UKUUG

news@UK

The Newsletter of UKUUG, the UK's Unix and Open Systems Users Group
Published electronically at http://www.ukuug.org/newsletter/

Volume 18, Number 3 **ISSN 0965-9412** **September 2009**

Contents

From the Secretariat

Jane Morrison

UKUUG continues to work with O'Reilly, organising successful tutorials. The recently organised RT tutorial held in London on 11th August was very well attended. This tutorial was a repeat of that originally held in October 2007 and the number of attendees this time totalled 15.

Future tutorials already planned for 2009 include 'Advanced DNS Administration Using BIND9', by Jim Reid and three tutorials on Perl ('Beginning', 'Intermediate' and 'Advanced') by Dave Cross, on 24th, 25th and 26th November. Details of both these events can be found as an insert with this Newsletter. At the time of writing we already have a number of bookings for each day and urge you to book you place without delay.

Josette Garcia (O'Reilly) has access to many authors who can also be of interest to our membership by providing tutorials etc. and we are working to bring you a series of tutorials for 2010.

OpenTech 2009, organised by ex-Council member Sam Smith, was again a great success with some 600 people attending. A write-up on the event appears elsewhere in this Newsletter.

This year's Summer Conference was held in Birmingham between the 7th and 9th August. This event was organised by Council members John Pinner and Niall Mansfield and I have heard that those who attended found it most enjoyable.

UKUUG is doing the organisational work for the upcoming EuroBSDCon being held in Cambridge from 18th – 20th September. For more information about the event see:
http://www.ukuug.org/events/eurobsdcon2009/

Looking further ahead to next year; enclosed with this Newsletter you will find the *Call For Papers* for the 2010 Spring Conference which will be held in Manchester from 23rd – 25th March 2010. Please put this event date in your diary now.

This year's annual general meeting will be held on 24th September in the Cruciform Building, Gower Street (part of University College, London, opposite the main gate). Full details including the agenda have recently been sent to all members, and can also be seen on our web site. The AGM starts at 6.15 p.m.

The next Newsletter will be the December issue and the copy date is 20th November. As usual any materials for publication should be sent to
newsletter@ukuug.org

From the Chairman

Paul Waring

Post-AGM Discussion

A number of active members have suggested that they would like to discuss how we take UKUUG forward from its present position. Based on this feedback, we will be holding a discussion session immediately after the AGM for approximately one hour (in place of the usual speaker slot), where we will be asking members to feed in any ideas they have as to what the future direction of UKUUG should be. Perhaps you think we should plough all of our efforts into one conference a year, or that we should engage in more lobbying, or maybe you're happy with everything and would like us to continue as normal. Whatever your ideas, we would like to encourage as many people as possible to come along and take part.

If you cannot attend the AGM but would like to input into this process, please let us know so that we can ensure that all views are taken into account.

Social Networks

UKUUG has been using Facebook and LinkedIn for networking and promoting events for some time, and we've now ventured out into the world of Twitter, where we post short updates announcing forthcoming events, reminders about registration deadlines etc.

You can find us at:
http://www.ukuug.org/facebook
http://www.ukuug.org/linkedin
http://www.ukuug.org/twitter

Get Involved

UKUUG exists to serve its members and we are always on the lookout for people who are keen to get involved in any capacity, whether that be through volunteering to help with organising events, writing newsletter articles, or entirely new activities which haven't been tried before. We are particularly keen to recruit a new member to Council this year, so if you feel you have the time to spare or would like to know more about what this involves, please get in touch and I will be happy to answer any questions you may have.

Summer Conference 2009

Roger Whittaker

UKUUG's Summer Conference this year was held over the second weekend in August in the civilised surroundings of the Birmingham Conservatoire.

As usual, the day prior to the main conference was given over to tutorials. On this occasion there was a choice of five tutorials. Peter Brownell and Robert Castelo of Code Positive Ltd gave a tutorial on Drupal; Andrew Eliasz offered a one-day course on the Arduino open-source electronics prototyping platform. Jonathan Fine and Alun Moon ran a LaTeX tutorial, while Neil Woolford gave an intensive session on the GIMP, and Quentin Wright offered a hands-on tutorial on VOIP using Asterisk.

I attended the LaTeX tutorial, which was a very good opportunity to pick the brains of two experts in the field and straighten out some confusions and fill some of the many gaps in my knowledge.

On the Friday evening an ad-hoc meal took place at a Moroccan restaurant very near the venue, with most of those who had taken part in the tutorials attending as well as many who had arrived on Friday afternoon to take part in the conference proper the next day.

During the two days of the conference itself, there were two parallel tracks at most times. Highlights on Saturday morning included the sessions by Michael Brunton-Spall and Paul Nasrat of the Guardian, first describing the Guardian's web presence in general and the open APIs that have been provided to allow reuse of content under relatively generous conditions, and then the detailed and entertaining story of how the Guardian was able to react quickly to the recent parliamentary expenses scandal by creating a dedicated site allowing users to examine, investigate and tag particular expense claims as being of interest.

The first session on Saturday afternoon was a keynote by Ross Anderson of Cambridge University entitled *Why security engineering will just keep on getting harder.* This talk was both entertaining and informative, ranging quite widely over recent security concerns, but with an emphasis on the security implications of the explosion in social networks. In this connection, Anderson discussed the implications of Gladman's principle: *"You can have security, or functionality, or scale. With good engineering you can have any two of these. But there's no way you can get all three."*

Saturday's afternoon's talks include Darren Moffat of Sun on ZFS, and Michael Meeks of Novell on the new Moblin netbook GUI.

The conference dinner took place on Saturday evening and was a good opportunity to continue the day's discussions and make new contacts in convivial surroundings.

Sunday morning's talks included John Collins and John Pinner on GNUspool (an advanced print management system which has recently been accepted as an official GNU project), and Howard Thompson on Eiffel.

Chris Proctor gave a highly entertaining talk on LVM, which was an update to the talk he gave at the

Spring Conference in London, based on his experiences (some of them slightly scary) dealing with large volumes in real-life situations.

The afternoon's talks included David Greaves' talk on Mer – a version of Linux for hand-held touch screen devices.

The conference concluded with a plenary session and several interesting lightning talks.

O'Reilly, Sun, Novell and ORG were represented by table-top stands in the foyer area, where coffee and biscuits were served during the breaks, and where occasional assorted musicians could be seen wandering to and from the practise rooms, not too surprised to find their college once again invaded by geeky conference delegates (this was the fourth event that I have attended at the venue).

Open Tech 2009

Roger Whittaker

This year's Open Tech event was held at the University of London Union on 4th July 2009. More than 500 people attended this extraordinary one-day conference, which was characterised by a strong feeling of engagement and "buzz" throughout the day.

Organised by UKUUG in conjunction with a wide range of groups and individuals, and with sponsorship from 4iP, this was a day mainly concerned with the social, political and cultural uses and effects of internet technologies, and the legal and other threats to on-line freedom.

The conference was divided into three tracks throughout the day (four at some points), and the rooms in use were full to overflowing most of the time.

During the two morning sessions I acted as a session chair in one of the smaller rooms, where there were three talks per hour. During these sessions I was particularly impressed by Charles Armstrong's talk on "One Click Orgs" in which he presented his open source web-based system for creating an organisation as a legal entity, and running its membership and voting system on-line.

Paul Downey gave a thought-provoking talk with the title "Standards are to Peace as Standardisation is to War", and Frances Davey spoke on the problems caused by current law as it applies to the Internet. Also among the morning sessions that I witnessed was a session by Richard Elen on setting up a virtual drama group and producing plays across the Internet using such technologies as Skype and Second Life, and a session by Hamish Cambell of VisionOn.tv about this open video project which uses a mixture of peer-caching and direct download for delivery. Steve Goodwin presented an interesting and amusing session on "digital archaeology of the microcomputer", describing some of the work that he has done on trying to preserve some of the legacy of the earliest home computer systems.

The first afternoon session in the main hall was one that I heard several people refer to in advance as the "Bill and Ben show": a very popular and entertaining session by Bill Thompson and Ben Goldacre. Bill Thompson's title was "10 cultures" a nice conflation of CP Snow's "two cultures" and the well known slogan *"there are 10 kinds of people in the world – those who understand binary and those who don't"*. His talk was a strong plea for a society with a better understanding and awareness of how digital technologies are created and how they work. He believed that a failure to achieve this was leading to a situation in which seriously bad political decisions were being made and in which the those with the requisite knowledge were able to exploit those without it. He made a particular mention of the decisions being made about ID cards by people without the means to understand the technological issues that they raise. He also strongly attacked the state of the so-called ICT curriculum in schools, and called for a return to a form of computer studies that actually encourages an understanding of how things work.

