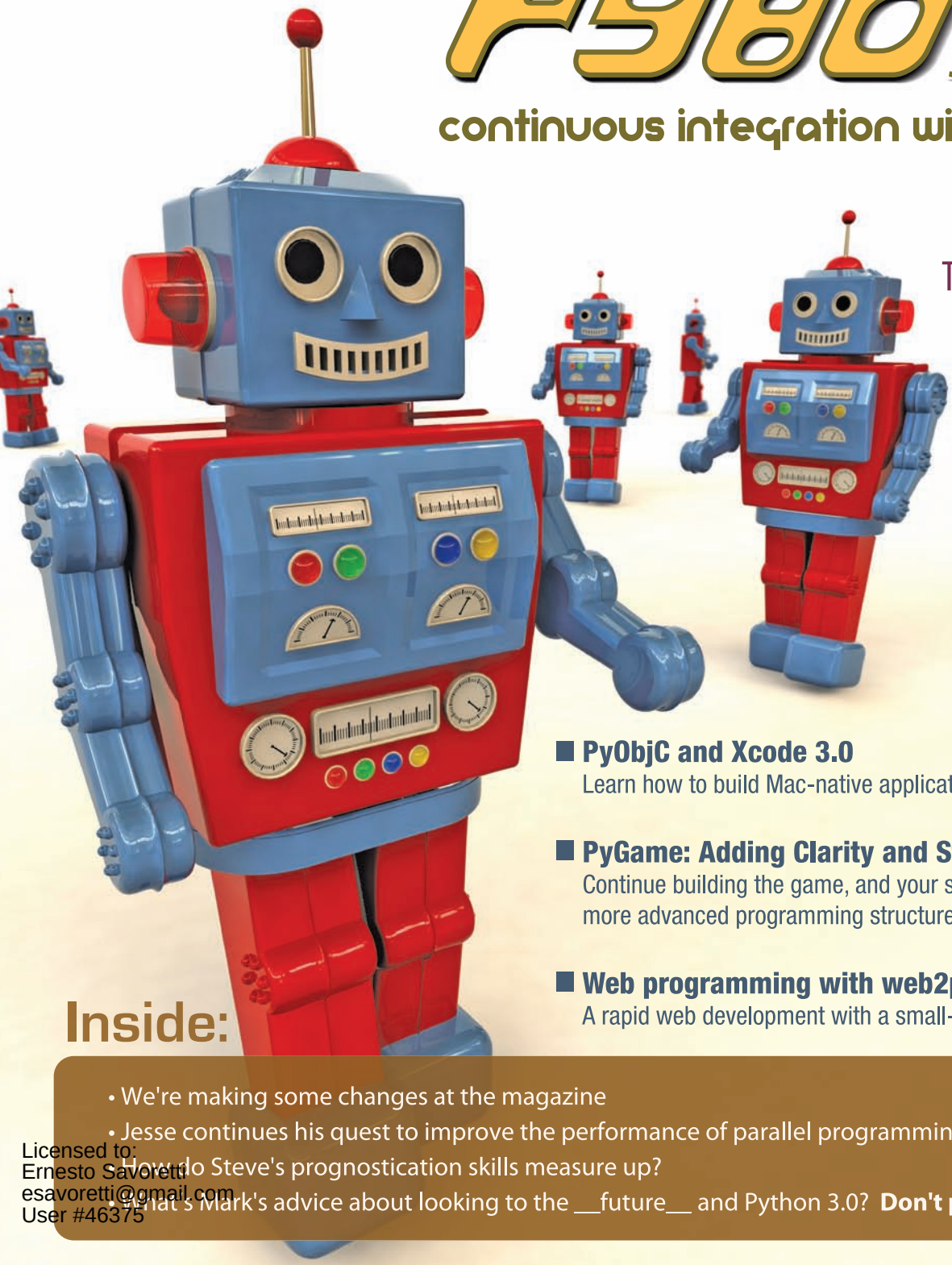
Python Magazine

JUNE 2008

PYBOTS

continuous integration with a twist



Tools and techniques for running a farm of automated test systems

- **PyObjC and Xcode 3.0**
  Learn how to build Mac-native applications with PyObjC
- **PyGame: Adding Clarity and Structure**
  Continue building the game, and your skills, while learning more advanced programming structures
- **Web programming with web2py**
  A rapid web development with a small-footprint framework

Inside:

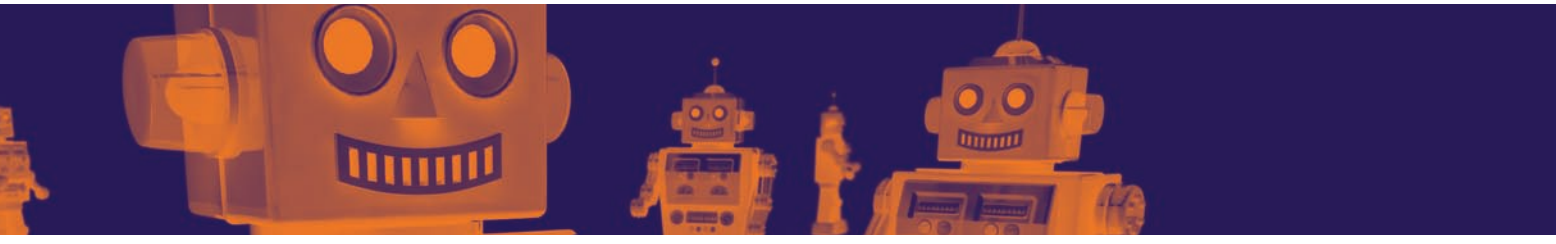
Python Magazine

CONTENTS

FEATURES

Download this month's code at:
http://code.pythonmagazine.com/2/6

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!

>>>import this

I am beyond thrilled, and beyond delighted, to announce that this will be the last "Import This" column I write, as I hand off the duty of Editor in Chief to none other than our own Doug Hellmann.

Lest you think this is a bittersweet moment for me, let me assure you that it is not. I went to our beloved publisher with the idea for Python Magazine because I felt the community deserved it, and I personally wanted to be a consumer of such a magazine, and thought others would as well. I also thought it would help the language penetrate further into the 'technological landscape', and all kinds of other wonderful things. In order to get the magazine off the ground, I took on the role of Editor in Chief to nurse the fledgling magazine until it could walk on its own.

I did *not* take on the role of Editor in Chief because I was the most qualified person to run a magazine about Python. I knew that going in, and the plan from the outset was to manage the mayhem until a suitable candidate could become my replacement. There is no way on Earth I could've ever hoped to find a better Editor in Chief than Doug Hellmann. Since the day he started with us as a technical editor, he has exceeded any and all expectations, taken initiative, developed ideas, and has been a rock solid member of our editorial team. When the time came that we needed to spread the load further, Doug took on the role of essentially managing the technical editing process and has done a superb job. Add to this his fantastic writing of the "Completely Different" column every month, and it's hard to imagine that this is also a guy with a day job and, presumably, a life.

I am happy to say that Python Magazine has legs, and is "off the ground". It's been profitable since the release of the first issue, the readership has grown, which has enabled us to give more back to the community – this year in the form of creating the programs for PyCon 2008. I don't think either Doug or myself were heavily involved in that, but our production team did a fantastic job with it, so kudos to Arbi and the team for pulling that together within the tight deadline.

So what shall become of me? Worry not. It's likely that I'll be doing something with Python Magazine for at least the next few months, probably just lending a hand where I'm needed and useful, and I'll also be doing work for the publisher in some capacity. I know it sounds corny, but we really are kind of a big family here and when one role ends, it doesn't necessarily end one's participation in the family. If there are opportunities, you can grab at them and if there's something

Python Magazine

JUNE 2008
Volume 2 - Issue 6

Publisher
Marco Tabini

Editor-in-Chief
Doug Hellmann

Associate Editors
Sharon Edgehill, Scott Leerssen, Jesse Noller, Brandon Rhodes, Drew Smathers

Contributing Editor
Steve Holden

Columnist
Mark Mruss

Graphics & Layout
Arbi Arzoumani

Managing Editor
Emanuela Corso

Authors
JC Cruz, Grig Gheorghiu, Terry Hancock, Steve Holden, Mark Mruss, Jesse Noller, Massimo Di Pierro

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

missing, you can create your own opportunities, as was the case with the creation of Python Magazine.

And what shall become of Python Magazine? Doug has plans. Big plans. He has already revamped the editorial workflow with the purpose of alleviating the strain on individual editors while at the same time pushing up deadlines so that the magazine gets edited more quickly, is published and printed sooner, gets to your doorstep in record time, and, if anything, is of better quality than it has ever been before. There are also plans for the web site, the column authors (Doug's old column will become a guest-written column)... there's seemingly no end to the flow of ideas.

So stay tuned, Pythonistas. This is not a "corporate shake up" that signals troubled times. It is the exact opposite. It is a sign of the magazine's success that we're able to continue to grow, and that people with strong ties to the language and the community are willing to expend their energy to take the magazine onward and upward. I am proud to be involved with Python Magazine and will do whatever is asked of me to support it going forward.

Cheers.

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

An Interview With Adam Olsen

Author of Safe Threading

by Jesse Noller

A world without a Global Interpreter Lock (GIL) - the very thought of it makes some people very, very happy. At PyCon 2007 Guido openly stated that he would not be against a GIL-less implementation of Python, provided someone coughed up the patch itself. Right now, that someone is Adam Olsen - an amateur programmer who has been working on a patch to the CPython interpreter since July of 2007.

REQUIREMENTS

PYTHON: *3.0*

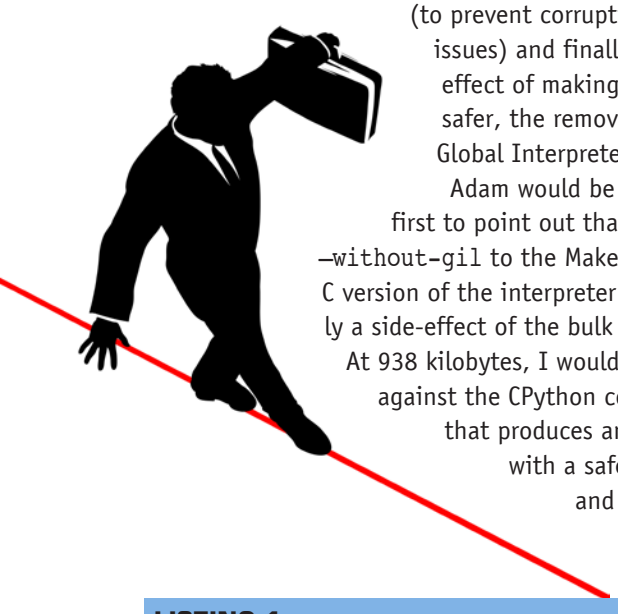
- *r36020 via the experimental bazaar mirror - http://www.python.org/dev/bazaar/*
- *The r36020 diff from the safethread page http://python-safethread.googlecode.com/files/safethread-bzr-36020.diff*

Useful/Related Links:

- python-safethread project - http://code.google.com/p/python-safethread/
- Project status - http://mail.python.org/pipermail/python-3000/2008-March/012548.html
- Wikipedia: The Actor Model - http://en.wikipedia.org/wiki/Actor_model
- MONITORS in Concurrent Programming - http://ei.cs.vt.edu/~cs5204/sp99/monitor.html

It's PyCon. I'm supposed to be listening to a talk, but I've fallen down the rabbit hole of a future without a global interpreter lock. I'm locked in on getting a patched version of the interpreter up and running on Mac OS/X and the patch author, Adam Olsen, is coaching me through changes to some of the deepest internals of Python itself.

For about a year, Adam has been working on the "safe threading" project for Python 3000. In this project, he has attempted to address many of the common issues programmers facing highly threaded and highly concurrent applications. These problems include deadlocks, isolation of shared objects (to prevent corruption/locking issues) and finally, as a side-effect of making threading safer, the removal of the Global Interpreter Lock.



Adam would be the first to point out that adding `–without-gil` to the Makefile for the C version of the interpreter was actually a side-effect of the bulk of his work. At 938 kilobytes, I would say his diff against the CPython code base that produces an interpreter with a safe, clear, and concise threading model for local concurrency is a bit more than a side effect.

It is clear that he lives for a concurrent and threaded world, and Adam has filled in a lot of gaps in my knowledge about concurrency in our past conversations. I've been lucky enough to interview him about the safe threading project and his outlook on all of this as well.

LISTING 1

```
# See the requirements: You must apply Adam's patch to Python 3000
# for this.
from __future__ import shared_module
from threadtools import Monitor, monitormethod, branch

class Counter(Monitor):
    """A simple counter, shared between threads"""
    __shared__ = True  # More shared_module boilerplate
    def __init__(self):
        self.count = 0

    @monitormethod
    def tick(self):
        self.count += 1

    @monitormethod
    def value(self):
        return self.count

def work(c):
    for i in range(20):
        c.tick()

def main():
    c = Counter()

    with branch() as children:
        for i in range(10):
            children.add(work, c)

    print("Number of ticks:", c.value())
```

First off, what's your background?

I'm an amateur programmer, self taught. I've had a long interest in object models and concurrency, such as how widgets in GTK interact.

I've explored twisted a bit, as well as Python's existing threading. Additionally, I've experimented a great deal with different ways to utilize generators or threads, actors, futures, cooperative versus preemptive scheduling, and so on.

Can you explain the basic premise behind the Safe Threading part of the project?

Make the common uses easy. Don't necessarily make it impossible to get wrong (everything is a tradeoff!), but give the programmer a fighting chance.

How about the "Free Threading" part (–without-gil)?

Everybody seems to know you don't need locking if you're not modifying an object, but Python demands a traceback, not a segfault if the programmer gets it wrong. Monitors and shareability provide a framework that satisfies both.

In essence, removing the GIL was a bonus to avoiding unintended conflicts between threads.

Many people accuse the "threaded programming" paradigm as impossible to get right. Even Brian Goetz has stated that it is extremely hard to "get right". If this is the case, why try to "fix" threading in Python?

What most people see as a problem with threads, I see as a problem with the memory model. They let threads modify the same object simultaneously, resulting in arbitrary, ill-defined results. The complexity here explodes, quickly turning the programmer's brain into mush.

The solution is to isolate most objects. Keep all these mutable objects back in the sequential, single-threaded world. Multiple processes let you do this. Actors do it too. So do monitors.

editor: *Actors can be thought of as Objects (for Object Oriented programmers) except that in the Actor model, all Actors execute simultaneously and can create more Actors, maintain their own state, and communicate via asynchronous message passing. A Monitor can be thought of as an object that encapsulates another object intended for use by multiple threads. The Monitor controls all locking semantics for the encapsulated objects.*

So from that view, processes, actors, and monitors are all equivalent. The only reason I use monitors and build them on OS threads is that it fits better with the existing Python language and is much more efficient for the way Python uses them. I could take a page from Erlang and call them "processes", but I think in the long run that would be more confusing, not less.

"With safethread integrated into Python I think many of the distributed/ multiprocess projects would die off"

In looking through the diff for safe thread, you've had to touch a lot. Everything from object allocation to the entire way threads are managed. What was the hairiest series of changes you've had to make?

Hard to say - I've been at this nearly a year with still lots to do. I could mention when I found that atomic refcounting didn't scale, it spelled doom for the removal of GIL until I came up with a viable asynchronous scheme. Straightening out object allocation was also pretty nasty, as stock CPython uses a twisted maze of macros and wrapper functions for that.

The worst was probably deciding how to handle class and module dictionaries. What you've got here are mutable objects, inherently used simultaneously by multiple threads, no clear cutoff point after which they're no longer modified, and a massive amount of implicit accesses ingrained as a fundamental part of the language. I really wanted to impose some order on this, add some clear boundaries to when they're modified versus accessed, and make them live up to the "explicit is better than implicit" ideal.

I couldn't do it though. Implicit access is too ingrained into the language. Eventually I conceded defeat, then embraced it, codifying dict's existing API as shareddict's actual API. In doing this I also switched to a relatively simple read/write lock as shareddict's protection (relative to what came before!). In the end, the only restriction was that the contents of shareddict must themselves be shareable.

Isn't a monitor equivalent to adding an @ synchronized or a with:lock statement around your code? Is using a monitor for the mutable objects that much faster than lock.acquire and lock.release?

editor: See Listing1.py for a monitor example. You will need to be running Python 3000 with Adam's patch for the code to work.

Superficially, yes, but it has deeper semantics as well. The biggest is that it imposes a shareability requirement to all objects passed in or out, and there's no way to bypass it. This basically forces you to be explicit about how you expect threads to modify mutable objects.

It also lets me use a bit saner recovery from deadlocks than would be possible using with:lock. Not that much though and there's certain circumstances when there are no ideal ways to recover.

Performance wise, lock.acquire/lock.release are irrelevant. The real competition is with adding a lock to every object, such as a list. What seems like a simple `if x: x.append(42)` actually requires 2 acquire/release pairs - something like `x.extend(y)` would require a pair for every item in y. This could easily add up to thousands of lock operations where a monitor lets you get away with just one.

How did you handle Garbage Collection?

Painfully. I originally attempted to use simple atomic integer operations for the refcounting, but I found they didn't work. Well, they were *correct*, but they didn't give me the benefit of removing the GIL. Multiple CPUs/cores would fight over the cache line containing the refcount, slowing everything to a crawl.

I solved that by adding a second mode for refcounting. An object starts out like normal, but once a second thread accesses the refcount it switches to an asynchronous mode. In this mode each thread buffers up their own refcount changes, writing out all the

changes to a single refcount at once. Even better, if the net change is 0 it can avoid writing anything at all!

The catch is, you can no longer delete the object when the refcount hits 0, as another thread might have outstanding changes. Instead, I modified the tracing GC to keep a list of *all* objects and had it occasionally flush the buffers and check for objects with a refcount of 0.

Your Branching-as-children method (see Listing 2) of spawning, implemented in your patch, deviates from the current threading module approach. Why not just overlay your work on the existing API?

Branch basically wraps up best practices into a single construct. It propagates exceptions, handles cancellation, lets you pass out return values, and ensures you don't accidentally leave a child thread running after you've returned.

You can still leave threads running after a function returns, you just need to use a branch that's higher up in the call stack. Later on, I might add a built-in one just above the main module just for this purpose.

What about stopping/pausing child threads?

Pausing isn't possible, but cancellation serves the purpose of stopping. Essentially it sets a flag on that thread to tell it to stop, as well as making sure participating I/O functions will check that flag and end themselves promptly.

How did you handle thread-safe imports?

Most of this isn't implemented yet, but the basic idea is that each module will be either shareable or unshareable. Unshareable modules work normally if imported from the main thread, but if another thread tries to import one they won't get past the parsing phase - just enough to try to detect `from __future__ import shared_module`.

Modules found to be shareable are placed in their own MonitorSpace (the underlying tool used by a Monitor) before the Python code in them is executed. This separates them from the main thread, so I won't need the main thread's cooperation to load them.

LISTING 2

```
1. with branch() as children:
2.     for i in range(10):
3.         children.add(work, arg1, arg2)
```

In your implementation, you use libatomic-ops, essentially adding a new python build/library dependency - what does this buy you over using standard locking primitives?

Scalability. I can use an atomic read and, so long as the memory doesn't get modified, all the CPUs/cores will pull it into their own cache. If I used a lock it would inherently involve a write as well, meaning only one CPU/core would have it cached at a time.

For some applications it also happens to be a great deal lighter than a lock. It may be both easier to use and faster.

You state on your page about the Dead Lock Fallacy, that "Ultimately, good style and a robust language will produce correct programs, not a language that tries to make it impossible to go wrong." What language tries to make it impossible to go wrong? Why (again) not just ditch threading and move to, say, Erlang?

Concurrent Pascal would be the great old example - they introduce monitors there, but apply a great deal more restrictions as well. Ultimately though, the language is focused on hard real-time applications, and it shows. Python and safethread are focused on general purpose applications, so usability is more important.

Erlang's a pretty similar situation. It was designed for real time, distributed, fault-tolerant applications. It wants you to use one-way messages (not the two-way function call). It copies everything passed in those messages. Good tradeoffs for its focus, but bad for a general purpose language.

What sorts of CPython bugs have you found delving this deep into the codebase?

Just a few scattered little ones. My favorite was a refcounting bug involving dicts, but it could only occur using recursive modification or threading - obviously with shareddict I make the latter a little more likely (but only recursive modification is possible with the normal dict).

Although, that's not including the threading/interpreter state APIs. Most of that code was pretty messy; lots of bugs lurking around. It was quite satisfying to rip it out.

How have your changes altered the API that C extension writers use? Given that the "bonus" of the GIL is a simple interface for extension writers via the Py_BEGIN_ALLOW_THREADS/

Py_END_ALLOW_THREADS macros - does safethread introduce more complexity?

Most of my changes are cleanup and simplification. The tp_alloc/tp_free slots are gone - everything uses PyObject_New and PyObject_Del. PyObject_GC_New/Del are gone too. The old GIL macros are directly replaced by PyState_Suspend()/PyState_Resume().

