(object):
2.   ...
3.
4.   def Push(self):
5.     self.entry = self.client._GetSpreadsheetsClient().UpdateRow(
6.         self.entry, self.content)
7.
8.   def Pull(self):
9.     if self.row_id:
10.      self.entry = self.client._GetSpreadsheetsClient().GetListFeed(
11.          self.spreadsheet_key, wksht_id=self.worksheet_id,
12.          row_id=self.row_id)
13.    self.__ExtractContentFromEntry(self.entry)
```

Spreadsheets. When you view the user interface in your browser, other collaborator's cursors are marked with a colored border, and you can see what other people have changed almost instantly.

Our humble library doesn't have a nice display like this and we need to query the server to see if other people have made changes. In Listing 11, to receive updates on the contents of a row, we'll implement a pull method and, for consistency's sake, we will rename the method to send our changes to the server as Push.

Now you can perform a Pull on any record to update it to the latest values. What happens if you Push on a record that has changed? The Spreadsheets server will send a 409 conflict error which will raise an exception. The Google Spreadsheets API, and other Google Data APIs, rely on a design called optimistic concurrency for handling write conflicts. Each Atom entry which can be modified contains an 'edit' link with a special version hash. Changes to the entry can only be made if the request is sent to the correct version of the URL. Each time the content of the entry is changed, the 'edit' link's URL changes.

With the push and pull methods, the program using the library is required to handle these errors, examine the server's conflict response to find the new values for the record, and decide whether or not to resubmit the request.

Room for Improvement?

At this point, the library should be functional, but there may be some ideas that would improve usability. For instance, it might be helpful for the library to provide a method for updating records which would merge changes as long as there are no conflicts. So if you wanted to change the first column in a particular record and someone else had changed the second column since you last pulled the record, the library could merge the changes for you. When you send your changes to the server and receive a 409 conflict error, the merge method would handle the exception. Because the conflict refers to the record (the entire row) it might be possible that the changes are compatible and do not conflict. In that case they could be merged, but if there is a true conflict an exception would need to be raised so that the application could determine how to proceed.

With the current Record class, deciding which columns have been changed would be difficult because it doesn't track the state of the values in the content dictionary. Implementing a merge solution might introduce a bit of overhead for a small payoff, so I won't add it just yet.

There is another path that might be fun to go down to make some changes to this library.

Originally, I had stated that working with this spreadsheets database should be a bit like working with a list of dictionaries. While this was a more an analogy that a literal goal, there is nothing stopping us from making these classes look more like lists and dictionaries when using the code. For example, to change a record in a worksheet, you could use:

```
client.GetDatabases(name='database')[0].
GetTables('worksheet')[0].GetRecord(row_number=1).
content['name']
```

But we could make the Database, Table, and Record classes act more like Python collections, so that becomes:

```
client['database'][0]['worksheet'][1]['name']
```

The way to accomplish this would be to implement the __getitem__ method for the requisite classes such that `client['database']` is equivalent to `client.GetDatabases(name='database')`. The code might look like this:

```
class DatabaseClient(object):

