Python Magazine

MAY 2008

learning python with pygame

The Simplest Thing that Can Possibly Work

■ Handling Configuration Files with ConfigObj
Take care of all your advanced configuration needs easily with ConfigObj

■ Writing a Simple Interpreter/Compiler with Pyparsing
If writing parsers leaves your brain in knots, pyparsing can help untangle the mess

■ LDAP backed initScripts in Python
Bring order and consistency to your data center with LDAP and Python

Python Magazine

CONTENTS

Download this month's code at:
http://code.pythonmagazine.com/2/5

FEATURES

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

In my day job, I code a decent bit of Python, but I'm also coding lots of PHP and doing SQL development, and mucking about with Perl on occasion, and... wishing I could do everything with Python, basically. At night, I sometimes wonder to myself "Where can I get training to really polish my Python skills so I can do more with it, more efficiently?".

While I'm ecstatic when I hear that Python is becoming a language of choice in Computer Science departments here and there, I'm maybe slightly surprised and disappointed that there aren't many local training centers offering Python training. I live in the most densely populated state in the nation (NJ - we all have to live somewhere), and I am hard-pressed to find even a single offering. If I want training, I'd probably have to travel, probably either to Washington DC, where I might convince Steve Holden to teach me for a somewhat hefty sum (though perhaps payable in part with a case of MacAllan 18 - but alas, that case is almost done), or Colorado, to learn from Mark Lutz (author of "Learning Python" for O'Reilly & Associates).

I caught a mention in Steve's column this month about Python certification, and the tone of the comment made it seem to me that it was perhaps not a favorite topic. I hadn't ever considered certification, because I find it orthogonal to what I want to achieve, which is increased productivity with Python. Certifications, in my experience, have little to do with how well you can use the language, but how well you know the material for the certification exam. But, as I often do, I took a moment to poke and prod at that sort of negative take on things.

How can certification be good for me, and good for Python, even if I don't want a certification?

It wasn't obvious to me, but hidden behind all of the Exam Cram booklets and cheesy certification marketing might be an opportunity for us both. If Python had some kind of certification guidelines (or even a licensable course that could be given by third parties), it would be seen as an opportunity for training centers to expand their horizons. Eventually, that course I've been looking for might be just around the corner!

In addition, with training available, the language may appear more attractive to more development shops. It is not uncommon for development shops to send students to training, groups at a time, on a rotating basis, as an investment in both their people and the technology they use. A somewhat lame certification may not help the businesses much directly, but having training available is one way that businesses measure the viability of a proposed technology.

So, who benefits from the certification itself? Getting a certification

Python Magazine

MAY 2008
Volume 2 - Issue 5

Publisher
Marco Tabini

Editor-in-Chief
Brian Jones

Technical Editor
Doug Hellmann

Contributing Editor
Patrick Bryant, Steve Holden

Columnist
Mark Mruss

Graphics & Layout
Arbi Arzoumani

Managing Editor
Emanuela Corso

Authors
Doug Hellmann, Jon Miller, Mark Mruss, Michael Foord, Paul McGuire, Steve Holden, Terry Hancock

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

could help some people in some situations prove that they meet at least some minimal level of knowledge with the language. If a hiring manager is looking for a “Junior Python Programmer”, that manager may well take a certified candidate over a non-certified candidate. He might well be getting a worse candidate, but people aren’t likely to question his decision to hire a certified programmer, and if someone pursued a certification, it probably is more than a hobby to them. So, issues with certifications aside, some people can get some good from them I’m sure.

One thing I do know is that figuring out how to put together a training session, developing the materials, figuring out how to deliver the training, and all of that is all child’s play compared to drawing up specification guidelines to define the scope and boundaries of a certification, especially one that might bear the PSF logo (if that’s been considered).

While that’s all being tossed around on mailing lists I’m probably not on, I suppose I’ll continue my search for training, and in the meantime I’ll keep reading Python Magazine and trying to spend as much free time as I can doing new things with Python, and working it into my daily routine at my day job as well.

Now, on with the show!

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

AND NOW FOR SOMETHING COMPLETELY DIFFERENT

by Doug Hellmann

Not content to leave well enough alone, Doug offers up some extensions to Ian Bicking's virtualenv script that make it even more useful.

Back in February, I introduced Ian Bicking's **virtualenv** script for creating isolated Python environments, complete with their own interpreter and `site-packages` directory. I've been using virtualenv for all of my projects since then, but keeping up with all of the environments I have been creating has turned a little messy. I've solved the problem by writing a few wrappers in the form of bash functions. If you are a virtualenv user, you may find them useful, too.

virtualenv

As I mentioned in the previous column, bringing you interesting articles about new projects from the far reaches of the internet means that I end up installing a lot of Python packages to review code before it is published. That's okay, but I want my system to be usable for working on my own development projects, too. Of course, creating a separate "sandbox" for each article or project means I can install whatever libraries I need, and not worry about version conflicts or deleting something accidentally. It's almost like Ian wrote virtualenv specifically for me.

For new subscribers who missed the February column, here's a quick primer in virtualenv: Every time you run virtualenv, it sets up a clean copy of Python in a new directory by copying or linking files from your primary Python installation to create new `bin` and `lib` directories. To use the sandbox, you simply run `source $path/bin/activate`. Your environment is then reconfigured so that the version of Python in the virtual environment takes precedent over the normal version installed globally on the system. Since you

REQUIREMENTS

PYTHON: *2.x*

Useful/Related Links:

- virtualenv
  http://pypi.python.org/pypi/virtualenv
- virtualenvwrapper
  http://www.doughellmann.com/projects/virtualenvwrapper/

LISTING 1

```
# Make sure there is a default value for WORKON_HOME.
# You can override this setting in your .bashrc.
if [ "$WORKON_HOME" = "" ]
then
    export WORKON_HOME="$HOME/.virtualenvs"
fi

# Verify that the WORKON_HOME directory exists
function verify_workon_home () {
    if [ ! -d "$WORKON_HOME" ]
    then
        echo "ERROR: $WORKON_HOME does not exist!"
        return 1
    fi
    return 0
}

# Create a new environment, in the WORKON_HOME.
#
# Usage: mkvirtualenv [options] ENVNAME
# (where the options are passed directly to virtualenv)
#
function mkvirtualenv () {
    verify_workon_home
    (cd "$WORKON_HOME"; virtualenv $*)
    workon "${@:-1}"
}

# Remove an environment, in the WORKON_HOME.
function rmvirtualenv () {
    typeset env_name="$1"
    verify_workon_home
    env_dir="$WORKON_HOME/$env_name"
    if [ "$VIRTUAL_ENV" == "$env_dir" ]
    then
        echo "ERROR: You cannot remove the active environment."
        return 1
    fi
    rm -rf "$env_dir"
}

# List the available environments.
function show_workon_options () {
    verify_workon_home
  ls "$WORKON_HOME" | egrep -v '*.egg' | sort
}

# List or change working virtual environments
#
# Usage: workon [environment_name]
#
function workon () {
  typeset env_name="$1"
  if [ "$env_name" = "" ]
    then
        show_workon_options
        return 1
    fi

    activate="$WORKON_HOME/$env_name/bin/activate"
    if [ ! -f "$activate" ]
    then
        echo "ERROR: No activate for $WORKON_HOME/$env_name"
        return 1
    fi

    if [ -f "$VIRTUAL_ENV/bin/predeactivate" ]
    then
        source "$VIRTUAL_ENV/bin/predeactivate"
    fi

    source "$activate"

    if [ -f "$VIRTUAL_ENV/bin/postactivate" ]
    then
        source "$VIRTUAL_ENV/bin/postactivate"
    fi
  return 0
}
```

created the directory yourself, you have all the permissions you need to install new modules or libraries into the environment. And since the environments are light weight and easy to create, you can have as many of them as you want.

Too Many Virtual Environments

So far, so good. An average user would probably create a new sandbox for each ongoing project, with some temporary environments thrown in once in a while for testing a new version of a library or playing with something discovered through the Python Package Index feed. I, on the other hand, found myself creating 4-6 new environments every few weeks as I created a separate environment for each article being reviewed for the magazine. The number of environments I had quickly grew to be unmanageable.

I started with all of my virtual environments in the same directories that held the other files for the article, so I could easily find the `activate` script used to enable the environment. This way of working meant I had environments scattered all through out my `Documents` folder, though, and I had to cd to each directory to make sure I was looking at the right environment before activating it. Another complication was the temporary environments were mixed together with "real" files that made up the articles and other parts of the magazine. Since we use version control to manage those files, I had to constantly tell the version control tools to ignore the virtual environment files.

I decided the first change I needed to make was to just put all of the environments together in one place, so I could see them all easily and select the right one quickly. I created a directory, `~/.virtualenvs` to hold all of my new virtual environments, and moved my existing environments there. I gave each one a name based on the project or article it was associated with to make it easy to tell them apart.

To create a new environment, all I had to do was:

```
$ cd ~/.virtualenvs
$ virtualenv newname
$ cd -
```

which I eventually shortened to just:

```
$ (cd ~/.virtualenvs; virtualenv newname)
```

I type pretty quickly, but I'm still averse to repeating myself. The only part of the command that changed each time I made a new environment was the `newname` argument to virtualenv. It looked like a perfect opportunity to use a wrapper to streamline the command. I thought about using a shell alias, but my bash alias-fu isn't all that strong and I wanted to include some basic error checking. Creating a *shell function* however, was very easy, and combined all of the benefits of an alias with the syntax of a script.

As these "simple" projects tend to do, this one quickly grew to include a few more features. Eventually it reached the point where my desire to hack on it was satisfied and I could actually use it. The results, called **virtualenvwrapper**, are available in Listing 1.

Managing Environments

The bash script in Listing 1 is intended to be run during your shell login sequence via the `source` command. The distributed version of the file is (unimaginatively) named `virtualenvwrapper_bashrc`, so to use it you would add a line like this to the end of your `~/.bashrc` file:

```
source $HOME/bin/virtualenvwrapper_bashrc
```

Of course the actual path depends on where you put the file when you download it. There is just the one file, so I keep a copy in `$HOME/bin`, but you might have other standard practices for extension scripts like this one.

The only other setup steps you need to take before using the wrapper is to define the shell variable `WORKON_HOME` to point to the directory where your virtual environments will go. The default value is `~/.virtualenvs`, and you don't have to change it if you want to put them somewhere else. You can place the setting before or after you source the script, so you would end up with something like this:

```
export WORKON_HOME="$HOME/Devel/Environments"
source $HOME/bin/virtualenvwrapper_bashrc
```

Once you have both commands in your login script, just `source ~/.bashrc` and then you can start creating environments. `mkvirtualenv` is a very thin wrapper around virtualenv itself that creates an environment and then immediately activates it.

```
$ mkvirtualenv newenv
New python executable in newenv/bin/python
Installing setuptools....................done.
(newenv)farnsworth:dhellmann:~:502 $
```

In fact, all of the arguments you give to `mkvirtualenv` are passed directly to virtualenv, so you can use `-h`, `—no-site-packages`, and all of the other normal options.

Switching Environments

One of the primary reasons I created virtualenvwrapper was to make it easier for me to alternate between different environments from the same shell. The `workon` function is the interface for switching, named that way because I use it when I want to "work on" a different project. To see a list of available environments, run `workon` with no arguments. To switch to an environment, give the name as an argument.

```
(newenv)$ workon
2.5
CastSampler
PyGameTutorial
docket
loghetti
newenv
personal
pymag
pymotw
```

As you see here, most of the available environments are for my own personal projects. You may recognize `PyGameTutorial` as being the environment I have been using to test Terry Hancock's code for his tutorial series, the first installment of which appears elsewhere in this issue. To switch to that environment, I simply run `workon PyGameTutorial`:

```
(newenv)$ workon PyGameTutorial
(PyGameTutorial)$
```

To switch to the sandbox I use for writing the "Python Module of the Week", I would run:

```
(PyGameTutorial)$ workon pymotw
(pymotw)$
```

Fancy Switching

Sometimes, as with the PyMOTW series, I want to change more about the shell environment than just the virtual environment settings. For example, I have a working directory where I create all of the code for PyMOTW, and when I "switch" to working on it, I want my shell to go there. To allow those sorts of actions, I added a feature to `workon` to look for hook scripts inside the `$VIRTUAL_ENV/bin` directory, and run them before and after activating the environment.

Before you move out of an environment, you usually want to save the project in your editor, clean up temporary files, and otherwise preserve its current state. The predeactivate hook offers an opportunity to clean up the *current* environment, before switching to the new one.

If the postactivate script is present, it is run after the new environment is configured. So to do any extra customization for an environment, you can simply create the postactivate script with the commands and they will be run automatically. You might want to open a new project file in your editor, set your terminal window title, or take any number of other customization steps.

Both of the hook scripts are sourced in the current shell, rather than being run as a new program, to give them an opportunity to change active environment variables. You can use them to change your PATH or re-configure your shell prompt, for example. Anything you can do from the shell command line can be done in the pre- and post-activation scripts.

In the case of PyMOTW, I have a simple call to cd in the postactivate script. I include a similar command in the postactivate script for each temporary environment I create for a new article, to make it easy to move around within the magazine source directories.

Cleaning Up

The problem that led me to create these shell functions in the first place was that I had too many virtual environments floating around my hard drive. At this point I have described tools I wrote to make creating new environments even *easier*. I also added a simple function, rmvirtualenv to make removing environments safe and easy.

Removing an environment is simple with rm, but after accidentally removing my *working* environment once, I thought the extra check was worth a new command. rmvirtualenv lets you remove any environment by name, as long as it is not the currently activated environment.

```
(pymotw)$ rmvirtualenv pymotw
ERROR: You cannot remove the active environment.
(pymotw)$ rmvirtualenv newenv
(pymotw)$ workon
2.5
CastSampler
PyGameTutorial
docket
loghetti
personal
pymag
pymotw
```

Conclusions

Some readers may wonder why I gave the shell functions mkvirtualenv and rmvirtualenv such long names if I was trying to avoid doing so much typing. Using tab completion at the shell prompt, I never have to type the whole command. mkvirt or rmvirt followed by TAB expands to the complete command name.

Although virtualenvwrapper meets my needs today, there are a few more features I would like to add, eventually. No software is ever actually *done*, right?

rmvirtualenv really needs another safety switch to make sure the user means it. It could, for example, list all of the modules or eggs installed in the private site-packages directory and then prompt the user to make sure they really want to remove the environment.

workon should offer a "none" option, to clear the virtual environment settings from the current shell without switching to another one. For now I work around that by having an empty "2.5" environment that mirrors my installed site-packages.

As I have said before, virtualenv has become an integral part of my toolbox. Whether for my own projects or editorial reviews, the ability to create a clean scratch area where I can install software with impunity gives me a sense of freedom that working directly out of /usr/local just can't match. I hope you give virtualenv a try, and if it suits you, maybe virtualenvwrapper will be a useful addition as well.

Next Month

Next month this series will continue with coverage of more tools to enhance your programming productivity. If you have a tip to share, feedback on something I've written, or if there is a topic you would like for me to cover in this column, send a note with the details to doug.hellmann@pythonmagazine.com and let me know, or add the link to your del.icio.us account with the tag pymagdifferent.

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

Handling Configuration Files with ConfigObj

by Michael Foord

ConfigObj is a powerful and flexible configuration file parser suitable for most configuration needs. This article explores the basic use of configuration files, and some of the more advanced features that make ConfigObj the right choice for your application.

Virtually every non-trivial application has to handle configuration data. There are a variety of options for storing it, ranging from using Python files to XML to a database. Each approach has its strengths and weaknesses, but many programmers default to plain text files in the simple "INI" format. This format has its origins in Windows 3.1, but its basic key-to-value pairing is so easy for users to work with that it has become a de-facto standard for configuration files on other platforms as well.

Naturally the Python standard library provides a module for working with INI files: **ConfigParser**. This article is about an alternative module for working with configuration files called **ConfigObj**. But why should you be interested in an alternative? Well, ConfigParser can be slightly awkward to work with. It isn't difficult to use, but can make you do more work than necessary for what should be a simple task. More importantly, ConfigParser has some serious limitations. In this article we'll be looking at how to use ConfigObj for

REQUIREMENTS

PYTHON: *2.3+*

Other Software:
- ConfigObj
  http://www.voidspace.org.uk/python/configobj.html

Useful/Related Links:
- ConfigParser
  http://docs.python.org/lib/module-ConfigParser.html
- YAML - *http://www.yaml.org/*
- JSON - *http://json.org/*
- configobj-develop mailing list
  https://lists.sourceforge.net/lists/listinfo/configobj-develop

configuration file handling, including some of its more advanced features.

In order to follow the examples in this article you will need to have ConfigObj installed. You can either use `easy_install configobj`, or download it from the ConfigObj homepage.

The Advantages of ConfigObj

ConfigObj comes out of my very first days of learning Python, where it started life as a set of functions in a project called 'Atlantibots'. Ironically if those of us involved had known that the standard library had a configuration parser, ConfigObj may never have been written.

The latest version of ConfigObj was written by Nicola Larosa and I, with contributions from many others, of course. It is widely used, even by some large projects like:

- Bazaar - a distributed version control system
- Turbogears - a web application framework
- Chandler - a Personal Information Manager
- IPython - an enhanced interactive Python shell
- matplotlib - a Python 2D plotting library
- Trac - online project management and issue tracking
- Elisa - an open source, cross-platform media centre solution

The biggest advantage of ConfigObj is simplicity. Even for trivial configuration files, where you just need a few key-value pairs, ConfigParser requires them to be inside a 'section'. ConfigObj doesn't have this restriction, and having read a config file into memory, accessing members is trivially easy.

Let's look at a simple example. Given a configuration file called "config.ini" and containing these members:

```
name = Michael Foord
# this comment belongs to 'DOB'
DOB = 12th August 1974 # an inline comment
nationality = English
```

We access it with the following snippet of code. After initializing a ConfigObj instance with a filename, you can then access members using dictionary like syntax:

```
>>> from configobj import ConfigObj
>>> config = ConfigObj('config.ini')
>>> config['name']
'Michael Foord'
>>> config['DOB']
'12th August 1974'
>>> config['nationality']
'English'
```

Beyond simplicity, ConfigObj has some more important advantages over ConfigParser. These include:

- Unicode support
- List values
- Multi-line values
- Nested sections (subsections) to any depth
- When writing out config files, ConfigObj preserves all comments and the order of members and sections
- Many useful methods and options for working with configuration files (like the `merge()` and `reload()` methods)
- An *unrepr* mode for persisting Python basic types
- An integrated validation and type conversion system. This allows you to check the validity of configuration files and supply default values.

Some of these you can take advantage of without having to do anything special in your own code. Particularly useful are multi-line values (with familiar triple quotes) and comma separated list values. Let's expand our simple config file with a couple of new members:

```
description = """
A hairy individual.
But with many redeeming features."""

attributes = 2 arms, 2 legs, "nose, slightly large"
```

When we access these members programmatically they become:

```
>>> config['description']
'A hairy individual.\nBut with many redeeming
features.'
>>> config['attributes']
['2 arms', '2 legs', 'nose, slightly large']
```

Sections and Writing

ConfigObj can handle config files with sections, in fact as we noted earlier it can handle sections nested to whatever horrific depth you desire. As the syntax for nesting sections is different from the usual (loosely defined) INI format, no introduction to ConfigObj can be complete without showing it:

```
name = Michael Foord
DOB = 12th August 1974
```

```
nationality = English

[Favourites]
   food = Steak & chips
   color = Vaguely Purple

   [[software]]
       ide = Wing
       os = Undecided
```

The 'Favourites' section has a 'software' subsection. Sections are indicated using the section name in square brackets. Subsections are made by increasing the number of matching square brackets around the section name.

Note that the whitespace is not significant in this example, but, like code indentation, visually shows the structure of the information.

As ConfigObj uses dictionary-like access (more properly it uses the 'mapping protocol'), sections are exposed as sub-dictionaries.

```
>>> config = ConfigObj('config.ini')
>>> config['Favourites']['food']
'Steak & chips'
>>> software = config['Favourites']['software']
>>> software['ide']
'Wing'
```

When you access the 'Favourites' member of the ConfigObj instance it returns a reference to a `Section`, a dictionary-like object. In fact, a `Section` is a subclass of `dict` and ConfigObj itself is a subclass of `Section`. All the dictionary methods are available on sections (including the top-level ConfigObj instance).

In addition to being able to load files and access the members, you can use the same API to modify members and create new ones. If you then call the `write()` method of ConfigObj, the changes are written back out. When writing files, ConfigObj does its best to preserve any comments that were in the original, as well as the order of members, and even the newline terminators used. It is also trivially easy to create entirely new config files programmatically:

```
config = ConfigObj()
config.filename = 'new_file.ini'
config['name'] = 'Fred Smith'
config['Favourites'] = {}
config['Favourites']['color'] = 'Mostly Red'
config['Favourites']['software'] = {'ide': 'Emacs'}
config.write()
```

The above example creates a fresh ConfigObj instance and then sets the filename on it. Passing in the filename when instantiating ConfigObj is just as valid. New sections are created by setting a member to be a dictionary; either an empty dictionary or with pre-existing members. Finally, the config file is written out by a call to the `write()` method.

There are lots more details you could learn about the syntax of files that ConfigObj works with and the methods available. The goal of this article is not to cover all of these exhaustively, but to look at some of the advanced uses.