However, there are new options to take advantage of. Extensions doing I/O or other blocking operations should use the cancellation API. Modules wishing to be shareable should be audited, then apply Py_TPFLAGS_SHAREABLE/METH_SHARED, as appropriate. However, if they do that they also need to call PyArg_RequireShareable/PyArg_RequireShareableReturn if there's the potential to share objects between threads (MonitorSpaces technically).

You don't support the old threading API right now - but would it be possible to add in backwards-compatible support, or is it simply unfeasible?

For the C API, it'd be easy to retain the old GIL macros. Other parts may not be so easy.

For the Python API, some are easy, some aren't possible, and some are just painful.

Adding equivalents to Lock, Semaphore, and Queue is easy. Easier than the originals in fact. Getting all the minor details right (such as, if you subclass it) might be harder/impossible. Lock would not support deadlock detection, but it would be cancelable.

Daemon threads will likely not be supported, but, in my opinion, they're broken by design anyway.

The painful part is resurrecting the GIL, so these "classic" threads can share arbitrary objects like they always did. However, I won't make it so global - they'll acquire/release the main MonitorSpace instead, so all the new-style threads (created using branch()) will not be slowed down.

Finally, you've pointed out that "real threading" does not equal distributed programming, only local concurrency (i.e: support for multiple cores). What do you think Python could do to support distributed computing (providing the GIL-less world comes to fruition)?

At this point, it's confusing. Much of the focus is to work around the GIL, to take advantage of multiple cores. With safethread integrated into Python, I think many of the distributed/multiprocess projects would die off. What's left would be the ones that *really* want to be distributed and need multiple boxes, not multiple cores.

In my mind, there are three main characteristics of distributed programming; although, a given framework, may only use one or two:

- *security* - you don't trust the other nodes, they don't trust you. This often takes the form of sandboxes on a local box or capability system.
- *fault-tolerance* - a hardware failure on one box should only bring down that box, not every other box connected to it. Upgrading the software of one box at a time should also be possible.
- *latency* - asking another node (even on a LAN) can easily be several orders of magnitude slower than reading from your own RAM or, even better, your cache.

All these lead to different tradeoffs. You really *need* to minimize the communication between nodes by pushing them apart, whereas safethread is only concerned about making it easier to write correct programs.

The bottom line is that safethread lets you do the easy stuff (local concurrency) so that you only need to do the hard stuff (distributed programming) when you really need it.

Conclusion

Everyone is welcome to download, contribute, and try out Adam's patches. Bug reports, code, emails are all welcome. There are active discussion on the Python 3000 mailing list about all of this and more suggestions are welcome.

In all, there is a lot of interest in Adam's work. There was a lot of discussion around concurrency, threads, and the GIL at Pycon this year, and with Python 3000 coming down the pipe with the "multicore future" looming, things are getting interesting.

JESSE NOLLER (jnoller@gmail.com) is a Software Engineer working for Hitachi Data System in Massachusetts. He has been working with Python for over five years.

PYBOTS

continuous integration with a twist

by Grig Gheorghiu

Do you have automated tests for your Python project? Would you like to run them on various platforms and operating systems, whenever there is a check-in to the Python core repository? Would you like to be notified whenever a new Python feature breaks your software? If the answer to these questions is Yes, then this article by Grig Gheorghiu will show you how to accomplish these goals by using the Pybots buildbot farm.

What is continuous integration?

Many readers of the Python Magazine are no doubt familiar with the concept of continuous integration, but just to start things off on the right foot, here is Martin Fowler's definition:

> *Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

You know that a concept has arrived when it is included in the title of a book. This is indeed the case with continuous integration. The book I'm referring to, and which I highly recommend, is "Continuous Integration: Improving Software Quality and Reducing Risk" by Paul Duvall, Steve Matyas, and Andrew Glover.

REQUIREMENTS

PYTHON: *2.3+*

Other Software: Buildbot - *http://buildbot.net/trac*

Useful/Related Links:

- "Continuous Integration" by Martin Fowler - *http://martinfowler.com/articles/continuousIntegration.html*
- Pybots Home Page - *http://pybots.org/*
- Setting up a Pybots buildslave - *http://agiletesting.blogspot.com/2006/08/setting-up-pybots-buildslave.html*
- "The (lack of) testing death-spiral" by Titus Brown - *http://ivory.idyll.org/blog/mar-08/software-quality-death-spiral.html*

It should be no surprise that the authors also use Martin Fowler's definition.

To me, the most important concepts in Fowler's definition are *automated build* (including test), *integrate frequently* and *detect quickly*. From the book, I would add *deployment* and *working software*. I will highlight these concepts in the next few paragraphs. From here on, I will abbreviate continuous integration by CI.

The canonical CI setup consists of a dedicated machine, usually called the CI server, which automatically retrieves the latest version of the source code from a central repository, then proceeds to build the project and run its automated tests. *Build* means running a command chain such as 'compile; make; make install' for C/C++ projects, or 'ant' for Java projects, or 'rake' for Ruby projects. Similarly, *automated tests* can mean unit tests, or functional tests, or integration tests, or all of the above. In the ideal case, the end result is a deployable piece of working software whose automated tests have passed. The automated build-and-test process can be started either periodically (every X hours) or upon every check-in to the repository. In each case, the process is kicked off **frequently** and **reliably**. Upon completion of the build-and-test process, the results are typically displayed on a Web page, or emailed to interested parties, or published via RSS. This ensures rapid feedback and quick detection of errors.

The main value that a CI system brings to the table is the fact that it builds and runs tests against the very latest version of the source code. I have seen people, even respected individuals in the agile community, argue against the necessity of a CI system by saying that it is sufficient for individual developers to run the unit tests for the project on their local machines, before each check-in because unit tests are fast. This argument misses the point entirely, the point being that the unit tests may pass for the developers on their local copy of the source code, but only because the local copy contains code that hasn't been synchronized with the central repository (either because the developers didn't check in their code or because they didn't update first from the central repository.) The same tests may very well fail when local changes are checked in against other people's changes.

There are many automated procedures that can be implemented inside the build-and-test cycle, things such as static code inspection, database integration testing, and feedback mechanisms. Most CI literature focus on the canonical single-server setup, perhaps because it is influenced by the design of a particular single-server CI tool called CruiseControl.

However, things start to get really interesting when the CI setup extends from a single build server to a server farm composed of machines running on various hardware platforms with different operating systems, or different versions of the same operating system, or different levels of patches/updates within the same OS. 'What if my application only runs on a single platform, or a single operating system?', you ask. Let's assume the 'single OS' that your application runs on is Microsoft Windows. Which version of Windows are we talking about? Is it Windows 98, or XP Home/Pro, or Windows NT, or 2000 Workstation/Server, or 2003 Server (Standard Edition/Web Edition/Enterprise Edition), or Vista? If it's Windows 2003 Server, what Service Pack level are we talking about? Is it SP1, or SP2? Have the latest critical updates been applied? Is it running on a 32-bit or 64-bit CPU architecture? Are the CPUs single-core or multi-core? This is just scratching the surface in terms of the multitude of scenarios within the same so-called 'single OS'. Similar diverse scenarios exist for other platforms and operating systems with Linux being the poster child of multi-platform versatility. In view of all this, there is a real benefit to having a distributed CI system covering as many of these diverse scenarios as possible.

Large scale Open Source projects such as the ones under the Mozilla umbrella use CI tools that are designed from the ground up to be used in a multiple server/multiple platform environment. Mozilla's own Tinderbox CI tool is such an example. Another one is Buildbot, which is used by many Open Source projects, notably the Python programming language. It is no coincidence that the Firefox browser and the Python programming language need to run ideally on any platform under the sun. Consequently, they need to be built and tested on as many different systems as possible. This almost forces their developers to create or use CI tools designed for this scenario.

Going back to the multi-server multi-platform distributed CI setup, there are nuances even here. We can distinguish between *independent* and *directed* distributed CI systems.

In an independent scenario, the CI servers interact with the Source Control Management (SCM) system independently from each other and from the master, and run the build-and-test process periodically, then publish their results (for example via email) to a central server that displays the results. Tinderbox is an example of such a system; to see a live Tinderbox page for a

server farm that builds and tests the Firefox browser, go to http://tinderbox.mozilla.org/showbuilds.cgi?tree=Firefox. In an independent system, the CI servers have more logic and operate autonomously, but they tend to be harder to set up because of the extra logic. The build-and-test commands are typically run by each server periodically, every N hours. It is harder to coordinate the server farm to execute these commands upon every check-in to the SCM, because the SCM system would have to send notifications to each and every server in the farm. Use this setup when you have CI machines that are not necessarily always online, and when you don't need immediate feedback upon every check-in.

In a directed scenario, a master server coordinates a farm of slave servers by sending them build-and-test commands. The slave servers then report back to the master with the results of those commands and the master displays the results. Buildbot is an example of such a system; to see a live Buildbot master page for a server farm that builds and runs the unit tests for the Python programming language, go to http://www.python.org/dev/buildbot/trunk/. In a directed system, the CI slaves blindly execute the commands sent to them by the master. In this sense, they are easier to set up and coordinate. The master server can also be easily hooked into the SCM check-in notification system (if the SCM is Subversion, this can be done by using Subversion post-commit hooks), so that every check-in results in a notification to the master, which then triggers a build-and-test step that is coordinated by the master across all slave machines. The drawback is that the slaves always need to be available on the network and in constant communication with the master. Use this setup when you need immediate and synchronized feedback upon every check-in to the SCM.

Traditional Distributed CI

When I use the term *Distributed CI*, what I mean is a multi-server, multi-platform, multi-OS CI setup. The benefits of such a system are many. By testing a project on different platforms, each with its own environment, you will uncover an enormous number of platform-specific issues. You will also realize that you have taken many things for granted by relying on the idiosyncrasies of your platform of choice, things that may very well not apply to other platforms or environments.

A typical distributed CI setup is exemplified by the CI server farm (based on Buildbot) used to build and test the Python programming language. Each time a Python developer checks their code into a Subversion repository, a notification is sent to the Buildbot master server. The master then tells each machine in the CI farm to check out the latest version of the Python code (either the trunk or some specific branch), compile and build the Python binaries and libraries, and finally to run the unit test suite. As of this writing, there are 25 machines in the Python CI server farm, running the gamut of CPU platforms and operating systems. The Web status page maintained by the Buildbot master server at http://www.python.org/dev/buildbot/all/ is the dashboard that provides rapid feedback to the developers of the project.

This setup is traditional because it builds and tests a given project upon modification to the source code of the project.

Dependency-based distributed CI

For the remainder of this article, I will describe a distributed CI server farm that tests Python-based projects and is composed of machines running Buildbot, hence its name, **Pybots**. This system is also known as the 'Python community buildbots', because test machines are donated and maintained on a volunteer basis by members of the community.

The twist in the title of the article comes into play in this setup because the machines in the Pybots farm run the automated tests for their projects **each time a modification is made to the Python language**. In other words, every time Python changes, all the machines recompile, rebuild, and reinstall a new version of the Python binaries and libraries, then run the automated tests for their projects using the newly built Python executable.

The wish for such a system was expressed by Glyph Lefkowitz, one of the code developers on the Twisted project, to the python-dev mailing list. I quote Glyph:

> *I would like to propose, although I certainly don't have time to implement, a program by which Python-using projects could contribute buildslaves which would run their projects' tests with the latest Python trunk. This would provide two useful incentives: Python code would gain a reputation as generally well-tested (since there is a direct incentive to write tests for your project: get notified when core python changes might break it), and the core developers would have instant feedback when a "small" change*

breaks more code than it was expected to.

Twisted (http://twistedmatrix.com), an event-driven networking engine written in Python, has thousands of unit tests that run within a traditional Buildbot-based distributed CI server farm every time the Twisted source code is modified. However, as Glyph Lefkowitz pointed out, the Twisted team also wanted to know if their software breaks in the presence of modifications to the Python language.

Enter Pybots. The Pybots server farm started out as one machine running the Twisted unit tests every time a Python developer commits code to the Python trunk or to the latest stable branch. This provides both the Python developers and the Twisted developers with instant feedback as to how changes in the language affect Twisted. The setup has grown to currently 9 machines running on a variety of platform and operating systems. If you're interested in more details, visit the Pybots home page.

I realize I keep repeating the word Python in almost every sentence. It would be easy to get the idea that this whole setup is Python-specific, but in fact it is not. It can be replicated quite easily for projects written in any Open Source language that evolves constantly, and also for projects that depend on an API, which is in constant flux. An example of such an API is the novel user interface software for the laptops that are built as part of the One Laptop Per Child project (OLPC; see http://laptop.org/). The UI software called Sugar is written in Python, and its API is used by activities such as Browse, Chat, Read, Write. Behind each activity is an application in which the children interact. Whenever something changes in the Sugar API, it would be nice to know how the applications that depend on it are faring. A setup similar to Pybots would accomplish this goal quite easily and is, in fact, being considered by the OLPC testing team.

A system such as Pybots is representative of what I call *dependency-based distributed CI*, where projects are tested whenever there is a change in their dependencies – be it the programming language in which they're written, or the libraries and modules they use. Very few projects are perfectly self-contained with no third-party dependencies. What if your project is not Open Source? What if you have a payroll application running on Windows and written in Microsoft C++ using the Microsoft Foundation Classes? Your application is probably still using third-party charting libraries for displaying various statistics. It is also probably using database modules and drivers. It would be nice to know if your application still works on various versions of Windows whenever these modules/libraries/drivers are updated. Granted, you don't need dedicated test machines to be online 24x7 for this purpose, but an automated system that checks for the availability of updated modules, downloads them, and then recompiles/rebuilds/reinstalls your application against them would be an asset in your overall testing setup and strategy.

"I have seen people, even respected individuals in the agile community, argue against the necessity of a CI system ... because unit tests are fast. This argument misses the point entirely."

Pybots setup and workflow

The Pybots master process, known as the buildmaster, runs on a server owned by the Python Software Foundation. All the CI servers, known as the buildslaves, are owned by people who are interested in running automated tests for their preferred Python-based projects.

Here is an example of a workflow identical to the one used in the Pybots system, but generalized so that it can be applied to projects based on other programming languages:

- Programming language L uses Subversion as its Source Control Management (SCM) system
- A core developer for language L checks in

some new code, either in the trunk or in a branch
- A Subversion post-commit notification is sent to the buildmaster
- The buildmaster coordinates the buildslaves by sending them a set of commands (called 'build steps' in Buildbot lingo); these commands include: check out the latest version of the code for language L; compile and build new binaries and libraries for L based on the latest L code; run the unit tests for L; install the new binaries and libraries for L; run the automated tests for project P based on L, using the newly built binaries and libraries for L
- The buildslaves report the results of the build steps back to the buildmaster
- The buildmaster summarizes the results on a Web page and optionally within an RSS feed
- The buildmaster also optionally sends notifications to owners of buildslaves via email

While I keep mentioning 'bots' and 'automation', the human element should not be underestimated. Before all this workflow could happen, somebody from the Python core development team had to enable the Subversion post-commit notifications to the buildmaster process. Somebody had to configure and maintain the buildmaster process. And, most importantly, people had to contribute their own buildslave machines for doing the actual testing work.

My job as the Pybots farm coordinator was made easier by the fact that the Pybots system has a directed-type setup, where the buildmaster controls and distributes work across buildslaves. I published a series of steps (see Related Links) that all buildslaves had to implement, which made it relatively easy for people to contribute their own buildslaves and join the project. More on this later when I talk about ways to build an Open Source community.

As of this writing, the Pybots server farm is composed of 8 buildslave machines testing more than 20 projects. The machines are x86 in their vast majority, running operating systems such as Debian, Ubuntu, Gentoo, Fedora Core 6, RedHat 9, Mac OS X, and Windows 2003). The setup also includes an AMD64 machine running Ubuntu), a Sun SPARC machine running Solaris 10, and a Mac G5 running OS X.

Some of the projects tested in the Pybots system are Twisted, Trac, Genshi, Django, SQLAlchemy, Bazaar, Storm, docutils, roundup, Kid, MySQLdb, CherryPy, lxml. Every time a check-in is made to the Python trunk (which represents version 2.6) or to the 2.5 branch, the Pybots buildslaves go through the workflow I described above. You can see the current state of the Pybots projects that are tested against the Python trunk here: http://www.python.org/dev/buildbot/community/trunk/. You can see the state of the projects tested against the 2.5 branch here: http://www.python.org/dev/buildbot/community/2.5/. When a test fails, buildslave owners are notified by email, and they can also subscribe to an RSS feed, either for their own machine only or for the entire server farm.

Issues uncovered by the Pybots system

Setting up a distributed CI system is not a walk in the park, no matter what architecture you choose, which means that it'd better provide some good feedback and find some juicy bugs! I'm happy to report that the Pybots system has indeed been finding issues in almost every project under test. Although the problems uncovered are related to Python, they belong to general categories that apply to any distributed CI system, regardless of the programming language used by the projects.

Platform-specific issues

This is supposed to be the bread-and-butter category for a distributed CI system, and indeed the Pybots setup turned up a sleuth of platform- specific issues. Here are some examples:

- the '%zd' string formatting expression was not working as expected for negative numbers on Mac OS X
- multicast traffic was misbehaving on Red Hat 9; we had to add a specific iptables firewall rule to allow that type of traffic
- tests involving paths to files and directories were failing miserably on Windows, while passing with flying colors on Unix-like systems
- TCP client code was failing on Windows and passing everywhere else.

Syntax-related and deprecation-related issues

There were a few syntax changes in Python 2.6 compared to its previous version. In particular, the words "as" and "is" became reserved keywords, and thus could

not be used as variable names anymore. We spotted a handful of projects that were using them as variable names.

In Python 2.6, raising a string exception is not supported anymore. This broke code relying on this feature, and we knew about it right away.

Test suite issues

We encountered an issue with a project whose automated tests weren't quite as automated as expected.

"Setting up a distributed CI system is not a walk in the park, no matter what architecture you choose, which means that it'd better provide some good feedback and find some juicy bugs!"

It turned out that when running the test suite on Windows, a DOS prompt would pop up during some of the unit tests, so these tests were modified to expect a key press from the user to close the dialog box. Of course, this was hardly conducive to running those tests automatically inside a CI server farm. Luckily, the developers responded very promptly and modified the tests so that no user input was required.

Another project had several separate test suites and there was no easy way to run them with a single command (the Integrate button that the CI book talks about). Again, the developers were responsive and changed their setup so all unit tests for the project could be run at once.

Build environment issues

Some Python modules, called extension modules, are written in C, and must be compiled on each target platform. On some platforms in the Pybots farm, these modules didn't get compiled at all, which resulted in test failures. This was because changes to certain header files didn't trigger a full rebuild of the Python source code. While this is an issue specific to the Python build environment, it is symptomatic of problems that can arise within any build system.

Installability issues

This was one of the most spectacular issues uncovered by the Pybots setup. When the Python trunk version changed from 2.5 to 2.6, not all files containing the version number were modified by the core Python developers. Those files did not affect the unit tests for the Python code itself, so the unit tests were passing merrily, green and happy. However, when we tried to install any Python project within the Pybots farm with the canonical `python setup.py install` command, we got nasty errors, such as:

```
error: invalid Python installation:
unable to open /tmp/python-buildbot/local/lib/
python2.6/config/Makefile
(No such file or directory)
```

This was a transient error fixed in a very short time, but it underlined the extreme importance of having functional tests for your project. In this case, the functional test was a simple install test, but the idea is that it exercised the newly built binary from the outside in, black-box style – and not only from the inside out, white-box style, as the unit tests did. Remember how I mentioned the important goal of having deployable and working software? Well, testing that the software can be installed at all is a good first step toward that goal.

Lessons learned

Perhaps the most important lesson from the Pybots project is that unit tests are not sufficient – as was evidenced by the installability issues mentioned above. The best approach to testing is to have an end-to-end suite of tests at all levels: unit, functional, acceptance, and yes, even GUI tests. In agile methodologies, we talk about the *tracer bullet* approach to development: start with a path through your application that goes from the GUI all the way to the database for example. Make sure this path works. Then keep adding features while making sure that nothing breaks (some people compared this method to the process of making candles: start by dipping the wick in wax to make a very thin candle, then repeat until you have a candle). The same approach can be applied to testing: start by writing a few tests at each level (unit, functional, GUI, etc). Schedule this test suite to run in a continuous

integration process, for example using buildbot. See how the tests turn green every time somebody checks code in. Rejoice inwardly. Then add more tests, rejoice some more, etc. You get the idea. It is a *virtuous circle*, as opposed to the *vicious cycle* of testing your software manually, finding bugs, and checking in fixes that cause more bugs in some other part of your application (see also Titus's article on **The (lack of) testing death-spiral**).