  def __getitem__(self, x):
    return self.GetDatabases(name=x)
```

You could do the same for the other classes in this library. The only drawback to using this form for requests is that the short form does not support alternate methods to find the desired object.

For example, the `__getitem__()` function will only search for Databases by name, not by the spreadsheet's key. Another potential drawback, when you look at code which uses this compact form, you are probably less likely to anticipate the cost of getting these objects. Dictionary lookups are usually quite fast, but these `__getitem__()` calls will likely be slow because they involve remote HTTP requests. In some cases, clarity is more important than brevity, so I won't dedicate more time developing this idea, though you are free to modify this library yourself.

This library has been released as part of the open source gdata-python-client project, and you are free to download the code and give it a try. Before you move your entire data storage infrastructure to Google Spreadsheets, there are a few limitations that I should mention.

In addition to the remote nature of these spreadsheets and the potential time delays and issues associated with networking, there are also size limitations on Google

Spreadsheets. There are currently limits on the maximum number of rows, columns, cells, etc. which can be found on the Google Documents Help Center. These limits are subject to change, so hopefully they will expand but even now the limits should be adequate for certain use cases, like storing application data for a small application or storing some of a user's settings for greater portability. To be specific, the current limits as of this writing are "10,000 rows, or up to 256 columns, or up to 100,000 cells, or up to 40 sheets – whichever limit is reached first."

Conclusion

Perhaps this idea of using Google Spreadsheets as a database isn't quite what you were looking for, or maybe it has sparked ideas about other things you could do with a spreadsheet in the cloud. You might be interested to know that there is another means of interacting with Google Spreadsheets using the API which I haven't touched on. The cells feed allows you to modify individual cells, use Google Spreadsheet's rich formula system, and create batch requests which modify hundreds of cells in a single HTTP call. The cells feed might provide a tantalizing alternative if you found my library design too limited and I encourage you to learn more about it on code.google.com.

The Google Spreadsheets API will likely go through quite a few changes in the not too distant future, and the team has been taking a hard look at some of the use cases and limitations that I've mentioned in this article. Keep your eyes peeled.

JEFF SCUDDER works on the Google Data APIs at Google and has recently been helping out with the gdata-python-client library, which provides an easy to use wrapper for specific web services.

Welcome to Python

AVC: Simplifying your GUI Code

by Mark Mruss

GUI programming, like many other types of programming, can sometimes prove exhausting because you must repeat yourself over and over again. AVC is one tool available to Python GUI programmers that attempts to simplify things by synchronizing application data and GUI widgets.

Introduction

Every once in a while I find myself browsing the Internet trying to find out what's new and exciting in the Python world. Sometimes I browse to find topics for this article; other times mere curiosity draws me across the web. While I was browsing the other day, I stumbled across *AVC: the Application View Controller* [1]. I was immediately intrigued by it because its name is so similar to the Model View Controller (MVC) pattern. Being familiar with the Model View Controller pattern, and admittedly having struggled with it in the past, I decided to check out AVC to determine if it might be a viable alternative.

After reading about AVC I was intrigued for several reasons. The main reason was the promise of "a multiplatform, fully automatic, live connection among graphical interface widgets and application variables." [2] This means that graphical widgets can be connected

REQUIREMENTS

PYTHON: 2.2+

Other Software:
- AVC 0.5.0
- PyGTK 2.8 or later

Useful/Related Links:
- http://avc.inrim.it/html/
- http://faq.pygtk.org/index.py?req=all#1.1
- http://www.pygtk.org/downloads.html
- http://avc.inrim.it/doc/user_manual.pdf

to variables and automatically synchronized. One of the (many?) problems with Graphical User Interface (GUI) programming is that you often find yourself doing the same thing over and over again. One of the things that you end of doing over and over again is setting the contents of a widget based on the value of a variable, and then subsequently, setting that variable's value based on the current state of the widget. Whenever someone promises me an automatic connection between GUI widgets and my variables, I'm interested.

The other reason for investigating AVC is my past struggles with the Model View Controller pattern. I like the decoupling that the MVC pattern seems to provide, but I often feel as though I am needlessly duplicating code in order to stick with the pattern. I looked into AVC hoping to find a simple framework that will avoid this and guide the separation of my application data from my GUI logic. However in the end, I realized that AVC (in its current state), does not appear to achieve this across all widget toolkits. It does allow you to separate your application from much of the GUI logic that one would normally need, but it does not appear to allow you to fully separate the data from the GUI.

AVC is being developed by Fabrizio Pollastri, and is released under version 3 of the GPL. I was unable to find much information about the project's future or design objectives. But judging by the release notes, it is still being actively developed. The project is only at version 0.5 so don't be surprised if the API (Application Programming Interface) or some of the goals change. Although relatively new, AVC bears further investigation because it already simplifies your GUI programming and contains the potential to do much more. In this article, I will introduce some of the main concepts behind AVC, and provide a small, working example using AVC and PyGTK.

Installing AVC

AVC is a "Python module written in pure python" [3]. This means that AVC is easy to install and does not have any dependencies aside from Python 2.2 or greater. You can download the current version (0.5.0) from the website or, if you are a lucky Linux user like me, you can install via your handy package manger. If you download the source, you can install it after extraction, using the following command:

```
python setup.py install
```

Depending on which widget toolkit you plan on using, the following requirements must also be met:

- GTK+: PyGTK 2.8 - 2.10
- Qt: PyQt or PyQt4v3 - v4
- Tk: Tkinter 2.4
- wxWidgets: wxPython 2.6

You can easily install any of the above toolkits using your favourite package manager, a source archive, or a binary installer found on each toolkit's main website. Once AVC is installed, you can test the installation by importing the avc module from the interactive interpreter. If all goes well you will see something similar to the following:

```
$ python
Python 2.4.4 (#2, Jan  3 2008, 13:36:28)
[GCC 4.2.3 (Debian 4.2.2-4)] on linux2
Type "help" ( . . . ) for more information.
>>> import avc
>>>
```

Widget Support

In its current state, AVC does not support all widgets available across all toolkits. It does currently support the following widgets:

- Button
- Check Box
- Combo Box (Not supported in Tk)
- Entry
- Label
- Radio Button
- Slider
- Spin Button
- Status Bar (Only supported by GTK+ and wxWidgets)
- Text View
- Toggle Button.

The support of each of these widget types is then handled by specific widgets in each toolkit.

Name-Matching

In order to automatically synchronize variables and GUI widgets, AVC uses a name-matching technique. AVC supports two methods of name matching. The first method requires that the variable and the widget have the same name. For example, if you have a variable named `my_data` it will match with a widget also named

my_data. In other words, if a variable and a widget have the same name they will be connected and their values synchronized.

The second name-matching technique allows you to connect variables to one or more widgets. Widgets can only be connected to one variable, but variables can be connected to any number of widgets. Using the second name-matching technique, you have a match if the widget name contains a double-underscore (__), **and** the portion before the double-underscore matches the variable name. For example, if you have variable named my_numeric_value and you want to match it with a Label widget and an Entry widget, you do this by naming the widgets as follows:

```
my_numeric_value__label
my_numeric_value__entry
```

" Widgets can only be connected to one variable, but variables can be connected to (or synchronized with) any number of widgets."

Again, the text before the "__" must match the variable name, and the text after the "__" can be whatever you want to differentiate your widgets or ensure uniqueness (if the toolkit requires it).

A GTK+, PyGTK, and AVC Example

For this AVC example, I will use the GTK+ widget toolkit and **PyGTK**. PyGTK a "is a set of bindings to the GTK+ user interface toolkit for the Python language"[4]. This means that it lets us write GUI applications in Python using the GTK+ widget toolkit. To avoid any flame wars: I did not choose GTK+ because it is the best toolkit but because it is the toolkit with which I am the most familiar. In order to run this example you will need to have PyGTK 2.8 or later installed. You can get the latest version of PyGTK from the PyGTK website. [5]

As I mentioned earlier, AVC works with many different widgets types. It also has many different features. I won't be covering all of the features or widgets types in this column; if you are interesting in learning about some of these features or how to work with different widgets types, please see the AVC documentation. [6]

This example is relatively simple, but it should illustrate some of the basic features of AVC. The example application asks for a person's name and then displays that name in pop-up dialog. It will have the following features:

1) A gtk.TextView widget to display the person's name and allow the name to be edited. The gtk.TextView widget will be synchronised with a variable.

2) A gtkButton that will pop-up a gtk.MessageDialog to show the person's name.

3) A gtk.Button that will reset the name to the name with which the application was initialized.

To begin we need to import all of the modules necessary:

```
import gtk
import avc.avcgtk
```

Note: I am using a gtk.TextView widget instead of a gtk.Entry widget. In its current state, AVC does not propagate changes in gtk.Entry widgets back to the connected variable. I'm not sure if this is a current limitation or part of the design.

The above code imports the PyGTK module necessary

LISTING 1