"The biggest advantage of ConfigObj is simplicity."

Unicode Support

Support for Unicode is one of the factors driving the use of ConfigObj in several projects. The internet has shrunk the world dramatically, making it easy for people all across the world to access your projects. If your application works with text, but you aren't aware of text encodings (and aren't using Unicode) then it is probably broken in subtle ways.

In our first example, ConfigObj read a text file and parsed it to separate the keys from values. Because we did not specify the encoding, the file was treated as a byte stream. If the file included any multi- byte encodings, like UTF8 and UTF16, they would have been parsed incorrectly because characters would be split on byte boundaries rather than character boundaries.

The best way of handling this is to use known character encodings for your config files and have ConfigObj decode them to Unicode for you. UTF8 is of course a great encoding to use, as it can represent the whole range of Unicode characters and it is backwards compatible with ASCII – using only one byte per character for the ASCII characters.

```
>>> config = ConfigObj('config.ini',
... encoding='UTF8')
>>> config['name']
u'Michael Foord'
```

This automatically decodes the keys and values into Unicode. If you later modify and write out the config file then the members will be encoded again using the encoding you specified when the ConfigObj was instantiated.

You can modify the encoding after parsing the input file by changing the encoding attribute on the ConfigObj instance. This argument/attribute pattern is used repeatedly to control its behavior. When you instantiate ConfigObj you specify how the file will be parsed. Most of the available options correspond to attributes that you can modify later. Changing the options changes what happens when you save a config file with `write()`. For more details see the *options* and *attributes* sections of the ConfigObj documentation.

If you don't know the encoding of the text files you are using, then you can use a heuristic approach to guess. I have written about techniques for guessing encodings previously (*http://www.pyzine.com/Issue008/Section_Articles/article_Encodings.html*). Using Unicode in config files will be easier in Python 3, where strings are Unicode by default, but it won't disappear altogether.

From discussing text encodings, it is a natural step to look at how we can use ConfigObj to store data other than strings.

" Part of the power of validation is how easy it is to write custom checks and use them in your configspecs."

Unrepr Mode

There are, broadly speaking, two different use cases for configuration files. The first is for retrieving options set by the user, this requires mainly read-only access (unless you also provide a user- interface for changing the settings). The second way of using config files is for persisting internal configuration data, and rather than plain strings, these may include data structures. If you want this data to be human readable (and possibly human editable as well), then you need some kind of text serialization protocol.

The Python pickle module does have a text mode, but unless your brain can do a reasonable impression of a stack-based interpreter the output isn't human readable. XML serialization is another option, but only barely more readable in my opinion. Two good alternatives are YAML and JSON. Both of these have good Python support, but are specialized for storing data-structures. ConfigObj has a special *unrepr* mode that makes it a good third option, particularly if what you really need is a mapping of names to data.

The unrepr mode was originally implemented for Turbogears, but has also proved popular with other projects. In this mode ConfigObj uses Python syntax for values in the configuration file. Any of the following Python data-types can be stored in values, including nested containers: integers; floats; complex numbers; scalars like None, True, and False; strings; tuples; lists; and dictionaries. The consequences of using Python syntax are that strings *must* be quoted and you can use the Python string escaping rules. Lists and dictionaries both use their normal Python syntax.

A typical section of a configuration file using unrepr mode looks like this:

```
kid.encoding = "utf-8"
tg.allow_json = False
tg.mapping = {'/one': 'one', '/two': 'two'}
```

When this is read in with unrepr mode on you can probably anticipate the result:

```
>>> config = ConfigObj(filename, unrepr=True)
>>> config['kid.encoding']
'utf-8'
>>> config['tg.allow_json']
False
>>> config['tg.mapping']
{'/one': 'one', '/two': 'two'}
```

The main departure from Python syntax in unrepr mode is the use of triple quoted strings. These will have the quotes removed first, so you can use triple quoting to spread long entries (like dictionaries or lists with many members) across multiple lines. If you write out the config file though, ConfigObj won't write out triple quotes and will put each value on a single line (using Python escaping rules for newlines). It may not surprise you to hear that `repr()` is used to create the string representation of values, and the operation is reversed when they are read back in again. What you *can't* store using unrepr mode are types and instances of user defined classes. The advantage of unrepr mode is that you get type marshaling for free. The disadvantage is that it uses a different and more rigorous syntax for configuration files.

ConfigObj also supports a more sophisticated way of doing type marshaling called *validation*.

Validation

Configuration files, by their very nature, are text files, which means working with strings. When you work with the configuration data inside your application, only some of it will actually be strings. Even if your configuration data is all user-supplied, at some point you are going to need to convert data to Python types. You may also want to check that the data supplied by the user is valid, that all the required options are present, and that values fall within correct bounds or make sense.

ConfigObj provides a mechanism called validation that does all of this. It also allows you to specify default values, so that users only need to supply members that differ from the defaults.

Validation is done in conjunction with an extra module, **Validate**, that comes in the zipfile distribution of ConfigObj, or can be downloaded separately. It isn't required to use ConfigObj; you only need it if you are using validation. To validate a configuration file, you provide a specification for what it should contain. The specification is usually from a text source, but *could* be built programmatically, and describes the type of value you expect for each key in your configuration file. You can also include any bounds or constraints on the value, and an optional default value.

When you perform validation, each of the members in your specification are checked, and they undergo a process that converts the values into the specified type. Missing values that have defaults will be filled in, and validation returns either `True` to indicate success or a dictionary with members that failed validation. The individual checks and conversions are performed by functions, and adding your own check function is very easy.

Before we look at how to use them with ConfigObj, let's take a look at the specifications themselves which are referred to as *configspecs* in the ConfigObj documentation. A configspec has the same basic layout as a configuration file, and in fact they follow the same structure as the configuration file they are intended to check. They use the same `key = value` format and the same section/subsection markers. The difference is that the value for each key is a type and constraint specification.

Here's a simple example:

```
name = string
age = float
attributes = string_list
likes_cheese = boolean
favourite_color = string
```

This is a simple mapping of keys to expected type. To validate a config file using this configspec is simple. Under the hood ConfigObj parses the configspec in the same way it does config files. This means we have various options as to how we provide the configspec. The two most obvious ways are either to pass in a filename pointing to the configspec or pass it in as a list of lines from the file (strings).

When a ConfigObj instance is created with a configspec we use a *validator* to do the validation. The standard `Validator` from the validate module has lots of useful checks built into it, so you can often use it directly. In a short while we will look at creating new checks and adding them to the validator, but first refer to Listing 1 for a basic example.

LISTING 1

```
import sys
from configobj import ConfigObj
from validate import Validator

config = ConfigObj('config.ini', configspec='configspec.ini')

validator = Validator()
result = config.validate(validator)

if result != True:
    print 'Config file validation failed!'
    sys.exit(1)

```

When this code runs, and validation succeeds, then we know that the `config.ini` file contains the specified members and that they have been converted to the specified types after being read into memory. If validation fails we probably want to provide a more informative error message, which we can do using the dictionary returned by `validate()`.

Checks with Arguments

A lot of the validation checks can also take arguments to specify bounds or constraints on the values as well as type. For example, you can specify minimum and maximum length for strings, a valid range of values for numbers, minimum and maximum lengths for lists, and so on. There is also a useful "option" check that allows us to specify that a value must be from a specific set of options.

Here's an alternative configspec that uses these arguments:

```
name = string(min=1, max=30)
age = float(min=0, max=200)
attributes = string_list(min=5, max=5)
likes_cheese = boolean()
favourite_color = option('red', 'green', 'blue')
```

These checks are starting to look suspiciously like function calls and, unsurprisingly, they do map directly to functions that receive the arguments provided in the checks. As in Python code, the keyword arguments used above are optional if you are providing both a `min` and a `max`, but they can be useful to make the intent of the arguments clear.

There are a couple of subtle points to notice in this example. In the second example, the boolean check includes parentheses. For checks without arguments, parentheses are optional. In the `attributes string_list()` settings we specified a list with exactly five entries by setting both the maximum and minimum length to five.

Default Values

Another important use of checks is to provide default values where a member is not present in a config file. This means that your end users only need to supply values that are different from the defaults. Default values are set by passing in the keyword argument `default` as part of the check.

```
name = string(min=1, max=30, default=Fred)
age = float(min=0, max=200, default=29)
attributes = string_list(min=5, max=5,
default=list('arms', 'legs', 'head'))
likes_cheese = boolean(default=True)
favourite_color = option('red', 'green', 'blue',
default="red")
```

There are several points worth drawing out here. Default values can be unquoted or use single or double quotes. As the checks come from text, the default value also has to go through the check function to be converted to the right type. This means that your default must pass the validation check! If you supplied 201 as the default age (which should have a maximum of 200), then a configuration file with a missing age would fail validation even though age has a default. The exception to this rule is where you specify a default of `None` (without quotes). When `None` is used as a default value it will always pass checks, so it can be a useful way of checking for missing values. Where the check indicates that the configuration option should be a list, the default value has to be a list. A default list can be set by using the list constructor syntax in the default, as you can see in the default value for the 'attributes' member above.

If we create a fresh ConfigObj instance using the above configspec, and then validate, you can see that the default values have been filled in:

```
>>> config = ConfigObj(configspec='configspec.ini')
>>> validator = Validator()
>>> config.validate(validator)
True
>>> config['name']
'Fred'
>>> config['age']
29.0
>>> config['attributes']
['arms', 'legs', 'head', 'body', 'others']
```

...checks and conversions are performed by functions, and adding your own check function is very easy.

It is worth mentioning that if you write out a config file, any values that were supplied by default values from validation (i.e. they were missing from the original file) *won't* be written back out. This can actually be a problem. The code we have just used could be a great way to create fresh config files. Simply create a new ConfigObj instance with a configspec providing default values, then validate. Unfortunately calling `write()` will then create an empty file for us. If you *want* ConfigObj to include default values when writing then you need to specify `copy=True` when you call `validate()`. In the example above, that means simply calling `config.validate(validator, copy=True)`.

So far we have only covered the cases when validation is successful. Let's see how we can make error reporting more helpful when validation fails.

Error Handling

There are basically two kinds of errors you might want to handle. The first is where the syntax used in the config file is invalid and parsing fails. This is basically a fatal error (at least as far as configuration is concerned), because invalid syntax makes it hard to know how much of the configuration file was read in correctly. The second type of error is where the config file was read in fine, but some of the values were either missing altogether or invalid in some other way.

You may not be able to do much about the first kind of error, but you probably want to be able to catch it to report to the user. When ConfigObj fails to read in a config file it raises an exception. There are various different kinds of exceptions it can raise (a DuplicateError for duplicated keys or sections, a NestingError for malformed section headers, and a ParseError for badly formed entries to name a few). These are all subclasses of ConfigObjError, so the simplest thing to do is to instantiate the ConfigObj in a 'try...except' block that catches this error. Normally if the config file pointed to by the filename you provide doesn't even exist, ConfigObj *won't* complain (using ConfigObj to create new config files is perfectly valid). If you would rather this condition raised an error, you can use the `file_error` keyword argument. To handle both kinds of error you can do the following error handling:

```
from configobj import ConfigObj, ConfigObjError
try:
   config = ConfigObj(filename, file_error=True)
except (ConfigObjError, IOError), e:
   print 'Could not read "%s": %s' % (filename, e)
```

The next kind of error is when validation fails. This doesn't raise an exception, but instead the `validate()` method returns a dictionary that tells you which members failed validation. The dictionary essentially follows the structure of the config file, with sections and subsections as dictionaries keyed by their name. Each member will have the value `True` (if validation succeeds) or `False` (if validation for that member fails). If a whole section validates then it will just have `True` in the results dictionary instead of a sub-dictionary. It is easy to write a function that recursively walks the results dictionary and reports (or stores) the ones that failed.

To make it easier for you, ConfigObj comes with an example implementation of a function called `flatten_errors()` that does most of this for you. You can use it as-is, or look at the implementation for inspiration. The function walks the results dictionary and returns a list of all the keys that failed. For each failed member, the list tells you what section or subsection the setting was in and the failed member name. If a whole section was missing then the member name will be `None`.

Listing 2 contains an example of how to use `flatten_errors()` to report validation errors.

Repeat Sections

One problem with validation is that it requires you know the names of your sections. Some applications use the names of sections in config files as part of the configuration data. ConfigObj allows you to validate files like this by providing a specification that will be used for all subsections in a section (including the root section if you want).

Refer to Listing 3 for part of a fictional server configuration. We can validate the 'sites' section of the configuration file with a section called '__many__':

LISTING 2

```
from configobj import ConfigObj, flatten_errors
from validate import Validator

config = ConfigObj('config.ini', configspec='configspec.ini')
validator = Validator()
results = config.validate(validator)

if results != True:
    for (section_list, key, _) in flatten_errors(config, results):
        if key is not None:
            print 'The "%s"key in the section "%s" failed validation' % (key, ', '.join(section_list))
        else:
            print 'The following section was missing' % ', '.join(section_list)

```

LISTING 3

```
[sites]

    [[voidspace]]
        directory = '/home/voidspace/webapps/www.voidspace.org.uk/'
        domain = 'voidspace.org.uk'
        subdomains = 'www', # single member list
        active = True
        ip = '68.54.95.48'

    [[resolverhacks]]
        directory = '/home/voidspace/webapps/www.voidspace.org.uk/'
        domain = 'resolverhacks.net'
        subdomains = 'www',
        active = True
        ip = '68.54.95.48'

    [[ironpython]]
        directory = '/home/voidspace/webapps/www.voidspace.org.uk/'
        domain = 'ironpython.info'
        subdomains = 'www',
        active = True
        ip = '68.54.95.48'

```

```
[sites]

    [[__many__]]
        directory = string
        domain = string
        subdomains = string_list
        active = boolean
        ip = ip_addr
```

When you validate, the '__many__' section in the configspec is used on every sub-section in the 'sites' section.

Writing a Custom Check

Part of the power of validation is how easy it is to write custom checks and use them in your configspecs. Checks are functions that you register with the validator. They receive the value being checked (plus any arguments passed in the spec), and should either return the converted value or raise an exception to indicate that the check failed.

Listing 4 illustrates a custom check that uses a regular expression to test that a value is an email address. As it isn't doing type conversion, it will return the value unchanged if the value matches, or raise an exception if it doesn't.

LISTING 4

```
import re
from validate import ValidateError

email_re = re.compile('\w+@\w+(?:\.\w+)')
def email_check(value):
    if isinstance(value, list):
        raise ValidateError('A list was passed when an email address was
expected')
    if email_re.match(value) is None:
        raise ValidateError('"%s" is not an email address' % value)

    return value

```

Check functions receive the value and arguments as strings (from the config file) or possibly a list of strings for list values. `ValidateError` is the correct exception to raise when the input is invalid. (Note – I don't actually recommend checking emails using this regular expression. This is for example purposes only!) Having written the check function, you pass it into the `Validator` constructor as a dictionary mapping the check name to the check function.

```
validator = Validator({'email': email_check})
```

You specify this check in the configspec from the check name:

```
contact_email = email()
```

Nice and simple!

Conclusion

ConfigObj is a deceptively simple tool for working with configuration files and persisting application data. Although the basic pattern can be illustrated with a handful of lines of code, even in this article we haven't covered everything.

The ConfigObj documentation is exhaustive, but here we have expanded out a few of the more advanced ways you can use it, particularly with validation and configspecs. There is plenty we haven't covered, like interpolation and some of the methods available to you like `merge()` and `walk()`. If you have any questions (or heaven forbid, find any bugs) then the mailing list is a friendly place to go for help.

MICHAEL FOORD has been a Python programmer for five years. For the last two years he has worked for Resolver Systems developing a spreadsheet development environment with IronPython. He has written many articles on Python and IronPython, as well as maintaining several Open Source Python projects. He is currently writing a book for Manning Publications called IronPython in Action.

LDAP backed initScripts in Python

by Jon Miller

How often have you had to correct someone's init script? Perhaps the application your company bought didn't come with an init script at all. Or maybe you host an application developed in-house, and the development team isn't too savvy with creating run control scripts. What if you could manage all of your add-on init scripts with a single Python script, and store the configuration data centrally in an LDAP server?

Introduction

How many times have you had to compensate for a lacking application and build an *init script* to enable it to start when the server boots? This is an obvious task for any daemon that needs to be started upon booting your server, but it is surprisingly common for developers to forget to include one with their application. It is common to build your own scripts when working for a company that uses in-house software to provide services to its customers. As a system administrator, you typically end up with the job of ensuring that the application starts properly and that it remains running. Writing an init script is a mundane task, but once you have completed it, you should also educate your co-workers on what the script does and how to configure and run it. All in all, the process is arguably ugly and if you are working within a team with a range of skill levels, the end result is that your server environment may resemble chaos (a state that any system administrators should strive to avoid at all costs).

This article will discuss how to solve this problem using a simple Python script that leverages a central-

REQUIREMENTS

PYTHON: *2.4+*

Other Software:
- ldap python module *http://pypi.python.org/pypi/python-ldap/2.3.1*
- initScripts.py - *http://code.google.com/p/lipy/*
- openldap-clients - *http://www.openldap.org*
- web2ldap - *http://www.web2ldap.de/*

Useful/Related Links:
- LDAP Schemas *http://www.openldap.org/doc/admin24/schema.html*
- IANA *http://www.iana.org/*
- LDAP Private Enterprise OID Assignments *http://www.iana.org/assignments/enterprise-numbers*

ized LDAP directory, which holds the configuration of an entire environment's init script requirements. Once setup, you can use LDAP to adjust the start order of your applications, which machines they start on, and the runlevels they start in. Best of all, the entire admin team now only needs to know one generic Python init script to administer all of the applications and daemons running within the server environment.

The solution proposed comes in the form of the script **initScripts.py**, named after the operating system package that normally provides the init infrastructure. The script connects to a centralized LDAP server to pull the application-specific data about services to start. The application data includes how to start the application, stop the application, what user should run the application, as well as how to determine the state of the application. By installing initScripts.py, the administrator can eliminate the need to build custom init scripts for add-on applications, and configure the data centrally within an LDAP server instead.

Merits of a centrally managed init framework

Why would you want to set up all of your non-OS init scripts in a centrally managed solution? Consistency. As a system administrator, you should strive to make every aspect of your servers consistent. This leads to predictable situations and is key to increasing your ability to manage more servers per admin. By abstracting the add-on init scripts to a generic Python script, you are further achieving the goal of building cookie cutter servers.

One of the main goals of my previous employer was to implement consistent solutions. For this reason, nearly every machine purchased was a Sun Solaris machine. We also tried hard to stay consistent in the OS build, including any add-on software. One of the key pieces of software we loaded on all of our machines was Veritas Cluster Server (VCS). The typical setup was a simple two-node cluster, but the idea was that we would use VCS on every machine even where there was no high availability requirements. Eventually budgets became tighter and this opened the door for an alternative build that included Linux running on commodity x86 hardware. Included in the budget cuts was VCS, even though it is supported fully in Linux. Now, after using VCS for several years and deploying numerous applications on top of it, I knew that we would miss having VCS – but not because of its clustering abilities.

Besides being a great clustering software, a less well known quality of VCS is its ability to abstract a particular application setup. When you deploy an application under VCS control, you must follow VCS constructs and rules. The application is then controlled by a standard set of commands for querying its status, stopping and starting the app, and performing failovers to other machines. This means that I can develop, test, and deploy a complicated application setup, but all that my coworker needs to know to restart it all is the application name I have given the group of resources. In an environment where you have hundreds of servers, the fact that you can consistently use the same command to query and control all applications running on the particular hardware is fantastic. With that inspiration, I created initScripts.py to provide that same abstraction from our add-on software.

LDAP

LDAP is the *lightweight directory access protocol*. Although it is technically a communication protocol, the term is also used for data being served and the server product used. The server accepts connections over TCP/IP and clients can retrieve, add, and manipulate data being stored in the server. Using LDAP is widely supported from many languages and Python is no exception.

LDAP is popular because it has kept to the singular job of storing and serving data and therefore can excel in its one job. A tree hierarchy is used to organize the data, so clients can quickly retrieve select portions of the data without complex queries. Each branch of the tree is called an *organizational unit* (OU). By organizing your data in disjoint OUs, you can limit the amount of data the LDAP server has to search to retrieve the data you need. As a general rule, LDAP is really good at storing data but it is not a replacement for a RDBMS. To describe an LDAP server in terms of a RDBMS, each OU could be seen as a separate table with an index for fast data retrieval. However, unlike a RDBMS, you cannot form queries that join multiple sources of data into a single result set. There are many other examples of common functions that a typical RDBMS can perform for you and that your LDAP server will not do. LDAP stores non-relational data in the form of a hierchical tree of objects.

When your configuration requirements include having a centralized copy of the settings to be used by many clients, then using LDAP is a great choice. Typically people think of account information when LDAP is mentioned and in fact that is the most popular use of an LDAP server, both in the Unix/Linux world as well as for Windows with it's implementation of Active Directory (AD). Your application's config data can fit nicely with LDAP for the same reasons.

" As a system administrator, you should strive to make every aspect of your servers consistent."