Here are some other lessons learned from the Pybots project:

- it is important to get all the tests to pass; if a build/test step is failing/red all the time, people will end up ignoring it (this is the broken window syndrome)
- you need a buildmaster for your distributed CI project, somebody who needs to stay on top of things and alerts people when their buildslaves are down; otherwise bitrot will manifest itself unfailingly
- it helps to have good documentation for the CI setup, and also a central repository of scripts and configuration files, so people who want to join the project can get up to speed in a short time (Google Code is a very easy way to achieve all this)
- for a community project such as Pybots to succeed, you need to promote/market the project tirelessly; some good methods for doing this are blogging and sending messages to mailing lists related to the area of your project (in this case, the developers mailing lists for the various projects under test)
- to sustain and grow an Open Source community project such as Pybots, you need to acknowledge the contributions of the volunteers (for example by blogging about them); you also need to respond quickly to offers for help, or to issues raised on the mailing list; it also helps if you demonstrate the usefulness of the project to the community at large, so that you can co-opt more people to contribute to the project

Future work

The Pybots project is always looking for more volunteers, so we can add more projects under test, running on more buildslaves, covering more platforms. By the time you read this, we should have added the Py3K branch into the mix, so the if you add your project to the Pybots farm, you will be notified of all check-ins to the Py3K branch. This assumes that you will have a project whose tests are all passing when run through Py3K. I don't know of many such projects, but it sure will be interesting to have a testbed for this type of testing.

Another idea that Titus Brown and I have thrown around for a while would be to have a set of buildslaves as virtual machines that would run the tests for PyPI packages both automatically and on-demand. The process would look something like this for an on-demand run: you upload the code and the tests for your Python project via some Web interface, we start up a virtual machine, check out the latest Python trunk (or a specific branch if you so desire), build and install the latest Python binaries/libraries, and then run the tests for your project. It also has been suggested that we package the buildslave VMs and letting folks use pre-configured slaves at their own site by importing the VMs. I think that's an interesting idea, definitely worth exploring.

This process can also run automatically in a setup similar to Pybots. In this case though an independent setup, where the slaves run their tests independently of each other and publish the results to a central server/repository – makes more sense because the slaves will come and go on the fly. Note that the idea of a virtual machine is essential in this case. You don't want to run the tests for an unknown piece of software in any environment other than the one you can create and tear down on the fly (think malware).

Acknowledgments

I would like to thank the volunteers who have contributed to the Pybots project by donating their machines and their time: Seo Sanghyeon, Mike Taylor, Manuzhai, Elliot Murphy, Sidnei da Silva, Matthew Flanagan, Skip Montanaro, John Hampton, Jeff McNeil, Graeme Glass.

GRIG GHEORGHIU (grig@gheorghiu.net) is the Director of Technology for RIS Technology, a managed services and Web hosting company. Grig is an active participant in the Python community, and has presented tutorials on agile/automated testing tools and techniques at the last 3 PyCon conferences. He blogs regularly at *http://agiletesting.blogspot.com*.

PyObjC and Xcode 3.0

by JC Cruz

In this article, Jose shows how to build a basic Cocoa application using PyObjC and Xcode 3.0. He covers designing a user interface with the new Interface Builder and demonstrates how to create a controller class and connect it to each interface widget. To complete the application, all of the controller's actions are defined using Python.

From Cocoa To Python

The Cocoa framework is the basic building structure for most, if not all, OS X applications. It consists of at least two major kits: the Foundation Kit and the Application Kit. The Foundation Kit consists of classes that are, by design, device-independent. Some classes define data structures, such as NSString and NSDictionary. Others define process structures, like NSTimer and NSThread. The Application Kit, or AppKit for short, is a group of classes for the user interface. Some classes in AppKit, such as NSButton, define control widgets; others, such as NSTableView, define display widgets.

The entire Cocoa framework is written in Objective C, or ObjC for short. If you are unfamiliar with this C dialect, you may find using the Cocoa classes frustrating at best. But if you are well-versed in Python, you can use your Python skills to build native OS X applications with Cocoa. This is made possible by using PyObjC.

PyObjC is an open-source bridge linking Python and ObjC classes. With this bridge, you can call and handle ObjC classes from Python, or vice versa. You can build an OS X application entirely in Python, or as a mixture of Python and ObjC. You can even use Python modules

REQUIREMENTS

PYTHON: *2.4+*

Other Software:

- Xcode 3.0
- MacOS X 10.5.x

Useful/Related Links:

- Installing PyObjC - https://svn.red-bean.com/pyobjc/tags/pyobjc-1.3/Install.html
- The Xcode User Guide - http://developer.apple.com/documentation/DeveloperTools/Conceptual/XcodeUserGuide/Contents/Resources/en.lproj/index.html
- Introduction to PyObjC - http://macdevcenter.com/pub/a/mac/2003/01/31/pyobjc_one.html
- The PyObjC Home Page - http://pyobjc.sourceforge.net
- Using PyObjC for Developing Cocoa Applications - http://developer.apple.com/cocoa/pyobjc.html

to enhance ObjC functionality.

MacOS X 10.5, also known as Leopard, is the first version to include PyObjC as part of its system. Older versions of OS X, obviously, do not have the same feature. If you want to install PyObjC on those versions of OS X, you can find the instructions at the PyObjC site (see Related Links).

OS X Leopard comes with PyObjC version 2.0. This version allows you to model ObjC classes as Python metaclasses. It has improved support for the NSDictionary class, and it fixes many bugs that plague earlier versions. To see a more detailed history list, visit the PyObjC web site.

At the time of writing, the latest version of PyObjC is 2.1, which is the first one that runs in 64-bit mode. To use 2.1, you must have OS X Leopard running on 64-bit hardware. It is also untested and may have some unseen issues. If you still want to try this version, you can download a copy from the PyObjC repository.

Enter Xcode 3.0

Xcode 3.0 is the first version of Xcode with integrated support for PyObjC 2.0. This version lets you build and manage your PyObjC project with the same ease as a Cocoa or Java project. Version 3.0 removes the need for a setup.py file to compile and link a PyObjC file. It compiles PyObjC classes faster and gives a smaller application size. But Xcode 3.0 still has some support issues with PyObjC. The most notable one is the lack of source-level debugging. The only debug options you have are either the utility method `NSLog()` or the Python logging module.

With Xcode 3.0, you can write your project from scratch or use one of the bundled project templates. To select a project template, choose **New Project** from the File menu. Then use the New Project Assistant dialog to save the template under a project name. Let's look at the four PyObjC project templates available in Xcode 3.0.

The **Cocoa-Python** template gives you a basic PyObjC project. It builds an application with a single window. It also uses a single PyObjC class to handle any messages from the main application thread.

The **Cocoa-Python Core-Data** template adds a single Core-Data class to the basic PyObjC project. Core-Data is a Cocoa kit used for data storage and retrieval. It supports various data formats including XML, binary, and SQLite.

Cocoa-Python, Document-based builds an application that handles more than one document window. It uses the NSDocument class as the basis of each window. That same class also handles most document tasks such as loading, printing, and saving.

Cocoa-Python Core-Data, Document-based is a variant of the previous template, but with a Core-Data class added to the mix.

All four project templates have at least one xib bundle, which defines the user interface of the application. Double-click the bundle to edit its contents with the Interface Builder tool. All four templates include the two source files `main.py` and `main.m`. The `main.py` file imports the AppHelper class from the PyObjCTools module. It then calls `runEventLoop()` from that class. The `main.m` file prepares the resources needed by AppHelper. Then it loads `main.py` and executes the code using the Python interpreter.

Xcode 3.0 also comes with four PyObjC file templates, each one defining a specific PyObjC class:

Python NSObject creates a subclass of NSObject. The NSObject class is the root class of most Cocoa classes. It serves as the link to the ObjC runtime engine, and handles basic tasks such as memory management and messaging. Use this template to define your own PyObjC class.

Python NSDocument defines an instance of the NSDocument class. As described earlier, NSDocument handles the basic tasks needed by a document window.

Python NSView will build an instance of the NSView class. The NSView class manages the basic drawing, event-handling, and printing tasks on the OS X application. It also serves as the basis of most interface widgets.

Python NSWindowController adds an instance of the NSWindowController class to the project. The NSWindowController class loads and displays the window resources from the xib bundle. It keeps track of the window's location on the screen, and saves its state to a file for later use.

You can use these templates to create files in your project, or write one from scratch. To select a file template, choose **New File** from the File menu. Use the New File Assistant dialog to save the file under a unique name.

Each file template imports the PyObjC modules needed for their functions. Each of the imported module links to a specific kit in the Cocoa framework. For example, the Foundation module refers to Foundation Kit, the AppKit module to Application Kit. The objc module, however, defines the constants, methods, and

keywords found in the Objective C runtime library. For instance, this module defines the constants YES and NO, which are the ObjC versions of True and False. All PyObjC sources must import the objc module at the start of their files.

Finally, there are other PyObjC modules that you can import into your source files. The ScreenSaver module lets you build an OS X screen-save module in PyObjC.

" If you are well-versed in Python, you can use your Python skills to build native OS X applications with Cocoa."

SearchKit allows you to use the new Spotlight engine to search for files with a specific content or metadata. XgridFoundation gives you the ability to handle distributed processes. And the WebKit module lets you handle and display web information.

These are just some of the 26 modules bundled with PyObjC 2.0. To see a more complete list, go to the directory:

```
/System/Library/Framework/Python.framework.
```

All the modules are in the subdirectory:

```
Versions/Current/Extras/lib/python/PyObjC.
```

Note that this is a read-only subdirectory. Do not use this subdirectory as a drop-off point for your own PyObjC modules. A better place to store your custom modules is in the public directory /Library/Python/2.5.

Starting A Project

We will now build a basic BMI calculator using Xcode 3.0 and PyObjC. BMI, or *body-mass index*, is a common health index, computed as the ratio of a person's weight and height squared. An underweight person will have an index less than 19, an overweight one an index greater than 24. BMI requires the height and weight values expressed in meters and kilograms. But our calculator will support other units of measures such as feet and pounds. The source for the BMI project is included in the ZIP file with the PDF copy of this magazine.

Start up your copy of Xcode 3.0 and choose **New Project** from the File menu. From the New Projects Assistant dialog, scroll down the list of templates and select **Cocoa-Python Application**. Click the **Next** button and enter BMI on the Project Name field. Leave the Project Directory field set to your home directory. Click the **Finish** button to create the project.

Next, we need to update the project's Info.plist file. This file describes the application to the OS X Finder. It stores the information as a series of key/value pairs. Xcode sets most of the keys to their default values.

CFBundleIconFile holds the name of the .icns file that has the application icon. Xcode places this file in the Contents/Resources directory of the application bundle. If you assign a null string to this key, the OS X Finder will use a generic icon for your application.

To learn how to make your own .icns file, refer to http://www.macinstruct.com/node/59. The BMI project sets this key to the name icon_bmi.icns. Make sure to keep the .icns file on the root level of the project directory.

CFBundleIdentifier is the bundle signature of the application. This string uniquely identifies the application to the OS X Finder. The application also saves its preferences file under the same string. This key has the string com.pymag.demo.BMI in the BMI project. Note that the string uses the reverse domain name scheme to ensure uniqueness.

CFBundleSignature holds the creator code of the application. The OS X Finder uses this code to assign a document file with the right application. The code consists of four printable characters. Apple recommends you use a mix of numbers and letters for the creator code. Keep in mind, however, that Apple reserves all codes that consists entirely of uppercase or lowercase letters. The BMI project uses the creator code to 8M1C. Feel free, of course, to provide your own four-character code.

CFBundleShortVersionString and CFBundleVersion are the version strings of the application. The first key sets the application's version, the second key the bundle's version. In the BMI project, the first key has the string

"1.0", the second "2.0".

Next, you need to update the file Info.plist.strings. This file stores the copyright string used by the application's About window. It has only one key named NSHumanReadableCopyright. The BMI project sets this key to the string "(c) Python Magazine, 2008". Again, feel free to provide your own string if you prefer.

Designing The Interface

Let us now design the user interface for our BMI calculator. Go to the **Groups & Files** pane of the Xcode window, and locate the **Resources** group. Double-click the group to reveal the `MainMenu.xib` bundle. Then double-click the bundle to open it with the Interface Builder tool. The Interface Builder tool will display at least two windows (Figure 1). The first window shows the contents of MainMenu.xib. There are six objects in this window, but the ones of interest are BMIAppDelegate and Window.

BMIAppDelegate is an instance of NSObject. Its sole task is to serve as the delegate to the main application thread. If the main thread gets a message that it cannot handle, it sends that message to the delegate. In our BMI project, however, we will leave this object and its source code unchanged.

The Window object is an instance of the NSWindow class. It will be the main window of the BMI Calculator. If you double-click the object, you will get a blank window, on which you will add the necessary UI widgets.

FIGURE 1

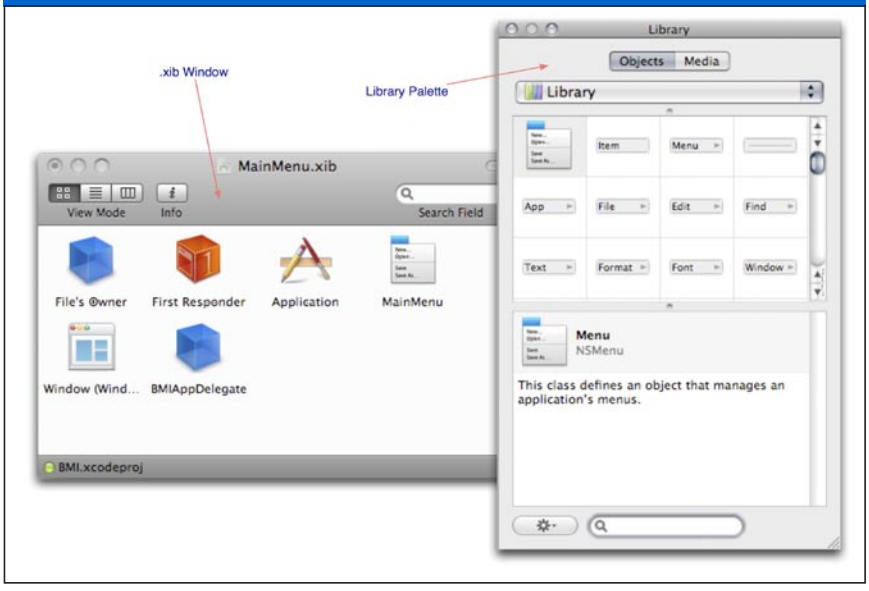


FIGURE 2

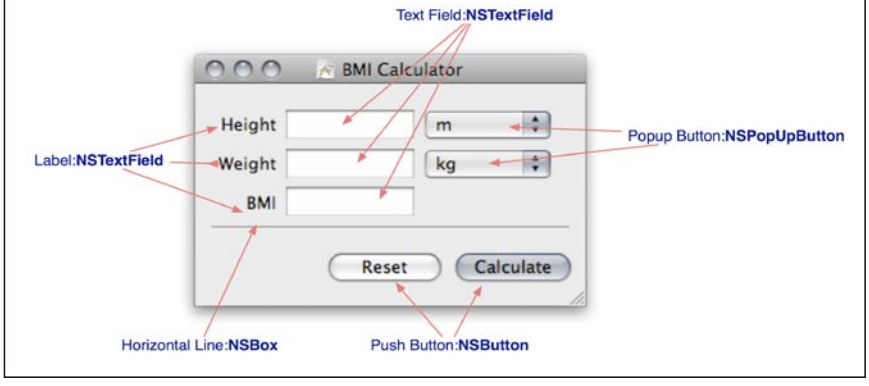
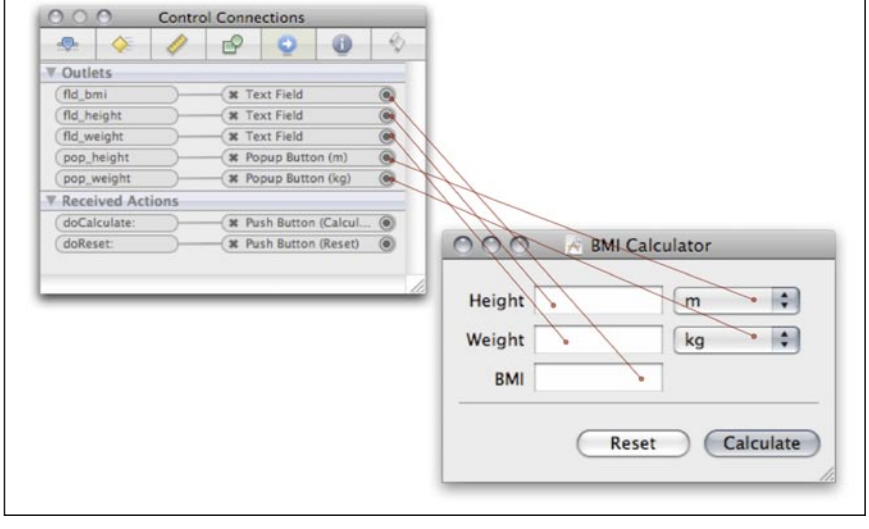


FIGURE 3



The second window shown by Interface Builder is the Library palette. This palette displays the resources that you can add to the xib bundle. At the top of the palette are two tab buttons. Clicking the **Objects** tab gives you a list of Cocoa objects. Clicking on the Media tab displays a list of audio and graphic files.

Below the tab buttons is the Library drop-down menu. Use it to show only those resources that fall under a specific category. For instance, to get a list of PDF objects, choose **PDFKit** from the menu. For a list of button widgets, choose **Cocoa -> Views & Cells -> Buttons** from the menu. Here, the **->** token refers to a subcategory on that menu.

At the bottom of the Library palette is a search field. When you enter a string in this field, the palette shows only those resources whose names match the string. For example, choose **Cocoa** from the palette's drop-down menu. Type the word `button` in the search field and press **Enter**. The palette then displays all the button widgets that you can use.

"Most OS X applications use the model-view-controller (or MVC) design concept."

To prepare the window, double-click the Window object on the MainMenu.xib window. Add the control widgets as shown in Figure 2. The first part of the widget's name is its type; the second part, after the colon and in bold text, is its Cocoa class. For example, to add a Label widget, choose **Cocoa -> Inputs & Values** from the Library menu. Locate and select the widget on the palette. The palette will show the widget's class on the field below that list. Drag the widget onto the window to add it. Once you have placed all the widgets, resize the window by dragging its lower-right corner.

Notice that two or more widgets can use the same class and yet behave differently. In our BMI window, the three Label widgets use the NSTextField class, the same class that the four Text Field widgets use. But a Text Field widget lets users enter a new string or change the displayed string. A Label widget, on the other hand, prevents users from changing the displayed text.

Defining the Controller

Most OS X applications use the model-view-controller (or MVC) design concept. It dictates that we factor out our application code into three sets of objects: model, view, and controller. The *model* stores, manipulates, or converts the data. In our BMI project, the model will be made up of the functions that convert the user's height and weight, and calculates the BMI value. The *view* displays the data and interacts with the user. In our project, this will be the Window object and its widgets. The *controller* manages the flow of data and messages between the model and the view. This object derives from either an NSController or an NSObject class.

Next, we will define the controller object BMIControl and connect it to the BMI window. Start with the Library palette and click the **Objects** tab. Choose **Objects & Controllers** from the drop-down menu. You will see the first object selected, which is the NSObject class. Click and drag its icon to the MainMenu.xib window. Click its name and change it to BMIControl. Now choose **Identity Inspector** from the Tools menu. On the **Class** field in the Class Identity pane, type the name `BMIControl`. Then scroll down to the Interface Builder Identity pane. Make sure the Name field has the same string `BMIControl`; if not, enter the string into that field. Close the inspector palette when done.

Now, let us define the controller's outlets and actions. Outlets and actions are the main lines of communication between a controller and a view. An *outlet* allows the controller to get or send data to a view. An *action*, on the other hand, lets a view send a signal to the controller.

Select the BMIControl object on the MainMenu.xib window. Choose **Identity Inspector** from the Tools menu. On the Class Actions pane, click the **+** button to add a blank entry. Change the entry's name to `doCalculate`. Repeat the same step, but this time set the entry's name to `doReset`. Make sure that both action names still end with a colon. Next, on the Class Outlet's pane, click the **+** button to add a blank entry. Change this entry's name to `fld_bmi`. Repeat the step four more times to add the following outlets: `fld_height`, `fld_weight`, `pop_height`, and `pop_weight`. Again, close the inspector palette when done.

Time to connect each Window widget to BMIControl.

To start, select the object BMIControl from the MainMenu.xib window. Then choose **Connections Inspector** from the Tools menu to display the inspector palette.

First, we will connect the five outlets. Figure 3 shows how each outlet in BMIControl is connected to which widget. On the inspector palette, click and hold on the circle next to each outlet name. Drag a line from that circle to one of the widgets, then release when the right widget is selected. To disconnect the outlet, click the **X** widget next to its name. Thus when BMIControl accesses `fld_height`, for example, it reads the data entered on the Height field. And when BMIControl updates `fld_bmi`, the data shows up on the BMI field.