```
1. def init_gui(self):
2.
3.     #Setup the window and the layout manager
4.     self.window = gtk.Window()
5.     self.window.connect("destroy", gtk.main_quit)
6.
7.     self.h_layout = gtk.HBox()
8.     self.window.add(self.h_layout)
9.
10.    #create the widgets
11.    self.text_label = gtk.Label("Say hello:")
12.    self.h_layout.add(self.text_label)
13.
14.    self.frame = gtk.Frame()
15.    self.h_layout.add(self.frame)
16.    self.scrolled_window = gtk.ScrolledWindow()
17.    self.frame.add(self.scrolled_window)
18.
19.    self.text_view = gtk.TextView()
20.    self.text_view.set_name("name__text_view")
21.    self.scrolled_window.add(self.text_view)
22.
23.    self.hello_button = gtk.Button("Hello!")
24.    self.hello_button.connect("clicked", self.on_hello_clicked)
25.    self.h_layout.add(self.hello_button)
26.
27.    self.reset_button = gtk.Button("Reset")
28.    self.reset_button.connect("clicked", self.on_reset_clicked)
29.    self.h_layout.add(self.reset_button)
30.
31.    self.window.show_all()
```

LISTING 2

```
def on_hello_clicked(self, widget):
    def close(dialog, response):
        #Close the dialog on ok
        dialog.destroy()
    message_dlg = gtk.MessageDialog(self.window
        , 0
        , gtk.MESSAGE_INFO
        , gtk.BUTTONS_OK
        , ("Hello: %s" % self.name))
    message_dlg.connect("response", close)
    message_dlg.run()

```

LISTING 3

```
#! /usr/bin/env python
import gtk
import avc.avcgtk

class gtkAVC(avc.avcgtk.AVC):

    def __init__(self, name=""):
        #set the variable and save the initial name
        self.initial_name = name
        self.name = name
        #setup GUI
        self.init_gui()

    def init_gui(self):

        #Setup the window and the layout manager
        self.window = gtk.Window()
        self.window.connect("destroy", gtk.main_quit)

        self.h_layout = gtk.HBox()
        self.window.add(self.h_layout)

        #create the widgets
        self.text_label = gtk.Label("Say hello:")
        self.h_layout.add(self.text_label)

        self.frame = gtk.Frame()
        self.h_layout.add(self.frame)
        self.scrolled_window = gtk.ScrolledWindow()
        self.frame.add(self.scrolled_window)

        self.text_view = gtk.TextView()
        self.text_view.set_name("name__text_view")
        self.scrolled_window.add(self.text_view)

        self.hello_button = gtk.Button("Hello!")
        self.hello_button.connect("clicked", self.on_hello_clicked)
        self.h_layout.add(self.hello_button)

        self.reset_button = gtk.Button("Reset")
        self.reset_button.connect("clicked", self.on_reset_clicked)
        self.h_layout.add(self.reset_button)

        self.window.show_all()

    def on_hello_clicked(self, widget):
        def close(dialog, response):
            #Close the dialog on ok
            dialog.destroy()
        message_dlg = gtk.MessageDialog(self.window
            , 0
            , gtk.MESSAGE_INFO
            , gtk.BUTTONS_OK
            , ("Hello: %s" % self.name))
        message_dlg.connect("response", close)
        message_dlg.run()

    def on_reset_clicked(self, widget):
        #Reset the name to the initial name
        self.name = self.initial_name

if (__name__ == "__main__"):
    gtk_avc = gtkAVC("Mark Mruss")
    gtk_avc.avc_init()
    gtk.main()

```

to work with GTK+. It also imports the avcgtk module, which is used to link our "application variables" with our GTK+ GUI. Depending on what widget toolkit you use, you will have to use different modules. For example, if you are working with Qt4, you will import:

```
import avc.avcqt4
```

The next step is to create the application class. This will be the class that contains the data and application logic. In this example it will also create the GUI:

```
class gtkAVC(avc.avcgtk.AVC):
```

Notice that we derive our application from the avcgtk module's AVC class. This is the GTK+ specific AVC class with which we will work. We will also create an `__init__()` method in the gtkAVC class. This is where we will initialize the data that will be connected to the GUI and create the GUI via the `init_gui()` method:

```
def __init__(self, name=""):
    #set the variable and save the initial name
    self.initial_name = name
    self.name = name
    #setup GUI
    self.init_gui()
```

What we have here is pretty simple. We have the optional parameter name that we store as the current name and as the initial name. We then call the `init_gui()` method that will create the GUI. The contents of the `init_gui` method can be seen in Listing 1. I won't explain most of this code since the majority of it is PyGTK specific. However I will make note of several relevant segments below.

The only AVC specific code segment in the `init_gui` function is where we set the name of the `gtk.TextView` widget:

```
self.text_view.set_name("name__text_view")
```

This is all the code that we need to connect the `gtk.TextView` widget with the name variable that we created at the beginning of the `__init__()` function. (Now, how's that for simple?) Notice that the name section of the "name__text_view" string will match with the `self.name` variable. It is also important to note that the variable and the widget are not yet connected. They will become connected when the `avc_init()` method is called.

The other non-AVC related code of note is the connection of both `gtk.Button` widget's clicked signals with signal handlers. We connect the "hello" button to the

on_hello_clicked() method and the "reset" button to the on_reset_clicked() method.

That's it for the init_gui() function. Now let's look at the two signal handlers. The on_hello_clicked() method, shown in Listing 2, simply displays the name in a gtk.MessageDialog. What should be interesting to GUI programmers about this function is that instead of reading the data from the gtk.TextView widget, we simply reference self.name for the dialog's text. We can do this because self.name and the gtk.TextView widget are connected and their values are always the same.

The on_reset_clicked() function is also very simple and straightforward. We simply reset the name to the initial name:

```
def on_reset_clicked(self, widget):
    #Reset the name to the initial name
    self.name = self.initial_name
```

Again, we don't have to worry about doing anything to the gtk.TextView widget that displays the current name. By simply modifying the data, what is displayed in the GUI will automatically be updated by AVC.

The final step in our test application is to create class instances and actually show the GUI:

```
if (__name__ == "__main__"):
    gtk_avc = gtkAVC("Mark Mruss")
    gtk_avc.avc_init()
    gtk.main()
```

Since I am working in a Python file, I perform a simple test to make sure that the file is being launched directly instead of being imported as a module. I then create an instance of the gtkAVC class setting the name to be my name. The avc_init() method is then called (which is a member of the avc.avcgtk.AVC class). The avc_init() method is where all of the magic happens. It is where AVC goes through an application's variables and connects them to the GUI widgets based upon the name matching technique explained earlier. avc_init() must be called **after** all of the data in the application and the widgets have been created.

The final step in this example is to run the gtk.main() function. This is the "main loop" of our PyGTK based

FIGURE 1

application and is standard when working with PyGTK.

The entire code can be seen in Listing 3. When you run the code you will be greeted with something similar to Figure 1. It may not seem like much, but what we have here is a fully functional AVC application. This simple application illustrates how easy it is to create an automatic connection between your data and your GUI.

Benefits and Limitations

For a project that is little more than one year old (the first release was in January 2007), AVC appears quite stable and ready to be used. This is not to say that AVC is perfect. I did encounter a few issues.

The most notable issue that I encountered is the tightly coupled structure that AVC seems to impose on your application. For a module that tries to simplify the relationship between application data and GUIs, this feels like a bit of a problem. For example, when testing using PyGTK I was able to easily separate the GUI and the application's logic and data into separate classes. But when I tried to do the same thing using PyQt4, I encountered an error because the `avc_timer()` function in the `avc.avcqt4.AVC` module expects that the application class is itself a `QObject` (namely, the actual `QApplication`). I find this counterintuitive since being able to separate the data from the GUI seems to be a goal of AVC and similar tools. If you are able to fully separate the application logic and data from the GUI, you can easily connect your application code to different GUIs, and (by changing the AVC base class) even different widget toolkits.

I would also be interested in the ability to connect more types of data to more types of widgets. For example, I am interested in synchronizing Python lists with the contents of List boxes and Combo boxes. This is in addition to the current ability to synchronize the selection in the two widgets with integers.

Of course AVC also has a lot going for it. For one, I really like that AVC works with different widget toolkits. I have not seen this very often. It is a real benefit, especially when it comes to people adopting AVC. Hopefully in the future AVC will allow for further separation of the data and the GUI, truly enabling us to use different GUI's and different widget toolkits with our data.

What I like most about AVC is its simplicity. I have played around with similar solutions and AVC is one of the leaders in terms of ease of use. I was able to get create a simple AVC based application in almost no time at all. (The time that it did take was spent trying to remember PyGTK.) Much of this has to do with the streamlined nature of AVC. There is very little to configure regardless of what widget toolkit you use. While this simplicity limits what AVC can currently achieve compared to other solutions, it also makes the barrier to entry low and the usage trivial. User friendliness is certainly a good thing in a package that aims to make GUI application development easier.

Conclusion

While the example we looked at in this column wasn't that complicated (a simple string), I hope that it illustrates how helpful AVC could be if you were working with a large amount of data or data that could be updated by an external source. Instead of having to worry about what widgets or toolkits the data is being displayed in, AVC lets us work with the data and GUI transparently, leaving the synchronization to AVC.

If what AVC offers sounds interesting to you, download it from the AVC website and try it out. Although it it's not there just yet, with enough attention, usage, and contributions from the Python community, AVC should reach full-feature status in no time.

FOOTNOTES

[1] http:avc.inrim.it/html/
[2] http:avc.inrim.it/html/
[3] http:avc.inrim.it/html/
[4] http:faq.pygtk.org/index.py?req=all#1.1
[5] http:www.pygtk.org/downloads.html
[6] http:avc.inrim.it/doc/user_manual.pdf

For the last seven years MARK MRUSS has worked as a software developer, programming in the much maligned C++. In 2005 Mark decided it was time to add another language to his arsenal. After reading Eric Raymond's well known article "Why Python?" he set his sights on the inviting world of Python.

Random Hits

Notes from the Outback

by Steve Holden

This month Steve speculates about the PyCon that will already be past by the time you read this column, discusses Python 3.0 progress, eclectically surveys the news from the Python world and discusses Python's two best-known web frameworks.

Post-PyCon Pre-PyCon Notes

As I write this column, the PyCon issue of *Python Magazine* has already been set and printed, and will be handed out at the conference in a week's time. This being the magazine world, however, PyCon hasn't yet started, and so I cannot yet give you my impressions of the conference. I can say with confidence, however, that this is the biggest PyCon ever, and also the biggest Python conference that has ever been held in the world. Right now the registration website is showing 954 registrations, so it is possible that with walk-ins we will see over 1000 registrations!

It seems amazing that the conference has arrived at this stage in so short a time – only five years since we started in Washington, DC with less than 250 delegates. PyCon has always been a three-day conference, but now it is preceded by a day of tutorials that have been eagerly snapped up and which are almost all completely full. Sprinting has now been established as an activity to follow the conference, and is standardized at four days, so the whole event can now occupy over a week, though only a few hardy souls stay that long.