Organizing Data in LDAP

When working with data in LDAP, you need to first describe the data being stored. The description is similar to how a DTD describes an XML document, in the sense that you are describing legal values and structure. You start by listing each of the various data elements that you plan to use. Each element is refered to as an *attributetype*, and you specify legal syntax, a short name, and a description for each. You can also define *objectclass* elements in the same file that are essentially made up of *attributetype* elements which you decide are valid ways of describing your objectclass elements. Collectively, the resulting document is referred to as a *schema*. Once you have produced the schema you can then configure your LDAP server to include it to add support for the various *objectclass* elements. This is a very broad description of the process and if you would like to learn more you should refer to the openldap documentation.

Before you decide to build your own custom schema, you should acquire a unique identifier for your objects. Each element within your schema is given a *unique object identifier* (OID). The OIDs are used in the same way OIDs are used in Simple Network Management Protocol (SNMP). You cannot have multiple objects with the same OID values. Should you try to reuse an existing OID value, your LDAP server will complain about the value already being associated with an existing object. Fortunately, you can easily acquire a unique OID from the IANA group. For initScripts.py, I used our company-issued OID. When and if you decide to use initScripts.py and need to import the schema, it is okay to use it without modifying the OIDs throughout the schema. The point is to ensure uniqueness and segregate custom values from other values that may exist in your LDAP server.

The initscripts Schema

For initScripts.py, I created a schema that defines an objectclass named "initScripts". As far as schemas are concerned, I do not believe it could be simpler. Each attributetype in the schema accepts any string value. The actual names of the attributetypes were inspired by VCS's generic application agent:

- Description: Brief description of application
- User: User to run commands under
- StartProgram: Command to execute when starting application
- StopProgram: Command to execute when stopping application
- CleanProgram: Command to execute when you absolutely need to stop application
- MonitorProgram: Command to execute that returns status of application
- PidFile: Full path to file containing the process ID
- Critical: Boolean indicating if this application is important enough to keep running
- scriptHost: Host to define this init script on
- RunLevel: System runlevel to live within
- OrderNumber: Denotes startup and shutdown order for the application

In order to use initScripts.py, you must edit your server configuration and include your new schema file. All of the other major LDAP servers support extending your schema. See documentation for your specific LDAP server to learn how to extend the schema to include the initScripts schema. In OpenLDAP, you just add a line that says `include /path/to/schema` to slapd.conf, OpenLDAP's default configuration file.

Populating an LDAP Database

Now that the LDAP server will accept an objectclass of `initScripts`, it is time to populate the tree with your data. The first step is creating a separate organizational unit in LDAP where we can keep our data segregated. I recommend creating the OU name `ou=initScripts` from the base of your tree. Within the OU, we create each init script entry and assign a *common name* (CN) value that serves as a short name for the data. So, when we are done we will have a separate OU named initScripts and it will be populated with the various CN entries for each of your 3rd party applications.

Populating the LDAP server is not as hard as it may seem. The most common method is to use the **openldap-clients** tools that include commandline utilities such as `ldapsearch`, `ldapmodify`, `ldapadd`, `ldapdelete`, and others. We can actually create the initScripts OU along with our first init script CN with one `ldapadd` call. Listing 1 illustrates the proper syntax to add data for a sample J2EE application named "BPN_QA". The CN name along with the OU location forms a unique `distinguished name` (DN). A valid DN is always required because it tells the LDAP server which object we are addressing.

To add this data into the LDAP server, save the example file as `example.ldif` and then use the command:

```
$ ldapadd -x -D "cn=Manager,dc=example,dc=com" \
    -W -f example.ldif
```

You will be prompted for the *Manager* password. See your LDAP administrator for a valid user name and password to use when authenticating to the LDAP server. For all subsequent applications, remove the first stanza from the input file since it create the initScripts OU, and that only needs to be done once.

The modify, add, and delete utilities operate on *LDAP Data Interchange Format* (LDIF) syntax. The easiest way to generate an example LDIF file is to use the `ldapsearch` command and include the `-L` option, that specifies to print the results in valid LDIF format. Continuing the example from above, you can dump the LDIF via the comand:

```
$ ldapsearch -L -x \
    -D cn=Manager,dc=my-domain,dc=com \
    -b ou=initScripts,dc=my-domain,dc=com \
    -W -h localhost cn=BPN_QA > bpn_qa.ldif
```

Since the command restricts the search criteria to just the CN of BPN_QA, that is all that written to the bpn_qa.ldif file. Next, simply edit `bpn_qa.ldif` to make any necessary changes and then apply your updates via:

```
$ ldapmodify -x \
    -D cn=Manager,dc=my-domain,dc=com \
    -W -h localhost -f bpn_qa.ldif
```

Being a system administrator, it is no surprise that my preference is to use the command line tools from the **openldap-clients** package. However, there are alternatives that will give you a GUI screen for doing your editing. One such utility is **web2ldap**, a Python-based LDAP client application that uses the same ldap module we are using in initScripts.py. Even if you too find that you prefer the command line tools, I'll admit that the web/GUI tools are nice for at least browsing the data in LDAP. Particularly the web based tools, such as **web2ldap**, since you can easily share a URL with other people within your organization.

Querying LDAP From Python

For initScripts.py, I am using the popular **ldap** python module developed by RedHat. You can do nearly anything with respect to LDAP using this module, but for our purposes we merely need to connect and perform searches for data. The module uses compiled components for very fast performance and is built on the **openldap** routines. Essentially, the ldap module is a one-to-one mapping of the C equivalents provided by the openldap folks. As such, you need to have the openldap development libraries in order to build the module; most Linux distributions include the package.

To get started, import the module and decide which server you can connect to:

```
import ldap
l = ldap.open(host)
```

LISTING 1

```
dn: ou=initScripts, dc=example,dc=com
ou: initScripts
objectclass: top
objectclass: organizationalUnit

dn: cn=BPN_QA, ou=initScripts, dc=example,dc=com
Description: Start J2EE application in QA
User: uqa4bpn1
StartProgram: /path/to/start/script
StopProgram: /path/to/stop/script
CleanProgram: /path/to/clean/script
MonitorProgram: /path/to/monitor/script
PidFile: /path/to/pidfile
Critical: 0
scriptHost: uqa4app[0-9]+
RunLevel: 3
OrderNumber: 99
objectClass: top
objectClass: initScripts
cn: BPN_QA

```

```
l.protocol_version = ldap.VERSION3
id = l.simple_bind(username,password)
```

When performing a search against the LDAP server, you can choose between synchronous and asynchronous search methods. In this example, we will perform an asynchronous search. Unlike a synchronous search, we are given a result id for the search, and then use that to retrieve the data.

```
baseDN = 'dc=example,dc=com'
result_id = l.search(baseDN, ldap.SCOPE_SUBTREE,
                     'cn=*')
result_type, result_data = l.result(result_id,
                                    all=1)
```

The search result are returned as a list of tuples including the DN of the object and a dictionary of the elements. As you can see in the following example, the dictionary data is the same as we've seen in the LDIF output. Each value within the dictionary is another list, even when the results are just one item. For sample searches, you can iterate through the results conveniently in Python like this:

```
for scriptentry in result_data:
    resultDN, ldif = scriptentry
    print '\nFound: %s' % resultDN
    cn = ldif['cn'].pop()
for key, values in ldif.iteritems():
        print '%s = %s' % (key,values)
```

How initScripts uses the data

Most of the initScript schema speaks for itself, while some of the values need further explaining, but let's first tackle the easy items. StartProgram and StopProgram are the full paths to programs that will start and stop the application. The Description is, as the name suggests, an area where you can a nice descriptive name to the application.

You have two choices for monitoring your application. The PidFile and MonitorProgram attributes are alternates for monitoring your application. The PidFile field should contain the name of a file that contains the running process ID of your application. InitScripts.py can use this and determine if that process is still running. As for MonitorProgram, it can be any executable. It will be run as the User specified in your config, and a return code of zero indicates that the application is still running. If you specify both, initScripts.py will only use PidFile values and ignore MonitorProgram but will still use MonitorProgram if the PidFile attribute has been deleted or cannot be read. The PidFile and MonitorProgram attributes are used whenever a 'status' is requested or on a 'condstart' where the application is only started if it is not already running.

" Unless you have cleaned up an ugly production environment, it is hard to appreciate the quality of conveniently listing out each application running on the system by a single command."

Not every application configuration will be used on every server. InitScripts.py uses the scriptHost value to determine if the particular setup should be used on the server where it is running. In the sample config, shown above in LDIF format, we used a scriptHost of uqa4app[0-9]+. The value is interpreted as a regular expression, so that means any server named like "uqa4appX", where the 'X' is any number of digits, will match the config and use it.

Since we can't rely too heavily upon both the network being available as well as your LDAP server being up, initScripts.py creates a local config with all of the matching values from LDAP. The config is written to the same working directory as the initScripts.py script itself and is named initScripts.cfg. Updating of the local cache is attempted each time initScripts.py runs in an effort to stay as up to date as possible.

You can control which runlevel and the order in which your applications are started by setting the values of `RunLevel` and `OrderNumber`. Your *nix and Linux systems have several runlevels during which different applications are started. A notable example is how certain distributions of Linux define a runlevel of 3 for a full multiuser environment and a runlevel of 5 for the exact same thing plus the graphical environment. Your system should be configured so that initScripts.py is run in each runlevel on your system, but only acts upon your application when the current runlevel on the machine matches the runLevel config value you've set in LDAP. Now, within the runlevel you can also control which applications start first by naming each script with a numerical convention that implements your start order. Since initScripts.py is a single script, it will only be executed once within the system's runlevel, but it needs to manage the order of multiple applications. Since the script is intended to manage non-OS services, the assumption is that it is safe to start your application last within the system's runlevel. Then, initScripts.py uses it's own `OrderNumber` values within it's configuration settings to control the applications started and their order. It is almost like a micro init structure within the system's own runlevel. I actually uses a very thin shell wrapper to call the Python version of the script during boot time since the OS executes each script in the Bourne shell.

Not every config value is used, however. The last config value to discuss, `Critical`, is not currently used. I wanted to allow for the possibility of the script becoming more than an init script replacement and become a monitoring component as well and the value of `Critical` would be used in that capacity, just like it is used in VCS. A value other than zero for `Critical` would indicate that the application should be automatically restarted upon determining that it is not running.

Although not an explicit config value in LDAP, knowing which LDAP server to connect to is an important configuration value. initScripts.py assumes that the machine is using LDAP for other system data and therefore already has a valid `/etc/ldap.conf` file. initScripts.py opens up the ldap config file and looks for either a line starting with `host` or `uri`. For normal LDAP configurations, those are your two choices in specifying your LDAP servers. InitScripts.py understands both syntaxes and tallies each of the LDAP servers into a list. Next, it iterates through the list of LDAP servers until it finds one it can establish a connection to. If you do not use LDAP for any other services and therefore have no ldap.conf file, then you can create a simple one-liner config for the sole purpose of the initScripts.py. Simply place the keyword "host" followed by all of your LDAP servers, space separated. A dirt simple ldap.conf config file could look like:

```
host  ldap1.example.com ldap2.example.com
```

Other Applications for initScripts

Beyond having the consistency and convenience of a single script for starting all applications, there are now additional benefits of keeping all of your init data centrally managed. For example, suppose you are planning to perform maintenance on a server and have to do numerous reboots of the server. You can now disable the applications from starting by updating your configuration in LDAP and re-enable them once you are done. The power of this is magnified when your maintenance work is for a very large number of servers.

initScripts.py can also act as a documentation tool. When logging into an unfamiliar server, instead of poking around the system for applications, with initScripts.py you can ask for a status via `initScripts.py status`. Unless you have cleaned up an ugly production environment, it is hard to appreciate the quality of conveniently listing out each application running on the system by a single command. If you have worked in a messy server environment, then you know it is too common for system administrators to waste time reverse engineering an application setup.

It is also easy to re-sequence the start order of applications and to add another copy of the same software on your server because your machine is underutilized. Re-ordering which application starts first, second, or last is quite convenient from LDAP.

Some operating systems provide mechanisms for setting dependencies within your init scripts, but not all. Traditionally, each init script is sequenced by a numeric value prefixed to the script name along with the letter 'S'. So, a script named `S01apache` would start before `S02tomcat`. initScripts.py uses the same convention, but the value is set within the initScripts object in LDAP. In the example of re-sequencing apache and tomcat, I can update the OrderNumber for apache to 03 to have tomcat started first.

As we have mentioned already, initScripts.py supports regular expressions when targeting which servers your application should be started on. Say you have a set of servers named `prod4app1`, `prod4app2`, and so on that all work together to provide a common service. In such a setup, you are managing increased demand by growing horizontally, by adding more servers. When configuring the entry in LDAP for this application, you should set `scriptHost` to `prod4app[0-9]+`. That way, when you add the next server (say `prod4app3`) you do not need to perform any setup to allow the application to automatically start on that server. You essentially have one master init script configuration for all of the servers at once. So again, making changes to what needs to be started and in what order is very convenient and does not require logging into each of your servers to make changes.

" I am a large proponent of solving your own IT problems yourself without outside software."

By having a single script for your add-on applications, it is easy to grant superuser rights to administer the applications. With data center facilities people, they have too much to handle to ever become experts on any one platform. So, when you ask them to handle the stopping and starting duties of the applications, you can grant that permission very easily by allowing them to run initScripts.py. Because it is the same simple script across all of your servers, the data center facilities people can easily learn that they only need to use initScripts.py for normal application administration. This also means that once you add another application to the same machine, they immediately have the necessary rights to aid with the administration of that application.

Summary

Hopefully you have learned how you can build custom data stores in LDAP and use it from your Python scripts. The common goal whenever you centralize data and move it away from the particular server is to avoid manually logging into each machine to make alterations. Once you have that data abstracted away, it seems like other possibilities for refinement seem to be more obvious than before.

In the case of initScripts.py, the goal was creating consistency. In general, consistency breeds predictability and with initScripts.py it is taken to the extreme considering you now have one script for all of your add-on applications. In my situation, we were deploying numerous J2EE based applications that had good start and stop scripts and created pidfiles, but they didn't have their own init script. When adding multiple copies of the same software in different directories and then scaling to other machines, not having a single cohesive script such as initScripts.py could have easily led to disarray in our environment.

The script had a modest beginning. I didn't want to get carried away with developing features until we started using it and recognized which features we wanted to implement. I am a large proponent of solving your own IT problems yourself without outside software. In my experience, the amount of effort expelled with a new product equals that of building your own application from scratch in the long run. So, if you administer a server environment that could use some cleanup with respect to it's init scripts, then perhaps initScripts.py is a good start toward cleaning up that environment. But do not be satisfied with the script as it is today. You are encouraged to modify it to suit your environment's specific needs. Enjoy.

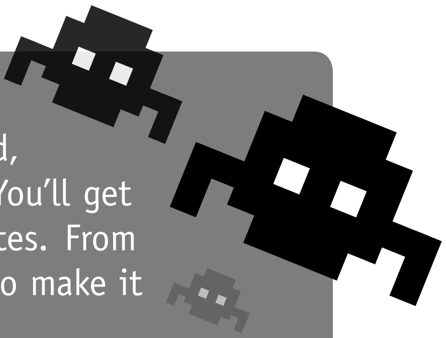
JON MILLER is a system administrator by trade with a programming background. As such, he is not intimidated to create programs to solve his sysadmin problems. You can contact him at *jonEbird@gmail.com*.

Learning Python with PyGame

The Simplest Thing that Can Possibly Work

by Terry Hancock

When I first learned to program, it was pretty pictures and games that motivated me to press on. Bearing that in mind, this is a tutorial for the impatient (tested on preteens!). You'll get something fun working on the screen in just a few minutes. From there, you'll refine and learn some programming concepts to make it better.



The goal of this tutorial is to teach Python with visually-interesting examples. You will also get a quick start with **PyGame**, though this isn't really a PyGame tutorial. In this lesson, we'll be starting with a very plain *script* that opens a graphical window, draws an image, and moves it around randomly. The point here is to "keep it simple". Later lessons will expand on that, to create a full-fledged game.

What You Need to Get Started

You will, of course, need Python and PyGame. These are included in major GNU/Linux distributions, so you will simply need to use your package installation system to install them. Users of Windows and other operating systems can find installable binaries (and source) at *http://www.python.org/download/* and *http://www.pygame.org/download.shtml*. PyGame is based on the Simple Di-

REQUIREMENTS

PYTHON: *2.3+*

Other Software:
- PyGame 1.6+ - *http://www.pygame.org*

Useful/Related Links:
- PyGame Documentation *http://www.pygame.org/docs/*
- Python Documentation - *http://docs.python.org*
- Python Library Reference - *http://docs.python.org/lib/lib.html*
- Simple Directmedia Layer *http://www.libsdl.org*

rectmedia Layer (SDL) library which supports just about any platform you are likely to be using. The SDL library will typically be installed by the PyGame package or handled by your package manager, so you won't have to worry about it.

You will also need a text editor, a terminal emulator, and a web browser that you are comfortable with. I use Vim, but Emacs, or even Windows Notepad will do. Really nice editors will have an option to automatically colorize your Python source code to make it easier to read and find mistakes, but you can get by with plain text. Just make sure you aren't using a word processor that will stick unwanted formatting codes into your file.

We'll use the terminal emulator to run the Python *interactive interpreter*, which is a great way to test code as you go. You can use the web browser to keep up with the documentation for Python and PyGame in order to understand the libraries we are using.

Your First Program

Enough chit-chat. Let's start Python (you almost certainly do this by entering the command `python` in the terminal, though some systems will let you start it with a mouse click). Once the interactive interpreter is started, you will see the python prompt: >>>, where you can type in Python commands to be run immediately. First, you should check that PyGame is installed:

```
>>> import pygame
>>>
```

The `import` statement is used to access library *modules*. It is this command that allows us to access PyGame functions later on. If no error message appears, then you're good to go (if you encounter an error, you need to go back and check your PyGame installation).

You should then import `random` in the same way (this is one of Python's many standard libraries, so you don't need to install it – look up the `random` module in the Python Library Reference for more information on what it does):

```
>>> import random
>>>
```

We need two more lines to get PyGame ready to use:

```
>>> from pygame.locals import *
>>> pygame.init()
>>>
```

This import statement is a little different. Instead of loading the module as a unit, it loads all of the contents of the module into the *namespace* of our program, which means we can call them without having to prepend "`pygame.`" onto everything. Normally, this is undesireable as it clutters up your program, but the `locals` module from PyGame contains constants that are convenient to use directly (which we'll need later on). The next line calls a function `init()` that PyGame requires us to call before it can work.

With PyGame started, we can now call up a PyGame window from the interpreter:

```
>>> screen = pygame.display.set_mode((800,600))
```

The `screen` is now a python *object* which we'll use to refer to the PyGame window (and get some properties from it). At the same time, of course, you should have seen an empty window pop up. What we're doing here is *calling* a *function* `set_mode()` in the module `display` which is contained in the `pygame` module. A function is just a reusable piece of code, in this case provided by the pygame module. See Figure 1 for more about how functions work.

We've called it with a single argument, `(800,600)`, to tell it how large a window to make. This is an example of a Python *tuple*, which is just a group of objects separated by commas, usually within parentheses. We'll typically use tuples of two numbers to represent coordinates.

That's why we have double parentheses: the inner set defines the tuple, the outer one is the function call. This is different from calling a function with two arguments (in which case, there'd be only one set of parentheses).

Next, we'll load a graphic *sprite* to animate on the screen. A sprite is just a small picture used in games. We want the graphic to be transparent, so we also set a *colorkey*, which is to say, a color that will be treated as transparent (the color is represented by a red-green-blue tuple, which in this case is just white).

```
>>> sprite = pygame.image.load('resource/mouse.bmp')
>>> sprite = sprite.convert()
>>> sprite.set_colorkey((255,255,255))
```

Afterwards, we'll get the width and height of both the screen and the sprite we loaded. The method, `get_size()`, returns another tuple of two numbers which we can directly assign to two numbers (this technique, which is common in Python, is called *tuple unpacking*). A *method* is a special kind of function that is attached to an object.

We'll need these numbers to figure out the offsets from the centers to the upper left corner of each object, which is the origin of the coordinate grid.

```
>>> spritew, spriteh = sprite.get_size()
>>> screenw, screenh = screen.get_size()
```

After that, we just figure out where the upper left corner of the sprite needs to be if we want it to appear in the center of the window. See Figure 2 for an explanation of the geometry we are using here. Whenever we assign locations to place *Surfaces* (images stored in memory) in PyGame, we're always specifying the upper left corner.

```
>>> x = screenw/2 - spritew/2
>>> y = screenh/2 - spriteh/2
```

Next we're going to make a *list* of four tuples, each representing a direction (up or "N", right or "E", left or "W", and down or "S"), in which our sprite can move:

```
>>> N = ( 0, -1)
>>> S = ( 0,  1)
>>> W = (-1,  0)
>>> E = ( 1,  0)
>>> directions = [N, E, S, W]
```

The directional moves may seem a little funny to you if you are used to regular graph coordinates as used in math class. That's because in computer graphics, the origin is in the upper left corner of the screen or window. Positive *x* values increase to the right, and positive *y* values increase towards the bottom of the screen. So, going up ("N"), means *decreasing* the value of the y coordinate. Each direction tuple is given a *name* via the = assignment operator so we can refer back to them later.