Our next task is to connect the two actions. Figure 4 shows how the two NSButton widgets connected to the BMIControl actions. First, locate the action name on the inspector palette. Click and hold on its circle, then drag a line to a widget. Release when you selected the right button widget. For our BMI project, the **Calculate** button links with the `doCalculate` action, and the **Reset** button with the `doReset` button. As before, if you want to disconnect an action, click the **X** widget next to its name. Thus when users click the **Reset** button, that button sends a signal to BMIControl via the `doReset` action. And when they click the **Calculate** button, the button sends another signal via the `doCalculate` action.

Implementing Controller and Model

We are now ready to write the controller code. To start, choose **Write Class Files** from the File menu. Use the Save File dialog to navigate to the BMI project directory. Click the **Save** button to create the source file as `BMIControl.py`. When done, click on the widget labelled `BMI.xcodeproj`, located at the lower-left corner of the MainMenu.xib window. This will return you to the Xcode editor window.

On the Xcode window, select the **Classes** group in the **Groups & Files** pane. Choose **Add To Project** from the Project menu. Locate the file `BMIControl.py` and click the **Add** button to add

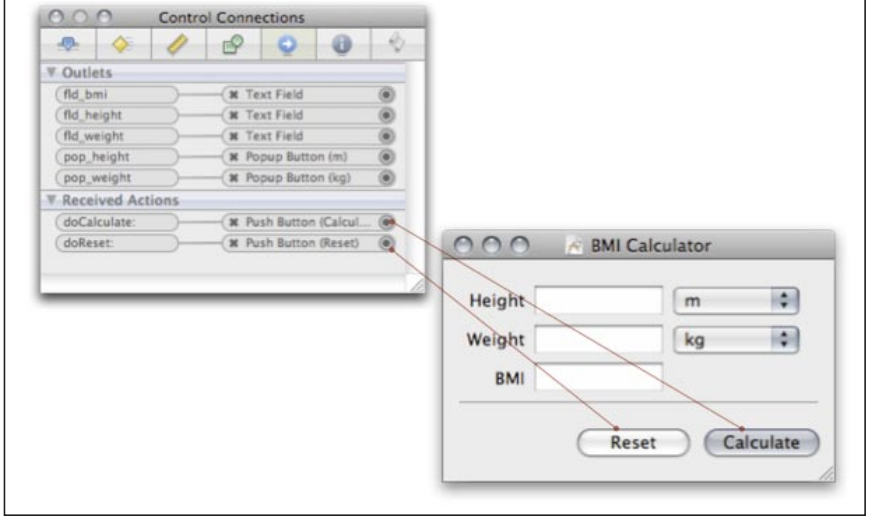
LISTING 1

```
from Foundation import *
from AppKit import *
import objc

class BMIControl:
    fld_bmi = objc.IBOutlet()

    fld_height = objc.IBOutlet()

    fld_weight = objc.IBOutlet()

    pop_height = objc.IBOutlet()

    pop_weight = objc.IBOutlet()

    @objc.IBAction
    def doCalculate_(self, sender):

    @objc.IBAction
    def doReset_(self, sender):

```

LISTING 2

```
@objc.IBAction
def doCalculate_(self, sender):
    # retrieve the height value
    tHgt = self.fld_height.floatValue()

    # convert the height value
    tUnt = self.pop_height.selectedItem()
    tUnt = tUnt.tag()
    tHgt = BMIModel.convertHeight(tHgt, tUnt)

    # retrieve the weight value
    tWgt = self.fld_weight.floatValue()

    # convert the weight value
    tUnt = self.pop_weight.selectedItem()
    tUnt = tUnt.tag()
    tWgt = BMIModel.convertWeight(tWgt, tUnt)

    # calculate the BMI value
    tBMI = BMIModel.calculateBMI(tHgt, tWgt)

    # display the BMI value
    self.fld_bmi.setFloatValue_(tBMI)
```

FIGURE 4



the file to the BMI project. Xcode then prompts you for instructions on how to add the file. For now, just use the default settings and click the **Add** button again.

Listing 1 shows the basic code of the BMIControl class. This code is not based on any of the PyObjC templates, but is made directly by Interface Builder. BMIControl starts by importing the three Cocoa modules Foundation, AppKit, and objc. Then it defines the five outlets (lines 5-13) as properties. Each property uses `objc.IBOutlet()` to get their assigned widgets. At first, it appears that all the outlets may refer to the same widget. But in practice, `objc.IBOutlet()` links each outlet to the right widget in `MainMenu.xib`.

Next, BMIControl defines the two actions `doCalculate` and `doReset` as methods. To differentiate them from other instance methods, each action is decorated with `@objc.IBAction`. Like `objc.IBOutlet()`, `@objc.IBAction` links each action to the right widget in MainMenu.xib.

Update the `doCalculate` action as shown in Listing 2. First, the action gets the user's height and converts it to meters (lines 4 to 9). Next, it gets the user's weight and converts it to kilograms (lines 12 to 17). The action then calculates the body-mass index and sends the result to the outlet `fld_bmi` (lines 20 to 23).

For the `doReset` action, use the code shown in Listing 3. When users click the **Reset** button, the action sets `fld_height` and `fld_weight` to 0. Then it sends a null string to `fld_bmi`. Finally, it tells `pop_height` and `pop_weight` to select the first unit of measure.

Let us now define our model. Choose **New File** from the File menu. From the Assistant dialog, select **Empty File** and click the **Next** Button. Save this file under the name `BMIModel.py`. Then, update the file as shown in Listing 4. The file defines a BMIModel class with three methods. The first method, `convertHeight()`, takes the user's height data, and returns the results in meters. The second one, `convertHeight()`, converts the weight data, returning the results in kilograms. And `calculateBMI()` computes the body-mass index using the converted height and weight. Once you finished writing `BMIModel.py`, add the line `import BMIModel` to the file `BMIControl.py`.

LISTING 3

```
1. @objc.IBAction
2. def doReset_(self, sender):
3.     # clear the following fields
4.     self.fld_height.setFloatValue_(0.0)
5.     self.fld_weight.setFloatValue_(0.0)
6.     self.fld_bmi.setStringValue_("")
7.
8.     # reset the following pop-up menus
9.     self.pop_height.selectItemWithTag_(0)
10.    self.pop_weight.selectItemWithTag_(0)
11.
```

Testing The Project

Time to give the BMI project a go. Choose **Console** from the Run menu to display the console window. This window will list any errors from either the build process or the BMI application. Click the **Build and Go** button on the window's toolbar to start the build process. First, Xcode compiles `MainMenu.xib` and two prefix headers, one for each PyObjC source file. Next, it compiles each PyObjC file, and links them to the runtime library. Xcode should find no errors during the build, and launch the BMI application as a result.

But once BMI displays its main window, it sends

LISTING 4

```
1. # Convert the given height to metres
2. def convertHeight(aVal, aUnt):
3.     # identify the height unit
4.     if (aUnt == 1):
5.         # centimetres to metres
6.         tHgt = aVal / 100.0
7.     elif (aUnt == 2):
8.         # feet to inches to centimetres to metres
9.         tHgt = aVal * 12
10.        tHgt *= 2.54
11.        tHgt /= 100.0
12.    elif (aUnt == 3):
13.        # inches to centimetres to metres
14.        tHgt = aVal * 2.54
15.        tHgt /= 100.0
16.    else:
17.        # metres
18.        tHgt = aVal
19.    # return the conversion results
20.    return (tHgt)
21.
22. # Convert the given weight to kilogrammes
23. def convertWeight(aVal, aUnt):
24.    # identify the weight unit
25.    if (aUnt == 1):
26.        # grammes to kilogrammes
27.        tWgt = aVal / 1000.0
28.    elif (aUnt == 2):
29.        # pounds to kilogrammes
30.        tWgt = aVal * 0.454
31.    elif (aUnt == 3):
32.        # stones to pounds to kilogrammes
33.        tWgt = aVal * 14
34.        tWgt *= 0.454
35.    else:
36.        # kilogrammes
37.        tWgt = aVal
38.    # return the conversion results
39.    return (tWgt)
40.
41. # Calculate the body-mass index
42. def calculateBMI(aHgt, aWgt):
43.    # parametre check
44.    if (aHgt <= 0.0):
45.        tBMI = 0.0
46.    else:
47.        # calculate the BMI value
48.        tBMI = aHgt * aHgt
49.        tBMI = aWgt / tBMI
50.
51.    # return the results
52.    return (tBMI)
53.
```

a list of errors to the Console window (Listing 5). These errors tell you that BMI was unable to bind the class BMIControl to its window's widgets. The first error in the list points to a possible cause. It states that BMIControl could not be found in the MainMenu.xib bundle. If you look at the file `BMIControl.py`, you will find that the BMIControl class did not specify a superclass. This is a bug in the Interface Builder tool. When the tool created the class definition for BMIControl, it failed to include the superclass to that definition. To correct this problem, change the line `class BMIControl:` to `class BMIControl(NSObject):`.

"The PyObjC bridge lets you build OS X applications using Python..."

Stop the project by choosing **Stop** from the Run menu. Then choose **Clean** from the Build menu to remove any files generated by the build process. Click the **Build and Go** button to rebuild the project. Again, Xcode follows the same steps to compile the BMI application. Yet the application still displays the same list of errors shown in Listing 5. To fix this problem, look at the file `main.py`. Note the file imports the class BMIAppDelegate prior to calling `AppHelper.runEventLoop()`. But it does not do the same for BMIControl. To correct this problem, add the line `import BMIControl` after the entry for BMIAppDelegate. Choose **Save** from the File menu, and follow the same steps to rebuild the BMI project. BMI should send no runtime errors after it displays its main window.

To test the application, enter 69 in the **Height** field, choosing **in** (for inches) as the unit of measure. Next, enter 140 in the **Weight** field, and choose **lb** (for pounds) as the unit. Click the **Calculate** button. BMI will display 20.6927 as the body-mass index. Click the **Reset** button. BMI will set both **Height** and **Weight** fields to 0.0, and the **BMI** field to a null string. It will also choose the units **m** (meters) and **kg** (kilograms) from the pop-up menus.

Final Remarks

The PyObjC bridge lets you build OS X applications using Python as the main language. This bridge gives you access to most Cocoa classes, especially those in the Foundation and Application Kit. It also allows you to call your own Python modules within a Cocoa class.

Xcode 3.0 is the first release with full support for PyObjC. It comes with project and file templates that users can use to start a PyObjC project. It builds the project transparently, without the need for a separate setup.py file. Also, its companion tool, Interface Builder, can create PyObjC source files from objects defined by a xib bundle. This support for PyObjC is not without problems. Issues like the lack of source-level debugging still needs to be addressed. Nevertheless, it is a solid first step by Apple, showing their commitment to viable open-source solutions.

So that is it for our first PyObjC article. As you can see, there is a lot of ground to cover in this topic. So if you are interested in more PyObjC articles, make sure to inform the editors of your interest. Until next time, I bid you all well.

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.

LISTING 5

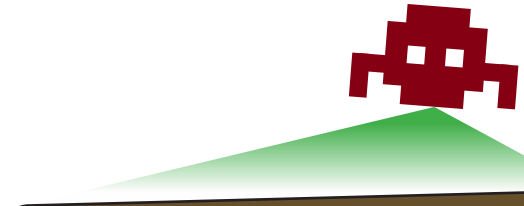
```
1. [Session started at 2008-03-17 16:23:14 -0700.]
2. 2008-03-17 16:23:21.962 BMI[20298:10b] Unknown class `BMIControl' in nib file, using `NSObject' instead.
3. 2008-03-17 16:23:21.990 BMI[20298:10b] Could not connect the action doReset: to target of class NSObject
4. 2008-03-17 16:23:22.010 BMI[20298:10b] Could not connect the action doCalculate: to target of class NSObject
5. 2008-03-17 16:23:22.688 BMI[20298:10b] Application did finish launching.
6.
```

Learning Python with PyGame PART II

Adding Clarity and Structure

by Terry Hancock

Most programs are more than just sequential scripts of instructions, but why? This month, we'll *refactor* our program to make use of programming structures like *dictionaries*, *functions*, and *classes* that will make it easy to take the next steps in turning our script into a game.



REQUIREMENTS

PYTHON: *2.3+*

Other Software:
- PyGame 1.6+ - http://www.pygame.org
- Simple Directmedia Layer http://www.libsdl.org

When we left off last month, we had a single mouse sprite bouncing around on the screen randomly (you may want to refer to the listing from last month, or `mouse2.py` in the source package). There are several improvements that would be nice to make: for example, the mouse could face the direction of motion, and it'd be a lot more fun to control the character from the keyboard.

The Refactor and Extend Cycle

Although we theoretically could make changes directly to the script we've written, the program would quickly become messy and unmanageable because we haven't taken any real steps to organize the code. So, the first thing we'll do is to *refactor* the program, which means we'll make changes to its structure, *without changing what it does*. We'll use three Python features — dictionaries, functions, and classes — to achieve this.

After each refactoring, we'll extend the program's behavior, taking advantage of the design improvements. This two step approach is very useful, because the program can be tested at each stage to ensure that mistakes aren't made in the refactoring process. For space reasons, I'll only provide full listings for the extension steps, skipping the refactoring steps. However, you should do these steps separately, and test your program after each refactoring. This is an important habit to form, especially as you write more complex programs. Each refactored version is provided in the source package for you to look at and test.

Dictionaries

In the previous installment, we managed the directions by using simple symbolic names:

```
N  = ( 0, -1)
S  = ( 0,  1)
W  = (-1,  0)
E  = ( 1,  0)
NE = ( 1, -1)
NW = (-1, -1)
SE = ( 1,  1)
SW = (-1,  1)
directions = [N, E, S, W, NE, NW, SE, SW]
```

These are simply variables, and they all occupy the global *namespace* of this *module*, which is the collection of all variable names in the top-level of our program file. Used together like this, it's pretty obvious what these variable names mean, but separately, `N` or `E` might accidentally be used somewhere else in our program to simply represent "a number" or "an endpoint" or some other mnemonic, and we'd wind up with a *name collision*: a case where we accidentally assign two different meanings to the same name. This almost always creates a bug, and that's why we generally want to avoid crowded namespaces in programs.

Dictionaries provide a simple one-to-one relationship between different objects: one is the *key*, and the other is the *value* (note the use of commas and colons in the dictionary example below):

```
directions = {  'n' :( 0,-1), 'ne':( 1,-1),
                'e' :( 1, 0), 'se':( 1, 1),
                's' :( 0, 1), 'sw':(-1, 1),
                'w' :(-1, 0), 'nw':(-1,-1)  }
```

This is almost the same information as contained in the previous representation of the directions, but instead of eight new variables, we just have one dictionary called `directions`. The directions are now explicitly named with text strings (this will help when we need to extend the program to do more with the direction information).

These changes require additional ones: instead of assigning values in a list called `directions`, we access the values in the the dictionary called `directions` by *indexing* it with the appropriate key. The result is the code in the source package file `mouse3.py`. You can try running this program, and you'll see that it behaves exactly as the previous listing does: we've changed the structure of the program, but not its function.

Turning the Mouse

One advantage of using strings to represent the directions is that we can also use the strings to label image filenames containing different versions of the mouse pointing in each of the desired directions:

```
sprites = {}
for direction in directions:
    filename = 'resource/mouse_%s_lg.bmp' %
direction
    img = pygame.image.load(filename)
    img = img.convert()
    sprites[direction] = img
    sprites[direction].set_colorkey((255,255,255))
```

Now we have a new dictionary, `sprites`, which maps the strings representing directions to image values instead of directional values. We can also see some basic rules about using a dictionary:

- When you loop over a dictionary, you are actually looping over its keys

" Dictionaries provide a simple one-to-one relationship between different objects"

LISTING 1

```
# Import the modules we need
import random
import pygame

# Get PyGame ready to work
from pygame.locals import *
pygame.init()

# Create a PyGame window and load a sprite graphic
screen = pygame.display.set_mode((800,600))

# Python dictionary with directions mapping to moves as (dx,dy) tuples
directions = {  'n' :( 0,-1), 'ne':( 1,-1),
                'e' :( 1, 0), 'se':( 1, 1),
                's' :( 0, 1), 'sw':(-1, 1),
                'w' :(-1, 0), 'nw':(-1,-1)  }

# Different sprites for each direction
# Note I've made sure that all the mouse sprites are the
# same size (64x64 pixels), which makes the math simpler.
sprites = {}
for direction in directions:
    sprites[direction] = pygame.image.load(
        'resource/mouse_%s_lg.bmp' % direction).convert()
    sprites[direction].set_colorkey((255,255,255))

# Get the width and height of the sprite & screen
spritew, spriteh = sprites['e'].get_size()
screenw, screenh = screen.get_size()

# Compute the starting point
x = screenw/2 - spritew/2
y = screenh/2 - spriteh/2

# Infinite "game loop"
i = 100

while True:
    i += 1
    if i > 100:
        # Change direction, and update the counter
        direction = random.choice(directions.keys())
        i = 0

    # Turn at boundaries
    if  x < 0:
        direction = random.choice(['e', 'ne', 'se'])
    elif  x > screenw-spritew:
        direction = random.choice(['w', 'nw', 'sw'])
    elif  y < 0:
        direction = random.choice(['s', 'se', 'sw'])
    elif  y > screenh-spriteh:
        direction = random.choice(['n', 'ne', 'nw'])

    # Update the position
    dx, dy = directions[direction]
    x += dx
    y += dy

    # Determine which sprite to use
    sprite = sprites[direction]

    # Update the display with the image
    screen.fill((75,255,55))
    screen.blit(sprite, (x,y))
    pygame.display.update()

    # Simple timing delay
    pygame.time.wait(5)

```

- When you index a dictionary using a key, the value is returned

This code retrieves images and associates them with each direction. So, for example, we load a west-facing mouse from the file `resource/mouse_w_lg.bmp` and associate it with the direction `"w"`. The `set_colorkey()` method call causes each image to also have white pixels (represented by the color tuple `(255,255,255)`) set as transparent. The resulting data structure is illustrated in Figure 1.

Now, we simply have to alter the loop to change which sprite image the `sprite` name is bound to, and that sprite image will be used to display the mouse, causing the mouse to face the direction of travel. This is the code in Listing 1 (`mouse4.py`), which is much more aesthetically pleasing, as you can see by running it.

Functions

The next refactoring is a little more significant. It's actually very odd to have so much code running directly in the module namespace; most programs are constructed of functions or classes. Last month, we used functions from the modules we imported, but now we'll define our own.

As a first attempt, let's move the program loop code into its own function, which we'll simply call `update()`. This code, in `mouse5.py`, is a partial success: it allows us to separate the concerns of managing the mouse sprite's position and appearance on screen from the mechanics of repainting the PyGame window and blitting the mouse image onto it.

The idea is that we simply pass the current position to the function, and it returns the new position and which sprite image to display. Ideally, the function would take care of all the other details — hiding or *encapsulating* them so they don't clutter the code that calls the function.

It's obvious that `x` and `y` should be parameters of the function, telling where the sprite should be placed. However, it's less clear what we should do with `direction` (the current direction of motion), `sprites` (the sprite image dictionary), and `i` (a counter used internally by the function).