As always, the food has been the subject of heated debate. Python has always tried to accommodate a wide variety of dietary preferences, and this year it is even

offering halal and kosher meals. Although the number of declared vegetarians is known, there was a suspicion that the vegetarian food might look more attractive than the other options, even to people who didn't express a vegetarian preference. So, if you're a vegetarian and you were faced with a plate of meat, I'm sorry. Blame the omnivores.

This year will see the most serious efforts yet to record PyCon in both sound and video. There have been many attempts to make the talks available to people who did not have the chance to get to PyCon, but so far video has not been successfully produced. Perhaps this year will be the first time you will be able to see moving pictures from the conference, though Andrew Kuchling has done a magnificent job providing podcasts of the 2007 sessions.

" Now most people seem to be comfortable with the idea that transition to 3.0 is best done by migrating first to Python 2.6."

My own tutorial is a little ambitious: I am aiming to teach the language to programmers in three hours. I am also arranging an Open Space session where the Twisted community gets to teach me how to use Twisted. I expect I'll need a drink after that – those guys are really sharp. All in all, though, I expect that the experience of PyCon will (as always) be worth the loss of sleep.

Python 3.0 Progress

This PyCon is special in other ways too. It is being held in the same year that we expect to see the first release of Python 3.0, which has just been released in its third alpha incarnation and is due for final release in August. It will be interesting to see if the Python community is better informed about Python 3.0 than it was last year, when there was a good deal of anxiety about how to make the transition. Now most people seem to be comfortable with the idea that migration to 3.0 is best done by migrating first to Python 2.6. Once you have reached that goal, you can switch on Python 3 warning mode and heed all the warnings it gives you about constructs that will not easily translate to Python 3.

Once the warnings have been eradicated, you can use the **2to3** migration tool to perform an automated translation to Python 3.0. You should continue to maintain your code as translatable Python 2.6, since that way you can target both language platforms using the same source. There is no automated 3to2 translator, and so moving to Python 3.0 (or its successors) effectively represents a burning of the boats and an abandonment of the Python 2 world. Considering that the developers have specifically allowed themselves to break backward compatibility, this seems like a reasonable story for migration to the new language if it works out as anticipated. Early experience seems encouraging.

The Python World is Busy

It will hardly be news by now that Sun Microsystems have increased their support for dynamic programming languages by hiring Frank Wierzbicki and Ted Leung to increase the visibility of Python inside and outside their organization. Frank has been one of the most active participants in the Jython development world for some time now, and has brought the language past a significant hurdle by implementing the new style classes required for 2.2. I first met Ted when he came to PyCon in DC with Mitch Kapor in the early days of the Chandler project. He is active in both the Python and Apache worlds, and I am sure he will be a most effective ambassador for Sun.

User group activities remain high, with groups meeting in Southern California, Atlanta, Bangalore and Chicago to name just a few places. The PSF is keen to help new user groups start up, so get in touch if you

PyCon 08: Day 1

need assistance. There are also useful pages in the PSF wiki to help you with ideas.

The Python bug day seems to be coming along nicely, and is now happening with some regularity. This is an activity that many more people could help with, and the developers have started to tag possible beginner-level activities in the issue tracker. You can take part in bug days remotely, and getting together with a few other people and calling in over IRC is a nice way to mix social activity with improvements to the code base and documentation.

"Maybe Chicago will indeed have produced the first kiloPythonista by the time you read this."

A quick glance at Planet Python shows furious activity from many different contributors. Brett Cannon wrote about the work involved in being the Sprint Chairman for PyCon, showed us how to use the Google social graphs API, and took a poke at Java; James Tauber advertised for collaborators on various Django projects; Guido van Rossum announced the second alpha release of Python 3.0 (followed by the third alpha less than three weeks later); Sean McGrath shared his experiences installing Ubuntu 7.10 on a ThinkPad; as the indefatigable Doug Hellmann continues his Python Module of the Week series I wonder where he finds the time. This level of activity represents a very good indication of the state of Python's health.

Web Frameworks

Over the past year I've been taking a look at the two best-known Python Web frameworks – TurboGears and Django. Both have their advantages, and I have to say I am impressed with the capabilities of both systems.

I started my investigation by taking the website from the final chapters of *Python Web Programming* and trying to implement it in TurboGears. I was surprised how easy it was to translate the functionality; the main problem being that the site was not originally built using a templating engine. When I did my original experiments TurboGears was just moving from the Kid engine to Genshi. Since a lot of the HTML in the site was not standards conformant I had a great deal of trouble debugging the templates I had to write, and I found that Kid did not give me as much assistance as I would have liked in doing this, making error messages difficult to interpret and act upon.