Ben Goldacre is best known for his "Bad Science" column in the Guardian and the associated blog. He gave some examples of recent "science" reports that have appeared across the UK press, despite having had virtually no factual content. He explained some of the processes whereby such stories get into the press, the most common one being press releases from companies containing some "quirky" or startling story,

the main purpose being to gain publicity for the company. One of the most shocking cases he mentioned was the way in which the claims of a company with a purported cure for dyslexia had been uncritically presented as fact by large trusted media organisations including the BBC. He applauded the work done by various bloggers in exposing and collating bogus scientific claims, and particularly mentioned the area of "alternative health practitioners", where some false scientific claims made by such people have been removed following their exposure. However, in the same context he mentioned the recent libel case involving Simon Singh which shows that there is a real threat to the right to speak freely about scientific matters. He hoped that some kind of "semantic web" type aggregation of information from multiple sources could be used to help counter bad science by collecting detailed information and evidence in one place. In the question and answer sessions after both these talks, there was some lively discussion from the floor.

For the second session of the afternoon, I attended the session by Rob McKinnon and Richard Pope entitled "Web of Power". They looked at some of the existing tools developed by MySociety and others that can track and collate information about the activities of MPs and political parties. Looking at a diagram showing overlapping circles of power taken from Anthony Sampson's book "Who runs this place", the speakers suggested some other areas where web based tools might provide similar useful scrutiny. In some cases there are problems with this: for example, the Government's published list of "approved suppliers" does not include company registration numbers, while trying to cross-reference Company House data is handicapped by the £1 fee for getting director information. There was considerable discussion about the ways in which automatically collating information about MPs, ministers, lobbyists and company directors could be useful in exposing corruption and investigating the "web of power".

For the penultimate slot I went back to the main hall where there were three talks, on "Ephemerality" by Gavin Bell, who discussed the extent to which web information is persistent, and whether deliberate "forgetting" can sometimes be useful. He also expanded on the uses of time sensitive display of dated information. Gary Gale then gave a talk on "Location, Privacy and opting out". With the proliferation of location-based services, he explained how the need for ways to protect privacy is becoming more important. In particular he called for clearer and more comprehensible policies on opt-outs and the right to "hide" yourself from such services. Gavin Starks of AMEE then spoke on the subject of "Energy Identity". AMEE is an open platform which aims to measure the energy consumption and carbon footprint of every possible item or activity. By looking aggregating all data about an individual or organisation's purchases, materials, buildings, travel, fuel and water usage, it can build up a unique "energy identity" or profile.

For the final session of the day I attended the joint session from NO2ID and the Open Rights Group. This featured a spoof appearance by a senior civil servant "Sir Bonar Neville Kingdom", defending the Government's plans for data interception. There then followed two very serious presentations by Jim Killick of ORG and Guy Herbert of NO2ID. Jim Killick described the Government's "Intercept Modernisation Plan" and its aim of monitoring all communications using deep packet inspection carried out by "black box" devices on every ISP's network. Guy Herbert spoke on the related theme of the Government's aim of "mastering the internet" and urged the audience to comment on the Government's consultation document "Protecting the Public in a changing Communications Environment" and on the current review of the RIPA act. I for one was sufficiently impressed and worried by these presentations that I joined ORG on the spot before leaving the room.

Overall, this was an exceptional event, for which much credit must go to Sam Smith for his part in its organisation. The standard of all the talks was very high, and the formal and informal discussions in, after and between them were interesting and thought provoking.

Python: an untapped resource in system administration

Corey Brune

Historically, system administrators chose Perl as their preferred scripting language. However, the functionality of Python may surprise those not familiar with the language. I hope to illustrate the benefits of Python within system administration while highlighting script functionality within a password-file-to-LDAP conversion script.

Python is an ideal language for beginning and expert programmers. Organisations such as Google, the New York Stock Exchange, and NASA have benefited from Python. Furthermore, Python is used behind Red Hat menu systems as well as BitTorrent clients. As a system administrator, I find Python to be an exciting and rewarding open source language. Many first-time users are surprised at the speed at which the code falls into place when beginning to program. However, it is in the large and demanding projects that you will find Python most beneficial. This is where you will find increased manageability and time savings as opposed to other languages. Not only does Python aid in rapid deployment, but its functionality, ease of use, portability, and dependability result in hassle-free administration. Furthermore, it is compatible with all standard operating systems.

Python's ease of use is achieved primarily in the language's maintainability and elimination of code redundancy. Elimination of code redundancy is important, since duplicating code consumes time and resources. In regard to maintainability, you will notice that reading scripts is made easy. For example, Perl can be time-consuming and difficult to maintain, primarily with large programs. You want a script that can be easily read and supported, especially when modifying past or unfamiliar programs. Syntax is easy to use, read, and understand. This is primarily achieved with the language's straightforward code and similarity to the English language. Since it is object-oriented, it lends itself to easily creating modules and reducing code duplication. Python also supports Functional Programming (FP), leaving it up to the programmer to use Object Oriented Programming (OOP) or FP. Furthermore, a user can easily reuse previously created code, and specific modules can be tested rather than the entire program.

My goal in highlighting this script is to illustrate the ease in which the code falls together while spotlighting process interactions. Furthermore, I hope to demonstrate code simplicity while defining methodology.

Password-to-LDAP Conversion Script

You can find the entire listing for this script online [1]. In this article, I will just cover the highlights of the script as a way of describing Python syntax. I will also show how modules make it easy to perform system administration tasks.

Similarly to Perl and Java, Python's extensive library contains built-in modules that may be used to simplify and reduce development time. These include `import pwd`, `sys`, `os`, `csv`, and `subprocess`.

Note that Python statements do not end in a terminating semicolon; rather, the terminator is the end of the logical line itself.

The `def` keyword is used to declare a method:

```
def main(argv):
```

The `main` declaration accepts a list named `argv`. `argv` is similar to Perl's `@ARGV` or C language's `**argv` and contains the command-line arguments passed to the program.

Exception handling is how errors are managed within a program. For example, consider the following:

```
try:
  <code>

except IOError, (errno, strerror):
  sys.exit('I/O error (%s): %s' % (errno, strerror))

except ValueError:
  sys.exit('Could not find x in string %s: %s' % (pwLine, sys.exc_info()[0]))
```

When an exception is thrown, the programmer decides whether the script will exit, call another method, or prompt for user action. Multiple exceptions may be nested in a `try` block. The `sys.exc_info()` method returns the last exception caught.

Parsing Comma Separated Value (csv) files is simplified with the `csv` module. Instead of using methods such as `split()` to parse these files, the `csv.reader()` or `csv.writer()` methods allow for a standard mechanism for reading or writing csv files:

```
fd = csv.reader(open('shadow','r'), delimiter=':')
```

The delimiter argument allows reading or writing different csv files. Notice that the results of `open()` are used as the argument to `csv.reader()`. This syntax is common usage throughout Python. Here is an example of file IO with Python:

```
ldifOut = open(ldifFile, 'w+')
ldifOut.close()
```

The first argument is the filename, and the second argument is the mode. There are different modes available depending on the type of operation required: read (`r`), read-append (`r+`), write (`w`), write-append (`w+`), or append (`a+`). The return value is a file object assigned to the variable `ldifOut`. The `close()` method closes the file object.

Lists are one of the most dynamic data types in Python. Open and close brackets with comma-delimited values declare a list. The example here declares an empty list, `pwLine`:

```
pwLine = []
```

Lists may be indexed, split, and searched. Furthermore, they may be used as a stack or queue and may contain other data types. In the following code:

```
for row in fd:
  if row[1] in '*LK*' 'NP' '*' '!!':
    continue
```

the `for` loop iterates through the file object `fd` until the end of file. Conditionals and loops are terminated with a colon. Unlike other languages, loops, conditionals, and other statements are "grouped by using indentation" (`python.org`). The `if` statement says if `row[1]` matches `*LK*`, `NP`, `**`, or `!!`, to continue to the next line in the file, since these are local accounts such as `root` or `nobody`. Python contains many of the C standard UNIX/Linux system functions and usage is similar to C functions. The `pwd.getpwnam()` method generates the list of users to be converted to the LDAP script:

```
String = pwd.getpwnam(line[0])
```

The argument passed is `line[0]`, which is the username.

The `pwd.getpwnam()` method returns a tuple. Tuples, like strings, are read-only or immutable data types. To modify the tuple, we convert the tuple to a list:

```
pwLine = list(String)
```

Python offers many types of conversion methods, such as `int()`, `float()`, and `str()`.

The code:

```
index = pwLine.index('x')
pwLine.pop(index)
pwLine.insert(index, line[1])
```

illustrates some of the methods available for list manipulation. `pwLine.index('x')` returns an integer value to where the value was found. If the value is not found, a `ValueError` exception is thrown. `pwLine.pop(index)` removes and returns the value at `index`. `pwLine.insert(index, line[1])` inserts the encrypted password (`line[1]`) into the list at `index`.