At first, you would be inclined to make `i` entirely internal to the function, because no other code needs to access it. You absolutely do not want to have to think about this in the code that calls the function (because

FIGURE 1

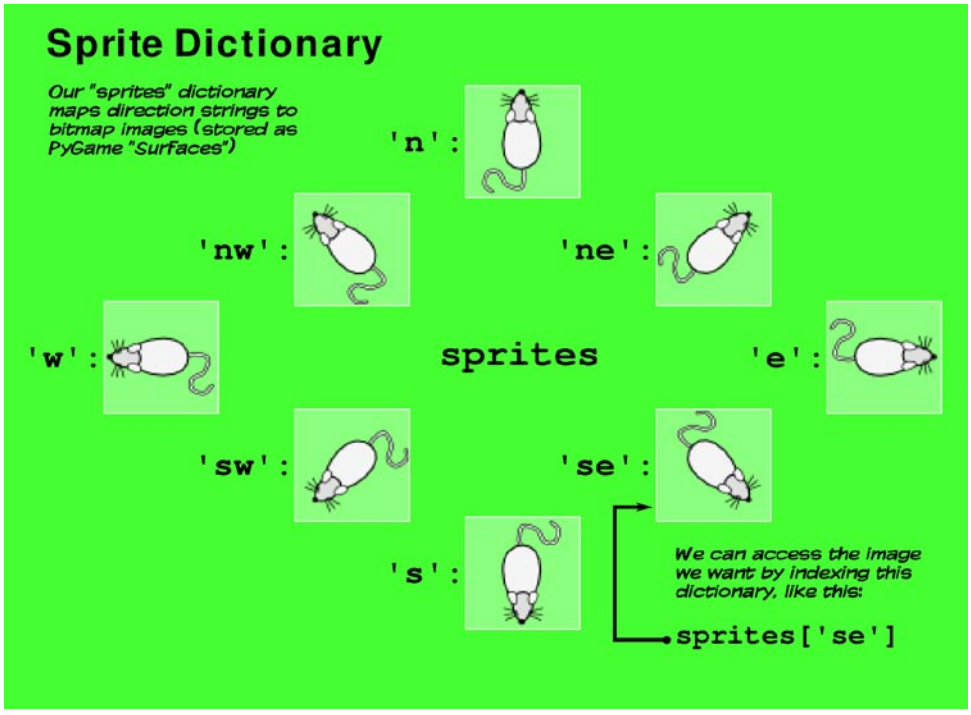
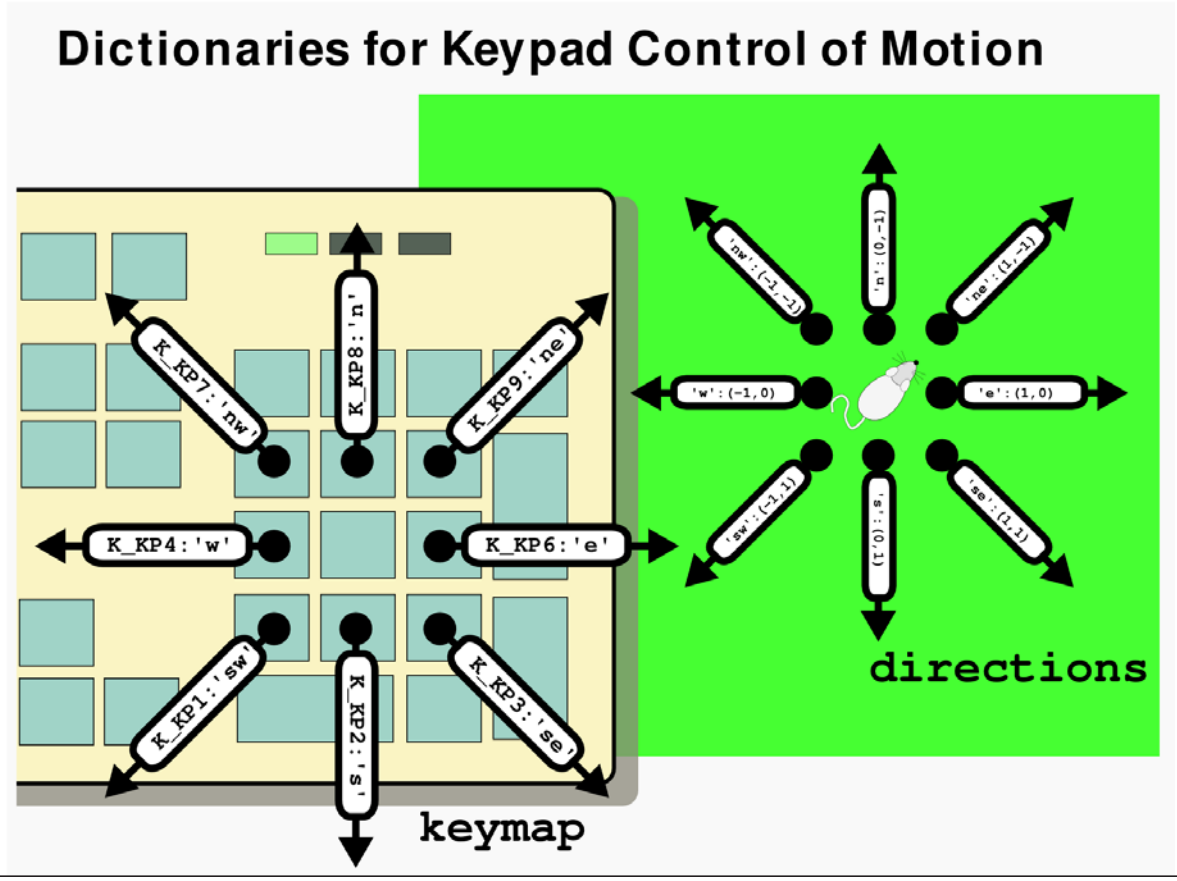


FIGURE 2



it's an internal concern of the function). Unfortunately, the function needs to remember the value of `i` from one call to the next! This is sometimes called a *static* variable.

We'll see some better ways to handle such variables shortly, but for now, we'll simply make this variable *global*. The `global` statement tells Python to use the variables from the module's scope instead of creating new variables inside the function

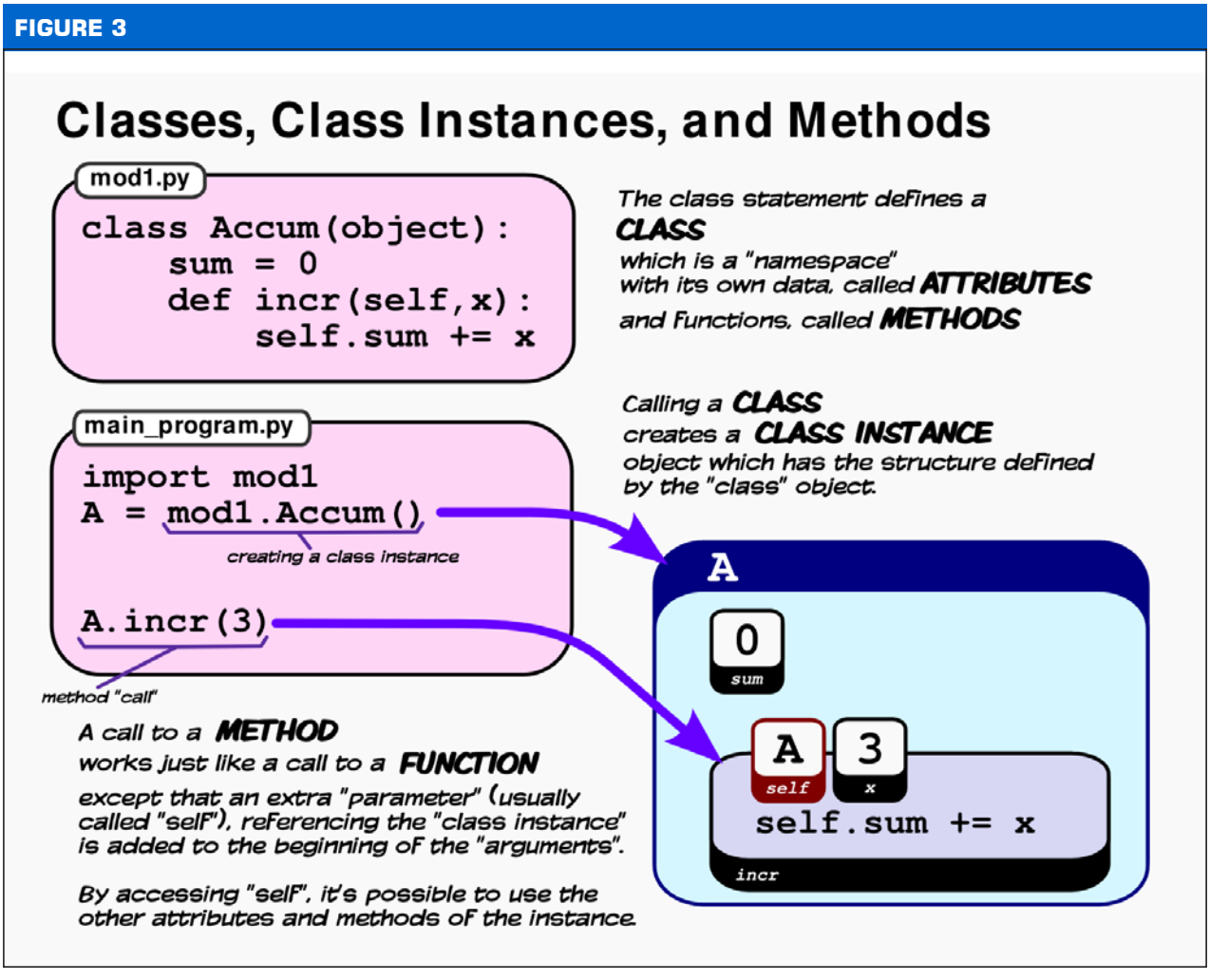
That way, the function call doesn't have to be changed, and the calling code doesn't need to understand this detail. On the other hand, we've introduced a risk: since `i` is global, we might accidentally modify it from some other part of the code, without realizing that the function is using the variable to remember something important. Similar judgement calls have to be made about `sprites` and `direction`.

By making the variables `x`, `y`, and `direction` local and returning the new values (plus the sprite to use), we raise the possibility of reusing the same update function for a second mouse character. We simply have to define two positions, `(xa, ya)` and `(xb, yb)`, and two directions, `dir_a` and `dir_b`, and then we can update and blit the two mice separately, without having to duplicate the contents of the `update` function:

```
xa, ya, dir_a, sprite_a = update(xa, ya, dir_a)
xb, yb, dir_b, sprite_b = update(xb, yb, dir_b)

screen.blit(sprite_a, (xa, ya))
screen.blit(sprite_b, (xb, yb))
```

Which is a big savings over repeating the entire function body separately for each, not to mention the confusion of having to keep track of the `a` and `b` variants all the way through, increasing the chance of making a programming error. This example is demonstrated in `mouse5b.py` in the source package, but we won't be building on that version.

FIGURE 3



Keyboard Controls

We can also introduce keyboard control of the mouse character, allowing the user to set the mouse's direction by pressing the arrows on the numeric keypad. To do this, we'll define another function which will scan PyGame's *event queue* and then update the mouse's direction accordingly.

Under the hood, capturing key presses or other input from the user is pretty tricky, because it can occur at any time. When it does, the computer actually has to stop what it's doing, process the input data, and then pick up where it left off. During a single loop in our program this might happen once, a hundred times, or not at all.

Fortunately, the operating system and PyGame take care of these details for us: they simply build a list of *event* objects, one for each event that occurred since the last update. All our program has to do is to look through this event list element-by-element and check for the events we're interested in: namely, key press or KEYDOWN events. When we catch one, we'll further check to see if it is one of the keys we're interested in, and then we'll respond accordingly.

" the first thing we'll do is to refactor the program"

For the direction controls, we can conveniently store the key symbols as another dictionary: mapping key symbols to directions. We'll add separate checks for other events we want to check, such as the user pressing the "Escape" key or closing the PyGame window to end the program. The result should look like this:

LISTING 2

```
# Import the modules we need
import sys
import random
import pygame

# Get PyGame ready to work
from pygame.locals import *
pygame.init()

# Create a PyGame window and load a sprite graphic
screen = pygame.display.set_mode((800,600))

# Python dictionary with directions mapping
# to moves as (dx,dy) tuples
directions = {  'n' :( 0,-1), 'ne':( 1,-1),
                'e' :( 1, 0), 'se':( 1, 1),
                's' :( 0, 1), 'sw':(-1, 1),
                'w' :(-1, 0), 'nw':(-1,-1)  }

# Different sprites for each direction
# Note I've made sure that all the mouse sprites are the
# same size (64x64 pixels), which makes the math simpler.
sprites = {}
for direction in directions:
    sprites[direction] = pygame.image.load(
        'resource/mouse_%s_lg.bmp' % direction).convert()
    sprites[direction].set_colorkey((255,255,255))

# Get the width and height of the sprite & screen
spritew, spriteh = sprites['e'].get_size()
screenw, screenh = screen.get_size()

# Compute the starting point
x = screenw/2 - spritew/2
y = screenh/2 - spriteh/2

# Infinite "game loop"
i = 100
direction='e'
def update(x, y, direction, auto=True):
    global i, sprites
    i += 1
    if i > 100 and auto:
        # Change direction, and update the counter
        direction = random.choice(directions.keys())
        i = 0

    # Turn at boundaries
```

LISTING 2: Cotinued...

```
    if  x < 0:
        direction = random.choice(['e', 'ne', 'se'])
    elif  x > screenw-spritew:
        direction = random.choice(['w', 'nw', 'sw'])
    elif  y < 0:
        direction = random.choice(['s', 'se', 'sw'])
    elif  y > screenh-spriteh:
        direction = random.choice(['n', 'ne', 'nw'])

    # Update the position
    dx, dy = directions[direction]
    x += dx
    y += dy

    # Determine which sprite to use
    sprite = sprites[direction]

    return x, y, sprite, direction

keymap = { K_KP8:'n', K_KP2:'s', K_KP4:'w', K_KP6:'e',
           K_KP7:'nw', K_KP9:'ne', K_KP3:'se', K_KP1:'sw' }

def move_player(direction):
    for event in pygame.event.get():
        if (event.type == QUIT or
           (event.type == KEYDOWN and
            event.key == K_ESCAPE)):
              sys.exit(0)
        elif event.type == KEYDOWN:
            if event.key in keymap:
                direction = keymap[event.key]
    return direction

while True:
    # Update position and sprite to display
    x, y, sprite, direction = update(
                          x, y, direction, auto=False)
    direction = move_player(direction)

    # Update the display with the image
    screen.fill((75,255,55))
    screen.blit(sprite, (x,y))
    pygame.display.update()

    # Simple timing delay
    pygame.time.wait(5)

```

LISTING 3

```
# Import the modules we need
import random
import pygame

# Get PyGame ready to work
from pygame.locals import *
pygame.init()

# Create a PyGame window and load a sprite graphic
screen = pygame.display.set_mode((800,600))
screenw, screenh = screen.get_size()

# Python dictionary with directions mapping to moves
# as (dx,dy) tuples
directions = {  'n' :( 0,-1), 'ne':( 1,-1),
                'e' :( 1, 0), 'se':( 1, 1),
                's' :( 0, 1), 'sw':(-1, 1),
                'w' :(-1, 0), 'nw':(-1,-1)  }

(AUTO, PLAYER) = range(2)

class Character(object):
    def __init__(self, name, position,
                     multisprite=False, behavior=AUTO):
        if multisprite:
            self.sprites = {}
            for direction in directions:
                self.sprites[direction] = pygame.image.load(
                 'resource/%s_%s_lg.bmp' % (name, direction)
                  ).convert()
                self.sprites[direction].set_colorkey((255,255,255))
            self.sprite = self.sprites['n']
        else:
            self.sprite = pygame.image.load(
                'resource/%s_lg.bmp' % name).convert()
            self.sprite.set_colorkey((255,255,255))

        self.w, self.h = self.sprite.get_size()
        self.x, self.y = position
        self.x -= self.w/2
        self.y -= self.h/2

        self.multisprite = multisprite
        self.w, self.h = self.sprite.get_size()
        self.i = 100
        self.direction = 'e'

        self.behavior = behavior


    def update(self):
        # Course changes:
        self.move_auto()

        # Turn at boundaries
        if  self.x < 0:
            self.direction = random.choice(['e', 'ne', 'se'])
        elif  self.x > screenw-self.w:
            self.direction = random.choice(['w', 'nw', 'sw'])
        elif  self.y < 0:
            self.direction = random.choice(['s', 'se', 'sw'])
        elif  self.y > screenh-self.h:
            self.direction = random.choice(['n', 'ne', 'nw'])

        # Update the position
        dx, dy = directions[self.direction]
        self.x += dx
        self.y += dy

        # Determine which sprite to use
        if self.multisprite:
            self.sprite = self.sprites[self.direction]
        screen.blit(self.sprite, (self.x, self.y))

    def move_auto(self):
        self.i += 1
        if self.i > 100:
            # Change direction, and update the counter
            self.direction = random.choice(directions.keys())
            self.i = 0

# Starting positions
position1 = (screenw/4, screenh/2)
position2 = (3*screenw/4, screenh/2)

mouse = Character('mouse', position2, multisprite=True)
```

```
keymap = { K_KP8:'n', K_KP2:'s', K_KP4:'w',
           K_KP6:'e', K_KP7:'nw', K_KP9:'ne',
           K_KP3:'se', K_KP1:'sw' }

def move_player(direction):
    for event in pygame.event.get():
        if ((event.type == QUIT) or
            (event.type == KEYDOWN and
             event.key == K_ESCAPE)):
              sys.exit(0)
        elif event.type == KEYDOWN:
            if event.key in keymap:
                direction = keymap[event.key]
    return direction
```

"With an *object-oriented* approach, we can organize things much better"

The first part of the `if` block checks for events meant to close the program. If it finds one, it calls a function from the `sys` module (don't forget to import the new module) which actually shuts down the program. For now, it's probably best to just remember this idiom instead of trying to understand the details. The second part scans for KEYDOWN events which are key presses, and simply checks to see if any are in the keymap dictionary above. If so, the new direction is chosen from it.

Figure 2 shows how the `keymap` and `directions` dictionaries interact, and also shows the layout of the keyboard controls for the program.

You may wonder where the key symbol names are coming from. This is one of the things that we got when we imported the contents of `pygame.locals` into our program. The PyGame documentation contains a complete list of these symbols (under "Key").

We can use this to put the mouse under keyboard control. In Listing 2 (`mouse5c.py`), I've also added a

LISTING 3: Continued...

```

# Infinite "game loop"
while 1:
    # Clear the screen
    screen.fill((75,255,55))

    # Update the sprite on screen
    mouse.update()

    # Update PyGame
    pygame.display.update()

    # Simple timing delay
    pygame.time.wait(5)

```

check for automatic control, so we can eliminate the random direction resets that occur with the standard update function.

Classes & Class Instances

Although the functional approach allows us to manage more than one character sprite, we still have a fairly complex job, and it's hard to specialize the separate objects (for example, to have more than one set of sprites). With an *object-oriented* approach, we can organize things much better if we define a *class* for character sprites and define the mouse as an *instance* of that class. This requires a fairly major reorganization of the code, and it will be a little bit larger when we finish, but this will really pay off in the extensions to the code that we will be making next month.

First of all, we'll create a class called `Character`. This is done with Python's `class` statement. The name `object` in parentheses after the class name is not an argument, but rather the parent or *base* class that we are sub-classing. For now, it's probably best to just accept the `object` base class as a necessity for modern Python code. We'll make more use of it next month. Listing 3 (`mouse6.py`) shows this object-oriented refactoring of the code.

LISTING 4

```
# Import the modules we need
import sys
import random
import pygame

# Get PyGame ready to work
from pygame.locals import *
pygame.init()

# Create a PyGame window and load a sprite graphic
screen = pygame.display.set_mode((800,600))
screenw, screenh = screen.get_size()

# Python dictionary with directions mapping to moves
# as (dx,dy) tuples
directions = {  'n' :( 0,-1), 'ne':( 1,-1),
                'e' :( 1, 0), 'se':( 1, 1),
                's' :( 0, 1), 'sw':(-1, 1),
                'w' :(-1, 0), 'nw':(-1,-1)  }

keymap = { K_KP8:'n', K_KP2:'s', K_KP4:'w', K_KP6:'e',
           K_KP7:'nw', K_KP9:'ne', K_KP3:'se', K_KP1:'sw' }

(AUTO, PLAYER) = range(2)

class Character(object):
    def __init__(self, name, position,
                 multisprite=False, behavior=AUTO):
        if multisprite:
            self.sprites = {}
            for direction in directions:
                self.sprites[direction] = pygame.image.load(
                 'resource/%s_%s_lg.bmp' % (name, direction)
                  ).convert()
                self.sprites[direction].set_colorkey((255,255,255))
            self.sprite = self.sprites['n']
        else:
            self.sprite = pygame.image.load(
                'resource/%s_lg.bmp' % name).convert()
            self.sprite.set_colorkey((255,255,255))

        self.w, self.h = self.sprite.get_size()
        self.x, self.y = position
        self.x -= self.w/2
        self.y -= self.h/2

        self.multisprite = multisprite
        self.w, self.h = self.sprite.get_size()
        self.i = 100
        self.direction = 'e'

        self.behavior = behavior


    def update(self):
        # Course changes:
        if self.behavior == PLAYER:
            self.move_player()
        else:
            self.move_auto()
```

LISTING 4: Continued...

```

        # Turn at boundaries
        if  self.x < 0:
            self.direction = random.choice(['e', 'ne', 'se'])
        elif  self.x > screenw-self.w:
            self.direction = random.choice(['w', 'nw', 'sw'])
        elif  self.y < 0:
            self.direction = random.choice(['s', 'se', 'sw'])
        elif  self.y > screenh-self.h:
            self.direction = random.choice(['n', 'ne', 'nw'])

        # Update the position
        dx, dy = directions[self.direction]
        self.x += dx
        self.y += dy

        # Determine which sprite to use
        if self.multisprite:
            self.sprite = self.sprites[self.direction]
        screen.blit(self.sprite, (self.x, self.y))

    def move_auto(self):
        self.i += 1
        if self.i > 100:
            # Change direction, and update the counter
            self.direction = random.choice(directions.keys())
            self.i = 0

    def move_player(self):
        for event in pygame.event.get():
            if (event.type == QUIT or
                (event.type == KEYDOWN and
                 event.key == K_ESCAPE)):
                sys.exit(0)
            elif event.type == KEYDOWN:
                if event.key in keymap:
                    self.direction = keymap[event.key]

# Starting positions
position1 = (screenw/4, screenh/2)
position2 = (3*screenw/4, screenh/2)

mouse = Character('mouse', position2,
    multisprite=True, behavior=PLAYER)

# Infinite "game loop"
while 1:
    # Clear the screen
    screen.fill((75,255,55))

    # Update the sprite on screen
    mouse.update()

    # Update PyGame
    pygame.display.update()

    # Simple timing delay
    pygame.time.wait(5)

```

You'll notice that every method has a first parameter called `self`. Python fills this with a reference which points to the class instance itself. So, for example, we'd be able to define a mouse character and update it like this:

```
mouse = Character('mouse', (100,100),
                  multisprite=True)
mouse.update()
```

" We can also introduce keyboard control of the mouse character"

Although we've passed no arguments to `update()`, it will actually receive an implicit one, which is just the object `mouse` (passed as the `self` parameter of `update()`). This allows us to use Python's dot notation to access the attributes and other methods of the instance: thus we can call `move_auto()` from within `update()`, by referring to `self.move_auto()`. Likewise, all of the persistent information about the class (such as position, what sprite is being displayed, and the current direction of travel) are accessible. Figure 3 shows these relationships between classes, class instances, attributes, and methods.