Genshi was much more forgiving, and once I started using that I made progress more quickly. The progress stopped when I had to try and model the way I had been saving state in the original Web. I have to emphasize that this was in no way TurboGears' fault, as the original site created a long-lived object for each page. Consequently, state management was, shall we say, a little idiosyncratic.

The ideal way to continue this experiment would have been to try the same task with Django. So, just to be helpful, I did something else completely. By the time I got round to trying Django the holdenweb.com site was in serious need of renovation, so I decided to bite the bullet. I rented space on a virtual private server and started to re-implement the site in Django. I am happy to say the new site went live two weeks later, although it did take me another two weeks to sort the e-mail out (a task which I had been looking forward to with slightly less pleasure than having my toenails pulled out by the roots).

I was particularly impressed by the ease with which I could implement the site's look and feel on all the body pages using Django's template inheritance features. There has been a lot of debate about Django templates, because the designers took a deliberate decision not to include features normally found in programming languages. If you want fancy data structures, you create them in Python and then reference them in your templates. I have to say this works extremely well for me, even though I have in the past used templating systems that were Turing complete.

The first goal of the rewrite was to duplicate the site functionality exactly. Once this was done, I started to tweak. The first such tweak was the selection of a random book for each page imprint, and that was a very easy change to make. Next, I decided it was time that my URLs stopped using file extensions. Since I have Django running on top of Apache and mod_python this mostly involved rewriting the URLs in the Django site configuration and inserting redirect directives in the Apache configuration file. By keeping an eye on the logs I have managed to make sure that anybody who follows a link from (for example) a Google search still gets to the right page. Since the redirects are permanent,

over time Google should learn the new locations.

One of the weaknesses of both TurboGears and Django from the naive user's point of view is that to get the best out of them you have to run a separate process on the server. This means that the standard cheap Web providers generally won't be able to support them, although there are companies who specialize in providing a friendly hosting environment, and of course you can always do what I did and rent a virtual private server.

I think if I had been starting from scratch either of the systems would have done a more than satisfactory job. At the moment, TurboGears is in the process of merging with the Pylons system, and we will have seen a preview release of the merged system at PyCon. Django continues to develop, and the built-in administration system (handy for simple database content maintenance) is being moved on to the new forms package.

Django's database management is currently built around SQLObject, which works very well for simple requirements; there is nothing to stop you from using the DBAPI directly and building your own SQL for more complex requirements, though typically new users prefer to complain that the system doesn't do everything they want. TurboGears uses SQLAlchemy, which is a much more capable system. You pay for this by having to cope with a somewhat greater complexity level in the code for doing relatively simple things. We can return to this topic in a later column.

I believe that the Django documentation is rather more mature than that of TurboGears, and this certainly made it easier to learn. The TurboGears community has acknowledged this deficiency, and is actively working to remedy it. Both systems suffer from the extremely rapid pace of development, which means that the documentation can be either sketchy or out of date (and occasionally both). Both communities are extremely knowledgeable and helpful, however, so it isn't difficult to find out how to do something you need to do just by going to the mailing lists or IRC. It's also good to see a high degree of mutual respect and cordiality between the two teams, each of whom acknowledges the advantages of the other's approach.

Since the holdenweb.com rewrite I have built another system using Django, to maintain a database of questions for a class I recently edited to prepare students for the CompTIA Security+ certification examination. As you may know by the time you read this (though as I write it I am anticipating a PSF Board discussion), I have plans to repurpose this system for Python, but more of that later. Since I wasn't using a huge amount of database in holdenweb.com (where the database is effectively a kind of content management system that allows all internal links to be generated dynamically, thereby ensuring perfect navigation) the second system gave me a chance to exercise SQLObject more than I previously had. It works, and it's relatively easy to use.

Signing Off

So, that's it for another month. I hope you are still having as much fun using Python as I am. I've been interested in object-oriented programming since 1973, but it was only when I found Python that I discovered a language in which I could naturally express object-oriented algorithms.

While I've been polishing this column the number of PyCon registrants has gone up to 960. Maybe Chicago will indeed have produced the first kiloPythonista by the time you read this!

See you next time.

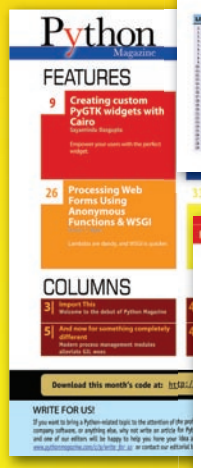
PyCon 08: Day 1

STEVE HOLDEN is a consultant, instructor and author active in networking and security technologies. He is Director of the Python Software Foundation and a recipient of the Frank Willison Memorial Award for services to the Python community.

Python Magazine

And now for something completely different

The first monthly magazine dedicated exclusively to Python.



The first issue is on us

Get a **free PDF copy** of the October issue.
No Purchase & no registration required
(because we know you'll be back anyway).

Print & PDF (1 year, 12 issues)	PDF only (1 year, 12 issues)
US & Canada: $69.99 CAD International: $89.99 CAD	Worldwide: $59.99 CAD

SUBSCRIBE TODAY!

For more info go to:
http://www.pythonmagazine.com