Using the `write()` method allows for updating or writing files:

```
ldifOut.write('dn: cn=' + pwLine[0] + ',' + fullDN + '\n')
```

The arguments to `write()` illustrate how to use string concatenation with the + sign. You can access individual elements in a list or string by using brackets `[]`. In the example here, `pwLine[0]` accesses the first data element. Note that lists and strings start at index number 0.

The subprocess module is the preferred mechanism when interacting with processes:

```
output = subprocess.Popen(ldapStr, shell=True, stdout=subprocess.PIPE)
output.wait()
stdout_value = output.communicate()[0]
```

`subprocess.Popen()` is used to invoke `ldapadd` and the associated arguments. `shell = True` indicates that the command will be passed to the shell; otherwise `os.execvp()` is executed.

`stdout=subproces.PIPE` contains a pipe to control the output. Other pipes may be created for `stderr` and `stdin`. The variable output is assigned a `Popen` object. `wait()` is called to allow for the process to finish. Process output is then retrieved with the `communicate()[0]` method.

Every module or method has many convenient built-in methods. These are denoted by underscores on either side of the name, for example:

```
if __name__ == '__main__':
  main(sys.argv[1:])
```

The build-in method __name__ defines the module's name. The next line passes `sys.argv[1:]` to `main()`. List elements may be accessed by `listname[<start index>:<end index>]`. In this example, the list `sys.argv[1:]` will pass elements starting at index 1 through the last element.

General Notes

Python uses indentation for code blocks, such as loops and conditionals, rather than semicolons, braces, or parentheses. Statements do not require semicolons for termination. Python contains built-in data types referred to as numbers, strings, lists, tuples, and dictionaries. These data types are represented by characters such as parentheses, brackets, and braces. Every data type is an object and has associated methods for manipulation. File parsing is made simple in Python with the module re. If you are familiar with Perl, you will notice that Regular Expression Syntax is similar. Python has the capability to handle large files such as XML, CSV, binary, and text files.

Conclusion

Although I have only skimmed the surface of Python's functionality and syntax, I hope to have provided a foundation for further exploration. The application range for Python crosses over many domains such as system administration, game programming, Web programming, and research and development. The extensive library and maintainability of code make this a versatile language. Examples of functionality are highlighted in the inter action of processes, file parsing, and exceptions. If you have dabbled with Python in the past, this is an opportunity to revisit the language. Soon after programming in Python, you may find its range spreading into all aspects of administration. For further information and resources visit **www.python.org**

Reference

[1]
http://www.sim10tech.com/code/python/passwd2ldap.py

Originally published in ***;login: The USENIX Magazine,*** *vol. 34, no. 1 (Berkeley, CA: USENIX Association, 2009). Reprinted by kind permission.*

Python 3: the good, the bad, and the ugly

David Beazley

In late 2008, and with much fanfare, Python 3.0 was released into the wild. Although there have been ongoing releases of Python in the past, Python 3 is notable in that it intentionally breaks backwards compatibility with all previous versions. If you use Python, you have undoubtedly heard that Python 3 even breaks the lowly print statement – rendering the most simple "Hello World" program incompatible. And there are many more changes, with some key differences that no conversion program can deal with. In this article I give you a taste of what's changed, outline where those changes are important, and provide you with guidance on whether you want or need to move to Python 3 soon.

By the time you're reading this, numerous articles covering all of the new features of Python 3 will have appeared. It is not my intent to simply rehash all of that material here. In fact, if you're interested in an exhaustive coverage of changes, you should consult "What's New in Python 3?" [1]. Rather, I'm hoping to go a little deeper and to explain why Python has been changed in the way it has, the implications for end users, and why you should care about it. This article is not meant to be a Python tutorial; a basic knowledge of Python programming is assumed.

Python's C Programming Roots

The lack of type declarations and curly braces aside, C is one of the foremost influences on Python's basic design, including the fundamental operators, identifiers, and keywords. The interpreter is written in C and even the special names such as `__init__`, `__str__`, and `__dict__` are inspired by a similar convention in the C preprocessor (for example, preprocessor macros such as `__FILE__` and `__LINE__`). The influence of C is no accident – Python was originally envisioned as a highlevel language for writing system administration tools. The goal was to have a high-level language that was easy to use and that sat somewhere between C programming and the shell.

Although Python has evolved greatly since its early days, a number of C-like features have remained in the language and libraries. For example, the integer math operators are taken straight from C – even truncating division just like C:

```
>>> 7/4
1
>>>
```

Python's string formatting is modeled after the C `printf()` class of functions. For example:

```
>>> print "%10s %10d %10.2f" % ('ACME',100,123.45)
ACME    100      123.45
>>>
```

File I/O in Python is byte-oriented and really just a thin layer over the C stdio functionality:

```
>>> f = open("data.txt","r")
>>> data = f.read(100)
>>> f.tell()
100L
>>> f.seek(500)
>>>
```

In addition, many of Python's oldest library modules provide direct access to low-level system functions that you would commonly use in C systems programs. For example, the os module provides almost all of the POSIX functions and other libraries provide low-level access to sockets, signals, fcntl, terminal I/O, memory mapped files, and so forth.

These aspects of Python have made it an extremely useful tool for writing all sorts of system-oriented tools and utilities, the purpose for which it was originally created and the way in which it is still used by a lot of system programmers and sysadmins. However, Python's C-like behaviour has not been without its fair share of problems. For example, the truncation of integer division is a widely acknowledged source

of unintended mathematical errors and the use of byte-oriented file I/O is a frequent source of confusion when working with Unicode in Internet applications.

Python 3: Breaking Free of Its Past

One of the most noticeable changes in Python 3 is a major shift away from its original roots in C and UNIX programming. Although the interpreter is still written in C, Python 3 fixes a variety of subtle design problems associated with its original implementation. For example, in Python 3, integer division now yields a floating point number:

```
>>> 7/4
1.75
>>>
```

A number of fundamental statements in the language have been changed into library functions. For example, `print` and `exec` are now just ordinary function calls. Thus, the familiar "Hello World" program becomes this:

```
print("Hello World")
```

Python 3 fully embraces Unicode text, a change that affects almost every part of the language and library. For instance, all text strings are now Unicode, as is Python source code. When you open files in text mode, Unicode is always assumed – even if you don't specify anything in the way of a specific encoding (with UTF-8 being the usual default). I'll discuss the implications of this change a little later – it's not as seamless as one might imagine.

Borrowing from Java, Python 3 takes a completely different approach to file I/O. Although you still open files using the familiar `open()` function, the kind of "file" object that you get back is now part of a layered I/O stack. For example:

```
>>> f = open("foo")
>>> f
<io.TextIOWrapper object at 0x383950>
>>>
```

So, what is this `TextIOWrapper`? It's a class that wraps a file of type `BufferedReader`, which in turns wraps a file of type `FileIO`. Yes, Python 3 has a full assortment of various I/O classes for raw I/O, buffered I/O, and text decoding that get hooked together in various configurations. Although it's not a carbon copy of what you find in Java, it has a similar flavour.

Borrowing from the .NET framework, Python 3 adopts a completely different approach to string formatting based on composite format strings. For example, here is the preferred way to format a string in Python 3:

```
print("{0:10} {1:10d} {2:10.2f}".format(name,shares,price))
```

For now, the old `printf`-style string formatting is still supported, but its future is in some doubt.

Although there are a variety of other more minor changes, experienced Python programmers coming into Python 3 may find the transition to be rather jarring. Although the core language feels about the same, the programming environment is very different from what you have used in the past. Instead of being grounded in C, Python 3 adopts many of its ideas from more modern programming languages.

Python's Evolution Into a Framework Language

If you're a current user of Python, you might have read the last section and wondered what the Python developers must be thinking. Breaking all backwards compatibility just to turn `print` into a function and make all string and I/O handling into a big Unicode playground seems like a rather extreme step to take, especially given that Python already has a rather useful print statement and fully capable support for Unicode. As it turns out, these changes aren't the main story of what Python 3 is all about. Let's review a bit of history.

Since its early days, Python has increasingly been used as a language for creating complex application frameworks and programming environments. Early adopters noticed that almost every aspect of the interpreter was exposed and that the language could be molded into a programming environment that was custom-tailored for specific application domains. Example frameworks include Web programming, scientific computing, image processing, and animation. The only problem was that even though Python was "good enough" to be used in this way, many of the tricks employed by framework builders pushed the language in directions that were never anticipated in its original design. In fact, there were a lot of subtle quirks, limitations, and inconsistencies. Not only that, there were completely new features that framework builders wanted – often inspired by features of other programming languages. So, as Python has evolved, it has gradually acquired a new personality.

Almost all of this development has occurred in plain sight, with new features progressively added with each release of the interpreter. Although many users have chosen to ignore this work, it all takes center stage in Python 3. In fact, the most significant aspect of Python 3 is that it sheds a huge number of deprecated features and programming idioms in order to lay the groundwork for a whole new class of advanced programming techniques. Simply stated, there are things that can be done in Python 3 that are not at all possible in previous versions. In the next few sections, I go into more detail about this.