Using a class is much better than defining all of the values at the module level, both because the information is defined close to the function that uses it, and also because there are separate values for each instance of `Character`, making it easy to keep the information separated, even if we have multiple characters.

Classes can define a "magic" method named `__init__()`, which will be called when a new instance of the class is defined. We'll use `__init__()` to define the specifics of each character: what its name is, what sprite image is used to represent it, whether it's a player or automatic character, and so on. We can save the information for the life of the `Character` instance by assigning values to *instance attributes*, which we access through the `self` object, as in these examples from the listing:

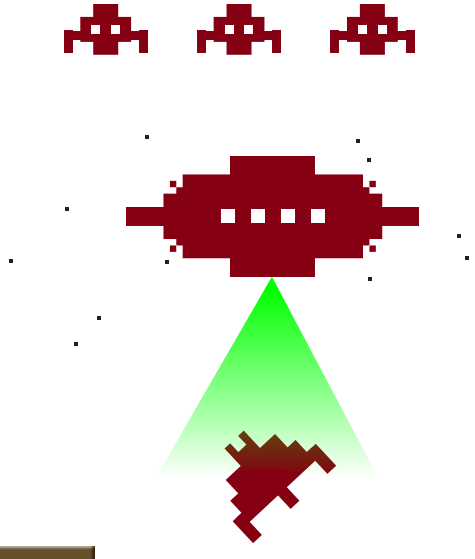
```
self.w, self.h = self.sprite.get_size()
self.x, self.y = position
self.x -= self.w/2
self.y -= self.h/2
```

Now, instead of defining things like the sprite images at the module level, we'll define them within the class, so that there is a separate collection of sprite images for each character. We can do similar things with the current position and anything else we need for the character to remember.

Finally, we'll create a `move_player()` method (adapted from the `move_player()` function we defined previously) to provide for a keyboard-controlled character. Then we'll need to add some symbols representing possible behaviors (currently just `AUTO` and `PLAYER`, but we might want to add more later). This results in the code in Listing 4 (`mouse7.py`), which gets us back to controlling the mouse within the PyGame window.

To Be Continued...

We've moved from simple scripting to object-oriented design, and we've been able to extend a simple "screensaver" program to something more like a game, with a degree of player control. Next month, we'll add backgrounds, another character, sound, an objective, and scoring, so as to make this into a real game.



TERRY HANCOCK is a writer and python developer with an interest in games, web applications, and a broader theoretical application of open source methodologies to fields of endeavor beyond software. He is also one of the founding directors of the Open Hardware Foundation.

Web programming with web2py

by Massimo Di Pierro

web2py is one of the most recent frameworks for web application development written in Python. While originally designed primarily as a teaching tool, it includes many features that make it attractive for rapid application development situations, too.

Introduction

web2py was originally designed as a didactic tool to teach what the author considers the most important issues in server-side web development:

- Data Persistence: sessions, cookies, database, and caching
- Security Issues: typical vulnerabilities and prevention
- Software Engineering Patterns, in particular the Model-View-Controller (MVC) pattern and form navigation patterns
- Debugging, Testing and Maintenance issues

REQUIREMENTS

PYTHON: 2.5, web2py -http://www.web2py.com

Useful/Related Links:

- Free web2py Appliances - http://mdp.cti.depaul.edu/appliances
- Interative WIKI FAQ - http://mdp.cti.depaul.edu/AlterEgo

- Web 2.0 technologies (AJAX, RSS, Web Services, and Internationalization)

The main design goal was to address the above issues and, at the same time, to provide a gentle learning curve for students and developers. For this reason, the framework was designed to require no installation, no configuration, no shell scripting, to have no dependencies, and to allow development, debugging, testing, deployment, database administration, and maintenance of applications via a provided web interface. web2py is distributed in both source code and binary versions. The binary versions (for Windows and for Mac) include the administrative interface, the interactive documentation, and also the Python interpreter, the CherryPy SSL-enabled WSGI web server, and the SQLite3 database.

These features alone make web2py unique in the world of web application frameworks, but it has additional features that are not found in other frameworks. For example, web2py makes no distinction between "debug" and "production" mode. All exceptions that occur and are not explicitly caught are logged, and can be retrieved by the administrator for debugging purposes. Applications can be byte-code compiled and distributed as closed source (the license allows it). Although web2py has a GPL 2 license, the license does not extend to applications developed with web2py unless they explicitly incorporate web2py code. The framework also includes helpers to generate HTML programmatically by mapping Python objects into HTML tags.

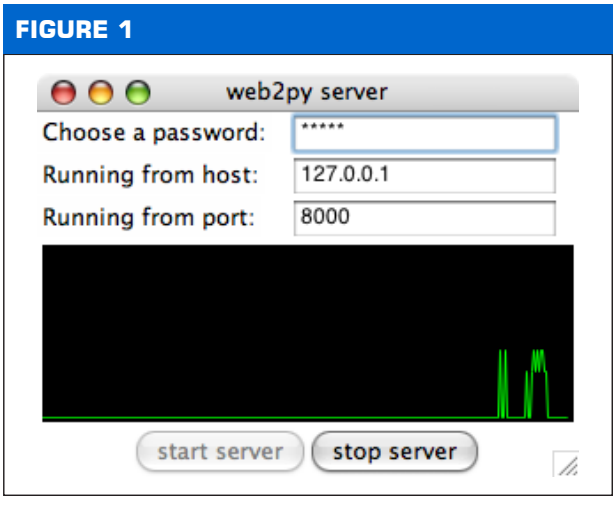
Despite its origin as a didactic tool, web2py has attracted users beyond the academic circle, and has grown in its capabilities. The current version, 1.33, includes libraries to handle HTML/XML, CSV, RTF, RSS, JSON, AJAX, REST, and WIKI markup. It can talk to SQLite, MySQL, PostgreSQL, and Oracle databases. Support for Sybase and MSSQL is in the works.

The latest version of web2py can also run on the Google App Engine, with the limitation that the ORM has to run on top of the Google Query Language instead of on top of SQL.

Although in this article we are focusing on the web-based administrative interface to web2py, the reader should bear in mind that it is possible to perform all of the operations described here using a Python shell and a normal text editor. The administrative interface is just an application that runs on top of web2py. There is also an Ajax-based Python shell available for download in the web2py repository.

After downloading web2py, follow the instructions to install and run the application, and you will be able to try out the examples in this article.

FIGURE 1



At startup

At startup, web2py presents the user with a simple TK GUI (see Figure 1). The GUI allows you to set parameters such as a one-time administrator password, start and stop the web server, monitor the server load graphically, and access the log file. In this article we will assume the web server is listening on 127.0.0.1 and port 8000, which is the default configuration.

web2py comes packaged with 3 *applications*, which we will refer to as *apps*:

- http://127.0.0.1:8000/admin is the administrative interface;
- http://127.0.0.1:8000/examples is the web-based interactive documentation;
- http://127.0.0.1:8000/welcome is the scaffolding application, and every new application is derived from this one.

After you log in through `/admin`, web2py presents a list of installed apps and allows you to either create a new one; install a packaged app from a tar file; pack an existing app as a tar file; create, update, and delete the files that comprise an app; and perform various maintenance tasks.

LISTING 1

```
import datetime; now=datetime.datetime.now()

db=SQLDB('sqlite://db.db')

db.define_table('page',
    SQLField('timestamp','datetime',default=now),
    SQLField('title'),
    SQLField('body','text'))

db.define_table('comment',
    SQLField('timestamp','datetime',default=now),
    SQLField('page',db.page),
    SQLField('author_name'),
    SQLField('author_email'),
    SQLField('body','text'))

db.define_table('document',
    SQLField('timestamp','datetime',default=now),
    SQLField('page',db.page),
    SQLField('name'),
    SQLField('file','upload'))

db.page.title.requires=[IS_NOT_EMPTY(),IS_NOT_IN_DB(db,'page.title')]
db.page.body.requires=IS_NOT_EMPTY()
db.comment.page.requires=IS_IN_DB(db,'page.id','%(title)s')
db.comment.author_name.requires=IS_NOT_EMPTY()
db.comment.author_email.requires=IS_EMAIL()
db.comment.body.requires=IS_NOT_EMPTY()
db.document.page.requires=IS_IN_DB(db,'page.id','%(title)s')
db.document.name.requires=[IS_NOT_EMPTY(),
    IS_NOT_IN_DB(db,'document.name')]

```

Model View Controller Design

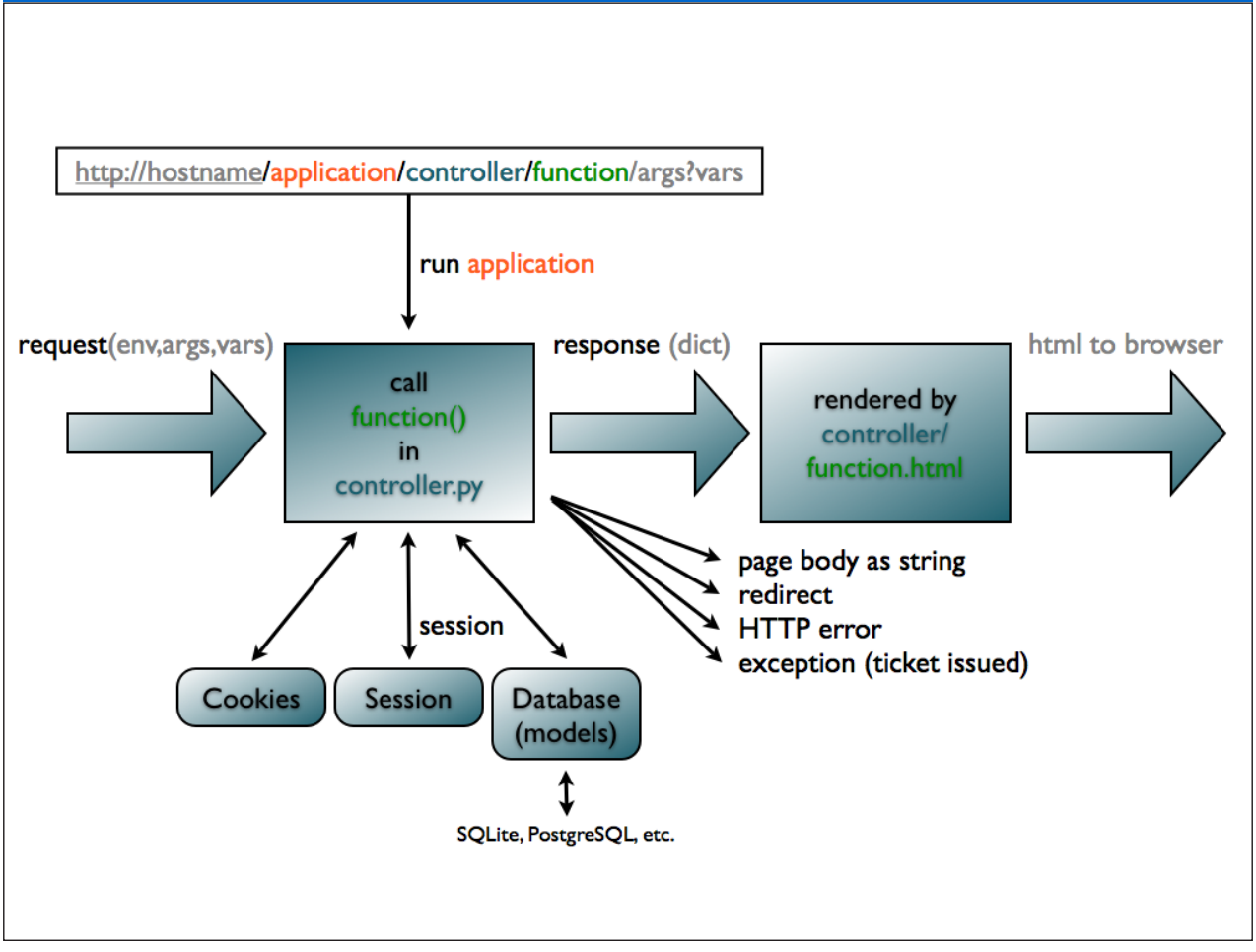
web2py follows a Model-View-Controller design (Figure 2) like all modern web frameworks, including Ruby on Rails, Django, TurboGears, and Pylons.

Models are files that contain a description of the data representation for the app. This includes a list of database tables, constraints on the fields, and shortcut functions to perform common database queries.

Controllers contain a description of the application logic and workflow. They are implemented as user-defined functions that are called when an associated URL is visited. A controller function can return a set of variables (in the form of dictionary), display a web page, stream a file, expose a service (xmlrpc, rss, flash, etc.), redirect the browser to a different URL, or return an HTTP error message.

Views are files that render the variables returned by a controller function into HTML. A view is an HTML file that embeds Python code delimited by special {{...}}

FIGURE 2



tags, in a way similar to Python Server Pages (PSP) or PHP. The difference in an MVC framework is that only presentation code should be placed in the view, while logic and control flow code should be placed in the controller. This separation forces the developer to organize his/her work in a way that is cleaner and easier to maintain than in PSP or PHP.

Take this URL as an example:

```
http://127.0.0.1:8000/a/c/f/x/y?v=value
```

It is mapped into a call to function `f()` in controllers/c.py in application `a`. The additional terms in the path, `x`, `y`, etc., are be stored in `request.args[0]`, `request.args[1]`, etc. GET and POST variable are stored as attributes of request.vars. For example `v=value` translates into `request.vars.v='value'`, and so forth. request.vars can also be accessed as a dictionary. All URLs are validated using regular expressions to prevent directory traversal attacks, malicious file execution, and insecure object reference.

Creating a New Application

As an example of the capabilities of web2py, we will create a wiki app that allows users to create pages, view and edit existing pages, post comments, and upload documents for linking in the page. Every page has a title, a body, comments, and attached documents. Once the basic wiki features are implemented, we will improve it by adding an AJAX-enabled search page, an RSS feed, and an XML-RPC service.

Create the new app using the form at the bottom of the `/admin/default/site` page. Type in a name - say, `mywiki` - and press the **submit** button. This results in an empty app that you can can modify via the **design** interface seen in Figure 3.

Building a Wiki: The Model

The next step is to create a model file called `db.py` using the "create file with filename" form. Place the code shown in Listing 1 into the new file.

db is a local object that represents a connection to the database, in the example a sqlite3 database stored in file `db.db`. The code defines three tables: `page`, `comment`, and `document`. If a table does not exist, web2py creates it; if it exists but differs from the above definition, web2py alters it.

In this example, both tables `comment` and `document` reference table `page` via a field also called `page`. This is an example of a one-to-many relation, which you can learn more about through the Wikipedia entry at http://en.wikipedia.org/wiki/One-to-many.

Consider the following line from Listing 1:

```
SQLField('timestamp', 'datetime', default=now),
```

The first argument of the SQLField constructor, "`timestamp`", is the field name. The argument "`datetime`" is the optional field type. The meaning of most of the field types, like "`text`" and "`datetime`", is obvious. The default, if you fail to specify these values, is type "`string`" with a length of 32. A field with the type "`upload`" is a special type of field used to store the name of an uploaded file. This name is automatically and securely generated by web2py upon

FIGURE 3

LISTING 2

```
def index():
    """ this controller returns a dictionary rendered by the view
        it list all wiki page
    >>> index().has_key('pages')
    True
    """
    mypages=db().select(db.page.id,db.page.title,orderby=db.page.title)
    return dict(pages=mypages)

def create():
    "creates a new empty wiki page"
    myform=SQLFORM(db.page,fields=['title','body'])
    if myform.accepts(request.vars,session):
        session.flash='page saved'
        redirect(URL(r=request,f='index'))
    elif myform.errors: response.flash='page not saved'
    return dict(form=myform)

def show():
    "shows a wiki page"
    try: thispage=db(db.page.id==request.args[0]).select()[0]
    except: redirect(URL(r=request,f='index'))
    db.comment.page.default=thispage.id
    myform=SQLFORM(db.comment,
                fields=['author_name','author_email','body'])
    if myform.accepts(request.vars,session):
       response.flash='comment posted'
    elif myform.errors:
       response.flash='incomplete form'
    pagecomments=db(db.comment.page==thispage.id).select()
    return dict(page=thispage,comments=pagecomments,form=myform)

def edit():
    "edit for for existing wiki page"
    try: thispage=db(db.page.id==request.args[0]).select()[0]
    except: redirect(URL(r=request,f='index'))
    myform=SQLFORM(db.page,thispage,deletable=False,
        fields=['title','body'])
    if myform.accepts(request.vars,session):
        session.flash='page saved'
           redirect(URL(r=request,f='show',args=request.args))
    elif myform.errors:
        response.flash='page not saved'
    return dict(form=myform)

def documents():
    "lists all documents attached to a certain page"
    try:
        thispage=db(db.page.id==request.args[0]).select()[0]
    except:
        redirect(URL(r=request,f='index'))
    db.document.page.default=thispage.id
    myform=SQLFORM(db.document,fields=['name','file'])
    if myform.accepts(request.vars,session):
        response.flash='document stored'
    elif myform.errors:
        response.flash= 'document not stored'
    pagedocuments=db(db.document.page==thispage.id).select()
    return dict(page=thispage,documents=pagedocuments,form=myform)

def download():
    "allows to download documents"
    import os
    try:
        filename = os.path.join(request.folder,
                                'uploads',
                               request.args[0])
        filehandle = open(filename,'rb')
    except:
        redirect(URL(r=request,f='index'))
    return response.stream(filehandle)

def find():
    "an ajax callback that returns a <ul> of links to wiki pages"
    pattern='%'+request.vars.keyword.lower()+'%'
    mypages=db(db.page.title.lower().like(pattern)).select(
       orderby=db.page.title)
    items=[A(row.title,_href=URL(r=request,f=show,args=[row.id]))
           for row in mypages]
    return UL(*items).xml()

def search():
    "an ajax wiki search page"
    form=FORM(INPUT(_id='keyword',
                    _onkeyup="ajax('find',['keyword'],'target');"))
    return dict(form=form, target_div=DIV(_id='target'))
```

upload to prevent directory traversal attacks.

Once you have defined a table, you will find that it exists as an attribute of the db object. For example, `db.page` is a reference to the page table, `db.comment` is a reference to the comment page, and `db.document` is a reference to the document page. `db.page.title` is the title field of a page table.

All tables have an automatic auto-increment integer field called `id`, (e.g. `db.page.id`).

At the bottom of the model we set some constraints on the field values. The most notable ones are

```
db.page.title.requires = [
    IS_NOT_EMPTY(), IS_NOT_IN_DB(db,'page.title')
    ]
```

which tells web2py that `db.page.title` has to be unique and not empty. `IS_NOT_EMPTY()` is a validator. The concept of the validator was inspired by the Django framework.

In the same fashion,

```
db.comment.page.requires = IS_IN_DB(db, 'page.id',
                                    '%(title)s')
```

tells web2py that the comment page must contain a valid `page.id`, and we prefer to have the `comment.page` represented by its `title` instead of its `id`.

These requirements are enforced at the level of the web forms generated by web2py. Moreover they influence the way the field is displayed. For example, the `IS_IN_DB` requirement results in a drop down menu with the possible page names.

Now click on the **Design** tab to go back to the design page. This page, under **Models** now shows a new link: **database administration**. Click on it.

FIGURE 4

The database administration pages have been automatically created for us. They allow you to search, create, update, and delete records for all of the application's tables, as illustrated in Figure 4. Notice that the database administration is part of the newly created application so it will inherit the look and feel of the new app and it can be edited without affecting other apps.

web2py includes an Object Relation Mapper (ORM) that abstracts the access to the specific SQL back-end and translates Python code into SQL. The ORM also escapes values that are passed to the database, in order to prevent SQL-Injection vulnerabilities. The SQL generated to CREATE, ALTER, and DROP tables can be seen by clicking on the **sql.log** button in the **design** page.