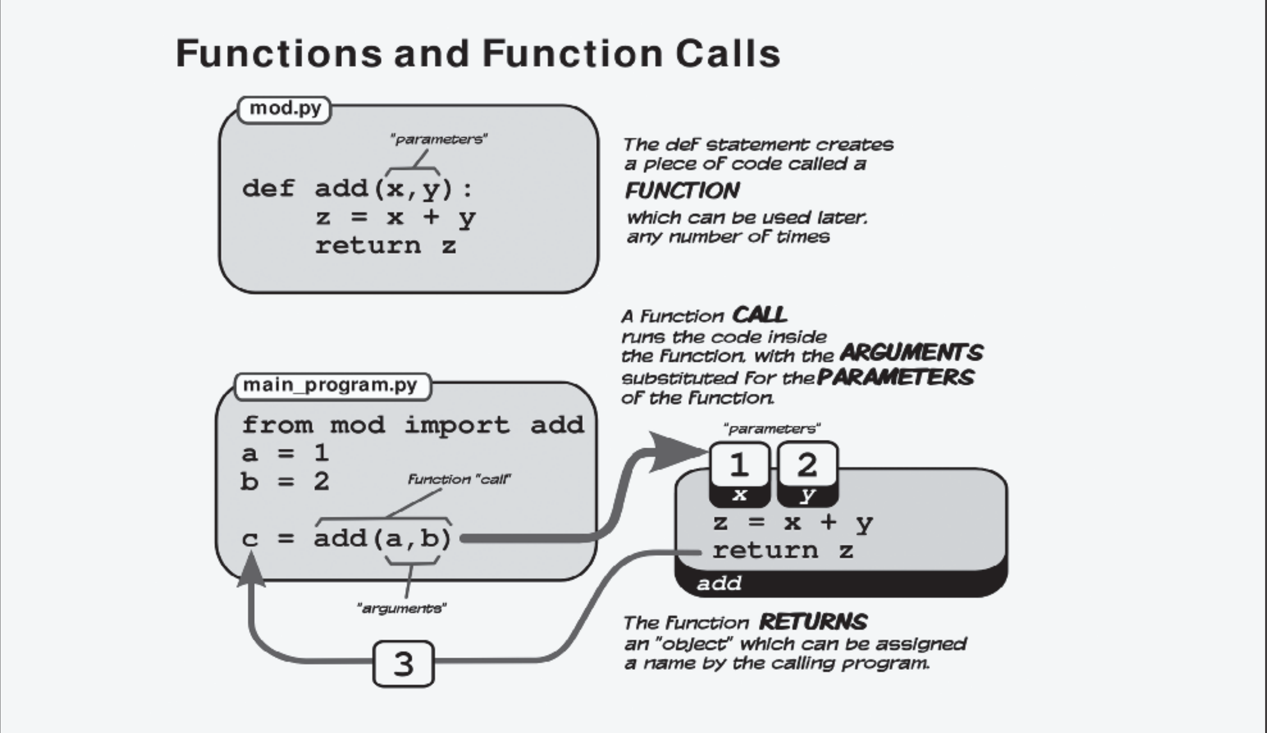
The last line above gathers these tuples up into a Python list, which is just an ordered collection of objects and is represented by square brackets `[]`.

Now we'll start defining the *game loop*. First of all, we use a common Python idiom for an infinite loop: `while True:`. Normally, the `while` statement will loop until its expression is no longer true. In this case, we want it to loop forever (or until we explicitly break out of it), so we just set the expression to `True`.

Later on, we'll provide better ways to stop the program, but in the meantime, you can use `Ctrl-C` to interrupt it. Be aware that this will only work when the terminal window has focus (is selected with the mouse). If you use `Ctrl-C` on the PyGame graphical window, nothing will happen.

Inside the loop, we need to do two basic things: first we have to compute the new location for the sprite,

FIGURE 1


Functions and Function Calls
mod.py
"parameters"
The def statement creates a piece of code called a FUNCTION which can be used later, any number of times
A Function CALL runs the code inside the Function with the ARGUMENTS substituted for the PARAMETERS of the Function.
main_program.py
Function "call"
"parameters"
1
2
x
y
add
"arguments"
3
The Function RETURNS an "object" which can be assigned a name by the calling program.

```
def add(x,y):
    z = x + y
    return z
```

```
from mod import add
a = 1
b = 2

c = add(a,b)
```

```
z = x + y
return z
```

and second we have to call PyGame functions to put the sprite in that place and update the display. For this very simple example, we're just going to use the function `random.choice()`, which picks a random object out of a Python list, to pick a random direction each time we pass through the loop. Then we'll update the `(x,y)` tuple with the new coordinates by moving one step in the selected direction.

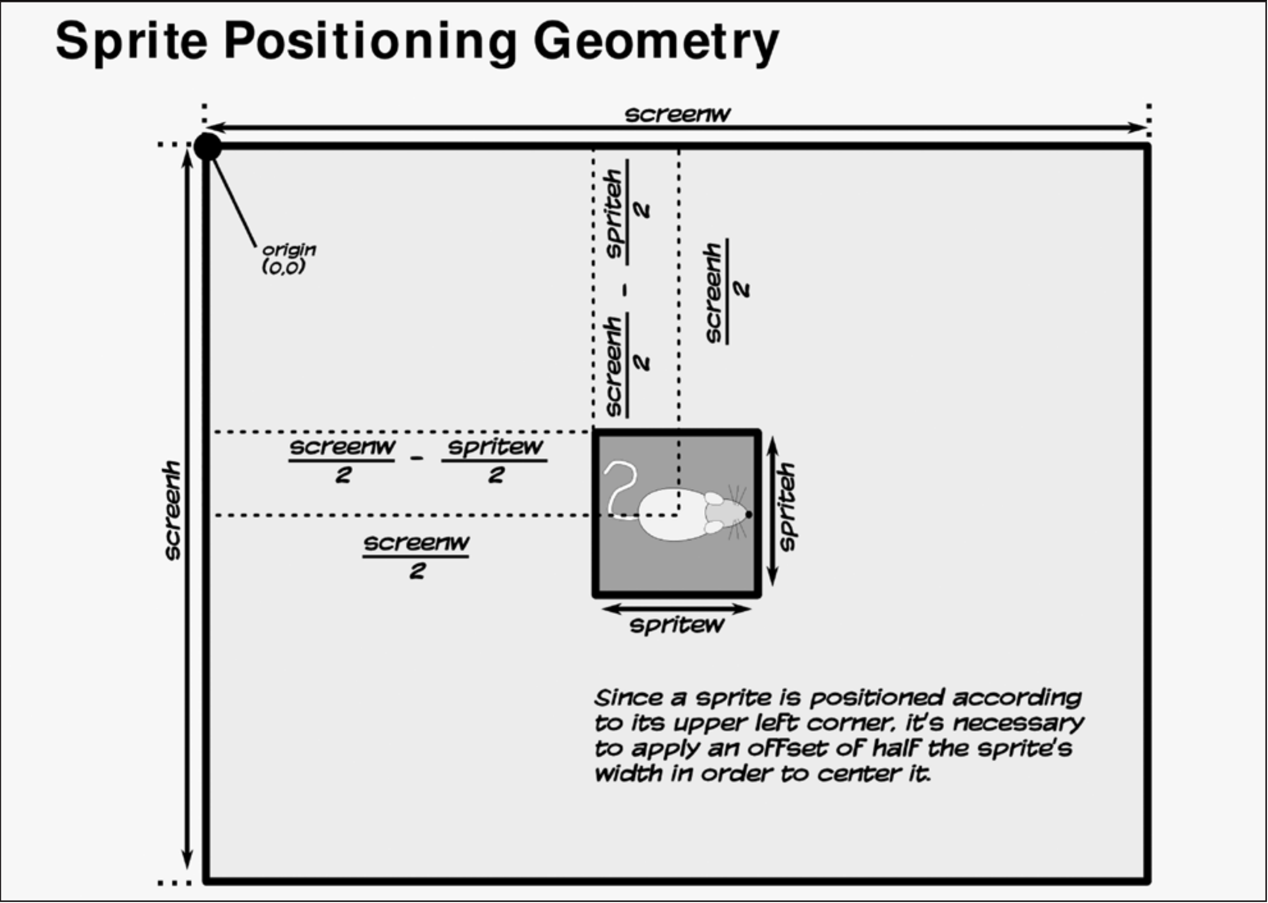
We're going to use a very, very simple update method in this example, as well. We'll just repaint the window green, which will erase the old sprite position, no matter where it was. Then we'll place the sprite onto the screen surface in what is called a *blit* in computer graphics jargon ("blit" is an acronym derived from "BLock Image Transfer"). None of this actually draws anything on the screen, though. That happens when we finally call PyGame's `update()` function. The reason PyGame works this way is so that you can make a whole lot of changes without disturbing the display, and then flip the result onto the screen quickly, resulting in smoother animation.

Here's the final game loop. Note that the stuff inside the loop has to be indented to let Python know it's part of the same loop. Be careful to use consistent indentation! One of the more common errors for new Python users is to indent inconsistently, often because of mixing spaces and tabs (when in doubt, just avoid using tabs at all).

```
>>> while True:
...     dx,dy = random.choice(directions)
...     x = x + dx
...     y = y + dy
...     screen.fill((75,255,55))
...     screen.blit(sprite, (x,y))
...     pygame.display.update()
...
>>>
```

Note that you'll have to press the Return key a couple of times to finish the loop and get back to the regular prompt. Once you do, though, your program will start to animate in the PyGame window. Congratulations! You now have a working PyGame program running. It should look something like Figure 3.

FIGURE 2


Sprite Positioning Geometry
screenw
origin (0,0)
screenh/2 - spriteh/2
screenh/2
screenw/2 - spritew/2
screenw/2
screenh
spriteh
spritew
Since a sprite is positioned according to its upper left corner, it's necessary to apply an offset of half the sprite's width in order to center it.

Save your work!

Naturally, you'll want to paste all of this code into your chosen text editor and save it. A copy of this example is included in the source code package for this tutorial as mouse0.py. You'll notice the file also has *comments* added, using the # sign. Those lines will be ignored by Python and are just included for your benefit.

So far, we've just been typing into the interpreter application. But from now on, we'll make modifications and then save the program and run it from the terminal emulator. Assuming you named your file mouse0.py as I did, you can run the program like this:

```
# python mouse0.py
```

Smoothing the motion

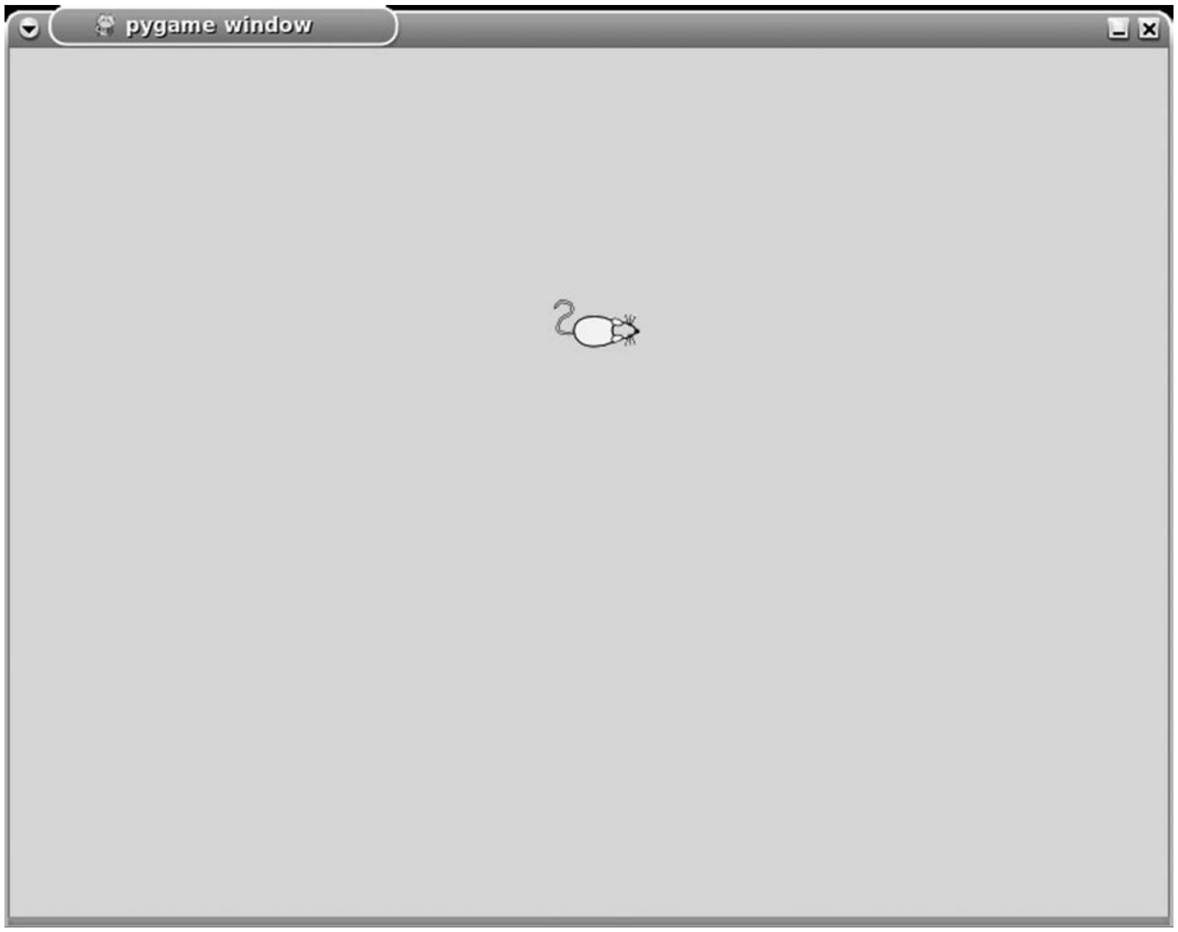
Now it's not a very smart program yet. The mouse image probably just seems to be randomly shaking around on screen. That's because we're not letting it move smoothly in any direction for more than one pixel. Instead, we change direction on every loop iteration.

Our first update to the program, therefore, will be to add a counter to wait a hundred steps before updating the direction. We'll use a *conditional expression* (Python's "if-elif-else" statement) to figure out when the counter has counted high enough, and then put our random direction selection in that part of the code. That way, on most passes the (x,y) position will be updated, but the direction will stay the same. This will produce a much smoother motion.

Here's what the new loop will look like:

```
i = 100
while True:
   i += 1
   if i > 100:
       dx,dy = random.choice(directions)
       i = 0
   x += dx
```

FIGURE 3



```
        y += dy
        screen.fill((255,255,255))
        screen.blit(sprite, (x,y))
        pygame.display.update()
```

The variable i is the *counter*. We use it to count to a hundred (once per loop iteration), and then choose a new direction. We set it to go off immediately so that the loop will pick a direction at the beginning (otherwise, we'd get an error message telling us that dx and dy aren't defined!). I've added one more change as well: Python allows us to use a special *increment* operator += when we want to assign a new value to a variable by adding something to the old value. We use it here to increment i, and we also replace `x = x + dx` with `x += dx`, which means the same thing and avoids repetition.

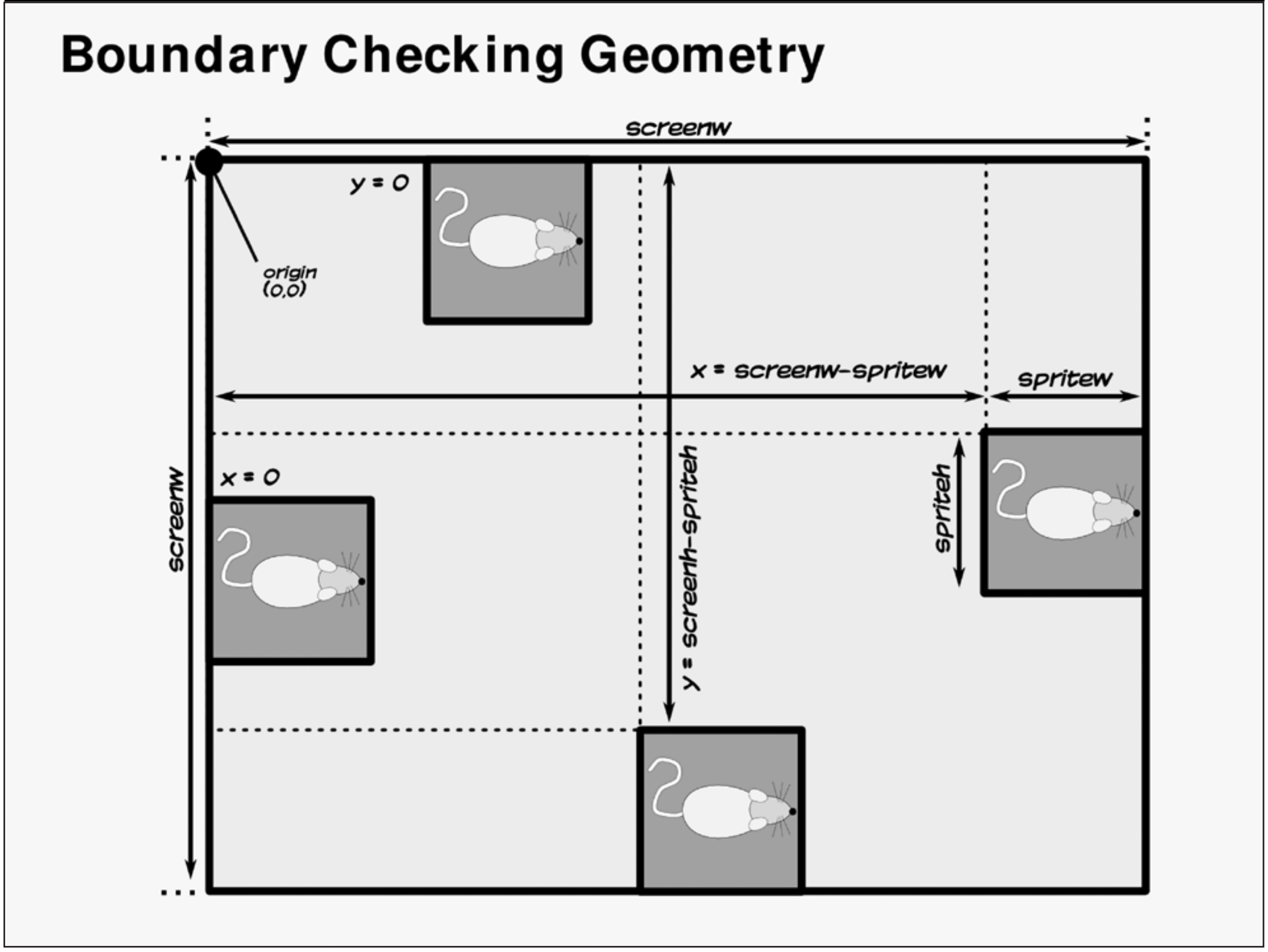
If you run this program (`mouse1.py` in the source package), you'll see a more satisfying display: the mouse now moves more purposefully across the screen, changing directions only every 100 pixels. The speed depends on your CPU and your graphics card. It may be quite slow or insanely fast, depending on your hardware.

Staying in bounds

If you let the example program run for just a bit, you'll see something definitely wrong: the mouse eventually drifts out of the window, never to be seen again! Fortunately, we can use another conditional statement block to fix this problem. We'll check for x getting too small (less than 0, which means off the left side of the window), or too big (closer than the width of the sprite to the right side of the window, which means subtracting the sprite width from the screen width). And of course, we'll do the same tests to y. See Figure 4 for a geometric explanation of these limits.