Python Metaprogramming

Python has always had two basic elements for organizing programs: functions and classes. A function is a sequence of statements that operate on some passed arguments and return a result. For example:

```
def factorial(n):
  result = 1
  while n > 1:
    result *= n
    n -= 1
  return result
```

A class is a collection of functions called methods that operate on objects known as instances. Here is a sample class definition:

```
class Rectangle(object):
  def __init__(self,width,height):
    self.width = width
    self.height = height
  def area(self):
    return self.height*self.width
  def perimeter(self):
    return 2*self.height + 2*self.width
```

Most users of Python are familiar with the idea of using functions and classes to carry out various programming tasks. For example:

```
>>> print factorial(6)
720
>>> r = Rectangle(4,5)
>>> r.area()
20
>>> r.perimeter()
18
>>>
```

However, an often overlooked feature is that function and class definitions have first-class status. That is, when you define a function, you are creating a "function object" that can be passed around and manipulated just like a normal piece of data. Likewise, when you define a class, you are creating a "type object". The fact that functions and classes can be manipulated means that it is possible to write programs that carry out processing on their own internal structure. That is, you can write code that operates on function and class objects just as easily as you can write code that manipulates numbers or strings. Programming like

this is known as *metaprogramming*.

Metaprogramming with Decorators

A common metaprogramming example is the problem of creating various forms of function wrappers. This is typically done by writing a function that accepts another function as input and that dynamically creates a completely new function that wraps an extra layer of logic around it. For example, this function wraps another function with a debugging layer:

```
def debugged(func):
  def call(*args,**kwargs):
    print("Calling %s" % func.__name__)
    result = func(*args,**kwargs)
    print("%s returning %r" % (func.__name__, result))
    return result
  return call
```

To use this utility function, you typically apply it to an existing function and use the result as its replacement. An example will help illustrate:

```
>>> def add(x,y):
...   return x+y
...
>>> add(3,4)
7
>>> add = debugged(add)
>>> add(3,4)
Calling add
add returning 7
7
>>>
```

This wrapping process became so common in frameworks, that Python 2.4 introduced a new syntax for it known as a decorator. For example, if you want to define a function with an extra wrapper added to it, you can write this:

```
@debugged
def add(x,y):
  return x+y
```

The special syntax `@name` placed before a function or method definition specifies the name of a function that will process the function object created by the function definition that follows. The value returned by the decorator function takes the place of the original definition.

There are many possible uses of decorators, but one of their more interesting qualities is that they allow function definitions to be manipulated at the time they are defined. If you put extra logic into the wrapping process, you can have programs selectively turn features on or off, much in the way that a C programmer might use the preprocessor. For example, consider this slightly modified version of the `debugged()` function:

```
import os
def debugged(func):
  # If not in debugging mode, return func unmodified
  if os.environ.get('DEBUG','FALSE') != 'TRUE':
    return func
  # Put a debugging wrapper around func
  def call(*args,**kwargs):
    print("Calling %s" % func.__name__)
    result = func(*args,**kwargs)
    print("%s returning %r" % (func.__name__, result))
    return result
  return call
```

In this modified version, the `debugged()` function looks at the setting of an environment variable and uses that to determine whether or not to put a debugging wrapper around a function. If debugging is turned

off, the function is simply left alone. In that case, the use of a decorator has no effect and the program runs at full speed with no extra overhead (except for the one-time call to `debugged()` when decorated functions are *defined*).

Python 3 takes the idea of function decoration to a whole new level of sophistication. Let's look at an example:

```
def positive(x):
  "must be positive"
  return x > 0
def negative(x):
  "must be negative"
  return x < 0
def foo(a:positive, b:negative) -> positive:
  return a - b
```

The first two functions, `negative()` and `positive()`, are just simple function definitions that check an input value `x` to see if it satisfies a condition and return a Boolean result. However, something very different must be going on in the definition of `foo()` that follows.

The syntax of `foo()` involves a new feature of Python 3 known as a function annotation. A function annotation is a mechanism for associating arbitrary values with the arguments and return of a function definition. If you look carefully at this code, you might get the impression that the positive and negative annotations to `foo()` are carrying out some kind of magic – maybe calling those functions to enforce some kind of contract or assertion. However, you are wrong. In fact, these annotations do absolutely nothing! `foo()` is just like any other Python function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
-7
>>>
```

Python 3 doesn't do anything with annotations other than store them in a dictionary. Here is how to view it:

```
>>> foo.__annotations__
{'a': <function positive at 0x384468>,
 'b': <function negative at 0x3844b0>,
 'return': <function positive at 0x384468>}
>>>
```

The interpretation and use of these annotations are left entirely unspecified. However, their real power comes into play when you mix them with decorators. For example, here is a decorator that looks at the annotations and creates a wrapper function where they turn into assertions:

```
def ensure(func):
  # Extract annotation data
  return_check = func.__annotations__.get('return',None)
  arg_checks = [(name,func.__annotations__.get(name))
                for name in func.__code__.co_varnames]
  # Create a wrapper that checks argument values and the return
  # result using the functions specified in annotations
  def assert_call(*args,**kwargs):
    for (name,check),value in zip(arg_checks,args):
      if check: assert check(value), "%s %s" % (name, check.__doc__)
    for name,check in arg_checks[len(args):]:
      if check: assert check(kwargs[name]), "%s %s" % (name, check.__ doc__)
    result = func(*args,**kwargs)
    assert return_check(result), "return %s" % return_check.__doc__
    return result
  return assert_call
```

This code will undoubtedly require some study, but here is how it is used in a program:

```
@ensure
def foo(a:positive, b:negative) -> positive:
  return a - b
```

Here is an example of what happens if you violate any of the conditions when calling the decorated function:

```
>>> foo(3,-2)
5
>>> foo(-5,2)
Traceback (most recent call last):
  File "", line 1, in
  File "meta.py", line 19, in call
    def assert_call(*args,**kwargs):
AssertionError: a must be positive
>>>
```

It's really important to stress that everything in this example is user-defined. Annotations can be used in any manner whatsoever – the behaviour is left up to the application. In this example, we built our own support for a kind of "contract" programming where conditions can be optionally placed on function inputs. However, other possible applications might include type checking, performance optimization, data serialization, documentation, and more.

Metaclasses

The other major tool for metaprogramming is Python's support for metaclasses. When a class definition is encountered, the body of the class statement (all of the methods) populates a dictionary that later becomes part of the class object that is created. A metaclass allows programmers to insert their own custom processing into the last step of the low-level class creation process. The following example illustrates the mechanics of the "metaclass hook". You start by defining a so-called metaclass that inherits from type and implements a special method `__new__()`:

```
class mymeta(type):
  def __new__(cls,name,bases,dict):
    print("Creating class :", name)
    print("Base classes :", bases)
    print("Class body :", dict)
    # Create the actual class object
    return type.__new__(cls,name,bases,dict)
```

Next, when defining new classes, you can add a special "metaclass" specifier like this:

```
class Rectangle(object,metaclass=mymeta):
  def __init__(self,width,height):
    self.width = width
    self.height = height
  def area(self):
    return self.height*self.width
  def perimeter(self):
    return 2*self.height + 2*self.width
```

If you try this code, you will see the `__new__()` method of the metaclass execute once when the `Rectangle` class is *defined*. The arguments to this method contain all of the information about the class including the name, base classes, and dictionary of methods. I would strongly suggest trying this code to get a sense for what happens.

At this point, you might be asking yourself, "How would I use a feature like this?" The main power of a metaclass is that it can be used to manipulate the entire contents of class body in clever ways. For example, suppose that you collectively wanted to put a debugging wrapper around every single method of a class. Instead of manually decorating every single method, you could define a metaclass to do it like this:

```
class debugmeta(type):
  def __new__(cls,name,bases,dict):
```

```
    if os.environ.get('DEBUG','FALSE') == 'TRUE':
      # Find all callable class members and put a
      # debugging wrapper around them.
      for key,member in dict.items():
        if hasattr(member,'__call__'):
          dict[key] = debugged(member)
    return type.__new__(cls,name,bases,dict)

class Rectangle(object,metaclass=debugmeta):
...
```

In this case, the metaclass iterates through the entire class dictionary and rewrites its contents (wrapping all of the function calls with an extra layer).

Metaclasses have actually been part of Python since version 2.2. However, Python 3 expands their capabilities in an entirely new direction. In past versions, a metaclass could only be used to process a class definition after the entire class body had been executed. In other words, the entire body of the class would first execute and then the metaclass processing code would run to look at the resulting dictionary. Python 3 adds the ability to carry out processing *before* any part of the class body is processed and to incrementally perform work as each method is defined. Here is an example that shows some new metaclass features of Python 3 by detecting duplicate method names in a class definition:

```
# A special dictionary that detects duplicates
class dupdict(dict):
  def __setitem__(self,name,value):
    if name in self:
      raise TypeError("%s already defined" % name)
    return dict.__setitem__(self,name,value)

# A metaclass that detects duplicates
class dupmeta(type):
  @classmethod
  def __prepare__(cls,name,bases):
    return dupdict()
```

In this example, the `__prepare__()` method of the metaclass is a special method that runs at the very beginning of the class definition. As input it receives the name of the class being defined and a tuple of base classes. It returns the dictionary object that will be used to store members of the class body. If you return a custom dictionary, you can capture each member of a class as it is defined. For example, the `dupdict` class redefines item assignment so that, if any duplicate is defined, an exception is immediately raised. To see this metaclass in action, try it with this code:

```
class Rectangle(metaclass=dupmeta):
  def __init__(self,width,height):
    self.width = self.width
    self.height = self.height
  def area(self):
    return self.width*self.height
  # This repeated method name will be rejected (a bug)
  def area(self):
    return 2*(self.width+self.height)
```

Finally, just to push all of these ideas a little bit further, Python 3 allows class definitions to be decorated. For example:

```
@foo class Bar:
  statements
```

This syntax is shorthand for the following code:

```
class Bar:
  statements

Bar = foo(Bar)
```

So, just like functions, it is possible to use decorators to put wrappers around classes. Some possible use cases include applications related to distributed computing and components. For example, decorators could be used to create proxies, set up RPC servers, register classes with name mappers, and so forth.

Head explosion

If you read the last few sections and feel as though your brain is going to explode, then your understanding is probably correct. (Just for the record, metaclasses are also known as Python's "killer joke" – in reference to a Monty Python sketch that obviously can't be repeated here.) However, these new metaprogramming features are what really sets Python 3 apart from its predecessors, because these are the new parts of the language that can't be emulated in previous versions. They are also the parts of Python 3 that pave the way to entirely new types of framework development.

Python 3: The Good

On the whole, the best feature of Python 3 is that the entire language is more logically consistent, entirely customizable, and filled with advanced features that provide an almost mind-boggling amount of power to framework builders. There are fewer corner cases in the language design and a lot of warty features from past versions have been removed. For example, there are no "old-style" classes, string exceptions, or features that have plagued Python developers since its very beginning, but which could not be removed because of backwards compatibility concerns.

There are also a variety of new language features that are simply nice to use. For example, if you have used features such as list comprehensions, you know that they are a very powerful way to process data. Python 3 builds upon this and adds support for set and dictionary comprehensions. For example, here is an example of converting all keys of a dictionary to lowercase:

```
>>> data = {'NAME': 'Dave', 'EMAIL':'dave@dabeaz.com'}
>>> data = {k.lower():v for k,v in data.items}
>>> data
{'name': 'Dave', 'email': 'dave@dabeaz.com'}
>>>
```

Major parts of the standard library – especially those related to network programming – have been re-organized and cleaned up. For instance, instead of a half-dozen scattered modules related to the HTTP protocol, all of that functionality has been collected into an HTTP package. The bottom line is that, as a programming language, Python 3 is very clean, very consistent, and very powerful.

Python 3: The Bad

The obvious downside to Python 3 is that it is not backwards compatible with prior versions. Even if you are aware of basic incompatibilities such as the print statement, this is only the tip of the iceberg. As a general rule, it is not possible to write any kind of significant program that simultaneously works in Python 2 and Python 3 without limiting your use of various language features and using a rather contorted programming style.

In addition to this, there are no magic directives, special library imports, environment variables, or command-line switches that will make Python 3 run older code or allow Python 2 to run Python 3 code. Thus, it's really important to stress that you must treat Python 3 as a completely different language, not as the next step in a gradual line of upgrades from previous versions.

The backwards incompatibility of Python 3 presents a major dilemma for users of third-party packages. Unless a third-party package has been explicitly ported to Python 3, it won't work. Moreover, many of the more significant packages have dependencies on other Python packages themselves. Ironically, it is the large frameworks (the kind of code for which Python 3 is best suited) that face the most daunting task of upgrading. As of this writing, some of the more popular frameworks aren't even compatible with Python 2.5 – a release that has been out for several years. Needless to say, they're not going to work with Python 3.

Python 3 includes a tool called `2to3` that aims to assist in Python 2 to Python 3 code migration. However, it is not a silver bullet nor is it a tool that one should use lightly. In a nutshell, `2to3` will identify places

in your program that might need to be fixed. For example, if you run it on a "hello world" program, you'll get this output:

```
bash-3.2$ 2to3 hello.py
RefactoringTool: Skipping implicit fixer: buffer
RefactoringTool: Skipping implicit fixer: idioms
RefactoringTool: Skipping implicit fixer: ws_comma
--- hello.py (original)
+++ hello.py (refactored)
@@ -1,1 +1,1 @@
-print "hello world"
+print("hello world")
RefactoringTool: Files that need to be modified:
RefactoringTool: hello.py
bash-3.2$
```

By default, `2to3` only identifies code that needs to be fixed. As an option, it can also rewrite your source code. However, that must be approached with some caution. `2to3` might try to fix things that don't actually need to be fixed, it might break code that used to work, and it might fix things in a way that you don't like. Before using `2to3`, it is highly advisable to first create a thorough set of unit tests so that you can verify that your program still works after it has been patched.

Python 3: The Ugly

By far the ugliest part of Python 3 is its revised handling of Unicode and the new I/O stack. Let's talk about the I/O stack first. In adopting a layered approach to I/O, the entire I/O system has been reimplemented from the ground up. Unfortunately, the resulting performance is so bad as to render Python 3 unusable for I/O-intensive applications. For example, consider this simple I/O loop that reads a text file line by line:

```
for line in open("somefile.txt"):
  pass
```

This is a common programming pattern that Python 3 executes more than 40 times slower than Python 2.6! You might think that this overhead is due to Unicode decoding, but you would be wrong – if you open the file in binary mode, the performance is even worse! Numerous other problems plague the I/O stack, including excessive buffer copying and other resource utilization problems. To say that the new I/O stack is "not ready for prime time" is an understatement.

To be fair, the major issue with the I/O stack is that it is still a prototype. Large parts of it are written in Python itself, so it's no surprise that it's slow. As of this writing, there is an effort to rewrite major parts of it in C , which can only help its performance. However, it is unlikely that this effort will ever match the performance of buffered I/O from the C standard library (the basis of I/O in previous Python versions). Of course, I would love to be proven wrong.

The other problematic feature of Python 3 is its revised handling of Unicode. I say this at some risk of committing blasphemy – the fact that Python 3 treats all text strings as Unicode is billed as one of its most important features. However, this change now means that Unicode pervades every part of the interpreter and its standard libraries. This includes such mundane things as command-line options, environment variables, filenames, and low-level system calls. It also means that the intrinsic complexity of Unicode handling is now forced on *all* users regardless of whether or not they actually need to use it.

To be sure, Unicode is a critically important aspect of modern applications. However, by implicitly treating all text as Unicode, Python 3 introduces a whole new class of unusual programming issues not seen in previous Python versions. Most of these problems stem from what remains a fairly loose notion of any sort of standardized Unicode encoding in many systems. Thus, there is now a potential mismatch between low-level system interfaces and the Python interpreter.

To give an example of the minefield awaiting Python 3 users, let's look at a simple example involving the file system on a Linux system. Take a look at this directory listing:

```
% ls -b
image.png jalape\361o.txt readme.txt
%
```

In this directory, there is a filename `jalepe\361o.txt` with an extended Latin character, "ñ", embedded in it. Admittedly, that's not the most common kind of filename one encounters on a day-to-day basis, but Linux allowed such a filename to be created, so it must be assumed to be technically valid. Past versions of Python have no trouble dealing with such files, but let's take a look at what happens in Python 3. First, you will notice that there is no apparent way to open the file:

```
>>> f = open("jalape\361o.txt")
Traceback (most recent call last):
IOError: [Errno 2] No such file or directory: 'jalapeño.txt'
>>>
```

Not only that, the file doesn't show up in directory listings or with file globbing operations – so now we have a file that's invisible! Let's hope that this program isn't doing anything critical such as making a backup.