If you don't want to type your model out by hand, the web page *http://mdp.cti.depaul.edu/sqldesigner* allows you to visually create and download web2py models.

Building a Wiki: The Controller

Now that the model is finished, we can build the control flow for our application. Begin by editing the existing controller in `default.py` and define the following functions that correspond to URLs within the application:

- `index`: to list available wiki pages.
- `create`: to create a new wiki page.
- `show`: to show an existing wiki page, and to post and read comments.
- `edit`: to edit an existing page.
- `documents`: to list all documents attached to a page, and to upload new ones.
- `download`: to download a document.
- `find` and `search`: to search our pages
- `rss`: to generate a feed from out pages
- `find_by` and `handler`: to expose the search via XML-RPC.

LISTING 2: Continued...

```
87.
88. def rss():
89.     "generates rss feed form the wiki pages"
90.     import gluon.contrib.rss2 as rss2
91.     import gluon.contrib.markdown as md
92.     mypages=db().select(db.page.ALL,orderby=db.page.title)
93.     rss = rss2.RSS2(title='mywiki news',
94.        link = 'http://127.0.0.1:8000/mywiki/default/index',
95.        description="mywiki news",
96.        lastBuildDate=now,
97.        items = [
98.           rss2.RSSItem(
99.             title = row.title,
100.            link = URL(r=request,f='show',args=[row.id]),
101.            description = md.WIKI(row.body),
102.            pubDate = row.timestamp)
103.         for row in mypages
104.         ]
105.     )
106.    response.headers['Content-Type']='application/rss+xml'
107.    return rss2.dumps(rss)
108.
109. def find_by(keyword):
110.     "finds paged that contain keyword for XML-RPC"
111.     return db(db.page.title.lower().like('%'+keyword+'%')).response
112.
113. def handler():
114.    "exposes the index function via XML-RPC"
115.    return resonse.xmlrpc(request,[find_by])
116.
```

" The main design goal was to provide a gentle learning curve for students and developers."

Listing 2 includes the complete list of functions for the controller. Each function returns a dictionary of variables to be passed to the associated view (or the default generic view). For example:

```
return dict(pages=mypages)
```

tells web2py we want to pass the mypages object (a list of pages) to the view and we want to reference it via a new variable called pages inside the view context. The list of pages is created through a query:

```
mypages = db().select(db.page.id, db.page.title,
                      orderby=db.page.name)
```

The `select()` call perfoms an `SQL SELECT` on `db()`, i.e. on all records, for page.id and page.title. The orderby argument orders the records alphabetically by page name.

Consider the function `show()` on lines 19-28 of Listing 2. `myform` is a *create form* for the comment table (the first argument of the SQLFORM constructor indicates the table associated to the form). The form lists fields `author_name`, `author_email` and `body`. The `comment.page` for a newly created comment is set to `thispage.id`, the `id` of the page we are visiting.

`accepts()` is the most important method of a SQLFORM object. It performs validation upon

submission, adds error messages to the form in case of invalid submission, performs an SQL INSERT in case of valid submission, and prevents double form submission and replay attacks.

response.flash is a variable that is rendered by the default view layout with a flashing banner, normally to notify the user about events. If the flash has to be displayed after an HTTP redirection, then session.flash must be used instead of response.flash. The idea of flash was inspired by the TurboGears framework.

In this line:

```
myform = SQLFORM(db.page, thispage, deletable=False,
```

LISTING 3

```
1. VIEW FILE: default/index.html
2.
3. {{extend 'layout.html'}}
4. <h1>Available wiki pages</h1>
5. [ {{=A('search',_href=URL(r=request,f='search'))}} ]<br/>
6. <ul>{{for page in pages:}}
7.      {{=LI(A(page.title,_href=URL(r=request,f='show',args=[page.id])))}}
8. {{pass}}</ul>
9. [ {{=A('create page',_href=URL(r=request,f='create'))}} ]
10.
11. VIEW FILE: default/create.html
12. {{extend 'layout.html'}}
13. <h1>Create new wiki page</h1>
14. {{=form}}
15.
16. VIEW FILE: default/show.html
17.
18. {{extend 'layout.html'}}
19. <h1>{{=page.title}}</h1>
20. [ {{=A('edit',_href=URL(r=request,f='edit',args=request.args))}}
21. | {{=A('documents',_href=URL(r=request,f='documents',args=request.
args))}} ]<br/>
22. {{import gluon.contrib.markdown}}
23. {{=gluon.contrib.markdown.WIKI(page.body)}}
24. <h2>Comments</h2>
25. {{for comment in comments:}}
26.          <p>{{=comment.author_name}} on {{=comment.timestamp}}
27.           says <I>{{=comment.body}}</i></p>
28. {{pass}}
29. <h2>Post a comment</h2>
30. {{=form}}
31.
32. VIEW FILE: default/edit.html
33.
34. {{extend 'layout.html'}}
35. <h1>Edit wiki page</h1>
36. [ {{=A('show',_href=URL(r=request,f='show',args=request.args))}} ]<br/>
37. {{=form}}
38.
39. VIEW FILE: default/documents.html
40.
41. {{extend 'layout.html'}}
42. <h1>Documents for page: {{=page.title}}</h1>
43. [ {{=A('show',_href=URL(r=request,f='show',args=request.args))}} ]<br/>
44. <h2>Documents</h2>
45. {{for document in documents:}}
46.    {{=A(document.name,_href=URL(r=request,f='download',args=[document.
file]))}}
47.    <br/>
48. {{pass}}
49. <h2>Post a document</h2>
50. {{=form}}
51.
52. VIEW FILE: default/search.html
53.
54. {{extend 'layout.html'}}
55. <h1>Search wiki pages</h1>
56. [ {{=A('listall',_href=URL(r=request,f='index'))}}]<br/>
57. {{=form}}<br/>{{=target_div}}
58.
```

```
                fields=['title','body'])
```

myform is an *update form* for the page table, modifying thispage. The option deletable=False tells web2py that users can update, but can not delete, thispage.

The following code in the function download():

```
try:
    file = open(os.path.
        join(request.folder, 'uploads',
             request.args[0]),
        'rb')
except:
    redirect(URL(r=request,f='index'))
return response.stream(file)
```

builds the path to the requested file (assuming the requested filename is in request.args[0]). All uploaded files are, by convention, in the "uploads" subfolder of request.folder. The last line streams the requested file to the client. File streaming is a basic feature of web2py that allows it to handle very large files, both in upload and download, with no memory overhead.

On failure, every controller invokes:

```
redirect(URL(r=request,f='index'))
```

which redirects the visitor to the index page within the requested app and controller. The URL function is used to build URLs within web2py.

Notice that, just as we can access a model without creating a controller, web2py also allows us to access a controller without creating views (using the built-in "generic.html" view).

Building a Wiki: The Views

At this point our app is almost completely working and the individual controller functions can be accessed via their corresponding URLs. For example

- http:127.0.0.1:8000/mywiki/default/create lets us create a new wiki page
- http:127.0.0.1:8000/mywiki/default/show/1 lets us see it and post comments
- http:127.0.0.1:8000/mywiki/default/edit/1 lets us edit it
- http:127.0.0.1:8000/mywiki/default/documents/1 lets us see documents and post them, and
- http://127.0.0.1:8000/mywiki/default/index lists all wiki pages.

What is now missing is a presentation layer that tells web2py how we would like the data to be represented.

For example, page.title should be between <h1> tags, and page.body should be converted from WIKI markup to HTML. We also need to provide information about how each page should be linked to by other pages. Since this is a wiki, clicking on a document's name should call the download() function.

Every controller function (except download()) needs an associated view file. The web2py convention is that a view named default/index is rendered by a view called default/index.html. We proceed by creating views for our controllers, as shown in Listing 3. The listing contains views for each controller. The name of the view is at the top, like "default/index.html". To create the view, just type "default/index.html" in the **create view form** in the **Design** page.

There are a few web2py peculiarities to observe. Most of the views start with

```
{{extend "layout.html"}}
```

In fact, one view can extend another view and include other views. This feature provides a tree structure for our views. The template renderer starts from the view associated to the requested function, and walks its way up and then down this tree to build the completed page.

The web2py template language is pure Python, enclosed between {{...}} tags, with one caveat: blocks are not identified by indentation. They are delimited by the pass command. The reason is that views should be indented according to the HTML content, not according to Python code. Using pass to delimit blocks solves this problem and makes programming views easy and readable. In those cases where the end of the a block can be determined from the context, like in {{if cond:}}...{{else:}}, the {{pass}} statement is not necessary.

FIGURE 5

The use of {{...}} delimiters for embedded code is inspired by the Django template system and presents the advantage over other choices that it works with existing HTML editors, and it works with non-HTML views as well. Nevertheless, our template language differs substantially from the Django one since the latter only allows a subset of the Python syntax plus some special commands, while web2py's is pure Python code with no limitations.

Lines of code starting with =, like in {{=value}}, are evaluated, and the output is escaped and embedded into the HTML. The escaping prevents Cross-Site-Scripting (XSS) vulnerabilities. If value is a helper object, then it is not escaped; it is first serialized, and the strings that it contains are then escaped.

web2py defines a number of helper objects that have the same names as their corresponding HTML tags: A, B, FORM, INPUT, H1, etc. A helper object can be instantiated in the controller as well as in the view. The constructor of any helper takes three types of arguments: unnamed arguments, named arguments starting with underscore, and named arguments not starting with underscore. For example, in:

```
{{=A('create page',_href=URL(r=request,f='create')
)}}
```

"create page" is an unnamed argument, and comprises the inner HTML between the <a> and tags. _href is a named tag starting with underscore so it represent a tag attribute, with the leading underscore removed. The above object is serialized in HTML as:

```
<a href="/mywiki/default/create">create page<a>
```

In the above case, "create page" is escaped, while "/mywiki/default/create" is URL-encoded according to specificaitons.

Helpers can be nested to build complex self-aware serialize-able structures like:

```
FORM(B("label",_class="label"),
     INPUT(_name="myfield"))
```

Notice that helpers are serialized when embedded in {{=...}} meta-tags in the view, and they can be manipulated by Python code before being serialized. SQLFORMs are built using helpers.

FIGURE 6

"web2py includes an Object Relation Mapper (ORM) that abstracts the access to the specific SQL back-end"

FIGURE 7

WIKI is a special helper that is defined on top of the third party markdown2 module (which is why it needs to be imported even if it ships with web2py). The module `gluon.contrib.markdown` supports the WIKI markup language described in http://daringfireball.net/projects/markdown/.

Examples of the index, show, edit pages can be found in Figure 5, Figure 6, and Figure 7 respectively.

Notice that we did not need to worry about sessions and cookies, since web2py handled that for us. It created a file-based server-side session, locked it, and generated the corresponding session cookie. It is also possible to store sessions in the database (see the documentation for details).

Using AJAX to search the pages

Web2py's default installation includes the jQuery library to support AJAX interaction and other web effects. It also defines a JavaScript function called `ajax()` that takes as arguments the URL of a web2py controller function, a list of source HTML element ids, and a target HTML element id. When it is executed, the values of the source elements are passed to the controller function, asynchronously, and the returned value (a piece of HTML, XML, or a JSON message) is stored in the target element.

In Listing 2 we defined two controller functions, `find()` and `search()`. The latter one builds a page with a text input form and a `<div>` tag called `target_div`. When the visitor edits the text input form (`onkeyup`) `ajax()` is executed and it performs an asychronous request to `find()`. The value in the field is passed to this `find()`, which looks for all of the wiki pages that contain the text in the `title` and builds an HTML snippet containing a list of links to the found pages. The output of the function is then inserted in `target_div`. This mechanism provides a very interactive experience when searching our wiki pages.

The web page http://mdp.cti.depaul.edu/layouts allows the creation and download of new web2py layouts with matching color schemes.

Adding web services

web2py is distributed with a couple of libraries for making it easier to work with syndication feeds. feedparser is a very general library for parsing RSS and ATOM feeds, and RSS2Gen is a library for creating RSS2 feeds. In the `rss` controller function in Listing 2, we use

RSS2 to generate an RSS feed from the content of the db.page table.

In web2py, any function can be exposed as an XML-RPC service. For example, in Listing 2, the handler function exposes a function find_by() via XML-RPC. It takes a string argument, a keyword, and returns a list of database records (each of them represented by a list of field values) that contain the keyword in the title.

Now find_by() can be called by any program via XML-RPC libraries; for example, interactively, from the python shell:

```
import xmlrpclib
hander = xmlrpclib.ServerProxy(
    'http://host:port/app/controller/handler'
    )
records = handler.find_by('test')
```

The value of records is now a list of lists with the response from the XML-RPC service.

Debugging

Debugging and testing are two critical characteristics of Agile development and web2py brings them to a new level. Regarding debugging, web2py make no distinction between development and production modes. We believe an app should always in debugging mode, even when it is in production! This means all application exceptions which are not caught by the application code are caught by the framework and recorded. The visitor who stumbled upon the exception is issued a ticket number in case he or she needs to communicate with the site administrator. The administrator has access to all tickets and can browse them, sort them, read the traceback, browse the code that caused the traceback, and delete the corresponding ticket. This is all done via the administrative interface, locally or remotely. The ticket mechanism prevents the application from ever exposing code to the visitors, which is a common vulnerability of web applications.

Testing

web2py uses the standard library module doctest for testing, but exposes it through an AJAX testing web interface accessible via the **test** button in the **design** page. Tests are embedded in docstring comments in the code. For example, in our wiki index() controller function we have a doctest in a docstring that says:

```
""" this controller returns a dictionary
rendered by the view
>>> index().has_key("pages")
True
"""
```

The docstring instructs web2py that a call to index() should return a dictionary that contains an item called "pages". When you click on the **test** button, web2py locates all existing tests, runs them, compares the expected output with actual output, and writes a report listing passed and failed tests. Failed tests also show a traceback explaining why the test failed. Code is always shown using syntax highlighting, and web2py keywords are clickable for documentation. Figure 8 shows sample output produced when a test passes.

Adding Internationalization

Any text string can be marked for translation using T(). In the controller and in the model, this looks like:

```
T("Hello World")
```

while in the views it takes the form:

```
{{=T("Hello World")}}
```

In this case, "Hello World" is marked for translation. The translation is performed when the page is serialized. By default, the translation language is selected based on the request.env.http_accept_language header value.

The **design** page in the **admin** app contains a

FIGURE 8

Testing controller "default.py" ... done.

Function show [passed]

Function find [passed]

Function edit [passed]

Function download [passed]

Function index [passed]

Function find_by [passed]

Function search [passed]

Function create [passed]

Function handler [passed]

Function rss [passed]

Function documents [passed]

Languages section. Via this web interface, the developer can add or edit supported languages. web2py will identify all strings that are marked for translation and create a web interface to input the translations. web2py also provides a special simplified "translator admin" appliance which can be used to give access to the translation page to a translator without giving access to the full source code of the application that needs to be translated.

Performance

web2py is designed to be a fast framework. There are a number of features that contribute to make it perform well:

- Starting with version 1.22, web2py includes the CherryPy `wsgiserver`. Our tests indicate that it is more than a factor of two faster than the Paste server used by Pylons and earlier versions of web2py.
- web2py code, including view templates, can be bytecode compiled. The main advantage of byte-compiling is the views are parsed only once, with all the `{{extend}}` and `{{include}}` string substitutions. The entire tree structure associated with each view is compiled into a single Python bytecode file. When an HTTP request arrives, there is no parsing and no text substitution as in other frameworks. (I am not reporting benchmarks because results depend on the page complexity, and this topic deserves an article of its own).
- Static files are only served if they changed on the server when compared with the static file cached by the browser (IF_MODIFIED_SINCE protocol).
- web2py can serve/stream 203 PARTIAL CONTENT on 'Range' requests.
- web2py includes a powerful and granular caching mechanism that supports combinations of RAM-based, disk-based, and memcache caches. web2py allows you to cache the result of any function, controller, view, or database select query.

As an example of cache usage, in the index controller function in the wiki we could replace:

```
mypages = db().select(db.page.id, db.page.title,
                      orderby=db.page.title)
```

with:

```
mypages = db().select(db.page.id, db.page.title,
                      orderby=db.page.title,
                      cache=(cache.ram,600))
```

and cache the result of the select, in RAM, for 600 seconds.

Conclusions

web2py is a new product and, as such, it still needs to earn the trust of Python users. Nevertheless, web2py has been a successful tool at DePaul University where it is used to teach a graduate course on the design of web frameworks and a course on web server programming. This tool enabled students, most with limited or no previous web programming experience, to become productive after less than 30 hours of training. Students have developed a number of web2py apps such as a log analyzer with plotting capabilities, an online store, various WIKIs and blogs (including a WordPress clone), an AJAX chat room, a Flash-based administrative interface, and a podcasting web site. Some of these applications are posted (or will be posted soon) on http://mdp.cti.depaul.edu/appliances.

We wish to thank all the volunteers that have contributed and continue to contribute to the development of web2py, including the members of web2py Google group.

MASSIMO DI PIERRO has a PhD in High Energy Physics and for five years he has been a full time Professor of Computer Science at the School of Computing of DePaul University in Chicago, where he teaches a variety of programming topics. He also does research in Quantum Chromo Dynamics with support from the Department of Energy, and manages the Master of Science in Computational Finance. Massimo uses Python in his classes and research projects. As one can imagine he does not have much free time, but what he has he likes to spend developing Python code.

Welcome to Python

Using Future Statements to Prepare for Python 3

by Mark Mruss

With the release of Python 3.0 only a few months away many Python programmers have visions of compatibility problems dancing in their heads. This article will introduce two new keywords that you can use to help prepare your code for version 3.0.

Introduction

Python programmers have started thinking about transitioning to Python 3.0 (or Python 3000, as many people know it). Besides being the next major version of our beloved programming language, version 3.0 of Python will break backwards compatibility with the current 2.X branch of Python. This means that a some of the code that you are writing right now in Python 2.X won't immediately work in Python 3.0. Since the scheduled release date for Python 3.0 is September 2008 [1], now is a good time to start thinking about future migration.

Of course there's no reason to be alarmed. Guido himself has said that there is no rush to switch over to Python 3.0. [2] According to his PyCon 2008 essay *Python 3000 and You*, you should switch when the following are true: "1. You're ready 2. All your dependencies have been ported."

In the same essay, Guido also says that Python programmers should be prepared and that they should "start writing future-proof code for 2.5". In that sprit, this article will introduce *future statements* and two

REQUIREMENTS

PYTHON: 2.5+

Useful/Related Links:

- PEP 238 - http://www.python.org/dev/peps/pep-0238/
- Python 3000 and You - http://www.python.org/doc/essays/ppt/pycon2008/Py3kAndYou.pdf
- PEP 361 - http://www.python.org/dev/peps/pep-0361/
- Future Module - http://docs.python.org/ref/future.html
- PEP 328 - http://www.python.org/dev/peps/pep-0328/#rationale-for-absolute-imports
- Package Imports - http://docs.python.org/tut/node8.html#SECTION008420000000000000000
- Future Proofing Python - http://wiki.python.org/moin/FutureProofPython

ways that you can use them now in order to make the migration process from the Python 2.x to 3.0 as smooth as possible.

The rest of this article, and the examples within, will assume that you are working with Python 2.5.

Import the Future

A very important step in getting your code ready for Python 3.0 is to make use of the `__future__` module. __future__ allows you to both import changes that are going to be made in future version of Python and use them in your current version of Python.

For example, if feature X is slated to change to feature Y in the future, there is a chance that it may be supported by the __future__ module already. If so, you can import feature Y to use it in the version of Python that is still using feature X by default. This means that instead of having to alter your code when feature X is deprecated and feature Y is implemented, you can write your code using feature Y now, and not worry about when the switch will occur.

When you use the __future__ module you create *future statements*. Future statements, and the purpose of __future__, are defined in the Python documentation:

"A future statement is a directive to the compiler that a particular module should be compiled using syntax or semantics that will be available in a specified future release of Python. The future statement is intended to ease migration to future versions of Python that introduce incompatible changes to the language. It allows use of the new features on a per-module basis before the release in which the feature becomes standard." [3]

Future statements import the new features from the __future__ module. They must be at the top of your module; only comments, blank lines, or other future statements can come before them.

Let's take a look at a specific example that will let us work with the changes that will happen to the division operation.

Floating Division

In the current branch of Python the division operation can be somewhat confusing. It can be integer division (returning the floor of the division, i.e. only the integer quotient) or floating point division (returning the result as a float including the remainder).

Confusion within the current state of Python's division operation can be seen in the following example:

```
>>> print 3/2
1
>>> print 3/2.0
1.5
```

"Future statements import the new features from the __future__ module."

Here Python is looking at the type of the operands in the operation. If they are both of type int, integer division is performed as shown in the first result. If one (or both) of the values is a float, the operation will return a float that will include the remainder (if there is one).

Note that division of this sort (where the result type is dependent on the operands) is common in other programming languages.