When triggered, we'll use `random.choice()` again, but this time, just pick from directions leading away from the wall. To make things a little more interesting,

FIGURE 4



" We'll use the terminal emulator to run the Python interactive interpreter, which is a great way to test code as you go"

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))
sprite = pygame.image.load('resource/mouse.bmp').convert()
sprite.set_colorkey((255,255,255))

# Get the sizes of the screen and sprite "surfaces"
spritew, spriteh = sprite.get_size()
screenw, screenh = screen.get_size()

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

# Define moves:
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]

# Infinite "game loop"
i = 100

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

    # Turn at boundaries
    if  x < 0:
        dx,dy  = random.choice([E, NE, SE])
    elif  x > screenw-spritew:
        dx,dy = random.choice([W, NW, SW])
    elif  y < 0:
        dx,dy = random.choice([S, SE, SW])
    elif  y > screenh-spriteh:
        dx,dy = random.choice([N, NE, NW])

    # Update the position
    x += dx
    y += dy

    # 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)


```

I've also added the four diagonal directions to this version, so we now have eight total directions to choose from.

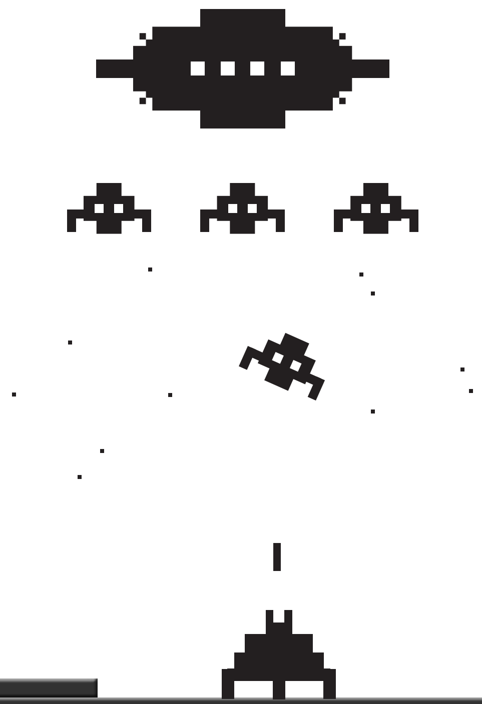
Finally, we can insert a timer call into the loop in order to reduce the speed to a reasonable rate on fast hardware. This will simply wait for five milliseconds:

```
pygame.time.wait(5)
```

This leads to our first simple PyGame program, seen in Listing 1 (also included in the source package as `mouse2.py`).

To be continued...

Next month we'll build on this simple example, making it into a real program, and adding a number of features, including interactive keyboard control and more convincing animation. Along the way, we'll learn how to use *dictionaries*, *functions*, and *classes*.



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.

Writing a Simple Interpreter/ Compiler with Pyparsing

by Paul McGuire

Did you ever want to create your own programming language? Maybe not one with the power and richness of C++ or Java, but perhaps just a simple command language to control a software application, or create graphs or forms, or interact with a web page. The trick is, once you have designed the commands, keywords, and syntax of your new language, the next step is to write the parser that will convert your language statements into executable code. This article will walk through the basic steps to create a parser and interpreter for your very own programming language.

Some are born to write parsers. Some become parser writers. And some have parser writing thrust upon them.

Parsing is one of the programming tasks that crops up again and again for software developers. As humans, our brains parse written symbols such as letters, words, and numbers into information every day. We even parse a variety of text forms, such as

```
Hello, World!
1-800-CALL-NOW
y = mx + b
lol u r my bff
```

and can make some reasonable guesses as to what they might mean. But writing software to enable a computer program to parse these same expressions can be a complicated task.

Parsing With Regular Expressions

One of the common programming tools used to tackle these problems is a *regular expression* (or RE). A regular expression acts as a pattern to test for matching

REQUIREMENTS

PYTHON: 2.3.1+, pyparsing 1.4.10

Useful/Related Links:

- pyparsing - *http://pyparsing.wikispaces.com/*

against a candidate text string. Special characters are used in defining that pattern, representing repetition, grouping, words, whitespace, and punctuation. For example, here is a regular expression for reading equations like "y=mx+b":

```
([a-z])=([a-z]+)([-+*/])([a-z]+)
```

To be able to interpret regular expressions, you have to be familiar with the specialized characters and codes used. In fact, an entire cottage industry of books and websites exists to help develop and debug regular expression strings. This example expression uses some common codes:

- Parentheses (`()`) are used to group characters into selectable fields.
- Square brackets (`[]`) are used to enclose a set of valid characters; character ranges can be abbreviated, for example `[abcdefABCDEF]` can be abbreviated to `[a-fA-F]`.
- Plus (+) means 1 or more (except when part of a `[]` set).
- Asterisk (*) means 0 or more (except when part of a `[]` set).

So this regular expression translates to "a single alphabetic character, followed by an equal sign (=), followed by one or more alphabetic characters, followed by one of the characters -, +, * or /, followed by one or more alphabetic characters."

The regular expression library can process the input text and return parsed bits of the input as marked by the grouping parentheses in the expression. A program using regular expressions will retrieve these parsed fields, and then interpret them in the order they are defined in the regular expression.

But as powerful as regular expressions are, they quickly become unwieldy when tackling more complex parsing applications. For instance, parsing often involves unpredictable whitespace, and regular expressions are very literal in their processing of text. In our previous example, if space characters are used to set off the equal sign and plus operator, as in `y = mx + b`, the regular expression must be expanded to allow for 0 or more intervening space characters (the code for a whitespace character is `\s`):

```
([a-z])\s*=\s*([a-z]+)\s*([-+*/])\s*([a-z]+)
```

The resulting description strings can be difficult to read or extend. One of the reasons for this is that regular expressions use the same characters to indicate repetition and grouping as may be found in the matched text itself. In the example above, * means "repeat 0 or more times," and + means "repeat 1 or more times." However, within brackets, these symbols just mean the characters * and +. If the expression is written using vertical bars (|) to indicate alternation instead of a character range, backslashes must be added to escape the repetition characters:

```
([a-z])\s*=\s*([a-z]+)\s*(\-|\+|\*|\/)\s*([a-z]+)
```

Before long, it is hard to see which characters in this string represent actual parsed characters, and which are overhead to indicate whitespace, repetition, or backslashes to escape RE-significant control characters. In the extreme, programs that use these expressions come to be regarded as unmaintainable, or "minefield code" (one of my pyparsing correspondents referred to such scripts as "code that scars all who touch it"). It is difficult to update this code when the application or data formats evolve, and attempts to actually make changes or bug-fixes are just as likely to introduce new errors as to fix the old ones.

Parser Generators

Another key deficiency of using regular expressions as parsers is that they cannot process recursive patterns like nested parentheses, such as you would find in an arithmetic expression:

```
x1 = (-b + sqrt(b**2-4*a*c))/(2*a)
```

To process recursive patterns, and to handle more complex text parsing tasks, we have to turn to utilities for creating parsers or parser code generators. The traditional choice for creating parsers is to use the pair of utilities **lex** and **yacc**, or the GNU versions **flex** and **bison**. To work with these tools, you must first define (using regular expressions) the formats of the lowest-level tokens in the grammar. lex uses these definitions to create C code that can be compiled to perform the scanning function, called *lexing* or *tokenizing*. yacc processes a second file defining the Backus-Naur Form (BNF) syntax for combining tokens into language expressions, statements, or commands, and generates C code based on this syntax definition (or grammar) to parse the sequence of tokens generated by the lexing code. The standard Python distribution does not include lex and yacc modules, but externally available modules **PLY**, **spark**, and **simpleparse** provide similar features.

I've never gotten past my resistance to the lex/yacc parser development model – on the pyparsing wiki I describe myself as "lex/yacc-challenged." I find that I must shift mental gears between lex's token definitions, yacc's grammar definition and implicit "yy" variables, and tying the parsed tokens back into the original calling program, whether it is C, Java, or Python. Writing parsers is already a mind-bending exercise as it is, without throwing in a mix of different coding syntaxes and languages. Some of the Python-based modules make use of Python's introspection features, such as representing grammar definitions in docstrings, or in auto-attaching productions to functions by imposing a "do_<production_name>" naming requirement. Using docstrings to store embedded BNF syntax simply provides a back door for injecting non-Python code, again creating a "need 2 languages to solve 1 problem" situation. And I've always bristled at API's that work only if I name my functions according to a particular naming template.

I felt that Python already offered enough power in its object definition and dynamic attribute model, plus its built-in support for operator overloading, and its native dictionary data type. Given these features, a parsing module should be able to be 100% Python in not only implementation, but also in how it is used from client code. An explicit object-based parser should be able to represent the basic grammar definition, any callback methods to be invoked during parsing, and the resulting parsed data. A module following these guidelines should result in parsers that are straightforward to write, clear to read, and easy to maintain and enhance.

Benefits of Using Pyparsing

I wrote **pyparsing** to simplify writing parser applications in Python. Here is a parser for the equation "y=mx+b" using pyparsing:

```
eqn = Word(alphas, exact=1) + '=' + \
      Word(alphas) + oneOf('+ - * /') + \
      Word(alphas)
```

That is, an equation is "a word composed of alphabetic characters that is exactly 1 character long, followed by =, followed by a word of 1 or more alphabetic characters, followed by one of the characters +, –, *, or /, followed by another word of 1 or more alphabetic characters."

While this may be more verbose than the regular expression above, it is fairly obvious where to make changes if you want to change the assignment operator from = to `:=`, or to add `%` as another arithmetic operation. In addition, when pyparsing uses this expression to parse an equation, it will automatically step over any intervening whitespace, without having to insert special "whitespace can go here" markers in the grammar.

Some grammars do use whitespace to signify meaningful structure, such as indentation or end-of-line. In those cases, pyparsing allows the user to modify the default whitespace character set of " \t\n", and whitespace skipping can be enabled or disabled globally or for individual parse expressions.

Also, note that the pyparsing expression is written directly into executable Python code. It is not necessary to learn non-Python code formats for regular expressions, token definitions, or grammar specifications. There is no intermediate code-generation step to distance the grammar definition from the actual running code. We can directly run this expression using the Python statement:

```
print eqn.parseString("y=mx+b")
```

and get the parsed results:

```
['y', '=', 'mx', '+', 'b']
```

Pyparsing returns the parsed tokens using a datatype called `ParseResults`. ParseResults can be built up in a hierarchical structure similar to an abstract syntax tree (or AST), and can have named data fields, making it easy to retrieve the information from the parsed tokens. If data fields are given names, they can be accessed as if they were entries in a Python dict, or attributes of a Python object. For instance, if I named the left-side variable of "y=mx+b" as "assignTo", using this modified definition:

```
eqn = Word(alphas, exact=1)("assignTo") + '=' + \
      Word(alphas)("term1") + \
      oneOf('+ - * /')("operator") + \
      Word(alphas)("term2")
```

then I could access it or any other named results field by name as:

```
results = eqn.parseString("y = mx+b")
print results["assignTo"]
print results.assignTo
print results.operator
```

The ability to access specific tokens by name becomes much more important when working with complex grammars with optional data fields or subexpressions.

Finally, pyparsing supports the attachment of user-defined callbacks to elements within a grammar, to be called during parsing when that element is matched. Pyparsing refers to these callbacks as *parse actions*, and they are a powerful tool for providing additional behavior during the parsing process, including semantic validation, update of global data structures, or even modification of the matched tokens.

Pyparsing Examples

By way of demonstration, here are some simple examples using pyparsing, with the regular expression and pyparsing form for each:

IPv4 address

(Internet protocol addresses such as “192.168.0.1” or “127.0.0.1”)

Regular expression:

```
\d{1,3}(\.\d{1,3}){3}
```

pyparsing:

```
Word(nums, max=3) + ("." + Word(nums, max=3))*3
```

This example uses the regular expression syntax for repetition, “`{count}`” or “`{min,max}`”. “\d” is another special character sequence representing any numeric digit; this is equivalent to the regular expression character set form “`[0-9]`”. It is necessary to use the backslash to “escape” the **.** delimiter in the IP address, since a **.** normally means “match any single character”. With pyparsing, the individual fields in the address are defined as “a word made up of numeric characters, up to a maximum of 3 characters long.” The repetition of the 3 “period + numeric field” subexpressions is indicated using the normal multiplication operator. If a min-max range is required, you can multiply an expression times a `(min,max)` tuple value.

ZIP+4 postal Code

Regular expression:

```
\d{5}(?:\-\d{4})?
```

pyparsing:

```
Word(nums, exact=5) + \
    Optional("-" + Word(nums, exact=4))
```

This example parses the US ZIP+4 postal code. The original ZIP code introduced in 1963 was 5 digits long, but in 1983, an optional 4-digit field was added. Even now, address parsing programs must recognize 5-digit or 5+4 digit ZIP codes. Similar to the IPv4 example, the pyparsing expression for matching a ZIP+4 code uses the pyparsing `Word` class to define groups of numeric “words”, beginning with a 5-digit group, optionally followed by a dash and a 4-digit group.

List of colors

Regular expression:

```
(red|blue|green)(,\s*(red|blue|green))*
```

pyparsing:

```
delimitedList( oneOf("red blue green") )
```

In this example, we see two common pyparsing helper methods, `delimitedList()` and `oneOf()`. `oneOf()` provides a quick short-cut for specifying a list of alternate literal strings. This short-cut replaces the more tedious

```
Literal("red") | Literal("blue") | Literal("green")
```

Lists of items separated by commas is a very common parsing expression, so pyparsing includes the `delimitedList()` helper. `delimitedList(expr,delim)` is a short-cut for

```
expr + ZeroOrMore(Suppress(delim) + expr)
```

The default value for delim is a `Literal(',')`, but any pyparsing expression can be used. Note that the delimiters themselves are suppressed from the returned tokens, since their purpose is merely for separation of values in the list, and do not have any token-worthy content themselves. To return these tokens as a nested list within the overall set of parsed data, wrap this delimited list in a pyparsing `Group`

```
Group(delimitedList(oneOf("red blue green")))
```

Date and time

Regular expression:

```
\d{4}/\d{2}/\d{2} \d{2}:\d{2}:\d{2}
```

pyparsing:

```
date_expr = Combine(Word(nums,exact=4) +
                    ("/"+Word(nums,exact=2))*2)
time_expr = Combine(Word(nums,exact=2) +
                    (":"+Word(nums,exact=2))*2)
date_time = date_expr + time_expr
```

This example starts to show how simple expressions can be built up into larger, more complex expressions, using operators such as + and |. It also shows the use of the `Combine()` expression, which suppresses whitespace skipping for enclosed expressions. This means that `2008/01/01` would be accepted as a valid date, but `2008 / 01 / 01` would not.

Combining expressions

Using the previous expressions as basic elements, we can combine them to parse more complex input text. For instance, if we needed to parse the contents of a log file, in which each log entry consists of a date-time timestamp, an IP address, a list of colors, and an optional quoted string for a description, we could combine the previous expressions into one overall expression representing a complete log entry.

```
log_entry = date_time + ip_address + \
    list_of_colors + Optional(quotedString)
```

`Optional` is another pyparsing class, used to indicate expressions that may or may not be present in the input text. It is similar to the ? qualifier in regular expressions. `quotedString` is a helpful expression that is pre-defined within pyparsing, since quoted strings are very common items found in parsing expressions. `quotedString` will match strings enclosed in single or double quotes.

Using this expression, parsing the following log data:

```
2008/04/01 12:34:56 192.168.0.1 red,blue \
    "System start"
```

returns these parsed results:

```
['2008/04/01', '12:34:56', '192.168.0.1',
   ['red', 'blue'], '"System start"']
```

LISTING 1

```
from pyparsing import Combine, Word, nums, Group,
    delimitedList, oneOf, Optional, quotedString

ip_address = Combine(Word(nums, max=3) +
                     ("." + Word(nums, max=3))*3)
list_of_colors = Group(
                delimitedList( oneOf("red blue green") ) )

date_expr = Combine(Word(nums,exact=4) +
                        ("/"+Word(nums,exact=2))*2)
time_expr = Combine(Word(nums,exact=2) +
                        (":"+Word(nums,exact=2))*2)
date_time = date_expr + time_expr

log_entry = date_time + ip_address + list_of_colors +
    Optional(quotedString)

data =
  '2008/04/01 12:34:56 192.168.0.1 red,blue "System start"'

print log_entry.parseString(data)

```

You can find the complete program and results in Listing 1.

This ability to assemble a parser by combining smaller parsing expressions into more complex expressions helps in decomposing a complex parsing problem into easily developed and understood pieces. For instance, if this log file parser were to be enhanced to support IPv6 addresses at some time in the future, it would be fairly obvious where the change should be made, and the change would be unlikely to affect other parts of the grammar.

In summary, the goal of pyparsing is to make parsers easy to write *and* easy to read. Parsers are written entirely in Python, using the standard Python object model and syntax, without using separate syntax or code generation. Parser elements use self-explanatory names such as Literal, OneOrMore, and Optional instead of using special punctuation characters that can be confused with parsing content. Parsers are built up from smaller parts to larger, using intuitive operators, such as + to combine sequential expressions and | and ^ for alternative expressions. Parsers are robust, in that they do not stumble over unexpected whitespace between defined expressions. And pyparsing provides a number of commonly used expressions, such as quoted strings, comments, comma-separated lists, and HTML tags.

From a packaging viewpoint, pyparsing is easy to incorporate into your applications. It is written in 100% Python and compatible back to Python 2.3.1, so it will run as-is on many Python platforms. Pyparsing has a very small installation footprint consisting of just a single source file, so applications that incorporate pyparsing can embed their own static version of the code without having to deal with extra installation or versioning issues. Lastly, pyparsing has a very liberal MIT license, permitting users to include pyparsing in any commercial or non-commercial application at no charge.

Writing an interpreter for a simple language

I first discovered brainf*ck (which I will henceforth genteelly refer to as "BF") about 8 years ago, when I was skimming through this list of "Hello, World!" implementations in various computing languages (http://www.roesler-ac.de/wolfram/hello.htm). I was intrigued to find the following program, written in the scatonymous language:

```
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.+++++
++..+++.>++.<<+++++++++++++++.>.+++.------.--------
.>+.
```

I was so intrigued at this notation that I had to decipher how this sequence of random-looking symbols could output anything human-readable, let alone "Hello, World!".

I learned that BF is the minimalist's computer language, consisting solely of 8 symbols, which direct the manipulation of the values in an integer array, and of a pointer into that array.

Here are the 8 instructions of BF:

- `>` increment the array pointer
- `<` decrement the array pointer
- `+` add 1 to the array value at the pointer
- `-` subtract 1 from the array value at the pointer
- `.` output the array value at the pointer as an ASCII byte
- `,` read one character of input, storing its value in the array at the location of the array pointer
- `[` skip to the command after the next matching `]` if the value at the pointer is zero ([]'s may be nested)
- `]` skip back to the command after the previous matching `[` if the value at the pointer is nonzero

There are a number of BF interpreters available in C, Perl, Python, Java, and even BF. When I was looking for examples of using pyparsing to convert parsed code into an executable structure, I wondered how difficult it would be to create a BF parser/interpreter.

The pyparsing grammar for BF is dead simple. Our first step is to define a separate literal token for each of BF's executable symbols. In pyparsing, we do this using the `Literal` class:

```
LT    = Literal("<")
GT    = Literal(">")
PLUS  = Literal("+")
MINUS = Literal("-")
DOT   = Literal(".")
COMMA = Literal(",")
```

This could also be abbreviated as:

```
LT,GT,PLUS,MINUS,DOT,COMMA = map(Literal,"<>+-.,")
```

We can create an expression that represents any BF instruction, using pyparsing's | operator.

```
bfInstr = LT | GT | PLUS | MINUS | DOT | COMMA
```

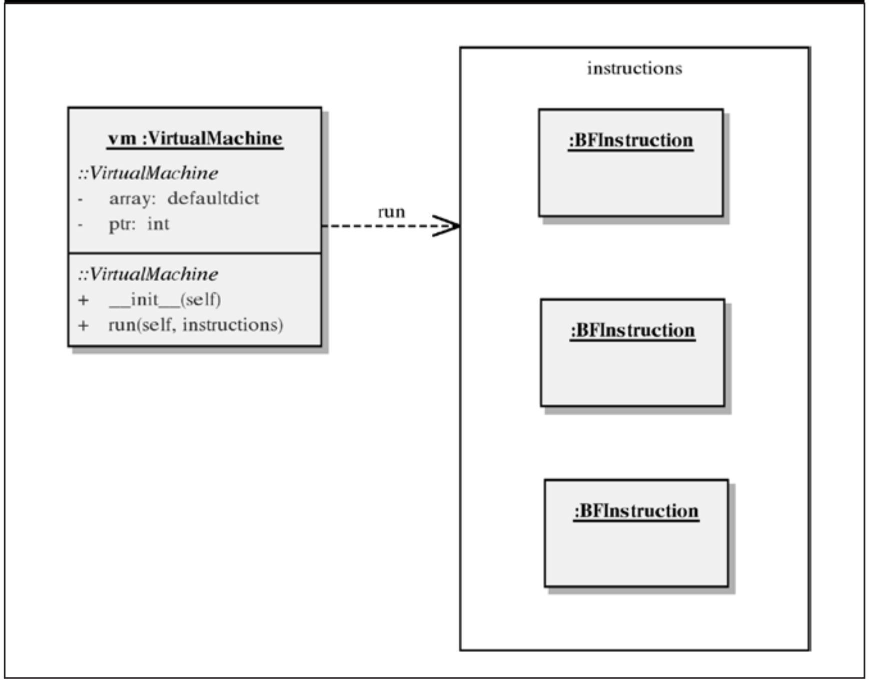
At first, I also created Literal tokens for `[` and `]`, but then I stepped back and looked at this as a parsing problem. In fact, `[` and `]` define looping structures within the larger BF program, and can be nested just like parentheses in an arithmetic expression. So instead of creating instruction expressions for the brackets, I used them to define a nested expression, using pyparsing's `nestedExpr()` helper function:

```
loop = nestedExpr("[", "]", bfInstr)
```

That is, a loop is a nested expression delimited by an opening bracket and closing bracket, enclosing zero or more `bfInstr` expressions, or inner loops nested within brackets. The `nestedExpr` helper method takes care of the recursive matching of opening and closing delimiters, and returns a hierarchical structure of the nested values.

This leaves the overall BF program expression as simply:

FIGURE 1



```
bfProgram = ZeroOrMore(bfInstr | loop)
```

This 4-line grammar can now parse the BF source code, giving us this list of tokens:

```
[
  '+', '+', '+', '+', '+', '+', '+', '+', '+',
  '+',
  [
    '>', '+', '+', '+', '+', '+', '+', '+', '>',
    '+', '+', '+', '+', '+', '+', '+', '+', '+',
    '+', '>', '+', '+', '+', '<', '<', '<', '-'
  ],
  '>', '+', '+', '.', '>', '+', '.', '+', '+',
  '+', '+', '+', '+', '+', '.', '.', '+', '+',
  '+', '.', '>', '+', '+', '.', '<', '<', '+',
  '+', '+', '+', '+', '+', '+', '+', '+', '+',
  '+', '+', '+', '+', '+', '.', '>', '.', '+',
  '+', '+', '.', '-', '-', '-', '-', '-', '-',
  '.', '-', '-', '-', '-', '-', '-', '-', '-',
  '.', '>', '+', '.'
]
```

Note that the loop of instructions within square brackets has been parsed as a sublist of tokens.

Using Parse Actions

To execute this script, we can now walk this list of parsed tokens, and for each token, call a method to execute the corresponding instruction. However, I find this to be only a little better than indexing through the original source string. All we have really done so far is to identify those regions nested within brackets.

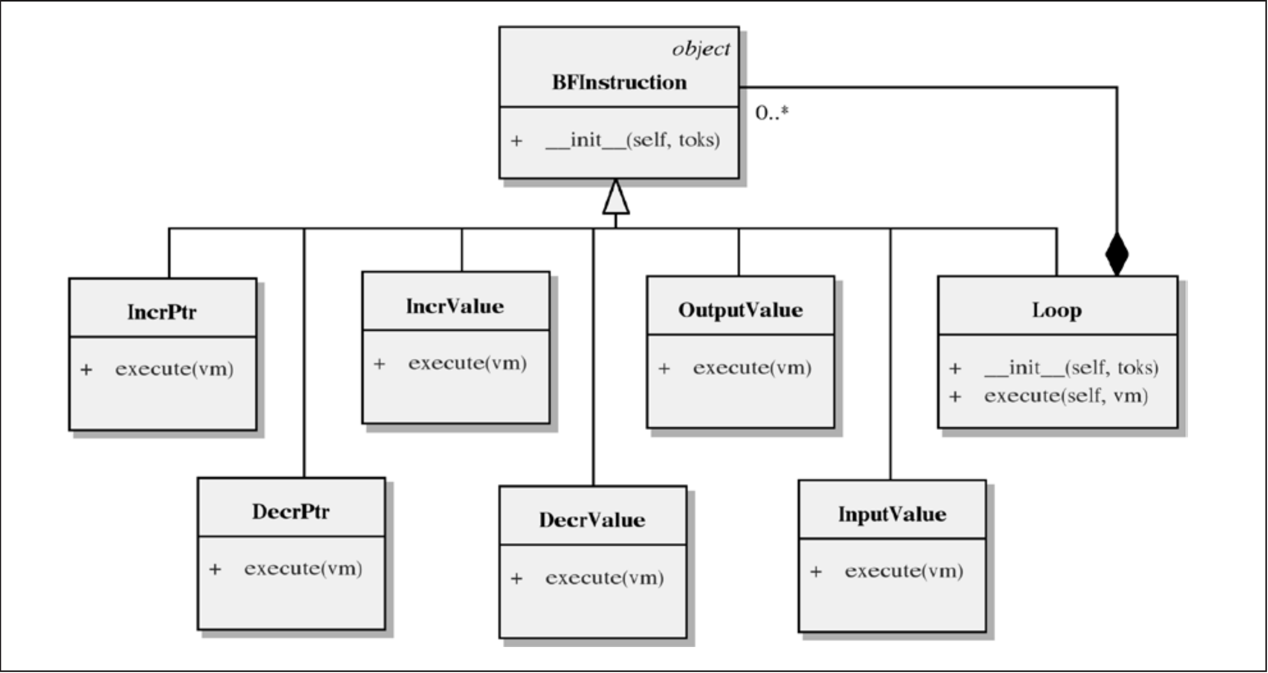
Instead, I prefer to extend the pyparsing grammar to attach a parse action to each of the instruction types. These parse actions will actually be class `__init__()` methods, so that, when the parsing process is completed, pyparsing will return a data structure of executable command objects. What kind of behavior should these command classes implement? Let's look at how they will be used once the parsing is completed.

Once the parsing step has converted the symbols into command objects, we'll use a virtual machine (VM) to execute them. The VM will maintain the BF array and array pointer, and the command objects will access these elements as they are executed. Here is the definition of the `VirtualMachine` class:

```
class VirtualMachine(object):
    def __init__(self):
        self.ptr = 0
        self.array = collections.defaultdict(int)
    def run(self, commands):
        for cmd in commands:
            cmd.execute(self)
```

Instead of defining a fixed-size array of integers, I chose to use a new feature introduced in Python 2.5, the `defaultdict`. When creating a defaultdict, you specify a factory method for initializing new entries the first time a cell is referenced. This essentially gives me an infinite array, whose indices can be positive or negative. It also creates a sparse array, creating only those cells needed to execute the BF program.

FIGURE 2



For the VM to `run()` a set of instructions, it simply invokes the `execute()` method on each one, passing itself as the argument. The command objects then access the array or array pointer, depending on the behavior of each type of command.

Figure 1 illustrates the relationship between the VM and the sequence of command objects.

That's it for the BF virtual machine. For the VM to run, the command objects will need to implement the method execute, and this execute method will take a single argument, the VM.

To create the command classes, I start with an abstract base class:

```
class BFInstruction(object):
    def __init__(self,tokens):
        pass
    def execute(vm):
        raise NotImplementedError(
            "execute method not implemented")
```

Given Python's dynamic typing, this class is completely optional. However, it does a few things for me conceptually. First of all, it helps me to document the purpose and relationship of the individual command classes by virtue of their subclassing from BFInstruction. Also, it provides a base class for implementing any common `__str__()` or debugging methods. All of the classes that implement the BF executable instructions will extend the BFInstruction class (see Figure 2).

The simplest classes to implement are those that update the array pointer:

```
class IncrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr += 1

class DecrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr -= 1
```

Each class implements the `execute()` method, and updates the VM's array pointer by incrementing or decrementing it. The remaining instructions follow similar patterns, for updating, reading, and writing values in the VM's array (see).

As these classes are defined, each is associated with its corresponding expression in the grammar, so that when that expression is parsed, the class `__init__()` method will be called, returning an executable instance of a BFInstruction subclass, instead of just a string token.

```
LT.setParseAction(DecrPtr)
GT.setParseAction(IncrPtr)
PLUS.setParseAction(IncrValue)
```

LISTING 2

```
# bf.py
# A brainf*ck interpreter using pyparsing to
# parse and compile the BF program into executable
# objects.
#
# -- Paul McGuire,  February, 2008
#
"""
BF commands:
> increment the pointer (to point to the next cell to
  the right).
< decrement the pointer (to point to the next cell to
  the left).
+ increment (increase by one) the byte at the pointer.
- decrement (decrease by one) the byte at the pointer.
. output the value of the byte at the pointer.
, accept one byte of input, storing its value in the byte
  at the pointer.
[ jump forward to the command after the corresponding ]
  if the byte at the pointer is zero.
] jump back to the command after the corresponding [ if
  the byte at the pointer is nonzero.
"""

from pyparsing import Literal, nestedExpr, ZeroOrMore,
    printables, oneOf
import collections
import sys
import pickle

# define grammar
LT,GT,PLUS,MINUS,DOT,COMMA = map(Literal,"<>+-.,")
bfInstr = LT | GT | PLUS | MINUS | DOT | COMMA
loop = nestedExpr("[","]", bfInstr)
bfProgram = ZeroOrMore(bfInstr | loop)

# ignore any characters that are not valid command chars
nonCommandChar = oneOf( list(c for c in printables
                             if c not in "<>+-.,[]") )
bfProgram.ignore(nonCommandChar)


class VirtualMachine(object):
    def __init__(self):
        self.ptr = 0
        self.array = collections.defaultdict(int)

    def run(self, instructions):
        for instr in instructions:
            instr.execute(self)

class BFInstruction(object):
    def __init__(self,toks):
        pass

class IncrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr += 1

class DecrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr -= 1

class IncrValue(BFInstruction):
    def execute(self, vm):
        vm.array[vm.ptr] = (vm.array[vm.ptr] + 1) % 128

class DecrValue(BFInstruction):
    def execute(self, vm):
        vm.array[vm.ptr] = (vm.array[vm.ptr] - 1) % 128

col = 0
class OutputValue(BFInstruction):
    def execute(self, vm):
        global col
        if vm.array[vm.ptr] == 10:
            sys.stdout.write( "\n" )
            col = 0
        else:
            sys.stdout.write(chr(vm.array[vm.ptr] % 256))
            col += 1
            if col >= 80:
                print
                col = 0

class InputValue(BFInstruction):
```

LISTING 2: Continued...

```
    def execute(self, vm):
        try:
            inchar = sys.stdin.read(1)
            if inchar == '\n':
                inchar = 10
            else:
                inchar = ord(inchar)
        except (IOError,TypeError):
            inchar = -1
        vm.array[vm.ptr] = inchar

class Loop(BFInstruction):
    def __init__(self,toks):
        self.instrs = toks[0]

    def execute(self,vm):
        while (vm.array[vm.ptr]):
            for instr in self.instrs:
                instr.execute(vm)

# attach parse actions to BF instructions
LT.setParseAction(DecrPtr)
GT.setParseAction(IncrPtr)
PLUS.setParseAction(IncrValue)
MINUS.setParseAction(DecrValue)
DOT.setParseAction(OutputValue)
COMMA.setParseAction(InputValue)
loop.setParseAction(Loop)


def dumpArray(arrayDict):
    """A debugging routine to list out the contents of the
        VM array"""
    last = max(arrayDict.keys())

    # print out a number bar
    if (last > 100):
        print "".join((i==0 or i%100) and " " or
                    str(i/100)[-1] for i in range(last))
    print "".join((i==0 or i%10) and " " or
                    str(i/10)[-1] for i in range(last))
    print "".join( str(i)[-1] for i in range(last) )
    print "-"*last

    # print out each element in the array, or '.' if not
    # a printable character
    print "".join( (32 <= arrayDict[i] < 127) and
                    chr(arrayDict[i]) or "."
                    for i in range(last) )


def runBFprogram(title, bfScript, dumpVMarray=False):
    "Function to run a BF program"
    print title+":"
    print '-'*(len(title)+1)
    compiled = bfProgram.parseString(bfScript)
    # Uncomment the next line to save compiled code
    #     for this BF program:
    # pickleString = pickle.dumps(compiled)

    vm = VirtualMachine()
    col = 0
    vm.run(compiled)
    if col: print

    if dumpVMarray:
        dumpArray(vm.array)
    print "\n"


if __name__ == "__main__":
    # from http://en.wikipedia.org/wiki/Brainfuck#Hello_world.21
    hw = """
    ++++++++++
    [>+++++++>++++++++++>+++>+<<<<-] The initial loop to
                                       set up useful values
                                       in the array
    >++.                             Print 'H'
    >+.                              Print 'e'
    +++++++.                         Print 'l'
    .                                Print 'l'
    +++.                             Print 'o'
    >++.                             Print ' '
    <<+++++++++++++++.               Print 'W'
    >.                               Print 'o'
    +++.                             Print 'r'
```

LISTING 3

```
Print greeting:
---------------
Hello World!


Sierpinski Triangle:
--------------------
                     *

                    * *

                   *   *

                  * * * *

                 *       *

                * *     * *

               *   *   *   *

              * * * * * * * *

             *               *

            * *             * *

           *   *           *   *

          * * * *         * * * *

         *       *       *       *

        * *     * *     * *     * *

       *   *   *   *   *   *   *   *

      * * * * * * * * * * * * * * * *


In conclusion:
--------------
That's BF with Pyparsing!
```

LISTING 2: Continued...

```
    ------.                          Print 'l'
    --------.                        Print 'd'
    >+.                              Print '!'
    >.                               Print newline
    """

    # from http://esoteric.sange.fi/brainfuck/bf-source/
    #
prog/triangle.bf
    sierp = """>++++[<++++++++>-]>++++++++[>++++<-]>>++>>
    >+>>>+<<<<<<<<<<[-[->+<]>[-<+>>>.<<]>>>[[->++++++++[>
    ++++<-]>.<<[->+<]+>[->++++++++++<<+>]>.[-]>]]+<<<[-[-
    >+<]+>[-<+>>>-[->+<]++>[-<->]<<<]<<<<]++++++++++.+++.
    [-]<]+++++"""

    adieu = """
    +++++++++[>+++++++++<-]>+++.<+++++[>++++<-]>.-------.
    +++++++++++++++++++.<+++++++++[>---------<-]>++++.<++
    ++++++++++[>++++++<-]>++++.<++++++++++++[>-------<-]>
    +.<++++++[>++++++<-]>--.++++.<++++++[>------<-]>--.<+
    ++++++++++[>++++++++<-]>-.--------------.+++++++++++.
    ------------.<++++++++++++[>------<-]>.<++++++++[>+++
    +++<-]>.<+++++++[>++++++<-]>-.---------.-------------
    --.+++++++++++++++++.+.----------.+++++.-------.<++++
    ++++++[>-------<-]>.<++++++[>----<-]>."""


    for title,test in (
            ("Print greeting",hw),
            ("Sierpinski Triangle",sierp),
            ("In conclusion", adieu),
            ):
        runBFprogram(title, test)
```

```
MINUS.setParseAction(DecrValue)
DOT.setParseAction(OutputValue)
COMMA.setParseAction(InputValue)
```

Now when a symbol such as + is parsed, the parse action will convert it to an IncrValue object.

The last class to implement is the class that performs the BF loop logic. In BF, a loop executes only if the array value pointed to by the VM's array pointer is not zero. At the end of the loop, the array value referenced by the array pointer is tested again, and if non-zero, the loop will repeat.

```
class Loop(BFInstruction):
    def __init__(self,toks):
        self.instrs = toks[0]

    def execute(self,vm):
        while (vm.array[vm.ptr]):
            for instr in self.instrs:
                instr.execute(vm)
```

Note that there is no code in `Loop.__init__()` to convert the tokens in the loop body into instances of BFInstruction subclasses. This is because the Loop instance is not created until the complete loop expression has been parsed, including all of the individual instructions in the loop body. Recall that each instruction has a parse action that will return the corresponding BFInstruction subclass. So when `Loop.__init__()` is called, all of the instructions have already been parsed, and the list of "tokens" passed in is really a list of BFInstruction instances.

Just as before, the Loop class must be set as the parse action for the loop expression, so that when a loop is parsed, it will be used to construct and return a Loop object.

```
loop.setParseAction(Loop)
```

Now, executing our BF program is a matter of just parsing the code to the sequence of command objects, creating a virtual machine, and then having the VM run the sequence of commands:

```
commands = bfProgram.parseString(test)
vm = VirtualMachine()
vm.run(commands)
```

And that completes the code for the BF interpreter. The complete program source code is given in Listing 2, and the output is shown in Listing 3.

Here is a sample BF program for us to parse:

```
>+>+[[-]<]
```

After our pyparsing BF parser processes this program, it returns a series of class instances as shown in Figure 3. This parsed structure can now be passed to the VM; Listing 4 shows how the VM executes this program instruction by instruction, and the values of the array and array pointer for each step.

Converting From an Interpreter to Compiler

One final note: If you like, you can pickle the commands and save them to an external file:

```
pickleString = pickle.dumps(commands)
file("hello.b_pckl","w").write(pickleString)
```

These commands can later be unpickled and executed

LISTING 4

```
 program     Instruction                  ptr    array
----------  ----------------------------  ---  -----------
>+>+[[-]<]  (Program start)                0   0 0 0 0 0 0
                                               ^

>+>+[[-]<]  Increment array ptr            1   0 0 0 0 0 0
^                                                ^

>+>+[[-]<]  Increment array value at ptr   1   0 1 0 0 0 0
 ^                                               ^

>+>+[[-]<]  Increment array ptr            2   0 1 0 0 0 0
  ^                                                ^

>+>+[[-]<]  Increment array value at ptr   2   0 1 1 0 0 0
   ^                                               ^

>+>+[[-]<]  Loop while array[ptr] != 0     2   0 1 1 0 0 0
    ^                                              ^

>+>+[[-]<]  Loop while array[ptr] != 0     2   0 1 1 0 0 0
     ^                                             ^

>+>+[[-]<]  Decrement array value at ptr   2   0 1 0 0 0 0
      ^                                            ^

>+>+[[-]<]  End loop if array[ptr] == 0    2   0 1 0 0 0 0
       ^                                           ^

>+>+[[-]<]  Decrement array ptr            1   0 1 0 0 0 0
        ^                                        ^

>+>+[[-]<]  End loop if array[ptr] == 0    1   0 1 0 0 0 0
         ^                                       ^

>+>+[[-]<]  Loop while array[ptr] != 0     1   0 1 0 0 0 0
    ^                                            ^

>+>+[[-]<]  Loop while array[ptr] != 0     1   0 1 0 0 0 0
     ^                                           ^

>+>+[[-]<]  Decrement array value at ptr   1   0 0 0 0 0 0
      ^                                          ^

>+>+[[-]<]  End loop (array[ptr] == 0)     1   0 0 0 0 0 0
       ^                                         ^

>+>+[[-]<]  Decrement array ptr            0   0 0 0 0 0 0
        ^                                      ^

>+>+[[-]<]  End loop (array[ptr] == 0)     0   0 0 0 0 0 0
         ^                                     ^

>+>+[[-]<]  (Program end)                  0   0 0 0 0 0 0

```

without having to reparse the input BF source code. With just a few extra statements, you have converted your interpreter into a compiler! Or you can take this idea a step further and break up the original BF interpreter program into separate compiler and executor programs.

The compiler program only needs the grammar definition, and the barest skeleton of the run-time BFInstruction classes. The base BFInstruction class remains the same, but the derived classes need only define enough to capture the run-time state so that their instances are "picklable". Only the Loop class requires any more than the default empty __init__ method. These classes reduce to:

```
class IncrPtr(BFInstruction): pass
class DecrPtr(BFInstruction): pass
class IncrValue(BFInstruction): pass
class DecrValue(BFInstruction): pass
class OutputValue(BFInstruction): pass
class InputValue(BFInstruction): pass
class Loop(BFInstruction):
    def __init__(self,toks):
        self.instrs = toks[0]
```

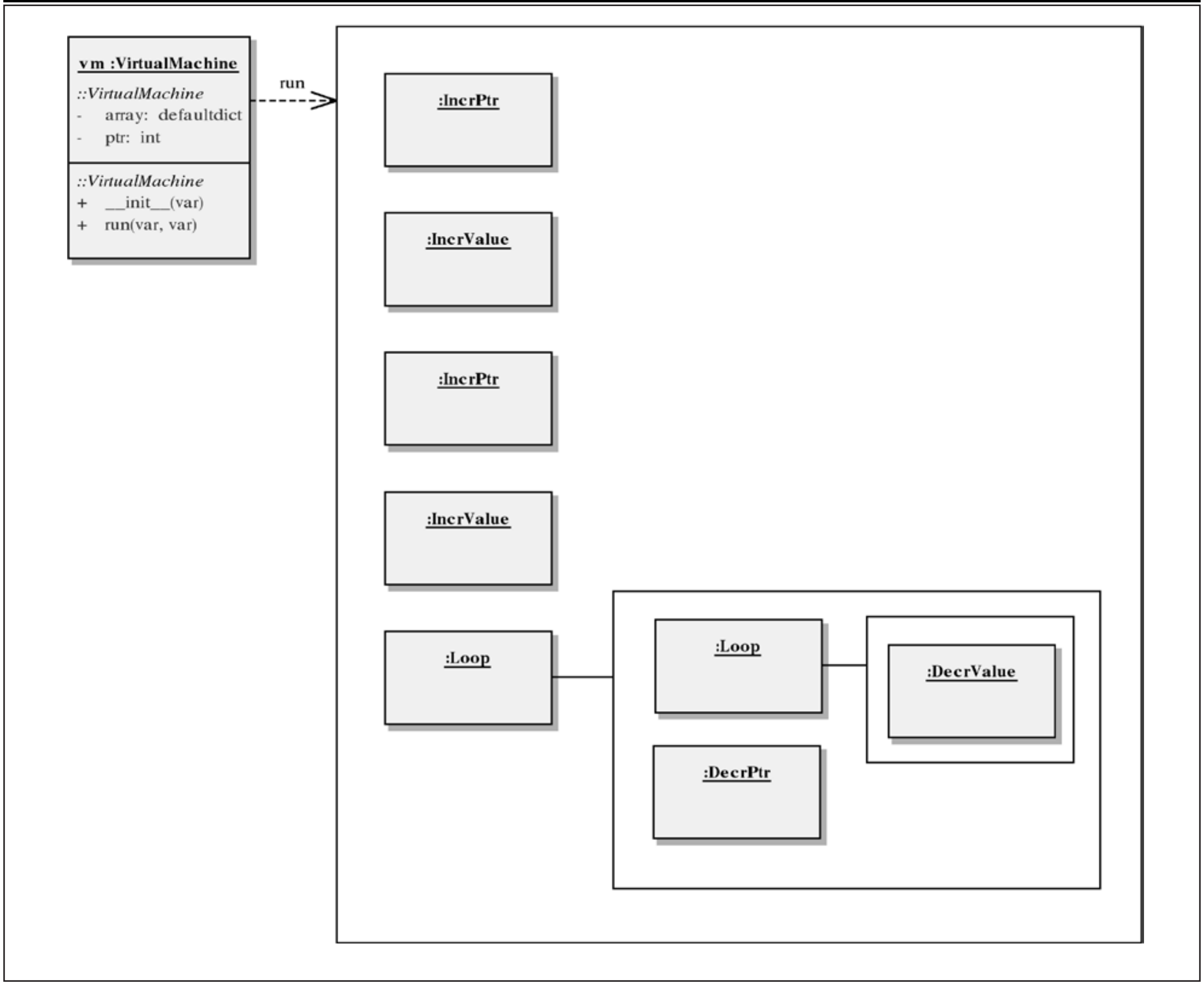
The grammar portion of the code also remains the same, including the parse actions that refer to these skeleton definitions of the run-time classes. But now instead of running the code immediately after parsing it, the executable classes are pickled and stored in external files.