```
>>> os.listdir(".")
['image.png', 'readme.txt']
>>> glob.glob("*.txt")
['readme.txt']
>>>
```

Let's try passing the filename into a Python 3.0 program as a command-line argument:

```
% python3.0 *.txt
Could not convert argument 2 to string
%
```

Here the interpreter won't run at all – end of story.

The source of these problems is the fact that the filename is not properly encoded as UTF-8 (the usual default assumed by Python). Since the name can't be decoded, the interpreter either silently rejects it or refuses to run at all. There are some settings that can be made to change the encoding rules for certain parts of the interpreter. For example, you can use `sys.setfilesystemencoding()` to change the default encoding used for names on the file system. However, this can only be used after a program starts and does not solve the problem of passing command-line options.

Python 3 doesn't abandon byte-strings entirely.

Reading data with binary file modes [for example `open (filename,"rb")`] produces data as byte-strings. There is also a special syntax for writing out byte-string literals. For example:

```
s = b'Hello World'       # String of 8-bit characters
```

It's subtle, but supplying byte-strings is one workaround for dealing with funny file names in our example. For example:

```
>>> f = open(b'jalape\361o.txt')
>>> os.listdir(b'.')
[b'jalape\xf1o', b'image.png', b'readme.txt']
>>>
```

Unlike past versions of Python, byte-strings and Unicode strings are strictly separated from each other. Attempts to mix them in any way now produce an exception:

```
>>> s = b'Hello'
>>> t = "World"
>>> s+t
Traceback (most recent call last):
  File "", line 1, in
TypeError: can't concat bytes to str
>>>
```

This behaviour addresses a problematic aspect of Python 2 where Unicode strings and byte-strings could just be mixed together by implicitly promoting the byte-string to Unicode (something that sometimes led to all sorts of bizarre programming errors and I/O issues). It should be noted that the fact that string types can't be mixed is one of the most likely things to break programs migrated from Python 2. Sadly, it's also one feature that the `2to3` tool can't detect or correct. (Now would be a good time to start writing unit tests.)

System programmers might be inclined to use byte-strings in order to avoid any perceived overhead associated with Unicode text. However, if you try to do this, you will find that these new byte-strings do not work at all like byte-strings in prior Python versions. For example, the indexing operator now returns byte values as integers:

```
>>> s[2]
108
>>>
```

If you print a byte-string, the output always includes quotes and the leading `b` prefix – rendering the `print()` statement utterly useless for output except for debugging. For example:

```
>>> print(s)
b'Hello'
>>>
```

If you write a byte-string to any of the standard I/O streams, it fails:

```
>>> sys.stdout.write(s)
Traceback (most recent call last):
  File "", line 1, in
  File "/tmp/lib/python3.0/io.py", line 1484, in write
    s.__class__.__name__)
TypeError: can't write bytes to text stream
>>>
```

You also can't perform any kind of meaningful formatting with byte-strings:

```
>>> b'Your age is %d' % 42
Traceback (most recent call last):
  File "", line 1, in
TypeError: unsupported operand type(s) for %: 'bytes' and 'int'
>>>
```

Common sorts of string operations on bytes often produce very cryptic error messages:

```
>>> 'Hell' in s
Traceback (most recent call last):
  File "", line 1, in
TypeError: Type str doesn't support the buffer API
>>>
```

This does work if you remember that you're working with byte-strings:

```
>>> b'Hell' in s
True
>>>
```

The bottom line is that Unicode is something that you will be forced to embrace in Python 3 migration. Although Python 3 corrects a variety of problematic aspects of Unicode from past versions, it introduces an entirely new set of problems to worry about, especially if you are writing programs that need to work with byte-oriented data. It may just be the case that there is no good way to entirely handle the complexity of Unicode – you'll just have to choose a Python version based on the nature of the problems you're willing to live with.

Conclusions

At this point, Python 3 can really only be considered to be an initial prototype. It would be a mistake to

start using it as a production-ready replacement for Python 2 or to install it as an "upgrade" to a Python 2 installation (especially since no Python 2 code is likely to work with it).

The embrace of Python 3 is also by no means a foregone conclusion in the Python community. To be sure, there are some very interesting features of Python 3, but Python 2 is already quite capable, providing every feature of Python 3 except for some of its advanced features related to metaprogramming. It is highly debatable whether programmers are going to upgrade based solely on features such as Unicode handling – something that Python already supports quite well despite some regrettable design warts.

The lack of third-party modules for Python 3 also presents a major challenge. Most users will probably view Python 3 as nothing more than a curiosity until the most popular modules and frameworks make the migration. Of course this raises the question of whether or not the developers of these frameworks see a Python 3 migration as a worthwhile effort.

It will be interesting to see what programmers do with some of the more advanced aspects of Python 3. Many of the metaprogramming features such as annotations present interesting new opportunities. However, I have to admit that I sometimes wonder whether these features have made Python 3 too clever for its own good. Only time will tell.

General Advice

For now, the best advice is to simply sit back and watch what happens with Python 3 – at the very least it will be an interesting case study in software engineering. New versions of Python 2 continue to be released and there are no immediate plans to abandon support for that branch. Should you decide to install Python 3, it can be done side-by-side with an existing Python 2 installation. Unless you instruct the installation process explicitly, Python 3 will not be installed as the default.

References

[1] What's New in Python 3?:
`http://docs.python.org/dev/3.0/whatsnew/3.0.html`

Originally published in ***;login: The USENIX Magazine,*** *vol. 34, no. 2 (Berkeley, CA: USENIX Association, 2009). Reprinted by kind permission.*

Using Drupal

Angela Byron, Addison Berry, Nathan Haug, Jeff Eaton, James Walker and Jeff Robbins

O'Reilly Media
ISBN: 978-0-596-51580-5
492pp.
£ 34.50
Published: December 2008

reviewed by Gavin Inglis

Content Management Systems are close to the hearts of many web publishers. They offer easy, intuitive site management and an end to all those pesky HTML tags. However, often publishers switch too quickly based on a recommendation or review, without considering in enough depth how that particular software will suit their needs. Indeed there is rather too much choice in the CMS market.

Drupal has a good reputation among CMS users for its longevity, extensive range of modules and suitability for collaborative work. *Using Drupal* explores its application to a variety of realistic web projects.

Rather than plunge straight in, an early chapter takes the time to break down Drupal and differentiate it from the web server plumbing beneath, and the style coding above. This is certainly worth the read to become clear on the concepts of modules, blocks, nodes and so on.

Case studies always help to ground a technical volume, and *Using Drupal* deploys no less than nine of them to cover its ground; in fact, they represent the conceptual spine of the book. The idea – and it comes off well – is to show the breadth of sites which can be created with Drupal, while delving deeper into its functionality at each stage. It also provides a context to each design decision: what modules are required, how the site should be laid out and styled, and so on.

First we have Jeanne and Mike's organic grocery store in the midwest US. Their objective is to move from a large static page edited by their next door neighbour to a dynamic site they can edit themselves. As beginners, a lot of the choices by their technical adviser (the next door neighbour) concern simplicity and functionality.

A job posting board for a University introduces us to more refined form handling, the Views module and a more traditional database-driven application. Super Duper Chefs, a specialist cookery equipment review site run by a perfectly awful sounding couple, covers interaction with Amazon, searching and advanced CSS rendering.

A wiki for student groups requires revision tracking; an online arts magazine with growing pains needs an editorial structure and a formal workflow; a son sets up a family photo gallery website with hassle-free uploads and image manipulation – in exchange, no doubt, for some extended World of Warcraft time.

Internationalisation and locality come into focus with a website reporting on migratory birds. Event management and attendance tracking are important to a book club more social than many. Finally, we see the creation of a full-blown online T-shirt store using the integrated e-commerce package Ubercart. This covers everything you would expect: shopping cart, order processing and product management.

Also welcome is the banishing of the "how to install" material to Appendix A. The online instructions are quite adequate and in many cases Drupal comes pre-installed with commercial webhosting.

This book is aimed at the everyday user rather than the system administrator. Its treatment of security gets little further than downloading new versions when the status page requires it. There is also little space given to the option of extracting the content should one want to change CMS later.

Nevertheless *Using Drupal* is a well judged work written with clarity and a hands-on approach, and is a good investment for anybody already leaning towards this particular CMS.

Android Application Development

Rick Rogers, John Lombardo, Zigurd Mednieks and Blake Meike

O'Reilly Media
ISBN: 978-0-596-52147-9
334pp.
£ 30.99
Published: May 2009

reviewed by Mike Smith

I was particularly interested in having a look at this book as I've been getting more and more interested in mobile app development – in my case for the iPhone using Apple's Xcode under Mac OS X. It's ok, OS X = Unix, so I'm not feeling guilty. As I write this review Apple has just removed Google Voice apps from their App Store, and the FCC writing letters to both Apple and Google, as well as AT&T. It will be interesting to see how this plays out, and whether this type of behaviour will push developers, their apps, and ultimately users away from the iPhone and towards Android. Or whether it's just a storm in a teacup.

Whilst I'm on the subject, and before I get into the review itself, are you aware of the iPhone lectures from Stanford in iTunes U? They're really good, and it's a welcome change to just using books and research materials on the web. I really hope we see more of this.

So, Android Application Development. At about 300 pages it seems quite a pricey book, coming in at £31, but that might be due to exchange rates these days (or maybe I'm just out of touch). It is targeted

at the beginner (surely almost every is a beginner with Android at the moment) and provides a very brief overview of how the Android environment and applications function before going into the code itself with some sample applications.

I worked through chapter one; installing Eclipse, the Android SDK and the Android Developer Tool (ADT) which is a plugin for Eclipse. The first thing to note is that this book was written at the end of 2008 when SDK 1.1_r1 was out. Google have now released 1.5_r3. There are some major changes.

I wrote my first HelloWorld Android application, following Chapter 2. It didn't work for me – I got some Android Davlik build errors which needed to be sorted out by refreshing the project by hitting F5 in Eclipse. I don't know why, but it worked after that; launching the Android emulator and eventually (it seemed to take an age) installed and ran the application.

The next couple of chapters work through a sample application which integrates mapping, web access and phone functions. The source code is downloadable from the O'Reilly website. Quite confusingly there are various packages there – I suspect this is because of code corrections so it's worth grabbing the latest `.tgz` (February 2009). I've always found the best way to learn is to type code in and make the mistakes. I didn't do it this time partly for expediency (so I could meet the copy date for this review), but also because only code fragments are given in the book – not the whole code. Whilst Chapter 3 asserts quite grandly that MicroJobs (the name of this app) is the main sample application for the book, we never seem to return to it after Chapter 4, and that only talks briefly about resources and what happens when the application launches. The later chapters all seem to have their own examples (perhaps incorporated into the main application, but if so I didn't get it.)

I noted a trivial typo on page 47 and have submitted an errata entry on the O'Reilly website, but when I got there I noticed quite a few others, and questions regarding the new SDK and creating Android Virtual Devices which seems to have confused people. This is a new requirement in 1.5 of the SDK, and I had stumbled through the process in the eclipse GUI myself when building HelloWorld wondering why the book hadn't talked about AVDs.

Chapter 5 covers debugging and adb. NO, it's not the general purpose debugger, or even the Absolute Debugger on HP-UX – those rotters have stolen the acronym (and command) for the Android Debug Bridge!

The next chapter talks about a sample application that comes with the Android SDK, ApiDemos. This apparently is a "treasure trove of code". The purpose of this chapter is to explain what is going on with the ApiDemo and how to find the code. It turns out that it's only a few pages long, and I didn't see much value in it. Finally in Part I of the book, we cover how to sign and publish applications. In contrast to the previous chapters this is quite detailed and possibly useful, though I haven't been through the process myself so can't vouch for its accuracy. At the time the book was written the Android Market was only in beta and only accepting submissions of free applications, but one can now impose a charge – though without an actual Android phone it's still not possible to see the full market, which seems to be a flaw to me.

So Part I was a collection of miscellaneous activities – installing the tools, starting to code, debugging and publishing. Part II of the book covers a random collection of programming topics. There are chapters for SQLite and content providers (I don't know why these are lumped together); location and mapping; building views (only now do we start talking about Model View Controller frameworks); Widgets; 2D and 3D graphics; Inter Process Communication and Telephony.

There's an appendix on wireless protocols (2G, 2.5G, 3G) tagged on the end. Goodness knows why.

So to summarise: the biggest problem with this book is that the SDK has already changed, and for someone just learning about Android (and eclipse and Java to a certain extent) that's an issue. Of course it's a fast moving area at the moment and things will continue to evolve.

It seems to me that the book has been written too early and rushed out, and there isn't enough detail to get people through the potential stumbling blocks (especially those who are learning). The chapters are too short and seemingly incomplete. The material is mixed up – why do we talk about publishing in the first half, and only then go back to developing code? It therefore doesn't seem to flow very well. I think there are too many assumptions to be useful for the beginner, but it's not for the experienced coder either. I can't recommend it, and actually it's rather put me off trying to develop Android apps too. Back to the iPhone I think.

The Art of Concurrency

Clay Breshears

O'Reilly Media
ISBN: 978-0-596-52153-0
302pp.
£ 34.50
Published: May 2009

reviewed by Mike Smith

It's been years since I studied concurrent programming as an undergrad. The last book I read was "Principles of Concurrent Programming" by Ben-Ari back in the 80s. That was a fairly rudimentary book, introducing the reader to semaphores and programming techniques ultimately to address the classic Dining Philosophers problem. Concurrency has become so important these days because of multi-threaded programming and multi-core processors; I was interested to know with "The Art of Concurrency" how thinking has moved on since that time.

Also, when a book's title begins "The Art of . . . ", I immediately think of Knuth's seminal works. Now that raises the bar, very high indeed, and I'm hoping its not going to be a disappointment.

The book starts off by outlining the differences between Parallelism and Concurrency. It's a distinction that, to be honest, I hadn't previously paused to consider. Breshears also used the word "Seminal" in chapter one – though I used it first as I wrote the introduction to this review before getting to that point. So it seems like we're going to get along just fine. He even slips in a Gordon Ramsey quote (without an 'F' surprisingly!)

This is not a reference book – it's quite a nice book to just sit down and read. Interesting pictures too. I assume the significance of the five combine harvesters on the cover is that they do the job five times quicker in parallel. The writing style is good, and when there's heavy or "boring" subject matter ahead we get a warning, and an indication of how important it is to read or whether you can get away with skipping it.

I was very pleased to read in Chapter 3 that Breshears actually refers back to the very book I mentioned in my introduction. Apparently the second edition was published much more recently, in 2006, and its title is now "Principles of Concurrent and Distributed Programming". He recommends it, and in this chapter goes on to repeat some of the same techniques to understand and address the Critical Section problem. . . culminating in Dekker's algorithm. He then goes on to discussing performance, and in Chapter 4 lays our eight important rules for designing multithreaded applications.

After the next chapter, which provides a brief overview of threading libraries, we're almost half way through the book in terms of chapters but only around a third of the way through in terms of material. Save for the very last brief summary chapter, we're now into algorithms and code. We have five major chapters covering Parallel Sums, MapReduce, Sorting, Searching and Graphs.