While the correct type of division to be performed may be obvious when literal numbers are being used, unexpected results can occur when you are working with variables whose type may be unknown:

```
def divide_by_three(value):
    return value/3
```

Calling `divide_by_three()` will result in either an integer or float return value, depending on the type of the argument `value`.

As a result of this confusion, Python 3.0 changes the division operator to make the return type consistent. All division using a single slash (/) will return a floating point number including the remainder. In order to perform integer division, you will need to use two slashes (//). The type of division performed now depends on the number of slashes used and not the type of the operands. The only gotcha to be aware of is that if one of the operands in the division operation is a float, the return value will also be a float with 0 as the remainder.

To use the new division operator in Python 2.5, you can use a future statement like:

```
from __future__ import division
```

This changes the implementation of the division operation, in whatever module the future statement occurs, from its current integer form to its future float form.

Listing 1 uses a future statement to import the changes to the division operation and replicates our previous division test. If you were to run the code found in Listing 1 the output would be as follows:

```
1.5
1.0
```

If you are writing code where you perform division of any sort, I'd suggest importing division from __future__ so that when the move to Python 3.0 occurs, you won't find yourself with any strange results.

Note that the // division operation has been in Python since version 2.2 so you can already use it

" Python 3.0 will break backwards compatibility with Python 2.x ... Of course there's no reason to be alarmed."

without importing division from __future__. Importing division from __future__ changes the functionality of the / division operation.

The Future of Imports

In Python 3.0 all imports are absolute. This means that when you perform an import, you will import "a module or package reachable from sys.path." [4] While at first glance this may seem to make importing local modules impossible, it is important to remember that when you launch a Python script file directly, the directory where that script file is located is added to the start of sys.path in position 0.

To illustrate the reason for the change to absolute imports, let's take a look at a silly example. Let's say we had a project that used a `tokenizer` package using a directory tree like:

```
tokenizer/
tokenizer/__init__.py
tokenizer/parse.py
tokenizer/string.py
my_proj.py
```

The file tokenizer/string.py is a helper module (with an unfortunate name) that the parse.py module needs to use.

In this example my_proj.py won't do anything besides import the tokenizer package and create an instance of a Parse object:

```
import tokenizer

if __name__ == '__main__':
    parser = tokenizer.Parse()
```

The __init__.py module in the tokenizer package imports the Parse class from the parse module. This is done so that the Parse class can be accessed as tokenizer.Parse as opposed to tokenizer.parse.Parse:

```
from parse import Parse
```

Finally, parse.py is where all of the magic happens:

```
import string

class Parse(object):
    def __init__(self):
        print(string)
```

Well it's not really magic! All that the parse.py file does is import the string module. When a Parse instance is created, a string representation of the string module that was imported is printed out to the command line.

In other words, what happens here is:

- my_proj.py imports the tokenizer package
- which then imports Parse from parse.py
- my_project.py then creates an instance of the Parse object
- The Parse instance will then print out a string representation of the string module it imported.

When we run my_proj.py, we get the following:

LISTING 1

```
1. #!/usr/bin/env python
2. from __future__ import division
3.
4. def main():
5.   print 3/2 #Float division with two ints
6.   print 3//2.0 #int division with two floats
7.
8. if __name__ == '__main__':
9.     main()
10.
```

```
<module 'tokenizer.string' from
'/home/project/tokenizer/string.py'>
```

As we can see, our local string module was imported by the parse.py module. This is a relative import which means that the "import statement first looks in the containing package before looking in the standard module search path." [5]

So What's the Problem?

I'm sure that many of you realize that there is a `string` module built into Python, and that our `string` module and the built-in module share the same name. As long as everything in the tokenizer package only needs the tokenizer.string module, and not the built-in string module, everything will be fine.

But what happens a few months down the road when some new module in the tokenizer package needs to import the built-in string module? Whenever it tries to import `string` it gets `tokenizer.string` instead. The obvious solution is to make sure that your module does not share the same name as a built-in module. But this does not fully solve the problem, as we can't anticipate the names of future modules that will be added to the standard distribution.

" In Python 3.0, all imports will be absolute."

Absolute Imports

PEP 328 describes the issue explained in our tokenizer example and the solution that the Python team has put in place:

"In Python 2.4 and earlier, if you're reading a module located inside a package, it is not clear whether import foo refers to a top-level module or to another module inside the package. As Python's library expands, more and more existing package internal modules suddenly shadow standard library modules by accident. It's a particularly difficult problem inside packages because there's no way to specify which module is meant. To resolve the ambiguity, it is proposed that foo will always be a module or package reachable from sys.path. This is called an absolute import." [6]

As stated earlier, once Python gets to version 3.0 all import statements use absolute imports. We can already use this functionality in Python 2.5 by using a future statement to import absolute_import:

```
from __future__ import absolute_import
```

When we add this future statement to parse.py, it will look like:

```
from __future__ import absolute_import
import string

class Parse(object):
    def __init__(self):
        print(string)
```

When we run my_proj.py, the output will be something similar to the following:

```
<module 'string' from '/usr/lib/python2.5/string.
pyc'>
```

As you can see, enabling absolute imports changes import string from a relative import into an absolute import. This solves our problem when we want import string to import the built-in module, but what about when we want to import the local string module in the tokenizer package? The solution is to use an absolute import to import the local module. The following version of parse.py will import the local string module:

```
from __future__ import absolute_import
from tokenizer import string

class Parse(object):
    def __init__(self):
        print(string)
```

We can also use the following absolute import to import the local string module:

```
from __future__ import absolute_import
import tokenizer.string

class Parse(object):
    def __init__(self):
        print(tokenizer.string)
```

Relative Imports in an Absolute World

Once you start using the absolute_import feature, you may still want to use relative imports. Relative imports generally are not needed in smaller packages. But if you have a very large package that contains many subpackages, you might not want to rewrite your import

statements every time the internal structure of the main package changes. Fortunately, it is still possible to perform relative imports in Python 3.0 or with absolute_imports enabled, using the following syntax:

```
from . import foo
```

Notice that a dot is used to indicate that the import is a relative import. "A single leading dot indicates a relative import, starting with the current package. Two or more leading dots give a relative import to the parent(s) of the current package, one level per dot after the first." [7] The "dot syntax" is only available to imports that use the "from" syntax. "import . foo" is not allowed.

To import our local string module in the parse.py module with absolute imports turned on, we do the following:

```
from __future__ import absolute_import
from . import string

class Parse(object):
    def __init__(self):
        print(string)
```

Conclusion

I hope I have convinced you that importing division and absolute_imports from the __future__ module is an easy way to add a little "3.0 protection" to your current Python projects. Deciding to make these changes as you create new code, will make migrating your code to Python 3.0 an easier process when the big day comes.

Of course these are not the only changes that you should start making to your code to prepare for Python 3.0. For example, some of you may have noticed that I was using the print function in this article and not the print statement. Since the print statement is no longer going to be available in Python 3.0, I'm trying to get used to using the print function. See the Python wiki for a more detailed description of all the upcoming changes to Python 3.0, and steps you can take to prepare your code now. [8]

You don't have to worry about Python 3.0 too much yet, since the true migration path is through Python 2.6 and the tools that will be made available. In fact for some people full migration to Python 3.0 may be many years away. But like Guido said, it's best to be prepared and start writing future-proof code now. [9]

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.

FOOTNOTES

[1] http:www.python.org/dev/peps/pep-0361/
[2] http://www.python.org/doc/essays/ppt/pycon2008/Py3kAndYou.pdf
[3] http:docs.python.org/ref/future.html
[4] http:www.python.org/dev/peps/pep-0328/#rationale-for-absolute-imports
[5] http:docs.python.org/tut/node8.html#SECTION008420000000000000000
[6] http:www.python.org/dev/peps/pep-0328/#rationale-for-absolute-imports
[7] http:www.python.org/dev/peps/pep-0328/#guido-s-decision
[8] http://wiki.python.org/moin/FutureProofPython
[9] http://www.python.org/doc/essays/ppt/pycon2008/Py3kAndYou.pdf

Random Hits

Technology Trends

by Steve Holden

Steve peers through a glass darkly at some technology futures and tries to discern the meaning of the various parallel changes that are overtaking us at the moment.

Some while ago now Tim Bray wrote a thought-provoking blog post called *Multi-Inflection-Point Alert*. In it, Tim examined the more significant technology trends and observed that for some of them at least the solutions are being determined *right now*. There are some interesting philosophical fine points that could be debated, but I'd like to focus on the meat of his argument. Tim's premise was that many things are at a critical rate of change at the same time, and the flux in the information technology world is likely to be with us for a good while yet as the issues it raises are decided.

Programming Languages

His thesis is that you'd be nuts not to look seriously at PHP, Python, and Ruby for all kinds of programming

REQUIREMENTS

PYTHON: 2.2+

Useful/Related Links:

- Multi-Inflection-Point Alert - http://www.tbray.org/ongoing/When/200x/2008/04/24/Inflection
- HAVING a Blunderful Time, or, Wish You Were WHERE - http://www.dcs.warwick.ac.uk/~hugh/TTM/HAVING-A-Blunderful-Time.html

tasks, where in past years the dynamic languages would have been regarded as unsuitable for many programming tasks. This is true, and it's personally gratifying to someone like me who pretty much eschewed the static languages where possible in favor of the esoteric ones like SNOBOL, Icon, SmallTalk and more recently Python.

The most effective strategy has always been to use the language that is best suited to purpose and as purpose changes, so should language. Of course, one problem is that many commercial programmers only know one language and aren't going to be comfortable in an environment where people say "Well, Python's obviously the tool for this task even though so far we've written everything in C". If you don't command a number of languages, you aren't even qualified to make such a decision. If you hold a Java hammer then all problems tend to look like Java nails.

The trend in software is more and more toward component architectures. As software systems become more like a collection of interacting components, the emphasis will be on interfaces rather than languages, and we will be much freer to choose our language according to our needs rather than the constraints of the environment. Microsoft's work on the DLR and object sharing between different languages is interesting in this context. Ultimately, end-users are more interested in the data than the algorithms. They couldn't care less whether their answers are produced in Python or by writing detailed instructions for a gang of pixies that live inside their computer. They just want to know what "the answer" is.

We're lucky that Python appears to be at the focus of much of the front line effort in this area, but that isn't an accident. It's due to the care with which Guido has designed the language, making it applicable to a wide range of problems and suitable for experience levels from beginner upwards. Its highly modular nature and flexible typing lends it to the development of, and interaction with, component architectures.

Database

Tim says "SQL's brain-lock on the development community for the past couple of decades has been actively harmful, and I'm glad it's now OK to look at the alternatives." He lists CouchDB, SimpleDB and BigTable as possible technologies to look at. I'm not clear on whether he's complaining about the language or about relational systems, and I'd have liked a little more clarification on that point.

CouchDB is a flat collection of "documents", each of which consists of a collection of named fields. As I understand it, there is no requirement for each of the documents in the collection to contain the same fields, and field values can be ordered lists, as well as the more usual primitive types. By contrast, values in the relational model cannot be repeated items but must be "atomic" types.

SimpleDB is Amazon's answer to the relational database and apart from the ability (again) to have multi-valued attributes, it is pretty relational in its approach. It's specifically not intended to store large objects, which Amazon contends should be saved in S3, and it doesn't do joins (which makes it pretty inflexible compared with an RDBMS).

BigTable is a more involved model and works with essentially two-dimensional data. Each cell in the table is versioned, allowing history to be maintained through a given number of versions or a specific length of time. It distributes the tables over multiple machines, but the row keys are lexicographically ordered, which means that a single machine will be able to return all rows for a small range of row keys. For queries over wider ranges of key values the load is inherently balanced across multiple computers, aiding scalability. Clearly this is not the relational model, but something optimized for Google's specific purpose, which may well be useful for others.

I still see relational stores as the most useful *general* storage model we have, however. When specific issues (usually issues of performance) arise then one can denormalize relatively easy, but there is nothing else as flexible and as powerful as the relational model, which will be with us for a long time to come. SimpleDB and CouchDB are fine for particular purposes, and a relational design can be denormalized to fit them when appropriate, but there will be many applications for which they will not be a good fit.

SQL is indeed the bastard child of relational algebra and grunge language design. Hugh Darwen and Colin Date, among others, have told us exactly what is wrong with it - most entertainingly in Darwen's *HAVING a Blunderful Time, or, Wish You Were WHERE*, expanded and amplified in their book *The Third Manifesto*. But very few people who criticize SQL do so because they dislike the relational model, they do so because SQL, which was a botch from the start, does not truly implement Codd's concepts of relational algebra and doesn't fully conform with that model. People tend to forget that relational architectures won out because unlike the

network and hierarchical models that preceded them, they could adapt to changes in processing without restructuring. That advantage would be lost by moving back to non-relational stores, which are, at best, point solutions to specific-performance problems.

Another blogger commented that they've been making heavy use of Oracle's DB XML, effectively a refinement of the Berkeley DB for hosting XML. Seeing all the current abuses of XML, I shudder to think of the horrendous architectures that are going to be visited upon us by the XML database (which will likely be popular). Is the Web an HTML database? Relational theory tells us that "XML databases" will be a mistake, but some people will resolutely refuse to get that point.

Network Programming

It's interesting that Bray confesses he has ceased to be a threading advocate, implying that he sees multi-process architectures as more flexible (which they are). Once you accept that your separate components live in separate address spaces, though, *all* large programming problems become network programming problems, and the only remaining differences are differences of locality, bandwidth and latency. The problems are engineering problems, and two-phase commit probably figures in there somewhere, too, despite the trend toward statelessness. The database is, in fact, just a huge state store. Deal with it.

Guido van Rossum has been saying for a long time to anyone who complained about the global interpreter lock (GIL) and demanded the ability to partition their task over multiple threads, that they should be looking at multi-process solutions. The advantage to that approach is not only can you scale over multiple cores in the same computer, but also across networked computers. Clearly, there will be problems that don't occur with the threading model, but overcoming the difficulties is important in the general case and the **processing** library does a lot to iron out the differences by providing a threading-compatible interface that makes it easier to run across multiple processes.

Yet, still, people complain about the GIL. I am forced to reluctantly conclude that some people just like complaining.

Processors

Bray asserts, with some degree of truth, that we still haven't figured out the right way for ordinary people to program many-core processors (see **Network Programming** above). Many of the remaining problems are actually problems of communication, but the Python world certainly has its share of great ideas. Twisted is a brilliant solution to single-processor asynchronous programming and Stackless adds the ability to pickle tasklets, migrate them to a different processor, and continue running them there. The Twisted Perspective Broker offers similar features.

Python, therefore, has a great head start on these tricky issues and there has even been some work done on gluing Twisted and Stackless together. That will be quite a combination when it happens since Stackless is the only framework I know that allows the development of Python applications to run across 200-node clusters.

I believe that in the future, the desktop applications we use day-to-day (well, the ones that need to harness multiple CPUs, anyway) will be largely indifferent to the location of processing power; the processing might be local, but equally it might be a grid elsewhere on the Internet. Forget the GIL: the future is definitely multi-process, and the GIL just won't be an issue then. Whether those processes are on your local machine or somewhere else isn't really going to matter.

Web Development

Bray sees a pretty uniform picture of Rails' market share advancing steadily. He doesn't think it will last, and I believe he's right. The problem with Rails isn't that it uses MVC with a relational store; that's an advantage, in that it usefully separates the unrelated facets of web applications. The problem is, like many frameworks, its architecture is limited, and as such, the wider the range of problems it's pointed to the more its deficiencies will be highlighted and the clunkier its complex solutions will become.

Rails was a great solution for 80% of the web–the 80% that is mostly CRUD data manipulation and straightforward reporting. That aspect was plugged by viral marketing that got people enthusiastic about it, but those were the people who hadn't already had to come to terms with the limitations of other frameworks before it, which similarly ran out of power. The remaining 20% of required functionality will tax Rails just as it taxes other frameworks, and there are signs that Rails is running out of steam, as it inevitably would. The web really needs architectures where the client- and server-side are better decoupled, but AJAX is only the beginning.

I'd like to say that Pylons, TurboGears, and Django represent the ultimate in web frameworks, but the web is the area where things are currently moving fastest and have furthest to go. I think we will see web technologies embedded in several areas we haven't imagined so far, and some of the portal solutions such as iGoogle are beginning to demonstrate the potential of the portal approach.

Business Models

Bray points out that the market is currently determining how to deploy software-as-a-service, and which applications it suits better than traditional client/server or standalone desktop models. While it's true that the cloud does have economic advantages, many of the applications for which grid computing will be a compelling requirement haven't been developed yet. I don't think the advantages are mostly economic at present.

When I use Google documents, for example, I do so because it's valuable to be able to access them from anywhere and share them with a geographically distributed community, not because it's cheaper. Google's App Engine has brought deployment on the cloud closer still than Amazon's services, but we still have a long way to go before it's a painless, seamless transition from the desktop to the cloud. That won't stop vendors from selling it before it's a reality, of course, and the early adopters will doubtlessly form a viable market.

Eventually there will come a time when we don't just want to produce reports on our computers. Instead we will want to design a new gadget (using software provided by the gadget manufacturers) and have a photorealistic and possibly holographic rendering of the design before we send the production data to the numerically controlled machining facility. People may find it hard to believe that this time will come (assuming the human race doesn't drown in its own sewage before then), but fifty years ago very few people thought that Western society would be pervaded by pocket devices that allow access to all the stored knowledge on the world's vast information resources and communication to anyone around the world. Perhaps we aren't there yet, but the iPhone is a pretty good start (apart from its determinedly proprietary setting, which is a licensing rather than a technology issue).

As far as the business model goes, you can be sure that less and less cash will be flying around the more technically up-to-date countries, so payment technology will always be important.

Desktops

The Bray take on this aspect of computing is to wonder how long people will continue spending money on inferior environments. The simple truth is that Microsoft's marketing has persuaded people to adopt technically inferior solutions. This should be no surprise, as the average purchaser doesn't make purchasing decisions on the technical issues. Right now Microsoft is frantically looking for a business model where the margins are as good as their traditional (operating system and desktop productivity) markets. Unfortunately, software has become a commodity while they've been stashing cash in the bank, and they are now ill- equipped to deal with the new reality, which is essentially a slow, but steady migration to an open source infrastructure. Although they now have an OSI-approved open source license, they still don't see (or can't afford to acknowledge) that Windows' heyday is over.

I am always cautious about declaring "the year of the Unix desktop", since Sun had plans for Unix to take over the desktop when I was working for them back in 1985. But have you compared the Windows and Linux installation processes recently? Even the various BSD distributions are easier to install than ever before, and the market is getting more computer-savvy all the time. Not this year, but soon, Linux will be the default choice of operating environment. This will happen in the corporate environment first, and make its way more slowly into consumer markets. I repeat an assertion I have made many times before: Microsoft will be the first IBM of the 21st century–still a significant player, but increasingly sidelined when it comes to setting the technological direction.

Will It Always Be Like This?

Bray says it's early days for technology yet, and so we can't expect these issues to be solved yet. He's right, but he appears to overlook the point that the speed of technology introduction is getting faster all the time in an exponential curve which appears to have no end. Even when these questions *are* decided, they will be replaced by others as pressing and apparently as crucial. It's comforting to tell ourselves that things will settle down once the standards are defined, but standardization (ignoring the OOXML disaster) is mostly a codification of existing engineering best practices.

Once a standard is defined, though, different people layer technologies over it in different ways, with

competing advantages. Some time ago, I made a joke about desktop components using HTTP to communicate with each other over the loopback interface. That seems less improbable by the week. Nobody really knows yet what the information bus of tomorrow is going to look like, but we can guarantee it's going to run at gigabit speeds in the local area and hundreds of megabits per second in the wide area because it already does, and nothing ever gets slower. Inefficiency of communication becomes less of an issue when you have bandwidth to burn.

People sometimes talk about technology battles as though there was an all-time "winner" and "loser". That inadequate two-dimensional view of the marketplace ignores the fact that technology remorselessly moves on. The Betamax war was replaced by the BlueRay war. Both have now been won (and lost), and we march to the next theater of operations and pitch our tents for the next skirmish. This is happening in many areas of technology in parallel.

I have wondered about whether initiatives like the OLPC project are really attacking the right problems: a computing infrastructure seems an unnecessary luxury when faced with the absence of other infrastructures that we take for granted in our cosseted lives. The one thing none of us should do is forget that we are an extremely privileged minority. For us, an inconvenience is having to slow down to DSL or T-1 speed when we visit Starbucks for our no-fat latte with extra foam. For others, inconvenience is having to walk three miles to the water source and back twice a day just to be able to cook and wash.

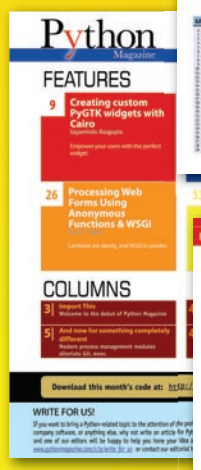
This cozy technological cocoon we inhabit has the ability to blind us to the plight of our fellow human being. If we allow it to, then we are truly living in the Matrix.

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