```
compiled = bfProgram.parseString(test)
pickleString = pickle.dumps(compiled)
file(title+".b_pckl","w").write(pickleString)
```

Listing 5 shows the compiler-only version of the BF interpreter.

Conversely, the run-time executor needs none of the

FIGURE 3



LISTING 5

```
"""
BF commands:
> increment the pointer (to point to the next cell to
  the right).
< decrement the pointer (to point to the next cell to
  the left).
+ increment (increase by one) the byte at the pointer.
- decrement (decrease by one) the byte at the pointer.
. output the value of the byte at the pointer.
, accept one byte of input, storing its value in the byte
  at the pointer.
[ jump forward to the command after the corresponding ]
  if the byte at the pointer is zero.
] jump back to the command after the corresponding [ if
  the byte at the pointer is nonzero.
"""

from pyparsing import *
import pickle

# define grammar
LT,GT,PLUS,MINUS,DOT,COMMA = map(Literal,"<>+-.,")
bfInstr = LT | GT | PLUS | MINUS | DOT | COMMA
loop = nestedExpr("[","]", bfInstr)
bfProgram = ZeroOrMore(bfInstr | loop)

# ignore any characters that are not valid command chars
nonCommandChar = oneOf( list(c for c in printables if c not in
"<>+-.,[]") )
bfProgram.ignore(nonCommandChar)

class BFInstruction(object):
    def __init__(self,toks):
        pass
class IncrPtr(BFInstruction): pass
class DecrPtr(BFInstruction): pass
class IncrValue(BFInstruction): pass
class DecrValue(BFInstruction): pass
class OutputValue(BFInstruction): pass
class InputValue(BFInstruction): pass
class Loop(BFInstruction):
    def __init__(self,toks):
        self.instrs = toks[0]

# attach parse actions to BF instructions
LT.setParseAction(DecrPtr)
GT.setParseAction(IncrPtr)
PLUS.setParseAction(IncrValue)
MINUS.setParseAction(DecrValue)
DOT.setParseAction(OutputValue)
COMMA.setParseAction(InputValue)
loop.setParseAction(Loop)

def compileBFprogram(title, bfScript):
    "Function to run a BF program"
    print title+":"
    print '-'*(len(title)+1)
    compiled = bfProgram.parseString(bfScript)
    pickleString = pickle.dumps(compiled)
    file(title+".b_pckl","w").write(pickleString)

if __name__ == "__main__":
    # from http://en.wikipedia.org/wiki/Brainfuck#Hello_world.21
    hw = """
    ++++++++++
    [>+++++++>++++++++++>+++>+<<<<-] The initial loop to
                                       set up useful values
                                       in the array
    >++.                             Print 'H'
    >+.                              Print 'e'
    +++++++.                         Print 'l'
    .                                Print 'l'
    +++.                             Print 'o'
    >++.                             Print ' '
    <<+++++++++++++++.               Print 'W'
    >.                               Print 'o'
    +++.                             Print 'r'
    ------.                          Print 'l'
    --------.                        Print 'd'
    >+.                              Print '!'
    >.                               Print newline
    """

    # from http://esoteric.sange.fi/brainfuck/bf-source/
    #
prog/triangle.bf
```

parser code, but only the VM class and the run-time portion of the BFInstruction classes. Listing 6 contains the source for the run-time executor.

An interesting by-product of this separation is that the run-time code can be changed drastically, such as running of the compiled code at the console, or within a GUI, or in part of a web service, all run from the same compiled `.b_pckl` files, without any reparsing required.

In Conclusion

Was this all really worth it? We could have skipped a lot of this parser business and just read the symbols one by one. However, the pattern of parser -> command objects -> VM -> command execution is useful to learn, and the simple syntax of BF allows us to go through this pattern step by step. The same pattern can be used to implement your own domain-specific languages, for search queries, program commands, game commands, schemas, state machines, etc., just to give you a few ideas.

What else can you do with pyparsing? Pyparsing has been put to a variety of text processing uses, including: an assembler for robot control; a web page data extractor (this is a very popular application); a Chinese-to-English Python script converter; a search query processor (another popular application); a code conversion utility for a library API upgrade; an adventure game command processor; an ad hoc Apache log file scanner for web attack diagnosis; a chess game log parser; and a Verilog language parser.

LISTING 5: Continued...

```
    sierp = """>++++[<++++++++>-]>++++++++[>++++<-]>>++>>
    >+>>>+<<<<<<<<<<[-[->+<]>[-<+>>>.<<]>>>[[->++++++++[>
    ++++<-]>.<<[->+<]+>[->++++++++++<<+>]>.[-]>]]+<<<[-[-
    >+<]+>[-<+>>>-[->+<]++>[-<->]<<<]<<<<]++++++++++.+++.
    [-]<]+++++"""

    adieu = """
    +++++++++[>+++++++++<-]>+++.<+++++[>++++<-]>.-------.
    ++++++++++++++++++++.<+++++++++[>---------<-]>++++.<++
    ++++++++++[>+++++++<-]>++++.<++++++++++++[>-------<-]>
    +.<++++++[>++++++<-]>--.++++.<++++++[>------<-]>--.<+
    ++++++++++[>++++++++<-]>-.--------------.+++++++++++.
    ------------.<++++++++++++[>------<-]>.<++++++++[>+++
    +++<-]>.<+++++++[>++++++<-]>-.---------.------------
    --.+++++++++++++++++.+.----------.+++++.-------.<++++
    ++++++[>-------<-]>.<++++++[>----<-]>."""


    for title,test in (
            ("Print greeting",hw),
            ("Sierpinski Triangle",sierp),
            ("In conclusion", adieu),
            ):
        compileBFprogram(title, test)

```

The simplicity and speed in creating parsers with pyparsing enables you to quickly crank out ad hoc utilities to sift through text files, or incorporate a clear and maintainable text parser into a long-term software project.

So in parting, let me just say:

```
+++++++++[>+++++++++<-]>+++.<+++++[>++++<-]>.------
-.+++++++++++++++++++.<+++++++++[>---------<-]>++++
.<++++++++++++[>++++++<-]>++++.<++++++++++++[>-----
--<-]>+.<++++++[>++++++<-]>--.++++.<++++++[>------<
-]>--.<+++++++++++[>++++++++<-]>-.--------------.++
+++++++++.------------.<++++++++++++[>------<-]>.<+
+++++++[>++++++<-]>.<+++++++[>++++++<-]>-.---------
.---------------.+++++++++++++++++.+.----------.+++
++.-------.<++++++++++[>-------<-]>.<++++++[>----<-
]>.
```

Or, `That's BF with Pyparsing!`

For more information on pyparsing

You can find more information, news, documentation, FAQs, example code, and sample expressions at the pyparsing wiki. You can post questions on the *Discussion* tab of the wiki home page, or use the pyparsing mailing list, *pyparsing-users@lists.sourceforge.net*.

LISTING 6

```python
import collections
import sys
import pickle

class VirtualMachine(object):
    def __init__(self):
        self.ptr = 0
        self.array = collections.defaultdict(int)

    def run(self, instructions):
        for instr in instructions:
            instr.execute(self)

class BFInstruction(object):
    def __init__(self,toks):
        pass

class IncrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr += 1

class DecrPtr(BFInstruction):
    def execute(self, vm):
        vm.ptr -= 1

class IncrValue(BFInstruction):
    def execute(self, vm):
        vm.array[vm.ptr] = (vm.array[vm.ptr] + 1) % 128

class DecrValue(BFInstruction):
    def execute(self, vm):
        vm.array[vm.ptr] = (vm.array[vm.ptr] - 1) % 128

col = 0
class OutputValue(BFInstruction):
    def execute(self, vm):
        global col
        if vm.array[vm.ptr] == 10:
            sys.stdout.write( "\n" )
            col = 0
        else:
            sys.stdout.write( chr(vm.array[vm.ptr] % 256) )
            col += 1
```

LISTING 6: Continued...

```python
            if col >= 80:
                print
                col = 0

class InputValue(BFInstruction):
    def execute(self, vm):
        try:
            inchar = sys.stdin.read(1)
            if inchar == '\n':
                inchar = 10
            else:
                inchar = ord(inchar)
        except (IOError,TypeError):
            inchar = -1
        vm.array[vm.ptr] = inchar

class Loop(BFInstruction):
    def __init__(self,toks):
        self.instrs = toks[0]

    def execute(self,vm):
        while (vm.array[vm.ptr]):
            for instr in self.instrs:
                instr.execute(vm)

def dumpArray(arrayDict):
    last = max(arrayDict.keys())
    if (last > 100):
        print "".join( (i==0 or i%100) and " " or
                        str(i/100)[-1] for i in range(last) )
    print "".join( (i==0 or i%10) and " " or
                        str(i/10)[-1] for i in range(last) )
    print "".join( str(i)[-1] for i in range(last) )
    print "-"*last
    print "".join( (32 <= arrayDict[i] < 127) and
            chr(arrayDict[i]) or "." for i in range(last) )

for title,test in (
            ("Print greeting",hw),
            ("Sierpinski Triangle",sierp),
            ("In conclusion", adieu),
        ):
    print title+":"
    pickleString = file(title+".b_pckl").read()
    compiled = pickle.loads(pickleString)

    vm = VirtualMachine()
    col = 0
    vm.run(compiled)
    if col: print
    dumpArray(vm.array)
    print "\n"
```

PAUL MCGUIRE earned his bachelor's degree in mechanical engineering at Rensselaer Polytechnic Institute and a master's in manufacturing systems engineering at the University of Texas at Austin. After dabbling in Python, Paul decided to pursue his interest in text processing, and developed the pyparsing class library. This project has taken on a life of its own and has been downloaded nearly 40,000 times. Paul has enjoyed hearing from pyparsing users in the United States, Canada, Germany, Sweden, Russia, the Czech Republic, France, Brazil, Mexico, Japan, Korea, the United Kingdom, Italy, India, the Netherlands, Finland, Poland, Denmark, Australia, Slovenia, Belgium, and Luxembourg.

Welcome to Python

Introducing Descriptors and Properties

by Mark Mruss

New-style classes were introduced to Python with the release of Python 2.2. And with these new-style classes came descriptors and properties. This article will introduce the descriptor protocol, descriptors, and properties.

Introduction

New-style classes were introduced to Python with the release of Python 2.2. A new-style class is any class that is derived from the `object` base class. New-style classes give Python programmers many new (and initially confusing) features. One such feature is the *descriptor protocol*, and more specifically *descriptors* themselves.

Descriptors give Python programmers the ability to easily and efficiently create *managed attributes*. Managed attributes can be thought of as attributes that are not accessed directly. Instead their access is "managed" by something else, generally a class or a function.

If you haven't come across this before, you are probably wondering why one would want to manage attribute access? One reason might be that you don't want people to be able to delete the attribute. Another reason may be that you need to ensure that your attribute

REQUIREMENTS

PYTHON: 2.2+

Useful/Related Links:
- Implementing Descriptors - *http://docs.python.org/ref/descriptors.html*
- Invoking Descriptors - *http://docs.python.org/ref/descriptor-invocation.html*
- Properties: attributes managed by get/set methods - *http://www.python.org/download/releases/2.2/descrintro/#property*
- "How-To Guide for Descriptors" by Raymond Hettinger - *http://users.rcn.com/python/download/Descriptor.htm*

data is always valid. Or perhaps attribute x is based on attribute y, so every time the value of y changes you want to update the value of x. From these few examples you can see the many possible cases where you might want to control access to certain attributes.

For those of you familiar with other programming languages, this type of access is often referred to as *getters* and *setters*. In many languages, implementing getters and setters means using private variables and public functions that get and set the variable's value. Since Python doesn't (really) have private variables, the descriptor protocol is basically a built-in and 'Pythonic' way to way to achieve something similar.

This article will introduce you to the descriptor protocol, descriptors, and properties. It will focus on demonstrating how to use them to create managed attributes. Since the descriptor protocol requires new-style classes, all of the examples in this article require Python 2.2 or newer.

A few definitions

Before moving forward, it is important to understand a few related terms. These terms will introduce some basic concepts and help you follow along with the remainder of the article.

A *descriptor* is an "object attribute with *binding behavior*", one whose attribute access has been overridden by methods in the descriptor protocol. [1] In other words an "attribute whose usage resembles attribute access, but whose implementation uses method calls." [2]

The *descriptor protocol* consists of three methods: `__get__`, `__set__`, and `__delete__`.

A *data descriptor* is a descriptor with the `__get__` and `__set__` methods of the descriptor protocol defined.

A *non-data descriptor* is a descriptor with only the `__get__` method of the descriptor protocol defined. "Python methods (including staticmethod() and classmethod()) are implemented as non-data descriptors." [3]

A *property* is a built-in type that implements the descriptor protocol and allows you to easily create data descriptors.

Don't worry if you don't fully understand these definitions, the remainder of this article will hopefully clarify any confusion you have.

The Descriptor Protocol

Let's take a closer look at the descriptor protocol and see how we can use it to create a descriptor. As previously mentioned, the descriptor protocol is made up of three methods: `__get__`, `__set__`, and `__delete__`. These methods have specific signatures and they are as follows, where `self` is the class that owns the methods:

```
__get__(self, instance, owner)
__set__(self, instance, value)
__delete__(self, instance)
```

These three methods represent the three basic operations that you perform on attributes in general: querying the value of that attribute; assigning a value to it; and, (very rarely) deleting it. They work as follows:

The `__get__` method is called when the attribute's value is being queried. The `__get__` "method should return the (computed) attribute value or raise an AttributeError exception." [4] This is where access to the attribute's value is managed.

The `__set__` method is used in the assignment operation. It is called when we want to set the attribute value. This is where you can control what values, or types of values are being assigned to your attribute.

Finally, the `__delete__` method is called when we want to delete the attribute. Here you can decide whether or not to delete the attribute.

There are also three different parameters passed to the three methods (excluding the standard `self` parameter for methods that belong to a class):

The `owner` parameter always holds the owner class. This means that it is the actual class, and not an instance of the class. So if the descriptor is in a class called `MyClass`, `owner` will be that class.

The `instance` parameter is an instance of class type `owner`. It is "the instance that the attribute was accessed through" [4], or `None` if the attribute is being accessed through the class (`owner`) instead of an instance.

The `value` parameter holds the new value the attribute is being set to.

This difference between `owner` and `instance` might be a bit confusing, so let's look at a quick example. Let's say we have a descriptor `my_descriptor` in the class `MyClass`, if we were to run the following code:

```
my_class_instance = MyClass()
print my_class_instance.my_descriptor
```

The second line queries the descriptor's value and results in the `__get__` method being called with the `instance` parameter being `my_class_instance`. The `owner` parameter will be the `MyClass` class.

Note: Notice that we treat the descriptor `my_descriptor` as though it is a normal attribute. We don't call:

```
print my_class_instance.my_descriptor.__get__(my_
class, MyClass)
```

This is what was meant by: "attributes whose usage resembles attribute access, but whose implementation uses method calls." [5]

If the following code were run:

```
print MyClass.my_descriptor
```

The `__get__` method will again be called. This time the `instance` parameter will be `None` and the `owner` parameter will be the `MyClass` class.

Note: Only the `__get__` method has the `owner` parameter. This means that it is the only function in the descriptor protocol that can be accessed through the class. Setting and deleting the descriptor through the class actually changes *what* the variable is. For example, if we tried to assign the numeric value 2 to a descriptor variable using the class, we would not access the descriptor's `__set__` method. Instead we would change the type of the variable from a descriptor to an integer with the value of 2.

A Simple Descriptor Example

A simple "transparent" descriptor example can be found in Listing 1. The first thing to notice in the code is that both the `SimpleDescriptor` and `MyClass` classes are "new-style" classes because they are derived from `object`. This is important because, as mentioned above, descriptors only work with "new-style" classes. The second point to notice is that the descriptor has class scope as opposed to instance scope. There are also two extra print statements included in the code. They are there to let us follow the execution a little more easily.

Using Listing 1 to execute the following code:

```
my_instance = MyClass()
my_instance.my_value = 416
print my_instance.my_value
```

The output is:

```
Setting to 416
Getting value: 416
416
```

As you can see the second line (`my_instance.data_descriptor = 416`) calls the `__set__` method and sets the `_value` attribute. When we call `print my_instance.data_descriptor` the `__get__`, method is called and the `_value` attribute is returned.

The Problem with the Simple Example

The previous example may look like a perfectly good descriptor, but there is something wrong with it. Take a look at what happens when we run this new code:

```
# Create the first instance
my_instance = MyClass()
my_instance.my_value = 416
# Create a second instance
my_second_instance = MyClass()
my_second_instance.my_value = 204
# What was the first instance's value?
print my_instance.my_value
```

We get the following output:

```
Setting to 416
Setting to 204
Getting value: 204
204
```

Notice that when we set the second instance's `my_value` descriptor to be 204, we are also setting the first instance's. This is because `my_value` has class scope, so both instances (and the class itself) share the same instance of the `SimpleDescriptor` class. Since `SimpleDescriptor` only stores one value, they all actually share the same value. We will get the same results if we check what the class's value of `my_value` is:

LISTING 1

```
class SimpleDescriptor(object):

    def __get__(self, instance, owner):
        # Check if the value has been set
        if (not hasattr(self, "_value")):
            raise AttributeError
        print "Getting value: %s" % self._value
        return self._value

    def __set__(self, instance, value):
        print "Setting to %s" % value
        self._value = value

    def __delete__(self, instance):
        del(self._value)

class MyClass(object):
    my_value = SimpleDescriptor()

```

```
# Create the first instance
my_instance = MyClass()
my_instance.my_value = 416
# Create the second instance
my_second_instance = MyClass()
my_second_instance.my_value = 204
# What's the class's value?
print MyClass.my_value
```

We get the following results:

```
Setting to 416
Setting to 204
Getting value: 204
204
```

In the last line of the code (`print MyClass.my_value`) we simply use the class (ignoring both instances) in order to get the value of `my_data`. This will be an instance where the `__get__` function will be called with the `instance` parameter set to `None`.

Note: This is not a problem if you want to share the value across all instances.

The Solution to the Problem

In order to get around this issue, you have to remember that if you want values unique to each instance your descriptors must store values that are unique to each instance. This can be in the instance itself, in a dictionary in the descriptor, or perhaps in a text file. Just make sure that the value is unique to each instance. Though it seems like a simple solution, in practice a suitable solution for all cases is difficult to implement. As a result you should pick an implementation for your specific situation.

LISTING 2

```
class FixedDescriptor(object):

    def __init__(self, value_name):
        self.value_name = str(value_name)

    def __get__(self, instance, owner):
        if (instance is None):
            raise AttributeError
        elif self.value_name not in instance.__dict__:
            raise AttributeError
        return instance.__dict__[self.value_name]

    def __set__(self, instance, value):
        print "Setting to %s" % value
        instance.__dict__[self.value_name] = value

    def __delete__(self, instance):
        if self.value_name in instance.__dict__:
            del(instance.__dict__[self.value_name])


class MyClass(object):
    my_value = FixedDescriptor("__value")

```

Using each instance as a key, one can store the value in a dictionary in the descriptor itself. The problem with this solution should be obvious to programmers familiar with Python dictionaries: only immutable types can be used as keys. This is fine if you know what object you are working with, but what about in the future when you want to add a descriptor to your sub-classed list?

Another solution is to store the value in the instance itself. You can do this by easily adding the value to the instance's `__dict__`. The limitation is that the descriptor needs to be given a suitable key so that there is no collision with anything already in the instance's `__dict__`.

Listing 2 shows a solution to the problem from the previous section where the value is stored in the instance's `__dict__`. Values are indexed using a key name provided during the creation of the descriptor. Aside from where we store the value, there is little difference between Listing 1 and Listing 2. Notice that the value name is a converted into a string. This is done to ensure that it can be used as a key.

The major usage difference between Listing 1 and Listing 2 is the fact that the `__get__` method no longer works at class level; it only works at the instance level. This should be obvious since this solution stores the value in the instance. If there is no instance, we have nowhere to store the value! If you attempt to access the descriptor at class level an attribute error will be raised (the first `if` statement):

```
def __get__(self, instance, owner):
    if (instance is None):
        raise AttributeError
    elif self.value_name not in instance.__dict__:
        raise AttributeError
    return instance.__dict__[self.value_name]
```

Easy Data Descriptors with Properties

The descriptor provided in Listing 2 will work for many situations but for my money the easiest way to implement data descriptors is to use the `property` type. I recommend using it unless you need the same descriptor across many different classes or attributes (i.e. for type validation). Properties can be thought of as a simple and easy way to create data descriptors. The `property` type implements the descriptor protocol and gets around the problem of where to store the descriptor value in an easy way: it lets the class that created the descriptor deal with it.

The full signature of the property function is as follows:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

Where fget, fset, fdel are methods that will be called when the __get__, __set__, and __del__ members of the descriptor protocol are called. The doc parameter is a string that will be used as the docstring for the descriptor. If the doc parameter is not specified, the docstring of the fget method is used.

The signature of the fget, fset, fdel functions are as follows:

```
fget(instance)
fset(instance, value)
fdel(instance)
```

In the three signatures, instance is a reference to the object that owns the property attribute. This is the same instance that is passed to the descriptor protocol. Since instance is the first parameter of each method, these are, for all intents and purposes, member methods of a class. The value parameter is the same as the value parameter that is passed to the __set__ method. It is what we are setting the attribute to.

LISTING 3

```
class MyClass(object):
    #Create the fget, fset, and fdel methods
    def __get_value(self):
        print "Getting value: %s" % self.__value
        return self.__value

    def __set_value(self, value):
        print "Setting to %s" % value
        self.__value = value

    def __del_value(self):
        del(self.__value)

    #Create the property
    my_value = property(__get_value
            , __set_value
            , __del_value
            , "This is my property")

    def __init__(self, value=0):
        self.my_value = value

```

LISTING 4

```
class WonkyDescriptor(object):
    """This Descriptor isn't read only"""
    def __init__(self, value_name):
        self.value_name = str(value_name)

    def __get__(self, instance, owner):
        if (instance is None):
            raise AttributeError
        elif (not instance.__dict__.has_key(self.value_name)):
            raise AttributeError
        print "Getting a value"
        return instance.__dict__[self.value_name]

class MyClass(object):

    my_value = WonkyDescriptor("_value")

    def __init__(self, value):
        #initial value
        self._value = value

```

" This article will introduce you to the descriptor protocol, descriptors, and properties."

The basic way to implement properties can be seen in Listing 3. As you can see, what we do for a property is very similar to what we did for our initial descriptor in Listing 1. We set up the three functions necessary for the descriptor protocol, and then use them to create our property attribute. Creating the property is very simple:

```
my_value = property(__get_value
        , __set_value
        , __del_value
        , "This is my property")
```

Make Data Attributes Data-Descriptors

A common reason to use descriptors is to create read-only attributes. This means allowing callers to get (or access) an attribute's value, but not allowing them to set its value. At first glance it seems as though one can accomplish this by using descriptors with only the __get__ method defined, i.e. a "non-data descriptor".

While this seems to create a read-only descriptor attribute, it actually does nothing of the sort. Instead of using the descriptor's __set__ function for the assignment operation (since no __set__ function is defined), the default Python assignment operation is used. This means that instead of stopping the assignment operation, a new attribute will be created in the instance (remember that descriptors have class scope) having the same name as the descriptor attribute and taking precedence in future operations.

This may be a bit confusing so let's look at the example in Listing 4. It's very much like our previous descriptor examples except it does not specify the __set__ and __del__ functions. Let's look at what happens when we try to use this as though it were a read-only attribute:

```
# Create the first instance
my_instance = MyClass(23)
print my_instance.my_value
my_instance.my_value = "Don't set me!"
print my_instance.my_value
```

When we run this, we get the following output:

```
Getting a value
23
Don't set me!
```

As you can see we did not succeed in creating a read-only attribute, instead something else happened. The string "Getting a value" is printed out when we print out my_value for the first time. This means that we are accessing the value through the descriptor. We then set the value of the descriptor attribute and print out that new value. Since "Getting a value" isn't printed out a second time, we know that the descriptor's __get__ method is not accessed when the second print statement is called.

Let's take a closer look at what is happening using the following code, which prints out the instance's __dict__:

```
#Create the first instance
my_instance = MyClass(23)
print my_instance.__dict__
my_instance.my_value = "Don't set me!"
print my_instance.__dict__
```

This results in the following, which explains what is happening:

```
{'_value': 23}
{'_value': 23, 'my_value': "Don't set me!"}
```

When we first access the __dict__, we see our value indexed by the _value key that we told the descriptor to use. Now look at the second __dict__. The old value is still there, but so is a new value my_value. When we assigned "Don't set me!" to my_value, we didn't overwrite the descriptor attribute at class scope, we created a new instance attribute! And it is not exactly a read-only attribute.

In order to create a read-only attribute, you need to create a data descriptor where the __set__ method raises an AttributeError exception. An easy way to do this is to create a property and only set the fget function. Instead of using the descriptor in Listing 4, we can construct a read-only attribute as follows:

```
def __get_value(self):
    return self._value
my_value = property(__get_value)
```

Conclusion

I hope that after reading this article you can see how powerful descriptors (especially data-descriptors) can be. Once you get the hang of them, they are a great way to implement getters and setters as well as read-only attributes. Data-descriptors are also very useful for performing validation during the assignment operation, i.e. ensuring that a value remains a specific type, or in a specific form.

The more you play with descriptors and use them in your code, the more you'll see how sophisticated and useful they can be. That being said, it's important to remember that unless you have a good reason to use descriptors you probably don't need them. There's no need to sacrifice the readability of your code or the dynamism of Python just for the sake of using descriptors. But when you have a real reason for managed attributes, you'll find that descriptors are more than up to the task.

FOOTNOTES

[1] http://docs.python.org/ref/descriptor-invocation.html
[2] http://www.python.org/download/releases/2.2/descrintro/#property
[3] http://docs.python.org/ref/descriptor-invocation.html
[4] http://docs.python.org/ref/descriptors.html
[5] http://www.python.org/download/releases/2.2/descrintro/#property

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

Random Hits

PyCon at Ninety Miles Per Hour

by Steve Holden

Steve offers a report on what he feels was a very special conference.

Wednesday

By the time you read this it will be old hat, but as I write it (two weeks after the event) I am still very energized by my experiences at PyCon. It was a heck of a week.

I arrived on the Wednesday afternoon so as to be in good time to present my tutorial on Thursday morning. I was greeted by Drew Moore, an old PyCon hand, who had arrived on Tuesday anticipating that the organizers would be able to use some help, only to discover that none of them were arriving until Thursday.

After unpacking and settling down I went downstairs to find that the conference was slowly beginning to wind up. I had lunch with Doug Napoleone, who has put together some incredible technology for the conference web site, and we were joined by a number of other organizers including Ted Pollari, who had been a local when the Chicago Python users won the conference bid, but had since moved away from the area. Ted has been a general factotum, taking on all kinds of responsibilities, not least among them the task of allotting financial support to those who wanted to come to the conference but couldn't cover all their bills. This year over $20,000 was shared among many recipients, and the conference was better for it.

I spent the rest of the afternoon catching up on e-mail and trying to keep up with my various client projects. I went downstairs in the early evening to share a meal of classic Chicago-style pizza with the organizers. Incredibly, pretty much everything was taken care of though I did manage to make myself useful helping to sort the badges for the registration desk. After a couple of drinks in the bar I went upstairs and tried to get a relatively early night to be ready to present first thing.

Thursday

My tutorial was 'Python 101 for Programmers', and covered pretty much the whole of the language (though not in full depth, naturally) in 3 hours. I was pleasantly surprised to discover that they had put the session in a room that would seat over 60 people, which meant that we could fit in a few additional wait-listed people with no problem. The idea behind the session was to provide a fast introduction to the language for people who were attending their first PyCon and were fairly new to the language. I don't know how it seemed to the audience, but from the front of the room the time fairly flew by, and I had to run over by ten minutes to cover all the materials.

In the afternoon I went to the tutorial on 'Getting Started with Pylons/TurboGears and WSGI'. Due to the wireless network being almost unusable, the tutorial got off to a very slow start, as Mark Ramm and Ben Bangert had assumed that everyone would be able to download their latest and greatest distributions from the Internet. Eventually, we created a sort of USB-drive bucket brigade, and everyone who needed software managed to get it. The primary material was a guided re-run of the TurboGears 'Wiki in 20 minutes' demonstration, which I would have liked to finish, but sadly there wasn't really time. Mark did have us build a couple of little WSGI servers, though, and that was educational. I'm sorry to say I didn't pay proper attention to Ben Bangert's talk, as I was still trying to get the TurboGears code to run. Both presenters dealt well with being put in difficult circumstances by the absence of a network, and having been in similar circumstances myself I know it was a tremendous learning experience for them.

A quick dinner was followed by a third tutorial, this time from Titus Brown and Grig Georghiu on 'Practical Applications of Agile (Web) Testing Tools'. This was a very interesting overview of web testing technologies. The talk covered a number of tools I had heard of but didn't have much familiarity with, including nose, twill, figleaf and scotch, plus a deeper look than I had taken before at Selenium and the buildbot continuous integration tool. I haven't yet downloaded the slides for the presentation, but I am looking forward to reviewing them.

I finished the evening in the hotel bar, where I am frequently to be found during conferences. Not only do you often meet the most interesting people there, it's good to have a couple of pints and catch up with the many people I am sporadically in touch with by email but normally only meet once a year – you guessed it, at PyCon. This was the sixth PyCon, and I've been to every one. Although it's always been billed as a community conference it has a special feel for me, which was echoed by a number of the other regular participants. It's a bit like coming home every year.

Friday

On Friday morning conference chair David Goodger's opening remarks underlined the value of being a Python community member. His first PyCon had been the second one in DC in 2004; at the time he had been unemployed and depressed, and was only able to attend the conference by virtue of donations from a number of open-handed individuals. He is now the secretary of the Python Software Foundation and happily ensconced in a job that he found through the Python Job Board. David is certainly repaying the community in a big way, as I know from personal experience how much effort it took to organize a conference with about 400 delegates. This year we had roughly 1,050, and were close to the capacity of the venue.

This was followed by a short talk from Chris Hanger of White Oak Technologies, one of the conference's Diamond sponsors, entitled *Why Python Sucks (But Works Great for Us)*. Chris managed to give a light-hearted presentation but to make the point that Python has been a strategic advantage for White Oak that has allowed them to win business from larger competitors.

Next came Guido van Rossum, talking about the State of Python 3000. I won't report this as I reported it in some detail on my blog (which I have been neglecting lately – there's just been too much to catch up with since I got back from the conference). *http://holdenweb.blogspot.com/2008/03/pycon-friday-guidos-keynote.html*

After the break the only session I managed to get to was Brett Cannon's 'How Import Does Its Thing', which was a clear presentation of one of the lesser-known parts of the Python interpreter. I have spent some time myself jumping through the various hoops necessary to implement a PEP 302-style importer (in my case to import modules from a relational database), and it would have been really good to have had Brett's advice available then. The good news is that with luck the import mechanism for some future version of Python will be implemented solely in Python. Brett has a working model, but he doesn't feel it's yet fast enough to replace the existing mechanisms. I should have liked to stay and see all of Tim Coupar's talk on Python references, which looked to have some neat animations in it, but alas I had a meeting to get ready for.

And so to lunch, and the annual PSF Members' meeting. Again we saw evidence of David Goodger's dedication, and this was the best-organized Members' meeting I can remember. The voting was orderly, and all formal business was out of the way in well under an hour, during which we elected two new directors (Raymond Hettinger and James Tauber) and eight new members (Titus Brown, Mark Dickinson, Amaury Forgeot d'Arc, Christian Heimes, Van Lindberg , David Mertz, John Pinner and James Tauber). This gave us time for discussions of various topics of interest including the sensitive subject of Python certification, something which I would like to address in the next year if I can. That particular topic has gone quiet in the post-conference rush, but I hope to be taking it up again soon.

In the afternoon I put in my time as a session chair, and got to hear some really interesting talks. Jonathan Ellis started us off with a talk about a Petabyte file system, and gave a diverting description of some of the problems you might not anticipate (such as the near-certainty of fairly regular disk failures). This was followed by Kevin Dangoor talking about how to build TurboGears applications with rich Web user interfaces using Dojo to build smarter client communications. Next Adrian Holovaty gave a summary of the status of Django, holding out hopes that a 1.0 release would not be too far away. At the end of his presentation he was joined by Jacob Kaplan-Moss and together they announced the formation of the Django Software Foundation. The afternoon was rounded out by Marty Alchin's talk on *Django: Under the Hood*. Apparently this was Marty's first speaking engagement, but you wouldn't have known it from the confidence of his presentation.

I then listened to a few of the lightning talks (more about them later) and spent more pleasant time in conversation with conference acquaintances old and new in the hallways, eventually leaving for the traditional PSF Board Meeting that follows the Members' meeting as soon as possible, and whose principal business is to elect the officers. Stephan Deibel, who has been the Foundation's chairman for four years, had already announced his intention to stand down, so we knew we had to elect a new chairman. Unfortunately, I failed to take the requisite backwards pace, and was elected the new chairman. I am honored by my Board colleagues' confidence in me (and annoyed at their superior ability to get out from under).

This meeting was followed by a trip to a nearby restaurant for dinner. Since the Board members and officers don't get much chance to meet face to face, it's a good opportunity to get to know each other a little better and exchange views when there isn't necessarily any pressing business to discuss. We had a very pleasant meal, and Brett Cannon polished off a steak that must have weighed twenty ounces, then followed that with a large dessert (which, to his credit, he generously shared around).

On the way out of the restaurant I fell in with another group of PyCon delegates who had come to the bar for a drink, and this gave me a chance to renew my acquaintance with several more old friends, including a rather fine single malt Scotch whisky. I finally got to bed after midnight, and nothing could have kept me awake.

" I finished the evening in the hotel bar, where I am frequently to be found during conferences."

Saturday

The day began with an announcement from the Twisted team, who were once more present at PyCon in force, about the creation of the Twisted Software Foundation. The catchphrase of the moment became 'Feel the lerve'. This was followed by the second Diamond keynote, another pleasant talk this time by Brian Fitzpatrick, the head of Google's local office. To bring us up to the break Van Lindberg, the PSF's attorney and also a software engineer, explained the relationship between *Intellectual Property and Open Source*. I thought Van explained how licensing of open source products addressed various problems of inequity rather well, and felt my time had been well spent.

After the break there were too many people to catch up with to follow my plans and attend the sessions I had planned, so I missed the one I had most been hoping to see, which was Alex Martelli's *Don't Call Us, We'll Call You* about the uses of callbacks in Python. Fortunately Alex, being the consummate professional, has uploaded his slides to the conference Web site. There were also some rumblings about dissatisfaction with some of the sessions, the sponsor lightning talks in particular, so I took some time to discuss the issues with the organizers involved so I could respond appropriately if it turned out to be immediately necessary (which it wasn't).

After lunch I took in Richard Jones' talk on *Sights and Sounds with piglet*. It always amazes me to see a Python program provide such direct control over accelerated graphics, and Richard had some wry insights into how to achieve cross-platform compatibility. Every time I see piglet it has become more capable, and it is already a truly awesome piece of software.

End-User Computing without Tears: Resolver, an IronPython Spreadsheet was to have been given by Giles Thomas, who unfortunately injured his back the day before he was due to travel. Michael Foord and Jonathan Hartley did a praiseworthy job of standing in.

I watched some more lightning talks and dined in the bar, again catching up with many old acquaintances, and before I knew it it was time for my Open Space session, *Teach Me Twisted*. Open Space is a special part of PyCon (and many other conferences too, nowadays) where delegates can claim a space to do pretty much whatever they like. This year there were about ten rooms available, far more than had been in any previous year, and particularly during the evenings they were well-used.

The idea behind my session was to have the Twisted representatives explain Twisted to me in front of an audience who also wanted to learn the topic. I decided that just to emphasize that the presentation wasn't entirely serious I would take a bottle of Scotch with me, so I could take a nip if anything seemed particularly worthy of celebration (or caused me mental anguish). The theory was that if the bloke at the front of the room, normally expected to be an 'authority', is comfortable confessing his ignorance then the other learners in the room will be much less inhibited about asking their own questions.

This seemed to work out well despite my rather aged laptop bluescreening twenty minutes into the presentation. I am deeply grateful to Cliff Dyer for immediately offering me his own machine (complete with windows containing everything I had already typed) as a replacement. I think I was also able to keep the Twisted team 'on message' by simplifying or paraphrasing some of the things that they said, and occasionally arguing that their objections to my simplifications (while strictly valid) would not assist the learners in the room. I had great fun, and can honestly say I have never enjoyed being part of a conference presentation more, so I hope we can do something similar next year.

At the end of the two hours I had built some modest but reasonable Twisted functionality, and understood a lot more about Twisted than I had before. I also handed out the new Holden Web t-shirts quite liberally. I asked for a show of hands as to who else had learned something about Twisted, and even Alex Martelli put his hand up! There were forty people still in the room and animatedly talking about Python twenty minutes after the session ended; that alone made it all worthwhile.

The buzz continued as we repaired once more to the bar, and when that closed at midnight we took ourselves off to the neighboring Holiday Inn, whose bar stayed open later. With one thing and another it was turned two o'clock by the time I got to bed.

Sunday

Rather to my own amazement I was up in time for a reasonably early breakfast and caught all but the very first plenary session. So I heard Ivan Krstic explain how he nearly died in Uruguay as he assisted with the deployment of the first OLPC hardware, and Mark Hammond talking about *Snake Charming the Dragon: the Past, Present and Future of Python and Mozilla*. After that I didn't attend any formal sessions as I was too busy catching up with people.

After lunch I gave a brief and unmemorable lightning talk to try to get people to contribute to *Python Magazine* and various blog projects of mine, and after a further short break we then went into the sprint preparations.

Brett Cannon gave a good introductory overview, and then I joined the Django sprint, which appeared to comprise between forty and fifty people. After that I joined Richard Jones, Sean Reifschneider and Evelyn Mitchell (all now long-time PyCon buddies) in traveling to Ian Bicking's house. He was bravely giving a party and entertaining many people from PyCon and elsewhere. As someone who spends a lot of time in hotels it was refreshing to be able to visit a 'real home' and get away from the concrete jungle for a few hours.

I left relatively early meaning to go to bed by eleven, but when Evelyn and I got back to the hotel we decided we would have a nightcap, and then fell in with the Django crowd, who kept me up (and amused) until heaven knows what time. In the end the party broke up because we had to help one member of the party who had imbibed quite well up to his room, he being incapable of making it under his own steam.

Monday

This had originally been intended to be the first of two sprint days for me. I awoke a little late, and decided I would take the hotel's formal breakfast, which gave me a chance to chat to Guido, though not about anything of much great consequence. I joined the Django sprint room to find that there were continual demands on my time from other directions, including (unfortunately) a call from my wife to tell me that she had fallen victim to the stomach flu that had been doing the rounds. Reluctantly I made arrangements to leave a day early, on the 9:51 pm flight.

I had a good meeting with Guido and Ted Leung, who had by then been working for Sun Microsystems for just less than three weeks, establishing a line of communications that I hope will work to everyone's benefit in future.

Back at the sprint I discovered that Cliff, my rescuer on Saturday night, was using a laptop from his company, as his own had just about given up the ghost. I used this as an excuse to get rid of bigboy, the aging laptop which had blue screened on me on the Saturday: I had bought a replacement two months before but had not got around to moving my information on to it. It just seemed the right thing to do to take the disk out and let bigboy go home with Cliff.

A flurry of goodbyes, and I left the hotel for the airport promising myself that next time would be even better. There is definitely something special about PyCon, and it's mostly down to the people.

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.

Want

YOUR

15 minutes

of

Python Magazine

wants to hear from you!

If you want to bring a Python-related topic to the community - be it your personal research, company software, or anything else the professional Python community might be interested in - why not write an article for Python Magazine?

Let us help you hone your idea into a beautiful article for Python Magazine!

Visit http://www.pythonmagazine.com or contact our editorial team at editors@pythonmagazine.com and get started!

Python Magazine

And now for something completely different

The first monthly magazine dedicated exclusively to Python.

Python Magazine
November 2007
VOLUME 1 - ISSUE 11
From the petri dish to global warming,
PYTHON MAKES WAVES IN SCIENTIFIC RESEARCH ENVIRONMENTS
Get Going with IDLE
Testing, Teaching, and Debugging with IDLE
Decorators on a Dime
Using Decorators and Metaclasses
Caching RSS Feeds with feedcache
Exploit RSS and ATOM feeds with Feedcache
Understanding GTKTreeView
Tame PyGTK's Elusive TreeView
INSIDE:
Welcome to Python Magazine!
Perspective on the Global Interpreter Lock Debate.
• Operator Overload! Learn how to change the behavior of equality operators.

COLUMN
Reading from the Web
Python Magazine
FEATURES
Creating custom PyGTK widgets with Cairo
Processing Web Forms Using Anonymous Functions & WSGI
Extending Python
COLUMNS
Download this month's code at:
WRITE FOR US!
FEATURE

Creating custom PyGTK widgets with Cairo

by Sayamindu Dasgupta

PyGTK, a set of Python bindings for the popular GTK+ graphical toolkit, provides a rich collection of commonly used windows, dialog boxes, buttons, layout elements, and other 'widgets'. However, often a programmer has needs which go beyond the functionality provided by the built-in widgets in PyGTK. This article explains how to create new widgets using the Python bindings for Cairo – the vector graphics library used by GTK+ to perform most of its drawing operations.

About GTK+

GTK+ is one of the most popular free/open source Graphical User Interface (GUI) toolkits around. Though best known as the basic building block of GNOME, a popular free/open source desktop, GTK+ was originally written for the GIMP image editing program (in fact, 'GTK' actually stands for 'Gimp Tool Kit'). Currently, apart from its role in the GNOME and GIMP projects, it is also used to create the GUIs for the XFCE and Rox desktops. In addition, it is also used in embedded devices such as the Nokia N800/N770 (as a part of the Hildon desktop), and the FIC Neo1973 (as a part of the OpenMoko framework).

REQUIREMENTS

PYTHON: 2.x

Other Software: PyGTK 2.10 or above

Useful/Related Links:
http://www.pygtk.org
http://www.cairographics.org

October 2007 - Python Magazine - 9

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