In each of these chapters there are plenty of diagrams and code fragments covering the topics. If anything I would have like to see more discussion on use cases, but I think there is sufficient material explaining how one moves from serial algorithms to parallelisation – which is the main goal.

Overall I found the book interesting and informative, and a pleasure to read. I think this is a positive vote for both this and the Ben-Ari texts.

Version Control with Git

Jon Loelinger
O'Reilly Media
ISBN: 978-0-596-52012-0
328pp.
£ 26.99
Published: June 2009

reviewed by James Youngman

O'Reilly has a history of publishing books about free software. On the face of it this is surprising. Often documentation is available on the web and the source code is available for you to peruse if you have detailed questions not covered in the provided documentation. Yet O'Reilly has been successful.

This book is a good example of why this has worked well for O'Reilly. Git is popular, growing, has documentation on the web (and included with the software) and has active IRC channels devoted to support. This book succeeds in providing something that isn't really available on the web, and is representative of O'Reilly books in this sense.

To use any complex system (software or something else) effectively, the user needs to have a conceptual model of how it works. This model is useful for doing things like visualising the effect of a command before it is run. This is especially vital for version control systems like CVS, in which your commit or tag operation is immediately visible and sometimes very hard to fully reverse if it was a mistake. This sort of thing is less vital in a decentralised version system, since changing the repository and sharing your changes are totally separate steps. However, having this sort of conceptual model is still very helpful.

I've used git for a couple of years, and I found that an early inability to visualise the effect of a command was a big obstacle to confidence; I was still used to a CVS world, where a mistake can be bad news. As I've progressed, my ideas about how git worked became more concrete and so these days I find git quite natural.

However, on reading the book, I certainly found that the conceptual material was very valuable. Some of it confirmed my understanding, and some of it extended my existing ideas about git.

The first half of the book deals with the fundamental concepts involved in using Git; commits, branches, objects, trees, diffs and merging. While I was familiar with all of these things, I certainly found this part of the book valuable.

The second half of the book relates to repository management, patching, hooks and submodules. This is the kind of material I was looking forward to getting to grips with when I read the book. However, some of this material (hooks and submodules in particular) is still subject to change and improvement. This has meant that I didn't get the in-depth treatment of topics like merge drivers that I had hoped for.

That being said I'd not hesitate to recommend the book to any prospective or current user of Git (though if you're a Git hacker you may not need it).

Things I'd be looking forward to in a second revision would include a worked example of building a merge driver, a treatment of however the submodules support in git works (when it settles down), some worked examples with hooks, and perhaps more guidance on operating public repositories.

Automating System Administration with Perl

David Blank-Edelman
O'Reilly Media
ISBN: 978-0-596-00639-6
666pp.
£ 30.99
Published: May 2009

reviewed by Paul Waring

As the owner of the first edition of this text, I've been waiting for several years for an updated text to cover everything which has changed in Perl during that time. At double the length of the previous edition, and with complete rewrites of some chapters (particularly spam filtering, which has necessarily come on a long way), this book does not disappoint.

The book begins with some basics, such as installing modules and getting started on the three major platforms which Perl supports (namely Unix/Linux, OS X and Windows). Some important ground rules are also covered, such as taking care when reading data – particularly if supplied by those notorious and untrusted "users" – and how to drop privileges as soon as possible. Most of this will be familiar territory to experienced Perl programmers, but it serves as a good grounding for sysadmins who are just starting to pick up the language.

The main section of this book is where the meaty topics reside. Manipulating filesystems, working with configuration files, network monitoring and email handling – most of the common system tasks which you could ever want to automate are covered in depth in dedicated chapters. Particularly pleasing is the chapter dedicated to security – something which every programming book should at least touch upon but so few manage to do so. All the code examples are written with `use strict`, so you can rest assured that there are no subtle bugs relating to mistyped or undefined variables lurking around.

The final part of the book contains several appendices covering topics such as XML, SQL and LDAP, which are highly useful short introductions to these areas. If you've ever wanted to know how a particular technology works and how to get it running with Perl in 10-15 minutes, these tutorials will give you the initial foothold to get started. You can even learn how to convert VBScripts to Perl, should you be in the unfortunate position of needing to do so.

Overall, I can only say that if you want to automate system tasks and Perl is your language of choice (or the one your boss has thrust upon you) you should buy this book, even if you have a copy of the first edition. This is the sort of book which has a permanent place reserved for it on my desk, as opposed to lying on a shelf gathering dust, and it already has several bookmarks scattered throughout it. Now, if only there was an Automating System Administration with *Python* title available...

[See Corey Brune's article in this newsletter, and the book "Python for Unix and Linux System Administration" by Jeremy Jones and Noah Gift reviewed in the last issue (Ed).]

Beautiful Architecture

Diomidis Spinellis and Georgios Gousios

O'Reilly Media
ISBN: 978-0-596-51798-4
426pp.
£ 34.50
Published: January 2009

reviewed by Paul Waring

Another release in the O'Reilly "Beautiful" series, this text consists of a number of articles by different authors which claim to "reveal the hidden beauty in software design". Starting off with the question of "what is architecture?", the articles move from the basics of architecture design, through some detailed explanations of specific examples before closing off with a round up of classic architecture designs.

Out of all the articles, I found the chapter on Xen particularly interesting – virtualisation is still a hot topic, but many articles focus on the performance details rather than the fundamental architecture choices. The two chapters on Web architectures, including the Facebook Platform, are well worth a read if you work with Web applications – the direction in which software seems to be moving. I was also pleased to see the acknowledgement by the authors that a single chapter cannot hope to cover any one architecture in depth, so several suggestions of further reading are made for the reader who wishes to explore further.

One point where this book does excel, in comparison to the other Beautiful texts I've read, is having a consistent theme throughout. Even though the articles are written by different authors, there is a clear strand throughout the entire book, and progression from chapter to chapter. As a result, you can read the whole text from start to finish and feel as if you've had the complete story, as opposed to a loose collection of articles. Despite this, I still found myself skipping some chapters, as my idea of software architecture is more of a gut feeling than any hard and fast rules (the problem with any rules being that you always need to bend or break them at some point). Furthermore, whilst the ideas presented in the book are interesting, the unfortunate reality is that most software engineers, developers and architects will be maintaining someone else's code as opposed to designing a beautiful architecture from scratch. However, if you are fortunate enough to be starting up a new project with the freedom to design the architecture, this book is worth a read and you may pick up a few ideas for your software.

Cloud Application Architectures

George Reese

O'Reilly Media
ISBN: 978-0-596-15636-7
204pp.
£ 22.99
Published: April 2009

reviewed by Raza Rizvi

An oddly readable book.

Why odd? Well it is part conceptual reference but is at the same time clearly based on the author's practical experience of his own adoption of cloud services. Well not cloud services, more *a* cloud service because it is firmly based on Amazon's EC2 and S3, but not so much that it can't speak to the wider management and take-up of other services (and some balance is provided towards the end of the book). It speaks to the reader who needs a technical background and is probably responsible for the strategic direction that led to the decision to go to the cloud, but includes more than a good handful of administrative guidance with a helpful scattering of programmatic snippets. So a jolly good mixed up 'storyline' but engaging nonetheless.

Obviously chapter 1 of course introduces the reader to what cloud computing means and how it compares to traditional server deployments. There is a good explanation of the Amazon EC2 (Elastic Cloud

Compute) and S3 (Simple Storage Service) which is expanded as we move into chapter 2 covering the mechanics of how the service is made up and addressed. There are clear warnings where one feels the author was bitten badly and I couldn't help but feel that the clear way he outlines these sets the practical and down-to-earth nature for the rest of the book. It isn't meant to be a reference book for the Amazon services but a summary of the practical experience gained in the author's deployment over the last few years.

Chapter 3 firmly sets out how you might calculate your costs for the take up of cloud services not just in financial terms but in reliability and availability against a more standard deployment. It is sensible to see it not just as a means of saving money at any cost. Assuming you are convinced it is the right thing to do for your project, chapters 4, 5, and 6 deal with the topics likely to be foremost in your mind – data management, security, and data recovery/backup. You are trusting someone else with your data whilst still trying to maintain your obligations to your regulators, customers, and stakeholders and the author (with both US and EU examples) knows what the overall issues are that you will want to question. He makes it clear that in many instances the problems you will face as the same as running applications in a hosted data centre (how do you get all that data from over there to over here…?) but he also explains what is so different in trying to secure your data in a cloud where you have no discrete physical access points to protect. I thought his approach and clarity was particularly good once you understand the additional issues.

The meat of the book rounds off with some good advice on scaling your deployment. If you thought that you didn't have to worry about this just because you had shifted everything into the cloud, the 15 pages, or so, explain what you still have to be concerned about.

The book rounds off with an EC2 command reference (it isn't clear why the S3 commands were not similarly gathered for ease of reference or at least a pointer to Amazon's own documentation provided), and then two small descriptions of alternate cloud architectures using GoGrid and Rackspace (from those vendors).

At times I thought the page layout a little scattered with text, code fragments, in-copy explanations, footnotes, diagrams, and command-line snippets. The process of editing should have made this less taxing on the eye and should have picked up errors like the two missing figures (but they are on the errata page online).

Overall then, and notwithstanding my quibble about the text layout at times, this is a useful book for those technically minded decision makers who want more than just marketing material to help them focus their thought on how (and why) they should deploy services outside of the traditional physical data centre. Because it was written from practical experience, I felt that the author was truthfully balanced rather than just evangelistic.

Contributors

David Beazley is an open source software developer and the author of Python Essential Reference (4th edition, Addison-Wesley, 2009). He lives in Chicago, where he also teaches Python courses.

Corey Brune is currently a Master Consultant in Plano, Texas. He has been a UNIX Engineer for over 13 years, specializing in s ystems development, networking, architecture, and performance and tuning. His latest project is the development of parsers for performance tuning and capacity planning.

Gavin Inglis works in Technical Infrastructure at the EDINA National Data Centre in Edinburgh. He is a teacher, photographer and musician who has recently discovered the joys of spreadsheets as a recreational tool.

Jane Morrison is Company Secretary and Administrator for UKUUG, and manages the UKUUG office at the Manor House in Buntingford. She has been involved with UKUUG administration since 1987. In addition to UKUUG, Jane is Company Secretary for a trade association (Fibreoptic Industry Association) that she also runs from the Manor House office.

Raza Rizvi is Technical Director of Activereach Ltd.

Mike Smith works in the Chief Technology Office of a major European listed outsourcing company, setting technical strategy and working with hardware and software vendors to bring innovative solutions to its clients. He has over 15 years in the industry, including mid-range technical support roles and has experience with AIX, Dynix/ptx, HP-UX, Irix, Reliant UNIX, Solaris and of course Linux.

Paul Waring is chairman of UKUUG and currently and the Technical Manager for an insurance intermediary. He is also responsible for organising the UKUUG Spring conference in 2010.

Roger Whittaker works for Novell Technical Services at Bracknell supporting major Linux accounts in the UK. He is also the UKUUG Newsletter Editor, and co-author of three successive versions of a SUSE book published by Wiley.

James Youngman spends his days downloading web pages and his nights emptying `/dev/null` as a member of the UKUUG's Waste Disposal directorate. Programs written by James that you might have used include "find", "xargs" and "epicycle". His hypothetical interests outside computing include rock climbing and photography.

Contacts

Paul Waring
UKUUG Chairman
Manchester

John M Collins
Council member
Welwyn Garden City

Phil Hands
Council member
London

Holger Kraus
Council member
Leicester

Niall Mansfield
Council member
Cambridge

John Pinner
Council member
Sutton Coldfield

Howard Thomson
Treasurer; Council member
Ashford, Middlesex

Jane Morrison
UKUUG Secretariat
PO Box 37
Buntingford
Herts
SG9 9UQ
Tel: 01763 273475
Fax: 01763 273255
office@ukuug.org

Roger Whittaker
Newsletter Editor
London

Alain Williams
UKUUG System Administrator
Watford

Sam Smith
Events and Website
Manchester